

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет комп'ютерних наук і технологій

(повне найменування факультету)

Кафедра програмних засобів

(повне найменування кафедри)

Пояснювальна записка

до дипломного проєкту (роботи)

магістр

(ступінь вищої освіти)

на тему ДОСЛІДЖЕННЯ ТА ПРОГРАМНА РЕАЛІЗАЦІЯ
МЕТОДІВ ВДОСКОНАЛЕННЯ ВИХІДНОГО КОДУ
RESEARCH AND SOFTWARE IMPLEMENTATION OF
SOURCE CODE IMPROVEMENT METHODS

Виконав(ла): студент(ка) 2 курсу, групи КНТ-124м

Спеціальності 121 Інженерія програмного

(код і найменування спеціальності)

забезпечення

Освітня програма (спеціалізація)

Інженерія програмного забезпечення

ШАМАНСЬКИЙ Р.В.

(ПРИЗВИЩЕ та ініціали)

Керівник ЗАЙКО Т.А.

(ПРИЗВИЩЕ та ініціали)

Рецензент ІЛЛЯШЕНКО М.Б.

(ПРИЗВИЩЕ та ініціали)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет КНТ
Кафедра програмних засобів
Ступінь вищої освіти магістр
Спеціальність 121 Інженерія програмного забезпечення
(код і найменування)
Освітня програма (спеціалізація) Інженерія програмного забезпечення
(назва освітньої програми (спеціалізації))

ЗАТВЕРДЖУЮ

Завідувач кафедри ПЗ, д.т.н, проф.
Сергій СУББОТІН
“ ” 2025_року

З А В Д А Н Н Я

НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА(КИ)

ШАМАНСЬКОГО Романа Володимировича

(ПРИЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Дослідження та програмна реалізація методів вдосконалення вихідного коду. Research and Software Implementation of Source Code Improvement Methods

керівник проєкту (роботи) к.т.н., доцент, ЗАЙКО Тетяна Анатоліївна,
(науковий ступінь, вчене звання, ПРИЗВИЩЕ, ім'я, по батькові)

затверджені наказом закладу вищої освіти від “ 30 ” вересня 2025 року № 447

2. Строк подання студентом проєкту (роботи) 02 грудня 2025 року

3. Вихідні дані до проєкту (роботи) рекомендована література, технічне завдання

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1. Аналіз предметної області. 2. Матеріали і методи. 3. Опис програми. 4. Експлуатація, тестування та експериментальне дослідження програми.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, кількість слайдів, плакатів) Слайди презентації

6. Консультанти розділів проєкту (роботи)

Розділ	ПРИЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-4 Основна частина	ЗАЙКО Т.А., доцент		
Нормоконтроль	ДЕЙНЕГА Л.Ю., ст. викладач		

7. Дата видачі завдання “ 30 ” вересня 2025 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Постановка завдання роботи.	1 тиждень	Завдання, ТЗ
2	Аналіз предметної області.	1 тиждень	Розділ 1
3	Вибір мови програмування та інших технологій розробки.	2 тиждень	Розділ 2
4	Розробка структури програми.	3 тиждень	Розділ 3
5	Розробка програми.	4-7 тижні	Розділи 3,4
6	Тестування та експериментальне дослідження програмного забезпечення.	8 тиждень	Розділ 4
7	Оформлення пояснювальної записки та документів до неї.	9-10 тижні	Додатки
8	Нормоконтроль та рецензування.	11 тиждень	
9	Захист роботи.	12 тиждень	

Студент(ка)

_____ Роман ШАМАНСЬКИЙ
(підпис) (Імя ПРИЗВИЩЕ)

Керівник проєкту (роботи)

_____ Тетяна ЗАЙКО
(підпис) (Імя ПРИЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи магістра:
89 с., 3 табл., 28 рис., 3 дод., 18 джерел.

JAVASCRIPT, REACT, ЗАСТОСУНОК, ВДОСКОНАЛЕННЯ КОДУ,
МОВА ПРОГРАМУВАННЯ, ПРОГРАМНИЙ КОД.

Об'єкт дослідження – процес вдосконалення вихідного програмного коду.

Предмет дослідження – програмні засоби для вдосконалення вихідного коду.

Мета роботи – дослідження та розробка програмного забезпечення для вдосконалення вихідного коду.

Матеріали, методи та технічні засоби: мова програмування JavaScript, бібліотека для створення інтерфейсів React.

Результати. Розроблено програмне забезпечення для вдосконалення вихідного коду за допомогою мови програмування JavaScript та бібліотеки для створення інтерфейсів React.

Практична цінність роботи полягає у розробці програмного забезпечення для вдосконалення вихідного коду.

Висновки. Виконано проєктування програмного забезпечення для вдосконалення вихідного коду. Розроблено програмне забезпечення для вдосконалення вихідного коду. Здійснено тестування розробленого програмного забезпечення для вдосконалення вихідного коду.

Галузь використання – вдосконалення програмного коду.

ABSTRACT

Explanatory note to the diploma qualifying work of the master: 89 pages, 3 tables, 28 figures, 3 appendixes, 18 sources.

JAVASCRIPT, REACT, APPLICATION, CODE IMPROVEMENT, PROGRAMMING LANGUAGE, PROGRAM CODE.

The object of research is the process of developing software for improving the source code.

The subject of the research is software for improving the source code.

The purpose of this work is to develop the software improving the source code.

Materials, methods and technical tools: JavaScript programming language, React interface creation library.

Results. The software tools for improving the source code using the JavaScript programming language and the React interface creation library.

The practical value of the work lies in the development of software for improving the source code.

Conclusions. The software for improving the source code has been designed. The software for improving the source code has been developed. The developed software for improving the source code has been tested.

Scope of use – improving the source code.

ЗМІСТ

	С.
Перелік скорочень та умовних позначок	8
Вступ	9
1 Аналіз предметної області	11
1.1 Методи вдосконалення вихідного коду	11
1.2 Аналіз програмного забезпечення для вдосконалення вихідного коду ..	22
1.3 Висновки за розділом 1	30
2 Матеріали і методи	32
2.1 Вибір мови програмування.....	32
2.2 Вибір бібліотеки для створення інтерфейсу програмного забезпечення для вдосконалення вихідного коду	34
2.3 Розробка методу вдосконалення вихідного коду	37
2.4 Висновки за розділом 2	41
3 Опис програми	42
3.1 Структура програмного забезпечення для вдосконалення вихідного коду	42
3.2 Функціонування програмного забезпечення для вдосконалення вихідного коду	44
3.3 Розробка бази даних програмного забезпечення для вдосконалення вихідного коду	46
3.4 Проєктування інтерфейсу програмного забезпечення для вдосконалення вихідного коду	50
3.5 Висновки за розділом 3	52
4 Експлуатація, тестування та експериментальне дослідження програми	54
4.1 Призначення й умови застосування програми	54
4.2 Характеристики програми для вдосконалення вихідного коду	55
4.3 Інструкція по експлуатації програми	55
4.3.1 Звернення до програми.....	55
4.3.2 Вхідні й вихідні дані.....	56

4.3.3 Повідомлення.....	56
4.4 Виконання програмного забезпечення для вдосконалення вихідного коду.....	56
4.5 Тестування програмного забезпечення для вдосконалення вихідного коду.....	59
4.6 Висновки за розділом 4	59
Висновки.....	60
Перелік джерел посилання	64
Додаток А Технічне завдання.....	66
Додаток Б Фрагмент тексту програми	70
Додаток В Слайди презентації	82

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

MVC – Model-View-Controller;

БД – база даних;

ВК – вихідний код;

ПЗ – програмне забезпечення.

ВСТУП

Актуальність методів удосконалення вихідного програмного коду зумовлена постійним зростанням складності програмних систем і підвищеними вимогами до їхньої надійності, продуктивності та масштабованості. У сучасних умовах швидкого розвитку інформаційних технологій програмні продукти повинні забезпечувати високу якість, бути легкими для підтримки й розширення. Наявність неефективних чи застарілих рішень у коді призводить до збільшення витрат на супровід, виникнення технічного боргу та зниження конкурентоспроможності продукту [1]-[3].

Програмна реалізація методів вдосконалення коду набуває особливого значення, оскільки автоматизація цього процесу дозволяє зменшити ймовірність людських помилок, прискорити оптимізацію та забезпечити систематичність змін. Використання відповідних інструментів і алгоритмів сприяє покращенню структурованості програм, підвищенню їх продуктивності, зниженню ресурсних витрат та полегшенню подальшої інтеграції нових функцій. Завдяки цьому забезпечується довготривала ефективність програмного забезпечення та його здатність адаптуватися до нових вимог [4]-[6].

Проте деякі програми для вдосконалення вихідного коду є орієнтованими на певні мови програмування чи конкретні середовища, що зменшує універсальність і робить їх менш ефективними у випадку багатоплатформних рішень. Іншим недоліком є ймовірність появи нових помилок у процесі рефакторингу або оптимізації, особливо якщо інструменти використовуються без належного тестування. Крім того, застосування таких програмних засобів може вимагати значних ресурсів, як обчислювальних, так і часових, що знижує ефективність при роботі з великими проектами. Також важливо враховувати, що деякі програмні засоби вдосконалення вихідного коду не завжди забезпечують гнучкість, оскільки їхні алгоритми ґрунтуються

на загальних правилах і не враховують унікальні вимоги конкретного програмного проєкту [2]-[5].

Тому актуальною є розробка програмного забезпечення для вдосконалення вихідного коду. У дипломній кваліфікаційній роботі магістра розв'язується актуальне завдання розробки та дослідження програмного забезпечення для вдосконалення вихідного коду.

Для досягнення поставленої мети у кваліфікаційній роботі магістра необхідно розв'язати такі задачі:

- виконати аналіз предметної області та програмних засобів для вдосконалення вихідного коду;
- розробити метод вдосконалення вихідного коду;
- здійснити проєктування програмного забезпечення для вдосконалення вихідного коду;
- створити програмне забезпечення для вдосконалення вихідного коду;
- виконати тестування розробленого програмного забезпечення для вдосконалення вихідного коду.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Процес та методи вдосконалення вихідного коду

Вдосконалення вихідного коду розглядається як систематичний процес покращення його якості, структури та ефективності без зміни основної функціональності. Такий підхід дозволяє зробити програмний продукт більш зрозумілим, надійним і стійким до помилок, а також полегшує подальшу підтримку та розвиток. У практиці розробки це є невід'ємною частиною забезпечення життєвого циклу програмного забезпечення, оскільки якість коду визначає його довговічність і масштабованість [1], [2].

Процес вдосконалення вихідного коду включає кілька взаємопов'язаних етапів. Спочатку здійснюється аналіз існуючої реалізації, виявлення недоліків і визначення місць, де код може бути оптимізований. Далі проводиться рефакторинг, що полягає в упорядкуванні структури, спрощенні логіки та усуненні дублювання. Важливим кроком є також перевірка безпеки, яка дає можливість виявити вразливості та знизити ризики потенційних атак. Після внесення змін відбувається тестування для перевірки коректності роботи та відповідності очікуваним результатам. Заключним елементом є документування змін і впровадження їх у систему контролю версій [1], [2].

Методи вдосконалення вихідного коду охоплюють кілька напрямів. Рефакторинг дозволяє оптимізувати внутрішню структуру, роблячи її більш зрозумілою та логічною. Використання автоматизованих інструментів аналізу сприяє виявленню синтаксичних і логічних помилок, а також підвищує безпеку. Дотримання принципів модульності, стандартизації та єдиного стилю написання полегшує колективну роботу та спрощує підтримку. Тестування, що супроводжує вдосконалення, гарантує збереження функціональності та вчасне виявлення помилок. Сукупність таких методів формує основу для створення надійного, оптимізованого та зручного у подальшому розвитку програмного забезпечення [1], [2].

Оптимізація вихідного є одним із найважливіших етапів у створенні програмного забезпечення, оскільки саме від неї залежить швидкість виконання програм, стабільність їх роботи та економне використання ресурсів. Коли код упорядкований і раціонально організований, система працює ефективніше, забезпечуючи кращий досвід для користувачів і зменшуючи навантаження на технічну інфраструктуру. Завдяки цьому підвищується надійність продукту, зменшуються витрати на його підтримку й експлуатацію, а також полегшується подальший розвиток [1], [2].

Під час удосконалення коду формується міцна основа для його довготривалого використання. Якщо програма побудована на складних і непослідовних рішеннях, внесення змін чи розширення функціональності перетворюється на надзвичайно витратний процес. Упорядкованість та оптимізація дозволяють спростити роботу над майбутніми версіями, роблячи процес адаптації швидшим та менш затратним. Таким чином, користь від оптимізації виходить за межі миттєвого підвищення продуктивності та проявляється у стратегічному зростанні проекту [1], [2].

Оптимізований код легше захистити від можливих загроз. Завдяки зменшенню надлишкових частин і використанню ефективних алгоритмів, вразливості стають більш помітними та простішими у виправленні. Такий підхід демонструє професійність команди розробників, формує довіру до проекту та підсилює його репутацію серед користувачів і партнерів [1], [2].

Основні принципи, що лежать в основі якісного коду, спрямовані на спрощення його читання, підвищення зрозумілості та забезпечення гнучкості під час подальших змін. Важливим завданням є збереження логічної структури, уникнення повторів і підтримка охайності. Використання зрозумілої номенклатури, додавання пояснювальних коментарів, модульність та регулярне тестування допомагають створювати програмні рішення, які не лише працюють коректно, а й легко підтримуються [2], [3].

Перед початком вдосконалення коду необхідно чітко визначити цілі та завдання, що допомагають сконцентрувати зусилля на тих ділянках, де вони

принесуть найбільшу користь. Ретельне планування дозволяє уникати хаотичних змін і гарантує, що оптимізаційні рішення будуть відповідати стратегічним потребам проєкту [2], [3].

Покращення ефективності здійснюється через виявлення фрагментів, які споживають найбільше ресурсів, і подальше усунення слабких місць. Використання більш досконалих алгоритмів та структур даних забезпечує відчутне прискорення обчислень, тоді як грамотне управління пам'яттю сприяє стабільності роботи. Значний ефект може дати також застосування паралельних обчислень, що дозволяє розподіляти завдання між декількома ядрами процесора [2], [3].

Розподіл вихідного коду на логічно завершені модулі робить його більш упорядкованим і зручним для підтримки. При цьому особливу увагу слід приділяти усуненню дублювання та забезпеченню можливості повторного використання компонентів. Такий підхід не лише полегшує подальшу роботу з кодом, а й спрощує процес оптимізації на наступних етапах розвитку проєкту [2], [3].

Вдосконалення коду зазвичай проходить у кілька циклів: від попереднього аналізу продуктивності та виявлення проблемних ділянок до внесення змін, перевірки їхнього впливу та повторної оцінки результатів. Завдяки цьому поступово досягається бажаний рівень швидкодії та стабільності [2], [3].

При вдосконаленні коду необхідно враховувати і ризики надмірної оптимізації, коли прагнення зробити систему максимально швидкою призводить до ускладнення коду та зниження його читабельності. У таких випадках користь від оптимізації може нівелюватися зростанням складності підтримки. Тому всі рішення мають бути збалансованими, а якість програмного коду повинна залишатися пріоритетом [2], [3].

Завершальним етапом є аналіз продуктивності, який дозволяє виміряти вплив внесених змін у різних умовах. Регулярний моніторинг і вдосконалення забезпечують довготривалу ефективність і надійність програмного

забезпечення, оскільки вимоги до продуктивності можуть змінюватися разом із розвитком системи [2], [3].

Оптимізація ніколи не завершується остаточно, оскільки з появою нових технологій виникають нові можливості для покращення. Постійна практика, навчання та відкритість до сучасних інструментів дозволяють підтримувати актуальність програмного продукту та забезпечувати його ефективність у довгостроковій перспективі [2], [3].

Оптимізований код дозволяє системі працювати швидше, використовувати менше ресурсів і забезпечувати користувачеві більш комфортну взаємодію з програмою. Для досягнення цього результату доцільно застосовувати різноманітні підходи, що допомагають усувати зайві обчислення, раціонально розподіляти пам'ять та добирати найпридатніші алгоритми й структури даних. Важливим етапом є виявлення так званих вузьких місць у коді, адже саме вони найчастіше сповільнюють виконання. За допомогою профілювання можна визначити ділянки, які потребують першочергової оптимізації, і зосередити зусилля саме там, де це принесе найбільший ефект [3], [4].

Процес підвищення продуктивності коду передбачає постійний перегляд та вдосконалення вже написаних фрагментів, оскільки навіть незначні зміни здатні позитивно позначитися на швидкодії системи. Водночас варто пам'ятати про баланс між ефективністю та зручністю підтримки: надмірне ускладнення структури коду внаслідок оптимізації може ускладнити його подальше супроводження. Тому кожне вдосконалення потребує перевірки й оцінки впливу в конкретному контексті, оскільки метод, що покращує роботу в одних умовах, може виявитися шкідливим у інших [3], [4].

Не менш важливим етапом у життєвому циклі вдосконалення коду є налагодження. Помилки в програмному забезпеченні здатні знижувати його надійність, створювати загрози безпеці та негативно впливати на досвід користувача. Для їхнього усунення застосовуються інструменти статичного аналізу, тести різних рівнів і засоби відстеження виконання. Ефективна робота

з помилками передбачає їх відтворюваність, локалізацію джерела, формування гіпотез щодо причин, перевірку цих припущень та впровадження рішень із подальшою перевіркою результатів [3], [4].

Визначення точного місця виникнення збоїв у великих і складних програмних кодах може бути особливо складним. Тому зрозуміла структура коду, продумана організація логів та коректні повідомлення про помилки значно полегшують роботу розробників. Проактивний підхід до підвищення якості вихідних текстів програм сприяє не лише швидшому виявленню несправностей, а й істотно спрощує весь процес їх усунення [3], [4].

Захист вихідних текстів програмних систем має вирішальне значення для їхньої надійності та стабільності. Будь-які вразливості можуть призвести не лише до збоїв у роботі, але й до втрати конфіденційної інформації, що становить серйозну загрозу як для користувачів, так і для цілісності самої системи. Забезпечення безпеки коду розглядається як один із головних пріоритетів під час створення програмного забезпечення. Усвідомлення потенційних ризиків і застосування профілактичних заходів проти них формує основу для побудови стійких цифрових продуктів [3], [4].

З поширенням кібератак значення питань безпеки зростає ще більше. Використання некоректних підходів до кодування, неправильних налаштувань чи вразливих сторонніх бібліотек створює сприятливі умови для зловмисників. Наслідком таких дій можуть стати витoki інформації, порушення роботи сервісів або втрата довіри користувачів. Саме тому безпека коду набуває не лише технічного, але й стратегічного значення, оскільки від неї залежить стабільність бізнес-процесів і репутація розробника [3], [4].

Безпека розглядається не як разовий етап, а як безперервний процес, що інтегрується в усі фази розробки. Вона передбачає постійне оновлення знань про актуальні загрози, дотримання безпечних практик програмування та застосування спеціалізованих інструментів. Використання засобів для статичного аналізу, систем контролю версій і платформ командної співпраці дає змогу не лише підвищити якість коду, але й створити організоване

середовище, у якому внески різних учасників узгоджуються без конфліктів. Таким чином, безпечний код є результатом системного підходу, що поєднує уважність до деталей, технічну дисципліну й готовність до постійного вдосконалення [3], [4].

Інструменти для аналізу та вдосконалення програмного коду допомагають глибше зрозуміти його сильні й слабкі сторони. Вони виявляють помилки, проблеми продуктивності та потенційні вразливості. Завдяки методам статичного й динамічного аналізу, а також профілюванню продуктивності, забезпечується підвищення надійності й ефективності програмних систем. Використання таких засобів сприяє підвищенню стабільності роботи застосунків і покращує якість кінцевого продукту [3], [4].

Застосування сучасних інструментів контролю якості програмного коду значно впливає на результат. Постійна перевірка коду, контроль за дотриманням стандартів і своєчасне виявлення помилок роблять процес створення програмних систем швидшим і передбачуванішим. У командній роботі ці інструменти відіграють особливу роль, оскільки забезпечують узгодженість і спрощують інтеграцію внесків різних розробників [3], [4].

У практиці програмування часто виникають помилки, що знижують ефективність та безпеку систем. Неправильне оброблення винятків може призвести до нестабільності роботи застосунку, а витоки пам'яті — до зниження продуктивності. Особливу небезпеку становлять уразливості безпеки, такі як SQL-ін'єкції чи XSS, адже вони створюють можливості для атак з боку злоумисників. Не менш серйозними є дублювання коду, невдало підібрані алгоритми чи невідповідні структури даних, які зменшують продуктивність та ускладнюють підтримку проєктів [3], [4].

Уникнення подібних проблем можливе завдяки систематичному використанню перевірки та вдосконаленню коду, регулярному тестуванню й впровадженню стандартів програмування. Приділення уваги оптимізації структур даних, правильному підбору алгоритмів і впровадженню сучасних

методів контролю допомагає створювати програмні системи, що відзначаються високою продуктивністю та надійністю [3], [4].

Процес вдосконалення програмного коду охоплює проектування, написання та тестування, оскільки саме на цих етапах закладається основа продуктивності та стабільності. Підвищення швидкодії, зниження ресурсозатрат та покращення користувацького досвіду стають головними результатами такої діяльності. При цьому важливо зберігати баланс між оптимізацією та читабельністю програмного коду, оскільки надмірне ускладнення алгоритмів може ускладнити майбутню підтримку й модифікацію програмного коду [3], [4].

Методи вдосконалення вихідного коду охоплюють різноманітні підходи. До них належить заміна неефективних алгоритмів на більш продуктивні рішення, оптимізація використання структур даних для раціонального управління пам'яттю, застосування кешування для зниження навантаження на зовнішні ресурси та прискорення доступу до інформації. Важливим напрямом є також паралельна обробка, що дозволяє використовувати потужність багатоядерних процесорів, а також оптимізація запитів до баз даних, яка забезпечує швидше отримання потрібних даних завдяки індексації та перепроєктуванню логіки запитів [3], [4].

Вдосконалення коду не завершується одноразовим підходом, а потребує постійного моніторингу. Регулярний аналіз продуктивності, виявлення вузьких місць та внесення змін допомагають підтримувати програму у стані максимальної ефективності. Паралельно з цим важливим стає впровадження систем контролю версій, які дають змогу організовано відстежувати всі зміни, координувати роботу команди та забезпечувати безперервний розвиток програмного продукту [3], [4].

Додатковим засобом вдосконалення підвищення якості коду є автоматизоване налагодження коду, завдяки якому можна оперативно виявляти помилки, усувати проблемні ділянки та забезпечувати відповідність встановленим стандартам. Успішне застосування таких підходів на практиці

дозволяє розробникам покращувати структуру коду, скорочувати витрати часу на підтримку та зберігати високу якість програмного забезпечення в умовах постійних змін вимог і технологій [4], [5].

Додаткову цінність становить використання систем статичної перевірки коду, які дозволяють заздалегідь помічати приховані вразливості та проблеми з підтримкою, створюючи ґрунтовну картину надійності проєкту. Поєднання таких рішень забезпечує не лише автоматичне виправлення помилок, а й надання пояснень щодо їхнього походження, що значно підвищує розуміння процесу оптимізації [4], [5].

Особлива увага приділяється відповідності стандартам безпечного кодування, адже дотримання відомих практик знижує ризики поширених загроз. Інструменти, здатні відстежувати небезпечні підходи у режимі реального часу, роблять цей процес більш ефективним і контрольованим. Важливим завданням також залишається перевірка залежностей, адже сторонні бібліотеки часто стають джерелом ризиків. Використання спеціалізованих сервісів у поєднанні з інтелектуальними помічниками оптимізує цей процес та допомагає підтримувати актуальний рівень захищеності [4], [5].

У процесі перетворення проміжних представлень програми використовуються спеціальні прийоми, що зменшують навантаження на процесор і пам'ять, забезпечуючи швидший результат. Важливо, щоб оптимізація не спотворювала логіку роботи застосунку, водночас сприяла підвищенню продуктивності й не перетворювала процес компіляції на надто тривалий. Зазвичай вона здійснюється наприкінці розробки, оскільки впроваджувані зміни ускладнюють читання й модифікацію коду [4], [5].

Потреба в оптимізації коду пояснюється прагненням зменшити розмір програм, прискорити виконання та зробити можливим повторне використання коду. Ручне досягнення цих цілей є трудомістким, тому широко застосовуються автоматизовані інструменти, що полегшують завдання. Серед підходів розрізняють машинно-незалежні, які діють із проміжними

представленнями та не враховують особливостей апаратури, і машинно-залежні, що адаптують код під конкретну архітектуру, оптимізуючи використання пам'яті та регістрів [4], [5].

Останні тенденції у сфері оптимізації коду показують зростання значення чітких метрик, які дозволяють оцінювати ефективність не лише за загальними відчуттями, а й за конкретними показниками часової та просторової складності. Значну роль починають відігравати інструменти штучного інтелекту, здатні виявляти проблемні ділянки в режимі реального часу та автоматично усувати дрібні недоліки [4], [5].

Управління пам'яттю також стає вирішальним чинником, адже відсутність контролю за її використанням здатна перетворити добре продуману програму на ресурсоємну систему. Не менш значущим вважається використання багатопотоковості та асинхронних операцій, що відкриває можливість повніше задіяти багатоядерні процесори. Водночас неправильне впровадження паралельності може створювати додаткові ризики, тому цей напрям вимагає обережності та досвіду [4], [5].

Сучасні компілятори здатні самостійно застосовувати різні техніки вдосконалення коду через налаштування та прапорці, що дозволяє отримати приріст швидкодії без змін у вихідному коді. Використовуються підходи, що видаляють непотрібні інструкції, об'єднують виклики функцій без зайвих накладних витрат, переміщують стабільні обчислення поза цикли та спрощують керування потоками, що дозволяє програмам працювати швидше та ефективніше [4], [5].

Створення дійсно оптимізованого коду передбачає впровадження культури оптимізації, а не лише застосування методів до конкретних ділянок коду. Модульна архітектура дозволяє розділити великі програми на керовані блоки, на які можна зосередити увагу під час оптимізації, не втрачаючи контроль над загальною структурою. Регулярні огляди коду допомагають виявляти недоліки, які автоматичні засоби пропускають, а інтеграція тестів продуктивності забезпечує раннє виявлення регресу швидкодії до

впровадження змін у виробниче середовище. Документація та стандарти кодування сприяють зрозумілості рішень і дозволяють новим розробникам підтримувати ефективність без випадкових порушень існуючих оптимізацій [4], [5].

Ефективні стратегії кодування охоплюють не лише технічні аспекти, а й підтримують бізнес-цілі, забезпечуючи чистий і продуктивний код. Алгоритмічне вдосконалення дозволяє значно скоротити час обробки та споживання ресурсів, навіть без зміни обладнання. Рефакторинг застарілого коду усуває надлишкові конструкції, спрощує логіку та оновлює бібліотеки, що підвищує швидкодію без зміни функціональності. Використання сучасних мовних можливостей перетворює складні та схильні до помилок реалізації на швидкі та надійні рішення, а стратегічна мікрооптимізація, така як розгортання циклів чи вбудовування функцій, зменшує витрати на кеш і додає продуктивності [4], [5].

Оптимізація продуктивності в розробці програмного забезпечення розглядається як ключовий аспект на всіх етапах життєвого циклу. Профілювання та моніторинг дозволяють виявляти проблемні місця ще до того, як вони стануть помітними користувачам, а ітеративний підхід із постійним тестуванням, налаштуванням і контролем забезпечує безперервне вдосконалення швидкодії без затримки розробки. Зусилля з оптимізації спрямовуються на реальні потреби користувачів, наприклад, на прискорення завантаження сторінок у вебзастосунках або зменшення часу запуску настільних програм, а також на забезпечення стабільної продуктивності на різних платформах, від старих ноутбуків до потужних робочих станцій. Інтеграція статичного аналізу коду дає змогу виявляти потенційні вузькі місця та підтримувати стандарти продуктивності ще на етапі розробки [5], [6].

Налаштування продуктивності програмного забезпечення починається з точного відстеження використання пам'яті та процесора за допомогою спеціалізованих інструментів, що скорочує час пошуку причин низької швидкодії. Автоматизовані засоби можуть динамічно регулювати пріоритети

процесів, підтримуючи швидку реакцію системи навіть при високому навантаженні. Тестування навантаження у реальних умовах допомагає виявляти проблеми, які залишаються непоміченими у теоретичних сценаріях, а безперервний моніторинг у реальному часі дозволяє оперативно реагувати на збої ще до того, як користувачі відчують незручності [5], [6].

Ведення чіткої документації та організація коду у модулі полегшують сприйняття та тестування, знижують ризик помилок при внесенні змін і створюють природні точки для оптимізації. Автоматизоване тестування допомагає своєчасно виявляти регресії та підтримувати високу якість коду в процесі розвитку проєкту [5], [6].

Хмарні архітектури та безсерверні підходи змінили традиційні стратегії оптимізації продуктивності програмного коду, дозволяючи динамічно масштабувати ресурси відповідно до попиту і звільняючи команди від складного планування потужностей. Це створює умови для зосередження на оптимізації алгоритмів та логіки програми без необхідності вручну керувати інфраструктурою. Інструменти хмарного моніторингу забезпечують видимість продуктивності в розподілених системах, виявляючи приховані вузькі місця й створюючи цикли зворотного зв'язку, що дозволяє оцінювати реальний ефект змін коду на швидкодію [5], [6].

Визначено, що вдосконалення вихідного коду являє собою систематичний процес покращення його якості, структури та ефективності без зміни основної функціональності. Такий підхід дозволяє зробити програмний продукт більш зрозумілим, надійним і стійким до помилок, а також полегшує подальшу підтримку та розвиток. Методи вдосконалення вихідного коду охоплюють кілька напрямів. Рефакторинг дозволяє оптимізувати внутрішню структуру, роблячи її більш зрозумілою та логічною. Використання автоматизованих інструментів аналізу сприяє виявленню синтаксичних і логічних помилок, а також підвищує безпеку. Сукупність таких методів формує основу для створення надійного, оптимізованого та зручного у подальшому розвитку програмного забезпечення.

1.2 Аналіз програмного забезпечення для вдосконалення вихідного коду

Проаналізуємо програмне забезпечення для вдосконалення вихідного коду [7]-[14].

Програмне забезпечення Codacy (рис. 1.1) призначене для автоматизованого статичного аналізу вихідного коду з метою підвищення його якості. Програмне забезпечення Codacy підтримує велику кількість мов програмування та здатна працювати разом із сучасними платформами контролю версій. Використання Codacy дає змогу здійснювати перевірку коду відповідно до встановлених стандартів і вимог, що робить інструмент ефективним рішенням для командної роботи над проєктами [7], [10], [12].

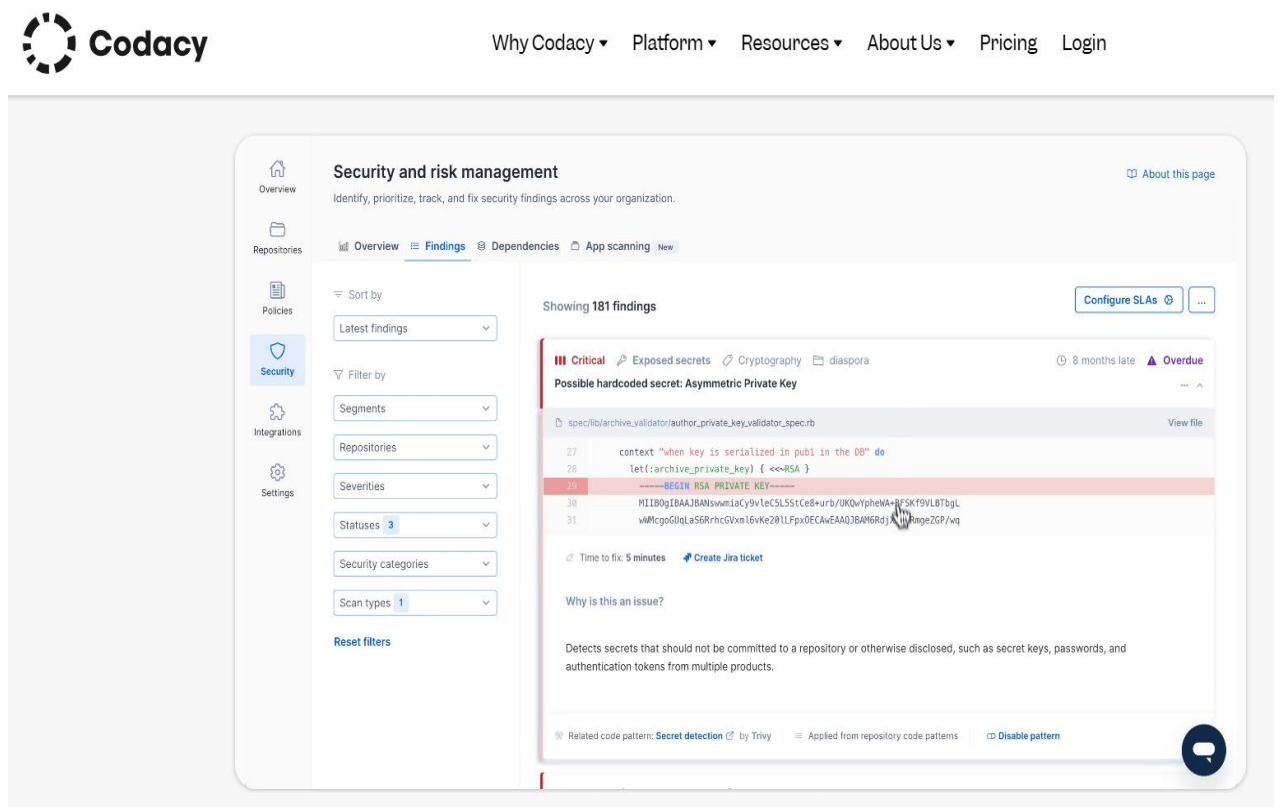


Рисунок 1.1 – Програмне забезпечення Codacy [12]

Застосування Codacy забезпечує формування єдиних правил написання коду, допомагає своєчасно виявляти недоліки та сприяє дотриманню

стандартів розробки. Завдяки цьому програмне забезпечення добре підходить для середовищ, де розробка ведеться постійно й вимагає безперервного контролю якості в процесах інтеграції та доставки. Особливістю системи є простота налаштування й використання, а також можливість індивідуального визначення критеріїв оцінки. Програмне забезпечення Codacy дозволяє організаціям швидко інтегрувати інструмент у робочі процеси, отримувати зворотний зв'язок щодо якості коду та оптимізувати його на ранніх етапах життєвого циклу [7], [10], [12].

Основними характеристиками програмного забезпечення Codacy є такі [7], [10], [12]:

- підтримка мов програмування: програмне забезпечення Codacy працює з великою кількістю мов, що робить його універсальним рішенням для різних проєктів [7], [10], [12];

- інтеграції з системами: програмне забезпечення легко підключається до GitHub, GitLab, Bitbucket та інших платформ контролю версій [7], [10], [12];

- автоматизований аналіз: програмне забезпечення Codacy виконує перевірку коду без втручання користувача, що забезпечує своєчасне виявлення помилок і недоліків [7], [10], [12];

- налаштування правил: програмне забезпечення Codacy надає можливість задавати власні стандарти кодування та визначати параметри відповідно до потреб команди [7], [10], [12];

- зручність використання: інтерфейс програмного забезпечення Codacy відзначається простотою і не потребує складного навчання для розробників та інших користувачів [7], [10], [12];

- контроль якості: забезпечується формування єдиних критеріїв оцінювання коду в межах команди [7], [10], [12];

- робота з великими проєктами: ефективність програмного забезпечення Codacy може знижуватися при аналізі надмірно об'ємних баз коду [7], [10], [12];

- доступність безкоштовного рівня: передбачено базовий пакет можливостей для невеликих і середніх команд [7], [10], [12];
- підтримка спільноти: користувачі програмного забезпечення Codacy мають можливість отримати допомогу завдяки активній спільноті [7], [12];
- обмежені можливості для підприємств: у масштабних організаціях налаштування та технічна підтримка програмного забезпечення Codacy можуть виявлятися недостатньо гнучкими [7], [10], [12].

Серед недоліків програмного забезпечення Codacy можна відзначити обмеженість налаштувань для великих організацій, можливе зниження швидкодії під час аналізу об'ємних кодових баз, а також залежність частини функцій від платних тарифів, що може створювати труднощі для команд з високими вимогами до продуктивності та масштабованості. При обробці надто великих обсягів коду швидкодія може знижуватися, що особливо помітно при використанні безкоштовних можливостей. Для масштабних підприємств рівень налаштувань й підтримки може виявитися менш гнучким, ніж у деяких спеціалізованих рішень [7], [10], [12].

Програмне забезпечення Korbit (рис. 1.2) призначене для вдосконалення вихідного коду та орієнтоване на автоматизацію процесів перевірки. Програмне забезпечення Korbit здатне інтегруватися з популярними платформами керування версіями, де під час обробки запитів відбувається автоматичне сканування коду з метою виявлення критичних помилок, загроз безпеці та проблем, що можуть впливати на продуктивність. Особливістю роботи програмного забезпечення Korbit є надання конкретних порад безпосередньо в середовищі розробки, що створює ефект персонального наставництва під час оптимізації коду. Програмне забезпечення Korbit також містить аналітичну панель, яка відстежує динаміку якості програмного забезпечення в часі, демонструє кількість виправлених недоліків і дає змогу оцінювати ефективність командної роботи. [9], [13].

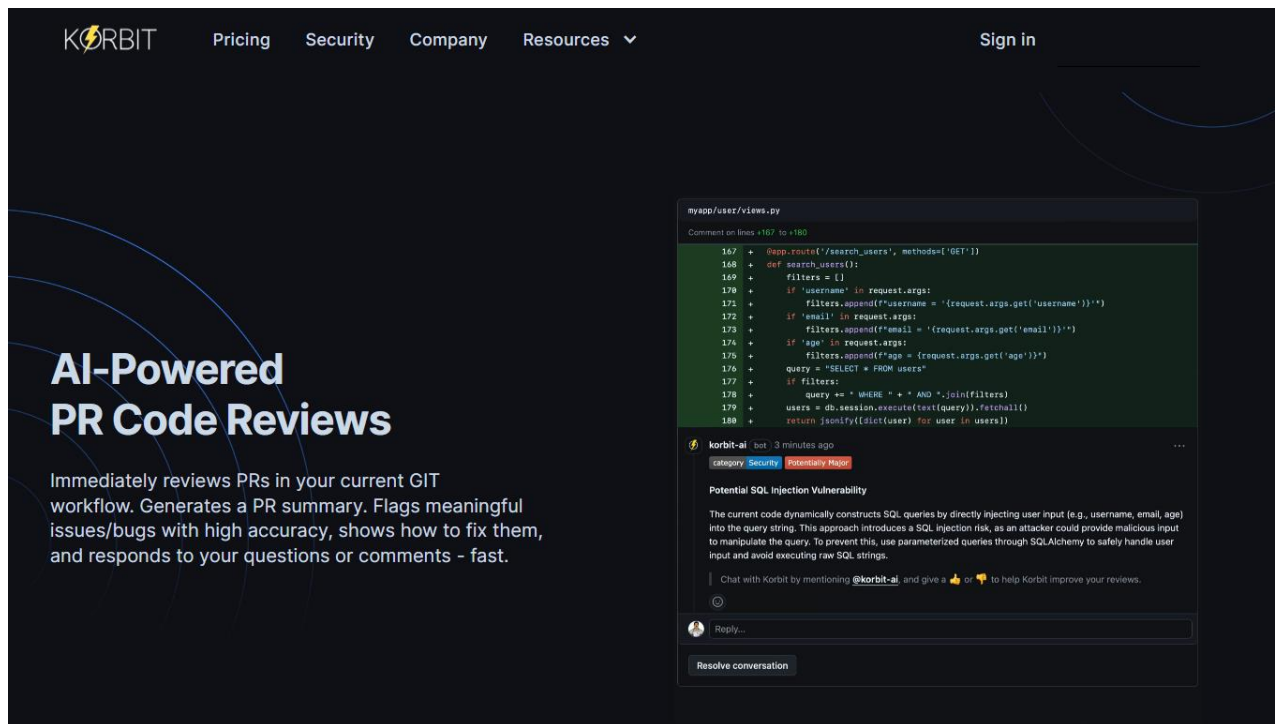


Рисунок 1.2 – Програмне забезпечення Korbit [13]

Програмне забезпечення Korbit дозволяє приймати більш виважені рішення щодо планування і підвищення результативності. Крім того, програмне забезпечення пропонує додаткові можливості навчання та вдосконалення навичок програмування через вправи та рекомендації [9], [13].

Програмне забезпечення Korbit має такі особливості [9], [13]:

- автоматизоване сканування: програмне забезпечення Korbit забезпечує автоматичну перевірку коду з виявленням помилок, вразливостей та проблем продуктивності [9], [13];

- контекстні підказки: зворотний зв'язок подається безпосередньо у код з конкретними порадами щодо виправлень [9], [13];

- панель аналітики: програмне забезпечення Korbit підтримує відображення ключових показників якості коду та динаміки усунення проблем з часом [9], [13];

- підтримка командної роботи: програмне забезпечення Korbit надає аналіз результатів діяльності розробників, що допомагає приймати обґрунтовані рішення при плануванні [9], [13];

- навчальні можливості: пропонуються вправи та рекомендації для вдосконалення практик програмування [9], [13];
- обмеження у складних випадках: програмне забезпечення Korbit менш ефективно обробляє глибокі логічні проблеми коду [9], [13];
- доступність функцій: повний набір можливостей програмного забезпечення Korbit розкривається у платних тарифах, що може бути перешкодою для невеликих команд [9], [13].

Серед недоліків програмного забезпечення Korbit можна відзначити обмежену ефективність у виявленні складних логічних помилок, які потребують глибшого аналізу структури та контексту коду. Також слід зважати на те, що повний набір можливостей стає доступним лише у розширених тарифних планах, що може створювати певні бар'єри для невеликих колективів або індивідуальних розробників, які прагнуть отримати максимальну користь від інструменту [9], [13].

Програмне забезпечення Snyk Code (рис. 1.3) створене для підвищення рівня безпеки та надійності вихідного коду шляхом його безперервного аналізу та виявлення вразливостей у процесі розробки [11], [14].

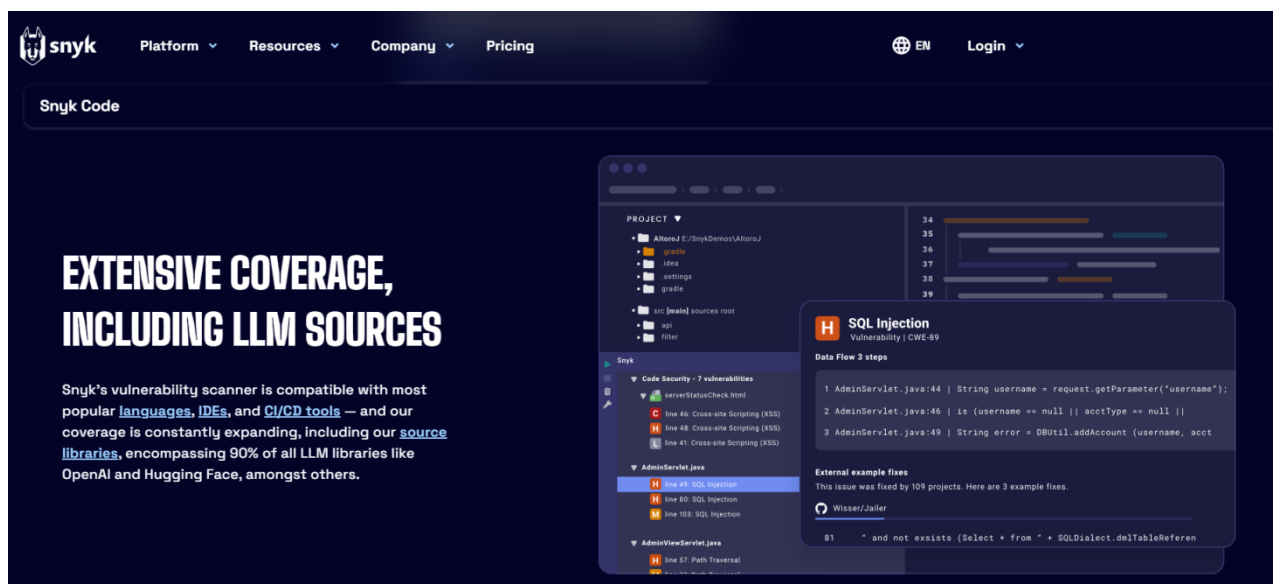


Рисунок 1.3 – Програмне забезпечення Snyk Code [14]

Програмне забезпечення Snyk Code базується на використанні штучного інтелекту, здатного проводити глибоке сканування програмних проєктів у режимі реального часу та пропонувати швидкі варіанти виправлень, що одразу можуть бути застосовані у середовищі розробки. Особливістю є можливість автоматичного присвоєння кожній знайденій проблемі рівня ризику, що дозволяє розробникам визначати, які помилки потребують найшвидшої реакції та виправлення [11], [14].

Програмне забезпечення Snyk Code тісно інтегрується з популярними репозиторіями та інструментами безперервної інтеграції і доставки, що робить його зручним для використання. Крім того, забезпечується підтримка підключення до різних середовищ розробки, що дозволяє виконувати аналіз навіть під час написання коду. Завдяки такому підходу стає можливим поєднання автоматизації процесів перевірки безпеки з гнучким налаштуванням параметрів відповідно до потреб проєкту [11], [14].

Окремо варто відзначити інтерфейс, орієнтований на простоту використання, завдяки якому перевірка та виправлення коду стають зрозумілими навіть для нових користувачів. Хоча у безкоштовному варіанті існують обмеження щодо кількості перевірок, програмне забезпечення все одно зберігає високу практичну цінність для невеликих команд та окремих розробників, що прагнуть отримати доступ до професійного аналізу безпеки коду [11], [14].

Програмне забезпечення Snyk Code має такі особливості [11], [14]:

- інтеграція з IDE: програмне забезпечення Snyk Code легко підключається до популярних середовищ розробки, що дозволяє розробникам отримувати результати аналізу в режимі реального часу [11], [14];

- статичний аналіз коду: програмне забезпечення Snyk Code застосовує методи статичного аналізу для виявлення вразливостей у коді ще на етапі розробки [11], [14];

- виявлення уразливостей: програмне забезпечення Snyk Code автоматичне виявлення проблем безпеки, включаючи SQL-ін'єкції, XSS та інші поширені загрози [11], [14];
- підтримка різних мов: програмне забезпечення Snyk Code підтримує широкий спектр мов програмування, серед яких Java, JavaScript, Python, Go та інші [11], [14];
- безперервний моніторинг: програмне забезпечення Snyk Code дозволяє здійснювати постійну перевірку та надавати сповіщення про нові вразливості у коді та бібліотеках [11], [14];
- рекомендації з виправлення: програмне забезпечення Snyk Code надає детальні поради та приклади щодо усунення виявлених проблем у програмному коді [11], [14];
- інтеграція з інтеграції та тестування: програмне забезпечення Snyk Code підтримує з'єднання з популярними засобами та системами безперервної інтеграції та тестування, що підвищує ефективність DevSecOps-процесів розробки ПЗ [11], [14];
- зручна візуалізація: інтерфейс програмного забезпечення Snyk Code дозволяє швидко переглядати знайдені проблеми, групувати їх за пріоритетністю та відстежувати прогрес усунення [11], [14].

Серед недоліків програмного забезпечення Snyk Code можна відзначити обмеження у повному охопленні рідкісних мов програмування, що зменшує його універсальність для проєктів із нестандартним стеком технологій. Також у деяких випадках інструмент може видавати хибні спрацьовування, що потребує додаткової перевірки результатів з боку розробників. Використання Snyk Code у великих проєктах може призводити до підвищеного навантаження на систему та сповільнення процесів CI/CD. Крім того, доступ до розширених функцій потребує платної підписки, що може стати бар'єром для невеликих команд чи стартапів [11], [14].

Порівняльну характеристику програмного забезпечення для вдосконалення вихідного коду наведено у таблиці 1.1.

Таблиця 1.1 – Порівняння програмного забезпечення для вдосконалення вихідного коду

Критерій порівняння	Codacy [12]	Korbit [13]	Snyk Code [14]
Виявлення вразливостей у коді	+—	+	+
Підтримка різних мов програмування	+	+—	+—
Оцінка ризиків для знайдених проблем	—	+—	+
Інтуїтивність інтерфейсу	+	+—	+
Автоматичні виправлення	—	+—	+
Гнучкість налаштувань	+	+—	+

За результатами проведеного аналізу можна зробити висновок, що у наш час існує досить багато програмних засобів для вдосконалення вихідного коду. Проте деякі програмні засоби для вдосконалення вихідного коду є орієнтованими на певні мови програмування чи конкретні середовища, що зменшує універсальність і робить їх менш ефективними у випадку багатоплатформних рішень. Іншим недоліком є ймовірність появи нових помилок у процесі рефакторингу або оптимізації, особливо якщо інструменти використовуються без належного тестування. Крім того, застосування таких програмних засобів може вимагати значних ресурсів, як обчислювальних, так і часових, що знижує ефективність при роботі з великими проектами. Також важливо враховувати, що деякі програмні засоби вдосконалення вихідного коду не завжди забезпечують гнучкість, оскільки їхні алгоритми ґрунтуються на загальних правилах і не враховують унікальні вимоги конкретного програмного проєкту. Тому актуальною є розробка програмного забезпечення для вдосконалення вихідного коду.

При розробці програмного забезпечення для вдосконалення вихідного коду необхідно забезпечити такі функціональні вимоги:

- підтримка можливості працювати з текстовими наборами даних, що являють собою програмний код;
- підтримка можливості вдосконалення текстів з програмними кодами, написаними на різних мовах програмування;
- можливість авторизації користувачів;
- можливість виведення інформації щодо покращень у програмному коді;
- підтримка можливості користувачів копіювання покращеного програмного коду для подальшої роботи.

1.3 Висновки за розділом 1

Визначено, що вдосконалення вихідного коду являє собою систематичний процес покращення його якості, структури та ефективності без зміни основної функціональності. Такий підхід дозволяє зробити програмний продукт більш зрозумілим, надійним і стійким до помилок, а також полегшує подальшу підтримку та розвиток. Методи вдосконалення вихідного коду охоплюють кілька напрямів. Рефакторинг дозволяє оптимізувати внутрішню структуру, роблячи її більш зрозумілою та логічною. Використання автоматизованих інструментів аналізу сприяє виявленню синтаксичних і логічних помилок, а також підвищує безпеку. Сукупність таких методів формує основу для створення надійного, оптимізованого та зручного у подальшому розвитку програмного забезпечення.

За результатами проведеного аналізу зроблено висновок, що у наш час існує досить багато програмних засобів для вдосконалення вихідного коду. Проте деякі програмні засоби для вдосконалення вихідного коду є орієнтованими на певні мови програмування чи конкретні середовища, що зменшує універсальність і робить їх менш ефективними у випадку

багатоплатформних рішень. Іншим недоліком є ймовірність появи нових помилок у процесі рефакторингу або оптимізації, особливо якщо інструменти використовуються без належного тестування. Крім того, застосування таких програмних засобів може вимагати значних ресурсів, як обчислювальних, так і часових, що знижує ефективність при роботі з великими проектами. Також важливо враховувати, що деякі програмні засоби вдосконалення вихідного коду не завжди забезпечують гнучкість, оскільки їхні алгоритми ґрунтуються на загальних правилах і не враховують унікальні вимоги конкретного програмного проєкту. Тому актуальною є розробка програмного забезпечення для вдосконалення вихідного коду.

Сформульовано функціональні вимоги до програмного забезпечення для вдосконалення вихідного коду.

2 МАТЕРІАЛИ І МЕТОДИ

2.1 Вибір мови програмування

При розробці програмного забезпечення для вдосконалення вихідного коду було використано мову програмування JavaScript [15], [16].

JavaScript є однією з найбільш поширених мов програмування, яка активно використовується не лише у веброзробці, а й у створенні серверних застосунків, мобільних програм та навіть настільних систем. Її головною особливістю вважається можливість працювати безпосередньо у браузері, що забезпечує інтерактивність і динамічність сучасних вебсторінок. Завдяки цьому мова стала стандартом у фронтенд розробці, а з появою середовища Node.js отримала широке застосування і на боці сервера [15], [16].

Завдяки динамічній природі та підтримці асинхронного програмування мова JavaScript дає змогу ефективно працювати з потоками даних і зовнішніми програмними сервісами. Екосистема цієї мови постійно зростає, а численні фреймворки та бібліотеки значно розширюють можливості розробників і скорочують час реалізації проєктів. Велика спільнота підтримує активний розвиток мови, що забезпечує швидке вдосконалення інструментів і появу нових рішень [15], [16].

Завдяки кросплатформеності та відкритості стандартів мова програмування JavaScript розглядається як універсальний інструмент, здатний об'єднувати різні технології та забезпечувати сумісність у межах складних програмних систем. Гнучкість і доступність роблять мову програмування JavaScript привабливою як для початківців, так і для досвідчених фахівців, а також забезпечують можливість створення програмного забезпечення різного рівня складності [15], [16].

Вибір JavaScript для створення програмного забезпечення, яке реалізує методи вдосконалення вихідного коду, є доцільним з кількох причин. Насамперед ця мова є універсальною та підтримується практично в усіх середовищах, що робить її зручною як для розробки інструментів у вигляді

окремих застосунків, так і для інтеграції з вебсервісами. Завдяки Node.js JavaScript можна використовувати не лише на клієнтській стороні, а й на сервері, що відкриває можливість створення повноцінних систем аналізу та обробки коду [15], [16].

JavaScript має розвинену екосистему з великою кількістю готових бібліотек та фреймворків. Це дозволяє спростити реалізацію складних алгоритмів, пов'язаних зі статичним аналізом, рефакторингом або перевіркою стилю коду. Прикладом можуть слугувати ESLint, Prettier чи Babel — інструменти, написані саме на JavaScript і широко застосовувані у світовій практиці [15], [16].

Ще однією перевагою є кросплатформеність: інструменти на JavaScript можуть однаково добре працювати на різних операційних системах і легко інтегруватися у процеси безперервної інтеграції. Крім того, популярність мови та велика спільнота розробників забезпечують активний розвиток технологій і швидкий доступ до підтримки [15], [16].

Обґрунтування вибору мови програмування для реалізації програмного забезпечення для вдосконалення вихідного коду наведено у таблиці 2.1.

Таблиця 2.1 – Обґрунтування вибору мови програмування для реалізації програмного забезпечення для вдосконалення вихідного коду

Критерій порівняння мов програмування	Мова програмування		
	JavaScript	Python	C#
Поширеність у вебсередовищах	+	+—	+—
Розвинена екосистема бібліотек	+	+	+—
Асинхронне програмування	+	+—	+—
Зручність для створення ПЗ для вдосконалення вихідного коду	+	—	—
Швидкість розробки	+	+	+—

Таким чином, JavaScript дає змогу створити гнучкі, масштабовані та зручні у використанні засоби для вдосконалення вихідного коду, які можна застосовувати як у локальних середовищах, так і у хмарних сервісах [15], [16].

Отже, для реалізації програмного забезпечення для вдосконалення вихідного коду обрано мову програмування JavaScript, що обґрунтовується універсальністю та підтримкою у більшості сучасних середовищ. Гнучкість синтаксису та підтримка асинхронних операцій роблять JavaScript ефективним інструментом для роботи з потоками даних і зовнішніми ресурсами. Екосистема мови стрімко розвивається, а велика кількість додаткових бібліотек і фреймворків значно розширює спектр можливостей і скорочує час реалізації складних проєктів. Активна спільнота забезпечує постійне вдосконалення стандартів і появу нових рішень, що дозволяє швидко адаптувати мову до сучасних потреб розробників.

2.2 Вибір бібліотеки для створення інтерфейсу програмного забезпечення для вдосконалення вихідного коду

В якості бібліотеки для створення інтерфейсу програмного забезпечення для вдосконалення вихідного коду було обрано React [17], [18].

Бібліотека React є ефективним інструментом для побудови інтерфейсів, здатним значно спростити створення інтерактивних програмних рішень. Його основна ідея полягає в оптимізації роботи з елементами користувацького середовища, які повинні змінюватися відповідно до дій користувача чи стану системи. Використання віртуального DOM забезпечує вибіркове оновлення потрібних компонентів сторінки, що позитивно впливає на швидкодію та ефективність застосунків [17], [18].

Бібліотека React побудована на компонентній архітектурі, яка передбачає поділ інтерфейсу на автономні модулі. Такий підхід дозволяє повторно використовувати створені елементи, комбінувати їх у різних конфігураціях і без проблем масштабувати проєкти. Структурованість коду,

яку забезпечує React, полегшує підтримку програмного забезпечення та робить його більш надійним у довгостроковому використанні. Окрім того, інтеграція з іншими інструментами екосистеми JavaScript відкриває можливості для розширення функціональності та налаштування під конкретні завдання [17], [18].

Ще однією перевагою є популярність React і велика спільнота, що гарантує постійний розвиток технології та доступність рішень для типових проблем. React також добре поєднується з Node.js та іншими JavaScript-інструментами, завдяки чому створюється єдиний технологічний стек для клієнтської та серверної частини. Важливою перевагою React є універсальність, адже на основі цієї бібліотеки можна створювати як вебзастосунки, так і мобільні рішення завдяки React Native. Широка популярність інструмента сприяла формуванню великої спільноти, що активно вдосконалює технологію, створює додаткові бібліотеки та допомагає швидко вирішувати проблеми, які виникають у процесі розробки [17], [18].

Для створення програмних систем, пов'язаних з удосконаленням вихідного коду, React вважається доцільним вибором завдяки своїй здатності формувати наочні, інтерактивні та гнучкі інтерфейси. Компонентний підхід робить можливим реалізацію складних панелей управління, зручне відображення результатів аналізу та підтримку функцій, що стосуються роботи з великими масивами даних. Віртуальний DOM гарантує стабільну продуктивність навіть у разі високих навантажень, що особливо актуально під час обробки коду [17], [18].

Розвинена екосистема бібліотеки React дозволяє швидко інтегрувати додаткові інструменти, серед яких редактори, модулі підсвічування синтаксису чи рішення для роботи з зовнішніми сервісами. Високий рівень поширеності React і його сумісність з Node.js формують єдиний технологічний простір, який охоплює як клієнтську, так і серверну частину програмного забезпечення [17], [18].

Таким чином, використання React забезпечує створення масштабованого та сучасного інтерфейсу для систем, орієнтованих на аналіз і покращення вихідного коду, поєднуючи продуктивність, гнучкість і якісний користувацький досвід [17], [18].

Обґрунтування вибору бібліотеки для створення інтерфейсу програмного забезпечення для вдосконалення вихідного коду наведено у таблиці 2.2.

Таблиця 2.2 – Обґрунтування вибору бібліотеки для створення інтерфейсу програмного забезпечення для вдосконалення вихідного коду

Критерій порівняння бібліотек для створення інтерфейсу	Бібліотеки для створення інтерфейсу		
	React	Angular	Svelte
Продуктивність інтерфейсу	+	+—	+
Гнучкість та масштабованість	+	+	+—
Розвинена екосистема та бібліотеки	+	+	—
Популярність та активність спільноти	+	+	+—
Підтримка інтеграції з інструментами для роботи з кодом	+	+—	—

Таким чином, для створення програмного забезпечення для вдосконалення вихідного коду обрано бібліотеку React, що поєднує продуктивність, гнучкість і зручність у використанні. Компонентна архітектура React дозволяє створювати інтерфейси, що легко масштабуються й підтримуються навіть у складних проєктах. Крім того, забезпечується

ефективне оновлення елементів, що дає можливість працювати з великими обсягами даних без зниження швидкодії. Розвинена екосистема бібліотек і численні готові рішення дозволяють швидко додавати необхідний функціонал, наприклад інтеграцію з редакторами коду, підсвічування синтаксису чи інструментами візуалізації. Кросплатформеність відкриває можливість створювати не лише вебінтерфейси, а й мобільні застосунки на основі тієї ж технології, що робить розробку універсальною.

2.3 Розробка методу вдосконалення вихідного коду

У сучасних умовах швидкого розвитку програмного забезпечення особливої актуальності набуває пошук нових підходів до вдосконалення вихідного коду. Традиційні методи оптимізації зосереджуються переважно на зменшенні складності алгоритмів, покращенні архітектурних рішень чи усуненні надмірностей. Однак постійна зміна вимог і необхідність підтримки великого життєвого циклу продуктів вимагають появи додаткових методів, які б не лише спрощували структуру коду, а й сприяли його довготривалій стабільності та гнучкості.

Запропоновано метод удосконалення програмного коду передбачає автоматичне додавання до ключових фрагментів програми спеціальних коментарів, які не лише описують логіку роботи, а й містять метадані про частоту викликів, критичність виконання та взаємозалежності з іншими компонентами. Такі коментарі можуть генеруватися засобами статичного й динамічного аналізу, а також оновлюватися під час роботи програми. Це створює можливість швидко виявляти ділянки з найбільшим навантаженням або ризиком виникнення помилок, адаптувати підхід до їх оптимізації та, за потреби, автоматично пропонувати варіанти їх переписування. На відміну від класичних методів рефакторингу, такий підхід дозволяє підтримувати "живий профіль" коду, який поєднує описову документацію з практичною статистичною інформацією.

Метод вдосконалення вихідного коду може бути поданий за допомогою рис. 2.1.

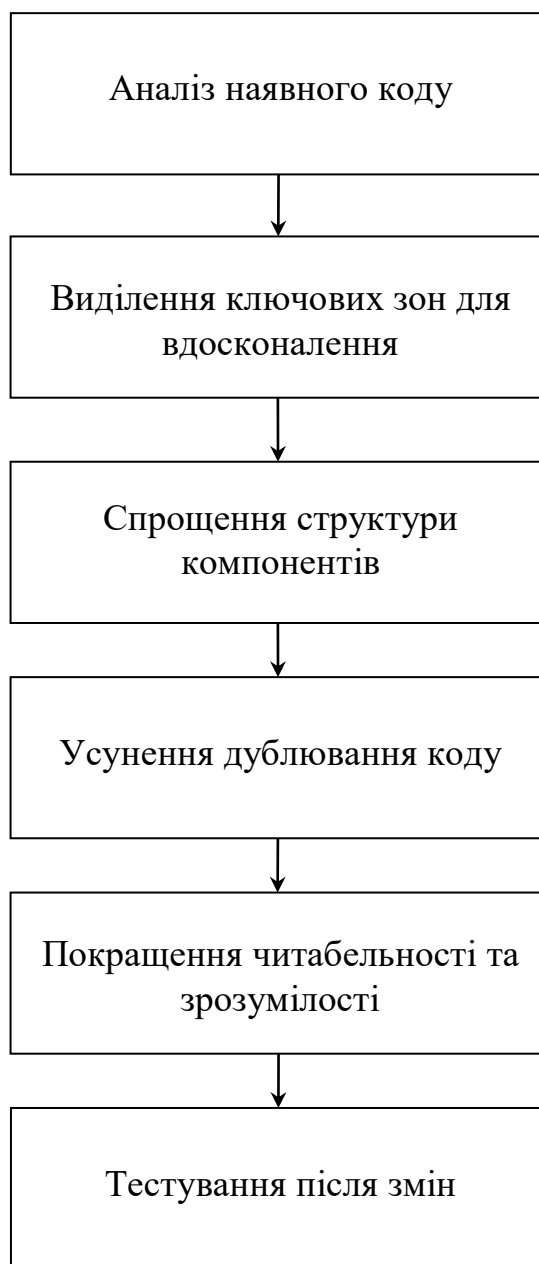


Рисунок 2.1 – Схематичне подання методу вдосконалення вихідного коду

Проаналізуємо основні етапи методу вдосконалення вихідного коду. Етап аналіз наявного коду: будь-яке вдосконалення починається з уважного вивчення того, що вже написано. На цьому етапі розробник переглядає структуру файлів, організацію компонентів, стиль написання функцій та

наявність повторюваних фрагментів. Головна мета – зрозуміти логіку програми та визначити проблемні місця: занадто громіздкі функції, дублювання коду, незрозумілі назви змінних чи надто тісне переплетення компонентів між собою. Саме на цьому етапі формується загальна "карта" проєкту, яка допоможе рухатися далі.

Виділення ключових зон для вдосконалення: після загального огляду варто визначити ті ділянки, які приносять найбільше труднощів у розумінні та підтримці. Це можуть бути компоненти з великою кількістю вкладеної логіки, функції з десятками умовних операторів, або місця, де код повторюється майже без змін. Також увага приділяється організації стану: чи використовується він надмірно у верхніх компонентах, чи можливо його спустити ближче до конкретних елементів.

Спрощення структури компонентів: один із ключових принципів вдосконалення коду полягає у тому, щоб робити компоненти якомога простішими. На цьому етапі великі та перевантажені компоненти розбиваються на менші. Кожен з них отримує лише одну чітку відповідальність, щоб було легко зрозуміти, для чого він створений. Наприклад, замість одного компонента, який і завантажує дані, і обробляє їх, і відображає таблицю, краще створити три: один для завантаження, другий для обробки та третій для візуалізації. Дуже часто код перевантажений станами, які зберігаються на занадто високому рівні. Це призводить до непотрібних дій та ускладнює логіку. На цьому етапі відбувається перегляд того, де саме має зберігатися стан, і чи можна його локалізувати. Також тут застосовуються React-хуки, які дозволяють зробити роботу зі станом більш зрозумілою та компактною.

Усунення дублювання коду: часто одна й та сама логіка зустрічається у різних частинах проєкту. У таких випадках найкраще винести її у функцію або окремий компонент. Це робить код більш передбачуваним, адже зміни потрібно буде вносити лише в одному місці, а не шукати всі повтори по

всьому проєкту. Крім того, це знижує ймовірність появи помилок у майбутньому.

Важливий етап, який стосується сприйняття, полягає у покращенні читабельності та зрозумілості. Тут розробник упорядковує імена змінних, робить їх інформативними, позбавляється від зайвих коментарів або, навпаки, додає пояснення там, де без них складно зрозуміти логіку. Також використовується єдиний стиль оформлення коду: відступи, форматування JSX, розташування пропсів у компонентах. Це робить код "чистим" і легким для сприйняття.

Жоден процес вдосконалення коду не може обійтися без перевірки результату. Після внесення змін проводиться ретельне тестування: перевіряється, чи все працює так само, як і до рефакторингу. Якщо проєкт має автоматизовані тести, вони виконуються, щоб переконатися у відсутності зламаних функцій. Якщо тестів немає, хоча б базові сценарії перевіряються вручну. Мета цього кроку – підтвердити, що покращення структури не вплинуло на коректність роботи.

Коли код уже став більш структурованим і зрозумілим, на останньому етапі можна звернути увагу на продуктивність. Це може включати оптимізацію рендерингу за допомогою `React.memo`, більш ефективного використання ключів у списках, зменшення зайвих перерахунків даних. Тут також переглядаються сторонні залежності: можливо, деякі бібліотеки можна замінити на легші аналоги, або взагалі від них відмовитися.

У результаті цього методу вихідний код стає більш структурованим, гнучким і зрозумілим. Кожен компонент виконує лише одну роль, стан використовується раціонально, дублювання усунуте, а код легко читати і підтримувати.

Таким чином, запропоновано метод динамічного маркування коду, який полягає у поєднанні коментарів з метаданими та автоматизованого оновлення цих даних під час виконання програм, що дозволяє підвищити

прозорість програмного коду, спростити його оптимізацію та забезпечити гнучку підтримку на всіх етапах життєвого циклу.

2.4 Висновки за розділом 2

Для реалізації програмного забезпечення для вдосконалення вихідного коду обрано мову програмування JavaScript, що обґрунтовується універсальністю та підтримкою у більшості сучасних середовищ. Гнучкість синтаксису та підтримка асинхронних операцій роблять JavaScript ефективним інструментом для роботи з потоками даних і зовнішніми ресурсами. Екосистема мови стрімко розвивається, а велика кількість додаткових бібліотек і фреймворків значно розширює спектр можливостей і скорочує час реалізації складних проєктів. Активна спільнота забезпечує постійне вдосконалення стандартів і появу нових рішень, що дозволяє швидко адаптувати мову до сучасних потреб розробників.

Для створення програмного забезпечення для вдосконалення вихідного коду обрано бібліотеку React, що поєднує продуктивність, гнучкість і зручність у використанні. Компонентна архітектура React дозволяє створювати інтерфейси, що легко масштабуються й підтримуються навіть у складних проєктах. Крім того, забезпечується ефективне оновлення елементів, що дає можливість працювати з великими обсягами даних без зниження швидкодії. Розвинена екосистема бібліотек і численні готові рішення дозволяють швидко додавати необхідний функціонал, наприклад інтеграцію з редакторами коду, підсвічування синтаксису чи інструментами візуалізації.

Запропоновано метод динамічного маркування коду, який полягає у поєднанні коментарів з метаданими та автоматизованого оновлення цих даних під час виконання програм, що дозволяє підвищити прозорість програмного коду, спростити його оптимізацію та забезпечити гнучку підтримку на всіх етапах життєвого циклу.

3 ОПИС ПРОГРАМИ

3.1 Структура програмного забезпечення для вдосконалення вихідного коду

Структуру програмного забезпечення для вдосконалення вихідного коду наведено на рис. 3.1.

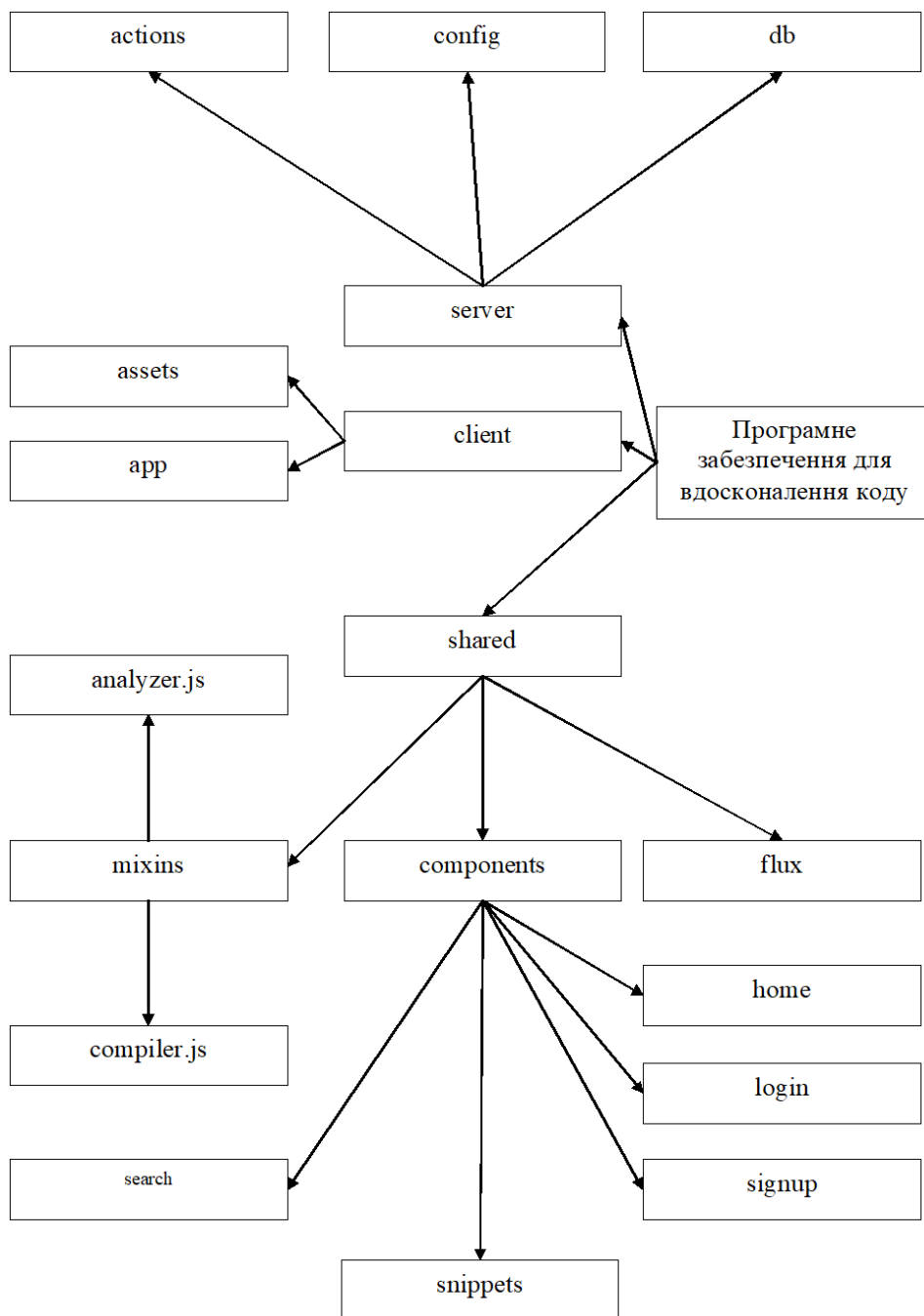


Рисунок 3.1 – Структура програмного забезпечення для вдосконалення вихідного коду

Архітектура програмного забезпечення, орієнтованого на вдосконалення вихідного коду, побудована як багаторівнева система, у якій окремо функціонують серверна частина, клієнтська частина та спільні модулі. Такий поділ дозволяє чітко розмежувати відповідальність між різними елементами системи та забезпечує зручність у підтримці й розширенні функціоналу. Сервер виступає центральною ланкою, що обробляє запити, керує взаємодією з базою даних і зберігає налаштування, необхідні для стабільної роботи застосунку. На клієнтській стороні розміщуються ресурси, які відповідають за відображення інтерфейсу та взаємодію з користувачем, включаючи стилі оформлення, графічні елементи й інструменти для візуалізації коду з підсвічуванням синтаксису.

У спільному сегменті системи зосереджено механізми, що забезпечують повторне використання компонентів і їх узгодженість у межах усього середовища. Тут реалізуються елементи управління, необхідні для роботи з різними модулями інтерфейсу: пошук та обробка фрагментів програмного коду, організація процесу авторизації й реєстрації, а також перехід між основними розділами застосунку. Додатково у спільній частині інтегровано логіку, що відповідає за оптимізацію та компіляцію вибраних ділянок коду, а також архітектурні рішення, які підтримують передбачуваність і керованість стану системи.

Важливою складовою є модуль аналізу програмного тексту, реалізований у вигляді окремого файлу, де виконуються основні перевірки. Його робота ґрунтується на регулярних виразах, завдяки яким здійснюється виявлення у коді шаблонів, пов'язаних із типовими помилками або неефективними рішеннями. Для кожної потенційної проблеми передбачені функції, що забезпечують відстеження певних аспектів. Серед таких можна виділити визначення некоректного використання оператора для видалення, ідентифікацію випадків застосування глобальних змінних та перевірку ситуацій, коли подієві обробники залишаються не відв'язаними від елементів сторінки.

Робота цих функцій побудована так, щоб у результаті аналізу формувалися рекомендації, спрямовані на усунення проблем та покращення структури коду. Наприклад, при виявленні помилкового використання `delete` пропонується заміна цього підходу на призначення значення `null` змінним, які більше не потрібні. Якщо ж виявляється використання глобальних змінних, система радить перенести їх у локальний контекст або застосувати інші методи інкапсуляції. У випадку з подієвими обробниками перевіряється не лише факт їхнього додавання, а й наявність відповідних механізмів для їхнього відв'язування.

Таким чином, структура програмного забезпечення для вдосконалення вихідного коду організована таким чином, щоб забезпечити чіткий поділ обов'язків між серверною логікою, клієнтським інтерфейсом і спільними інструментами, а також надати можливість здійснювати автоматизований аналіз тексту програм високого рівня з метою пошуку і подальшого усунення проблемних рішень, що негативно впливають на продуктивність або стійкість роботи коду.

3.2 Функціонування програмного забезпечення для вдосконалення вихідного коду

Функціонування програмного забезпечення для вдосконалення вихідного коду подамо за допомогою схеми, зображеної на рис. 3.2.

Функціонування програмного забезпечення для вдосконалення вихідного коду можна описати як послідовний процес перетворення та інтерпретації, у якому задіяні кілька рівнів обробки. Початкова точка цього процесу зосереджена у JSX-файлах, що містять опис структури інтерфейсу та логіку його взаємодії. Вони не можуть бути безпосередньо виконані браузером, тому на наступному етапі піддаються трансформації за допомогою інструментів React, у результаті чого утворюються так звані React JSX-файли.

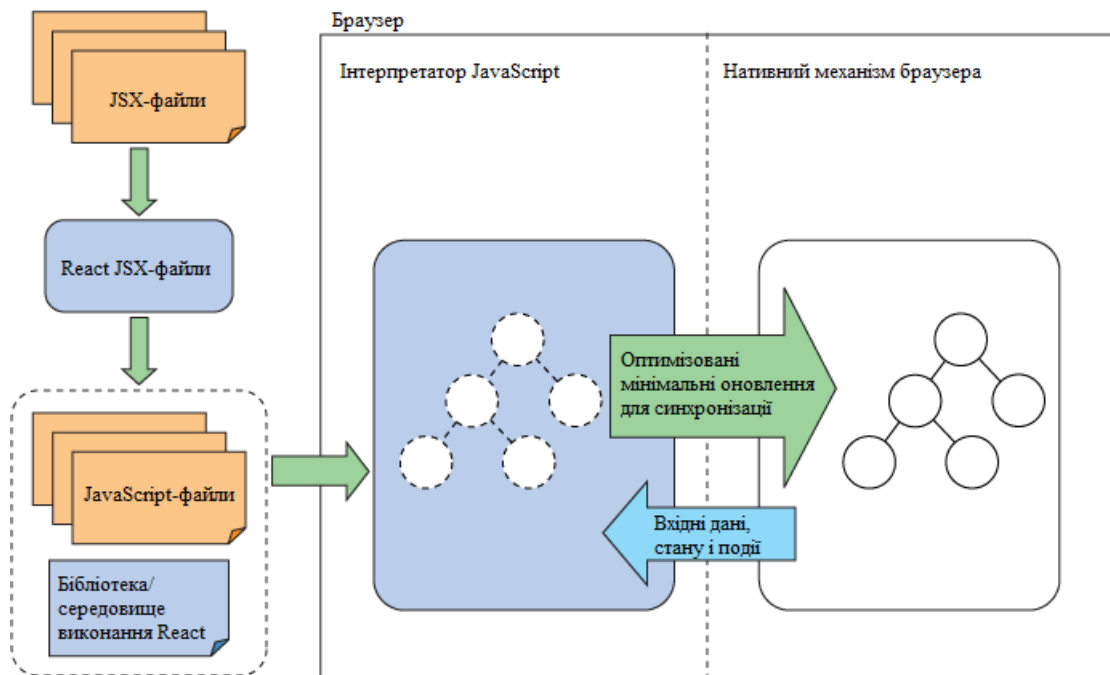


Рисунок 3.2 – Схема функціонування програмного забезпечення для вдосконалення вихідного коду

На цьому рівні забезпечується відповідність внутрішньому представленню компонентів та їхньому відображенню у віртуальному середовищі, де відбувається підготовка до подальшої оптимізації.

Після перетворення React JSX-файли компілюються у стандартні JavaScript-файли, які вже можуть бути інтерпретовані будь-яким сучасним браузером. Саме тут формується кінцева логіка поведінки програми та визначаються усі залежності, потрібні для її коректного виконання. Важливим є те, що ці файли містять не лише інструкції для побудови інтерфейсу, а й механізми оптимізації оновлень, які є ключовими у зменшенні навантаження на систему.

Далі весь сформований JavaScript-код передається безпосередньо до браузера. Усередині нього робота програми розподіляється між двома основними складовими: інтерпретатором JavaScript та нативним механізмом браузера. Інтерпретатор відповідає за обробку коду, а також за формування оптимізованих оновлень, що надсилаються у нативний механізм. Ці оновлення мають мінімальний характер і синхронізуються з відображенням лише у тих

місцях, де відбулися зміни, що дозволяє уникати зайвих перерендерів та економити ресурси.

У зворотному напрямку нативний механізм браузера постачає інтерпретатор даними про поточний стан, вхідними подіями користувача та іншою контекстною інформацією. Це забезпечує постійний цикл взаємодії, у якому відбувається не лише синхронізація візуального середовища з логікою додатку, а й реагування на будь-які дії користувача. Завдяки такому обміну підтримується висока динамічність інтерфейсу, його інтерактивність та стійкість навіть за умов значних змін даних.

У результаті створюється ефективний механізм функціонування програмного забезпечення для вдосконалення вихідного коду, у якому послідовні етапи трансформації JSX-описів у JavaScript-файли поєднуються з роботою інтерпретатора та нативного механізму браузера, що забезпечує як оптимізовану обробку, так і мінімальні оновлення у візуальному середовищі.

3.3 Розробка бази даних програмного забезпечення для вдосконалення вихідного коду

База даних програмного забезпечення для вдосконалення вихідного коду є основою для збереження всієї інформації, яка виникає в процесі роботи із вдосконаленням вихідного коду. Вона виконує роль центрального сховища, де поєднуються як початкові дані у вигляді вихідних текстів програм, так і вторинні артефакти, отримані після запуску алгоритмів автоматичного аналізу. У такій структурі фіксується стан коду, відображаються знайдені недоліки, рекомендації щодо покращення та зміни, які були зроблені розробниками після внесення виправлень.

Основу цієї системи становить інформація про програмний проєкт, який виступає відповідною точкою для організації всієї іншої логіки. До проєкту прив'язуються його файли, кожен із яких детально описується шляхом до сховища, назвою та додатковими характеристиками, що

відображають його розмір і останню модифікацію. Усередині файлів можуть виділятися фрагменти, що відповідають окремим класам, функціям чи методам, і саме ці елементи часто стають об'єктами аналізу.

Важливим складником є збереження інформації про запуски аналізу, які документуються у вигляді окремих записів. Для кожного запуску зберігається тип перевірки, назва інструменту, час виконання та результат. На основі цих запусків у базі формуються результати обчислення метрик, що можуть стосуватися як усього файлу, так і окремого фрагмента. Паралельно з метриками фіксуються знайдені проблеми, які класифікуються за типами та рівнем критичності. Кожна проблема отримує опис і час виявлення, що дозволяє відслідковувати її історію.

Щоб надати корисність цьому процесу, для проблем зберігаються рекомендації, які містять конкретні пропозиції щодо виправлення, пояснення їх доцільності та рівень пріоритетності. Важливим елементом є також відображення факту внесених змін у код: для кожної проблеми зберігаються варіанти старого та нового коду, час модифікації й автор. Це дозволяє простежити повний шлях від виявлення недоліку до його усунення.

У випадку, коли система використовується декількома розробниками, у базі даних ведеться облік користувачів. Для кожного з них фіксуються контактні дані, роль у системі та внесок у спільну роботу. Для відстеження активності використовується журнал дій, у якому зберігаються записи про виконані кроки, час їх здійснення та деталі, що дозволяють аналізувати ефективність командної роботи.

Призначення такої бази полягає не лише у збереженні даних, але й у створенні основи для систематичного вдосконалення коду. Вона дозволяє бачити динаміку якості проєкту, відстежувати історію внесених змін, підтримувати автоматизований аналіз і пропонувати варіанти оптимізації. Завдяки інтеграції із журналом дій забезпечується можливість командної співпраці, що робить процес вдосконалення більш прозорим і контрольованим.

Схему бази даних програмного забезпечення для вдосконалення вихідного коду наведено на рис. 3.3.

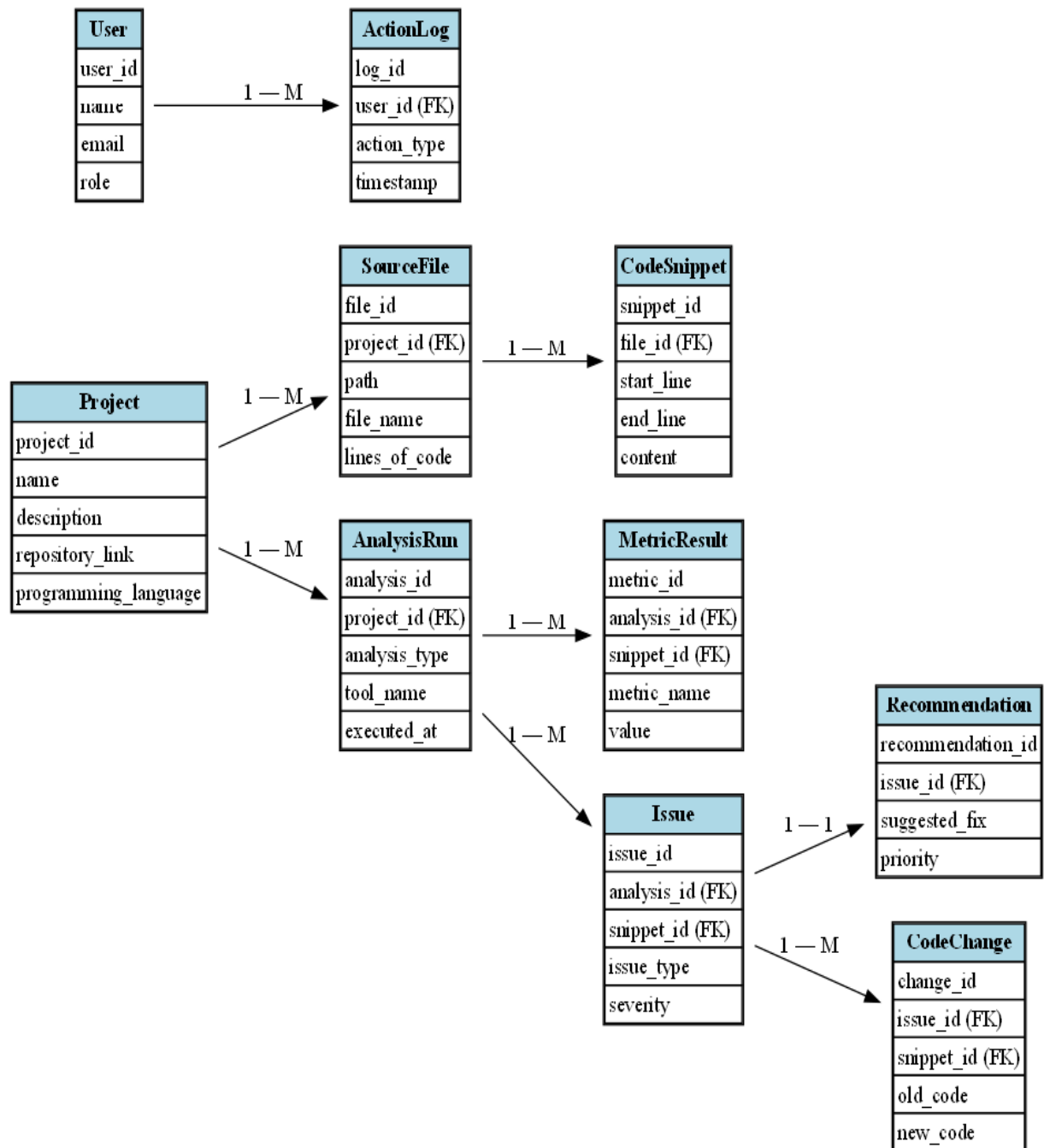


Рисунок 3.3 – Схема бази даних програмного забезпечення для вдосконалення вихідного коду

Схема бази даних програмного забезпечення для вдосконалення вихідного коду відображає концептуальні зв'язки між обраними елементами у

відповідній предметній області. Основними сутностями бази даних обрано такі: проєкт (Project), Файл вихідного коду (SourceFile), Фрагмент коду (CodeSnippet), Аналіз (AnalysisRun), Виявлена проблема (Issue), Рекомендація (Recommendation), Зміна коду (CodeChange) та інші.

Сутність Проєкт (Project) пістить інформацію про програмний продукт або окремий модуль. Атрибути: `project_id`, `name`, `description`, `repository_link`, `programming_language`, `created_at`, `updated_at`.

Сутність Файл вихідного коду (SourceFile) описує окремий файл у межах проєкту. Атрибути: `file_id`, `project_id` (FK), `path`, `file_name`, `lines_of_code`, `last_modified`.

Сутність Фрагмент коду (CodeSnippet) зберігає виділені частини коду (наприклад, функції, класи, методи). Атрибути: `snippet_id`, `file_id` (FK), `start_line`, `end_line`, `content`, `complexity_level`.

Сутність Аналіз (AnalysisRun) фіксує факт запуску інструменту аналізу коду. Атрибути: `analysis_id`, `project_id` (FK), `analysis_type` (статичний, динамічний, метрики якості), `tool_name`, `executed_at`, `status`.

Сутність Метрика якості (MetricResult) містить результати обчислення метрик для файлів чи фрагментів коду. Атрибути: `metric_id`, `analysis_id` (FK), `snippet_id` (FK, optional), `metric_name`, `value`.

Сутність Виявлена проблема (Issue) репрезентує знайдену помилку чи недолік. Атрибути: `issue_id`, `analysis_id` (FK), `snippet_id` (FK), `issue_type` (наприклад, `code smell`, баг, порушення стилю), `severity`, `description`, `detected_at`.

Сутність Рекомендація (Recommendation) містить пораду для виправлення або покращення. Атрибути: `recommendation_id`, `issue_id` (FK), `suggested_fix`, `rationale`, `priority`.

Сутність Зміна коду (CodeChange) описує, які зміни були внесені для усунення проблеми. Атрибути: `change_id`, `issue_id` (FK), `snippet_id` (FK), `old_code`, `new_code`, `modified_at`, `author`.

Сутність Користувач (User) призначена для ведення обліку користувачів. Атрибути: `user_id`, `name`, `email`, `role` (`developer`, `reviewer`, `admin`).

Сутність Журнал дій (ActionLog) призначена для відстеження роботи користувачів у системі. Атрибути: log_id, user_id (FK), action_type, timestamp, details.

Отже, база даних програмного забезпечення для вдосконалення вихідного коду дозволяє зберігати історію стану коду та його вдосконалень, відстежувати еволюцію якості програмного забезпечення, давати змогу автоматизувати аналіз і рекомендації, підтримувати спільну роботу команди (через журнал дій і призначення завдань).

3.4 Проєктування інтерфейсу програмного забезпечення для вдосконалення вихідного коду

Проєктування інтерфейсу взаємодії користувача з програмним забезпеченням для вдосконалення вихідного коду виконано з урахуванням вимог технічного завдання. При розробленні програмного забезпечення для вдосконалення вихідного коду враховані функціональні вимоги до програми:

- підтримка можливості працювати з текстовими наборами даних, що являють собою програмний код;
- підтримка можливості вдосконалення текстів з програмними кодами, написаними на різних мовах програмування;
- можливість авторизації користувачів;
- можливість виведення інформації щодо покращень у програмному коді;
- підтримка можливості користувачів копіювання покращеного програмного коду для подальшої роботи.

Інтерфейс головної форми програмного забезпечення для вдосконалення вихідного коду наведено на рис. 3.4.

Оберіть мову програмування



Оптимізатор Python

Оптимізуйте фрагменти коду Python для кращої продуктивності.

Відкрити



Оптимізатор JavaScript

Оптимізуйте фрагменти коду JavaScript для кращої продуктивності.

Відкрити



Оптимізатор C

Оптимізуйте фрагменти коду C для кращої продуктивності.

Відкрити



Оптимізатор C++

Оптимізуйте фрагменти коду C++ для кращої продуктивності.

Відкрити

Рисунок 3.4 – Інтерфейс головної форми програмного забезпечення для вдосконалення вихідного коду

Форма введення програмного коду для подальшого вдосконалення наведена на рис. 3.5.

Оптимізуйте код на мові Python

```
iris = load_iris()
X, y = iris.data, iris.target
clf = tree.DecisionTreeClassifier()
clf = clf.fit(X, y)
tree.plot_tree(clf)
dot_data = tree.export_graphviz(clf, out_file=None)
graph = graphviz.Source(dot_data)
graph.render("iris", format="png")
Image.open("iris.png").show()
dot_data = tree.export_graphviz(clf, out_file=None,
feature_names=iris.feature_names,
class_names=iris.target_names,
filled=True, rounded=True,
special_characters=True)
graph = graphviz.Source(dot_data)
graph.render("iris", format="png")
Image.open("iris.png").show()
```

Оптимізуйте код

Рисунок 3.5 – Форма введення програмного коду для подальшого вдосконалення

Область пояснення виконаних вдосконалень у програмному кодї наведена на рис. 3.6.

Пояснення:

1. **Імпорт:** Код був оптимізований шляхом імпорту необхідних модулів на початку скрипта, що покращує читабельність і зменшує обсяг коду.
2. **Відображення зображення дерева рішень:** функція з модуля (Python Imaging Library) використовується для відкриття згенерованого файлу PNG і його відображення. `Image.open()` `PIL`

Оптимізації:

1. **Зменшена складність коду:** оптимізований код містить менше рядків коду та є більш лаконічним, що полегшує його читання та розуміння.
2. **Покращена читабельність:** код упорядковано, а імпорт розміщено на початку, що покращує загальну читабельність скрипту.
3. **Ефективне завантаження даних:** Функція `from` використовується для завантаження набору даних райдужної оболонки ока, що є поширеним та ефективним способом завантаження набору даних. `load_iris()` `sklearn.datasets`

Загалом, оптимізований код є більш лаконічним, ефективним і легшим для читання та розуміння, водночас досягаючи тієї ж функціональності, що й оригінальний код.

Рисунок 3.6 – Область пояснення виконаних вдосконалень у програмному коді

3.5 Висновки за розділом 3

Розроблено програмне забезпечення для вдосконалення вихідного коду.

Запропоновано структуру програмного забезпечення для вдосконалення вихідного коду. Визначено, що структура програмного забезпечення для вдосконалення вихідного коду організована таким чином, щоб забезпечити чіткий поділ обов’язків між серверною логікою, клієнтським інтерфейсом і спільними інструментами, а також надати можливість здійснювати автоматизований аналіз тексту програм високого рівня з метою пошуку і подальшого усунення проблемних рішень, що негативно впливають на продуктивність або стійкість роботи коду.

Описано функціонування програмного забезпечення для вдосконалення вихідного коду. Функціонування програмного забезпечення для вдосконалення вихідного коду подано за допомогою схеми як послідовний процес перетворення та інтерпретації, у якому задіяні кілька рівнів обробки. Показано, що при функціонуванні програмного забезпечення для вдосконалення вихідного коду послідовні етапи трансформації JSX-описів у JavaScript-файли поєднуються з роботою інтерпретатора та нативного механізму браузера, що забезпечує як оптимізовану обробку, так і мінімальні оновлення у візуальному середовищі.

Розроблено базу даних програмного забезпечення для вдосконалення вихідного коду, яка є основою для збереження всієї інформації, що виникає в процесі роботи із вдосконаленням вихідного коду. Вона виконує роль центрального сховища, де поєднуються як початкові дані у вигляді вихідних текстів програм, так і вторинні артефакти, отримані після запуску алгоритмів автоматичного аналізу. У базі даних фіксується стан коду, відображаються знайдені недоліки, рекомендації щодо покращення та зміни, які були зроблені розробниками після внесення виправлень. Схема бази даних програмного забезпечення для вдосконалення вихідного коду відображає концептуальні зв'язки між обраними елементами у відповідній предметній області. Основними сутностями бази даних обрано такі: проєкт (Project), Файл вихідного коду (SourceFile), Фрагмент коду (CodeSnippet), Аналіз (AnalysisRun), Виявлена проблема (Issue), Рекомендація (Recommendation), Зміна коду (CodeChange) та інші.

Виконано проєктування інтерфейсу взаємодії користувача з програмним забезпеченням для вдосконалення вихідного коду.

4 ЕКСПЛУАТАЦІЯ, ТЕСТУВАННЯ ТА ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ПРОГРАМИ

4.1 Призначення й умови застосування програми

Програма призначена для вдосконалення вихідного коду. Програма написана на мові JavaScript, що обґрунтовується універсальністю та підтримкою у більшості сучасних середовищ. Для створення програмного забезпечення для вдосконалення вихідного коду використано бібліотеку React, що поєднує продуктивність, гнучкість і зручність у використанні.

Програмне забезпечення для вдосконалення вихідного коду забезпечує виконання таких функцій:

- підтримка можливості працювати з текстовими наборами даних, що являють собою програмний код;
- підтримка можливості вдосконалення текстів з програмними кодами, написаними на різних мовах програмування;
- можливість авторизації користувачів;
- можливість виведення інформації щодо покращень у програмному коді;
- підтримка можливості користувачів копіювання покращеного програмного коду для подальшої роботи.

Для експлуатації програмного забезпечення для вдосконалення вихідного коду необхідно таке апаратне та програмне забезпечення:

- процесор із тактовою частотою не нижче 2,0 ГГц (рекомендовано багатоядерний, наприклад Intel Core i5 або AMD Ryzen 5 і вище);
- оперативна пам'ять обсягом не менше 4 ГБ;
- жорсткий диск або SSD з вільним місцем не менше 500 МБ для інсталяції та додатково від 2 ГБ для збереження даних і проєктів;
- відеоадаптер з підтримкою роздільної здатності не нижче 1280×720 (рекомендовано 1920×1080 для зручності роботи);

- доступ до мережі Інтернет для завантаження оновлень, бібліотек та залежностей;

- операційна система Windows 10/11 (64-bit), Linux (Ubuntu 20.04 і новіші) або macOS (версія 10.15 і вище) та веббраузер.

Функціональні характеристики розробленого програмного забезпечення для вдосконалення вихідного коду наведено у технічному завданні (додаток А). Фрагмент тексту програмного забезпечення для вдосконалення вихідного коду наведено у додатку Б.

4.2 Характеристики програми для вдосконалення вихідного коду

Програмне забезпечення для вдосконалення вихідного коду характеризується здатністю автоматизувати процеси аналізу та оптимізації програмних рішень. Його робота ґрунтується на використанні алгоритмів, які визначають неефективні ділянки коду, пропонують шляхи покращення структури та забезпечують підвищення продуктивності. Програмне забезпечення для вдосконалення вихідного коду підтримує роботу з мовами програмування. Забезпечується можливість візуалізації результатів, що сприяє наочності й зручності використання.

Програмне забезпечення для вдосконалення вихідного коду орієнтоване на стабільність, масштабованість і сумісність з різними системами.

4.3 Інструкція по експлуатації програми

4.3.1 Звернення до програми

Для запуску програмного забезпечення для вдосконалення вихідного коду необхідно перейти на відповідну вебсторінку, де розташовано застосунок.

4.3.2 Вхідні й вихідні дані

Вхідними даними до програмного забезпечення для вдосконалення вихідного коду є тексти з програмними модулями, що містять алгоритмічні конструкції, структурні елементи коду та коментарі розробника.

Вихідними даними програмного забезпечення для вдосконалення вихідного коду є оптимізовані версії програмних кодів, що відрізняються покращеною структурою, підвищеною ефективністю виконання та зменшеною кількістю помилок або дублювань.

4.3.3 Повідомлення

Користувач програмного забезпечення для вдосконалення вихідного коду може отримати такі повідомлення: сповіщення про виявлені помилки синтаксису, попередження щодо неефективних конструкцій, повідомлення про підтвердження успішного завершення процесу вдосконалення коду.

4.4 Виконання програмного забезпечення для вдосконалення вихідного коду

Після запуску програмного забезпечення для вдосконалення вихідного коду користувачу відображається головна сторінка (рис. 4.1). Тут користувачу пропонується обрати мову програмування, на якій написано програмний код для вдосконалення. Також існує можливість авторизації.

Після вибору мови програмування відображається відповідна сторінка (рис. 4.2), де в область коду необхідно вставити текст з кодом програми для вдосконалення.

Оберіть мову програмування



Оптимізатор Python

Оптимізуйте фрагменти коду Python для кращої продуктивності.

Відкрити



Оптимізатор JavaScript

Оптимізуйте фрагменти коду JavaScript для кращої продуктивності.

Відкрити



Оптимізатор C

Оптимізуйте фрагменти коду C для кращої продуктивності.

Відкрити



Оптимізатор C++

Оптимізуйте фрагменти коду C++ для кращої продуктивності.

Відкрити

Рисунок 4.1 – Інтерфейс головної сторінки програмного забезпечення для вдосконалення вихідного коду

Оптимізуйте код на мові Python

```
iris = load_iris()
X, y = iris.data, iris.target
clf = tree.DecisionTreeClassifier()
clf = clf.fit(X, y)
tree.plot_tree(clf)
dot_data = tree.export_graphviz(clf, out_file=None)
graph = graphviz.Source(dot_data)
graph.render("iris", format="png")
Image.open("iris.png").show()
dot_data = tree.export_graphviz(clf, out_file=None,
feature_names=iris.feature_names,
class_names=iris.target_names,
filled=True, rounded=True,
special_characters=True)
graph = graphviz.Source(dot_data)
graph.render("iris", format="png")
Image.open("iris.png").show()
```

Оптимізуйте код

Рисунок 4.2 – Введення програмного коду для подальшого вдосконалення

Після натискання кнопки «Оптимізуйте код» внизу цієї ж сторінки відображається область (рис. 4.3) з вдосконаленим кодом. Ця область є текстовою, тому користувач має можливість копіювати вдосконалений код та використовувати його у своїй подальшій діяльності.

Оптимізований код

Оптимізований код:

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier, plot_tree, export_graphviz
import graphviz
from PIL import Image

# Load the iris dataset
iris = load_iris()
X, y = iris.data, iris.target

# Train the decision tree classifier
clf = DecisionTreeClassifier()
clf.fit(X, y)

# Plot the decision tree
dot_data = export_graphviz(clf, out_file=None, feature_names=iris.feature_names,
                           class_names=iris.target_names, filled=True, rounded=True,
                           special_characters=True)
graph = graphviz.Source(dot_data)
graph.render("iris", format='png')

# Display the decision tree image
Image.open('iris.png').show()
```

Пояснення:

Рисунок 4.3 – Вдосконалений програмний код

Після області з вдосконаленим кодом відображається область з поясненнями виконаних оптимізацій (рис. 4.4), де наведено перелік виконаних модифікацій у коді.

Пояснення:

1. **Імпорт:** Код був оптимізований шляхом імпорту необхідних модулів на початку скрипта, що покращує читабельність і зменшує обсяг коду.
2. **Відображення зображення дерева рішень:** функція з модуля (Python Imaging Library) використовується для відкриття згенерованого файлу PNG і його відображення. `Image.open()` `PIL`

Оптимізації:

1. **Зменшена складність коду:** оптимізований код містить менше рядків коду та є більш лаконічним, що полегшує його читання та розуміння.
2. **Покращена читабельність:** код упорядковано, а імпорт розміщено на початку, що покращує загальну читабельність скрипту.
3. **Ефективне завантаження даних:** Функція `from` використовується для завантаження набору даних райдужної оболонки ока, що є поширеним та ефективним способом завантаження набору даних. `load_iris()` `sklearn.datasets`

Загалом, оптимізований код є більш лаконічним, ефективним і легшим для читання та розуміння, водночас досягаючи тієї ж функціональності, що й оригінальний код.

Рисунок 4.4 – Область пояснення виконаних вдосконалень у програмному коді

4.5 Тестування програмного забезпечення для вдосконалення вихідного коду

Виконано тестування програмного забезпечення для вдосконалення вихідного коду.

Виявлено, що програма для вдосконалення вихідного коду функціонує правильно та злагоджено. Вона реалізує всі функціональні вимоги та успішно виконує свою основну задачу підтримки процесу вдосконалення програмного коду.

Розроблене програмне забезпечення дозволяє забезпечувати автоматизацію процесів, пов'язаних з підтримкою вдосконалення програмного коду.

4.6 Висновки за розділом 4

Описано програмне забезпечення для вдосконалення вихідного коду. Виконано тестування розробленого програмного забезпечення для вдосконалення вихідного коду. Результати тестування програмного забезпечення показали, що розроблена програма дозволяє забезпечити автоматизацію процесів, пов'язаних з вдосконаленням програмного коду.

ВИСНОВКИ

В ході виконання дипломної кваліфікаційної роботи магістра було проаналізовано та досліджено процес вдосконалення вихідного програмного коду.

Визначено, що вдосконалення вихідного коду являє собою систематичний процес покращення його якості, структури та ефективності без зміни основної функціональності. Такий підхід дозволяє зробити програмний продукт більш зрозумілим, надійним і стійким до помилок, а також полегшує подальшу підтримку та розвиток. Методи вдосконалення вихідного коду охоплюють кілька напрямів. Рефакторинг дозволяє оптимізувати внутрішню структуру, роблячи її більш зрозумілою та логічною. Використання автоматизованих інструментів аналізу сприяє виявленню синтаксичних і логічних помилок, а також підвищує безпеку. Сукупність таких методів формує основу для створення надійного, оптимізованого та зручного у подальшому розвитку програмного забезпечення.

За результатами проведеного аналізу зроблено висновок, що у наш час існує досить багато програмних засобів для вдосконалення вихідного коду. Проте деякі програмні засоби для вдосконалення вихідного коду є орієнтованими на певні мови програмування чи конкретні середовища, що зменшує універсальність і робить їх менш ефективними у випадку багатоплатформних рішень. Іншим недоліком є ймовірність появи нових помилок у процесі рефакторингу або оптимізації, особливо якщо інструменти використовуються без належного тестування. Крім того, застосування таких програмних засобів може вимагати значних ресурсів, як обчислювальних, так і часових, що знижує ефективність при роботі з великими проектами. Також важливо враховувати, що деякі програмні засоби вдосконалення вихідного коду не завжди забезпечують гнучкість, оскільки їхні алгоритми ґрунтуються на загальних правилах і не враховують унікальні вимоги конкретного

програмного проєкту. Тому актуальною є розробка програмного забезпечення для вдосконалення вихідного коду.

Сформульовано функціональні вимоги до програмного забезпечення для вдосконалення вихідного коду.

Для реалізації програмного забезпечення для вдосконалення вихідного коду обрано мову програмування JavaScript, що обґрунтовується універсальністю та підтримкою у більшості сучасних середовищ. Гнучкість синтаксису та підтримка асинхронних операцій роблять JavaScript ефективним інструментом для роботи з потоками даних і зовнішніми ресурсами. Екосистема мови стрімко розвивається, а велика кількість додаткових бібліотек і фреймворків значно розширює спектр можливостей і скорочує час реалізації складних проєктів. Активна спільнота забезпечує постійне вдосконалення стандартів і появу нових рішень, що дозволяє швидко адаптувати мову до сучасних потреб розробників.

Для створення програмного забезпечення для вдосконалення вихідного коду обрано бібліотеку React, що поєднує продуктивність, гнучкість і зручність у використанні. Компонентна архітектура React дозволяє створювати інтерфейси, що легко масштабуються й підтримуються навіть у складних проєктах. Крім того, забезпечується ефективне оновлення елементів, що дає можливість працювати з великими обсягами даних без зниження швидкодії. Розвинена екосистема бібліотек і численні готові рішення дозволяють швидко додавати необхідний функціонал, наприклад інтеграцію з редакторами коду, підсвічування синтаксису чи інструментами візуалізації.

Новизна роботи полягає у тому, що запропоновано метод динамічного маркування коду, який полягає у поєднанні коментарів з метаданими та автоматизованого оновлення цих даних під час виконання програм, що дозволяє підвищити прозорість програмного коду, спростити його оптимізацію та забезпечити гнучку підтримку на всіх етапах життєвого циклу.

Практична цінність роботи полягає у тому, що розроблено програмне забезпечення для вдосконалення вихідного коду.

Запропоновано структуру програмного забезпечення для вдосконалення вихідного коду. Визначено, що структура програмного забезпечення для вдосконалення вихідного коду організована таким чином, щоб забезпечити чіткий поділ обов'язків між серверною логікою, клієнтським інтерфейсом і спільними інструментами, а також надати можливість здійснювати автоматизований аналіз тексту програм високого рівня з метою пошуку і подальшого усунення проблемних рішень, що негативно впливають на продуктивність або стійкість роботи коду.

Описано функціонування програмного забезпечення для вдосконалення вихідного коду. Функціонування програмного забезпечення для вдосконалення вихідного коду подано за допомогою схеми як послідовний процес перетворення та інтерпретації, у якому задіяні кілька рівнів обробки. Показано, що при функціонуванні програмного забезпечення для вдосконалення вихідного коду послідовні етапи трансформації JSX-описів у JavaScript-файли поєднуються з роботою інтерпретатора та нативного механізму браузера, що забезпечує як оптимізовану обробку, так і мінімальні оновлення у візуальному середовищі.

Розроблено базу даних програмного забезпечення для вдосконалення вихідного коду, яка є основою для збереження всієї інформації, що виникає в процесі роботи із вдосконаленням вихідного коду. Вона виконує роль центрального сховища, де поєднуються як початкові дані у вигляді вихідних текстів програм, так і вторинні артефакти, отримані після запуску алгоритмів автоматичного аналізу. У базі даних фіксується стан коду, відображаються знайдені недоліки, рекомендації щодо покращення та зміни, які були зроблені розробниками після внесення виправлень. Схема бази даних програмного забезпечення для вдосконалення вихідного коду відображає концептуальні зв'язки між обраними елементами у відповідній предметній області. Основними сутностями бази даних обрано такі: проєкт (Project), Файл вихідного коду (SourceFile), Фрагмент коду (CodeSnippet), Аналіз

(AnalysisRun), Виявлена проблема (Issue), Рекомендація (Recommendation), Зміна коду (CodeChange) та інші.

Виконано проєктування інтерфейсу взаємодії користувача з програмним забезпеченням для вдосконалення вихідного коду.

Виконано тестування розробленого програмного забезпечення для вдосконалення вихідного коду. Результати тестування програмного забезпечення показали, що розроблена програма дозволяє забезпечити автоматизацію процесів, пов'язаних з вдосконаленням програмного коду.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Why is Optimization of Source Codes Important [Electronic resource]. – Access mode: <https://www.hostragons.com/en/blog/optimization-of-source-codes/>.
2. Best Practices for Software Source Code Management: Proven Strategies for Success [Electronic resource]. – Access mode: <https://blog.kodezi.com/best-practices-for-software-source-code-management-proven-strategies-for-success/>.
3. Source Code Management, Tools, and Best Practices [Electronic resource]. – Access mode: <https://www.turing.com/blog/source-code-management-tools-best-practices>.
4. Code Optimization in Compiler Design [Electronic resource]. – Access mode: <https://www.geeksforgeeks.org/compiler-design/code-optimization-in-compiler-design/>.
5. Code Optimization Strategies for Faster Software [Electronic resource]. – Access mode: <https://www.index.dev/blog/code-optimization-strategies>.
6. Steps to Improve Code Quality [Electronic resource]. – Access mode: <https://stackify.com/7-steps-to-improve-code-quality/>.
7. Top 10 Automated Code Review Tools For Developers [Electronic resource]. – Access mode: <https://reliasoftware.com/blog/automated-code-review-tools>.
8. List of the Top 25 Source Code Management Software [Electronic resource]. – Access mode: <https://topbusinesssoftware.com/categories/source-code-management/>.
9. Best Automated Code Review Tools for Developers [Electronic resource]. – Access mode: <https://www.qodo.ai/blog/automated-code-review/>.
10. Best Code Quality Tools For Error Free Coding [Electronic resource]. – Access mode: <https://www.softwaretestinghelp.com/code-quality-tools/>.
11. Best Code Analysis Tools [Electronic resource]. – Access mode: <https://thectoclub.com/tools/best-code-analysis-tools/>.

12. Codacy [Electronic resource]. – Access mode: <https://www.codacy.com/>.
13. Korbit [Electronic resource]. – Access mode: <https://www.korbit.ai/>.
14. Snyk Code [Electronic resource]. – Access mode: <https://snyk.io/product/snyk-code/>.
15. JavaScript Operators [Electronic resource]. – Access mode: https://www.w3schools.com/js/js_operators.asp.
16. Java vs. JavaScript: What's the Difference [Electronic resource]. – Access mode: <https://www.coursera.org/articles/java-vs-javascript>.
17. React Quick Start [Electronic resource]. – Access mode: <https://react.dev/learn>.
18. React Introduction [Electronic resource]. – Access mode: <https://www.geeksforgeeks.org/reactjs/reactjs-introduction/>.

ДОДАТОК А
Технічне завдання

Вступ

Програмне забезпечення може використовуватися для вдосконалення вихідного коду.

A.1 Підстава для розробки

Підставою для розробки є завдання на дипломну кваліфікаційну роботу на тему «Дослідження та програмна реалізація методів вдосконалення вихідного коду», затверджене наказом Національного університету «Запорізька політехніка» № 447 від 30 вересня 2025 р.

A.2 Призначення розробки

Програмний продукт призначений для вдосконалення вихідного коду.

A.3 Основні вимоги до програми, що розробляється

A.3.1 Вимоги до функціональних характеристик

Програмне забезпечення для вдосконалення вихідного коду забезпечує виконання таких функцій:

- підтримка можливості працювати з текстовими наборами даних, що являють собою програмний код;
- підтримка можливості вдосконалення текстів з програмними кодами, написаними на різних мовах програмування;
- можливість авторизації користувачів;
- можливість виведення інформації щодо покращень у програмному коді;
- підтримка можливості користувачів копіювання покращеного програмного коду для подальшої роботи.

А.3.2 Вимоги до інтерфейсу програми

Інтерфейс програмного забезпечення для вдосконалення вихідного коду повинен бути зручним для користувачів. Повинна бути забезпечена можливість роботи в візуальному режимі.

А.3.3 Вимоги до надійності

Програмне забезпечення для вдосконалення вихідного коду повинно забезпечити надійне функціонування.

А.3.4 Умови експлуатації

Для експлуатації програмного забезпечення для вдосконалення вихідного коду необхідна наявність персонального комп'ютера.

А.3.5 Вимоги до складу та параметрів технічний засобів

Для експлуатації програмного забезпечення для вдосконалення вихідного коду необхідно таке апаратне та програмне забезпечення:

- процесор із тактовою частотою не нижче 2,0 ГГц (рекомендовано багатоядерний, наприклад Intel Core i5 або AMD Ryzen 5 і вище);
- оперативна пам'ять обсягом не менше 4 ГБ;
- жорсткий диск або SSD з вільним місцем не менше 500 МБ для інсталяції та додатково від 2 ГБ для збереження даних і проєктів;
- відеоадаптер з підтримкою роздільної здатності не нижче 1280×720 (рекомендовано 1920×1080 для зручності роботи);

- доступ до мережі Інтернет для завантаження оновлень, бібліотек та залежностей;
- операційна система Windows 10/11 (64-bit), Linux (Ubuntu 20.04 і новіші) або macOS (версія 10.15 і вище) та веббраузер.

A.3.6 Вимоги до маркування і пакування

Програма для вдосконалення вихідного коду може бути записана на будь-якому носії інформації.

На пакуванні повинна бути назва програми – «Програмне забезпечення для вдосконалення вихідного коду».

A.4 Вхідні дані до роботи

Вхідними даними до програмного забезпечення для вдосконалення вихідного коду є тексти з програмними модулями, що містять алгоритмічні конструкції, структурні елементи коду та коментарі розробника.

Вихідними даними програмного забезпечення для вдосконалення вихідного коду є оптимізовані версії програмних кодів, що відрізняються покращеною структурою, підвищеною ефективністю виконання та зменшеною кількістю помилок або дублювань.

ДОДАТОК Б
Фрагмент тексту програми

```

export function rmvWhtSpc(cd) {
  if (!cd || typeof cd !== "string") {
    return { sts: "flr", msg: "nvldid npt" };
  }
  let lnArr = cd.split("\n");
  let nwArr = [];
  for (let i = 0; i < lnArr.length; i++) {
    let l = lnArr[i].trim();
    if (l.length > 0) {
      nwArr.push(l);
    }
  }
  let rsl = nwArr.join(" ");
  return { sts: "ok", dt: rsl };
}

export function mnfyVrnm(cd) {
  if (!cd || typeof cd !== "string") {
    return { sts: "flr", msg: "nvldid npt" };
  }
  let mtch = cd.match(/let\s+([a-zA-Z_][a-zA-Z0-9_]*)/g) || [];
  let replcd = cd;
  for (let i = 0; i < mtch.length; i++) {
    let vnm = mtch[i].split(" ")[1];
    let shrt = "v" + i;
    let rgx = new RegExp("\\b" + vnm + "\\b", "g");
    replcd = replcd.replace(rgx, shrt);
  }
  return { sts: "ok", dt: replcd };
}

export function clnCmnts(cd) {
  if (!cd || typeof cd !== "string") {
    return { sts: "flr", msg: "nvldid npt" };
  }
  let stp1 = cd.replace(/\\/\\*[\s\S]*?\\*\\/g, "");
  let stp2 = stp1.replace(/\\/\\.*/g, "");
  let lnArr = stp2.split("\n");
  let nwArr = [];
  for (let i = 0; i < lnArr.length; i++) {
    if (lnArr[i].trim() !== "") {
      nwArr.push(lnArr[i]);
    }
  }
  return { sts: "ok", dt: nwArr.join("\n") };
}

```

```

export function cmprCd(cd) {
  if (!cd || typeof cd !== "string") {
    return { sts: "flr", msg: "nvlid npt" };
  }
  let bfrSz = cd.length;
  let cmp = cd.replace(/\s+/g, " ");
  let aftrSz = cmp.length;
  let rdc = bfrSz - aftrSz;
  return { sts: "ok", dt: cmp, rdc: rdc, bfr: bfrSz, aftr:
aftrSz };
}

```

```

export async function cnntDB() {
  let cnntn;
  try {
    cnntn = await indexedDB.open("optmzDB", 1);
    return cnntn;
  } catch (err) {
    let rtrn = { sts: "flr", msg: "cnntn fld" };
    return rtrn;
  }
}

```

```

export async function svRcd(nm, dt) {
  let cnntn = await cnntDB();
  return new Promise((rslv, rjct) => {
    let trns = cnntn.result.transaction(["tbl"],
"readwrite");
    let str = trns.objectStore("tbl");
    let rqst = str.put({ id: nm, val: dt });
    rqst.onsuccess = () => {
      rslv({ sts: "ok", msg: "rcd svd" });
    };
    rqst.onerror = () => {
      rjct({ sts: "flr", msg: "fld t sv rcd" });
    };
  });
}

```

```

export async function ldRcd(nm) {
  let cnntn = await cnntDB();
  return new Promise((rslv, rjct) => {
    let trns = cnntn.result.transaction(["tbl"], "readonly");
    let str = trns.objectStore("tbl");
    let rqst = str.get(nm);
    rqst.onsuccess = () => {
      if (rqst.result) {

```

```

        rslv(rqst.result);
    } else {
        rjct({ sts: "flr", msg: "rcd nt fnd" });
    }
};
rqst.onerror = () => {
    rjct({ sts: "flr", msg: "fld t ld rcd" });
};
});
}

export async function dlRcd(nm) {
    let cnntn = await cnntDB();
    return new Promise((rslv, rjct) => {
        let trns = cnntn.result.transaction(["tbl"],
"readwrite");
        let str = trns.objectStore("tbl");
        let rqst = str.delete(nm);
        rqst.onsuccess = () => {
            rslv({ sts: "ok", msg: "rcd dlt" });
        };
        rqst.onerror = () => {
            rjct({ sts: "flr", msg: "fld t dlt rcd" });
        };
    });
}

export function clctMtrx(cd) {
    let ln = cd.split("\n").length;
    let chr = cd.length;
    let fnc = (cd.match(/function/g) || []).length;
    let vrb = (cd.match(/let|const|var/g) || []).length;
    return { ln, chr, fnc, vrb };
}

export function gnrRpt(dt) {
    let ttl = `Rpt Fr Cde`;
    let bdy = `Ln: ${dt.ln}, Chr: ${dt.chr}, Fnc: ${dt.fnc}, Vrb:
${dt.vrb}`;
    let rpt = `${ttl}\n${bdy}`;
    return rpt;
}

export function sndRpt(rpt) {
    let prgrph = document.createElement("p");
    prgrph.textContent = rpt;
    document.body.appendChild(prgrph);
}

```

```

export function vrfCdQlty(cd) {
  let scr = 0;
  if (cd.includes("===")) scr += 10;
  if (cd.includes("let")) scr += 5;
  if (cd.length > 200) scr += 5;
  if (cd.includes("=>")) scr += 10;
  return { scr, sts: scr > 20 ? "gd" : "bd" };
}

export async function synSrv(cd) {
  try {
    let rsp = await fetch("/upld", {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify({ cd })
    });
    let dt = await rsp.json();
    return dt;
  } catch (err) {
    return { sts: "flr", msg: "srv nt rchbl" };
  }
}

export async function ldSrv(id) {
  try {
    let rsp = await fetch(`/ld/${id}`);
    let dt = await rsp.json();
    return dt;
  } catch (err) {
    return { sts: "flr", msg: "srv ld fld" };
  }
}

export async function dltSrv(id) {
  try {
    let rsp = await fetch(`/dlt/${id}`, { method: "DELETE"
  });
    let dt = await rsp.json();
    return dt;
  } catch (err) {
    return { sts: "flr", msg: "srv dlt fld" };
  }
}

export async function lstSrv() {
  try {
    let rsp = await fetch(`/lst`);

```

```

        let dt = await rsp.json();
        return dt;
    } catch (err) {
        return { sts: "flr", msg: "srv lst fld" };
    }
}

```

```

export function shwMsg(msg, typ = "info") {
    let dv = document.createElement("div");
    dv.textContent = msg;
    dv.className = `msg ${typ}`;
    dv.style.padding = "10px";
    dv.style.margin = "5px";
    dv.style.border = "1px solid #444";
    dv.style.borderRadius = "5px";
    dv.style.backgroundColor = typ === "err" ? "#fdd" : "#dfd";
    document.body.appendChild(dv);
    setTimeout(() => {
        dv.remove();
    }, 4000);
}

```

```

export function shwRprt(rpt) {
    let ct = document.getElementById("rptCt");
    if (!ct) {
        ct = document.createElement("div");
        ct.id = "rptCt";
        document.body.appendChild(ct);
    }
    let prg = document.createElement("pre");
    prg.style.border = "1px solid #ccc";
    prg.style.padding = "10px";
    prg.style.backgroundColor = "#fafafa";
    prg.style.whiteSpace = "pre-wrap";
    prg.textContent = rpt;
    ct.appendChild(prg);
}

```

```

export function shwFrm() {
    let frm = document.createElement("form");
    frm.style.display = "flex";
    frm.style.flexDirection = "column";
    frm.style.gap = "10px";
    frm.style.margin = "20px";
    let txt = document.createElement("textarea");
    txt.rows = 10;
    txt.cols = 60;
}

```

```

txt.placeholder = "pst cde hre...";
frm.appendChild(txt);
let btn = document.createElement("button");
btn.type = "button";
btn.textContent = "Optmz Cde";
btn.onclick = () => {
    let cd = txt.value;
    if (cd.trim().length === 0) {
        shwMsg("n cd prvdd", "err");
        return;
    }
    let res = cd.toUpperCase();
    shwRprt(res);
    shwMsg("cde prcssd", "info");
};
frm.appendChild(btn);
document.body.appendChild(frm);
}

export function shwTbl(dt) {
    let tbl = document.createElement("table");
    tbl.style.borderCollapse = "collapse";
    tbl.style.margin = "20px";
    let hd = document.createElement("tr");
    for (let k in dt[0]) {
        let th = document.createElement("th");
        th.style.border = "1px solid #000";
        th.style.padding = "5px";
        th.textContent = k;
        hd.appendChild(th);
    }
    tbl.appendChild(hd);
    for (let i = 0; i < dt.length; i++) {
        let rw = document.createElement("tr");
        for (let k in dt[i]) {
            let td = document.createElement("td");
            td.style.border = "1px solid #000";
            td.style.padding = "5px";
            td.textContent = dt[i][k];
            rw.appendChild(td);
        }
        tbl.appendChild(rw);
    }
    document.body.appendChild(tbl);
}

export function ldCdFrmFl(fl) {

```

```

return new Promise((rslv, rjct) => {
  let rdr = new FileReader();
  rdr.onload = (ev) => {
    let cnt = ev.target.result;
    if (typeof cnt !== "string" || cnt.length === 0) {
      rjct({ sts: "flr", msg: "fl emty" });
      return;
    }
    let ln = cnt.split("\n").length;
    let chrs = cnt.length;
    rslv({ sts: "ok", dt: cnt, lns: ln, sz: chrs });
  };
  rdr.onerror = () => {
    rjct({ sts: "flr", msg: "cnt rd fl" });
  };
  rdr.readAsText(fl);
});
}

export function svCdTF1(nm, cd) {
  if (!cd || typeof cd !== "string" || cd.trim().length === 0)
  {
    return { sts: "flr", msg: "nvldid cde" };
  }
  let blb = new Blob([cd], { type: "text/plain" });
  let lnk = document.createElement("a");
  lnk.href = URL.createObjectURL(blb);
  lnk.download = nm || "cde.txt";
  document.body.appendChild(lnk);
  lnk.click();
  setTimeout(() => {
    document.body.removeChild(lnk);
    URL.revokeObjectURL(lnk.href);
  }, 1000);
  return { sts: "ok", msg: "fl svd" };
}

let lgSt = [];

export function lgEvnt(tp, dt) {
  let tms = new Date().toISOString();
  let ev = { tp, dt, tms };
  lgSt.push(ev);
  if (lgSt.length > 500) {
    lgSt.shift();
  }
  return ev;
}

```

```

export function gtLg() {
  return lgSt.map(e => `${e.tms} [${e.tp}]
  ${JSON.stringify(e.dt)}\`)).join("\n");
}

export function clrLg() {
  lgSt = [];
  return { sts: "ok", msg: "lg clr" };
}

export function msrPrfrmnc(fn, itrs = 100) {
  if (typeof fn !== "function") {
    return { sts: "flr", msg: "nvlid fn" };
  }
  let strt = performance.now();
  for (let i = 0; i < itrs; i++) {
    try {
      fn();
    } catch (e) {
      return { sts: "flr", msg: "fn fld", err: e };
    }
  }
  let nd = performance.now();
  let dur = nd - strt;
  let avg = dur / itrs;
  return { sts: "ok", ttl: dur, avg: avg, itrs: itrs };
}

export function cmpFn(fn1, fn2, itrs = 100) {
  let r1 = msrPrfrmnc(fn1, itrs);
  let r2 = msrPrfrmnc(fn2, itrs);
  if (r1.sts !== "ok" || r2.sts !== "ok") {
    return { sts: "flr", msg: "cmp fld" };
  }
  let bst = r1.avg < r2.avg ? "fn1" : "fn2";
  return { sts: "ok", bst, r1, r2 };
}

export function chckUsrPrmsn(usr, actn) {
  let rslt = false
  if (usr && usr.rl && actn) {
    if (usr.rl === "adm") {
      rslt = true
    } else {
      if (actn === "rd") {
        rslt = true
      }
    }
  }
}

```

```

    } else if (actn === "wrt") {
        if (usr.prmsn && usr.prmsn.includes("wrt")) {
            rslt = true
        }
    } else if (actn === "dl") {
        if (usr.prmsn && usr.prmsn.includes("dl")) {
            rslt = true
        }
    }
}
}
return rslt
}

```

```

export function hshDt(str) {
    let hsh = 0
    if (str && str.length > 0) {
        for (let i = 0; i < str.length; i++) {
            let chr = str.charCodeAt(i)
            hsh = ((hsh << 5) - hsh) + chr
            hsh |= 0
        }
    }
    return hsh.toString()
}

```

```

export function encrtTxt(txt, ky) {
    let rslt = ""
    if (txt && ky) {
        for (let i = 0; i < txt.length; i++) {
            let c = txt.charCodeAt(i)
            let k = ky.charCodeAt(i % ky.length)
            let x = c ^ k
            rslt += String.fromCharCode(x)
        }
    }
    return rslt
}

```

```

export function dcrptTxt(txt, ky) {
    let rslt = ""
    if (txt && ky) {
        for (let i = 0; i < txt.length; i++) {
            let c = txt.charCodeAt(i)
            let k = ky.charCodeAt(i % ky.length)
            let x = c ^ k
            rslt += String.fromCharCode(x)
        }
    }
}

```

```

    }
    return rslt
}

export function vldtnInpt(inpt) {
  let rslt = true
  if (typeof inpt === "string") {
    if (inpt.includes("<") || inpt.includes(">") ||
inpt.includes("script")) {
      rslt = false
    }
    if (inpt.includes("'") || inpt.includes("\") ||
inpt.includes(";")) {
      rslt = false
    }
  } else if (typeof inpt === "object") {
    for (let ky in inpt) {
      if (inpt.hasOwnProperty(ky)) {
        if (!vldtnInpt(inpt[ky])) {
          rslt = false
          break
        }
      }
    }
  }
  return rslt
}

export function gnrtsnTkn(usr) {
  let tkn = ""
  if (usr && usr.id) {
    let bss = usr.id.toString() + Date.now().toString()
    for (let i = 0; i < bss.length; i++) {
      let c = bss.charCodeAt(i)
      let v = (c * (i + 7)) % 97
      tkn += v.toString(16)
    }
  }
  return tkn
}

export function lgnAttmpt(usrm, psswr, db) {
  let rslt = { scs: false, msg: "" }
  if (!usrm || !psswr) {
    rslt.msg = "nvld npt"
    return rslt
  }
  let usr = db.find(u => u.usrm === usrm)

```

```
if (!usr) {  
    rslt.msg = "користувача немає"  
    return rslt  
}  
let hshpss = hshDt(psswrd)  
if (usr.psswrd === hshpss) {  
    rslt.scs = true  
    rslt.msg = "логін"  
} else {  
    rslt.msg = "невірний пароль"  
}  
return rslt  
}
```

ДОДАТОК В
Слайди презентації

Дослідження та програмна реалізація методів вдосконалення вихідного коду

Виконав

студ. КНТ-124м
ШАМАНСЬКИЙ Роман Володимирович

Керівник

кандидат технічних наук, доцент
ЗАЙКО Тетяна Анатоліївна

Рисунок В.1 – Слайд 1

Об'єкт, предмет та мета роботи

Об'єкт дослідження – процес вдосконалення вихідного програмного коду.

Предмет дослідження – програмні засоби для вдосконалення вихідного коду.

Метою роботи є дослідження та розробка програмного забезпечення для вдосконалення вихідного коду.

Рисунок В.2 – Слайд 2

Завдання роботи

Для досягнення поставленої мети у кваліфікаційній роботі магістра необхідно розв'язати такі задачі:

- виконати аналіз предметної області та програмних засобів для вдосконалення вихідного коду;
- розробити метод вдосконалення вихідного коду;
- здійснити проектування програмного забезпечення для вдосконалення вихідного коду;
- створити програмне забезпечення для вдосконалення вихідного коду;
- виконати тестування розробленого програмного забезпечення для вдосконалення вихідного коду.

Рисунок В.3 – Слайд 3

Порівняння програмного забезпечення для вдосконалення вихідного коду

Критерій порівняння	Codacy	Korbit	Snyk Code
Виявлення вразливостей у коді	+—	+	+
Підтримка різних мов програмування	+	+—	+—
Оцінка ризиків для знайдених проблем	—	+—	+
Інтуїтивність інтерфейсу	+	+—	+
Автоматичні виправлення	—	+—	+
Гнучкість налаштувань	+	+—	+

Рисунок В.4 – Слайд 4



Функціональні вимоги

При розробці програмного забезпечення для вдосконалення вихідного коду необхідно забезпечити такі функціональні вимоги:

- підтримка можливості працювати з текстовими наборами даних, що являють собою програмний код;
- підтримка можливості вдосконалення текстів з програмними кодами, написаними на різних мовах програмування;
- можливість авторизації користувачів;
- можливість виведення інформації щодо покращень у програмному коді;
- підтримка можливості користувачів копіювання покращеного програмного коду для подальшої роботи.

Рисунок В.5 – Слайд 5

Вибір мови програмування

Критерій порівняння мов програмування	Мова програмування		
	JavaScript	Python	C#
Поширеність у вебсередовищах	+	+-	+-
Розвинена екосистема бібліотек	+	+	+-
Асинхронне програмування	+	+-	+-
Зручність для створення ПЗ для вдосконалення вихідного коду	+	-	-
Швидкість розробки	+	+	+-

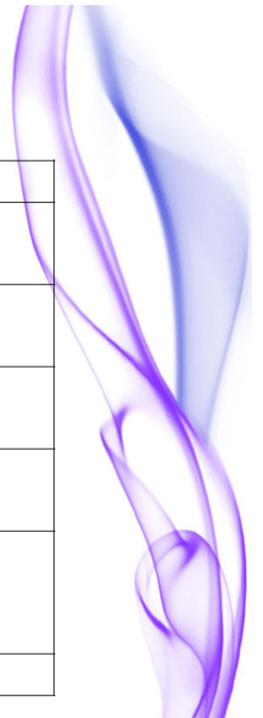


Рисунок В.6 – Слайд 6

Вибір бібліотеки для створення інтерфейсу

Критерій порівняння бібліотек для створення інтерфейсу	Бібліотеки для створення інтерфейсу		
	React	Angular	Svelte
Продуктивність інтерфейсу	+	+–	+
Гнучкість та масштабованість	+	+	+–
Розвинена екосистема та бібліотеки	+	+	–
Популярність та активність спільноти	+	+	+–
Підтримка інтеграції з інструментами для роботи з кодом	+	+–	–

Рисунок В.7 – Слайд 7

Метод вдосконалення вихідного коду



Рисунок В.8 – Слайд 8

Структура програмного забезпечення для вдосконалення вихідного коду

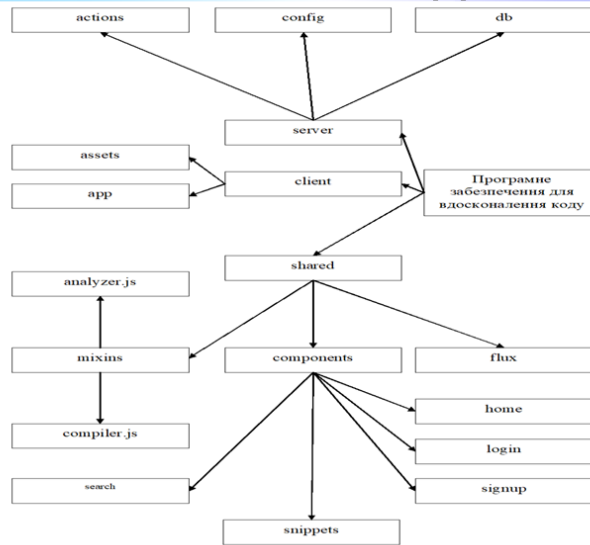


Рисунок В.9 – Слайд 9

Схема функціонування ПЗ для вдосконалення вихідного коду

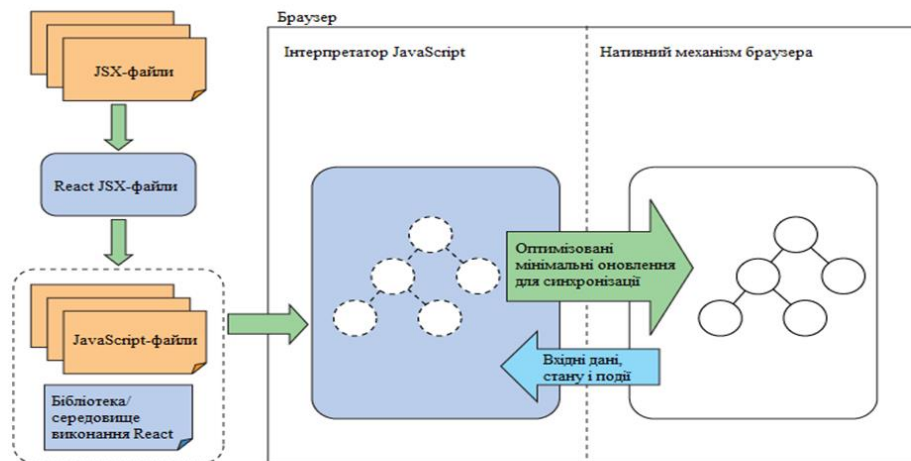


Рисунок В.10 – Слайд 10

Схема бази даних ПЗ для вдосконалення вихідного коду

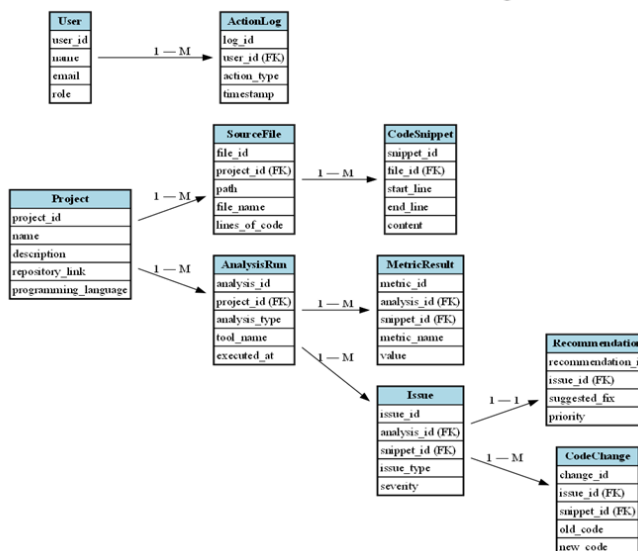


Рисунок В.11 – Слайд 11

Експлуатація програми. Вибір мови програмування для оптимізації коду

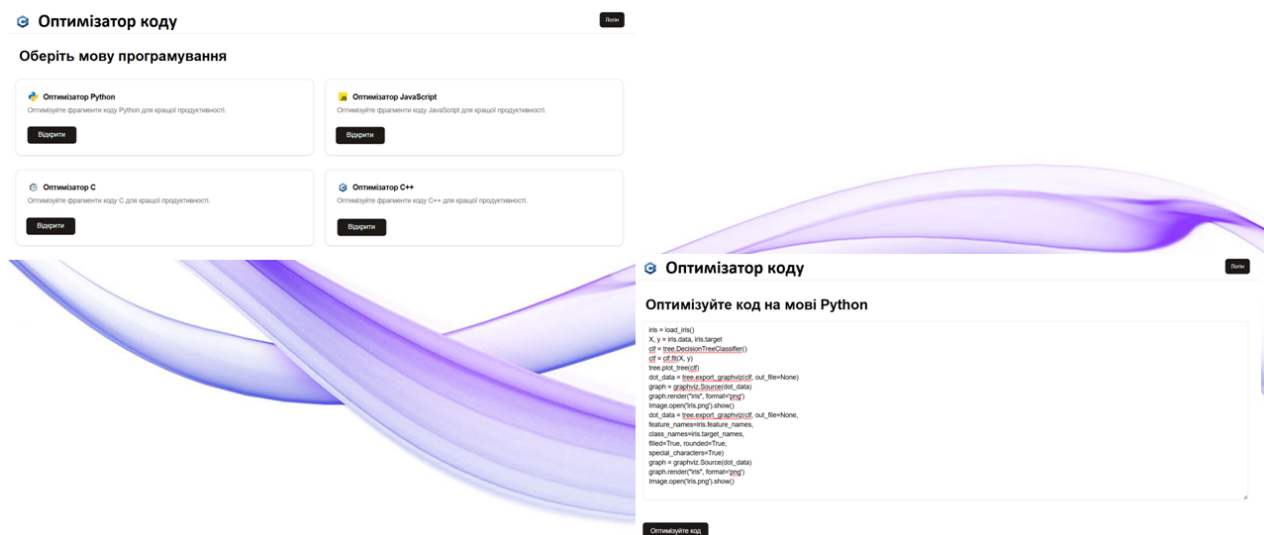


Рисунок В.12 – Слайд 12

Експлуатація програми. Оптимізований код. Пояснення щодо вдосконалень

Оптимізований код

```

Оптимізований код:
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier, plot_tree, export_graphviz
import graphviz
from IPython import Image

# load the iris dataset
iris = load_iris()
X, y = iris.data, iris.target

# train the decision tree classifier
clf = DecisionTreeClassifier()
clf.fit(X, y)

# Plot the decision tree
dot_data = export_graphviz(clf, out_file=None, feature_names=iris.feature_names,
                           class_names=iris.target_names, filled=True, rounded=True,
                           special_characters=True)
graph = graphviz.Source(dot_data)
graph.render("iris", format="png")

# Display the decision tree image
Image.open("iris.png").show()

```

Пояснення:

Пояснення:

1. **Імпорт:** Код був оптимізований шляхом необхідних імпорту на початку скрипта, що покращує читабельність і зменшує обсяг коду.
2. **Візуалізація зображення дерева рішень:** функція з модуля (Python Imaging Library) використовується для відкриття згенерованого файлу PNG (його відображення: `Image.open()`).

Оптимізації:

1. **Зменшена складність коду:** оптимізований код містить менше рядків коду та є більш лаконічним, що полегшує його читання та розуміння.
2. **Покращена читабельність:** код упорядкований, а імпорти розташовані на початку, що покращує загальну читабельність скрипта.
3. **Ефективне завантаження даних:** функція `load_iris()` використовується для завантаження набору даних іриси, що є поширеним та ефективним способом завантаження набору даних. `load_iris()` `sklearn.datasets`

Загалом, оптимізований код є більш лаконічним, ефективним і легшим для читання та розуміння, водночас досягаючи тієї ж функціональності, що й оригінальний код.

Рисунок В.13 – Слайд 13

Висновки

В ході виконання роботи було проаналізовано та досліджено процес вдосконалення вихідного програмного коду.

Новизна роботи полягає у тому, що запропоновано метод динамічного маркування коду, який полягає у поєднанні коментарів з метаданими та автоматизованого оновлення цих даних під час виконання програм, що дозволяє підвищити прозорість програмного коду, спростити його оптимізацію та забезпечити гнучку підтримку на всіх етапах життєвого циклу.

Практична цінність роботи полягає у тому, що розроблено програмне забезпечення для вдосконалення вихідного коду.

Усі завдання роботи повністю виконано.

Рисунок В.14 – Слайд 14