НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

«На правах рукопису»
УДК 004.41

До захисту допущено:
В. О. Завідувача кафедри
_____ Михайло Новотарський
« » _____ 2024 р

Магістерська дисертація

на здобуття ступеня магістра

за освітньо-професійною програмою «Комп'ютерні системи та мережі» зі спеціальності 123 «Комп'ютерна інженерія»

на тему: «Система аналізу та оптимізації програмного коду»

Виконав (-ла):
студент (-ка) ІІ курсу, групи ІО-32мп
Денисенко Михайло Олексійович _____

Керівник:
проф. каф. ОТ, д.т.н., професор,
Кулаков Юрій Олексійович _____

Консультант з нормоконтролю:
асистент каф. ОТ, к.т.н.
Кулаков Олексій Юрійович _____

Рецензент:
к.т.н, доцент кафедри ІСТ
Шимкович Володимир Миколайович _____

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2024 року

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет/ННІ Факультет інформатики та обчислювальної техніки
(повна назва)

Кафедра Кафедра обчислювальної техніки
(повна назва)

Рівень вищої освіти – другий (магістерський)

Спеціальність, спеціалізація 123 «Комп'ютерна інженерія»

Освітньо-професійна програма «Комп'ютерні системи та мережі»

ЗАТВЕРДЖУЮ

В. О. Завідувача кафедри

_____ Михайло Новотарський

«__» _____ 2024 р

ЗАВДАННЯ
на магістерську дисертацію студенту

Денисенку Михайлу Олексійовичу
(прізвище, ім'я, по-батькові)

1. Тема дисертації «Система аналізу та оптимізації програмного коду»

науковий керівник дисертації Кулаков Юрій Олексійович, д.т.н., проф. ,
(прізвище, ім'я, по-батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від « 08 » листопада 2024 р. № 5016с

2. Термін подання студентом дисертації:

3. Об'єкт дослідження: Обробка програмного коду

4. Вихідні дані:

5. Перелік завдань, які потрібно розробити: огляд існуючих рішень, опис предметної області, вибір технологій для реалізації поставленого завдання, розробка програмного забезпечення

6. Орієнтовний перелік графічного (ілюстративного) матеріалу: загальнотематичні ілюстрації, структурна схема роботи застосунку

7. Орієнтовний перелік публікацій:

8. Консультанти розділів дисертації:

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
нормоконтроль	асист. Кулаков О. Ю.		

9. Дата видачі завдання _____

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської дисертації	Примітка
1	Огляд існуючих рішень	05.09.2024 - 11.09.2024	
2	Планування розробки	12.09.2024 - 18.09.2024	
3	Вибір технологій для розробки	18.09.2024 - 21.09.2024	
4	Реалізація системи	22.09.2024 - 30.10.2024	
5	Тестування системи	31.10.2024 - 05.11.2024	
6	Розрахунки стартапу	06.11.2024 - 12.11.2024	
7	Оформлення документації	13.11.2024 - 25.11.2024	

Студент _____ Михайло Денисенко
(підпис)

Науковий керівник _____ Юрій Кулаков
(підпис)

РЕФЕРАТ

Робота складається зі вступу та чотирьох розділів. Загальний обсяг роботи: 105 аркушів основного тексту, 15 рисунків та 23 таблиць. У процесі підготовки використано 19 джерел.

Актуальність проблеми. У сучасних умовах розробка програмного забезпечення вимагає постійного вдосконалення процесів аналізу та оптимізації коду. Автоматизовані інструменти, такі як лінтери та форматувальники, є ключовими для зменшення кількості помилок, покращення продуктивності та забезпечення якості програмних систем.

Мета та завдання дослідження. Метою роботи є розробка системи аналізу та оптимізації програмного коду для спрощення його підтримки та підвищення якості. Завдання дослідження включають огляд існуючих рішень, вибір відповідних технологій, реалізацію системи та її тестування.

Об’єкт дослідження. Процеси аналізу та оптимізації програмного коду.

Предмет дослідження. Система, яка дозволяє адаптувати інструменти управління кодом для зменшення кількості помилок та покращення якості програмного забезпечення.

Практична цінність. Розроблена система забезпечує зручні інструменти для аналізу коду, які можуть бути інтегровані як розширення для Visual Studio Code. Це сприяє зниженню витрат на розробку та підвищенню якості кінцевого продукту.

Ключові слова: аналіз коду, оптимізація, лінтери, автоматизація.

ABSTRACT

The work consists of an introduction and four chapters. The total volume of the work: 105 pages of main text, 15 figures, and 23 tables. 19 sources were used during the preparation.

The urgency of the problem. In modern conditions, software development requires constant improvement of code analysis and optimization processes. Automated tools such as linters and formatters are essential for reducing errors, improving productivity, and ensuring the quality of software systems.

The purpose and tasks of the research. The goal of the work is to develop a code analysis and optimization system to simplify its maintenance and improve its quality. The tasks of the research include reviewing existing solutions, selecting appropriate technologies, implementing the system, and testing it.

Object of study. Processes of code analysis and optimization.

Subject of research. A system that allows adapting code management tools to reduce errors and improve software quality.

Practical value. The developed system provides convenient tools for code analysis, which can be integrated as an extension for Visual Studio Code. This helps reduce development costs and improve the quality of the final product.

Keywords: code analysis, optimization, linters, automation.

ЗМІСТ

ПЕРЕЛІК ТЕРМІНІВ ТА СКОРОЧЕНЬ

API – Application Programming Interface

AST – Abstract Syntax Tree

CI/CD – Continuous Integration, Continuous Delivery

CSS – Cascading Style Sheets

DevTools – Development Tools

ES – EcmaScript

FS – File System

HTML – HyperText Markup Language

HTTP – HyperText Transfer Protocol

IDE – Integrated Drive Electronics

JS – JavaScript

JSON – JavaScript Object Notation

SQL – Structured Query Language

TS – TypeScript

UI – User Interface

UX – User Experience

VSCode – Visual Studio Code

ПЗ – Програмне Забезпечення

РОЗДІЛ 1
АКТУАЛЬНІСТЬ ТЕМИ АНАЛІЗУ І ОПТИМІЗАЦІЇ ПРОГРАМНОГО КОДУ

1.1. Значення якості коду для розробки ПЗ

У сучасному світі розробка програмного забезпечення відіграє вирішальну роль у багатьох галузях діяльності, що вимагає постійного вдосконалення процесів створення, тестування та оптимізації коду. Зі збільшенням масштабів і складності програмних продуктів виникає необхідність у підвищенні якості та продуктивності коду, а також забезпеченні його стабільності й безпеки. Одним із важливих інструментів для досягнення цих цілей є автоматизовані системи аналізу та оптимізації коду, зокрема лінтери та форматувальники коду.

1.1.1. Вплив якості коду на продуктивність систем

Якість коду є одним із ключових чинників, що визначають продуктивність будь-якої програмної системи. Оптимізований і добре структурований код дозволяє системі працювати ефективніше, забезпечуючи мінімізацію використання ресурсів, таких як процесорний час та оперативна пам’ять. Натомість неоптимізований код може призводити до серйозних проблем із продуктивністю, що особливо критично в масштабованих системах та сервісах з великим навантаженням.

Продуктивність програмного забезпечення можна визначити як здатність виконувати поставлені завдання за мінімально можливий час, із найменшими витратами ресурсів. Основна мета оптимізації коду полягає в тому, щоб зробити програму більш ефективною і стабільною під час виконання. Це особливо важливо в контексті сучасних високонавантажених систем, які обслуговують мільйони користувачів щоденно.

У дослідженні продуктивності з використанням інструментів мікро-бенчмаркінгу, проведеному Kodezi Content Team, виявлено, що такі техніки, як скорочення кількості викликів функцій, кешування даних та використання вбудованих функцій, можуть суттєво зменшити час виконання програм [1]. На рис. 1.1 представлено вплив різних технік оптимізації на продуктивність ПЗ. Ці оптимізації показали статистично значуще покращення продуктивності при порівнянні до і після оптимізацій.

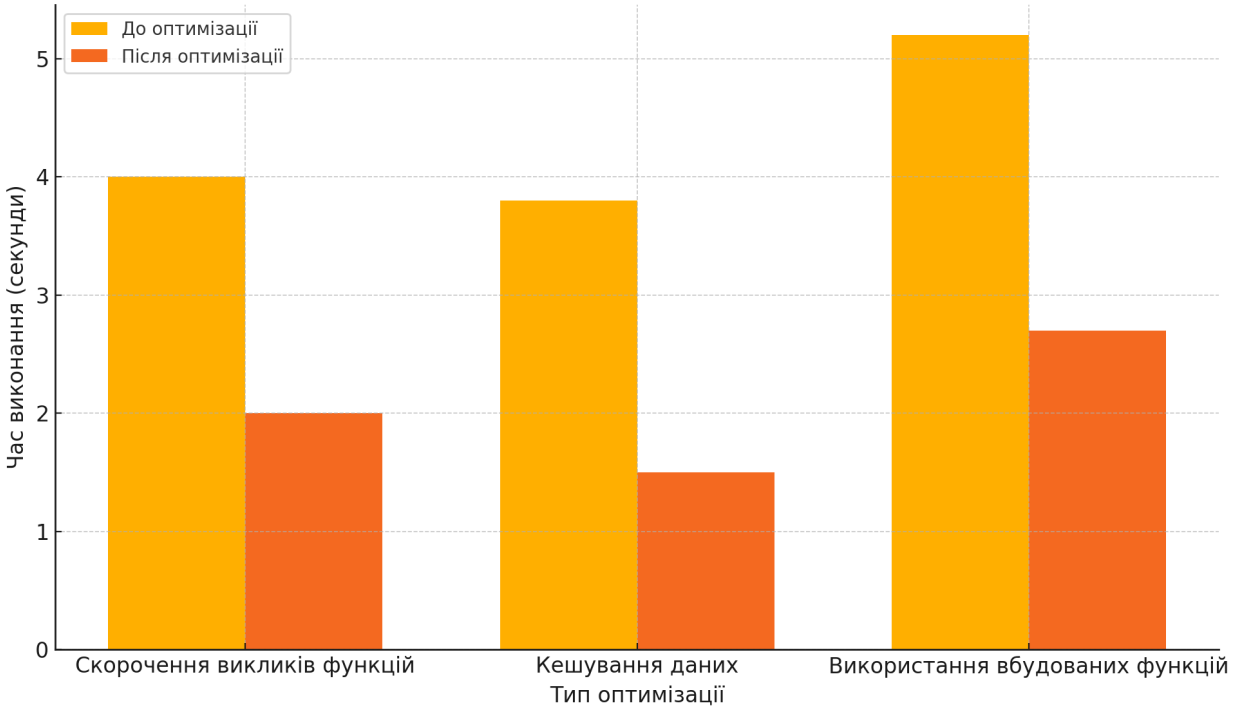

Рис. 1.1. – Вплив різних технік оптимізації на продуктивність

Щодо наслідків низької якості коду, можна виділити збільшення часу виконання операцій. Як приклад, неоптимізований алгоритм може мати складність O(n^2), тоді як його оптимізована версія може працювати за O(n log n), що значно скорочує час виконання для великих обсягів даних. Таким чином, некоректний вибір алгоритмів чи структур даних безпосередньо впливає на продуктивність системи.

Крім вибору алгоритмів, важливу роль у продуктивності відіграє стиль написання коду [2]. Коли код написано з урахуванням стандартів, що забезпечують його підтримуваність, це сприяє швидшому вирішенню проблем, полегшує командну роботу та дозволяє уникати повторних помилок.

Важливим фактором зниження продуктивності є також надмірне використання ресурсів через неоптимальні конструкції в коді. Наприклад, зайве дублювання операцій або неефективне використання пам’яті може призводити до затримок у роботі програми та навіть до її аварійного завершення. Оптимізація таких аспектів, як управління пам’яттю, паралельні обчислення та багатопоточність, може мати значний вплив на покращення продуктивності. Важливо забезпечити ефективне використання цих ресурсів, оскільки це дозволяє програмі швидше виконувати обчислення та обробляти більші обсяги даних, що є ключовим для сучасних додатків, які працюють із великими масивами інформації.

На рис. 1.2 продемонстровано вплив різних типів ресурсів (ЦП, пам’ять та дисковий простір) на затримку виконання до та після оптимізації за резуьтатами досліджень, що стосуються продуктивності хмарних систем, опублікованих у журналах з обчислювальних систем та хмарних технологій, таких як PLOS ONE [3] і Journal of Cloud Computing [4].

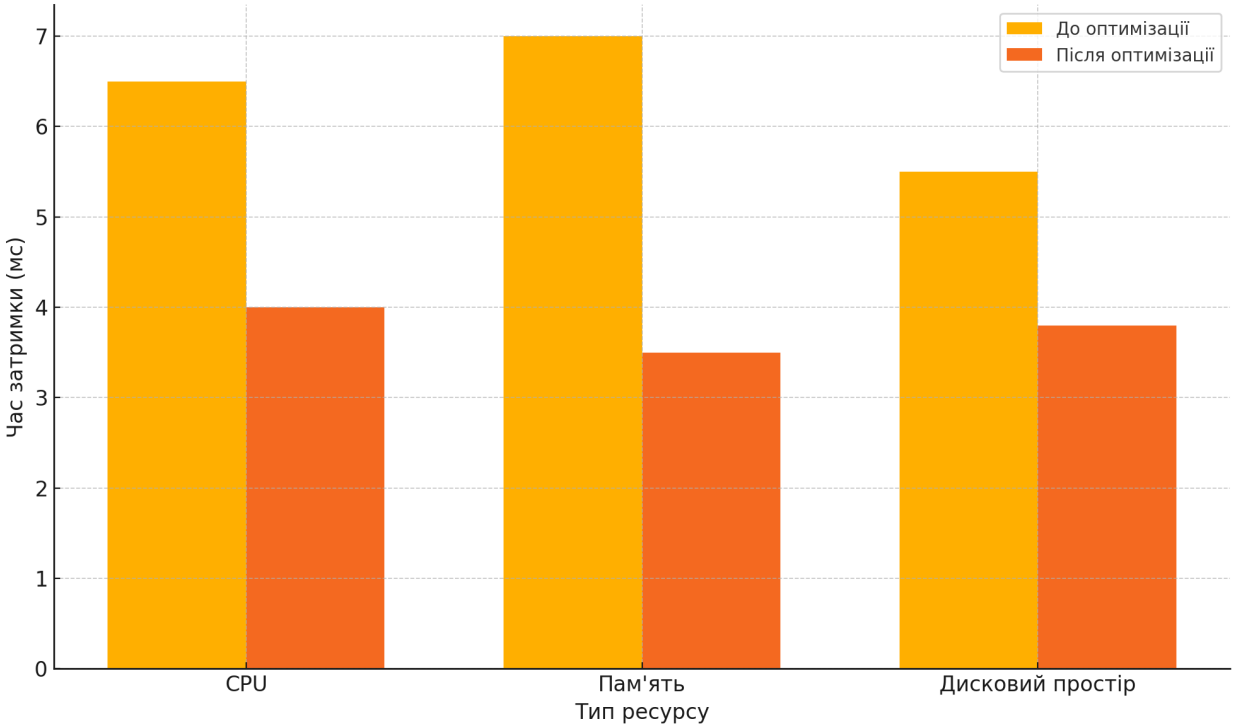

Рис. 1.2. – Вплив різних типів ресурсів на затримку виконання до та після оптимізації

Ще одним важливим аспектом є підтримка читабельності та спрощення коду, адже простота коду полягає не лише у меншій кількості рядків, а у зменшенні складності та забезпеченні його зрозумілості, підтримуваності та масштабованості [5]. Погана структура коду, велика кількість повторюваних конструкцій і відсутність модульності призводять до його ускладнення. Це робить важчим внесення змін у майбутньому та призводить до потенційних збоїв, якщо код потрібно масштабувати або адаптувати до нових умов.

Загалом, якісний код позитивно впливає на продуктивність системи шляхом оптимального використання ресурсів, зменшення часу на обробку даних і підтримки масштабованості, в свою чергу, висока якість коду забезпечує не лише кращі показники продуктивності, але й стійкість системи до змін і можливість її тривалого використання з мінімальними витратами на підтримку та розвиток.

1.1.2. Ризики помилок і вразливостей через неоптимізований код

Неоптимізований код також створює значні ризики з точки зору помилок та вразливостей, через що можуть виникати баги, які призводять до нестабільної роботи системи, а також критичні вразливості, які відкривають можливості для зловмисників. У сучасних умовах підвищених вимог до безпеки, особливо в корпоративних і державних системах, ці ризики можуть мати катастрофічні наслідки.

Один із головних ризиків неоптимізованого коду полягає в тому, що він часто включає зайві або неефективні операції, які створюють додаткові точки відмови. Це може призвести до труднощів під час тестування та налагодження, оскільки такі проблеми часто приховані та проявляються лише за певних умов, як, наприклад, неочікувана поведінка ПЗ через неоптимальний обробник виключень або неправильне використання пам’яті можуть призводити до збоїв у роботі програми та втрати даних.

Вразливості в неоптимізованому коді також часто виникають через відсутність належної перевірки вхідних даних або недостатній захист під час

їхньої обробки. Це може призводити до таких проблем, як SQL-ін’єкції, вразливості типу «буфер переповнений», або інші атаки, що експлуатують слабкі місця в структурі коду [6]. За оцінками експертів, близько 70% вразливостей у програмному забезпеченні пов’язані з низькою якістю коду та його неправильним управлінням даними.

Проблему слабкої стійкісті до змін або розширення функціональності можа виділити як одну з ключових проблем неоптимізованого коду. Коли код написаний неефективно або неструктуровано, кожне внесення змін може створювати нові баги або загострювати наявні проблеми. Це особливо актуально для великих командних проектів, де різні розробники можуть по-різному інтерпретувати структуру та логіку роботи коду. Нестандартний підхід або відсутність оптимізації призводять до непередбачуваних наслідків, які важко відстежити та виправити. Неоптимізований код також може викликати проблеми з підтримкою системи у довгостроковій перспективі та навіть бути небезпечним Такий код часто є результатом поганих практик написання коду та відсутності оптимізації, що призводить до вразливостей, які можна використати [7]. Підтримка складного та недокументованого коду стає трудомісткою, що збільшує ймовірність нових помилок при виправленні старих, тобто чим більше змін вносять у код, тим більше помилок виникає.

Окрім підвищеної ймовірності атак, неоптимізований код призводить до збільшення витрат на підтримку та розвиток системи. Команди розробників змушені витрачати більше часу на пошук та виправлення помилок, тестування змін і забезпечення стабільності. Це також уповільнює процес впровадження нових функцій, оскільки щоразу необхідно ретельно перевіряти існуючий код на предмет можливих конфліктів або нових помилок, які можуть виникнути в результаті внесених змін.

1.1.3. Неконсистентність стилю коду в командних проектах

У командних проектах консистентність стилю коду має критичне значення для підтримки продуктивності та ефективності роботи всієї команди.

Коли декілька розробників працюють над одним і тим самим кодовим базисом, важливо, щоб усі дотримувалися єдиних стандартів і стилістичних підходів. Неконсистентність стилю призводить до значних проблем у процесі розробки, таких як ускладнення читання коду, зменшення ефективності код-рев'ю та збільшення ймовірності виникнення помилок.

Відсутність єдиних правил у команді при неконсистентному стилю витікає в те, що кожен розробник може мати свої звички та уподобання в написанні коду, що відображається в різному використанні відступів, найменуванні змінних, розташуванні блоків коду та інших аспектах. Це не лише ускладнює розуміння логіки програми іншими членами команди, але й збільшує час на обробку та перевірку коду, оскільки доводиться пристосовуватися до різних підходів. На рисунках 1.3 – 1.4 продемострovано різницю між стилями коду. Так можемо побачити, що добре структурований код зі сталим стилем допомагає розробнику швидше орієнтуватися в логіці програми, легше виправляти помилки та підтримувати код у майбутньому. Він зменшує кількість непорозумінь і покращує командну роботу, оскільки всі учасники проекту легше розуміють код один одного.

```javascript
function calculateTotalPrice(items) {
    if (!Array.isArray(items)) {
        throw new Error( message: "Input must be an array");
    }

    let total = 0;

    for (let i = 0; i < items.length; i++) {
        const item = items[i];

        if (item.price && item.quantity) {
            total += item.price * item.quantity;
        } else {
            console.warn( message: `Item ${i} is missing price or quantity`);
        }
    }

    return total;
}

const shoppingCart = [
    { price: 10, quantity: 2 },
    { price: 5, quantity: 5 },
    { price: 15, quantity: 1 }
];

console.log(`Total price: $${calculateTotalPrice(shoppingCart)}`);
```

Рис. 1.3. – Приклад чистого коду

```javascript
function calcTotal(items){
    if (typeof items !== "object") throw "Not array";

    var total = 0;

    for (let i = 0; i<items.length; i++){
        let it = items[i];
        if(it.price && it["quantity"]){
            total+=it.price*it.quantity;
        } else {
            console.log("Error on item",i);
        }
    }
    return total;
}
const cart=[{price:10,quantity:2},{price:5,qUantity:5},{price:15,Quantity:1}]
console.log("Total Price = "+ calcTotal(cart));

```

Рис 1.4. – Приклад коду, що демонструє неконсистентність і ускладнення читання

Неконсистентний стиль також погіршує підтримуваність коду. Наприклад, коли новий розробник приєднується до проекту, йому важко адаптуватися до різних стилістичних підходів, що використовуються його

колегами. Це ускладнює розуміння структури проекту та збільшує ймовірність внесення некоректних змін.

Ще одна важлива проблема, пов’язана з неконсистентним стилем, – це зниження ефективності автоматизованих інструментів для аналізу та оптимізації коду. Інструменти, такі як лінтери або форматери коду, створені для того, щоб допомогти розробникам автоматично виправляти помилки стилю та забезпечувати консистентність. Проте, якщо в команді не встановлено чітких стандартів, то такі інструменти можуть конфліктувати або використовуватися лише частково, що призводить до непослідовних результатів.

Як зазначається в багатьох дослідженнях, консистентність стилю не лише полегшує роботу розробникам, а й покращує загальну якість продукту. Наприклад, у дослідженні, опублікованому в Journal of Software Engineering, було зазначено, що команди, що впроваджують послідовні стандарти кодування, стикаються з меншою кількістю помилок і швидше проводять код-рев’ю, що призводить до кращої загальної якості програмного забезпечення [8]

Таким чином, для успішної розробки великих командних проектів необхідно впроваджувати та дотримуватися єдиних стандартів стилю коду. Це дозволяє не лише підвищити продуктивність роботи команди, але й забезпечити стабільність та підтримуваність проекту у довгостроковій перспективі.

1.2. Необхідність автоматизації процесів аналізу та оптимізації коду

1.2.1. Переваги автоматизації для покращення якості коду

Автоматизація процесів аналізу та оптимізації коду є одним із найважливіших факторів, що впливають на покращення його якості. У сучасних умовах, коли розробники працюють з великими обсягами коду і складними системами, ручний аналіз стає неефективним. Автоматизовані інструменти, такі як лінтери, статичні аналізатори коду та форматери, значно

спрощують цей процес, забезпечуючи швидкий зворотний зв’язок та допомагаючи уникнути помилок на ранніх етапах розробки.

Основна перевага автоматизації полягає в тому, що вона дозволяє зменшити людський фактор у процесі написання та перевірки коду. Люди схильні до помилок, особливо коли йдеться про монотонні та рутинні завдання, такі як перевірка стилю коду або пошук типових помилок. Автоматизовані інструменти здатні швидко виявляти ці помилки і пропонувати можливі виправлення, що значно підвищує ефективність роботи команди розробників. Крім того, автоматизація забезпечує послідовність у перевірці коду. Це особливо важливо для команд, що працюють над великими проектами, де різні розробники можуть мати свої уподобання щодо стилю або підходів до вирішення завдань. Використання єдиних автоматизованих інструментів допомагає забезпечити стандартизацію коду, що покращує його читабельність і полегшує підтримку.

Ще однією перевагою автоматизації є швидкість виявлення проблем. Автоматизовані інструменти, такі як ESLint або Prettier, здатні миттєво аналізувати код під час його написання, що дозволяє розробникам отримувати зворотний зв’язок у реальному часі. Це значно знижує ймовірність того, що помилки або недоліки залишаться непоміченими до етапу тестування або продакшн-експлуатації. Інструменти, такі як статичні аналізатори коду, можуть перевіряти відповідність стандартам, дотримання правил безпеки або оптимізації використання ресурсів. Завдяки цьому можна виявити неочевидні помилки або місця, де код можна покращити для підвищення продуктивності. Інструменти автоматизованого аналізу коду стали необхідними для забезпечення відповідності програмного забезпечення як функціональним, так і нефункціональним вимогам, значно знижуючи ризик помилок та вузьких місць продуктивності [9].

Автоматизація також дозволяє розробникам зосередитися на важливіших завданнях, таких як вирішення складних алгоритмічних питань або реалізація

нових функцій. Менша увага приділяється рутинним завданням, що підвищує загальну продуктивність команди та пришвидшує процес розробки.

Автоматизовані інструменти також сприяють покращенню командної співпраці. У великих проектах, де над одним кодом працюють багато розробників, підтримка єдиного стилю та стандартів стає ключовим фактором для успішної розробки. Лінтери та інші інструменти допомагають зберігати консистентність коду, навіть коли над проектом працюють фахівці з різними навичками та підходами до програмування. Це полегшує читання та розуміння коду іншими членами команди, а також зменшує кількість помилок, які можуть виникати через неконсистентність.

Ще одним важливим аспектом автоматизації є можливість легкої інтеграції таких інструментів у CI/CD-процеси. Це дозволяє забезпечити постійну перевірку коду на відповідність стандартам та якості на кожному етапі розробки. Наприклад, при кожному коміті або злитті гілок код проходить через автоматизовані тести та аналізатори, що дозволяє виявити проблеми ще до того, як вони вплинуть на кінцевий продукт.

Загалом, автоматизація процесів аналізу та перевірки коду значно покращує його якість, знижує кількість помилок і підвищує продуктивність роботи команди. Автоматизовані інструменти допомагають швидко виявляти проблеми, забезпечують стандартизацію та спрощують співпрацю між розробниками, що робить їх незамінним інструментом у сучасному розробленні програмного забезпечення.

1.2.2. Використання автоматизованих інструментів для підвищення продуктивності

Автоматизовані інструменти стали незамінною частиною процесу розробки програмного забезпечення, значно впливаючи на продуктивність команди. Використання таких інструментів дозволяє прискорити виконання рутинних завдань, що, у свою чергу, звільняє час для складніших аспектів розробки. Оптимізація робочих процесів через автоматизацію дає змогу

командам швидше і ефективніше досягати результатів, не знижуючи при цьому якості продукту.

Однією з головних переваг автоматизації для підвищення продуктивності є автоматичне форматування коду та виявлення помилок. Інструменти, такі як ESLint і Prettier, дозволяють розробникам автоматично виправляти проблеми зі стилем та структурою коду під час написання. Це зменшує кількість часу, необхідного для ручного перегляду коду, знижує ризик виникнення помилок та забезпечує більш послідовний підхід до стилю коду у великих командах.

Таблиця 1.1.

Порівняльний аналіз зниженням кількості ручних перевірок коду для різних автоматизованих інструментів

Автоматизований інструмент	Роль	Зменшення часу ручного перегляду (%)
Лінтери (ESLint, Pylint)	Автоматична перевірка синтаксису та стилю коду	30-40
Unit tests	Автоматична перевірка функцій коду	50
CI/CD	Автоматичне виконання тестів і деплой	60

Таблиця 1.1. демонструє взаємозв’язок між використанням автоматизованих інструментів і зниженням кількості ручних перевірок коду

Автоматизовані інструменти також дозволяють швидко знаходити та виправляти помилки, що підвищує продуктивність шляхом зниження необхідності на тривалий процес дебагінгу. Завдяки цьому розробники можуть зосередитися на розробці нових функцій або поліпшенні існуючих компонентів системи, замість того, щоб витрачати час на пошук дрібних помилок або недоліків.

Автоматизовані системи також значно прискорюють процес тестування. Інструменти для автоматизованого тестування, такі як Jest, Mocha або Selenium, дозволяють швидко і надійно перевіряти коректність роботи різних частин програми. Це дає змогу розробникам переконатися, що нові зміни не впливають на вже існуючий функціонал, і дозволяє мінімізувати ризики виникнення нових помилок. Автоматизація тестування підвищує ефективність роботи команди, оскільки тестові сценарії можуть виконуватись багаторазово без участі людини, що прискорює цикли розробки.

Інтеграція автоматизованих інструментів у процеси CI/CD також дозволяє значно підвищити продуктивність команди. Завдяки автоматизованим перевіркам коду, тестуванню та випуску нових версій, команди можуть швидше отримувати зворотний зв’язок і вносити необхідні зміни. Це прискорює цикл розробки та дозволяє постійно вдосконалювати продукт.

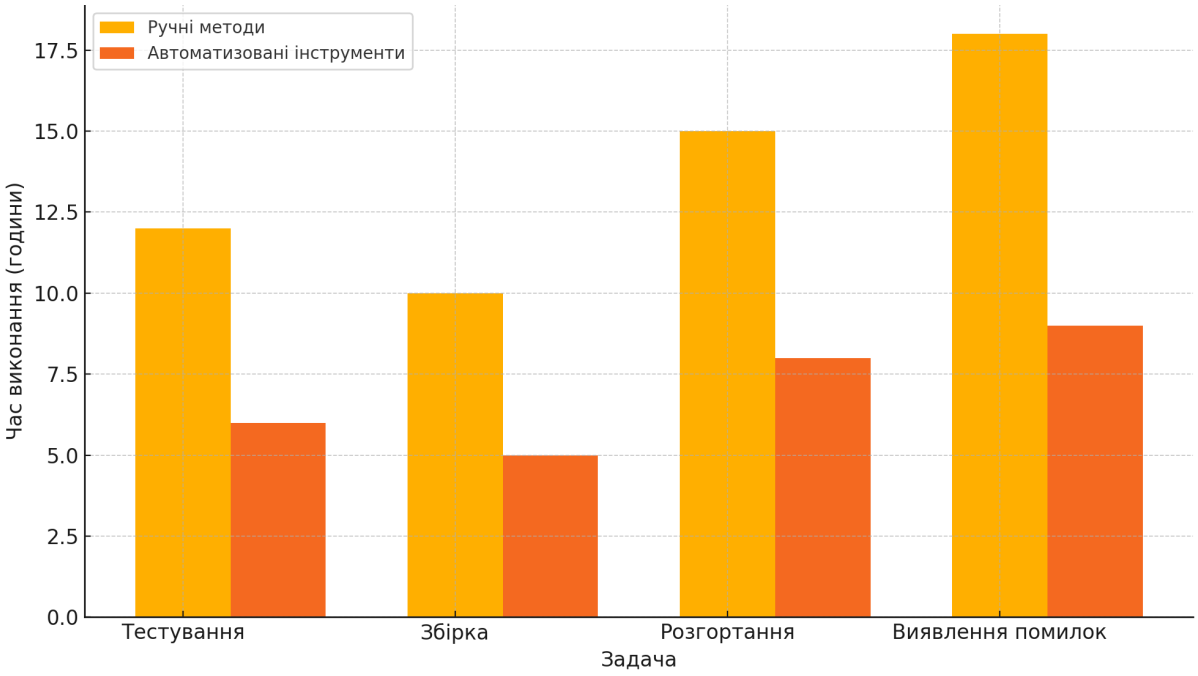

Рис. 1.5. - Скорочення часу на розробку після впровадження автоматизованих інструментів

Як зазначено в дослідженні, опублікованому IEEE Software, «інструменти безперервної інтеграції та автоматизованого тестування зменшують час,

витрачений на ручні завдання, до 50%, що призводить до швидших циклів розробки та покращення якості програмного забезпечення» [10].

Крім пришвидшення процесу розробки, автоматизовані інструменти також дозволяють уникати так званого «людського фактору», тобто помилок, пов’язаних із недоглядом або втомою. Системи автоматичного аналізу коду, тестування та випуску дозволяють команді працювати стабільно і з високою продуктивністю навіть у стресових умовах.

1.3 Роль ESLint і Prettier у сучасному програмуванні

У сучасному програмуванні інструменти для автоматизації аналізу коду, такі як ESLint і Prettier, стали важливою частиною робочого процесу більшості команд розробників. Вони допомагають забезпечувати високу якість коду, спрощують командну роботу і значно прискорюють процес розробки. Завдяки цим інструментам, програмісти можуть виявляти помилки, покращувати стиль та структуру коду в автоматичному режимі, що робить розробку більш ефективною та менш схильною до людських помилок. Хоча ці два інструменти мають різні завдання, вони доповнюють одне одного та використовуються разом для досягнення оптимальних результатів.

1.3.1 Основні функції ESLint і Prettier

Основною функцією ESLint є статичний аналіз коду для виявлення помилок і порушень стилю програмування. Цей інструмент дозволяє знаходити такі помилки, як неправильне використання змінних, порушення правил синтаксису, застарілі або небезпечні конструкції, а також неефективні підходи до написання коду. Завдяки статичному аналізу, ESLint може запобігати багатьом проблемам ще до того, як код потрапляє в стадію тестування або продакшн. Користувачі можуть налаштовувати правила ESLint відповідно до вимог конкретного проекту або використовувати стандартні конфігурації.

Однією з ключових особливостей ESLint є можливість не тільки виявляти помилки, але й автоматично їх виправляти. За допомогою команди *eslint –fix* [11], розробник може виправити прості порушення правил коду автоматично. Це дозволяє зекономити час на ручному виправленні стилістичних помилок або незначних багів.

Іншою важливою функцією ESLint є його інтеграція з популярними середовищами розробки (IDE), такими як VSCode, WebStorm, та інші. Це дозволяє розробникам отримувати миттєвий зворотний зв’язок під час написання коду, коли ESLint підсвічує проблемні місця та пропонує варіанти виправлення прямо в редакторі. Така інтеграція значно полегшує робочий процес, оскільки не вимагає запуску окремих перевірок – все відбувається в реальному часі, що знижує кількість потенційних помилок ще на етапі написання коду.

Prettier, хоча й не є інструментом для виявлення помилок, виконує важливу роль у забезпеченні послідовного стилю коду. Його основна функція полягає в автоматичному форматуванні коду відповідно до заданих правил, незалежно від того, як код був написаний. Prettier усуває всі варіанти неправильного форматування, наприклад, недоречні відступи, пробіли або розміщення дужок, що гарантує однаковий стиль по всьому проекту. Такий підхід допомагає уникнути неузгодженостей у стилі між різними розробниками в команді.

Однією з головних переваг Prettier є те, що він повністю автоматизує процес форматування, дозволяючи розробникам зосередитися на логіці програмування, не турбуючись про форматування коду. Коли розробник зберігає файл у своєму редакторі, Prettier автоматично застосовує необхідні правила форматування, тим самим забезпечуючи єдиний стиль коду по всьому проекту. Це особливо корисно в командних проектах, де різні розробники можуть мати різні стилі написання коду.

Prettier також підтримує багато мов програмування, таких як JavaScript, TypeScript, HTML, CSS, і навіть інші, менш поширені мови. Це робить його

універсальним інструментом, який може бути використаний у різних типах проектів для забезпечення єдиного стилю форматування.

Крім того, Prettier і ESLint добре інтегруються один з одним. У багатьох випадках їх використовують разом для досягнення максимальної ефективності в забезпеченні якості коду. ESLint виявляє і виправляє логічні помилки та порушення стандартів програмування, тоді як Prettier піклується про стилістику і форматування, що робить їх разом потужним інструментом для підтримки якості коду.

Таким чином, ESLint і Prettier забезпечують різні, але доповнювальні функції, які допомагають підвищити якість, підтримуваність та читабельність коду в будь-якому проекті. Їх використання в сучасному програмуванні стало стандартом для багатьох команд, що дозволяє покращити процес розробки та забезпечити високу якість кінцевого продукту.

1.3.2 Інтеграція цих інструментів у середовища розробки

Інтеграція таких інструментів, як ESLint і Prettier, у середовища розробки стала важливою складовою сучасних робочих процесів розробників. Вони дозволяють автоматично аналізувати код і забезпечувати його відповідність стандартам без необхідності виконання додаткових команд або запуску окремих інструментів. Така інтеграція підвищує продуктивність та знижує ймовірність появи помилок або проблем із форматуванням ще на етапі написання коду.

ESLint інтегрується з багатьма популярними середовищами розробки, такими як Visual Studio Code, WebStorm, Sublime Text, Atom та іншими. У випадку з VSCode, інтеграція є особливо гнучкою та потужною завдяки існуючим плагінам. Плагіни ESLint для VSCode дозволяють здійснювати перевірку коду в режимі реального часу, підсвічуючи проблеми та надаючи пропозиції щодо виправлення безпосередньо під час написання коду. Це значно спрощує робочий процес і дозволяє розробникам швидко реагувати на проблеми без необхідності запускати сторонні інструменти.

Однією з найбільших переваг такої інтеграції є автоматичне виправлення стилістичних помилок. Розробник може налаштувати ESLint так, щоб він не лише виявляв помилки, а й автоматично виправляв їх під час збереження файлу або натискання гарячих клавіш. Це значно зменшує час на ручне виправлення, дозволяючи зосередитись на логіці та функціональності коду.

Prettier також легко інтегрується з багатьма середовищами розробки, особливо з Visual Studio Code, що робить процес форматування коду повністю автоматизованим. Після інтеграції Prettier виконує форматування коду під час кожного збереження файлу, що усуває необхідність у ручних виправленнях. Це дозволяє розробникам зосередитись на логіці коду, не хвилюючись про відступи, пробіли та інші дрібні аспекти форматування.

Особливо корисною функцією є можливість інтеграції Prettier та ESLint разом, оскільки ці інструменти доповнюють один одного таким чином – ESLint налаштовує правила щодо правильного використання синтаксису та логічних помилок, а Prettier відповідає виключно за форматування коду. Це дозволяє команді забезпечити не лише чистоту коду, але й єдиний стиль по всьому проекту.

Крім того, обидва інструменти підтримують розширену налаштовуваність. У середовищах розробки, таких як VSCode, розробники можуть створювати індивідуальні правила та конфігурації для проектів, що дозволяє кожній команді налаштувати інструменти під свої конкретні потреби. Наприклад, можна налаштувати Prettier для специфічного форматування HTML або CSS, а ESLint для перевірки відповідності стандартам JavaScript або TypeScript.

Ще одним важливим аспектом інтеграції ESLint і Prettier у середовища розробки є можливість підключення до процесів CI/CD. Наприклад, під час кожного коміту або пушу в репозиторій код автоматично перевіряється на відповідність стандартам ESLint і форматування Prettier. Це дозволяє забезпечити високий рівень якості коду на всіх етапах розробки та запобігає введенню у проект погано відформатованого або помилкового коду.

Інтеграція цих інструментів також полегшує командну роботу. У великих проектах, де над кодом працюють кілька розробників, підтримка єдиного стилю і стандартів є надзвичайно важливою. Завдяки ESLint і Prettier всі учасники команди можуть бути впевнені, що їхній код відповідає єдиним правилам, незалежно від особистих вподобань у написанні коду.

Таким чином, інтеграція ESLint і Prettier у середовища розробки, такі як VSCode, дозволяє однразу і підвищити продуктивність розробників, і забезпечити високий рівень стандартизації та якості коду. Ці інструменти автоматизують багато аспектів перевірки та форматування, що робить їх важливою частиною сучасного процесу розробки програмного забезпечення.

1.4. Визначення ефективних стратегій аналізу коду

Ефективний аналіз коду є основою для забезпечення високої якості програмного забезпечення, а також для мінімізації кількості помилок і підвищення продуктивності системи. Визначення стратегії аналізу залежить від багатьох факторів, включаючи розмір проекту, його складність, кількість учасників команди і вимоги до безпеки та продуктивності. Важливо обирати стратегії, що відповідають специфіці конкретного проекту, і застосовувати їх послідовно на всіх етапах розробки.

Серед стратегій аналізу коду найпоширенішим є статичний аналіз, який передбачає перевірку коду без його виконання. Статичні аналізатори, такі як ESLint, дозволяють знаходити потенційні помилки, порушення стилістичних правил або проблеми з оптимізацією коду на ранніх етапах розробки. Цей підхід особливо корисний для великих проектів, де навіть незначні помилки можуть призвести до серйозних наслідків. Статичний аналіз коду забезпечує високу швидкість виявлення проблем і дозволяє запобігти їх накопиченню.

Також потрібно виділити динамічний аналіз, який передбачає тестування коду під час його виконання. Цей підхід дозволяє виявити проблеми, що виникають лише під час роботи програми, наприклад, помилки з пам’яттю або

некоректне оброблення даних. Динамічний аналіз є ключовим для забезпечення стабільності і безпеки програмного забезпечення, оскільки дозволяє побачити, як код поводиться у реальних умовах.

Динамічний аналіз коду часто використовується в поєднанні з автоматизованим тестуванням. Інструменти, такі як Jest, Selenium або Mocha, дозволяють виконувати тестові сценарії, що моделюють різні умови роботи програми, включаючи стрес-тести під високим навантаженням. Це допомагає не лише виявити помилки, але й перевірити, як ефективно програма справляється з великим обсягом даних або множинними запитами.

Іншою важливою стратегією є комбінований підхід, який поєднує в собі як статичний, так і динамічний аналіз. Використання обох підходів забезпечує комплексну перевірку як логіки програми, так і її стабільності під час виконання. Статичні аналізатори можуть виявити типові помилки, такі як неправильні змінні або некоректний синтаксис, тоді як динамічний аналіз дозволяє виявляти проблеми з продуктивністю або безпекою, що виникають тільки під час виконання коду. Така стратегія є однією з найбільш ефективних, особливо для великих проектів, де важливо знизити ризики на всіх етапах розробки.

Ефективні стратегії також включають безперервну інтеграцію (Continuous Integration, CI) і безперервне постачання (Continuous Delivery, CD) [12]. Ці підходи передбачають автоматизацію процесу перевірки та тестування коду на кожному етапі розробки. CI/CD дозволяють проводити регулярні перевірки після кожного коміту, що значно знижує ймовірність введення помилок у код і забезпечує швидкий зворотний зв’язок для розробників.

Інтеграція автоматизованих інструментів для аналізу коду в процеси CI/CD є однією з найефективніших стратегій для великих проектів. Цей підхід забезпечує автоматичний запуск статичних і динамічних аналізаторів кожного разу, коли новий код додається до репозиторію. Завдяки цьому, розробники отримують швидкий зворотний зв’язок про можливі проблеми в коді, що дозволяє знизити витрати на виправлення помилок у майбутньому.

Використання практик CI/CD допомагає знизити кількість дефектів у коді до 50% і значно покращує швидкість розробки [13].

Також важливо використовувати рев'ю коду, як одну з ключових стратегій для покращення якості. Хоча автоматизовані інструменти можуть знаходити багато помилок і стилістичних порушень, людський погляд все ще є важливим елементом перевірки коду. Рев'ю дозволяє більш досвідченим членам команди перевірити логіку, архітектуру та загальну структуру коду, забезпечуючи його читабельність і підтримуваність.

Таким чином, ефективні стратегії аналізу коду включають комбінування статичних і динамічних підходів, інтеграцію автоматизованих інструментів у CI/CD процеси та регулярне рев'ю коду. Це забезпечує високий рівень якості коду, зменшує кількість помилок і підвищує продуктивність команд, що працюють над великими проектами.

ВИСНОВОК ДО РОЗДІЛУ 1

Перший розділ роботи демонструє важливість аналізу та оптимізації коду в сучасній розробці програмного забезпечення. Високоякісний код, що є добре структурованим і оптимізованим, є ключовим фактором у забезпеченні ефективності та продуктивності програмних систем. Недотримання стандартів якості може призвести до серйозних проблем з продуктивністю, стабільністю та безпекою програмного забезпечення, що особливо критично у великих і масштабованих проектах.

Розглянуті аспекти, такі як вплив якості коду на продуктивність систем, свідчать про те, що правильний вибір алгоритмів, структур даних і способів управління ресурсами може значно скоротити час виконання операцій і зменшити навантаження на апаратні ресурси. Окрім того, аналіз можливих ризиків через неоптимізований код показав, що він може створювати критичні вразливості та баги, що в подальшому призводить до збільшення часу на підтримку та тестування коду. Це не лише уповільнює процес розробки, але й створює додаткові фінансові витрати для компаній.

Автоматизація процесів аналізу та оптимізації, зокрема за допомогою інструментів, таких як ESLint і Prettier, є необхідною умовою для зниження людського фактору в написанні коду. Завдяки використанню цих інструментів команди розробників можуть забезпечити послідовність у стилі коду, зменшити кількість помилок і прискорити процес виправлення проблем. Інтеграція цих інструментів у середовища розробки та CI/CD процеси значно підвищує ефективність командної роботи, забезпечуючи безперервний контроль за якістю коду.

Отже, перший розділ роботи підкреслює актуальність аналізу та оптимізації коду як необхідної умови для створення продуктивного, стабільного та безпечного програмного забезпечення. Використання сучасних автоматизованих інструментів дозволяє розробникам не тільки зосередитися на основних аспектах проектування та архітектури систем, але й забезпечити

стабільну та безперебійну роботу програмного продукту на всіх етапах його розвитку та підтримки.

РОЗДІЛ 2
НЕДОЛІКИ ІСНУЮЧИХ РІШЕНЬ І ПЕРЕВАГИ ЗАПРОПОНОВАНОГО РІШЕННЯ

2.1. Недоліки сучасних інструментів аналізу та форматування коду

Інструменти ESLint та Prettier є невід'ємною частиною для обробки коду таких мовах програмування, як JavaScript і TypeScript, забезпечуючи стандартизацію коду і виявлення помилок ще на етапі розробки. Однак, попри їхню популярність та ефективність у спрощенні процесів, вони мають кілька важливих недоліків, що обмежують їхнє використання в складних проектах та великих командах.

1. Обмежена інтерактивність у процесі виправлення помилок

Основна проблема ESLint полягає у відсутності інтерактивного підходу до виправлення помилок. Інструмент пропонує автоматичне виправлення за допомогою команди «*eslint –fix*», але цей процес не завжди ідеально відповідає потребам розробника. Часто ESLint автоматично виправляє синтаксичні або стилістичні помилки без можливості запропонувати варіанти виправлень або пояснити, чому обране виправлення є правильним. Це зменшує контроль над кодом і може призводити до ситуацій, коли автоматичне виправлення не враховує специфічні вимоги проекту.

Більш гнучкий підхід до виправлень, такий як інтерактивні підказки, дозволив би розробникам вибирати між кількома варіантами виправлення, що сприяло б кращому розумінню коду і зберігало б контекст.

2. Жорсткі правила форматування

Prettier, хоч і вирішує важливу проблему узгодженості стилю коду, має дуже жорсткі правила форматування, які не завжди можна налаштувати під специфіку проекту [14]. Наприклад, Prettier автоматично вирівнює дужки та відступи за власними стандартами, які не завжди відповідають стилістичним вимогам проекту. Він не дозволяє значного налаштування стилю –

розробникам доводиться приймати його стандарти або шукати альтернативні рішення. Це обмежує гнучкість і викликає конфлікти у великих командах, де кожен розробник може мати свої уподобання щодо стилю коду.

3. Обмеження виявлення типових помилок та складних проблем

ESLint чудово підходить для виявлення синтаксичних помилок і порушень стилю в JavaScript і TypeScript, але він має обмежені можливості у виявленні більш складних логічних помилок. Наприклад, ESLint не завжди може ідентифікувати потенційні проблеми з багатопоточністю або продуктивністю коду, оскільки він виконує лише статичний аналіз. Проблеми з пам’яттю, обробкою виключень чи оптимізацією обчислень часто залишаються поза увагою ESLint, оскільки ці аспекти проявляються тільки під час виконання коду.

Існують динамічні аналізатори, такі як Sentry або AppDynamics, які здатні моніторити роботу програми під час її виконання, але вони не інтегруються безпосередньо з ESLint для надання комплексного аналізу. Це створює прогалину між статичним аналізом і реальним виконанням, через що команди розробників не отримують повної картини щодо можливих проблем у коді.

4. Неможливість перевірки окремої чатини проекту

Проблема перевірки збережених файлів з кодом, коли зміни вносяться лише у невеликі частини проекту, є однією з ключових у використанні інструментів статичного аналізу, таких як ESLint. Величезні кодові бази, характерні для великих проектів, можуть містити десятки тисяч рядків коду, і проведення повної перевірки кожного разу після невеликої зміни значно уповільнює процес розробки. Кожен раз, коли розробник вносить зміни в код, ESLint за замовчуванням аналізує весь файл або навіть весь проект, незалежно від того, скільки рядків було змінено.

Аналіз файлів, які не зазнали змін, не має сенсу з точки зору продуктивності. Вже перевірені файли, що не містять помилок, фактично не потребують повторного аналізу, оскільки їхній стан залишається незмінним. У проєктах з великою кодовою базою процес перевірки може займати значний

час. Наприклад, у випадку з проектом на кілька тисяч файлів повний аналіз може тривати кілька хвилин або більше, що не є продуктивним, особливо якщо зміни внесені в лише один модуль або кілька функцій. Набагато ефективнішим підходом було б інтегрування можливості перевірки тільки тих частин проекту, які були змінені з часу останнього коміту. Перевірка тільки змінених частин дозволила б заощадити час і підвищити продуктивність команди, оскільки розробники зможуть швидше отримувати зворотний зв'язок про внесені зміни. Така первірка допомагає зосередитися на виправленні помилок і проблем, які були додані останніми комітами, замість того щоб постійно перевіряти незмінні частини проекту.

2.2. Потреба в нових рішеннях для більш ефективної розробки

Сучасні інструменти для аналізу та форматування коду, хоча й значно полегшують роботу розробників, не завжди відповідають новим вимогам і викликам, які виникають у сучасних умовах програмної інженерії. Зі збільшенням складності програмних проектів, розширенням команд розробників і впровадженням нових методологій, таких як Agile та DevOps, стає очевидним, що існуючі рішення мають свої обмеження, що створює потребу в нових інструментах, які могли б покращити ефективність процесів розробки та забезпечити вищу якість програмного забезпечення [15].

1. Інтерактивні інструменти для аналізу коду

Замість автоматичного застосування виправлень, як це роблять ESLint чи Prettier, нові рішення могли б надавати розробникам можливість вибору між кількома варіантами виправлення. Такий підхід дозволить уникнути типових конфліктів стилю чи логіки, коли автоматичне виправлення не враховує специфіку проекту, що допоможе зменшити ризики виникнення помилок через некоректні виправлення, які можуть бути внесені автоматичними інструментами.

2. Інтеграція статичного і динамічного аналізу

Як ми вже згадували, статичний аналіз добре підходить для перевірки синтаксичних помилок та порушень стилю, тоді як динамічний аналіз здатен виявляти проблеми, що виникають лише під час виконання програми, такі як витоки пам’яті або проблеми з продуктивністю. Створення інструменту, який поєднував би обидва ці підходи, дозволило б командам розробників отримувати більш комплексну інформацію про стан їхнього коду. Наприклад, інтеграція з CI/CD-системами, де спершу виконуватиметься статичний аналіз на етапі розробки, а динамічний – на етапі тестування або безпосередньо перед випуском, могла б значно підвищити ефективність перевірки. Це дозволило б виявляти потенційні проблеми на різних етапах розробки і випускати більш стабільні та безпечні продукти.

3. Покращення підтримки великих кодових баз

Зі збільшенням складності проектів, постає питання ефективного управління великими кодовими базами. Для покращення роботи з такими проектами інструменти повинні використовувати більш ефективні методи аналізу, такі як інкрементальний аналіз, коли перевіряються тільки нові або змінені частини коду. Наприклад, інструмент міг би запам’ятовувати результати аналізу для незмінних частин коду, і використовувати їх повторно, перевіряючи тільки ті частини, що були модифіковані.

4. Потреба в інструментах для полегшення рефакторингу

Нині більшість інструментів тільки сигналізують про наявність проблеми, але не надають конкретних рекомендацій для покращення архітектури або структури коду. Інтерактивні підказки для рефакторингу могли б допомогти уникнути типової проблеми, коли рефакторинг відкладається через складність процесу. Інструмент, що пропонує різні варіанти рефакторингу, може значно спростити роботу для розробників і зменшити кількість технічного боргу.

За вищеописаними недоліками можемо зробити висновок – насьогодні присутня необхідність у створенні більш адаптивних інструментах обробки коду, що повинні враховувати виклики сучасної розробки, такі як інтерактивність, тісна інтеграція з системами контролю версій, підтримка

великих кодових баз та допомога в рефакторингу. Створення таких рішень дозволить розробникам працювати швидше, ефективніше та з меншими ризиками, що позитивно вплине на якість програмного забезпечення.

2.3. Переваги запропонованого рішення над існуючими інструментами

2.3.1. Інтерактивні підказки для покращення коду

Однією з головних відмінностей мого рішення від існуючих інструментів, таких як ESLint або Prettier, є впровадження функції Hinting Mode – інтерактивних підказок для покращення коду. Інтерактивні підказки в Hinting Mode забезпечують гнучкість вибору. Наприклад, коли ESLint виявляє помилку або порушення стилю, він одразу пропонує автоматичне виправлення. Однак цей підхід не враховує контекст проекту або індивідуальні вимоги розробника. Hinting Mode дозволяє уникнути цього завдяки можливості обрати серед кількох варіантів виправлень або навіть залишити код без змін, якщо розробник вважає це більш доречним.

Це робить Hinting Mode значно більш дружнім для користувача, порівняно з існуючими інструментами, де автоматичні виправлення можуть викликати додаткові проблеми, особливо у великих проектах з високими вимогами до специфіки коду.

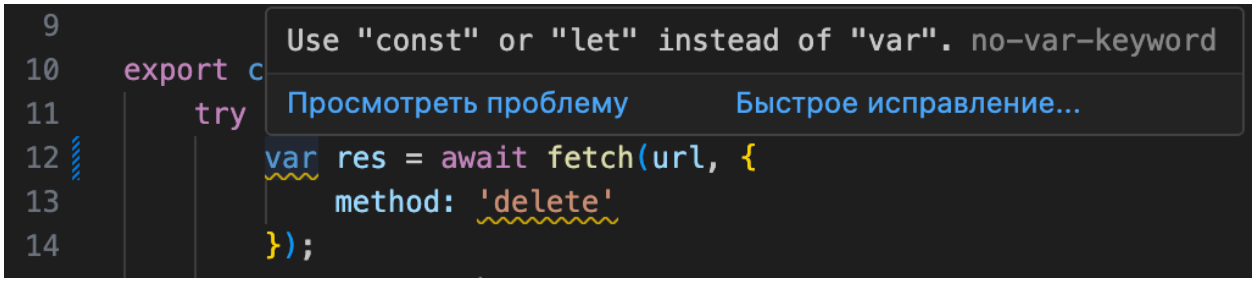

Рис. 2.1. – Приклад роботы Hinting Mode

Ще однією важливою перевагою Hinting Mode є його здатність пояснювати, чому пропонується те чи інше виправлення. Наприклад, замість того щоб просто виправити проблему стилю або логіки, Hinting Mode надає

короткі пояснення, що базуються на найкращих практиках розробки. Це допомагає розробникам не тільки уникати помилок, але й розуміти їхню природу, що особливо корисно для менш досвідчених членів команди. У результаті, кожне виправлення стає не просто механічною зміною, а можливістю для навчання.

Для порівняння, такі інструменти, як Prettier або ESLint, просто виконують автоматичне форматування або виправлення, що робить процес менш прозорим. Це може бути швидко та зручно для стандартних проектів, але в складних системах автоматичне застосування може призвести до виникнення додаткових проблем або до необхідності коригування виправлень вручну.

Ця функція також має важливе значення для командної роботи. Розробники з різним рівнем досвіду зазвичай по-різному реагують на виправлення, запропоновані автоматичними інструментами. Hinting Mode зменшує кількість непорозумінь та конфліктів у команді, дозволяючи кожному розробнику обирати найбільш відповідне рішення для конкретної ситуації. У великих проектах з нестандартною архітектурою це дозволяє уникнути конфліктів між правилами автоматичного форматування та логікою самого проекту. Наприклад, якщо проект вимагає специфічного розташування дужок чи іншого підходу до обробки даних, Hinting Mode може бути налаштований таким чином, щоб пропонувати лише ті варіанти виправлень, які відповідають конкретним вимогам.

2.3.2. Покроковий аналіз коду для полегшення рефакторингу

Переваги покрокового аналізу полягають у тому, що він дає змогу проводити глибокий аналіз окремих модулів або функцій без необхідності перевірки всього проекту одразу. За допомогою Step-by-step linting, розробники отримують можливість працювати над вдосконаленням коду поетапно, мінімізуючи ризики і підвищуючи продуктивність.

Чому важливо фокусуватися на окремих частинах коду? У великих проектах аналіз всього коду одразу може бути занадто трудомістким і

непродуктивним. Часто рефакторинг потребує детальної уваги до конкретних модулів або компонентів, які можуть мати більш складну логіку або бути застарілими. Аналіз повного коду файлу в таких випадках створює ризик того, що значна частина проекту залишається без уваги або що зміни, зроблені в одній частині проекту, можуть не відповідати решті системи. Фокусуючись на окремих частинах коду, розробник зосереджується на конкретних проблемах, проводячи більш глибокий аналіз і виправлення, що дозволяє уникати ситуацій, коли виправлення одного багу або стилістичної помилки призводить до виникнення нових проблем у іншій частині проекту. Фокусування на окремих частинах коду також допомагає уникнути перевантаження розробників. Коли проект налічує тисячі або навіть десятки тисяч рядків коду, одномоментний аналіз усієї бази може призвести до втрати уваги до критичних деталей або навіть до помилкових рішень. Покроковий аналіз дозволяє вирішувати проблеми поступово, даючи розробнику можливість краще розуміти контекст, в якому виникають ті чи інші помилки.

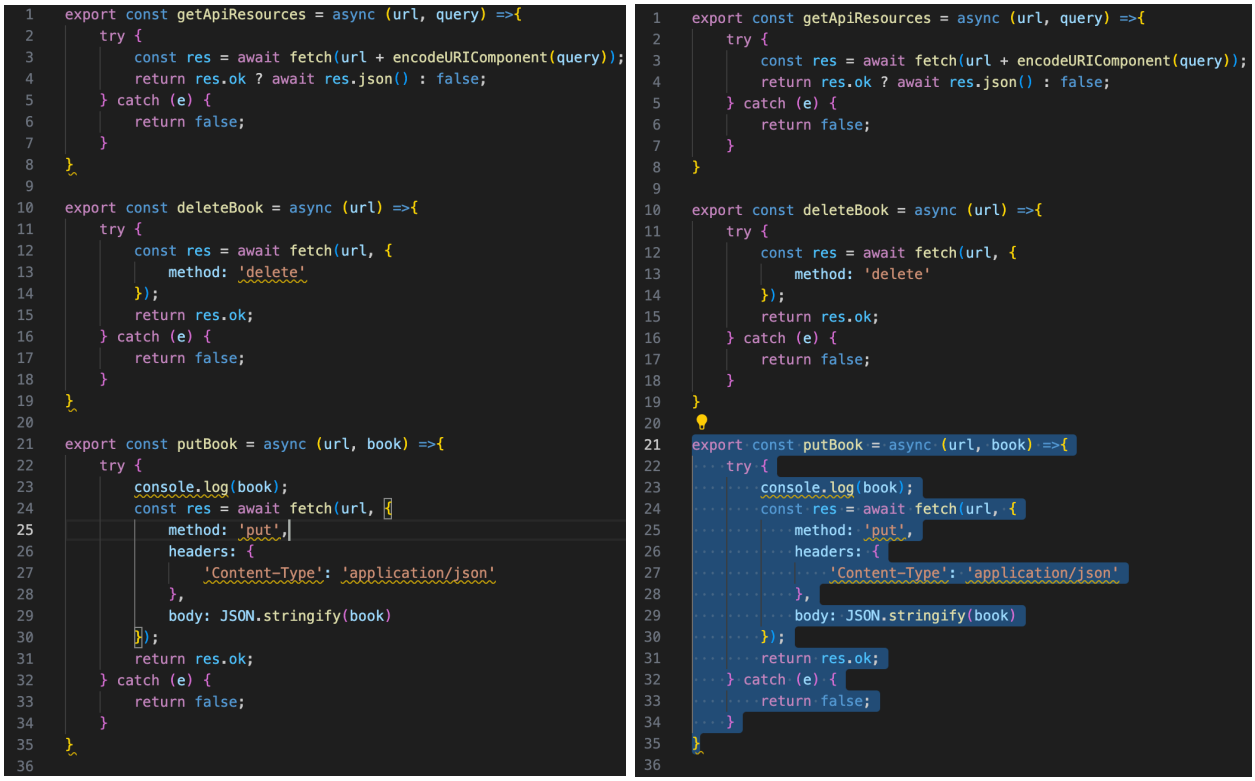

Рис 2.2. – Приклад роботы step-by-step linting

Покроковий підхід також значно полегшує роботу в команді, коли декілька розробників працюють над різними частинами проекту, застосування глобального аналізу до всього коду може спричинити конфлікти, особливо якщо вносяться зміни в різні модулі одночасно. Step-by-step linting дає змогу кожному члену команди зосередитися на своїй частині коду, не турбуючись про зміни в інших частинах проекту.

ВИСНОВОК ДО РОЗДІЛУ 2

У другому розділі магістерської роботи проведено огляд та аналіз основних інструментів для автоматизації аналізу та форматування коду, таких як Prettier та ESLint, а також розглянуто їхні ключові переваги, недоліки та сфери застосування. Розділ чітко структурує інформацію щодо поточних рішень, підкреслюючи їхню роль у забезпеченні стилістичної та функціональної узгодженості коду, що важливо для сучасних командних проектів.

На початку розділу зосереджено увагу на інструментах для автоматизації форматування, таких як Prettier, який забезпечує єдиний стиль коду, автоматизуючи процес його форматування. Використання Prettier дозволяє уникнути суперечок у команді щодо стилю, знижує ймовірність помилок, пов’язаних із неправильним форматуванням, та сприяє підтримуваності коду в довгостроковій перспективі. Інші інструменти для форматування також забезпечують функції автоматизації, проте їхні можливості обмежені у порівнянні з Prettier.

Далі розглядаються обмеження існуючих рішень, таких як відсутність гнучкості у налаштуванні та недостатня інтерактивність під час виправлення помилок. Наприклад, ESLint забезпечує автоматизоване виправлення помилок, але обмежує можливість індивідуалізації процесу, оскільки виправлення застосовуються автоматично, без урахування контексту проекту. Такий підхід може призвести до небажаних змін або невідповідностей, особливо у великих проектах із високою частотою оновлень та багатокомандною розробкою. Розділ акцентує на необхідності впровадження нових рішень для більш ефективного, індивідуального підходу до забезпечення якості коду, який враховував би специфіку кожного проекту.

Останній підрозділ розділу присвячений перевагам запропонованого рішення над існуючими інструментами. Акцент зроблено на важливості інтерактивних підказок, покроковому аналізі коду, що допомагає забезпечити

гнучкість, індивідуальність підходу та ефективність процесу лінтингу й рефакторингу. Використання інтерактивного режиму підказок дозволяє розробникам обирати з декількох варіантів виправлень, що забезпечує глибший контроль над якістю коду. Покроковий лінтинг також спрощує роботу з великими проектами, дозволяючи сфокусуватися лише на змінених частинах коду, що знижує навантаження на систему та економить час.

Отже, другий розділ роботи підтверджує актуальність автоматизованих інструментів для підтримки якості коду та виділяє потребу у вдосконаленні сучасних підходів. Пропонується інтеграція нових рішень, які здатні забезпечити більшу гнучкість, інтерактивність і продуктивність у процесах лінтингу та рефакторингу, що є важливим для ефективної роботи з масштабованими проектами та командної розробки.

РОЗДІЛ 3.
РЕАЛІЗАЦІЯ ЗАПРОПОНОВАНОГО РІШЕННЯ

3.1. Загальна архітектура та компоненти системи

Сучасні інструменти для аналізу коду, такі як ESLint і Prettier, значно полегшили роботу розробників, забезпечуючи автоматизовану перевірку синтаксису та форматування. Проте, вони часто мають обмеження, коли йдеться про гнучкість виправлень, інтеграцію з системами контролю версій або роботу з великими проектами. Наприклад, ESLint автоматично виправляє помилки без варіантів вибору, а Prettier застосовує жорсткі правила стилю, не враховуючи специфіку проекту. Щоб подолати ці обмеження, було створено своє рішення, що має назву «lintify». Розроблене розширення для vscode пропонує новий підхід до аналізу та оптимізації коду, об’єднуючи інтерактивність, адаптивність і гнучкість у одному інструменті.

Lintify – це розширення для VSCode [16], що об’єднує можливості лінтингу, форматування і рефакторингу, розроблене для того, щоб полегшити процес аналізу та оптимізації коду. На відміну від стандартних інструментів, які зосереджені лише на окремих аспектах кодування, Lintify пропонує комплексне рішення для всіх етапів роботи з кодом, забезпечуючи як автоматичну перевірку, так і інтерактивні підказки, доступні в режимі реального часу. Головна мета розширення – підвищити продуктивність та якість коду в JavaScript проектах, об’єднуючи всі основні функції аналізу та форматування під одним інтерфейсом.

Розширення включає кілька взаємозалежних компонентів, кожен з яких виконує свою важливу функцію: модуль лінтингу відповідає за базовий аналіз коду, інтерактивні підказки пропонують користувачеві вибір кращих виправлень, а покроковий аналіз дозволяє зосереджувати увагу лише на актуальних змінах. Така архітектура забезпечує гнучкість у процесі розробки,

а також можливість адаптації до великих проектів із численними файлами та складною логікою.

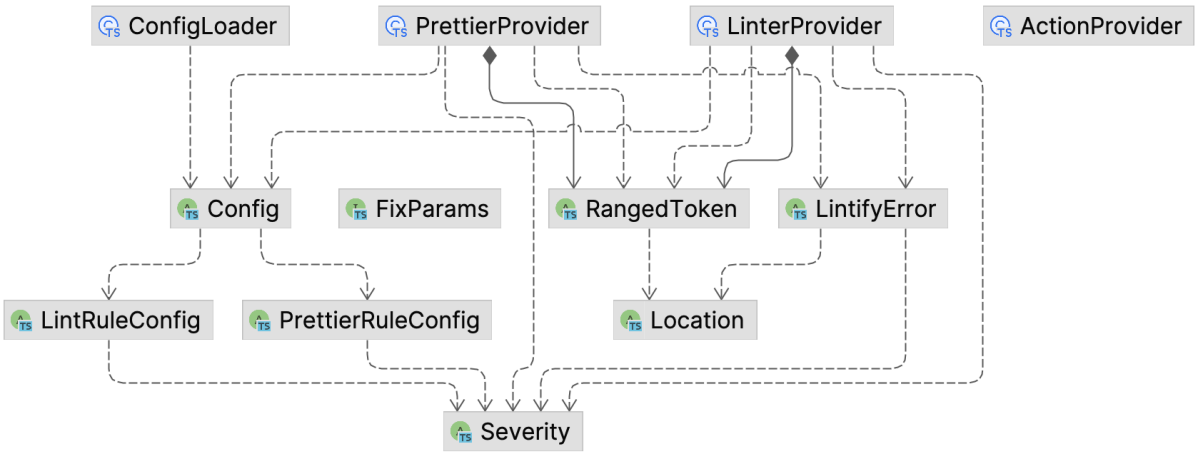

Рис. 3.1. – Функціональна схема (діаграма класів)

На рис. 3.1. представлена структура Lintify, де кожен компонент позначено як окремий блок зі своїм призначенням. Linter та Prettier працюють спільно, щоб забезпечити гнучкість вибору виправлень, у той час як покроковий аналіз (Step-by-step linting) дає змогу розробникам аналізувати лише вибрані частини коду, що значно підвищує продуктивність при роботі з великими кодовими базами.

Такий підхід робить Lintify не просто інструментом для перевірки коду, а комплексною системою підтримки якості, яка дозволяє командам розробників працювати з максимальною ефективністю. Далі ми розглянемо кожен з компонентів окремо та покажемо, як вони взаємодіють один з одним для досягнення основної мети розширення – зробити процес розробки коду більш простим, швидким і якісним.

3.1.1. Основні компоненти системи

У запропонованій системі аналізу та оптимізації коду використовуються кілька ключових компонентів, кожен з яких виконує свою важливу роль. Ці компоненти взаємодіють один з одним для забезпечення якісного аналізу та покращення коду в режимі реального часу. Опис кожного компонента

дозволить зрозуміти, як саме вони доповнюють один одного і сприяють загальній ефективності системи.

Модуль лінтингу

Лінтер є основним компонентом системи, відповідальним за аналіз коду. Він виконує перевірку синтаксичних помилок, логічних невідповідностей та порушень стилю коду. Модуль лінтингу працює з JavaScript файлами, забезпечуючи узгодженість коду з найкращими практиками розробки. Лінтер використовує правила ESLint та інші користувацькі налаштування, щоб автоматично перевіряти код при збереженні файлу або при виклику відповідної команди.

Модуль інтерактивних підказок

Цей модуль надає користувачам можливість взаємодіяти з виправленнями коду в інтерактивному режимі. На відміну від традиційних інструментів, які просто виправляють помилки автоматично, Hinting Mode пропонує кілька варіантів виправлень, дозволяючи розробникам не тільки швидко виправляти помилки, але й розуміти їхню природу, що підвищує загальну якість коду. Модуль взаємодіє з лінтером, щоб пропонувати виправлення на основі виявлених проблем.

Модуль покрокового лінтингу

Однією з головних відмінностей цієї системи є можливість покрокового лінтингу, що дозволяє розробникам аналізувати код не цілком, а по частинах. Він тісно взаємодіє з лінтером, передаючи йому тільки ті частини коду, які необхідно проаналізувати на поточному етапі, іншими словами – забезпечує можливість більш детальної роботи над певними частинами проекту, дозволяючи розробникам фокусуватися на окремих функціональних блоках, не відволікаючись на весь проект, що підвищує якість рефакторингу та знижує ймовірність виникнення нових помилок під час великих змін.

Модуль рефакторингу

Модуль рефакторингу забезпечує можливість виправляти помилки в коді та покращувати його архітектуру, оптимізуючи структуру без зміни

функціональності. Цей компонент може автоматично запропонувати поліпшення, наприклад, видалення зайвого коду, розбиття великих функцій на менші, або об’єднання дубльованих логічних блоків. Модуль рефакторингу також використовує інтерактивні підказки, дозволяючи розробнику самостійно вибирати, які покращення внести в код, щоб забезпечити коректність змін після внесення будь-яких поліпшень у код.

Модуль конфігурацій та налаштувань

Цей компонент дозволяє розробникам налаштовувати систему під свої індивідуальні потреби та відповідає за зберігання та застосування правил лінтингу, рефакторингу. Він працює за допомогою файлу конфігурацій. На рис. 3.2 представлено як працює алгоритм створення цього файлу. Модуль конфігурацій робить систему гнучкою та легкою в налаштуванні, дозволяючи інтегрувати її у будь-який проект без необхідності змінювати основні компоненти.

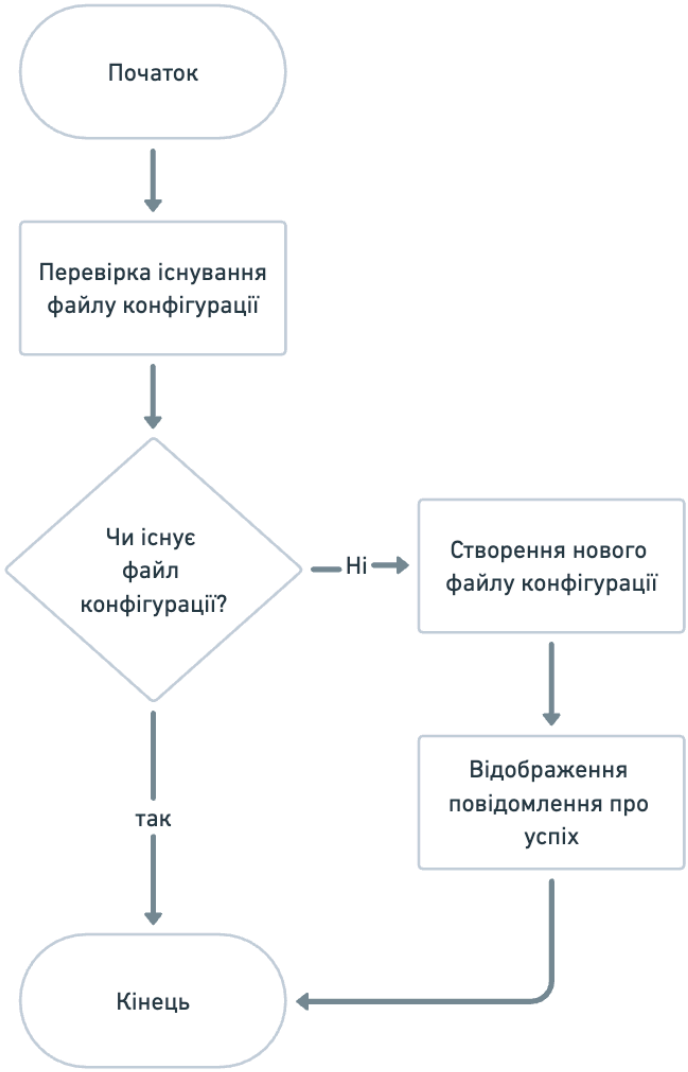

Рис. 3.2. – Блок-схема алгоритму створення файлу конфігурацій

Усі вищеописані компоненти системи працюють як окремі, незалежні частини, однак їхня справжня ефективність розкривається саме в процесі взаємодії один з одним. Лінтер, рефакторинг та інтерактивні підказки об’єднуються, щоб забезпечити динамічний і комплексний процес перевірки та покращення коду. Наприклад, лінтер ідентифікує проблеми, а Hinting Mode надає варіанти їхнього вирішення, після чого модуль рефакторингу може запропонувати додаткові покращення.

3.1.2. Взаємодія компонентів

Взаємодія компонентів Lintify є гнучкою та адаптивною, забезпечуючи повноцінну роботу системи в різних сценаріях використання. Нижче

розглянемо, як кожен з компонентів працює спільно з іншими при виконанні різних дій.

Аналіз коду під час роботи з файлом

Рис. 3.3. – блок-схема алгоритму аналізу коду під час роботи з файлом

Алгоритм працює таким чином: коли файл змінюється, система автоматично запускає аналіз коду за допомогою лінтера. Лінтер перевіряє код на наявність проблем, таких як помилки або порушення стилю. Якщо ніяких

проблем не знайдено, процес завершується. Але якщо в коді є проблеми, система передає результати аналізу в Hinting Mode. У цьому режимі користувач бачить підказки та варіанти для виправлення цих проблем, що дозволяє йому вирішувати, як діяти далі.

Рефакторинг і покращення коду на основі інтерактивних підказок

Рис. 3.4 - блок-схема алгоритму рефакторингу коду на основі інтерактивних підказок

Якщо розібрати алгоритм рефакторингу коду на основі інтерактивних підказок, то можемо бачити, що він працює наступним чином: користувач починає процес аналізу коду, наприклад, викликавши відповідну команду. Лінтер сканує код активного файлу або всього проекту та визначає, чи є помилки або проблеми зі стилем. Якщо проблем немає, система завершує процес і повідомляє, що код чистий. У випадку, коли помилки знайдені, система переходить у Hinting Mode. У цьому режимі для кожної помилки генерується список можливих виправлень. Користувачу надаються підказки та варіанти, як саме можна виправити помилку. Користувач може обрати автоматичне виправлення, пропустити помилку або відкласти її виправлення на пізніше. Якщо обрано виправлення, система застосовує його до коду, оновлює файл і зберігає історію виправлень. Якщо користувач вирішує пропустити помилку, алгоритм переходить до аналізу наступної. Процес триває, доки всі знайдені помилки не будуть оброблені. Після цього система повідомляє про завершення роботи, а код оновлюється відповідно до внесених змін.

Завдяки такій інтегрованій взаємодії, Lintify забезпечує повний цикл аналізу, оптимізації та форматування коду. Кожен з компонентів відіграє свою унікальну роль, водночас доповнюючи роботу інших модулів, що дозволяє системі працювати цілісно і ефективно.

3.2. Використовувані модулі та бібліотеки

Розробка Lintify потребує детального підходу до вибору технологій і компонентів, що забезпечують стабільну роботу розширення, його інтеграцію із середовищем VSCode, а також гнучкість у налаштуваннях під різні проекти. Функції аналізу, рефакторингу та інтерактивного виправлення базуються на кількох важливих модулях і бібліотеках, які складають архітектурну основу системи. Кожен із компонентів дозволяє реалізувати специфічні аспекти роботи розширення, роблячи його адаптивним і продуктивним у застосуванні.

Мова программування

Через те, що розроблюваний продукт представляє собою розширення для VSCode, для повноцінної роботи потрібно взаємодіяти з VSCode API. У свою чергу, для взаємодії з VSCode API, можна вікористовувати мови програмування JavaScript та TypeScript [17]. Порівнюючи ці два варіанти слід звернути увагу на швидкість самої розробки і читаність коду. Особливу увагу слід приділити тому, як TypeScript полегшує роботу з такими інструментами, як Esprima, що використовується в проєкті для створення абстрактного синтаксичного дерева (AST). Завдяки можливості визначати типи для вузлів дерева та функцій, які взаємодіють з ними, можна досягти високої точності аналізу. Це, своєю чергою, сприяє стабільній роботі алгоритмів, навіть коли вони обробляють складні конструкції JavaScript.

Також варто зазначити, що насьогодні існує активна підтримка TypeScript спільнотою розробників. Постійний розвиток мови забезпечує доступ до найновіших інструментів і бібліотек, що робить TypeScript перспективним вибором для довгострокових проєктів, робить TypeScript основою стабільної, масштабованої та зручної у використанні системи аналізу коду.

VSCode API

Використання VSCode API є важливим аспектом інтеграції Lintify у середовище розробки, що забезпечує доступ до широкого набору функціональних можливостей для створення команд, налаштувань, а також взаємодії з користувацьким інтерфейсом. Основною перевагою використання цього API є можливість глибокої інтеграції з VSCode: Lintify може автоматично запускати лінтер при збереженні файлу, відображати результати перевірки безпосередньо в редакторі (підсвічування помилок, попереджень) і дозволяти користувачеві швидко застосовувати виправлення або змінювати форматування коду за допомогою контекстного меню або гарячих клавіш.

За допомогою VSCode API створюється інтерфейс інтерактивних підказок (Hinting Mode), що відображає варіанти виправлень у вигляді вікон підказок, де користувач може обрати оптимальне рішення для виправлення конкретної

помилки. API також надає можливість роботи з командною палітрою, що спрощує доступ до функцій Lintify. Наприклад, користувач може запускати команди для перевірки або форматування коду через швидкі команди VSCode, що значно підвищує зручність роботи з розширенням.

Крім того, VSCode API дозволяє зберігати та управляти налаштуваннями для кожного проекту, що робить Lintify адаптивним для різних середовищ та проектних стандартів. Це особливо важливо для командних проектів, де необхідно забезпечити відповідність коду певним стилістичним стандартам або вимогам, встановленим для команди. Наприклад, користувач може налаштувати підсвічування на різні кольори для помилок і попереджень, а також змінити параметри виправлення відповідно до специфіки проекту.

Робота з AST

Парсери та AST-генератори є ключовими для роботи з деревом синтаксичного розбору (AST), що надає доступ до внутрішньої структури коду.

Для розробки було обрано Esprima [18], адже парсер модуль дозволяє створювати не тільки ast дерево, але і виконувати різні дії безпосередньо над списком токенів. Це дозволяє скоротити час для деяких алгоритмів: не потрібно даремно витрачати час для проходження по дереву, адже можна просто віконувати дії над списком токенів всього оброблюваного коду

Завдяки використанню AST парсера Lintify може не лише виконувати базовий лінтинг, а й виявляти складні логічні помилки, які важко помітити на рівні окремих рядків коду. AST дозволяє глибоко аналізувати код і виявляти специфічні порушення, які можуть стосуватися не тільки стилістики, але й більш широких проблем на рівні модулів чи навіть компонентів.

Використання AST також дає можливість рефакторингу – наприклад, модуль може ідентифікувати й видаляти дубльований код, знаходити великі функції та пропонувати їхній поділ на менші, більш керовані блоки. У процесі роботи з AST Lintify може перевіряти залежності між функціями, ідентифікувати дубльовані вирази та розглядати варіанти оптимізації, зокрема видалення непотрібного або застарілого коду. Це дає розробникам більш

детальний контроль над тим, як виглядає структура проекту, і дозволяє підтримувати його на високому рівні якості.

Path модулі

Для роботи з файловою системою та шляхами використовується потужний набір інструментів, зокрема модулі FS та Path [19]. Вони дозволяють швидко й надійно зчитувати файли проекту, змінювати та зберігати їх безпосередньо в середовищі VSCode, де працює розширення. Це значно підвищує продуктивність розробника, оскільки всі зміни можна вносити у реальному часі. FS- та Path-модулі надають можливість автоматизувати перевірку і внесення виправлень у файли, що спрощує інтеграцію з іншими компонентами Lintify.

Наприклад, використовуючи модуль FS, розширення може зберігати файл автоматично після виправлення або запускати лінтер при кожному збереженні. Це також дозволяє безпечно інтегруватися з іншими командами у VSCode, полегшуючи налаштування автоматичного збереження та миттєвого аналізу файлів. Path, у свою чергу, забезпечує простоту роботи з шляхами до файлів, дозволяючи застосовувати Lintify до файлів у будь-якому розташуванні проекту. Це робить розширення адаптивним до різних структур проектів, що особливо важливо для великих кодових баз.

Модулі для налаштування та конфігурації

Модуль налаштувань відіграє ключову роль у забезпеченні гнучкості Lintify, оскільки дозволяє розробникам адаптувати розширення під специфічні вимоги проекту. У цьому модулі зберігаються параметри конфігурації, які визначають правила лінтингу, стилі форматування та поведінку інтерактивних підказок. Кожен проект може мати індивідуальні вимоги щодо стилю коду, структурних елементів і перевірок, і завдяки модулю налаштувань користувачі можуть швидко адаптувати розширення під ці потреби.

Рис. 3.5. – Приклад налаштувань файлу конфігурацій

Наприклад, користувач може вказати конкретні стилістичні правила для JavaScript чи TypeScript, встановити специфічні вимоги до форматування, такі як кількість відступів або порядок імпортів, а також налаштувати підсвічування для різних видів помилок. Модуль конфігурації зберігає ці налаштування у вигляді конфігураційних файлів, що робить їх легкодоступними для модифікацій.

Крім того, модуль конфігурації дозволяє налаштовувати поведінку інтерактивних підказок, наприклад, які саме підказки будуть відображатися, а які виправлення будуть виконуватися автоматично. Це дозволяє створювати гнучке середовище, яке підтримує особисті налаштування розробника або вимоги команди. Такий підхід підвищує зручність користування Lintify і забезпечує його інтеграцію у різні робочі процеси без зайвих зусиль.

Модулі для створення інтерфейсу підказок

Модуль підказок дозволяє інтегрувати інтерактивні елементи в редактор VSCode, забезпечуючи зручний інтерфейс для взаємодії з виправленнями та рекомендаціями. Це критичний компонент Hinting Mode, який надає розробнику кілька варіантів виправлень або оптимізацій, дозволяючи обирати найбільш відповідні рішення. Наприклад, коли лінтер знаходить потенційні помилки або стилістичні порушення, модуль підказок генерує вікна з варіантами виправлень, де користувач може обрати оптимальний варіант або залишити код без змін.

Підказки також включають пояснення до кожного запропонованого рішення, що допомагає розробникам краще розуміти логіку виправлення і мотиви його застосування. Це особливо корисно для навчання нових членів команди або для підтримки консистентності підходів до кодування. Крім того, цей модуль може бути налаштований так, щоб підказки враховували специфіку проекту – наприклад, специфічні стилістичні або архітектурні вимоги. Це робить роботу з розширенням ще більш зручною і продуктивною, оскільки дозволяє налаштовувати систему під конкретні вимоги без втрати функціональності.

Разом ці модулі та бібліотеки формують гнучку, масштабовану й інтегровану структуру Lintify, яка дозволяє ефективно вирішувати завдання лінтингу, форматування та рефакторингу у середовищі VSCode. Кожен компонент системи працює як окремий блок, одночасно доповнюючи інші модулі, що забезпечує якість і продуктивність розширення. Це робить Lintify потужним інструментом для командного і індивідуального використання,

забезпечуючи високий рівень контролю над кодом і можливість гнучкого налаштування для будь-яких проектів.

3.3. Опис основних функцій системи

3.3.1. Логіка надання підказок і взаємодія з користувачем

Логіка надання підказок у системі Lintify є одним із центральних елементів, що надає користувачу контроль над процесом оптимізації та виправлення коду. Замість автоматичного внесення змін, як це відбувається у традиційних інструментах лінтингу, Lintify використовує підхід з інтерактивними підказками. Такий підхід надає розробникам можливість обирати варіанти виправлень для конкретних проблем, а також отримувати пояснення щодо кожного варіанту. Це значно підвищує ефективність роботи, оскільки користувач не лише виправляє помилки, а й розуміє причини їх виникнення та можливі наслідки різних підходів до виправлення.

Процес надання підказок у Lintify починається з аналізу коду лінтером, який ідентифікує потенційні проблеми. Після виявлення помилок система активує Hinting Mode, що генерує список варіантів виправлень з поясненнями до кожного з них. Це може включати як стилістичні зміни, так і більш глибокі виправлення, що впливають на логіку програми.

Інтерактивна природа Hinting Mode дозволяє розробнику не просто приймати виправлення, а й обирати між кількома можливими варіантами, кожен з яких має свої переваги. Наприклад, якщо у коді є стилістичні невідповідності, система може запропонувати варіанти виправлення з поясненням, чому саме цей підхід є кращим для певного проекту або команди. Це забезпечує не тільки чистоту коду, але й підтримку єдиного стилю, узгодженого з вимогами проекту. У великих командах така функція особливо корисна, адже вона дозволяє всім розробникам дотримуватися загальноприйнятих стандартів.

Крім вибору виправлень, Hinting Mode надає користувачам гнучкість у налаштуванні підказок, що дозволяє зробити систему максимально адаптованою до специфіки проекту. Користувач може налаштувати вид підказок та їхню інтенсивність – наприклад, отримувати сповіщення лише для критичних помилок або включати детальне пояснення для кожного варіанту виправлення.

Hinting Mode також підтримує збереження історії обраних виправлень, що дозволяє розробникам легко відстежувати внесені зміни та розуміти мотивацію попередніх виправлень. Це корисно як для нових членів команди, так і для досвідчених розробників, оскільки забезпечує зворотний зв’язок і допомагає уникати повторення тих самих помилок у майбутньому. Історія виправлень стає свого роду документом, який можна використовувати для аналізу та оптимізації підходів до розробки в цілому.

Таким чином, Hinting Mode виступає не лише інструментом для усунення помилок, але й засобом підвищення професійної культури команди. Він дає можливість кожному розробнику вносити свій внесок у підтримку стилю та якості коду, одночасно забезпечуючи легкий і гнучкий процес виправлення проблем.

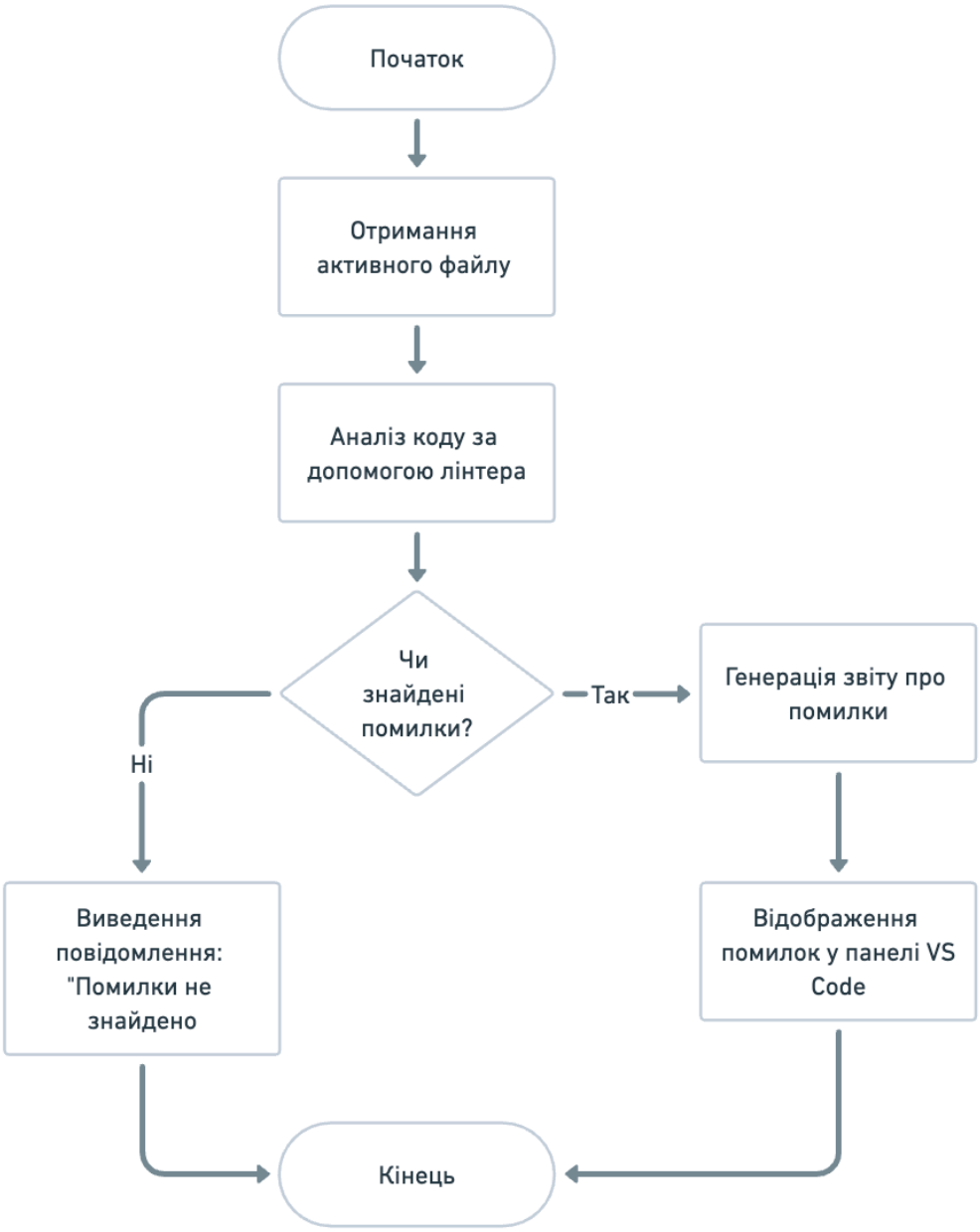

Рис. 3.6 – блок-схема роботи алгоритму Hinting Mode

3.3.2. Процес покрокової перевірки

Процес покрокової перевірки є однією з важливих функцій Lintify, що дозволяє розробникам ефективно працювати з великими кодовими базами, оптимізуючи перевірку коду. Замість аналізу всього проекту або великих файлів одразу, система пропонує можливість перевірки окремих фрагментів або блоків коду. Це дозволяє значно знизити навантаження на систему та прискорити процес аналізу, що особливо корисно для великих проектів з численними модулями та складною архітектурою.

У покроковому режимі розробник може вибрати певну ділянку коду – наприклад, функцію, клас або навіть кілька рядків – для здійснення лінтингу. Система виконує перевірку тільки для обраного блоку, ідентифікуючи синтаксичні помилки, стилістичні невідповідності та можливі проблеми саме в межах цієї ділянки. Такий підхід не тільки забезпечує фокусування на конкретній частині коду, але й дозволяє виявляти помилки поступово, зберігаючи при цьому узгодженість коду.

Покроковий аналіз особливо ефективний для команд, які працюють над проектами з розгалуженою структурою, де зміни часто вносяться в окремі модулі або функціональні блоки. За допомогою цього режиму розробники можуть сфокусуватися на певних частинах проекту, залишаючи стабільні частини коду поза процесом аналізу. Це знижує ризик порушення стабільності великого проекту і забезпечує швидший зворотний зв’язок для розробника, дозволяючи йому одразу побачити результати виправлень.

Крім того, покрокова перевірка дозволяє ефективно впроваджувати рефакторинг у великі проекти. Наприклад, якщо необхідно оптимізувати роботу певної функції, можна вибрати її для перевірки, внести необхідні зміни, а потім знову перевірити код для виявлення можливих конфліктів чи невідповідностей. Це дозволяє покращувати код поступово, зберігаючи контроль над якістю проекту і зменшуючи ймовірність виникнення помилок під час масштабних змін.

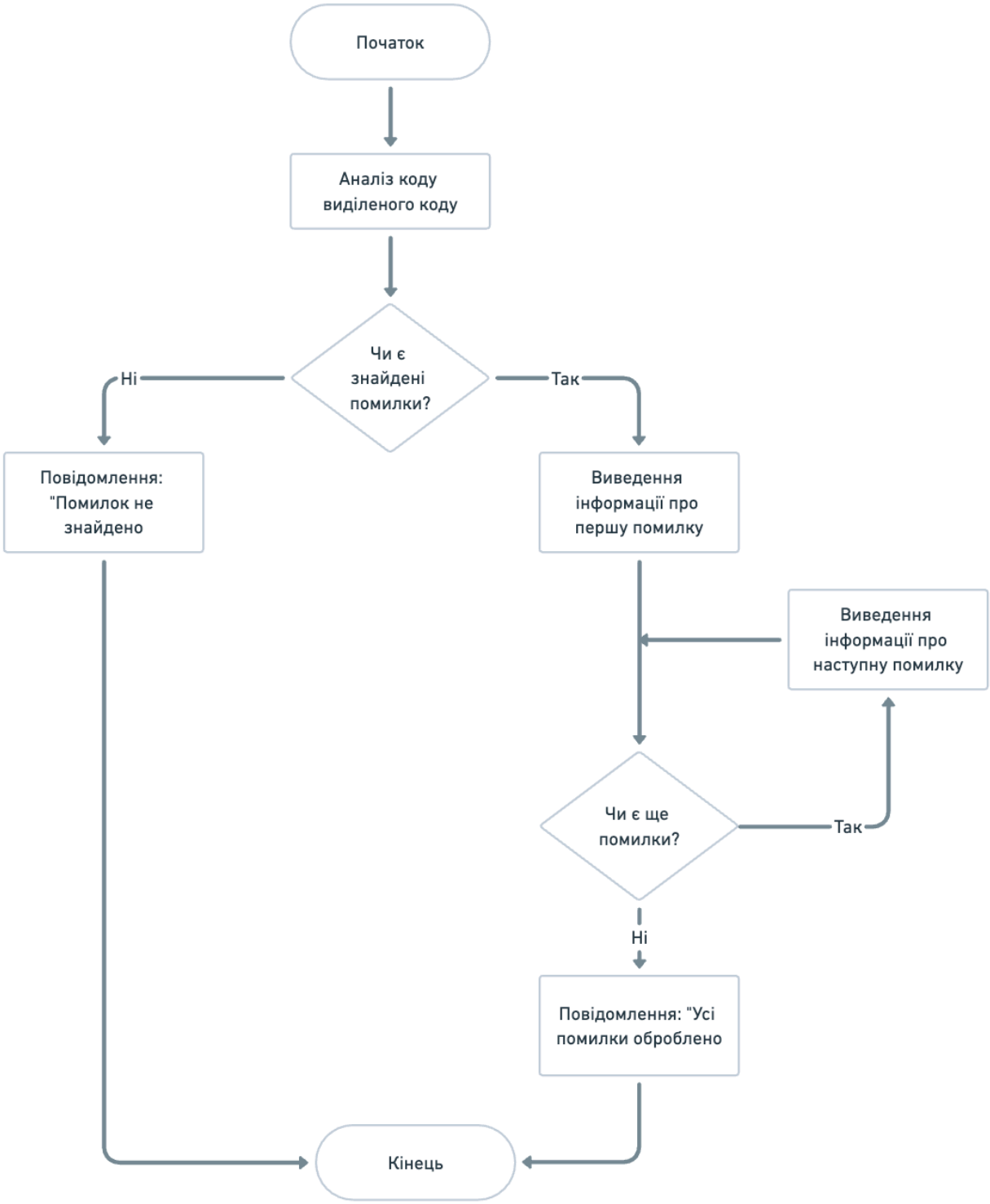

Рис 3.7 – блок-схема роботи алгоритму step-by-step linting

Процес покрокового аналізу коду в системі Lintify починається з того, що розробник вибирає режим покрокового аналізу. У цьому режимі користувач може аналізувати окремі фрагменти коду, не охоплюючи весь файл або проект, що особливо корисно при роботі з великими кодовими базами.

Після вибору фрагмента, система запускає лінтер для аналізу цього блоку коду. Лінтер перевіряє, чи є в обраному фрагменті коду синтаксичні або стилістичні помилки. Якщо помилки не виявлені, процес завершується. Якщо ж помилки є, система переходить до наступного етапу – інтерактивного режиму підказок, де користувачеві пропонуються варіанти виправлень для кожної знайденої помилки.

На етапі Hinting Mode розробник має можливість вибрати один з варіантів виправлення. Якщо користувач обирає виправлення, система застосовує його до коду і оновлює фрагмент. Якщо розробник вирішує не застосовувати жодне виправлення, він може пропустити цей крок або зберегти код без змін.

Після внесення виправлень система проводить подальшу перевірку для переконання, що нові зміни відповідають загальним стандартам проекту. Завершальним етапом є рефакторинг і перевірка оновленого фрагмента коду на можливі конфлікти або порушення стилю.

Процес завершується після успішної оптимізації вибраного фрагмента коду, і внесені зміни застосовуються до проекту.

3.3.3. Процес автоматичного застосування виправлень

Також користувачу доступна функція автоматичного виправлення, що робіть розробку більш швидшою, коли річ заходить про виправлення незначних помилок у коді, задля збереження стилістики. Алгоритм автоматичного виправлення зменшує час на написання коду, а також дозволяє користувачеві не відволікатись на маленькі стилістичні помилки. Налаштувавши файл конфігурацій один раз, надалі можна не звертати уваги на відсутність крапки з комою у кінці виразу, або різний стиль лапок – при застосуванні автоматичного виправлення, такі помилки більше не будуть звертати на себе увагу користувача і він зможе зосередитися на опрацюванні логіки роботи програми. На рис. 3.8. можемо побачити блок-схему цього алгоритму.

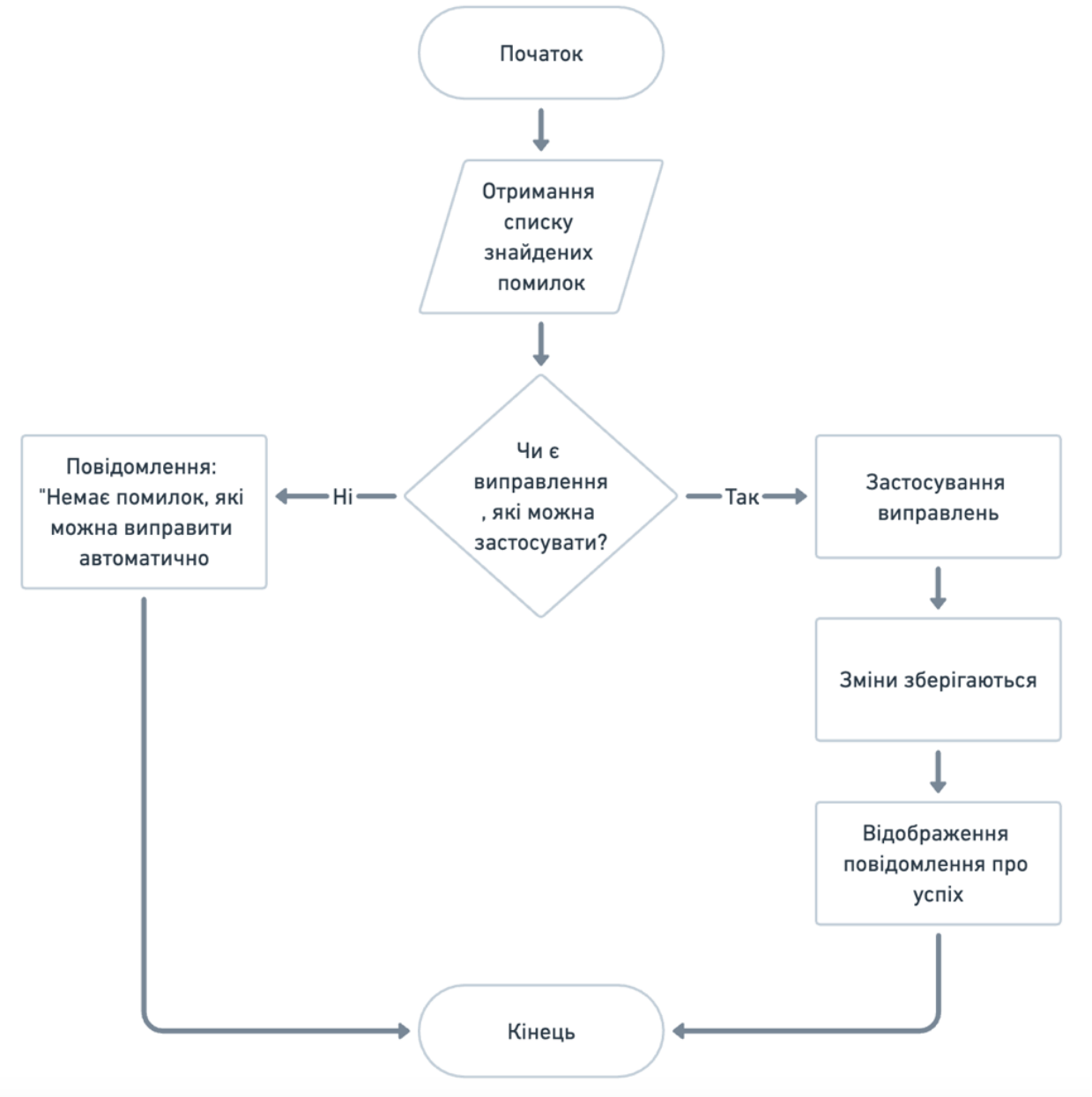

Рис. 3.8 – блок-схема алгоритму автоматичного виправлення

Ось як він працює:

Коли процес починається, система отримує список знайдених помилок у коді. Після цього перевіряється, чи є серед знайдених помилок такі, які можна автоматично виправити. Якщо автоматичних виправлень немає, система повідомляє користувача про це, виводячи повідомлення: «Немає помилок, які можна виправити автоматично», і процес завершується. Якщо ж автоматичні виправлення доступні, система застосовує їх до коду. Після цього зміни зберігаються, і користувач отримує повідомлення про успішне внесення виправлень. Потім процес завершується.

ВИСНОВОК ДО РОЗДІЛУ 3

У третьому розділі магістерської роботи докладно розглянуто процес реалізації запропонованого розширення Lintify для аналізу та оптимізації коду. Основну увагу приділено структурі, компонентам та взаємодії між різними модулями системи, що забезпечують інноваційний підхід до підтримки якості коду. Описано як архітектурні аспекти, так і специфічні особливості компонентів, які забезпечують гнучкість, масштабованість та інтеграцію з інструментами, що вже використовуються в командних проектах.

На початку розділу увагу акцентовано на ключових компонентах, що відповідають за аналіз, рефакторинг та забезпечення консистентності коду. Такий підхід забезпечує індивідуальний підхід до аналізу коду, що дозволяє зосередитися на змінених частинах коду, знижуючи навантаження на систему та спрощуючи робочий процес для розробників.

У підрозділі про інтерактивні підказки для покращення коду наголошено на тому, що користувачі мають змогу обирати оптимальні варіанти виправлень, що підвищує точність та якість внесених змін. Це є особливо важливим для командної роботи, де кожен член команди може використовувати різні підходи до виправлення коду. Таким чином, інтерактивність Hinting Mode забезпечує гнучкий і персоналізований підхід до рефакторингу.

Особливу увагу в розділі також приділено процесу покрокового аналізу коду для полегшення роботи з великими проектами. Замість того, щоб перевіряти всю кодову базу, Lintify дозволяє розробникам працювати з окремими ділянками коду, що не лише підвищує ефективність процесу лінтингу, а й дає змогу зосередитися на пріоритетних завданнях.

Отже, третій розділ підтверджує, що запропоноване рішення Lintify здатне забезпечити ефективну взаємодію між компонентами, інтеграцію з існуючими інструментами розробки та високий рівень гнучкості, що робить його оптимальним вибором для великих командних проектів. Lintify не лише сприяє підтримці консистентності коду, але й забезпечує продуктивність,

інтерактивність та підтримуваність коду, що відповідає сучасним вимогам до програмного забезпечення.

РОЗДІЛ 4.
РОЗРОБКА СТАРТАП ПРОЕКТУ

4.1. Опис ідеї проєкту

Основна ідея проєкту полягає у створенні унікального інструменту, що має назву «Lintify», для спростити процес аналізу та оптимізації JavaScript коду, підвищуючи його якість, а також роботу команд розробників.

Таблиця 4.1.

Опис ідеї стартап-проєкту

Зміст ідеї	Напрямки застосування	Вигоди для користувача
Система аналізу та оптимізації програмного коду	Аналіз та оптимізація JavaScript коду	Підвищення продуктивності розробників
	Спрощення співпраці в командних проєктах	Зниження ризику помилок у коді
	Впровадження якісних стандартів кодування	Покращення читаємості та підтримуваності коду

У таблиці 4.2. визначено показники, що мають а) гірші значення (W, слабкі); б) аналогічні (N, нейтральні) значення; в) кращі значення (S, сильні)

Таблиця 4.2.

Визначення сильних, слабких та нейтральних характеристик ідеї проеєкту

№	Техніко-економічні характеристики ідеї	(потенційні) товари/концепції конкурентів			Оцінка
		Мій проєкт	Конкурент 1 (ESLint)	Конкурент 2 (Prettier)	

Продовження таблиці 4.2

1	Інтерактивні підказки	Так	Ні	Ні	S
2	Покроковий лінтинг	Так	Ні	Ні	S
3	Легкість інтеграції у VSCode	Так	Так	Так	N
4	Автоматизоване виправлення помилок	Частково	Так	Так	N

4.2. Технологічний аудит ідеї проєкту

Таблиця 4.3.

Технологічна здійсненність ідеї проекту

№	Ідея проєкту	Технології її реалізації	Наявність технологій	Доступність технологій
1	Система аналізу та оптимізації програмного коду	Мова програмування TypeScript	Наявна	Доступна безкоштовно
2		Інструмент роботи з AST - esprima	Наявна	Доступна безкоштовно

Продовження таблиці 4.3.

3		VSCode API	Наявна	Доступна безкоштовно
4		Утиліта yoman для генерації VSCode розширення	Наявна	Доступна безкоштовно
5		Середовище розробки WebStorm	Наявна	Доступна безкоштовно з базовим функціоналом
6		VSCode debugger	Наявна	Доступна безкоштовно

4.3. Аналіз ринкових можливостей запуску стартап-проєкту

Ринок розробницьких інструментів для аналізу та форматування коду є стабільно зростаючим. Основними користувачами таких рішень є розробники, які працюють із мовою JavaScript. Впровадження додаткових функцій, таких як інтерактивні підказки, створює додаткову цінність і може збільшити зацікавленість серед професійної аудиторії. У таблиці 4.4 представлена попередня характеристика потенційного ринку для розроблюваного стартап-проєкту.

Таблиця 4.4.

Попередня характеристика потенційного ринку стартап-проєкту

№	Показники стану ринку	Характеристика
1	Кількість головних гравців	3-5 великих конкурентів (ESLint, Prettier)

Продовження таблиці 4.4.

2	Загальний обсяг продаж	>10 млн завантажень у VSCode Marketplace
3	Динаміка ринку	Зростає
4	Наявність обмежень для входу	Низька. Головна вимога – якісний продукт
5	Специфічні вимоги до сертифікації	Не потрібні
6	Середня норма рентабельності	20-30% (за рахунок платних функцій чи підтримки)

Таблиця 4.5.

Характеристика потенційних клієнтів стартап-проєкту

№	Потреба, що формує ринок	Цільова аудиторія	Відмінності у поведінці різних цільових груп	Вимоги споживачів до товару
1	Підвищення якості коду	Індивідуальні розробники	Висока орієнтація на простоту використання	Зрозумілий інтерфейс, мінімальні налаштування
2	Забезпечення єдиного стилю в команді	Команди розробників	Орієнтація на інтеграцію в CI/CD, спільне налаштування	Гнучкість у конфігураціях, швидкість аналізу
3	Покращення продуктивності робочих процесів	Технічні лідери, менеджери команд	Високі вимоги до стабільності та оновлень	Надійність, підтримка документації

Таблиця 4.6.

Фактори загроз

№	Фактор	Зміст загрози	Можлива реакція компанії
1	Домінування ESLint і Prettier	Висока популярність конкурентів серед розробників	Розробка унікальних функцій (Hinting Mode, Step-by-step linting)
2	Безкоштовні аналоги	Більшість користувачів віддають перевагу безкоштовним рішенням	Запропонувати безкоштовну версію з базовим функціоналом
3	Низький бар’єр для входу	Легкість створення аналогічного продукту	Позиціонування як інноваційного інструменту з унікальними можливостями
4	Низька обізнаність про продукт	Обмеженість ресурсів для просування нового інструменту	Використання партнерств та колаборацій із лідерами думок

Таблиця 4.7.

Фактори можливостей

№	Фактор	Зміст можливості	Можлива реакція компанії
1	Зростання популярності TypeScript	Потреба у спеціалізованих інструментах для TS	Акцент на оптимізації роботи з TypeScript
2	Потреба у покращеній інтерактивності	Попит на інструменти, що спрощують навчання та роботу з кодом	Створення функцій Hinting Mode для інтерактивного UX

Продовження таблиці 4.7.

3	Інтеграція з середовищами розробки	Популярність VSCode як основного редактора для JS/TS	Глибока інтеграція з VSCode API
4	Зростання ринку DevTools	Підвищення уваги до інструментів, що забезпечують продуктивність	Активне позиціонування як інноваційного продукту
5	Відкритий доступ до технологій	Безкоштовність інструментів для розробки розширень	Використання перевірених технологій для швидкого запуску

Таблиця 4.8.

Ступеневий аналіз конкуренції на ринку

№	Особливості конкурентного середовища	У чому проявляється дана характеристика	Вплив на діяльність підприємства (можливі дії компанії, щоб бути конкурентоспроможною)
1	Тип конкуренції – Олігополія	Декілька провідних гравців (ESLint, Prettier) займають значну частину ринку завдяки високій якості та надійності своїх рішень	Для конкуренції необхідно запропонувати інноваційні функції, які недоступні в конкурентних продуктах, наприклад, Hinting Mode та Step-by-step linting

Продовження таблиці 4.8.

2	Рівень конкурентної боротьби – локальний	Основна конкуренція відбувається у межах VSCode Marketplace, де продукти борються за увагу розробників через відгуки, завантаження та оцінки	Необхідно забезпечити високу якість продукту з самого початку, створити привабливу документацію, активну комунікацію у спільнотах розробників та партнерства з лідерами думок
3	Галузева конкуренція – Внутрішньогалузева	Інструменти для JavaScript та TypeScript активно змагаються за користувачів, оскільки ці мови є найбільш популярними серед веб-розробників	Орієнтація на підтримку TypeScript, який швидко зростає у популярності, дозволить залучити розробників, що шукають спеціалізовані рішення
4	Конкуренція за видами товарів – Товарно-видова	Продукти для лінтингу (ESLint), форматування (Prettier) та їхні розширення змагаються у забезпеченні ключових потреб розробників	Необхідно запропонувати комплексний інструмент, який інтегрує лінтинг, форматування та унікальні функції, що знижують потребу у використанні кількох окремих рішень

Продовження таблиці 4.8.

5	Характер конкурентних переваг – Нецінова	Конкуренція базується на якісному UX/UI, продуктивності та зручності використання інструментів	Вкладення ресурсів у покращення інтерфейсу, інтерактивності (Hinting Mode), швидкості та точності лінтингу забезпечить конкурентну перевагу
6	Інтенсивність конкуренції – Марочна	ESLint і Prettier є впізнаваними брендами, яким довіряє більшість розробників, що ускладнює вихід нових продуктів	Створення бренду з чітким позиціонуванням (Lintify), акцент на унікальності продукту та партнерства з великими проєктами для підвищення впізнаваності

Таблиця 4.9.

Аналіз конкуренції в галузі за М. Портером

Складові аналізу	Прямі конкуренти в галузі	Потенційні конкуренти	Постачальники	Клієнти	Товари-замінники
	ESLint, Prettier	Нові інструменти для лінтингу /форматування, що можуть з’явитися на ринку	Microsoft (VSCode API), бібліотеки esprima, TypeScript	Індивідуальні розробники, команди розробників, технічні лідери	Окремі утиліти для лінтингу (ESLint) або форматування (Prettier)
Висновки	Висока конкуренція через популярність та впізнаваність брендів	Низький бар’єр входу на ринок дозволяє з’являтися новим гравцям	Постачальники є доступними, що забезпечує технічну стабільність	Клієнти орієнтуються на безкоштовний продукт із простим інтерфейсом	Замінники не забезпечують інтегрованого підходу (лінтинг + форматування)

Таблиця 4.10

Обґрунтування факторів конкурентоспроможності

№	Фактор конкурентоспроможності	Обґрунтування
1	Унікальність функціоналу	Продукт пропонує інтерактивний режим Hinting Mode і Step-by-step linting, які дозволяють користувачам покращувати код інтерактивно та поступово. Це функції, яких немає у конкурентів.
2	Інтеграція з популярними платформами	Глибока інтеграція з VSCode через його API забезпечує сумісність із найпоширенішим середовищем розробки для JavaScript.
3	Гнучкість використання	Продукт дозволяє працювати як із невеликими блоками коду, так і з цілими проєктами, завдяки пошаговому лінтингу, що відповідає вимогам розробників.
4	Простота доступу та використання	Продукт буде розповсюджуватися через VSCode Marketplace безкоштовно з базовим функціоналом, що знижує бар’єр для залучення клієнтів.
5	Сучасність технологій	Використання актуальних бібліотек та мови TypeScript забезпечує стабільність і високу продуктивність роботи продукту.
6	Фокус на потребах розробників	Продукт орієнтований на розробників JS, забезпечуючи необхідну функціональність для аналізу, форматування та оптимізації коду відповідно до потреб клієнтів.

Таблиця 4.11.

Порівняльний аналіз сильних та слабких сторін

№	Фактор конкурентоспромож-ності	Бали 1-20	Рейтинг товарів-конкурентів у порівнянні з розроблюваним продуктом						
			-3	-2	-1	0	+1	+2	+3
1	Унікальність функціоналу	16			+				
2	Інтеграція з популярними платформами	15				+			
3	Гнучкість використання	12					+		
4	Простота доступу та використання	15			+				
5	Сучасність технологій	16			+				
6	Фокус на потребах розробників	13		+					

Таблиця 4.12.

SWOT- аналіз стартап-проекту

Сильні сторони	Слабкі сторони
– унікальність функціоналу; – інтеграція з популярними платформами; – гнучкість використання; – простота доступу та використання; – сучасність технологій; – фокус на потребах розробників;	– відсутність впізнаваного бренду порівняно з конкурентами; – велика кількість конкурентів; – обмежені ресурси на маркетингову кампанію; – вузький фокус лише на JS;

Продовження таблиці 4.12.

Можливості	Загрози
– попит на інтерактивні інструменти серед розробників; – можливість інтеграції з іншими devtools у майбутньому; – розширення функцыональності для TS;	– зростання кількості нових конкурентів; – швидкий розвиток технологій, що може зменшити актуальність продукту;

Таблиця 4.13.

Альтернативи ринкового впровадження стартап-проекту

№	Альтернатива (орієнтовний комплекс заходів) ринкової поведінки	Ймовірність отримання ресурсів	Строки реалізації
1	Активна реклама у популярних спільнотах розробників (форумів, конференцій, соціальних мереж)	Висока	Протягом 3-6 місяців
2	Позиціонування як інноваційного рішення з унікальними функціями (Hinting Mode, Step-by-step linting)	Середня	Протягом 6-9 місяців
3	Розробка безкоштовної версії з базовими функціями та платної – з розширеними можливостями	Висока	Протягом 12 місяців
4	Розширення функціоналу для роботи з TypeScript, інтеграція додаткових DevTools	Середня	12-18 місяців

4.4. Розроблення ринкової стратегії проекту

Таблиця 4.14.

Вибір цільових груп потенційних споживачів

№	Опис профілю цільової групи потенційних клієнтів	Готовність споживачів сприйняти продукт	Орієнтовний попит в межах цільової групи (сегменту)	Інтенсивність конкуренції в сегменті	Простота входу у сегмент
1	Індивідуальні розробники	Готові сприйняти	Продукт необхідний зараз	Середня	Простий вхід
2	Командні проєкти розробників	Не всі користувачі готові сприйняти	Не всі зацікавлені в продукті	Висока	Простий вхід
3	Приватні ІТ-компаії	Не всі компанії готові сприйняти	Не всі компанії зацікавлені в продукті	Висока	Складний вхід
Які цільові групи обрано: Обрано групу індивідуальних розробників, тому що інтенсивність конкуренції найбільша в сегменті команд та в приватні ІТ-компаніях, але простота інтеграції продукту дає перевагу для залучення клієнтів, тому можна працювати зі всіма цільовими групами.					

Таблиця 4.15.

Визначення базової стратегії розвитку

№	Обрана альтернатива розвитку проєкту	Стратегія охоплення ринку	Ключові конкурентоспроможні позиції відповідно до обраної альтернативи	Базова стратегія розвитку
1	Розширення функціоналу під TypeScript	Зосередження зусиль на вузькому сегменті розробників, які активно використовують TypeScript. Такий підхід дозволяє максимально адаптувати продукт до специфічних потреб цього сегмента, виділяючи його на фоні конкурентів.	Фокус на задоволенні потреб TypeScript-розробників, які потребують спеціалізованих інструментів	Стратегія спеціалізації

Таблиця 4.16

Визначення базової стратегії конкурентної поведінки

№	Чи є проєкт «першопрохідцем» на ринку?	Чи буде компанія шукати нових споживачів, або забирати існуючих у конкурентів?	Чи буде компанія копіювати основні характеристики товару конкурентів та, які?	Стратегія конкурентної поведінки
1	Ні, на ринку вже є конкуренти	Компанія орієнтуватиметься на нових споживачів (розробників TypeScript) та частково на споживачів конкурентів	Так, будуть частково використані базові функції конкурентів (лінтинг, форматування), але з унікальними доповненнями (Hinting Mode, Step-by-step linting)	Стратегія наслідування лідеру

Таблиця 4.17.

Визначення стратегії позиціонування

№	Вимоги до товару цільової аудиторії	Базова стратегія розвитку	Ключові конкурентоспроможні позиції власного стартап-проєкту	Вибір асоціацій, які мають сформувати комплексну позицію власного проєкту (три ключові)
1	Інтерактивність, простота використання, орієнтація на JavaScript	Стабілізація позицій у ніші	Унікальні функції: Hinting Mode, Step-by-step linting Глибока інтеграція з VSCode Простота доступу до продукту через VSCode Marketplace	Інноваційність Доступність Орієнтація на сучасні технології

4.5. Розроблення маркетингової програми стартап-проекту

Таблиця 4.18.

Визначення ключових переваг концепції потенційного товару

№	Потреба	Вигода, яку пропонує товар	Ключові переваги перед конкурентами (існуючі або такі, що потрібно створити)
1	Зручний та ефективний інструмент для оптимізації коду	Інтерактивні підказки (Hinting Mode) допомагають розробникам швидко знаходити та виправляти помилки у коді	Унікальність Hinting Mode, що дозволяє вибір способу виправлення, якого немає у конкурентів
2	Можливість працювати з великими проєктами	Пошаговий лінтинг (Step-by-step linting) дозволяє фокусуватися лише на частинах коду, що потребують перевірки	Гнучкість у використанні інструменту для проєктів будь-якого масштабу
3	Сучасні інструменти для роботи з JS та TS	Продукт адаптований до потреб JavaScript-розробників завдяки інтеграції з сучасними бібліотеками AST	Орієнтація на TS як на мову зростаючої популярності
4	Простий доступ до інструменту	VSCode Marketplace забезпечує легкий доступ до продукту та швидке встановлення	Простота інтеграції у робочий процес через знайому платформу

Продовження таблиці 4.18.

5	Надійний і доступний інструмент для командного використання	Продукт підтримує спільну роботу через стандартизований стиль коду та гнучкі налаштування	Поєднання функцій лінтингу та форматування з інтерактивними можливостями

Таблиця 4.19.

Опис трьох рівнів моделі товару

Рівні товару	Сутність та складові	
I. Товар за задумом	Система аналізу та оптимізації програмного коду у вигляді розширення для VSCode	
II. Товар у реальному виконанні	Властивості/характеристики	Значення
	1. Модуль обробки JS коду	Покращення якості коду
	2. Модуль знаходження помилок	Зручність у використанні
	3. Модуль форматування за налаштуваннями файлу конфігурації	Підтримка роботи з великими проєктами
	Якість: відповідність стандартам розробки (JS), тестування на сумісність із великими проєктами.	
	Пакування: -	
	Марка: «Lintfy» – назва, що відображає суть і унікальність продукту.	
III. Товар із підкріпленням	До продажу: стартап-проєкт із докладною документацією та доступним кодом для встановлення.	
	Після продажу: повноцінна робоча система із регулярніими оновленнями, технічна підтримка, інтеграція з іншими DevTools.	

Продовження таблиці 4.19.

За рахунок чого потенційний товар буде захищено від копіювання: Захист товару забезпечується через використання унікальних модулів (наприклад, модуль Hinting Mode), які інтегрують інноваційні підходи до аналізу та форматування коду. Крім того, реалізація постійних оновлень, адаптація під нові стандарти JavaScript і TypeScript та реєстрація інтелектуальної власності (патенти, авторські права на алгоритми) підвищують захищеність продукту.

Таблиця 4.20.

Визначення меж встановлення ціни

№	Рівень цін на товари-замінники	Рівень цін на товари-аналоги	Рівень доходів цільової групи споживачів	Верхня та нижня межі встановлення ціни на товар/послугу
1	$0 – $100	$0 – $100	$1000/місяць	$0 – $100

Таблиця 4.21.

Формування системи збуту

№	Специфіка закупівельної поведінки цільових клієнтів	Функції збуту, які має виконувати постачальник товару	Глибина каналу збуту	Оптимальна система збуту
1	Орієнтація на швидкий доступ до продукту	Забезпечення простого встановлення без участі сторонніх посередників	Одноріневий канал	Розповсюдження через відкриту платформу

Таблиця 4.22.

Концепція маркетингових комунікацій

№	Специфіка поведінки цільових клієнтів	Канали комунікації, якими користуються цільові клієнти	Ключові позиції, обрані для позиціонування	Завдання рекламного повідомлення	Концепція рекламного звернення
1	Цільова аудиторія активно використовує онлайн-ресурси та спільноти розробників	Соціальні мережі (Twitter, LinkedIn), форуми (Stack Overflow), конференції розробників	Простота використання, доступність, надійність, підтримка, актуальність	Підкреслити унікальність функцій Hinting Mode та Step-by-step linting	«Lintify – розумний шлях до якісного коду. Інтерактивні рішення для сучасних розробників!»

ВИСНОВОК ДО РОЗДІЛУ 4

На основі проведеного аналізу ринкових можливостей та конкурентного середовища стартап-проєкту можна зробити висновок, що продукт володіє значним потенціалом для успішного виходу на ринок. Зростаюча популярність інструментів для аналізу та форматування коду серед розробників JavaScript створює сприятливі умови для комерціалізації. Зокрема, динаміка ринку свідчить про високий попит на подібні продукти, а платформа VSCode Marketplace забезпечує доступ до широкої аудиторії потенційних клієнтів.

Унікальні функції продукту, такі як Hinting Mode та Step-by-step linting, надають значну конкурентну перевагу , відповідають сучасним запитам розробників та дозволяють суттєво спростити процес покращення коду, орієнтуючись на інтерактивний підхід. У поєднанні з гнучкістю використання, продукт здатний зайняти нішу, яка наразі недостатньо охоплена існуючими рішеннями. Для успішної реалізації проєкту необхідно врахувати високий рівень конкуренції, що вимагає ретельно продуманої маркетингової стратегії. Обраний підхід із розробкою базової безкоштовної версії продукту для початкового залучення користувачів та впровадження платних розширених функцій є доцільним і дозволяє адаптуватися до різних сегментів аудиторії. Такий підхід не лише залучає більшу кількість клієнтів, але й створює умови для монетизації у довгостроковій перспективі.

Підсумовуючи, впровадження проєкту "Lintify" є перспективним і обґрунтованим з точки зору ринкової потреби і технічної реалізації. Продукт відповідає актуальним тенденціям, демонструє високу ймовірність рентабельності та сприяє впровадженню інновацій. Подальша імплементація проєкту рекомендована, оскільки він має потенціал для закріплення позицій на ринку та забезпечення сталого розвитку.

ВИСНОВКИ

У процесі виконання магістерської дисертації здійснено глибоке дослідження проблеми аналізу та оптимізації програмного коду. Робота охоплює теоретичні й практичні аспекти, що дозволило розробити систему, яка відповідає сучасним вимогам програмної інженерії. Виконаний аналіз підтверджує актуальність обраної теми, оскільки розвиток сучасних програмних продуктів потребує підвищення продуктивності, стабільності й безпеки, що безпосередньо залежить від якості коду.

Дослідження показують, що низька якість коду є основним джерелом помилок, вразливостей і зниження продуктивності програмного забезпечення. Недоліки існуючих підходів до аналізу коду, таких як обмежена інтерактивність, жорсткі правила форматування та відсутність комплексного аналізу, вказують на необхідність впровадження нових рішень. Особливої уваги заслуговує проблема перевірки великих кодових баз, яка часто призводить до затримок у розробці через тривалі процеси аналізу. Це підкреслює важливість інструментів, здатних виконувати інкрементальний аналіз, фокусуючись лише на змінених частинах проекту.

Результатом дослідження є розробка системи аналізу та оптимізації коду, яка забезпечує інтерактивний підхід до аналізу та виправлення коду. Основною інновацією є впровадження режиму Hinting Mode, що дозволяє розробникам отримувати не лише автоматичні виправлення, а й пояснення до кожної запропонованої зміни. Це сприяє підвищенню рівня знань у команді, особливо для менш досвідчених розробників, і допомагає уникати типових конфліктів, пов’язаних із автоматичними виправленнями. Окрім того, система реалізує покроковий аналіз (Step-by-step linting), що дозволяє поетапно перевіряти та рефакторити окремі модулі проекту. Такий підхід підвищує продуктивність і знижує ризики, пов’язані з масштабними змінами.

Практичне значення розробленої системи полягає в її інтеграції із середовищем розробки Visual Studio Code, що робить її доступною та зручною

для використання у повсякденній роботі програмістів. Система дозволяє автоматизувати рутинні завдання перевірки й оптимізації коду, зменшуючи кількість помилок, що виникають через людський фактор. Це значно прискорює процеси розробки, тестування та впровадження змін, що є критично важливим у сучасному швидкоплинному середовищі програмної інженерії.

Особливу увагу приділено аналізу ринкових перспектив розробленої системи. Проведений аналіз підтверджує її потенціал для комерційного використання. Визначено ключові сегменти ринку, які можуть отримати найбільшу вигоду від впровадження даного продукту, та розроблено стратегію його просування. У контексті стартап-проекту система може стати конкурентоспроможним рішенням, що поєднує в собі функціональність і гнучкість для команд різного рівня.

Отже, виконана робота підтверджує, що запропонована система здатна вирішити низку актуальних проблем у розробці програмного забезпечення. Її використання сприятиме підвищенню продуктивності роботи команд розробників, забезпечить якість і підтримуваність коду, а також допоможе зменшити витрати на розробку та підтримку продуктів. Результати дисертації можуть стати основою для подальших досліджень і вдосконалення автоматизованих інструментів аналізу та оптимізації програмного коду, що, у свою чергу, сприятиме розвитку галузі загалом.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Optimizing Code for Faster Performance: Techniques and Strategies [Електронний ресурс]. – Режим доступу: https://blog.kodezi.com/optimizing-code-for-faster-performance-techniques-and-strategies/ – Дата доступу: Вересень 2024.
2. Сучасний підручник з JavaScript [Електронний ресурс] – Режим доступу: https://uk.javascript.info/coding-style – Дата доступу: Вересень 2024.
3. Scalability analysis comparisons of cloud-based software services [Електронний ресурс]. – Режим доступу: https://journalofcloudcomputing.springeropen.com/articles/10.1186/s13677-019-0134-y – Дата доступу: Вересень 2024.
4. Performance Evaluation of Resource Management in Cloud Computing Environments [Електронний ресурс]. – Режим доступу: https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0141914 – Дата доступу: Вересень 2024.
5. Steven C. McConnell, 2022, «Code Complete» [Електронний ресурс]. – Режим доступу: https://library-it.com/wp-content/uploads/2020/09/stiv_makkonnell_sovershennyj_kod.pdf – Дата доступу: Вересень 2024.
6. Тестування безпеки: SQL-ін’єкції [Електронний ресурс]. – Режим доступу: https://training.qatestlab.com/blog/technical-articles/security-testing-sql-injection/ – Дата доступу: Вересень 2024.
7. Poor Code Quality [Електронний ресурс]. – Режим доступу: https://owasp.org/www-project-mobile-top-10/2016-risks/m7-client-code-quality – Дата доступу: Вересень 2024.
8. Toward a Reference Model for Adopting Software Continuous Delivery: A Practical Approach [Електронний ресурс]. – Режим доступу: https://scialert.net/fulltext/fulltextpdf.php?pdf=ansinet/jse/2018/12-19.pdf – Дата доступу: Вересень 2024.

9. Automated Code Review Tools for Security [Електронний ресурс]. https://ieeexplore.ieee.org/document/4712512 – Режим доступу: – Дата доступу: Вересень 2024.
10. Implementation of Continuous Integration and Continuous Delivery (CI/CD) on Automatic Performance Testing [Електронний ресурс]. https://ieeexplore.ieee.org/document/9527496 – Режим доступу: – Дата доступу: Вересень 2024.
11. Command Line Interface Reference [Електронний ресурс]. – Режим доступу: https://eslint.org/docs/latest/use/command-line-interface – Дата доступу: Вересень 2024.
12. Continuous Integration VS Continuous Delivery VS Continuous Deployment [Електронний ресурс]. https://dou.ua/forums/topic/46804/ – Режим доступу: – Дата доступу: Вересень 2024.
13. Continuous Delivery - Get Started with CI/CD [Електронний ресурс]. – Режим доступу: https://www.atlassian.com/continuous-delivery/continuous-integration – Дата доступу: Жовтень 2024.
14. Why Prettier? [Електронний ресурс]. – Режим доступу: https://prettier.io/docs/en/why-prettier – Дата доступу: Вересень 2024.
15. DevOps vs. Agile [Електронний ресурс]. https://www.atlassian.com/devops/what-is-devops/agile-vs-devops – Режим доступу: – Дата доступу: Жовтень 2024.
16. Extension API [Електронний ресурс]. https://code.visualstudio.com/api – Режим доступу: – Дата доступу: Жовтень 2024.
17. VS Code API [Електронний ресурс]. https://code.visualstudio.com/api/references/vscode-api – Режим доступу: – Дата доступу: Жовтень 2024.
18. Esprima [Електронний ресурс]. https://readthedocs.org/projects/esprima/downloads/pdf/4.0/ – Режим доступу: – Дата доступу: Жовтень 2024.

19. File system [Електронний ресурс]. https://nodejs.org/api/fs.html - file-system – Режим доступу: – Дата доступу: Жовтень 2024.

ДОДАТОК 1

Система аналізу та оптимізації програмного коду

Текст програмного коду

Аркушів 20

Київ 2024

extension.ts

```
import * as vscode from 'vscode';
import { scanDocument, createConfig, autoFix, stepByStep,
removeSelection } from './features/lint-commands';
import { onDidChangeFileJS, onDidOpenFileJS, onDidSaveConfig }
from './features/lint-handlers';
import { ActionProvider } from './features/providers/action-
provider';

export function activate(context: vscode.ExtensionContext) {

    console.log('Congratulations, your extension "lintify" is
now active!');

    const editor = vscode.window.activeTextEditor;

    const scanCommand =
vscode.commands.registerCommand('lintify.scan', () => {
        removeSelection()
        editor && scanDocument(editor.document);
    });

    const createConfigCommand =
vscode.commands.registerCommand('lintify.createConfig', () => {
        createConfig();
    });

    const autofixCommand =
vscode.commands.registerCommand('lintify.autoFix', async () => {
        editor && await autoFix();
    });

    const stepByStepCommand =
vscode.commands.registerCommand('lintify.stepByStep', () => {
        editor && stepByStep(editor);
    });

    const codeActionProvider =
vscode.languages.registerCodeActionsProvider('javascript', new
ActionProvider(), {
        providedCodeActionKinds:
[vscode.CodeActionKind.QuickFix]
    });


vscode.commands.executeCommand('lintify.createConfig').then();

    vscode.workspace.onDidOpenTextDocument(onDidOpenFileJS);
    vscode.workspace.onDidChangeTextDocument(onDidChangeFileJS);
    vscode.workspace.onDidSaveTextDocument(onDidSaveConfig);

    context.subscriptions.push(scanCommand, createConfigCommand,
```

```
autofixCommand, stepByStepCommand, codeActionProvider);
}

export function deactivate() {}
```

lint-commands.ts

```
import fs from 'node:fs';
import path from 'node:path';
import * as esprima from 'esprima';
import { CodeAction, Diagnostic, Position, Range, TextDocument,
TextEditor, window, workspace } from 'vscode';

import { CONFIG_NAME, DEFAULT_CONFIG, WORK_DIR } from
'../constants';
import { lintDocumentWithDiagnostics } from './lint-handlers';
import { ConfigLoader } from '../config/config-loader';
import { LinterProvider } from './providers/linter-provider';
import { PrettierProvider } from './providers/prettier-
provider';

export let selectionRange: Range | null;

export const diagnosticToActionMap = new Map<Diagnostic,
CodeAction[]>();

export const scanDocument = (document: TextDocument) => {
    const code = selectionRange ?
document.getText(selectionRange) : document.getText();

    console.log(code);

    const config = ConfigLoader.getConfig();

    try {
        const ast = esprima.parseModule(code,
            {
                tolerant: true,
                loc: true,
                comment: true,
                tokens: true
            }
        );

        const linterProvider = new LinterProvider(ast, config);
        const prettierProvider = new PrettierProvider(document,
ast, config);
        const lintErrors = linterProvider
            .getErrors()
            .sort((a, b) => a.loc.start.line - b.loc.start.line)
            .filter(error => error.severity === 'error')
            .concat(linterProvider.getErrors().filter(error =>
error.severity === 'warning'));
```

```
        const prettierErrors = prettierProvider
            .getErrors()
            .sort((a, b) => a.loc.start.line - b.loc.start.line)
            .filter(error => error.severity === 'error')
            .concat(prettierProvider.getErrors().filter(error =>
error.severity === 'warning'));

        lintDocumentWithDiagnostics(lintErrors, prettierErrors,
document);

    } catch (e) {
        console.log(e);
    }
};

export const autoFix = async () => {
    diagnosticToActionMap.forEach((actions, diagnostic) => {
        const { source } = diagnostic;

        if (source === 'strict-equality' || source === 'strict-
inequality' || source === 'semicolon' || source === '"' ||
source === '\'') {
            actions.forEach(({ edit }) => edit &&
workspace.applyEdit(edit));
            diagnosticToActionMap.delete(diagnostic);
        }
    });

    await window.showInformationMessage('All auto-fixable issues
have been resolved.');
};

export const stepByStep = (editor: TextEditor) => {
    const {start, end} = editor.selection;

    selectionRange = JSON.stringify(start) ===
JSON.stringify(end) ? null : new Range(
        new Position(start.line, start.character),
        new Position(end.line, end.character)
    );

    console.log(start, end, JSON.stringify(start) ===
JSON.stringify(end));
    scanDocument(editor.document)
}

export const createConfig = (): boolean => {
    try {
        const userConfigPath = path.join(WORK_DIR, CONFIG_NAME);

        if (fs.existsSync(userConfigPath)) return true;

        fs.openSync(userConfigPath, 'w');
```

```
        fs.writeFileSync(userConfigPath,
JSON.stringify(DEFAULT_CONFIG));

        console.log('User configuration added');

        ConfigLoader.updateConfig();
        return true;
    } catch (e) {
        console.warn('Something gone wrong: ', e);
        return false;
    }
};

export const removeSelection = () => {
    selectionRange = null;
}
```

lint-handlers.ts

```
import path from 'node:path';
import vscode, { TextDocument, TextDocumentChangeEvent } from
'vscode';

import { CONFIG_NAME, WORK_DIR } from '../constants';
import { addDeleteFix, addFixForSemicolon, addReplaceFix,
addReplaceQuotesFix } from '../utils/code-actions';

import { diagnosticToActionMap, scanDocument, selectionRange }
from './lint-commands';
import { LintifyError } from '../types';
import { ConfigLoader } from '../config/config-loader';

export const lintDiagnostics =
vscode.languages.createDiagnosticCollection('lintify');
export const prettierDiagnostics =
vscode.languages.createDiagnosticCollection('prettify');

let timeout: NodeJS.Timeout | undefined;

export const onDidChangeFileJS = ({ document }:
TextDocumentChangeEvent) => {
    document.languageId === 'javascript' &&
triggerDelayedLint(document);
};

export const onDidOpenFileJS = (document: TextDocument) => {
    document.languageId === 'javascript' &&
vscode.commands.executeCommand('lintify.scan');
};

export const onDidSaveConfig = (document: TextDocument) => {
    if (document.uri.fsPath === path.join(WORK_DIR,
CONFIG_NAME)) {
```

```
        ConfigLoader.updateConfig();
        vscode.commands.executeCommand('lintify.scan').then();
    }
};

export const lintDocumentWithDiagnostics = (lintErrors:
LintifyError[], prettierErrors: LintifyError[], document:
vscode.TextDocument) => {
    diagnosticToActionMap.clear();

    lintDiagnostics.set(document.uri,
createDiagnostic(lintErrors));
    prettierDiagnostics.set(document.uri,
createDiagnostic(prettierErrors));


vscode.languages.getDiagnostics(document.uri).forEach(diagnostic
=> {
        const { source } = diagnostic;

        switch (source) {
            case 'no-console-log':
                diagnosticToActionMap.set(diagnostic, [
                    addReplaceFix({ document, diagnostic, text:
'console.warn' })
                ])
                break;
            case 'no-var-keyword':
                diagnosticToActionMap.set(diagnostic, [
                    addReplaceFix({ document, diagnostic, text:
'const' }),
                    addReplaceFix({ document, diagnostic, text:
'let' })
                ])
                break;
            case 'strict-equality':
                diagnosticToActionMap.set(diagnostic, [
                    addReplaceFix({ document, diagnostic, text:
'===' })
                ])

                break;
            case 'strict-inequality':
                diagnosticToActionMap.set(diagnostic, [
                    addReplaceFix({ document, diagnostic, text:
'!==' })
                ])
                break;
            case 'no-debugger':
                diagnosticToActionMap.set(diagnostic, [
                    addDeleteFix({ document, diagnostic, text:
'debugger' })
                ])
```

```
                break;
            case '"':
            case '\'':
                diagnosticToActionMap.set(diagnostic, [
                    addReplaceQuotesFix({
                        document,
                        diagnostic,
                        text: source
                    })
                ])
                break;
            case 'semicolon':
                diagnosticToActionMap.set(diagnostic, [
                    addFixForSemicolon(document, diagnostic)
                ])
                break;
        }
    });
};

const createDiagnostic = (errors: LintifyError[]):
vscode.Diagnostic[] => {
    const diagnostics: vscode.Diagnostic[] = [];

    errors.forEach(error => {
        const startLine = selectionRange ?
selectionRange.start.line + error.loc.start.line - 1 :
error.loc.start.line - 1
        const endLine = selectionRange ?
selectionRange.start.line + error.loc.end.line - 1 :
error.loc.end.line - 1
        const startCol = selectionRange ?
selectionRange.start.character + error.loc.start.column :
error.loc.start.column
        const endCol = selectionRange ?
selectionRange.start.character + error.loc.end.column :
error.loc.end.column

        const range = new vscode.Range(
            new vscode.Position(startLine, startCol),
            new vscode.Position(endLine, endCol)
        );

        const diagnostic = new vscode.Diagnostic(
            range,
            error.message,
            vscode.DiagnosticSeverity[error.severity === 'error'
? 'Error' : 'Warning']
        );

        diagnostic.source = error.source;
        diagnostic.code = error.code;
```

```
        diagnostics.push(diagnostic);
    })

    return diagnostics
}

export const triggerDelayedLint = (document:
vscode.TextDocument) => {
    timeout && clearTimeout(timeout);

    timeout = setTimeout(() => scanDocument(document), 1000);
};
```

code-actions.ts

```
import vscode, { Diagnostic, TextDocument } from 'vscode';

interface FixParams {
    document: vscode.TextDocument,
    diagnostic: vscode.Diagnostic,
    text: string
}

export const addFixForSemicolon = (document: TextDocument,
diagnostic: Diagnostic): vscode.CodeAction => {
    const fix = new vscode.CodeAction('Add semicolon?',
vscode.CodeActionKind.QuickFix);

    fix.edit = new vscode.WorkspaceEdit();
    fix.edit.replace(document.uri, diagnostic.range, ';');
    fix.diagnostics = [diagnostic];

    return fix;
}

export const addReplaceFix = ({ document, diagnostic, text }:
FixParams): vscode.CodeAction => {
    const fix = new vscode.CodeAction(`Replace with ${ text }`,
vscode.CodeActionKind.QuickFix);

    fix.edit = new vscode.WorkspaceEdit();
    fix.edit.replace(document.uri, diagnostic.range, text);
    fix.diagnostics = [diagnostic];

    return fix;
}

export const addReplaceQuotesFix = ({ document, diagnostic, text
}: FixParams): vscode.CodeAction => {
    const quote = text;
    const fix = new vscode.CodeAction(`Replace with ${ quote }`,
vscode.CodeActionKind.QuickFix);
```

```
    const firstRange = new vscode.Range(
        diagnostic.range.start,
        new vscode.Position(diagnostic.range.start.line,
diagnostic.range.start.character + 1)
    );

    const secondRange = new vscode.Range(
        new vscode.Position(diagnostic.range.end.line,
diagnostic.range.end.character - 1),
        diagnostic.range.end
    );

    const replacement = quote === '"' ? '\'' : '"';
    const valueRange = new vscode.Range(firstRange.end,
secondRange.start)

    fix.edit = new vscode.WorkspaceEdit();
    fix.edit.replace(document.uri, firstRange, quote);
    fix.edit.replace(document.uri, secondRange, quote);
    fix.edit.replace(
        document.uri,
        valueRange,
        document.getText(valueRange).replaceAll(text,
replacement)
    )

    fix.diagnostics = [diagnostic];

    return fix;
}

export const addDeleteFix = ({ document, diagnostic, text }:
FixParams): vscode.CodeAction => {
    const fix = new vscode.CodeAction(`Remove redundant ${ text
}`, vscode.CodeActionKind.QuickFix);

    fix.edit = new vscode.WorkspaceEdit();
    fix.edit.delete(document.uri, diagnostic.range);
    fix.diagnostics = [diagnostic];

    return fix;
}
```

config-loader.ts

```
import * as path from 'node:path';
import * as fs from 'node:fs';

import { Config } from '../types';
import { CONFIG_NAME, DEFAULT_CONFIG, WORK_DIR } from
'../constants';

export class ConfigLoader {
```

```
    private static config: Config | null = null;

    private static loadConfig (): Config  {
        try {
            const userConfigPath = path.join(WORK_DIR,
CONFIG_NAME);

            if (!fs.existsSync(userConfigPath)) {
                console.warn('No configuration file found. Using
default settings.');
                return DEFAULT_CONFIG;
            }

            const userConfigData =
fs.readFileSync(userConfigPath, 'utf-8');
            const config = JSON.parse(userConfigData) as Config;
            console.log('User configuration loaded:', config);

            return config || DEFAULT_CONFIG;
        } catch (e) {
            console.warn('Something gone wrong: ', e);
            return DEFAULT_CONFIG;
        }
    };

    static getConfig() {
        if (!this.config) this.config = this.loadConfig()
        return this.config;
    }

    static updateConfig() {
        this.config = this.loadConfig()
        console.log('User configuration updated:', this.config);
    }
}
```

action-provider.ts

```
import vscode, { CodeAction, Diagnostic } from 'vscode';
import { diagnosticToActionMap } from '../lint-commands';

export class ActionProvider implements vscode.CodeActionProvider
{

    public provideCodeActions(
        document: vscode.TextDocument,
        range: vscode.Range,
        context: vscode.CodeActionContext,
        token: vscode.CancellationToken
    ) {
        const codeActions: CodeAction[] = [];

        context.diagnostics.forEach((diagnostic: Diagnostic) =>
```

```
{

codeActions.push(...diagnosticToActionMap.get(diagnostic) || [])
        })

        return codeActions;
    }

}
```

linter-provider.ts

```
import { Config, LintifyError, RangedToken, Severity } from
'../../types';
import { Program } from 'esprima';
import { Identifier, Node } from 'estree';


export class LinterProvider {
    private readonly ast: Program;
    private readonly tokens: RangedToken[];
    private readonly config: Config;

    constructor(ast: Program, config: Config) {
        this.ast = ast;
        this.tokens = ast.tokens as RangedToken[];
        this.config = config;
    }

    getRules() {
        return [
            this.noConsoleLog,
            this.noVarKeyword,
            this.requireStrictEquality,
            this.camelCaseVariables,
            this.noEval,
            this.noDebugger,
            this.capitalizedConstants,
            this.noSelfAssignment
        ];
    }

    noConsoleLog = (severity: Severity): LintifyError[] => {
        const errors: LintifyError[] = [];
        const { tokens } = this;

        tokens
            .map(({ type, value, loc }, index) => {
                if (
                    type === 'Identifier' && value === 'console'
&&
                    tokens[index + 1].type === 'Punctuator' &&
tokens[index + 1].value === '.' &&
```

```
                    tokens[index + 2].type === 'Identifier' &&
tokens[index + 2].value === 'log'
                ) return {
                    type: 'Console log',
                    value: 'console.log',
                    loc: {
                        start: loc.start,
                        end: tokens[index + 2].loc.end
                    }
                };
            })
            .forEach(token => token && errors.push({
                loc: token.loc,
                message: 'Avoid using console.log in production
code.',
                severity,
                source: 'no-console-log'
            }));

        return errors;
    };

    noVarKeyword = (severity: Severity): LintifyError[] => {
        const errors: LintifyError[] = [];
        const { tokens } = this;

        tokens
            .filter(({ type, value }) => type === 'Keyword' &&
value === 'var')
            .forEach(token => token && errors.push({
                loc: token.loc,
                message: 'Use "const" or "let" instead of
"var".',
                severity,
                source: 'no-var-keyword'
            }));

        return errors;
    };

    requireStrictEquality = (severity: Severity): LintifyError[]
=> {
        const errors: LintifyError[] = [];
        const { tokens } = this;

        tokens
            .filter(token => token.type === 'Punctuator' &&
(token.value === '==' || token.value === '!='))
            .forEach(token => token && errors.push({
                loc: token.loc,
                message: token.value === '=='
                    ? 'Use strict equality (===) instead of ==
for comparisons.'
```

```
                    : 'Use strict inequality (!==) instead of !=
for comparisons.',
                severity,
                source: `strict-${ token.value === '==' ?
'equality' : 'inequality' }`
            }));

        return errors;
    };

    camelCaseVariables = (severity: Severity): LintifyError[] =>
{
        const errors: LintifyError[] = [];
        const { tokens } = this;

        tokens
            .map(({ type, value }, index) => {
                if (
                    type === 'Keyword' &&
                    (value === 'var' || value === 'let' || value
=== 'const' || value === 'function') &&
                    tokens[index + 1].type == 'Identifier'
                ) return tokens[index + 1];
            })
            .filter(token => token !== undefined)
            .forEach(token => {
                const { value, loc } = token;

                if (!/^[a-z][a-zA-Z0-9]*$/.test(value)) {
                    errors.push({
                        loc,
                        message: `Variable name '${ value }'
should be in camelCase.`,
                        severity,
                        source: 'camel-case-variables'
                    });
                }
            });

        return errors;
    };

    noEval = (severity: Severity): LintifyError[] => {
        const errors: LintifyError[] = [];
        const { tokens } = this;

        tokens
            .filter(({ type, value }) => type === 'Identifier'
&& value === 'eval')
            .forEach(token => token && errors.push({
                loc: token.loc,
                message: 'Avoid using "eval" in production
code.',
```

```
                severity,
                source: 'no-eval'
            }));

        return errors;
    };

    noDebugger = (severity: Severity) => {
        const errors: LintifyError[] = [];
        const { tokens } = this;

        tokens
            .filter(({ type, value }) => type === 'Keyword' &&
value === 'debugger')
            .forEach(token => token && errors.push({
                loc: token.loc,
                message: 'Avoid using "debugger" in production
code.',
                severity,
                source: 'no-debugger'
            }));

        return errors;
    };

    capitalizedConstants = (severity: Severity): LintifyError[]
=> {
        const errors: LintifyError[] = [];
        const { body } = this.ast;

        body
            .filter(node => node.type === 'VariableDeclaration')
            .filter(({ kind, declarations }) =>
                kind === 'const' && declarations[0].init?.type
=== 'Literal'
            )
            .forEach(node => {
                const { name }: Identifier =
node.declarations[0].id as Identifier;

                node.loc && errors.push({
                    loc: node.loc,
                    message: `Constant \'${ name }\' should be
written in uppercase.`,
                    severity,
                    source: 'capitalized-constants',
                });
            });

        return errors;
    };

    noSelfAssignment = (severity: Severity): LintifyError[] => {
```

```
        const errors: LintifyError[] = [];
        const { ast } = this;

        const traverse = (node: Node) => {
            if (!node) return;

            if (node.type === 'AssignmentExpression') {
                const { left, right } = node;

                if (left.type === 'Identifier' && right.type ===
'Identifier' && left.name === right.name) {
                    left.loc && right.loc && errors.push({
                        loc: {
                            start: left.loc.start,
                            end: right.loc.end
                        },
                        message: `Variable '${ left.name }' is
assigned to itself.`,
                        severity,
                        source: 'no-self-assignment'
                    });
                }
            }

            Object
                .values(node)
                .filter(value => value && typeof value ===
'object')
                .forEach(traverse);
        };

        ast.body.forEach(traverse);

        return errors;
    };

    getErrors(): LintifyError[] {
        const { config } = this;
        const { linter } = config;
        const errors: LintifyError[] = [];

        if (!linter || !Object.keys(linter).length) return [];

        this.getRules().forEach((rule) => {
            if (!Object.keys(linter).includes(rule.name))
return;

            const { enabled, severity } = linter[rule.name];
            enabled && errors.push(...rule(severity));
        });

        return errors;
```

```
    }
}
```

prettier-provider.ts

```
import vscode, { TextDocument } from 'vscode';

import { Config, LintifyError, RangedToken, Severity } from
'../../types';
import { Program } from 'esprima';
import { Node } from 'estree';

import { selectionRange } from '../lint-commands';

export class PrettierProvider {
    private readonly document: TextDocument;
    private readonly ast: Program;
    private readonly tokens: RangedToken[];
    private readonly config: Config;

    constructor(document: TextDocument, ast: Program, config:
Config) {
        this.document = document;
        this.ast = ast;
        this.tokens = ast.tokens as RangedToken[];
        this.config = config;
    }

    getRules() {
        return [
            this.identicalQuotes,
            this.semicolonRequired,
        ];
    }

    identicalQuotes = (severity: Severity, quote: string =
'single'): LintifyError[] => {
        const errors: LintifyError[] = [];
        const { tokens } = this;

        const quotes: { [key: string]: any } = {
            double: {
                target: '\'',
                replacement: '"'
            },
            single: {
                target: '"',
                replacement: '\''
            }
        };
```

```
        const { target, replacement } =
Object.keys(quotes).includes(quote) ? quotes[quote] :
quotes.single;

        tokens
            .filter(({ type, value }) => type === 'String' &&
value[0] === target)
            .forEach(token => token && errors.push({
                message: `Use ${ replacement } instead of ${
target }`,
                loc: token.loc,
                severity,
                source: replacement
            }));

        return errors;
    };

    semicolonRequired = (severity: Severity): LintifyError[] =>
{
        const errors: LintifyError[] = [];
        const { document, ast } = this;

        const traverse = (node: Node) => {
            if (!node) return;

            const {type, loc} = node;

            if (type === 'VariableDeclaration' || type ===
'ExpressionStatement' || type === 'ReturnStatement') {
                if (!loc || !loc.end) return;

                const location = {
                    start: {
                        line: loc.end.line,
                        column: loc.end.column
                    },
                    end: {
                        line: loc.end.line,
                        column: loc.end.column + 1
                    }
                }

                const startLine = selectionRange ?
selectionRange.start.line + location.start.line - 1 :
location.start.line - 1
                const endLine = selectionRange ?
selectionRange.start.line + location.end.line - 1 :
location.end.line - 1
                const startCol = selectionRange ?
selectionRange.start.character + location.start.column - 1 :
location.start.column - 1
```

```
                const endCol = selectionRange ?
selectionRange.start.character + location.end.column :
location.end.column

                const range = new vscode.Range(
                    new vscode.Position(startLine, startCol),
                    new vscode.Position(endLine, endCol)
                );

                if (document.getText(range) !== ';') {
                    errors.push({
                        loc: location,
                        message: `Missing semicolon at the end
of the statement.`,
                        severity: severity,
                        source: 'semicolon'
                    });
                }
            }

            Object
                .values(node)
                .filter(value => value && typeof value ===
'object')
                .forEach(traverse);

        }

        ast.body.forEach(traverse);
        return errors;
    };

    getErrors(): LintifyError[] {
        const { prettier } = this.config;
        const errors: LintifyError[] = [];

        if (!prettier || !Object.keys(prettier).length) return
[];

        this.getRules().forEach((rule) => {
            const { name } = rule;

            if (!Object.keys(prettier).includes(name)) return;

            const { enabled, severity, options } =
prettier[name];

            if (!enabled) return;

            switch (name) {
                case 'identicalQuotes':
                    errors.push(...rule(severity,
options?.quote));
```

```
                    break;
                case 'maxLineLength':
                    errors.push(...rule(severity,
options?.maxLength));
                    break;
                default:
                    errors.push(...rule(severity));
            }
        });

        return errors;
    }
}
```

constants.ts

```
import { Config } from './types';
import vscode from 'vscode';

export const DEFAULT_CONFIG: Config = {
    linter: {
        noConsoleLog: { enabled: true, severity: 'warning' },
        noVarKeyword: { enabled: true, severity: 'warning' },
        requireStrictEquality: { enabled: true, severity:
'warning' },
        camelCaseVariables: { enabled: true, severity: 'warning'
},
        capitalizedConstants: { enabled: true, severity:
'warning' },
        noEval: { enabled: true, severity: 'warning' },
        noDebugger: { enabled: true, severity: 'warning' },
        noSelfAssignment: { enabled: true, severity: 'warning'
},
    },
    prettier: {
        identicalQuotes: { enabled: true, severity: 'warning',
options: { quote: 'double' } },
        semicolonRequired: { enabled: true, severity: 'warning'
},
    }
};

export const WORK_DIR = vscode.workspace.workspaceFolders ?
vscode.workspace.workspaceFolders[0].uri.fsPath : '';

export const CONFIG_NAME = 'lintify.config.json';
```

types.ts

```
export type RangedToken = {
    type: string,
    value: string,
```

```
    loc: Location
}

type Location = {
    end: { line: number, column: number },
    start: { line: number, column: number },
}

export type LintifyError = {
    loc: Location;
    message: string;
    severity: Severity;
    source: string;
    code?: string
}

export type Severity = 'warning' | 'error';

export type LintRuleConfig = {
    enabled: boolean,
    severity: Severity
}

export type PrettierRuleConfig = {
    enabled: boolean,
    severity: Severity,
    options?: {
        [key: string]: any
    }
}

export type Config = {
    linter: {
        [ruleName: string]: LintRuleConfig
    },
    prettier: {
        [ruleName: string]: PrettierRuleConfig
    }
}
```

lintify.config.json

```
{
    "linter":{
        "noConsoleLog":{
            "enabled":true,"severity":"warning"
        },
        "noVarKeyword":{
            "enabled":true,"severity":"warning"
        },
        "requireStrictEquality":{
            "enabled":true,"severity":"warning"
        },
```

```
        "camelCaseVariables":{
            "enabled":true,"severity":"warning"
        },
        "capitalizedConstants":{
            "enabled":true,"severity":"warning"
        },
        "noEval":{
            "enabled":true,"severity":"warning"
        },
        "noDebugger":{
            "enabled":true,"severity":"warning"
        },
        "noSelfAssignment":{
            "enabled":true,"severity":"warning"
        }
    },
    "prettier":{
        "identicalQuotes":{
            "enabled":true,
            "severity":"warning",
            "options":{
                "quote":"double"
                }
            },
        "semicolonRequired":{
            "enabled":true,"severity":"warning"
        }
    }
}
```