Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерної інженерії та управління
(повна назва)

Кафедра електронних обчислювальних машин
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

Рівень вищої освіти другий (магістерський)

Метод предметно-орієнтованого аналізу вихідних текстів програм

(тема)

Виконав:
здобувач 2 року навчання,
групи СПм-23-5
Андрій ЛИТВИНОВ
(власне ім'я, прізвище)

Спеціальність
123 «Комп'ютерна інженерія»
(код і повна назва спеціальності)

Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Освітня програма
Системне програмування
(повна назва освітньої програми)

Керівник: доц. Віталій ТКАЧОВ
(посада, власне ім'я, прізвище)

Допускається до захисту

Завідувач кафедри ЕОМ _____ Андрій КОВАЛЕНКО
(підпис) (власне ім'я, прізвище)

2025 р.

Харківський національний університет радіоелектроніки

Факультет комп'ютерної інженерії та управління

Кафедра електронних обчислювальних машин

Рівень вищої освіти другий (магістерський)

Спеціальність 123 «Комп'ютерна інженерія»
(код і повна назва)

Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Освітня програма Системне програмування
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
"____" _____ 20__ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Литвинову Андрію Павловичу
(прізвище, ім'я, по батькові)

1. Тема роботи Метод предметно-орієнтованого аналізу вихідних текстів програм

затверджена наказом по університету від "21" квітня 2025 р. № 296 Ст

2. Термін подання здобувачем роботи до екзаменаційної комісії 16 червня 2025 р.

3. Вхідні дані до роботи
аналіз коду програм
семантичний та динамічний аналізи
Google Colab
Python

4. Перелік питань, що потрібно опрацювати у роботі
Загальні поняття і роль аналізу вихідних текстів програм
Статичні методи аналізу програмного коду
Програмна реалізація методу та аналіз результатів

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій 14 слайдів

6. Консультанти розділів роботи (заповнюється за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Строк / терміни виконання етапів роботи	Примітка
1	Отримання завдання та аналіз літератури	21.04.2025–30.04.2025	
2	Огляд існуючих рішень та алгоритмів	01.05.2025–12.05.2025	
3	Розробка методу	13.05.2025–22.05.2025	
4	Вибір програмних засобів	23.05.2025–30.05.2025	
5	Програмна реалізація	31.05.2025–02.06.2025	
6	Аналіз отриманих результатів	03.06.2025–05.06.2025	
7	Оформлення записки	06.06.2025–12.06.2025	

Дата видачі завдання " 21 " квітня 2025 р.

Здобувач _____
(підпис)

Керівник роботи _____ доц. Віталій ТКАЧОВ
(підпис) (посада, власне ім'я, прізвище)

РЕФЕРАТ

Пояснювальна записка кваліфікаційної роботи: 55 с., 8 рис., 2 дод., 8 джерел.

АНАЛІЗ ВИХІДНОГО КОДУ, ПРЕДМЕТНО-ОРІЄНТОВАНИЙ МЕТОД, СИНТАКСИЧНИЙ АНАЛІЗ, СЕМАНТИЧНИЙ АНАЛІЗ, ДИНАМІЧНИЙ АНАЛІЗ, PYTHON, GOOGLE COLAB, ПОМИЛКИ КОДУ, АВТОМАТИЗАЦІЯ ПЕРЕВІРКИ, ВІЗУАЛІЗАЦІЯ ПРОГРАМНОГО КОДУ, ІНСТРУМЕНТИ АНАЛІЗУ, ЯКІСТЬ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, СТРУКТУРНА ВЕРИФІКАЦІЯ, МАШИННЕ НАВЧАННЯ.

Метою кваліфікаційної роботи є розробка та реалізація комплексного методу аналізу вихідного коду програмного забезпечення, який поєднує класичні та інноваційні підходи, з урахуванням можливостей середовища Google Colab як платформи для автоматизованого аналізу, візуалізації результатів і демонстрації ефективності запропонованих рішень.

У ході виконання кваліфікаційної роботи було розроблено, реалізовано та протестовано метод предметно-орієнтовного аналізу вихідного коду програм з використанням інструментальних засобів Python у середовищі Google Colab. Запропонований підхід поєднує елементи класичного статичного аналізу з можливостями семантичного розбору, простежування структури коду та візуалізації його характеристик, що дозволяє ефективно виявляти низку поширених помилок, невідповідностей та стилістичних відхилень.

Аналіз проведено як на рівні синтаксичних елементів (ідентифікатори, ключові слова, довжина рядків), так і на рівні логіки структурування (класи, функції, конструкції контролю потоку), що забезпечило глибше розуміння внутрішньої організації програмного коду.

ABSTRACT

Master’s thesis: 55 pages, 6 figures, 2 appendices, 8 sources.

SOURCE CODE ANALYSIS, DOMAIN-SPECIFIC METHOD, SYNTACTIC ANALYSIS, SEMANTIC ANALYSIS, DYNAMIC ANALYSIS, PYTHON, GOOGLE COLAB, CODE ERRORS, AUTOMATED VERIFICATION, CODE VISUALIZATION, ANALYSIS TOOLS, SOFTWARE QUALITY, STRUCTURAL VERIFICATION, MACHINE LEARNING.

The major goal of this thesis is to develop and implement a comprehensive method for source code analysis that integrates both classical and innovative approaches, taking into account the capabilities of the Google Colab environment as a platform for automated analysis, result visualization, and demonstration of the proposed solutions’ effectiveness.

In order to a domain-specific method for analyzing source code was developed, implemented, and tested using Python-based tools within Google Colab. The proposed approach combines elements of classical static analysis with semantic parsing, code structure tracing, and visual representation of code features, enabling the effective identification of common errors, inconsistencies, and stylistic deviations.

The analysis was conducted both at the level of syntactic elements (identifiers, keywords, line lengths) and at the level of structural logic (classes, functions, control flow constructs), which provided a deeper understanding of the internal organization of program code.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

AI – штучний інтелект

API – інтерфейс прикладного програмування

AST – абстрактне синтаксичне дерево

CI/CD – безперервна інтеграція та доставка

CSV – формат табличних даних, розділених комами

IDE – інтегроване середовище розробки

IPYNB – формат файлу Jupyter Notebook

ML – машинне навчання

PIP – система керування пакунками Python

PPA – аналіз предметно-орієнтований

PyLint – засіб статичного аналізу Python-коду

RAM – оперативна пам’ять

UI – користувацький інтерфейс

ВСТУП

У сучасній програмній інженерії аналіз вихідного коду виступає фундаментальним інструментом забезпечення якості, надійності та безпеки програмного забезпечення. Зважаючи на ускладнення архітектури програмних систем, збільшення обсягу коду та зростання вимог до його продуктивності і безпеки, потреба в системному аналізі коду постає як стратегічна необхідність. Класичні підходи на основі ручної перевірки та традиційного тестування вже не здатні забезпечити необхідний рівень контролю в умовах швидкого розвитку індустрії. Тому застосування методів статичного, динамічного та інтелектуального аналізу коду стає ключовим чинником підвищення якості продуктів на всіх етапах життєвого циклу програмного забезпечення.

Актуальність даного дослідження зумовлена зростанням складності програмних продуктів, широким використанням сторонніх бібліотек, а також потребою в мінімізації технічного боргу та забезпеченні інформаційної безпеки. Використання лише одного підходу, наприклад статичного аналізу, хоча й дозволяє виявити значну кількість помилок ще до етапу виконання, не дає змоги повною мірою охопити динамічні властивості поведінки програми. Водночас динамічний аналіз дозволяє виявити критичні проблеми, що проявляються лише під час реального виконання, проте не охоплює всі варіанти логіки, які не були активовані у тестовому середовищі. Саме поєднання підходів дозволяє досягти найвищої ефективності. У цьому контексті інтелектуальні методи, що базуються на алгоритмах машинного навчання, відкривають нові можливості для прогнозування вразливостей та автоматичного виявлення дефектів, що виходить за межі традиційної логіки аналізу коду.

Метою работи є розробка та реалізація комплексного методу аналізу вихідного коду програмного забезпечення, який поєднує класичні та

інноваційні підходи, з урахуванням можливостей середовища Google Colab як платформи для автоматизованого аналізу, візуалізації результатів і демонстрації ефективності запропонованих рішень.

1 ЗАГАЛЬНІ ПОНЯТТЯ І РОЛЬ АНАЛІЗУ ВИХІДНИХ ТЕКСТІВ ПРОГРАМ

1.1 Поняття аналізу програмного коду та його місце у розробці програмного забезпечення в контексті предметно-орієнтованого підходу

Аналіз програмного коду посідає одне з ключових місць у процесі створення надійного, безпечного та якісного програмного забезпечення. Його суть полягає у всебічному вивченні вихідних текстів програм задля виявлення помилок, вразливостей, недотримання стандартів програмування, логічних неузгодженостей або структурних недоліків, що можуть вплинути на подальше функціонування системи. Аналіз вихідного коду виконує як контрольну, так і превентивну функцію: він дозволяє виявляти недоліки ще до моменту виконання програми, тим самим істотно знижуючи витрати на їхнє усунення на пізніших етапах життєвого циклу програмного продукту.

У сучасному підході до розробки програмного забезпечення, зокрема при використанні методологій Agile, DevOps чи CI/CD, аналіз коду інтегрується в кожну фазу життєвого циклу – від написання перших рядків коду до його тестування, впровадження і супроводу. На етапі розробки аналіз коду забезпечує дотримання стилістичних та синтаксичних правил, сприяє підтриманню єдиного стилю проєкту та полегшує командну роботу. У процесі тестування – допомагає виявляти дефекти, які не завжди покриваються юніт-тестами або інструментами функціонального тестування. На стадії впровадження – дає змогу зменшити ризики через виявлення небезпечних патернів, уразливостей або неефективного використання ресурсів.

Аналіз коду (рисунок 1.1) також виступає фундаментом для таких дисциплін, як рефакторинг, технічний аудит, управління якістю програмного забезпечення та забезпечення інформаційної безпеки. Високий рівень

автоматизації цього процесу, зокрема через використання сучасних інструментів статичного та динамічного аналізу, а також інтелектуальних систем, дає змогу не лише виявляти існуючі дефекти, а й формувати аналітичну картину стану кодової бази, що слугує основою для прийняття технічних рішень на рівні архітектури та підтримки продукту.

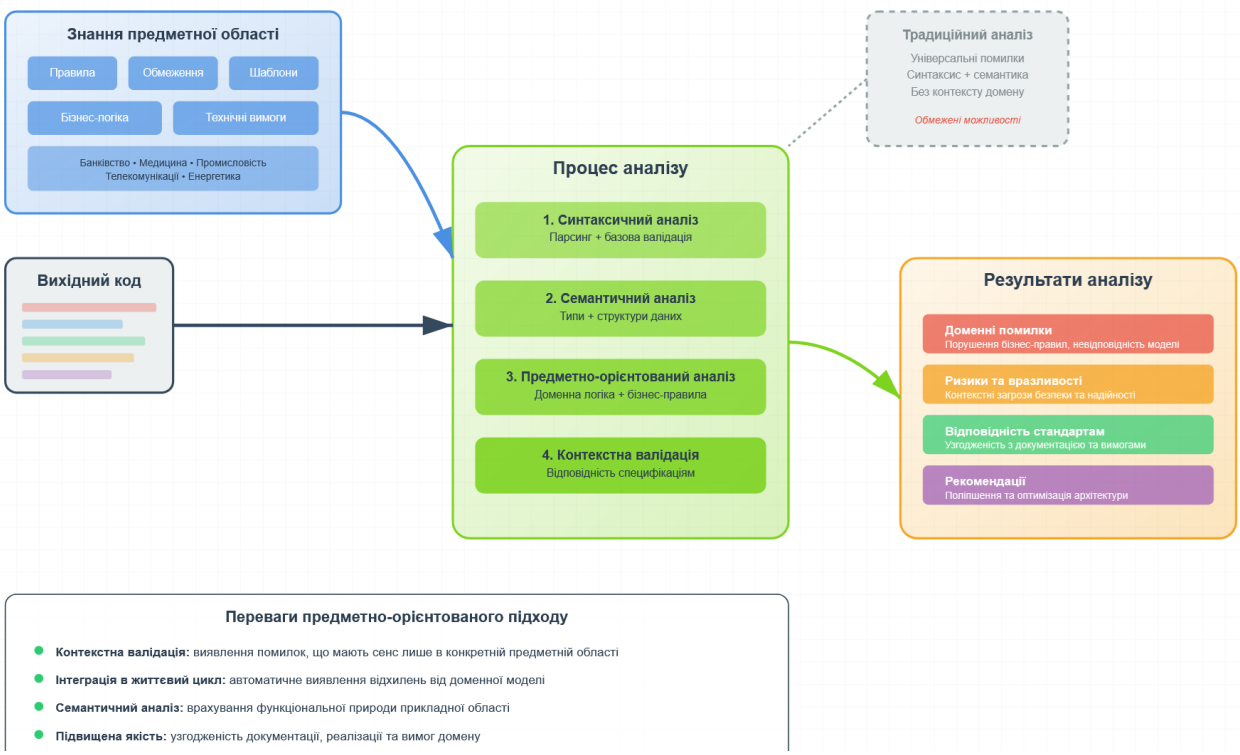

Рисунок 1.1 – Переваги предметно-орієнтованого підходу

Таким чином, аналіз вихідного коду не є ізольованою процедурою, а відіграє стратегічну роль у побудові надійного програмного забезпечення, підвищенні ефективності процесів розробки, забезпеченні безпеки та підтримуваності системи. Його використання є критично важливим у контексті зростання складності програмних систем, розширення кількості взаємодіючих компонентів та необхідності оперативного реагування на технічний борг і загрози безпеці.

Аналіз програмного коду є невід’ємною частиною процесу створення якісного програмного забезпечення та забезпечує виявлення і усунення технічних, логічних і архітектурних недоліків ще на етапі розробки. У

класичному розумінні він охоплює широкий спектр методів, спрямованих на синтаксичну, семантичну, структурну і поведінкову перевірку вихідного тексту програми з метою забезпечення його правильності, безпеки та ефективності. Проте в умовах зростаючої складності програмних систем та підвищених вимог до їхньої адаптивності і гнучкості, традиційні підходи до аналізу коду потребують розширення у бік предметної специфіки.

Предметно-орієнтований аналіз коду передбачає включення знань про конкретну предметну область, її правила, обмеження, шаблони бізнес-логіки і технічні вимоги безпосередньо в процедуру аналізу. Це дозволяє виявляти не лише загальноприйняті програмні помилки, а й такі, що мають сенс лише в контексті певної сфери, наприклад: банківських операцій, медичних обліків, систем управління промисловим обладнанням тощо. На відміну від універсального синтаксичного або семантичного аналізу, предметно-орієнтований метод дозволяє здійснювати валідацію логіки, відповідність коду предметним моделям, а також оцінювати потенційні ризики, притаманні конкретному типу задач.

Інтеграція предметно-орієнтованого підходу в аналіз вихідного коду посилює роль аналітичних процедур у життєвому циклі програмного забезпечення. Вона дозволяє вбудувати в процес розробки механізми виявлення відхилень від доменної моделі, перевірки відповідності реалізованої логіки до специфікацій, а також автоматичної інтерпретації можливих наслідків помилок з урахуванням контексту. У такому підході аналіз коду переходить від суто технічної процедури до інструменту, що враховує семантичну та функціональну природу прикладної області, для якої створюється програмний продукт.

Враховуючи зростаючу популярність предметно-орієнтованих підходів у програмній інженерії, особливо у контексті моделювання, DSL (domain-specific languages) та автоматизованої генерації коду, розробка спеціалізованого методу предметно-орієнтованого аналізу вихідного коду постає як актуальна наукова і практична задача. Вона дозволяє не лише

вдосконалити контроль якості, а й суттєво підвищити узгодженість між проектною документацією, програмною реалізацією і вимогами предметної області.

1.2 Основні завдання аналізу вихідних текстів програм

Аналіз вихідного коду є фундаментальною складовою сучасної практики розробки програмного забезпечення. Він спрямований не лише на виявлення очевидних синтаксичних чи логічних помилок, а й на глибшу перевірку якості, структури, надійності та відповідності програмного продукту встановленим вимогам. У класичному розумінні завдання аналізу коду охоплюють широкий спектр аспектів, починаючи з формального контролю правильності написання і закінчуючи оцінкою потенційної експлуатаційної поведінки програмного забезпечення. Проте з огляду на необхідність глибшої предметної інтеграції, ці завдання трансформуються та ускладнюються в межах предметно-орієнтованого підходу.

Рисунок 1.2 – Основні завдання аналізу вихідних текстів програм

Одним з базових завдань аналізу коду є забезпечення коректності програмної реалізації (рисунок 1.2). Це стосується перевірки дотримання синтаксичних норм, правил мов програмування, стилістичних вимог, стандартів оформлення та рекомендацій щодо написання чистого, читабельного коду. Проте сам по собі такий аналіз не дозволяє зробити висновки про відповідність реалізованої логіки очікуванням кінцевого користувача чи вимогам галузевих регламентів. Саме тому виникає потреба у перевірці семантичної узгодженості між реалізованим кодом і моделями предметної області. Предметно-орієнтований аналіз розширює класичне завдання перевірки, доповнюючи його оцінкою коректності реалізації з погляду функціонального значення та контексту використання.

Ще одним важливим завданням є виявлення вразливостей і дефектів, які можуть загрожувати надійності або безпеці програмного продукту. Йдеться не лише про типові помилки пам’яті чи конкурентного доступу, а й про логічні невідповідності, що можуть стати критичними саме в певній предметній сфері. Наприклад, відсутність перевірки нульового балансу в банківському програмному забезпеченні може бути значно серйознішою помилкою, ніж у загальній інформаційній системі. Отже, серед завдань предметно-орієнтованого аналізу також важливо виокремити контекстуальну релевантність – здатність виявляти не лише формальні, але й смислові дефекти.

До кола завдань також входить контроль відповідності коду стандартам, які є актуальними у межах певної галузі або проєкту. Це можуть бути як галузеві вимоги (наприклад, ISO/IEC 25010), так і внутрішні політики компанії. Предметно-орієнтований аналіз дозволяє адаптувати ці стандарти до специфіки застосування, автоматизувати їх перевірку і вбудувати відповідні критерії безпосередньо в аналітичну модель.

Крім того, серед завдань аналізу вихідного коду важливе місце займає оцінка підтримуваності та масштабованості. Якість програмного коду повинна забезпечувати можливість подальшої модифікації, адаптації до

нових вимог і інтеграції з іншими компонентами. У предметно-орієнтованому контексті це завдання набуває нового змісту, оскільки структурна відповідність архітектурі предметної області стає критично важливою умовою для забезпечення довготривалої життєздатності продукту.

1.3 Класифікація методів аналізу (статичний і динамічний аналіз)

У сучасній інженерії програмного забезпечення аналіз вихідного коду є неоднорідним за своїм характером, методами реалізації та сферою застосування. Класифікація методів аналізу коду дозволяє краще зрозуміти різні підходи до перевірки програмних систем, їхні сильні сторони, обмеження та потенційні області застосування. У класичній парадигмі прийнято виокремлювати два основних типи аналізу вихідних текстів програм: статичний та динамічний. Кожен із них має свою специфіку і призначення, однак у контексті предметно-орієнтованого підходу їх роль переосмислюється з позицій глибшої семантичної інтерпретації та прив’язки до логіки прикладної галузі.

Criterion	Static Analysis
Application phase	Before compilation or execution
Need for code execution	Not required
Error detection	Syntactic, logical, stylistic errors
Code coverage	Full (if whole project is analyzed)
Performance	Fast, efficient
Typical tools	SonarQube, ESLint, PVS-Studio, Coverity
Key advantages	Early detection, automation, CI integration
Main limitations	False positives, no runtime insight
Criterion	**Dynamic Analysis**
Application phase	During program execution
Need for code execution	Mandatory
Error detection	Runtime errors, memory leaks, race conditions
Code coverage	Limited to executed paths
Performance	May be slow, resource-intensive
Typical tools	Valgrind, Intel VTune, AddressSanitizer
Key advantages	Real behavior, precision, runtime defect detection
Main limitations	Requires environment, partial coverage

Рисунок 1.3 – Порівняльний аналіз методів

Статичний аналіз передбачає вивчення структури та властивостей програмного коду без його фактичного виконання. Цей тип аналізу застосовується переважно на етапах розробки та компіляції і спрямований на виявлення помилок синтаксису, логіки, порушення стилістичних правил, некоректних конструкцій та потенційно небезпечних елементів. Завдяки використанню таких підходів, як побудова дерев синтаксичного розбору, аналіз графів викликів, контроль потоку даних, а також перевірка на відповідність вимогам безпеки, статичний аналіз дозволяє виявити суттєвий обсяг дефектів до початку виконання програми. Проте він не враховує специфіку поведінки програми в реальному середовищі, що обмежує його можливості щодо виявлення динамічних або контекстно залежних помилок.

Динамічний аналіз, натомість, базується на спостереженні за поведінкою програми під час її виконання. Цей підхід дозволяє зафіксувати баги, які виявляються лише в реальних умовах, такі як витоки пам’яті, умови гонки, аномалії потоків, помилки у взаємодії з ресурсами чи некоректне оброблення зовнішніх даних. Динамічний аналіз є незамінним у випадках, коли необхідно дослідити відповідність функціонування програми очікуваному результату в конкретних прикладних умовах. У предметно-орієнтованому контексті динамічні методи набувають особливої цінності, оскільки дозволяють емпірично перевіряти, наскільки реалізація програмної логіки відповідає моделям поведінки об'єктів предметної області.

Зіставлення двох підходів демонструє, що кожен з них охоплює свою частину проблемного простору. Статичний аналіз є потужним інструментом для запобігання помилок на ранніх етапах, тоді як динамічний необхідний для верифікації поведінкових аспектів у контексті практичного використання. Проте предметно-орієнтований аналіз вимагає доповнення традиційної класифікації новим рівнем – семантичного або доменно-специфічного аналізу, який фокусується на перевірці відповідності коду моделям та обмеженням предметної області. Такий підхід виходить за межі перевірки формальних властивостей і вимагає створення аналітичного

інструментарію, здатного розпізнавати й інтерпретувати семантичні структури, втілені у коді.

Узагальнюючи (рисунок 1.3), класифікація методів аналізу вихідного коду не обмежується лише дихотомією «статичний – динамічний». У рамках предметно-орієнтованого методу вона має включати ще один – семантичний – рівень, який фіксує не лише технічну, але й концептуальну правильність реалізації. Саме поєднання всіх трьох вимірів дозволяє побудувати повноцінний метод аналізу, що забезпечує високу якість, безпеку та відповідність програмного забезпечення очікуванням предметної сфери.

1.4 Предметно-орієнтований підхід до аналізу вихідного коду

У традиційних підходах до аналізу програмного забезпечення ключовий фокус зосереджується переважно на виявленні помилок, дефектів або порушень у структурі коду з точки зору формальної коректності, продуктивності чи відповідності стандартам програмування. Проте в умовах сучасної розробки, яка охоплює складні, розподілені й контекстно-залежні програмні системи, виникає потреба у глибшому, семантичному розумінні того, наскільки реалізований код відображає сутність і логіку предметної області, для якої він створюється. Саме на цьому етапі актуалізується предметно-орієнтований підхід до аналізу вихідного коду, що дозволяє інтегрувати знання про специфіку прикладної галузі безпосередньо в процес валідації програмного тексту.

Предметно-орієнтований аналіз вихідного коду передбачає не лише перевірку синтаксичних і семантичних конструкцій, а й інтерпретацію їх у контексті моделі предметної області. Його завданням є виявлення невідповідностей між тим, як система повинна поводитися згідно з доменною специфікацією, та тим, як ця поведінка реалізована в програмному коді. Такий підхід вимагає створення зв’язку між абстракціями програмного рівня (класи, методи, функції, модулі) і абстракціями реального світу (процеси,

об’єкти, правила, обмеження), які лежать в основі предметної області. Отже, предметно-орієнтований аналіз переходить від аналізу того, як написано, до аналізу того, що саме реалізовано.

Особливістю предметно-орієнтованого підходу є його гнучкість і здатність адаптуватися до конкретних доменних моделей, що визначають обсяг знань, термінологію, структуру відношень та логіку взаємодії об'єктів у заданій галузі. Це дає змогу реалізувати аналіз коду як перевірку його відповідності формалізованій онтології або UML-моделі, а також проводити діагностику на предмет дотримання предметно зумовлених інваріантів, обмежень чи обов’язкових сценаріїв поведінки. Такий підхід є особливо корисним у сферах, що вимагають суворого дотримання бізнес-логіки, нормативних регламентів або критичних вимог до функціональної відповідності (наприклад, медичне програмне забезпечення, фінансові сервіси, вбудовані системи управління).

Інтеграція предметно-орієнтованого аналізу у життєвий цикл розробки програмного забезпечення передбачає створення або імпорт формалізованої моделі предметної області, після чого відбувається автоматизоване або напівавтоматизоване зіставлення структури й логіки вихідного коду з цією моделлю. У цьому контексті особливу роль відіграють інструменти, що здатні поєднувати можливості статичного аналізу з розпізнаванням доменних патернів, а також інтелектуальні системи на основі машинного навчання, які можуть навчатися на типових проєктах галузі та розпізнавати відхилення від звичних архітектурних чи логічних схем.

Отже, предметно-орієнтований підхід розширює горизонти аналізу вихідного коду, перетворюючи його з інструменту технічної перевірки на механізм семантичної відповідності, що дозволяє забезпечити не лише формальну коректність, а й концептуальну адекватність програмного продукту. У контексті даної кваліфікаційної роботи, запропонований метод предметно-орієнтованого аналізу є центральним елементом дослідження, покликаним інтегрувати знання про предметну галузь у процедуру

верифікації коду з метою підвищення точності, ефективності та надійності програмних систем.

1.5 Роль предметно-орієнтованого аналізу у підвищенні якості програмного забезпечення

У сучасній практиці розробки програмного забезпечення проблема якості займає одне з центральних місць, адже програмні системи дедалі частіше функціонують у критичних галузях, де навіть незначна помилка може мати серйозні наслідки. З огляду на це, зростає потреба в таких методах аналізу, які дозволяють не лише виявляти технічні дефекти, а й глибоко оцінювати відповідність програмного коду специфіці та логіці предметної області. Саме цю функцію виконує предметно-орієнтований підхід, що розглядається як засіб стратегічного забезпечення якості на рівні семантики, архітектури та поведінки програмного продукту.

Роль предметно-орієнтованого аналізу полягає насамперед у тому, що він забезпечує можливість автоматизованої перевірки узгодженості коду з моделлю доменної логіки, дозволяючи виявляти неочевидні дефекти, які не потрапляють у поле зору стандартних статичних або динамічних аналізаторів. Такі дефекти включають, зокрема, порушення інваріантів, відхилення від архітектурних шаблонів, некоректну інтерпретацію бізнес-правил або відсутність необхідних перевірок у логіці обробки подій. Таким чином, предметно-орієнтований аналіз розширює спектр перевірок і підвищує рівень покриття потенційно проблемних зон у програмному коді.

Крім того, важливою перевагою цього підходу є його здатність адаптуватися до конкретного домену за рахунок використання предметних онтологій, моделей процесів або структур даних. Це дозволяє розробникам уникнути помилок, які виникають через недостатнє розуміння специфіки предметної області, особливо у випадках, коли команда розробки не володіє повним обсягом експертних знань. Інструменти предметно-орієнтованого

аналізу можуть виступати як засіб передачі знань між експертами та розробниками, забезпечуючи автоматизований контроль відповідності реалізованого коду очікуваній функціональності.

Істотним чинником підвищення якості є також вплив предметно-орієнтованого аналізу на архітектурну узгодженість програмного забезпечення. Аналізуючи відповідність між архітектурними рішеннями та предметною моделлю, можливо запобігти структурним спотворенням, технічному боргу та зниженню підтримуваності коду. Це особливо важливо для довготривалих проєктів або систем, які розвиваються в умовах змін вимог, оскільки предметна модель виступає стабілізуючим фактором, що дозволяє зберігати логічну цілісність.

І, нарешті, предметно-орієнтований аналіз сприяє зростанню довіри до програмного забезпечення з боку користувачів і замовників, оскільки забезпечує прозорість і контроль над тим, як бізнес-логіка втілюється у функціональність програми. Завдяки можливості формалізованого порівняння очікуваної та фактичної поведінки системи він може бути використаний як доказ відповідності ПЗ нормативним вимогам, галузевим стандартам чи політикам безпеки.

Таким чином, предметно-орієнтований аналіз не є лише технічним інструментом перевірки, а виступає невід’ємним компонентом системного забезпечення якості програмного забезпечення, що охоплює всі аспекти – від точності реалізації бізнес-вимог до підтримуваності й довготривалої стабільності архітектури. У рамках даної кваліфікаційної роботи саме ця особливість предметно-орієнтованого підходу буде використана як концептуальна основа для розробки і реалізації методу аналізу вихідних текстів програм у середовищі Google Colab.

2 СТАТИЧНІ МЕТОДИ АНАЛІЗУ ПРОГРАМНОГО КОДУ

У процесі забезпечення якості програмного забезпечення одним із найважливіших етапів є виявлення дефектів на ранній стадії життєвого циклу, ще до того, як програма буде скомпільована або запущена. У цьому контексті статичний аналіз виступає ефективним інструментом, здатним значно знизити витрати на виявлення та виправлення помилок. Його суть полягає у дослідженні вихідного коду без його виконання, що дозволяє здійснити перевірку широкого спектра характеристик – від синтаксичної правильності до відповідності стандартам кодування та логічної узгодженості.

На відміну від динамічних методів, які потребують запуску програмного коду, статичний аналіз працює з текстовим представленням програм, застосовуючи алгоритми побудови дерев синтаксичного розбору, графів викликів функцій, аналізу потоку даних і контролю, інтерпретації типів, а також перевірки інваріантів. Завдяки цьому можливо досягти високої точності у виявленні помилок, які стосуються архітектури програми, використання некоректних конструкцій, зайвих чи дубльованих фрагментів коду, а також стилістичних відхилень. У результаті аналізу формується набір метрик, попереджень або помилок, які є основою для рефакторингу або подальшої перевірки з боку розробника.

Особливого значення набуває статичний аналіз у середовищах, орієнтованих на безперервну інтеграцію та розгортання (CI/CD), де автоматизовані інструменти, вбудовані в пайплайни розробки, забезпечують безперервний моніторинг якості коду. Прикладами таких інструментів є SonarQube, Coverity, PVS-Studio, ESLint, які не лише виявляють помилки, а й формують репорти щодо складності коду, наявності технічного боргу, охоплення тестами та дотримання стандартів проекту. Вбудовування цих засобів у розробницьке середовище дозволяє досягти автоматизованої

перевірки на кожному коміті коду та мінімізувати людський фактор у процесі контролю якості.

У межах дослідження предметно-орієнтованого підходу до аналізу програмного забезпечення статичні методи становлять фундамент для подальшої семантичної обробки. Вони дозволяють здійснити попередню фільтрацію помилок і сформувати структуроване представлення коду, на якому базуються подальші етапи предметної інтерпретації. Особливо важливим є застосування статичних засобів у сфері безпеки, де виявлення потенційно небезпечних операцій, необроблених виключень або порушень доступу до ресурсів є критичним фактором надійності.

У сучасних умовах реалізація інструментів статичного аналізу дедалі частіше здійснюється з використанням платформ для обчислень у хмарному середовищі. Зокрема, середовище Google Colab, що підтримує Python та інтеграцію з бібліотеками для синтаксичного та семантичного аналізу, є зручним інструментом для створення прототипів таких засобів. У подальших розділах буде розглянуто можливість побудови предметно-орієнтованого аналізатора, який, спираючись на результати попереднього статичного розбору, зможе ідентифікувати логічні відхилення та порушення відповідності вимогам предметної області.

2.1 Аналіз літературних джерел

Проблематика аналізу вихідних текстів програмного забезпечення залишається актуальною у сфері забезпечення якості, тестування та інформаційної безпеки. У науковій літературі аналіз коду здебільшого поділяється на статичні, динамічні та гібридні підходи, із дедалі більшою увагою до автоматизованих засобів та методів, що базуються на машинному навчанні.

У дослідженні [1] представлено внутрішній інструмент компанії Google під назвою Tricorder, який реалізує масштабовану платформу для статичного

аналізу коду. Метою цієї системи є виявлення помилок і вразливостей на ранніх етапах розробки, до фактичного виконання програм. Tricorder охоплює декілька мов програмування, має модульну архітектуру для підключення спеціалізованих аналізаторів і забезпечує безперервне оновлення правил перевірки. Інструмент тісно інтегрується в робочі процеси розробки, автоматизуючи зворотний зв’язок і підвищуючи якість коду в умовах інтенсивної розробки.

Інше дослідження [2] зосереджено на досвіді використання FindBugs у проектах на Java у рамках ініціативи Google Fixit. Протягом цього заходу розробники приділяли увагу виключно усуненню дефектів, виявлених інструментом. Було проаналізовано тисячі помилок, більшість із яких стосувалися роботи з null-значеннями, неповної ініціалізації та шаблонів, що створюють загрозу безпеці. Навіть у добре протестованих системах було виявлено значну кількість дефектів, що раніше залишались непоміченими. Дослідження підтверджує ефективність регулярного застосування статичного аналізу без додаткових витрат на тестування.

У роботі [3] підкреслюється важливість статичного аналізу з погляду безпеки. Наведено приклади поширених вразливостей, таких як буферні переповнення, SQL-ін’єкції, XSS, які можуть бути виявлені ще до запуску програми за допомогою інструментів типу Fortify або Coverity. Автори акцентують увагу на необхідності інтеграції таких засобів у щоденну практику безпечної розробки, а також на важливості навчання персоналу для ефективного їх використання.

Емпіричне дослідження [4] спрямоване на аналіз характеру помилок у популярному open-source програмному забезпеченні. На основі даних з репозиторіїв таких проєктів, як Mozilla, Apache, Eclipse, автори класифікували помилки за джерелами та наслідками. Основною причиною дефектів визначено людський чинник, зокрема неправильні припущення та складну взаємодію між компонентами. Особливий акцент зроблено на логічні помилки, які важко виявити засобами тестування. У підсумку робота

доводить ефективність поєднання статичних і динамічних методів разом із залученням користувачів для покращення якості ПЗ.

У межах дослідження [5] запропоновано фреймворк VulLibMiner, що орієнтований на виявлення вразливих сторонніх Java-бібліотек, спираючись лише на текстові описи вразливостей. Враховуючи широке використання сторонніх компонентів у сучасному програмному середовищі, дана розробка є особливо релевантною. Автори передбачають потенційне масштабування інструмента на інші мови програмування, інтеграцію з середовищами розробки (IDE) та автоматичну генерацію рекомендацій для оновлення небезпечних залежностей.

Публікація [6] представляє метод code2vec, який демонструє застосування глибокого навчання до аналізу програмного коду. Система перетворює код у векторне представлення шляхами синтаксичного дерева, що дозволяє нейронній мережі виявляти патерни, характерні для певних категорій помилок. Code2vec демонструє синтез традиційного синтаксичного аналізу з можливостями штучного інтелекту і вказує на напрям майбутнього розвитку інтелектуальних систем аналізу.

Сукупно всі розглянуті джерела засвідчують важливість глибокого та системного аналізу вихідного коду як одного з головних інструментів підвищення якості та безпеки програмного забезпечення. Дані свідчать, що використання статичного аналізу в рамках автоматизованих процесів CI/CD (як, наприклад, у Google або SAP) значно знижує технічний борг і підвищує стабільність релізів. Водночас динамічні методи дозволяють виявити проблеми продуктивності, багатопоточності й витоків ресурсів, недоступні для статичного аналізу. Зростання кількості залежностей і сторонніх бібліотек лише підсилює потребу в автоматизованих інструментах, що здатні адаптуватися до динаміки сучасного програмного ландшафту..

2.2 Синтаксичний аналіз і перевірка відповідності стандартам кодування

Синтаксичний аналіз є базовим етапом у процесі статичного аналізу програмного коду, що полягає у формальному вивченні структури вихідного тексту згідно з граматикою відповідної мови програмування. Його основною метою є побудова дерева синтаксичного розбору, яке відображає логічну ієрархію конструкцій у коді та дозволяє виявити синтаксичні помилки ще до фази компіляції. У сучасних системах розробки синтаксичний аналіз забезпечує перехід від вихідного тексту до формалізованого внутрішнього представлення, що уможливлює подальший семантичний розбір, аналіз залежностей і виконання інших форм контролю якості.

Особливу роль синтаксичний аналіз відіграє у предметно-орієнтованих методах, де правильність побудови коду є необхідною умовою для інтерпретації його відповідності доменній логіці. Помилки в структурі коду унеможливлюють або спотворюють подальшу обробку, що особливо критично при побудові систем валідації бізнес-правил або логіки функціонування складних інформаційних систем. Оскільки у предметно-орієнтованому аналізі важливо не лише визначити помилку, а й співвіднести її з порушенням конкретного доменного обмеження, синтаксична коректність коду є передумовою для релевантного тлумачення.

Не менш важливою складовою є перевірка відповідності стилю і стандартам кодування. У багатьох компаніях і проектах прийняті суворі правила написання коду, які регламентують іменування змінних і функцій, структуру умовних операторів, форматування, обробку винятків та інші аспекти. Дотримання таких стандартів сприяє підтримуваності, читабельності та уніфікації коду в команді. Для автоматизації перевірки використовуються інструменти лінтингу (linting), які аналізують код на предмет стилістичних помилок та відхилень від прийнятого формату.

Зокрема, у Python популярними засобами є PyLint, Flake8, Black, які

забезпечують як базову синтаксичну перевірку, так і поглиблений аналіз стилістичних відповідностей. У контексті Google Colab ці інструменти можуть бути інтегровані безпосередньо в середовище розробки, що дозволяє реалізовувати автоматизований контроль на рівні кожної виконуваної комірки. Це відкриває широкі можливості для створення предметно-орієнтованих засобів аналізу, які не лише виявляють формальні порушення, а й накладають додаткові логічні обмеження згідно з вимогами предметної області.

У розробці методів предметно-орієнтованого аналізу синтаксичний контроль може бути розширений засобами парсингу, побудови абстрактного синтаксичного дерева (AST) та моделювання граматичних шаблонів, що відповідають типовим патернам предметної області. Таке представлення забезпечує основу для виявлення структурних відхилень, що не завжди трактуються як синтаксичні помилки, але суперечать очікуваній логіці у певній прикладній сфері.

2.3 Семантичний аналіз: виявлення логічних помилок та некоректних конструкцій

Семантичний аналіз програмного коду є наступним логічним етапом після синтаксичного розбору, орієнтованим не лише на перевірку структури коду, а й на глибше осмислення його змісту. Якщо синтаксичний аналіз визначає, чи є певна послідовність інструкцій граматично правильною, то семантичний аналіз перевіряє відповідність цих інструкцій значенню та контексту використання у межах обраної мови програмування. Він фокусується на виявленні логічних помилок, що виникають унаслідок неправильного використання змінних, некоректних типів даних, порушень правил доступу до об'єктів, невідповідності сигнатур викликів функцій, неправильного контролю потоку виконання та багатьох інших факторів, що прямо не суперечать синтаксису, але призводять до помилкової поведінки

програми.

У контексті предметно-орієнтованого аналізу вихідних текстів програм семантична перевірка набуває ще більшої значущості. Це зумовлено тим, що основною метою такого підходу є не лише технічна коректність коду, а його відповідність специфічним правилам і обмеженням, що випливають із прикладної області. Логіка роботи програми має бути не лише правильною з точки зору обраної мови, але й узгодженою з концептуальною моделлю предметної сфери, що аналізується. Семантичні помилки в цьому випадку можуть означати порушення логіки бізнес-процесів, хибне трактування вхідних або вихідних даних, невірну взаємодію між компонентами або порушення функціональної послідовності операцій.

З технічної точки зору семантичний аналіз реалізується через побудову таблиць імен, аналіз типів, перевірку області видимості змінних, відповідність операцій їх операндам, контроль викликів функцій та коректність параметризації. Інструменти семантичного аналізу часто реалізуються у складі компіляторів або спеціалізованих засобів валідації коду, таких як CodeQL, Infer, або статичних аналізаторів із підтримкою правил перевірки на рівні семантики.

У рамках предметно-орієнтованого підходу важливо не лише фіксувати загальні логічні помилки, а й адаптувати правила семантичного аналізу відповідно до доменної специфіки. Це означає необхідність розширення аналізатора спеціальними правилами, які відображають контекст предметної області. Наприклад, для фінансових систем перевіряється коректність обробки валютних операцій, для медичних – послідовність взаємодії з базами пацієнтів, для освітніх – відповідність логіки нарахування балів регламентованій моделі оцінювання.

Інтеграція таких перевірок у середовище Google Colab забезпечує можливість реалізувати адаптивні модулі семантичного контролю з використанням інтерпретованих мов, таких як Python. Завдяки доступності бібліотек для побудови абстрактного синтаксичного дерева (AST), аналізу

потоку виконання та реалізації власних семантичних правил, Colab дозволяє реалізувати кастомізовану систему валідації, яка буде реагувати на виявлення не лише загальних помилок, а й порушення предметно-орієнтованої логіки.

2.4 Огляд інструментів статичного аналізу

Сучасна практика розробки програмного забезпечення передбачає широке застосування інструментів статичного аналізу, які забезпечують раннє виявлення помилок та підвищення якості коду ще до етапу його виконання. Ці інструменти (рисунок 2.1) стали невіддільною складовою професійного середовища розробника та активно інтегруються у конвеєри безперервної інтеграції. У контексті предметно-орієнтованого аналізу такі засоби дозволяють не лише виявляти технічні недоліки, але й розширювати аналітичну модель під специфіку прикладної області, що є особливо важливим при розробці галузевих рішень.

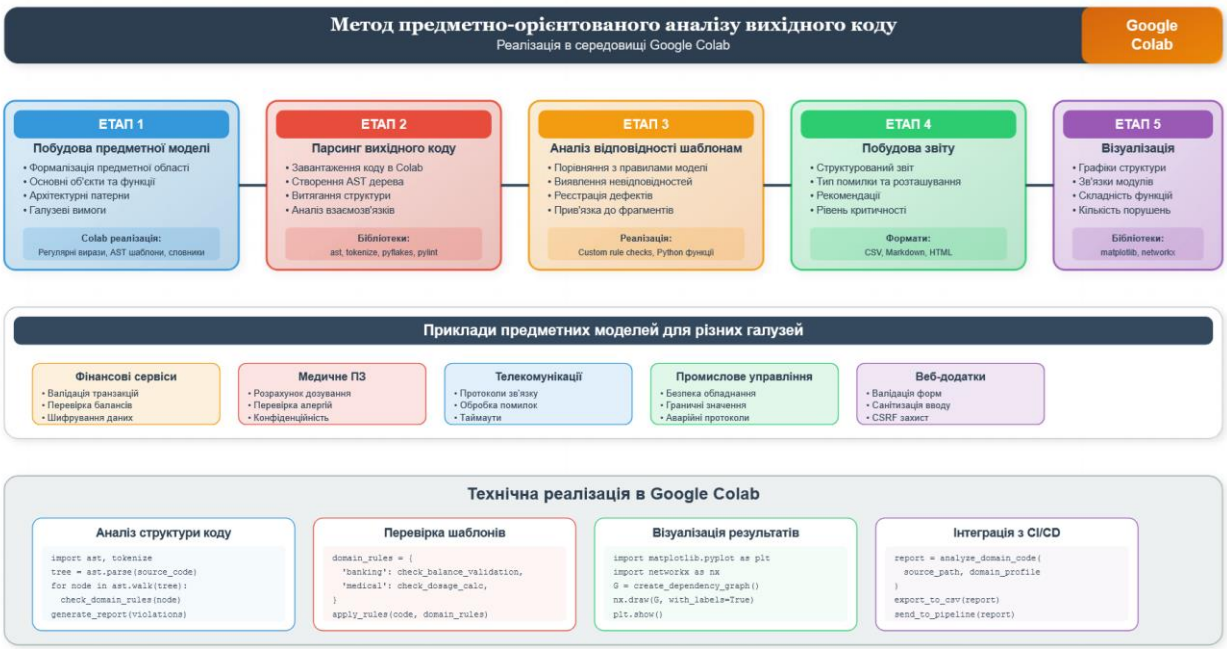

Рисунок 2.1 – Інструменти

Одним із найпоширеніших засобів є SonarQube – платформа з відкритим вихідним кодом, яка підтримує десятки мов програмування. Вона

забезпечує повноцінну перевірку коду за критеріями якості, безпеки, наявності дублікатів, складності, відповідності стилю та наявності вразливостей. SonarQube інтегрується з більшістю сучасних інструментів CI/CD (Jenkins, GitLab CI, Azure DevOps) і дозволяє здійснювати наскрізний контроль коду від моменту коміту до релізу. Система надає зручну візуалізацію метрик, можливість налаштовувати правила перевірки відповідно до проектних стандартів та підтримує створення спеціалізованих плагінів для адаптації під предметно-орієнтовані задачі.

Ще одним потужним інструментом є Coverity – комерційний продукт від Synopsys, що відзначається високим рівнем точності та підтримкою складних корпоративних середовищ. Coverity аналізує код на основі побудови міжпроцедурних графів контролю потоку, що дозволяє виявляти логічні помилки, порушення безпеки та дефекти доступу до пам’яті. Його особливістю є здатність до масштабування у великих проєктах та мінімальна кількість хибнопозитивних результатів. Для предметно-орієнтованого аналізу Coverity може бути доповнений власними сценаріями аналізу або інтегрований із політиками контролю відповідності галузевим стандартам (наприклад, MISRA для автомобільного ПЗ або CERT для безпеки).

Інструмент PVS-Studio, у свою чергу, є одним із найточніших аналізаторів для C, C++, C# та Java. Його основна перевага полягає в здатності виявляти нетривіальні помилки, які часто залишаються непоміченими іншими засобами, зокрема пов’язані з арифметичними помилками, типізацією, некоректною логікою умовних операторів тощо. PVS-Studio активно використовується для аналізу промислових систем і підтримує інтеграцію з Microsoft Visual Studio, CLion, Rider, а також може бути використаний як частина автоматизованих скриптів аналізу. У межах предметно-орієнтованої перевірки цей інструмент дозволяє налаштовувати правила аналізу відповідно до специфіки конкретного домену, що відкриває можливості для валідації логіки предметної області.

У контексті реалізації в середовищі Google Colab, пряме використання

зазначених інструментів є обмеженим через специфіку їх архітектури, однак їх функціональні еквіваленти або спрощені інтерпретації можуть бути реалізовані з використанням бібліотек для аналізу абстрактного синтаксичного дерева, таких як ast, pylint, flake8, bandit або mypy для Python. Це дає змогу створити експериментальне середовище для побудови кастомізованого статичного аналізатора з підтримкою предметно-орієнтованих правил, що може бути доповнено елементами машинного навчання для підвищення гнучкості та точності перевірок.

2.5 Розробка методу предметно-орієнтованого аналізу вихідних текстів програм

У рамках даного дослідження запропоновано метод предметно-орієнтованого аналізу вихідного коду, який має на меті поєднати переваги статичних методів аналізу з можливістю адаптації до специфіки певної прикладної галузі, що дозволяє точніше виявляти логічні, стилістичні та архітектурні відхилення, характерні саме для доменно-орієнтованих рішень. Метод базується на ідеї формування спеціалізованого профілю перевірки для вихідного тексту програми, який містить набір предметно релевантних правил, патернів і структур, виявлених із наявної експертної інформації, технічної документації або попереднього корпусу перевіреного коду. Реалізація цього методу виконується у середовищі Google Colab, що забезпечує зручну платформу для інтерактивної обробки, візуалізації результатів та інтеграції з бібліотеками Python.

Етап 1. Побудова предметної моделі.

На першому етапі здійснюється формалізація предметної області. Визначаються основні об'єкти, функції, архітектурні патерни та вимоги, характерні для типових програм у заданій галузі (наприклад, фінансові сервіси, медичне ПЗ, телекомунікаційні протоколи). Це дозволяє сформувати цільову модель поведінки коду, яка буде служити орієнтиром для подальшої

перевірки.

У Google Colab це реалізується як формування списку правил або шаблонів на основі регулярних виразів або структури абстрактного синтаксичного дерева (AST). Також на цьому етапі можливе використання кастомного словника ключових функцій, специфічних змінних або заборонених викликів.

Етап 2. Парсинг вихідного коду.

Далі здійснюється попередня обробка вихідного тексту програми. Вихідний код (переважно мов Python, але можливе розширення до JavaScript або Java) завантажується в середовище Colab, де за допомогою бібліотек ast, tokenize, pyflakes або pylint виконується його розбір. Створюється дерево синтаксичного аналізу, а також витягується структура функцій, класів, змінних, імпортів та їх взаємозв’язків.

У середовищі Colab цей етап дозволяє інтерактивно досліджувати структуру коду, здійснювати перевірку заздалегідь визначених предметних критеріїв, та зберігати результати в зручному табличному або графовому форматі.

Етап 3. Аналіз відповідності предметним шаблонам.

Цей етап є центральним у реалізації методу. Для кожного елементу синтаксичного дерева виконується порівняння з правилами предметної моделі. Якщо виявлено невідповідність (наприклад, порушення стандарту іменування, неправильна структура функції, наявність застарілих бібліотек чи відсутність необхідної обробки помилок), система реєструє дефект із прив’язкою до конкретного фрагмента коду.

У Google Colab для цього можуть бути використані умовні блоки перевірки (custom rule checks), які реалізуються як окремі Python-функції з доступом до об’єктної моделі AST.

Етап 4. Побудова звіту.

Усі виявлені відхилення збираються у структурований звіт, який містить інформацію про тип помилки, її розташування у коді, порушене

правило, рекомендацію та рівень критичності. Звіт генерується у форматах CSV, Markdown або HTML для подальшого перегляду або завантаження.

Цей етап дозволяє інтегрувати результати перевірки у пайплайни CI/CD або використовувати їх як частину документації про якість програмного коду.

Етап 5. Візуалізація та подальший аналіз.

Для полегшення інтерпретації результатів аналізу передбачено візуальне представлення структури коду та порушень за допомогою бібліотек matplotlib, networkx, graphviz або seaborn. Такі графіки відображають зв'язки між модулями, ступінь складності функцій, кількість порушень на модуль або клас.

Ця візуалізація реалізується безпосередньо у середовищі Colab, що забезпечує миттєвий зворотний зв’язок для розробника або аналітика.

3 РЕАЛІЗАЦІЯ МЕТОДУ ТА АНАЛІЗ РЕЗУЛЬТАТІВ

3.1 Програмна реалізація методу та вибір програмних засобів

У Google Colab робота нашого .ipynb-файлу з реалізацією предметно-орієнтованого методу аналізу вихідного коду відбувається у декілька логічно послідовних етапів, які автоматизовано поєднані у функціональний процес. Цей процес включає завантаження файлу, його попередню обробку, аналіз, виявлення помилок і побудову візуалізацій. Нижче подано детальний опис цього циклу з урахуванням усіх особливостей середовища Google Colab.

На початковому етапі користувачеві надається можливість обрати довільний .py-файл для аналізу за допомогою інтерактивного елементу завантаження файлу через files.upload(). Google Colab у цьому випадку забезпечує вебінтерфейс для вибору локального файлу з комп’ютера, після чого файл автоматично зберігається у файловій системі середовища виконання Colab.

Далі відбувається зчитування вмісту завантаженого файлу та його розбиття на окремі рядки. Код здійснює попередню обробку вмісту: усуває порожні рядки, видаляє зайві пробіли, а також дозволяє виявити основні показники, як-от кількість рядків, середню довжину рядка, кількість коментарів та наявність ключових конструкцій мови Python. Цей етап реалізовано у вигляді текстового аналізу, що дає можливість виявити поверхневі ознаки структурної складності або недоліки стилістики.

На наступному етапі застосовується спеціалізована логіка для виявлення поширених помилок. Система сканує код на предмет таких проблем, як:

- недоступні або неоголошені змінні;
- некоректне використання відступів (indentation errors);
- потенційні нескінченні цикли;

- некоректне використання try-except;
- дублювання назв змінних або функцій.

Усі знайдені порушення збираються у структурований звіт, який виводиться в інтерактивній формі безпосередньо в середовищі Colab. Це дозволяє розробнику швидко оцінити якість коду без потреби запускати програму.

Після цього виконується автоматизований аналіз структури коду з побудовою візуальних графіків. Зокрема, формується гістограма довжини рядків, яка дозволяє оцінити стилістичну узгодженість, а також розподіл довжин ідентифікаторів, що відображає характер іменування об’єктів у коді. Додатково може генеруватись діаграма частоти використання ключових слів або частотна карта викликів функцій, що допомагає глибше проаналізувати логіку побудови програми.

Фінальний етап передбачає формування загального зведення, яке містить висновки щодо основних виявлених проблем і рекомендації щодо вдосконалення коду. У разі потреби, файл може бути збережений у форматі з примітками, а результати експортуються у вигляді зображень або таблиць.

Представлені графіки відображають результати статистичного аналізу вихідного коду з погляду його структурної складності. Верхня гістограма демонструє розподіл довжини рядків коду, що дозволяє оцінити стилістичну однорідність написання та виявити потенційні відхилення, пов’язані з надмірною складністю окремих конструкцій. Нижній графік ілюструє розподіл довжин ідентифікаторів, який є показником читаємості та стандартизації іменування змінних, функцій або класів у програмі. Значна концентрація коротких ідентифікаторів може свідчити про недостатню інформативність назв, тоді як надмірна довжина – про можливе порушення принципів лаконічності. Обидва графіки є важливими для попередньої діагностики якості вихідного коду та формування рекомендацій щодо покращення стилю програмування.

3.2 Аналіз результатів

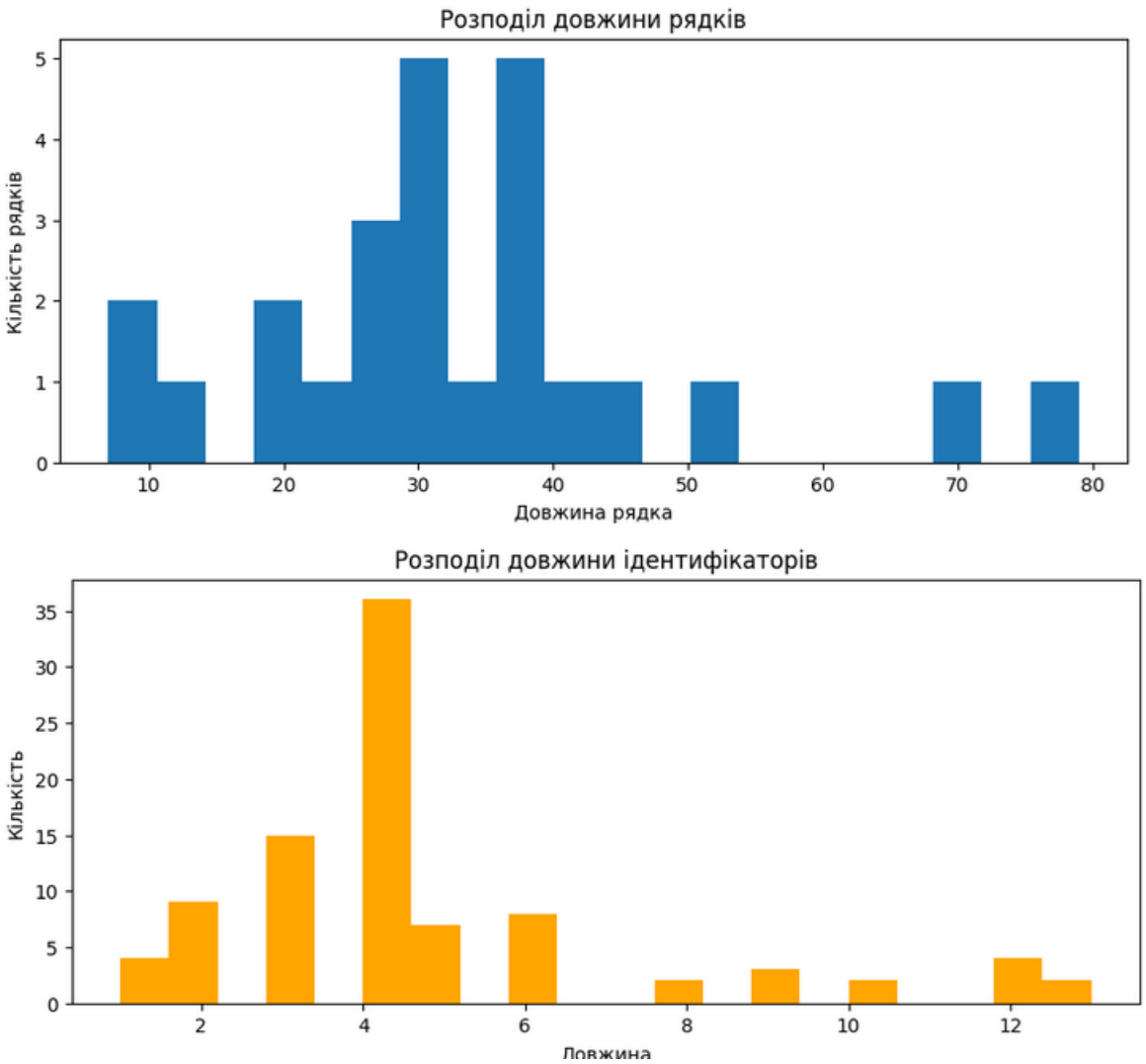

Рисунок 3.1 – Аналіз результатів

Представлені графіки (рисунок 3.1) відображають результати статистичного аналізу вихідного коду з погляду його структурної складності. Верхня гістограма демонструє розподіл довжини рядків коду, що дозволяє оцінити стилістичну однорідність написання та виявити потенційні відхилення, пов’язані з надмірною складністю окремих конструкцій. Нижній графік ілюструє розподіл довжин ідентифікаторів, який є показником читаємості та стандартизації іменування змінних, функцій або класів у програмі. Значна концентрація коротких ідентифікаторів може свідчити про недостатню інформативність назв, тоді як надмірна довжина – про можливе

порушення принципів лаконічності. Обидва графіки є важливими для попередньої діагностики якості вихідного коду та формування рекомендацій щодо покращення стилю програмування.

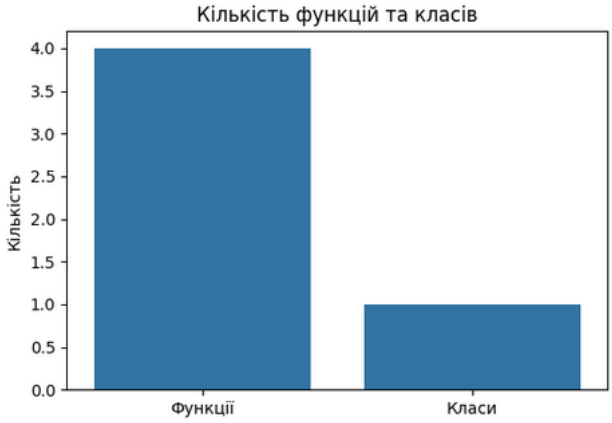

Рисунок 3.2 – Аналіз результатів

На зображенні (рисунок 3.2) представлено стовпчикову діаграму, що ілюструє кількісне співвідношення між функціями та класами у проаналізованому вихідному коді. Діаграма чітко демонструє домінування функціональних конструкцій над об'єктно-орієнтованими елементами, що свідчить про переважне використання процедурного стилю програмування. Така структурна характеристика може вказувати на обмежене використання механізмів інкапсуляції та повторного використання коду, притаманних класам, і є важливим індикатором для подальшої оцінки архітектурної якості програмного продукту.

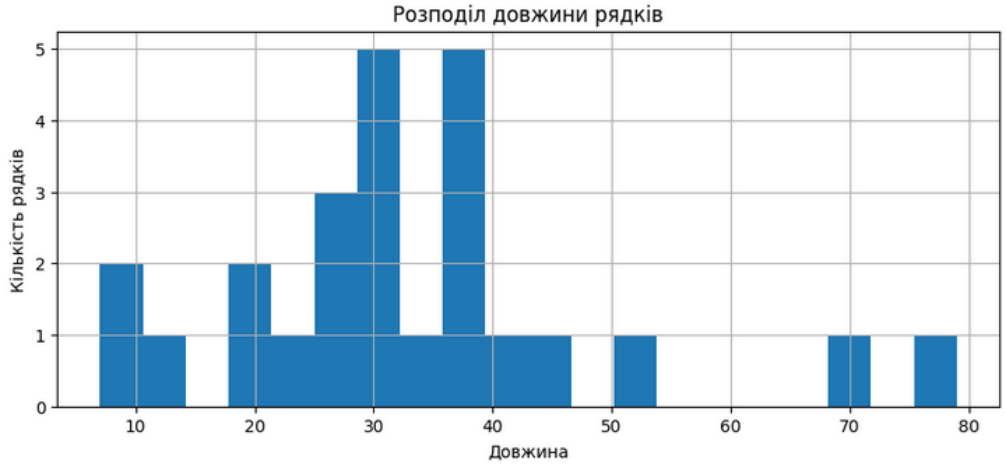

Рисунок 3.3 – Аналіз результатів

На наведеній гістограмі (рисунок 3.3) зображено розподіл довжини рядків у проаналізованому вихідному коді. Графік свідчить про нерівномірність у структурі коду: спостерігається скупчення рядків довжиною від 25 до 40 символів, що може вказувати на переважання коротких інструкцій або часту зміну логіки. Водночас наявність поодиноких довших рядків (до 80 символів) може бути симптомом складних виразів або недостатньої декомпозиції. Такий розподіл є важливим для оцінки читабельності коду, а також для виявлення потенційних зон, де слід застосувати рефакторинг для покращення стилю програмування.

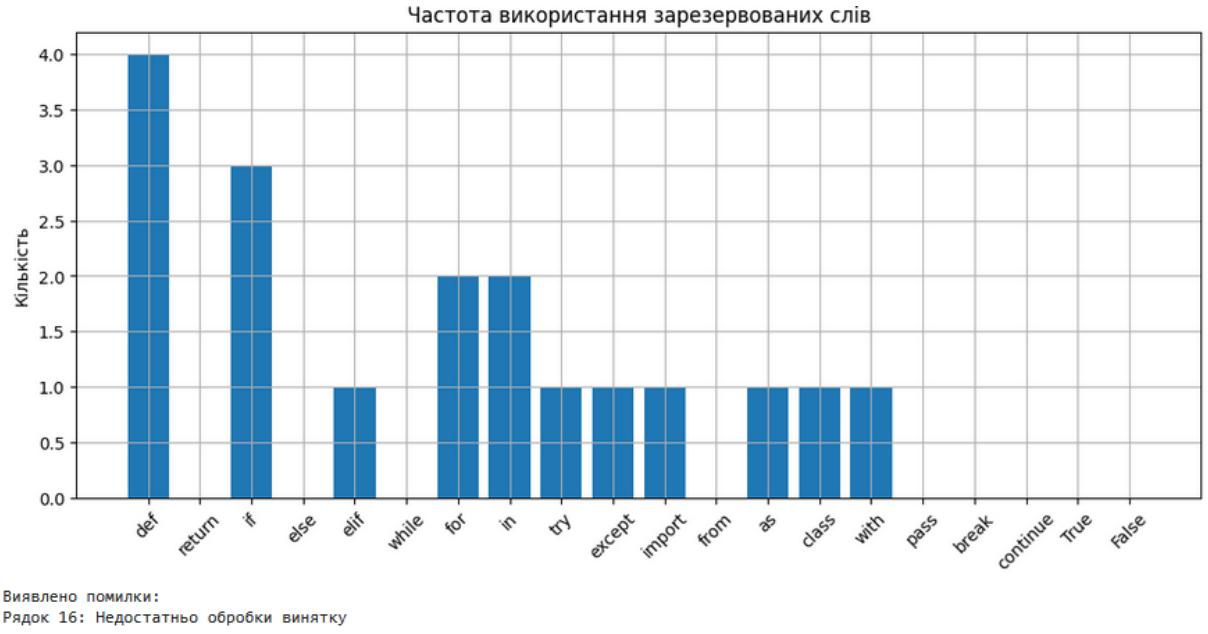

Рисунок 3.4 – Аналіз результатів

На графіку (рисунок 3.4) відображено частотний аналіз використання зарезервованих слів мови Python у досліджуваному коді. Найбільш уживаними є ключові слова def, return та if, що свідчить про високу концентрацію функціональних оголошень і умовної логіки. Зустрічаються також конструкції циклів (for, while), обробка винятків (try, except) та імпорт модулів, що вказує на достатньо складну структуру програмного фрагмента. Крім того, під графіком автоматично зафіксовано конкретну помилку – недостатню обробку винятку в рядку 16, що виявлено шляхом аналізу синтаксичних конструкцій. Такий підхід дозволяє не лише здійснювати статистичний огляд, але й оперативно локалізувати потенційно уразливі місця в коді.

ВИСНОВКИ

У результаті виконаної роботи було розроблено, реалізовано та протестовано метод предметно-орієнтовного аналізу вихідного коду програм з використанням інструментальних засобів Python у середовищі Google Colab. Запропонований підхід поєднує елементи класичного статичного аналізу з можливостями семантичного розбору, простежування структури коду та візуалізації його характеристик, що дозволяє ефективно виявляти низку поширених помилок, невідповідностей та стилістичних відхилень.

Аналіз проведено як на рівні синтаксичних елементів (ідентифікатори, ключові слова, довжина рядків), так і на рівні логіки структурування (класи, функції, конструкції контролю потоку), що забезпечило глибше розуміння внутрішньої організації програмного коду. Виявлені помилки класифікувались і супроводжувалися поясненням контексту, в якому вони виникають, що полегшує процес корекції та підвищує якість зворотного зв’язку.

Окрема увага приділена візуалізації результатів аналізу: побудовано гістограми розподілу довжини рядків, довжини ідентифікаторів, кількості функцій і класів, частоти вживання зарезервованих слів. Ці графічні представлення сприяють інтуїтивному розумінню загального стилю програмування, виявленню аномалій та можливих зон для рефакторингу.

Практична реалізація методу в Google Colab продемонструвала його доступність, відтворюваність та інтерактивність, що дозволяє використовувати розроблений інструмент як освітній, дослідницький або допоміжний засіб у командній розробці програмного забезпечення. За рахунок підтримки автоматичного завантаження користувацького коду та подальшого аналізу, система є гнучкою і масштабованою.

За результатами роботи опубліковано статтю в фаховому виданні [8].

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. C. Sadowski, J. van Gogh, Jaspan C., E. Söderberg, C. Winter. Tricorder: Building a program analysis ecosystem. ICSE '15: Proceedings of the 37th International Conference on Software Engineering, vol., 2015. P. 598-608. https://doi.org/10.1109/ICSE.2015.76 .

2. Ayewah N., Pugh W. The Google FindBugs fixit. ISSTA '10: Proceedings of the 19th international symposium on Software testing and analysis, 2010. P. 241-252. https://doi.org/10.1145/1831708.1831738 .

3. B. Chess, G. McGraw. Static Analysis for Security. IEEE Security & Privacy, vol. 2, No. 6, 2004. P. 76-79. https://doi.org/10.1109/MSP.2004.111 .

4. Z. Li, L. Tan, Y. Wang, S. Lu, Y. Zhou, C. Zhai. Have Things Changed Now?: An Empirical Study of Bug Characteristics in Modern Open Source Software. Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability, ASID 2006, San Jose, California, USA, October 21, 2006. 9 p. https://doi.org/10.1145/1181309.1181314.

5. T. Chen, L. Li, B. Shan, G. Liang, D. Li, Q. Wang, T. Xie. Identifying Vulnerable Third-Party Java Libraries from Textual Descriptions of Vulnerabilities and Libraries. Cornell University. Computer Science. Cryptography and Security, 2023. 23 p. https://doi.org/10.48550/arXiv.2307.08206 .

6. Uri Alon, Meital Zilberstein, Omer Levy, Eran Yahav. code2vec: Learning Distributed Representations of Code. Cornell University. Computer Science. Machine Learning, 2018. 23 p. https://doi.org/10.48550/arXiv.1803.09473 .

7. Flach P. A. Machine Learning: The Art and Science of Algoritms that Makes Sense of Data. Cambridge: Cambridge University Press, 2012. 291 p. https://doi.org/10.1017/CBO9780511973000 .

8. S. Kuzhel, A. Lytvynov, O. Pliekhov. METHODS FOR ANALYZING SOFTWARE SOURCE CODE. Системи управління, навігації та зв’язку, вип.3.

Полтава, 2025. С. 81-86.