МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Західноукраїнський національний університет
Факультет комп'ютерних інформаційних технологій
Кафедра комп'ютерної інженерії

Тимчук Євген Іванович

«Алгоритм оцінки якості програмного коду на основі структурного аналізу / Software code quality assessment algorithm based on structural analysis»

спеціальність: 123 - Комп'ютерна інженерія
освітньо-професійна програма - Комп'ютерна інженерія
Кваліфікаційна робота

Виконав студент групи КІм-21
Є.І. Тимчук
_____

Науковий керівник:
к.т.н., доц. Ю.М. Батько
_____

Кваліфікаційну роботу допущено
до захисту:

"___" _____ 20___ р.

Завідувач кафедри
_____Л. О. Дубчак

Тернопіль – 2023

РЕЗЮМЕ

Кваліфікаційна робота на тему “Алгоритм оцінки якості програмного коду на основі структурного аналізу ” зі спеціальності 123 «Комп’ютерна інженерія» освітнього ступеня «магістр» написана обсягом 85 сторінок та містить 26 ілюстрацій, 5 таблиць, 2 додатки та 60 джерел за переліком посилань.

Метою роботи є розробка алгоритму аналізу та оцінки якості програмного коду на основі структурного аналізу.

Методи досліджень. Для розв’язання поставлених задач у кваліфікаційній роботі використано методи: семантичного аналізу (для попереднього аналізу текстів програмних кодів); оцінки програмного коду (для виділення кільісних та якісних ознак текстів програмних кодів); об’єктно-орієнтованого програмування (для проектування програмного додатку аналізу та оцінки якості програмних кодів).

Результати дослідження: алгоритм оцінки якості програмних кодівна основі семантичного аналізу, програмний додаток аналізу та оцінки якості текстів програмних кодів.

Результати роботи можуть бути використані в створенні нових систем автомативаного оцінювання знань, системах оцінки рівня знань та практичних навичок розробників програмних кодів, для наукових досліджень та в навчальному процесі.

Орієнтовні напрямки розвитку досліджень: розроблення алгоритмів автоматичної корекції програмних кодів для підвищення рівня їх читабельності, створення нових програмних засобів та моделей для навчання.

КЛЮЧОВІ СЛОВА: СЕМАНТИЧНИЙ АНАЛІЗ, КІЛЬКІСНІ ТА ЯКІСНІ ОЗНАКИ, АВТОМАТИЗОВАНІ СИСТЕМИ ОЦІНЮВАННЯ.

RESUME

Graduate qualification work on “Software code quality assessment algorithm based on structural analysis” specialty 123 – Computer Engineering is 85 pages long and contains 26 illustrations, 5 tables, 2 appendices and 60 references.

The aim of the work is to develop an algorithm for analyzing and evaluating the quality of software code based on structural analysis.

Research methods. To solve the tasks in the qualification work, the following methods were used: semantic analysis (for preliminary analysis of program code texts); evaluation of software code (for selection of quantitative and qualitative features of software code texts); object-oriented programming (for designing a software application for analyzing and evaluating the quality of software codes).

Research results: an algorithm for evaluating the quality of software codes based on semantic analysis, a software application for analyzing and evaluating the quality of software code texts.

The results of the work can be used in the creation of new automated knowledge assessment systems, systems for assessing the level of knowledge and practical skills of developers of software codes, for scientific research and in the educational process.

Indicative directions of research development: development of algorithms for automatic correction of program codes to increase their readability, creation of new software tools and models for learning.

KEYWORDS: SEMANTIC ANALYSIS, QUANTITATIVE AND QUALITATIVE CHARACTERS, AUTOMATED EVALUATION SYSTEMS.

ЗМІСТ

ВСТУП

Актуальність роботи. Інструменти автоматичної перевірки вихідного коду допомагають оцінювати, контролювати та покращувати якість коду. Оскільки тільки ці інструменти вивчають кодову базу проекту програмного забезпечення, вони не помічають інші можливі фактори, які можуть вплинути на якість коду та оцінку технічної завершеності. Очевидним є той факт, що людський фактор пов’язаний з розробниками програмного забезпечення, наприклад кодування знання, комунікативні навички та досвід роботи в проекті мають певний вимірний вплив на якість коду.

Цифрова трансформація – викликана технологіями такими як гнучка розробка, AI (штучний інтелект), блокчейн і відкриті API, серед іншого – надає величезні соціально-економічні переваги. Цифрова трансвормація в індустрії програмного забезпечення вимагає скоротити час виходу на ринок і покращити взаємодію з клієнтами, операційну досконаліть та якість вихідної продукції. Ці вимоги сприяли появі додатків програмних технологій. Наприклад, інструменти автоматичної перевірки вихідного коду спрямовані на зменшення людських зусиль, що зазвичай затрачаються в процесі розробки. З іншого боку, системи керування версіями (VCS) допомагають створювати, підтримувати та розвивати кодову базу проектів. Крім того, VCS зберігає системні журнали та дані розробки, такі як дати фіксації, автори фіксації тощо.

Очевиднно, людські ресурси є критично важливим фактором навіть важливішим, ніж технології в середовищах цифрової трансформаці. Зокрема, розробники відіграють важливу роль для кінцевої якості програмного забезпечення. З цього приводу дослідження показали, що такі фактори розробника, як досвід, участь, компроміс і комунікація безпосередньо впливають на якість коду.

Інструменти автоматичної перевірки вихідного коду зазвичай вимірюють технічну завершеність і представляють відповідний показник, оскільки він обчислюється як сукупний результат інших відповідних факторів якості.

Тому задача проектування та реалізації алгоритму аналізу та оцінки якості програмного коду на основі структурного аналізу є актуальною.

Метою роботи є розробка алгоритму аналізу та оцінки якості програмного коду на основі структурного аналізу.

Для досягнення даної мети ставились наступні завдання:

- провести класифікацію та виділити особливості мов програмування;
- проаналізувати використання технологій структурного аналізу під час створення програмного коду;
- провести аналітичний огляд інтегрованих середовищ розробки програмних засобів;
- проаналізувати існуючі метрики оцінки якості програмного коду;
- розробити алгоритм оцінки якості програмного коду на основі структурного аналізу;
- реалізувати програмний додаток аналізу та оцінки програмного коду мовою високого рівня.

Об’єкт дослідження – процес створення програмних кодів.

Предмет дослідження – методи і алгоритми оцінки якості програмного коду.

Наукова новизна одержаних результатів визначається наступним чином:

- проведено комплексний аналіз та класифікацію метрик та алгоритмів визначення рівня якості програмного коду, що дозволило виділити групу критеріїв для обчислення рівня якості програмного коду;
- розроблено алгоритм оцінки якості програмного коду на основі семантичного підходу, що дозволило підвищити об’єктивність оцінювання програмного коду.

Практична цінність одержаних результатів полягає в тому, що:

- розроблено структуру додатку аналізу програмного коду та проведено її моделювання, що дозволило в подальшому програмно реалізувати та провести дослідження запропонованих алгоритмів;

- реалізовано програмне забезпечення аналізу та оцінки якості програмного коду на мовах високого рівня з використанням об’єктно-орієнтованого підходу та технологіями структурного аналізу.

Публікації та апробація до випускної кваліфікаційної роботи. За результатами наукових досліджень, проведених у випускній кваліфікаційні роботі, підготовлено тези доповіді «Оцінка якості програмного коду на основі аналізу рівня його читабельності» обсягом 1 сторінка на VIII Науково-практичній конференції молодих вчених і студентів «Інтелектуальні комп’ютерні системи та мережі», а також «Оцінка рівня агресії на основі аналізу текстових повідомлень» обсягом 1 сторінка на VIII Науково-практичній конференції молодих вчених і студентів «Інтелектуальні комп’ютерні системи та мережі».

1 СИСТЕМИ ТА ЗАСОБИ СТВОРЕННЯ ПРОГРАМНИХ КОДІВ

1.1 Мови програмування, їх класифікація та сфери застосування

Мови програмування – це комунікаційні системи, які дозволяють програмістам давати конкретні інструкції машинам і комп'ютерам, щоб вони могли виконувати дії, необхідні для досягнення певних цілей. За допомогою наявних мов програмування програмісти можуть спілкуватися з машинами, використовуючи їхню мову (програмні коди), за допомогою якої вони можуть писати інструкції у формі алгоритмів і повідомляти комп'ютерним системам, що вони хочуть, щоб вони робили.

На сьогоднішній день існує велика кількість мов програмування які є адаптованими під різні технічні завдання. Кожна з них має ряд особливостей, переваг та недоліків.

Один із способів класифікації мов полягає в розгляді рівня їхньої абстракції, тобто того, чи вони більш чи менш схожі на спосіб спілкування машин. Їх можна розділити їх на три групи:

- машинні мови;
- мови низького рівня;
- мови високого рівня.

Машинна мова – це мова, яку машини можуть зрозуміти безпосередньо, оскільки вона використовує лише нулі та одиниці (двійковий). Мови низького рівня дуже близькі до способу спілкування машин, але не досягають двійкового. Недоліком цих мов є те, що вони є специфічними для кожної машини. У цій групі найбільш характерною мовою є мова асемблера. Мови високого рівня ближче до способу спілкування людей. Вони портативні і значно полегшують процес написання, читання та модифікації програм. Python і C++ є двома прикладами мов високого рівня.

Інший спосіб їх класифікації – відповідно до парадигм програмування, які визначають різні способи структурування та впорядкування дій, які програма

повинна виконувати. У цьому випадку їх можна розділити на імперативні та декларативні. Імперативні мови вказують послідовність операцій, які програма повинна виконати для вирішення проблеми. PHP, Java або Python є прикладами імперативних мов. Декларативні мови визначають бажаний результат, а мова відповідає за отримання того, що потрібно для його досягнення. Prolog, Lisp і Haskell є декларативними мовами.

Іншим критеріїм класифікації є спосіб отримання кінцевого результату. За цим критерієм існує два типи мов: компільовані та трансляційні. Мови перекладу діють за принципом перекладача, який виконує синхронний переклад мовця, який говорить іншою мовою. Програма перекладається під час виконання, тобто одночасно з її використанням – інтерпретатор читає рядок, перекладає його та виконує. JavaScript і PHP є прикладами інтерпретованих мов. На від міну від них компільовані мови функціонують як перекладач, який бере повний твір і перекладає його повністю. Замість того, щоб робити це рядок за рядком, програма повністю перекладається на мову, близьку до мови машини перед її запуском, таким чином генеруючи об’єктний файл. Об'єктна програма використовується під час виконання. Java і C++ є двома прикладами скомпільованих мов.

Окремо можна розділити мови за сферами призначення, а саме мови загального призначення та доменно орієнтовані. Мови загального призначення – це мови, розроблені для вирішення багатьох проблем. Прикладом таких мов є C++, Java і Python та багато інших. Доменно-орієнтовані мови були створені для певної речі, і немає сенсу використовувати їх поза межами їхньої сфери. Типовим прикладом мови такого типу є мова запитів SQL.

Даний набір критеріїв є не повним, проте дозволє отримати загальні уявлення про кількість та різноманітність сучасних мов програмування, особливості їх функціонування та сфери застосування. Очевидним є той факт, що велика кількість мов які містять свої особливості написання програмного коду породжує неоднозначність оцінки. Проведений аналіз найбільш поширених мов програмування дозволив здійснити їх класифікацію (рисунок 1.1).

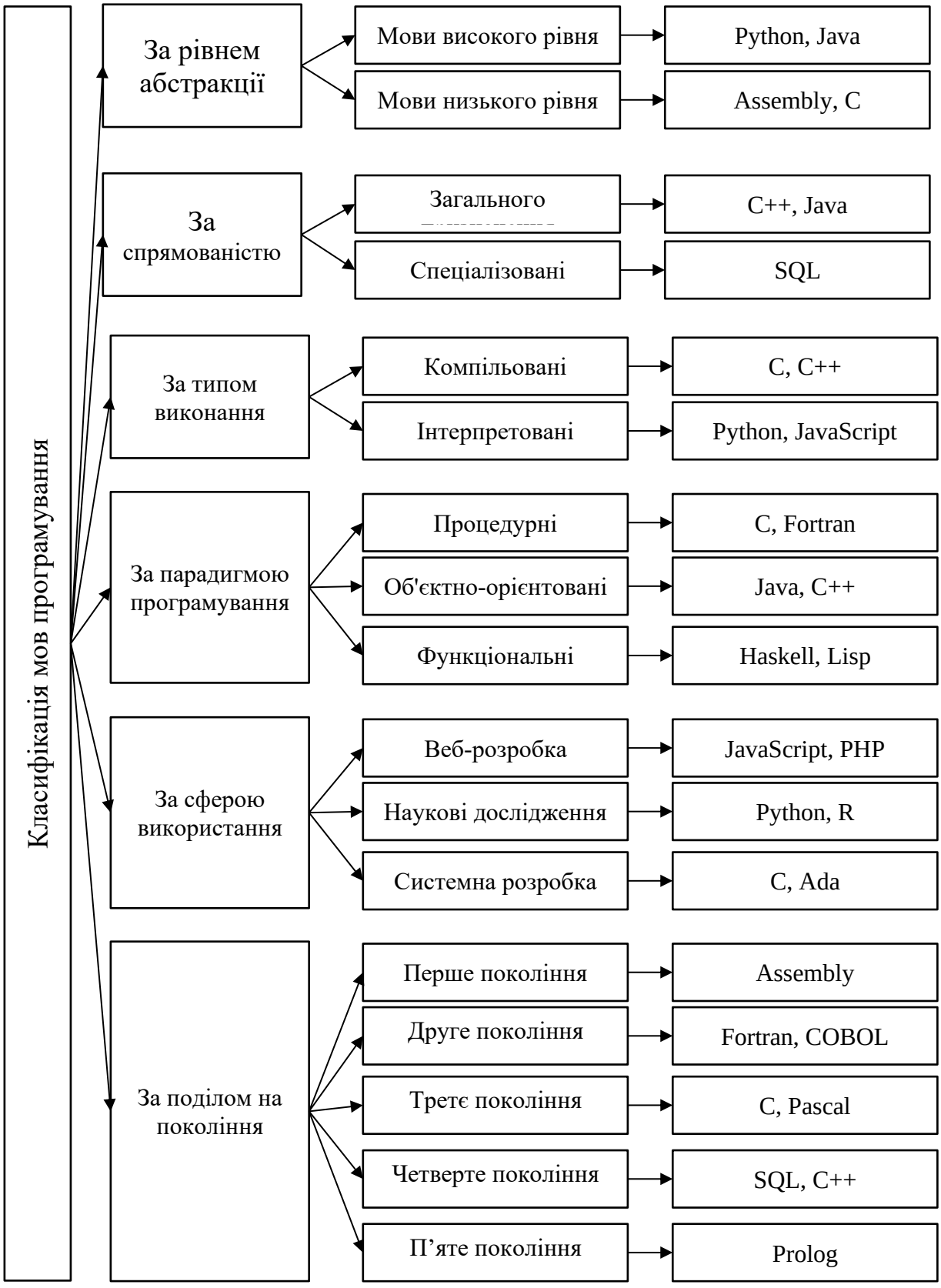

Рисунок 1.1 – Класифікація існуючих мов програмування

Машинна мова складається з числових кодів для операцій, які конкретний комп’ютер може виконувати безпосередньо. Коди являють собою рядки нулів і одиниць або двійкові цифри («біти»), які часто перетворюються як із шістнадцяткового числа, так і в нього для перегляду та модифікації людиною. Інструкції машинної мови зазвичай використовують деякі біти для представлення операцій, таких як додавання, а деякі для представлення операндів або, можливо, розташування наступної інструкції. Машинну мову складно читати й писати, оскільки вона не схожа на звичайну математичну нотацію чи людську мову, а її коди відрізняються від комп’ютера до комп’ютера.

Мова асемблера на один рівень вище машинної мови. Вона використовує короткі мнемонічні коди для інструкцій і дозволяє програмісту вводити імена для блоків пам’яті, які містять дані. Таким чином можна написати «додати плату, загальна» замість «0110101100101000» для інструкції, яка додає два числа. Мова асемблера призначена для легкого перекладу на машинну мову. Хоча на блоки даних можна посилатися за іменами, а не за їхніми машинними адресами, мова асемблера не надає більш витончених засобів організації складної інформації. Як і машинна мова, мова асемблера вимагає детального знання внутрішньої архітектури комп’ютера . Це корисно, коли такі деталі важливі, як-от програмування комп’ютера для взаємодії з периферійними пристроями (принтерами, сканерами, пристроями зберігання даних тощо).

Алгоритмічні мови призначені для вираження математичних або символьних обчислень. Вони можуть виражати алгебраїчні операції в нотації, подібній до математичної, і дозволяють використовувати підпрограми, які упаковують загальновживані операції для повторного використання. Вони були першими мовами високого рівня. Першою важливою алгоритмічною мовою був Fortran. Він був призначений для наукових обчислень з дійсними числами та їх колекціями, організованими як одно- або багатовимірні масиви. Його керуючі структури включали умовні оператори if, повторювані цикли (так звані цикли do) і оператор goto, який дозволяв непослідовне виконання програмного коду.

Fortran зробив зручним наявність підпрограм для типових математичних операцій і створив їх бібліотеки.

Fortran також був розроблений для перекладу на ефективну машинну мову. Він одразу став успішним і продовжує розвиватися.

Мова програмування С була розроблена в корпорації AT&T для програмування комп'ютерних операційних систем. Її особливість це здатність структурувати дані та програми за допомогою композиції менших одиниць. Вона використовує компактну нотацію та надає програмісту можливість оперувати як адресами даних, так і їх значеннями. Ця здатність важлива в системному програмуванні , і С має спільну з мовою асемблера можливість використовувати всі особливості внутрішньої архітектури комп'ютера. С разом зі своїм нащадком С++ залишається однією з найпоширеніших мов.

SQL (структурована мова запитів) – це мова для визначення організації баз даних (колекцій записів). Бази даних, організовані за допомогою SQL, називаються реляційними, оскільки SQL надає можливість запитувати в базі даних інформацію, яка потрапляє в задане відношення.

BASIC (універсальний символічний інструкційний код для початківців) був призначений для легкого освоєння новачками , особливо для спеціалістів, які не займаються комп'ютерними науками, і для успішної роботи на комп'ютері з розподілом часу для багатьох користувачів. Він мав прості структури даних і позначення, а також його інтерпретували: програма BASIC перекладалася рядок за рядком і виконувалася в міру перекладу, що полегшувало пошук програмних помилок.

Паскаль призначався для навчання структурованому програмуванню, яке наголошувало на впорядкованому використанні умовних і циклічних керуючих структур без операторів goto.

Hypertalk був розроблений для Macintosh від Apple . Використовуючи простий англійський синтаксис, Hypertalk дозволяв будь-кому швидко об'єднувати текст, графіку та аудіо в «пов'язані стеки», якими можна було переміщатися, натискаючи мишею на стандартні кнопки, які надає програма.

Незважаючи на те, що Hypertalk мав багато можливостей об’єктно-орієнтованих мов, Apple не розробила його для інших комп’ютерних платформ і залишила його млявим

Об'єктно-орієнтовані мови допомагають керувати складністю великих програм. Об’єкти упаковують дані та операції над ними так, що лише операції є загальнодоступними, а внутрішні деталі структур даних приховані. Це приховування інформації полегшувало масштабне програмування, дозволяючи програмісту думати про кожну частину програми окремо. Крім того, об’єкти можуть бути похідними від більш загальних, «успадковуючи» їхні можливості. Така ієрархія об’єктів дозволяла визначати спеціалізовані об’єкти, не повторюючи всього того, що є в більш загальних.

Мова C++ розширила мову C , додавши до неї об’єкти, зберігши при цьому ефективність програм на C. Це була одна з найважливіших мов як для освіти, так і для промислового програмування. Значні частини багатьох операційних систем були написані мовою C++. C++ разом із Java став популярним для розробки комерційних пакетів програмного забезпечення, які включають численні взаємопов’язані програми. C++ вважається однією з найшвидших мов і дуже близька до мов низького рівня, що дозволяє повністю контролювати розподіл пам’яті та керування нею. Саме ця функція та багато інших можливостей також роблять її однією з найскладніших мов для вивчення та використання у великому масштабі.

На початку 1990-х років Java була розроблена Sun Microsystems, Inc. як мова програмування для всесвітньої павутини (WWW). Хоча зовнішнім виглядом вона нагадував C++, вона була повністю об’єктно-орієнтованою. Зокрема, Java відмовилася від функцій нижчого рівня, включаючи можливість маніпулювати адресами даних, можливість, яка не є ні бажаною, ні корисною в програмах для розподілених систем. Щоб бути портативними, програми Java транслюються віртуальною машиною Java, специфічною для кожної комп’ютерної платформи, яка потім виконує програму Java. Окрім додавання інтерактивних можливостей до Інтернету через веб-«аплети», Java широко

використовується для програмування малих і портативних пристроїв, таких як мобільні телефони .

Мова з відкритим кодом Python. Вона була розроблена як проста у використанні мова з такими функціями, як використання відступів замість дужок для групування операторів. Python також є дуже компактною мовою, розробленою таким чином, що складні завдання можуть виконуватися лише кількома операторами. У 2010-х роках Python став однією з найпопулярніших мов програмування разом із Java та JavaScript .

Декларативні мови, також звані непроцедурними або дуже високого рівня, є мовами програмування, в яких (в ідеалі) програма визначає, що має бути зроблено, а не те, як це зробити. У таких мовах існує менше відмінностей між специфікацією програми та її реалізацією, ніж у процедурних мовах, описаних досі. Двома поширеними видами декларативних мов є логічні та функціональні мови.

Логічні мови програмування , з яких Пролог є найвідомішим, формулює програму як набір логічних зв’язків (наприклад, дідусь і бабуся є батьком батьків когось). Такі мови подібні до мови бази даних SQL . Програма виконується «системою висновків», яка відповідає на запит шляхом систематичного пошуку цих зв’язків, щоб зробити висновки , які дадуть відповідь на запит. PROLOG широко використовується в обробці природної мови та інших програмах ШІ .

Мови сценаріїв іноді називають маленькими мовами. Вони призначені для вирішення відносно невеликих проблем програмування, які не потребують накладних витрат на оголошення даних та інших функцій, необхідних для того, щоб зробити великі програми керованими. Мови сценаріїв використовуються для написання утиліт операційної системи, для програм спеціального призначення для обробки файлів і, оскільки їх легко вивчити, іноді для значно більших програм.

Perl був розроблений для використання з операційною системою UNIX . Передбачалося, що він матиме всі можливості попередніх мов сценаріїв. Perl

запропонував багато способів вказівки типових операцій і, таким чином, дозволив програмісту прийняти будь-який зручний стиль.

Мови форматування документів визначають організацію друкованого тексту та графіки. Вони поділяються на кілька класів: нотація форматування тексту, яка може виконувати ті самі функції, що й програма обробки тексту , мови опису сторінок, які інтерпретуються пристроєм друку, і, загалом, мови розмітки, які описують призначену функцію частин документа.

TeX був розроблений, для покращення якості математичних записів у книгах. Системи форматування тексту, на відміну від текстових процесорів WYSIWYG («Що бачиш, те й отримуєш»), вбудовують у документ команди форматування простого тексту, які потім інтерпретуються мовним процесором для створення форматованого документа для відображення або друку. TeX позначає текст курсивом, наприклад, як {\it this is italiced}, який потім відображається як *this is italiced* .

PostScript – це мова опису сторінок. Такі мови описують документи термінами, які можуть бути інтерпретовані персональним комп'ютером для відображення документа на його екрані або мікропроцесором у принтері чи іншому пристрої.

Команди PostScript можуть, наприклад, точно позиціонувати текст у різних шрифтах і розмірах, малювати зображення, які описуються математично, і вказувати колір або затінення. PostScript використовує постфікс, також званий зворотною польською нотацією, у якому ім'я операції слідує за її аргументами. Таким чином, «300 600 20 270 дугового ходу» означає: намалювати («штрихувати») дугу 270 градусів із радіусом 20 у місці (300, 600). Хоча PostScript може читати та писати програміст, зазвичай він створюється програмами форматування тексту, текстовими процесорами або інструментами графічного відображення.

HTML (мова розмітки гіпертексту) – це мова розмітки для кодування веб - сторінок . Теги розмітки HTML визначають такі елементи документа, як заголовки, абзаци та таблиці. Вони розмічають документ для відображення

комп’ютерною програмою, відомою як веб-браузер. Браузер інтерпретує теги, відображаючи заголовки, абзаци та таблиці в макеті адаптованому до розміру екрана та доступних шрифтів. HTML не дозволяє визначати нові текстові елементи; тобто він не розширюється.

XML (розширювана мова розмітки) – це спрощена форма SGML, призначена для документів, які публікуються в Інтернеті. Подібно до SGML, XML використовує DTD для визначення типів документів і значень використовуваних у них тегів. У XML використовуються угоди , які полегшують аналіз, наприклад те, що сутності документа позначаються початковим і кінцевим тегами, наприклад <begin>…</begin>. XML надає більше видів гіпертекстових посилань, ніж HTML, наприклад двонаправлені посилання та посилання, що стосуються підрозділу документа.

Веб-сторінки, розмічені за допомогою HTML або XML, є переважно статичними документами. Веб-сценарії можуть додавати інформацію до сторінки, коли читач використовує її, або дозволити читачеві вводити інформацію, яка може, наприклад, бути передана відділу замовлень онлайн-бізнесу.CGI (загальний інтерфейс шлюзу) забезпечує один механізм; він передає запити та відповіді між веб-браузером читача та веб-сервером, який надає сторінку. Простий скрипт може запитати ім'я зчитувача, визначити Інтернет-адресу системи, яку використовує зчитувач, і надрукувати привітання. Сценарії можуть бути написані на будь-якій мові програмування.

Проведений аналіз продемонстрував різноманітність існуючих мов програмування та неоднозначність в принципах подудови програмних кодів на кожній з них. Тому розробка алгоритму загальної оцінки якості напианні програмного коду потребує визначення максимально ефективних метрик для проведення повного аналізу. Оскільки різні мови, в своїй основі, мають різні особливості форматування програмного коду, то очевидним є необхідність використання структурного підходу для аналізу написаний текстів. Це допоможе узагальнити та стандартизувати вимоги до різних мов програмування та частково нівелювати різниці в їхньому синтаксисі.

1.2 Структурний аналіз при розробці програмного коду

Структурний аналіз в програмуванні є методологією, спрямованою на аналіз та розкладання програмного коду з метою визначення структури та організації програми. Цей підхід став важливим елементом розробки програмного забезпечення, оскільки дозволяє розглядати складні системи як сукупність взаємозалежних елементів. Структурний аналіз використовується для кількох ключових цілей у сфері програмування. По-перше, він допомагає в розкладанні програм на менші, легше керовані компоненти, що сприяє покращенню читабельності та розуміння коду. Відокремлення коду на модулі дозволяє здійснювати ізольовані зміни та спрощує процес розробки. Окрім того, він забезпечує полегшення обслуговування та розширення кодової бази. Структурний аналіз розробляє структуру, яка підтримує майбутні зміни, роблячи програму гнучкою та розширюваною. Це особливо важливо для великих проектів, де планування майбутнього розвитку є ключовим етапом розробки.

Структурний аналіз також допомагає в управлінні складністю програмного коду. Поділ програми на логічні блоки полегшує відстеження та розуміння взаємодій між різними частинами системи. Це важливо для того, щоб забезпечити ефективне управління кодовою базою, зокрема при розробці великих та довгострокових проектів.

Ще однією важливою функцією структурного аналізу є полегшення тестування та відлагодження програмного коду. Модульна структура дозволяє проводити тестування окремих частин програми, що робить виявлення та виправлення помилок більш ефективними.

Один з класичних прикладів використання структурного підходу в процесі написання програмного куду є розробка систем управління базами даних. Системи такого роду використовують структурний аналіз для визначення логічних компонентів, таких як табличні моделі, запити та процедури збереження. Це робить систему більш гнучкою та легше розширюваною, щоб

враховувати різноманітні потреби користувачів. Інший приклад - розробка веб-застосунків. Тут структурний аналіз використовується для визначення архітектури, обробки подій та взаємодії між користувачем та сервером. Відокремлення фронтенду та бекенду, наприклад, є результатом структурного аналізу, що полегшує розвиток та підтримку веб-додатків.

Крім того, структурний аналіз є необхідним елементом роботи з вбудованими системами, такими як автомобільні електронні системи керування. В таких системах аналізується та розкладається код, відповідальний за різні аспекти, такі як система гальм, двигун та електронні компоненти.

Загалом, структурний аналіз відіграє критичну роль у полегшенні розробки, управлінні та підтримці програмних проектів. Він дозволяє розглядати систему як сукупність взаємозалежних частин, що полегшує архітектурне планування та забезпечує високу якість програмного забезпечення.

Якщо проводити дослідження програмного коду з погляду структурного аналізу то можна визначитиосновні структурні елементи цих мов (рисунок 2.1). Даний перелік є узагальненим і відповідає більшості мов програмування. Базуючись на обраних структурах і буде проводитись розробка про

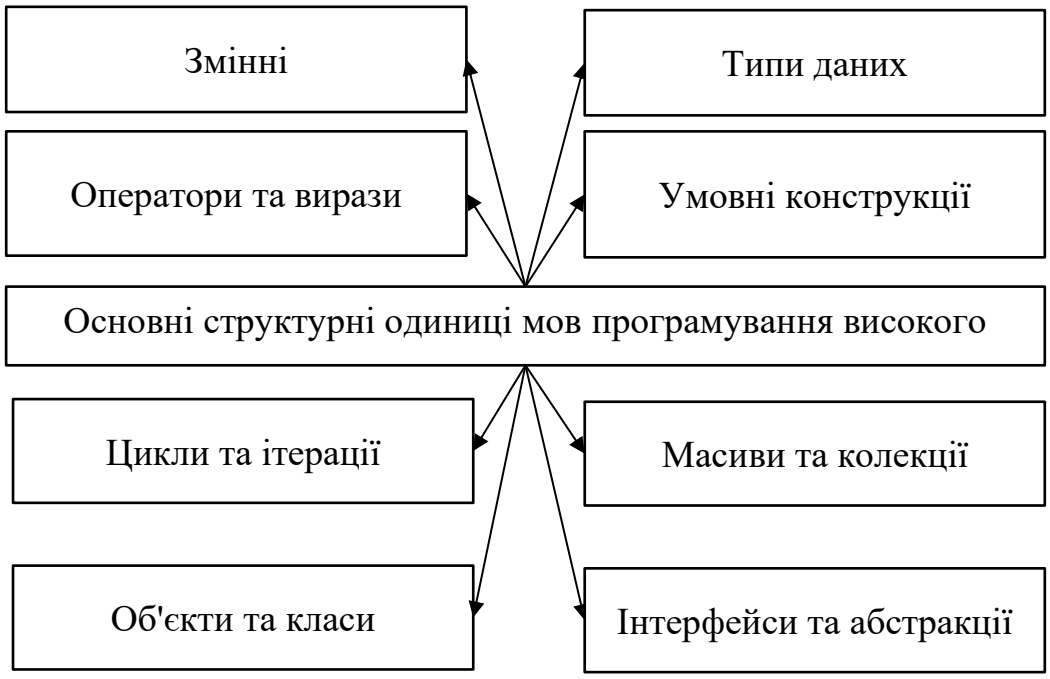

Рисунок 1.2 – Основні структурні одиниці мов програмування високого рівня

Для детального розуміння окремих структурних елементів дамо вихначення для кожного з них. Змінні - це символічні імена, які вказують на місце в пам'яті для зберігання даних. Змінні дозволяють звертатися до значень та об'єктів у програмі, надаючи їм імена для зручності користувача. Типи даних визначають характеристики даних, такі як числа, рядки, булеві значення тощо. Вони важливі для того, щоб комп'ютер розумів, як правильно обробляти дані. Під оператором будемо розуміти символи чи слова, що виконують конкретні дії, такі як додавання, віднімання, порівняння тощо. А вирази – це суміш операторів та операндів і представляють обчислення чи обробку даних. Умовні конструкції дозволяють виконувати різні дії в залежності від умов. Зазвичай виражаються через конструкції типу "if-else". Циклічні струтурид озволяють виконувати блок коду декілька разів. Це основний спосіб автоматизації повторюваних завдань.Масиви - це структури для зберігання кількох елементів одного типу за індексами, а колекції – це розширені масиви, які можуть містити елементи різних типів та мають багато додаткових функцій. Функції мають власний блок коду, який можна викликати з інших місць програми. Вони часто повертають значення. Процедури – це різновид функцій, але не повертають значення. Вони використовуються для виконання певних завдань без повернення результату. Екземпляри класів (об’єкти) це складні типи даних, що мають властивості та методи. Класи визначають структуру та поведінку об'єктів. Концепція ООП дозволяє групувати дані та функціональність в один об'єкт. Інтерфейси визначають контракти для взаємодії об'єктів чи модулів. А абстракції використовують для приховання деталей реалізації та надання взаємодії на високому рівні.

Ці структурні елементи представляють собою основу для розробки програмного забезпечення на високорівневих мовах. Розуміння цих компонентів дозволяє програмістам конструювати складні та ефективні програми, використовуючи потужність високорівневих мов програмування.

Програмування – це складна інтелектуальна діяльність і аналіз сучасних досліджень показав, що більшість програмістів здатні писати програми; однак

їхні програми часто погано побудовані, оскільки вони не розглядають різні рішення програми. Програмісти-початківці часто намагаються вирішити проблему якомога швидше, не замислюючись про якість своїх програм. Автоматизований аналіз програм може покращити процес оцінювання, який виконують при оцінці результатів робти програмістів.

Статичний аналіз – це процес перевірки вихідного коду без виконання програми. Він використовується для виявлення проблем у коді, включно з потенційними помилками, непотрібною складністю та високими вимогами до обслуговування. Динамічний аналіз – це процес запуску програми через набір даних. Основна мета динамічного аналізу – виявити помилки виконання та допомогти оцінити правильність програми. Програми для навчання та автоматичного виставлення оцінок використовують або статичний аналіз, або динамічний аналіз, або обидва для оцінювання програм студентів. Існує багато методів реалізації статичного аналізу; однак підходи, які були прийняті в освітніх програмах з інформатики, відрізняються від зіставлення рядків на основі джерела програми (найпростіша форма) до зіставлення представлень програмного графа (складна форма). При проведенні простого пострічкового аналізу можна виділити ряд основних проблем (Таблиця 1.1).

Таблиця 1.1 – Приклади типових помилок програмного коду

Тип помилки	Опис помилки
Забагато циклів і умовних операторів	Надлишковість у використання циклічних операцій призводить до ускладнення процесу читання коду
Недостатньо методів	Мінімальна кількість методів перешкоджає масштабуванню програмної системи
Глобальні змінні замість параметрів методу	Доступ до глбальних змінних можна отримати з довільної точки програмного коду, що може призвести до незапланованих їх змін та некоректної роботи програми
Занадто великі методи	Узагальнені методи не дають змоги створювати більш гнучкий код
Невикористані змінні	Невиправдані втрати пам’яті
Неініціалізовані змінні	Може призвести до невизначених ситуацій
Невідповідні модифікатори доступу	Вільний доступ сприяє до неконтрольованих змін значень, а занадто закритий стиль вимагає реалізації додаткових методів

Крім простих помилок при реалізації програмного коду можна виділити групу логічних помилок. Даний набір помилок може взагалі не вплинути на роботу програмного коду, або спрацювати в окремих випадках. Перелік таких помилок програмного коду наведено в таблиці 1.2.

Таблиця 1.2 – Приклади типових логічних помилок програмного коду

Тип помилк	Опис помилки
Пропущений оператор «break» у блоці case	Відсутність переривання виконання програмного коду після виконання блоку може мати як негативний результат так і позитивний у випадку реалізації множинного вибору
Пропущений регістр «» у операторі switch	Використання оператора «default» не є обов'язковим, проте в деяких випадках доцільно описувати помилкові ситуації
Плутанина між глобальними та локальними змінними	Використання однакових назв як для глобальних так і локальних змінних може призвести до некоректного опрацювання програмного коду
Пропущений виклик конструктора суперкласу	Пропуск визову конструктора суперкласа (конструктора базового класу) в підкласі може призвести до ряду проблем у програмі, особливо коли суперклас має власний конструктор.

Для перевірки правильності та коректності написання програмного коду рекомендується проводити візуальний аналіз який повинен включати в себе первинну перевірку на шаблонні помилки. Приклад шаблонних перевірок для оцінки програмного коду наведено в таблиці 1.3

Таблиця 1.3 – Критерії перевірки якісного коду

Тип перевірки	Операції для перевірки
Статистика програми	Підрахунок загальної кількості змінних, операторів і виразів у проміжку
Тіньові змінні	Перевірка, чи змінна оголошена як в області класу, так і в області методу
Цикломатична складність	Підрахунок кількість логічних та циклічних рішень у програмі
Невикористані параметри	Аналіз, чи є в методі невикористані параметри
Надлишковий логічний вираз	Виявлення надлишкових логічних напр. вирази "x==true"
Модифікатори доступу	Перевірка чи змінні та методи мають правильні модифікатори
Оголошення Switch	Перевірка на наявність що всі оператори switch мають регістр «default» і в кожному у блоці case є оператор "break".
Тип форматування	Перевірка, що є для форматування пробіл чи клавіша Tab

В загальному, структурний аналіз є методом вивчення та розкладання програмного коду на більш прості елементи з метою зрозуміння його структури та організації. Цей підхід дозволяє виділити ключові компоненти програми, такі як функції, модулі та їхні взаємовідносини. Основними елементами при такому аналізі слід вважати: модулі, зв’язки, класи, методи, структури даних, команди, змінні. Основною метою використання стуктурного аналізу при розробці прогамного коду є підвищення рівня розуміння коду та підтримка модульності. Інструментами виконання стуктурного аналізу можна вважати діаграми структур, стандарти кодування, метрики оцінки коду. Ці поняття структурного аналізу формують основу для розуміння та оцінки якості програмного коду на основі його структурної організації.

1.3 Програмні засоби створення та редагування програмних кодів

Програмне забезпечення IDE (інтегроване середовище розробки) – це тип програми, який використовується для програмування та розробки комп’ютерних програм. Він поєднує текстовий редактор, компілятор, налагоджувач та інші інструменти в одному пакеті, щоб допомогти розробникам створювати програмне забезпечення швидко та ефективно. IDE зазвичай надає графічний інтерфейс користувача, який дозволяє розробникам легко отримувати доступ до всіх компонентів, які їм потрібні під час роботи над проектами. Це може включати меню, панелі інструментів, функції пошуку, редактори вихідного коду та підтримку мови. Деякі IDE мають додаткові функції, такі як інтеграція з системами контролю версій, такими як Git або SVN.

Більшість IDE розроблено для підтримки певних мов програмування або фреймворків. Підтримувані мови можуть відрізнятися від Java, JavaScript, Python і C++ до більш складних, таких як Ruby on Rails і ASP .NET Core. Кожна мова має власний набір функцій, як-от підсвічування синтаксису та автозаповнення,

що полегшує програмування для розробників. Функція налагодження допомагає розробникам швидко виявляти помилки у своєму коді, надаючи їм докладний відгук про проблеми, які виникли під час виконання програми. Налагодження також включає виявлення помилок під час виконання, перш ніж програма аварійно завершить роботу або несподівано зависне під час виконання. Крім того, деякі IDE пропонують варіанти рефакторингу коду, які дозволяють пришвидшити цикли розробки за рахунок скорочення часу, витраченого на ручну реструктуризацію логіки коду в складних програмах.

IDE стають дедалі популярнішими серед програмістів, оскільки вони полегшують розробку програмного забезпечення швидше, забезпечуючи при цьому якість. Вони забезпечують інтуїтивно зрозумілий інтерфейс, який спрощує навіть складні завдання, такі як налагодження та написання чистого коду, що економить дорогоцінний час у довгостроковій перспективі. Крім того, багато сучасних IDE тепер мають потужні плагіни, які додають нові функціональні можливості, такі як візуалізації та можливості розгортання в хмарі, що робить їх надзвичайно універсальними програмами для арсеналу будь-якого програміста!

Програмне забезпечення IDE (інтегроване середовище розробки) надає повний набір інструментів, які допомагають розробникам створювати проекти програмного забезпечення та керувати ними:

Текстовий редактор: програмне забезпечення IDE зазвичай містить текстовий редактор, який дозволяє розробникам писати та редагувати код, а також підтримує декілька мов програмування.

Інструменти налагодження: інструменти налагодження забезпечують видимість того, як працює код, дозволяючи розробникам виявляти та виправляти помилки у своїх програмах. Їх також можна використовувати для тестування.

Підсвічування синтаксису: підсвічування синтаксису полегшує читання вихідного коду, призначаючи кольори різним елементам синтаксису, таким як ключові слова, рядки, класи, функції та коментарі.

Автозавершення: функція автозавершення пропонує повні слова або фрази під час введення коду, що робить написання складних фрагментів коду швидшим і легшим.

Інтеграція системи контролю версій: IDE часто інтегруються з такими популярними системами контролю версій, як Git і SVN, тож ви можете вносити зміни безпосередньо з самого IDE. Це полегшує співпрацю з іншими розробниками над одним проектом.

Компілятори/інтерпретатори: компілятори та інтерпретатори є важливими компонентами IDE, що дозволяє розробникам швидко компілювати або інтерпретувати свої програми для цілей тестування без необхідності перемикатися між програмами вручну.

Підтримка між платформами: багато IDE пропонують підтримку між платформами, щоб розробники могли працювати над одним проектом незалежно від того, яку платформу вони використовують – наприклад, Windows, macOS або Linux – що ще більше покращує співпрацю.

Плагіни та додаткові модулі: IDE надають розробникам платформу для розширення функціональних можливостей своєї IDE за допомогою спеціальних плагінів і додаткових компонентів. Це дозволяє розробнику налаштувати IDE відповідно до власних потреб.

C++Builder – це швидкий комплексний пакет для проектування та розробки сучасних програм. Створіть свій основний макет інтерфейсу користувача один раз, а потім легко налаштуйте перегляди для платформи та пристрою без дублювання зусиль щодо розробки. Візуально підключайте елементи інтерфейсу користувача до джерел даних за допомогою LiveBindings Designer. Перевірка дизайну в реальному часі за допомогою Live On-Device Preview для трансляції активної форми на кілька пристроїв одночасно. Додайте адаптивний дизайн із компонентами, що відповідають роздільній здатності, для настільних ПК, планшетів і смартфонів. Справжні власні елементи керування для конкретної платформи для кращої взаємодії з користувачем.

Dev-C++ – це новий і вдосконалений форк (спонсорований Embarcadero) Bloodshed Dev-C++ і Orwell Dev-C++. Це повнофункціональне IDE і редактор коду для мови програмування C/C++. Він використовує Mingw порт GCC як компілятор. Embarcadero Dev-C++ також можна використовувати в поєднанні з Cygwin або будь-яким іншим компілятором на основі GCC. Embarcadero Dev-C++ створено з використанням останньої версії Embarcadero Delphi. Embarcadero Dev-C++ займає мало пам'яті, оскільки це рідна програма Windows і не використовує Electron. Оптимізовано для паралельної компіляції на сучасних багатоядерних машинах.

Visual LANSA – це платформа розробки з низьким вмістом коду, яка використовується IT-спеціалістами для створення корпоративних веб-додатків і мобільних додатків швидше, простіше та з меншою ціною, ніж традиційне кодування. Visual LANSA прискорює розробку додатків, усуваючи необхідність опанувати численні технічні навички, які зазвичай потрібні для створення програмних додатків. Після розробки програми її можна розгорнути на сервері IBM i, Windows або Linux. Одна IDE, одна мова, без обмежень. Visual LANSA містить такі функції, як керування доступом/дозволи, допомога в коді, рефакторинг коду, інструменти для співпраці, тестування сумісності, моделювання даних, налагодження, керування розгортанням, графічний інтерфейс користувача, розробка мобільних пристроїв, No-Code, звітування/аналітика, розробка програмного забезпечення, джерело контроль і контроль версій. Visual LANSA пропонує цілодобову живу підтримку та онлайн-підтримку.

IntelliJ IDEA аналізує код, шукаючи зв'язки між символами в усіх файлах проекту та мовами. Використовуючи отриману інформацію, він надає поглиблену допомогу в кодуванні, швидкий перехід та навігацію, розумний аналіз помилок і, звичайно, рефакторинг. Надає список найбільш релевантних символів, застосовних у поточному контексті. названі та інші доповнення постійно навчаються у вас, переміщуючи членів найбільш часто використовуваних класів і пакетів у верхню частину списку пропозицій.

Пропонує список символів, які відповідають вашим введенням, і автоматично додає необхідні оператори імпорту.

Eclipse IDE Провідна відкрита платформа для професійних розробників, яка використовується в комп'ютерному програмуванні. Eclipse IDE забезпечує те, що вам потрібно для швидкого впровадження інновацій. Простіша конфігурація IDE Інсталятор Eclipse IDE 2020-09 і кілька пакетів тепер включають Java Runtime Environment (JRE). Покращена тематика та стиль. Покращено темну тему Windows і світлу тему GTK. Eclipse IDE тепер потребує Java 11 як мінімальну версію для роботи, але можете скомпілювати будь-яку версію, як зазвичай. Нові експериментальні функції. Підтримка aarch64. Підтримка Linux принесла цю версію. Node.js тепер вбудовано. Для всіх наших інструментів на основі LSP тепер вбудовано Node.js, щоб все працювало одразу. Безкоштовно та з відкритим кодом Безкоштовно з відкритим кодом; випущений згідно з умовами Eclipse Public License 2.0. Завдяки участі. Велика екосистема плагінів від активної спільноти

Android Studio – це найбільш ефективний інструмент для розробки програм під операційну систему Android на різних типах пристроїв. Використовує ConstraintLayout для створення складних макетів, де легко встановлювати обмеження між елементами та регулювати їх розміщення. Зручний перегляд макетів на різних екранах або при зміні розмірів вікна попереднього перегляду дозволяє ефективно адаптувати додаток до різних умов відображення. Android Studio також надає засоби для аналізу розміру додатка, дозволяючи перевіряти вміст файлу apk та виявляти можливості для оптимізації. Інтелектуальний редактор коду сприяє швидкій і точній роботі, забезпечуючи автодоповнення для мов Kotlin, Java та C/C++. Це робить Android Studio ідеальним інструментом для створення додатків, які працюють ефективно та адаптуються до різних умов використання.

Середовище Ultimate RAD, улюблене розробниками за швидке створення високопродуктивних власних кросплатформних програм на сучасних C++ і Delphi за допомогою потужних інструментів візуального дизайну та

інтегрованих інструментальних ланцюжків. Розумніша навігація коду під час рефакторингу. Автоматичне доповнення коду за допомогою клавіші Tab. LSP обізнаність про включені файли. Автоматичний перезапуск сервера LSP. Підтримка помічника в класі. Пропозиції масивів під час призначення масивів. Неактивне виділення коду в редакторі коду. Підтримка високої роздільної здатності в IDE з повною підтримкою найновіших моніторів 4k+, а також чистішими та чіткішими шрифтами та значками.

Visual Studio. Повнофункціональна IDE для кодування, налагодження, тестування та розгортання на будь-якій платформі. Кодуйте швидше. Працюйте розумніше. Створіть майбутнє за допомогою найкращої у своєму класі IDE. Розробляйте весь набір інструментів від початкового проектування до остаточного розгортання. Покращена продуктивність IntelliSense для файлів C++. Локальна розробка з багатьма поширеними емуляторами. Спрощений тестовий доступ у Solution Explorer. Керування Git і створення репо в IDE. Підтримка Kubernetes тепер включена в робоче навантаження Microsoft Azure.

В результаті проведеного аналізу сучасних IDE отримані результати були згруповані на наведені в таблиці 1.4.

Таблиця 1.4 – Узагальнена таблиця порівняння сучасних IDE

Назва інтегрованого середовища розробки	Зручність користування	Допомога під час написання коду	Оцінка отриманого програмного коду	Підтримка від розробників	Загальна оцінка
C++Builder	4	5	4	4	4,2
Dev-C++	4	3	3	4	3,5
Visual LANSA	4	4	4	4	4
IntelliJ IDEA	5	5	5	5	5
Eclipse IDE	4	4	4	5	4,2
Ultimate RAD	5	5	4	5	4,8
Android Studio	5	5	5	5	5
Visual Studio	5	5	5	5	5

1.4 Постановка задач дослідження

В даному розділі було проведено дослідження основних відмінностей та характеристик мов програмування. На основі отриманих даних було здійснено класифікації поширених мов програмування на основі виділених критеріїв. Дана класифікація дозволила виділити основні групи мов програмування та технологій, які використовуються для реалізації програмних систем різної складності в різних сферах. Іншим напрямом дослідження було визначення можливостей використання підходів структурного аналізу для написання програмних кодів з подальшим оцінюванням на основі різних метрик. Дуло виділено основні групи помилок, що допускаються при написанні програмних кодів та шляхи їх уникнення. Зокрема, прості помилки повинні перевірятись менеджерами проектів для оцінки якості та ефективності роботи програмного розробника. Більш складні типи логічних помилок повинні визначатись на рівні тестових перевірок на різних етапах розробки програмного забезпечення. Додатково було проведено аналітичний огляд середовищ інтегрованої розробки програмних засобів з метою виділення внутрішніх функціональних можливостей для перевірки та оцінки правильності написання коду.

Для досягнення поставленої мети необхідно розв’язати наступні задачі.

- провести класифікацію та виділити особливості мов програмування;
- проаналізувати використання технологій структурного аналізу під час створення програмного коду;
- провести аналітичний огляд інтегрованих середовищ розробки програмних засобів;
- проаналізувати існуючі метрики оцінки якості програмного коду;
- розробити алгоритм оцінки якості програмного коду на основі структурного аналізу;
- реалізувати програмний додаток аналізу та оцінки програмного коду мовою високого рівня.

1.5 Висновки до розділу

Проведено дослідження мов програмування на основі їх структурних особливостей, функціональних можливостей та сфер застосування, що дозволило провести їх класифікацію та визначити групу програмних мов які найчастіше використовуються при створенні програмних продуктів.

Проведено аналіз використання технологій та інструментів структурного аналізу під час створення програмного коду, що дозволило виділити основні елементи в структурі програмного коду та описати ситуації які сприяють погіршенню оцінки програмної реалізації.

Проведено дослідження інтегрованих середовищ розробки, що дозволило виділити та додатково проаналізувати основні функціональні складові які виконують основні задачі програмних систем даного типу.

2 МЕТОДИ ТА АЛГОРИТМИ АВТОМАТИЗОВАНОЇ ОЦІНКИ ПРОГРАМНОГО КОДУ

2.1 Метрики оцінки програмного коду

Високоякісний код ефективний і надійний, добре працює без помилок і відповідає потребам користувачів. Він може впоратися з помилками або незвичними умовами. Його також легко зрозуміти, підтримувати та розширювати новими функціями. Крім того, його портативність означає, що він може працювати на якомога більшій кількості машин.

Команди розробників працюють із кодовими базами, які постійно змінюються. Вони додають, видаляють і змінюють існуючий код, щоб покращити швидкість або реалізувати нові функції. Це означає, що вони читають багато коду, тому для них якість дорівнює читабельності.

На жаль, постійні зміни коду часто призводять до поступового погіршення його якості. Це завдання дає можливість дослідити, що робить код читабельним, зрозумілим і стабільно вищої якості.

Чим простіше читати код, тим легше його розуміти та редагувати, і він працює швидше та з меншою кількістю помилок, що забезпечує його покращену підтримку та розширюваність. Усе це веде до швидшої розробки та меншої кількості помилок, забезпечуючи довгострокове зниження витрат.

Надання описових імен змінних є основною практикою, але це значно покращує читабельність. У той час як перше легко застосувати на будь-якому рівні кваліфікації, друге легко не помітити – незалежно від стажу. Ефективне використання ресурсів вимагає знання використовуваних інструментів і уважності для ефективного управління ними.

Щоб найкраще виміряти якість коду, використовувані показники поділяються на одну з двох груп (рисунок 2.1):

- кількісні;
- якісні.

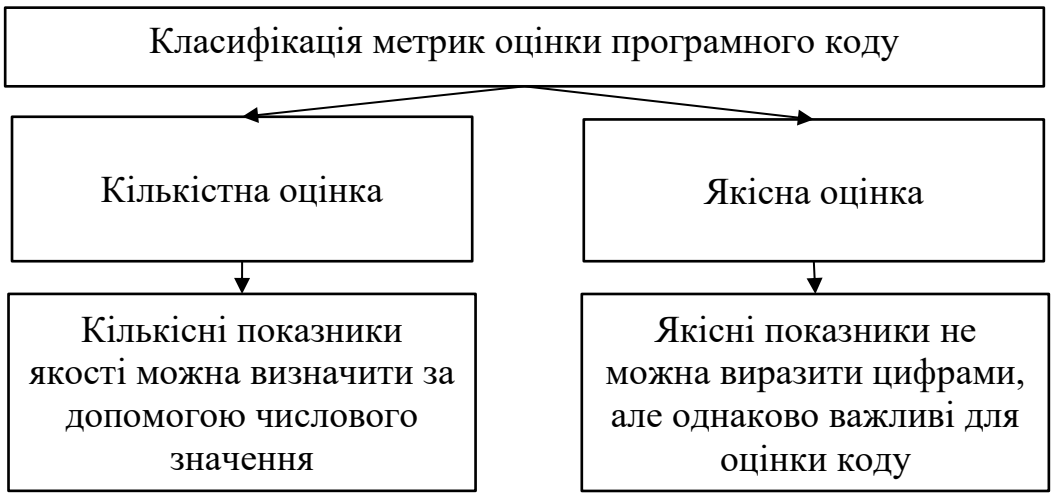

Рисунок 2.1 – Класифікація методів оцінки якості програмного коду

Кількісні показники якості можна визначити за допомогою числового значення. Наприклад, покриття коду — відсоток коду, що виконується під час модульних тестів — є дискретним значенням від 0% до 100%. Чим більше значення, тим менша ймовірність невиявлених помилок. Тому якість коду, ймовірно, вища.

Одним із ключових показників здорової кодової бази є хороше покриття тестів. Це не тільки забезпечить збільшення тестів з тією ж швидкістю, що й код, але й допоможе контролювати робочий процес розробки за допомогою перевірок «пройшов/не пройшов» і PR-коментарів, які показують, де не вистачає охоплення та як його покращити.

Розглянемо середню кількість рядків коду (LOC) на функцію або клас. Як правило, менша кількість рядків забезпечує кращу читабельність. Більшість досвідчених програмістів, безсумнівно, стикалися з більш ніж 1000 функціями LOC, і навряд чи їх було легко прочитати чи зрозуміти.

Існує також більш складна метрика щодо ремонтопридатності, яка називається цикломатичною складністю. Хоча ми рідко зустрічаємося з ним, Visual Studio (наприклад) надає підтримку за замовчуванням для його вимірювання.

Цикломатична складність обчислює кількість шляхів, які містить вихідний код. Інструкції if-then, цикли, перемикачі та інші керуючі оператори збільшують цикломатичну складність коду. Не дивно, що вища складність означає підвищені труднощі з обслуговуванням і більшу ймовірність появи помилок.

Незважаючи на те, що кількісні показники мають вирішальне значення, вони дають лише частину картини. Навіть 100% тестове покриття не означає, що код буде без проблем. І навпаки, висока цикломатична складність не обов’язково створює серйозні проблеми з ремонтопридатністю.

Інтерпретація кількісних показників коду є суб’єктивною практикою. Один член команди може вважати, що 50% покриття коду є достатнім, а інший член може вважати, що 80% є мінімально прийнятним. На щастя, кількісні показники знаходять баланс зі своїми якісними відповідниками.

Якісні показники не можна виразити цифрами, але однаково важливі для оцінки коду. Важливо дотримання стандартів кодування, використання значущих назв для об’єктів або встановлення максимальної ширини рядка в кодовій базі (шириною рядка є число, але його вплив на якість коду ні). Разом ці практики можуть сформувати основу чудового коду.

Якісні показники можуть бути дуже суб'єктивними. Деякі програмісти віддають перевагу довшим іменам змінних, які виражають їх мету, тоді як іншим зручніше використовувати скорочені імена, наприклад ordабо cust. Правила іменування відрізняються залежно від мови. Це ускладнює визначення якісних показників, особливо для тих, чиї основні мови можуть відрізнятися від мов їхніх однолітків.

Одним із життєво важливих способів покращити якість коду та підтримувати відкрите обговорення суб’єктивних показників якості є регулярні перевірки коду. У цьому процесі колеги можуть оцінювати загальну якість коду один одного.

Наприклад, команда з п’яти розробників може зафіксувати кожен код, який має бути переглянуто та прийнято принаймні двома партнерами, перш ніж його можна буде об’єднати назад у основну гілку. З одного боку, цей процес може

виявити проблеми, які могли залишитися непоміченими. Що ще важливіше, програмісти можуть вчитися один в одного та постійно бути в курсі останніх змін програмного забезпечення.

Існує багато інструментів і рекомендацій, які допомагають командам розробників відстежувати та оцінювати якість коду. Незалежно від того, якими інструментами користуватись, можна легко забути перевірити код перед тим, як відправити його в систему керування джерелами.

Найефективніші показники якості коду – це ті, які допомагають визначити помилки та баги в коді. Загальні типи показників для еволюції якості коду

Якісні показники є більш інтуїтивно зрозумілими і не можуть бути виражені цифрами. Вони не піддаються вимірюванню. Ці показники допомагають класифікувати код як прийнятний або відхилений. Якісні показники допомагають вам оцінити, чи ваші команди розробників дотримуються стандартів кодування, призначають об’єктам значущі назви чи реалізують максимальну ширину лінії в кодовій базі, серед інших передових методів кодування. Однак ці показники дуже суб’єктивні (рисунок 2.2).

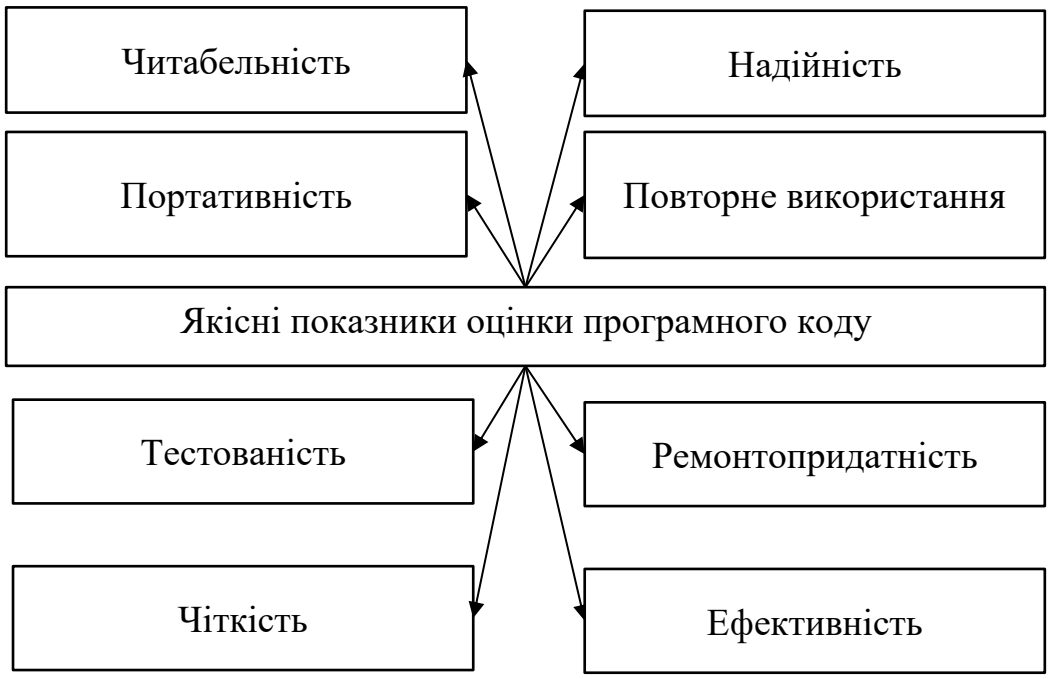

Рисунок 2.2 – Класифікація якісних показників оцінки програмного коду

Наприклад, деякі розробники віддають перевагу довшим іменам змінних, які допомагають зрозуміти призначення об’єкта, тоді як іншим може бути зручно використовувати короткі імена, як-от Ord або Cust. Отже, це ускладнює визначення якісних показників. Однією з найкращих практик для визначення суб’єктивних показників якості та підвищення якості коду є регулярні перевірки коду. Деякі з ключових якісних показників якості коду, які вам потрібно відстежувати, це:

1. Читабельність. Читабельність є найважливішим показником якості коду, оскільки вона веде до вищого рівня розуміння коду серед інших розробників. Ваш код має бути легким для читання та розуміння. Тому що код, який легко читати або розуміти, легко покращити. Правильні відступи, форматування та інтервали роблять код більш читабельним. Це також робить структуру коду більш послідовною та видимою та полегшує процес налагодження. За потреби додайте коментарі до коду зі стислими поясненнями для кожного методу. Також використовуйте узгоджені стилі іменування, як-от camelCase, PascalCase і snake_case. Читабельність коду також можна покращити шляхом зменшення рівня вкладеності.

2. Надійність. Надійність – це здатність коду працювати без збоїв протягом певного періоду часу. Отже, вимірювання надійності коду може допомогти вам визначити успішність вашого програмного забезпечення чи програми. Ви можете визначити надійність свого коду, провівши статичний аналіз коду. Цей тест визначає будь-які дефекти чи помилки у вашому коді. Потім ви можете внести необхідні зміни в код, щоб виправити помилки та покращити якість коду. Низька кількість дефектів є обов’язковою для розробки надійної кодової бази.

3. Портативність. Показник переносимості визначає, наскільки ваш код придатний для використання в різних середовищах. Це показує, наскільки добре інші розробники можуть використовувати ваш код в інших середовищах. Ви можете забезпечити переносимість свого коду, регулярно тестуючи його на різних платформах. Інша найкраща практика - встановити якомога вищий рівень

попередження компілятора. Обов’язково використовуйте два компілятори. Ви також можете покращити переносимість, застосувавши стандарт кодування.

4. Повторне використання. Показник багаторазового використання визначає, чи можна повторно використовувати існуючий код або перепрофілювати його для інших програм чи проектів. Такі характеристики, як модульність або слабий зв’язок, роблять код придатним для повторного використання. Ви можете виміряти багаторазове використання свого коду кількістю взаємозалежностей, які він має. Взаємозалежності – це елементи коду, які функціонують належним чином, коли інші елементи працюють належним чином. Проведення статичного аналізу коду може допомогти вам знайти ці взаємозалежності.

5. Простота тестування. Показник тестованості вимірює, наскільки добре код підтримує різні процеси тестування, що проводяться на ньому. Це залежить від вашої здатності контролювати, ізолювати та автоматизувати тести. Ви можете виміряти тестоздатність свого коду та кількість тестів, необхідних для виявлення потенційних помилок у коді. Розмір і рівень складності коду впливають на кількість тестів, необхідних для виявлення помилок. Тому найкраще тестувати на рівні коду, наприклад цикломатичної складності, щоб покращити тестування. Деякі інші найкращі практики для покращення тестування:

- Спочатку проведіть модульний тест.
- Витягніть увесь нетестований код у класи-оболонки.
- Використовуйте інверсію контролю / впровадження залежностей.

6. Ремонтопридатність. Показник ремонтопридатності коду вимірює, наскільки легко вносити зміни в код, зберігаючи ризики, пов’язані з такими змінами, якомога меншими. Його можна оцінити за кількістю рядків коду в додатку. Якщо цих ліній більше, ніж середня кількість, то ремонтопридатність вважається низькою. Нижче наведено кілька найкращих методів покращення ремонтопридатності.

Код має бути добре розробленим – він має бути максимально простим, легким для розуміння, легким для внесення змін, легким для тестування та простим у роботі:

- Рефакторинг коду.
- Правильно задокументуйте, щоб допомогти розробникам зрозуміти код.
- Автоматизуйте збірку, щоб легко скомпілювати код.
- Використовуйте автоматичне тестування, щоб легко перевіряти зміни.

7. Чіткість. Показник ясності визначає, наскільки зрозумілим є код. Якісний код не повинен бути неоднозначним. Він має бути достатньо зрозумілим, щоб його легко зрозуміли інші розробники, не займаючи багато часу. Нижче наведено кілька найкращих порад щодо покращення чіткості коду:

- Переконайтеся, що ваш код має просту логіку та потік керування.
- Використовуйте порожні рядки, щоб розділити свій код на логічні розділи.

8. Ефективність. Показник ефективності – це міра кількості активів, які використовуються для створення коду. Він також враховує час, витрачений на виконання коду. Створення ефективного коду займає менше часу, і його легко налагодити. Зрештою, ефективний код повинен відповідати визначеним вимогам і специфікаціям.

9. Розширюваність. Показник розширюваності визначає, наскільки добре ваш код може включати майбутні зміни та зростання. Хороша розширюваність означає, що ваші розробники можуть легко додавати нові функції до коду або змінювати існуючі функції, не впливаючи на продуктивність усієї системи. Використання таких концепцій, як слабий зв’язок і поділ проблем, може зробити ваш код більш розширюваним.

10. Документація. Якісний код визначається як код, який можна «використовувати довгостроково, можна перенести на майбутні випуски та продукти, не розглядаючи його як застарілий код». Для цього вам потрібна документація. Добре задокументований код дозволяє іншим розробникам

зрозуміти його та використовувати без особливих витрат часу та зусиль. Документація гарантує, що код читається, а також підтримується будь-яким розробником, який має справу з ним у будь-який момент часу.

Якість коду може покращити його читабельність, зручність обслуговування та розширюваність. Краща якість означає менше технічних боргів і помилок, що спрощує розробку та знижує витрати.

Впровадження якісного коду в процес розробки не повинно бути складним. В ідеалі ви зробили це з самого початку, але ніколи не пізно почати вимірювати та покращувати якість коду.

2.2 Кількісні характеристики якісного програмного коду

Швидкість стала новою валютою в цьому стрімкому цифровому світі. Клієнти очікують, що послуги надаватимуться з високою швидкістю, і компанії, які задовольняють цей попит, мають значну перевагу на ринку. А DevOps дає можливість компаніям швидко постачати програмне забезпечення. Однак ця швидкість не повинна відбуватися за рахунок якості. Код має працювати відповідно до очікувань бізнесу. Випадкові дефекти, тупі помилки та поганий досвід користувача шкодять підприємствам, які прагнуть залишатися конкурентоспроможними на ринку. Забезпечуючи високу якість коду, підприємства можуть підвищити прибутковість, забезпечити кращий досвід для своїх клієнтів і процвітати на ринку.

Якість коду є ключовим аспектом розробки програмного забезпечення. Незалежно від мови, яка використовується для написання коду, якість коду впливає на якість кінцевого продукту, зрештою, на успіх організації.

Якість коду – це показник того, наскільки низьким або високим є значення певного коду набору. Хоча визначення якості коду суб’єктивне, високоякісний код може бути чистим, простим, ефективним і надійним кодом. Чим простіше

читати код, тим легше його зрозуміти та редагувати. Чим ефективніший код, тим швидше він працює з меншою кількістю помилок. Деякі інші властивості коду, які сприяють високій якості коду, це чіткість коду, складність, безпека, безпека, ремонтопридатність, тестування, переносимість, багаторазове використання та надійність. Ці властивості коду визначають, як окрема одиниця коду може вплинути на загальну якість вашої кодової бази.

Однак покращення потребує вимірювання. KPI – це вимірювання, які дозволяють лідерам DevOps бачити, де знаходяться їхні команди, і визначати, куди вони рухаються.

KPI – це показники, які допомагають усім в організації, від інженера до технічного директора, інформаційного директора чи CISO. Коли люди на всіх рівнях працюють на одній сторінці, використовуючи одні й ті самі дані, прогрес відбувається легше та швидше. Команда інженерів використовує інформацію для вдосконалення, тоді як виконавча команда використовує її для прийняття рішень, які змінюють DevOps в організації.

KPI також узгоджує кожного з кількома речами, які є найбільш важливими. Вони пояснюють і допомагають узгодити цілі компанії, покращують співпрацю та прозорість. Наведемо перелік показників ефективності (KPI) і показників DevOps, які повинен відстежувати кожен лідер проекту під час написання та розробки програмного коду (рисунок 2.3).

Швидкість. Частота розгортання, час виходу на ринок і зміни обсягу тощо відіграють роль у швидкості DevOps. Є явна перевага для організацій, які можуть швидко рухатися, і кілька KPI можуть допомогти ефективно виміряти швидкість. Висока частота розгортання та обсяг змін контролюють одне одного: зрештою, швидкий рух нічого не означає, якщо не відбувається реальних змін. Ці KPI забезпечують цикл зворотнього зв’язку, який інформує процес розгортання виробництва для майбутніх функцій.

Аналіз розгортань за збіркою, розробником та іншими факторами також дає корисну наскрізну інформацію про процес. Це заходи ефективність розробникаі продуктивність, показуючи, скільки фіксацій коду, запитів на

злиття, збірок було зроблено для кожного середовища до коду було переведено на виробництві. Ці додаткові ключові показники ефективності дозволяють усім – розробникам, інженерам DevOps, командам із безпеки та якості та керівникам зрозуміти моделі розгортання та виявити прогалини в якості та безпеці.

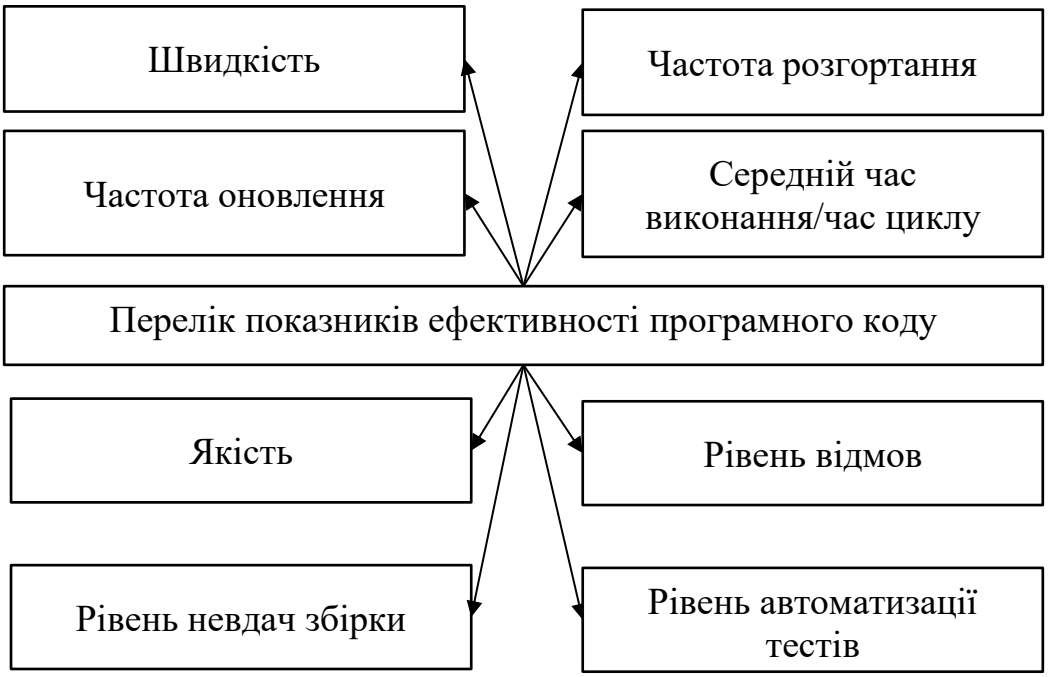

Рисунок 2.3 – Класифікація якісних показників оцінки програмного коду з точки зору його використання

Частота розгортання. Це стосується частоти, з якою нові функції, оновлення або можливості запускаються для загального використання. Як правило, частота вимірюється щодня, хоча деякі організації прагнуть вимірювати щотижневі показники. Однак щоденні розгортання дають більш точний показник ефективності команди. Частота розгортання повинна залишатися достатньо постійною або збільшуватися помірними темпами. Раптове зменшення числа зазвичай означає певну перешкоду в конвеєрі, яка впливає на робочий процес у цілому. З іншого боку, якщочастота розгортання спостерігає різке підвищення, це може вказувати на можливий високий рівень збоїв після того, як оновлення/функція буде фактично у виробництві.

Частота оновлення. Це стосується рядків нового коду, які надсилаються до виробництва під час кожного розгортання. Навіть якщо частота розгортання висока, це нічого не означатиме, якщо команди просто випускатимуть велику кількість несуттєвих, незначних змін. Якщо відбулося 50 розгортань, але лише з кількома змінами або 2-3 рядками нового коду, це мало що означає для ефективності команди. У такому випадку ці розгортання не мають великого значення. Відстежуйте кількість зміненого та статичного коду в кожному розгортанні. Якщо змінений код мінімальний, це означає, що проблема з прогресом існує в конвеєрі. Оновлення можуть бути не просто численними, вони також мають бути впливовими.

Розгортання за проектом, розробником тощо. Додатково уточнюйте показники та відстежуйте частоту розгортання та обсяг змін для кожного розробника та проекту. Без цього огляду менеджери можуть не помітити погано ефективних людей або проекти. Крім того, це також може призвести до того, що вони не помічають проблем або вузьких місць, які можуть перешкодити цим конкретним розробникам/тестерам запропонувати свій найкращий потенціал.

Помітивши нижчі показники розгортання, почніть з оцінки того, чому ця конкретна особа чи окрема особа може мати низьку продуктивність. Вивчайте будь-які прогалини в доступі до ресурсів, оскільки це, як правило, є найпоширенішою причиною слабких цифр. У великих, загалом добре працюючих командах, обмеження, з якими стикаються певні оператори, легко не помітити.

Середній час виконання/час циклу. Час виконання. Вимірювання часу виконання та часу циклу дає змогу зрозуміти ефективність робочого процесу розробки – від ідеї до розгортання. Його слід оцінювати у зв’язку з поточними очікуваннями та вподобаннями бази користувачів. Як правило, довший час виконання та циклу вказують на прогалини в ефективності, тоді як коротші вказують на те, що відгуки користувачів розглядаються швидко.

Однак зауважте, що скорочення часу виконання та циклу не слід прагнути ціною більшої кількості дефектів. Не поспішайте переглядати відгуки

користувачів лише для того, щоб скоротити час, оскільки це призведе до тривалої втрати довіри та прибутку.

Якість. Тестування є ключовим фактором забезпечення якості. Ключові показники ефективності, зосереджені на якості, пропонують систему зворотного зв'язку, яка дозволяє командам рухатися до розробки, керованої тестуванням. Надійний процес забезпечення якості в поєднанні з автоматизацією дає усій команді чітке уявлення про передбачуваність, доступність і продуктивність програми. Вони також надають командам інформацію про окремі точки відмови та час до відновлення в рамках тестування.

Швидке виявлення, реагування та час відновлення у разі збою мають вирішальне значення для підтримки високої доступності. Доступність, звичайно, є основним фактором сприйняття якості програмного забезпечення — зрештою, важко бути високоякісним, коли програма не працює. KPI, зосереджені на якості, включають:

Рівень відмов. Частоту відмов вказує на те, як часто випуски програмного забезпечення призводять до неочікуваних збоїв, збоїв або серйозних аномалій. Низький відсоток невдалих змін свідчить про плавне та позитивне розгортання. Зворотне вказує на недостатню стабільність продуктивності додатків, що є масовим червоним прапорцем і вимагає негайного дослідження та вирішення проблеми. Високий відсоток невдалих змін сигналізує про суттєво поганий досвід кінцевого користувача, що знову ж таки вкрай негативно вплине на імідж бренду, довіру та дохід.

Рівень невдач збірки. Build Failure Rate відстежує кількість збірок, які виявилися невдалими (з будь-якої причини) протягом певного періоду часу. Він прагне бути барометром здоров'я продукту під час його розробки. У цьому випадку ступінь невдачі збірки або її причина не відстежується. Швидше, він просто відстежує кількість кінцевих успіхів і невдач.

Високий відсоток невдач збірки вказує на нестабільний продукт, який, можливо, страждає від помилок, які важко знайти, недоліків дизайну або навіть аномалій у процесі розробки. Переконайтеся, що показники відмов вимірюються

протягом певного проміжку часу, щоб намалювати чесну картину продуктивності програмного забезпечення.

Рівень автоматизації тестів. Слідкуйте за тим, як виконуються автоматизовані тести порівняно з ручними тестами. Якщо рівень відмов підозріло високий, подумайте, що помилка може полягати в механізмі налаштування інфраструктури, використанні інструментів автоматизації або сценаріях тестування. Тому цей показник необхідно ретельно контролювати, а аномалії необхідно негайно усувати.

Безпека. З розвитком технологій безпека стає все більш важливою для організацій, підприємств і клієнтів. Зменшення ризиків безпеки має вирішальне значення для забезпечення безпечної та конкурентоспроможної програми. Впровадження безпеки як етапу DevOps переносить процес на DevSecOps і дозволяє кожній організаційній команді визначати ризики та відповідність SLA на підготовчій стадії виробництва. Це допомагає запобігти збоям і проблемам з дотриманням або аудитом. Огляд ключових показників ефективності безпеки також допомагає команді зменшити кількість запитів, пов’язаних із безпекою, і дозволяє технологічній команді запускати безпечну програму в робочому стані. У міру того як компанії все більше усвідомлюють ризики безпеки в технологіях, ці KPI можуть продемонструвати конкурентну перевагу організації. Виміряйте свою безпеку за допомогою:

Швидкість уникнення дефектів. Цей показник відстежує, як часто дефекти виявляються на підготовчому виробництві порівняно з виробництвом. Це один із найцінніших показників якості програми, особливо в постпродакшн. Зауважимо, що деякі дефекти майже завжди втечуть у виробництво. Якщо вам пощастить, деякі з них буде виявлено за допомогою приймального тестування, але деякі можуть відображатися лише кінцевим користувачам.

Однак останнє число має бути мінімальним, наскільки це можливо для людини. Коефіцієнт уникнення дефектів відображає якість самого програмного забезпечення, а також процесу розробки, у якому воно створене.

Вразливість. Ця метрика фактично включає ряд інших метрик, усі призначені для перевірки працездатності, ефективності та безпеки операцій і результатів бізнесу. Кілька загальних показників уразливості, які можуть бути ефективними для механізмів DevOps, це: час для виявлення вразливості, час для локалізації/пом’якшення вразливості, ефективність керування виправленнями та часові рамки посилення системи.

Інфра/контейнерне сканування. Сканування контейнерів потрібні для перевірки зображень, які надсилаються у виробниче середовище. Це гарантує, що зображення контейнерів не мають критичних, високих і середніх уразливостей (CVE). Сканування контейнерів має вирішальне значення для розробки будь-яких хмарних програм, щоб уникнути атак на безпеку.

Операції. Операційна команда відповідає за підвищення ефективності всього процесу. Вони піклуються про показники протягом усього життєвого циклу. Справжній успіх DevOps означає збільшення швидкості та маневреності ваших команд. Операційний відділ зазвичай несе тягар обміну ключовими показниками ефективності з усіма зацікавленими сторонами для підвищення успіху на кожному етапі.

Нижче наведено два ключові KPI, які мають найбільше значення.

Середній час відновлення (MTTR). MTTR – це середній час, потрібний системі для виправлення та відновлення після збою. Він включає час від моменту збою системи до повернення до повної працездатності. Щоб отримати MTTR, обчисліть середнє значення всіх часів, необхідних для усунення всіх збоїв у певній системі.

Як очевидно, MTTR є невід’ємною частиною ефективного управління інцидентами, пропонуючи необхідну інформацію про швидкість, з якою вирішуються проблеми простоїв, щоб перезавантажити систему в повну функціональність.

MTTR = сума часу до тривалості відновлення / кількість інцидентів, які необхідно вирішити

Відповідність угоді про рівень обслуговування (SLA). SLA – це контракт (письмовий) між постачальником послуг і компанією. У ньому детально описано тип і рівень послуги, яка буде надаватися, показники, які вимірюватимуть цю послугу, і скільки часу знадобиться для відновлення послуги до оптимального рівня в разі проблеми.

Метрикою, про яку йде мова, є коефіцієнт відповідності SLA, який визначає вплив IT-послуги на роботу кінцевого користувача. Коефіцієнт відповідності SLA – це відсоток від загальної кількості проблем із обслуговуванням, вирішених у межах попередньо узгоджених критеріїв SLA, а також таких параметрів, як категорія проблеми, вартість, час, пріоритет тощо.

Коефіцієнт відповідності SLA = кількість IT-інцидентів, вирішених відповідно до відповідності SLA / Загальна кількість IT-інцидентів

Кількісні показники можна визначити за допомогою числового значення, яке допомагає визначити життєздатність вашого коду. Ці показники вимагають від вас використання формули, а також певних алгоритмів, які вимірюють якість коду з точки зору рівня складності. Ось деякі з ключових кількісних показників якості коду, які вам потрібно відстежувати:

Зважені мікрофункціональні точки. Метрика Weighted Micro Function Points (WMFP) – це найновіший програмний алгоритм визначення розміру, який став наступником наукових методів SOLID. Метрика аналізує вихідний код і фрагментує його на мікрофункції. Потім алгоритм використовує ці мікрофункції для створення кількох показників, що відображають різні рівні складності. Потім ці показники інтерполюються в єдиний рейтинг, який показує складність існуючого вихідного коду. Коментарі, арифметичні обчислення, структура коду та шлях керування потоком – це деякі з показників, які використовуються для визначення значення WMFP.

Міри складності Холстеда. Показники складності Холстеда оцінюють обчислювальну складність коду. Цей показник використовує такі показники, як

кількість операторів і операндів, для вимірювання складності коду з точки зору словникового запасу, довжини програми, помилок, складності, зусиль, розміру, часу тестування та кількості помилок у модулі. Чим більше складність, тим нижча якість коду.

Цикломатична складність. Цикломатична метрика складності вимірює структурну складність коду. Це міра кількості лінійно незалежних шляхів через вихідний код. Якщо цикломатична складність перевищує 10, код містить дефекти та потребує виправлення. Цей показник дає змогу розробникам зрозуміти, наскільки важко буде тестувати, підтримувати та налагоджувати їхній код. Поєднання цього показника з показниками розміру, такими як застави коду, дає змогу оцінити легкість, з якою код можна змінювати, модернізовувати та підтримувати.

Підтримка високої якості коду є ключовим аспектом для розробників. Будь-який погано написаний код може призвести до технічної заборгованості, проблем з продуктивністю та ризиків безпеки. Головною проблемою для команд розробників у забезпеченні високої якості коду є кодові бази, що постійно змінюються. Сучасна цифрова екосистема швидко змінюється завдяки розвитку технологій, а також очікувань клієнтів. Тому кодові бази, з якими працюють розробники, постійно змінюються. Команда розробників регулярно додає, видаляє та змінює існуючий код, щоб покращити швидкість або оновити нові функції. Однак ці постійні зміни коду часто погіршують якість коду. Саме тут показники якості коду виявляються корисними для розробників та дозволяють ще на початкових етапах врахувати можливі проблеми.

Показники якості коду відстеження дають змогу командам розробників аналізувати та знаходити те, що робить код читабельним, зрозумілим і високої якості. А висока якість коду означає кращу якість програмного забезпечення. А висока якість програмного забезпечення означає хороший проект. Саме тут якість коду може мати великий вплив, вимагаючи від розробників відстежувати показники якості коду.

2.3 Алгоритм оцінки якості програмного коду

В основу запропонованого алгоритму визначення якості програмого коду було закладено декілька класичних параметрів для проведення аналізу та визначення рівня читабельності програмого коду. Кожному з цих параметрів було надано відповідний ваговий коефіцієнт. Перелік параметрів наведено в таблиці 2.1.

Таблиця 2.1 – Параметри оцінювання якості програмого коду

Кількісні параметри	Позначення	Ваговий коефіцієнт, %	Якісні параметри	Позначення	Ваговий коефіцієнт, %
Кількість стрічок коду	N	10	Загальна читабельність	R	30
Кількість пустих стрічок	N0	10	Рівень зрозумілості назв змінних	VarName	20
Кількість змінних	V	15	Достатність коментарів	programComents	30
Кількість коментарів	Coment	15	Відповідність стандартам оформлення	stand	20
Середня довжина назви змінної	averVarName	10			
Кількість кількість функцій/методів	F	15			
Кількість циклів	C	15			
Кількість розгалужень	D	10			

Даний набір параметрів дозволяє провести оцінку програмного коду окремо використовуючи групу кількісних та якісних показників так і виконати аналіз окремо. При обчисленні кількісних показників дані отримують шляхом проведення аналізу над програмним кодом, при цьому дані показники є чіткими та при повторній оцінці будуть мати те саме значення. У випадку використання

набуру якісний параметрів оцінка буде напряму залежати від оператора, який проводить дослідження.

Для проведення оцінки запропонованого алгоритму його блок-схема наведена на рисунку 2.5.

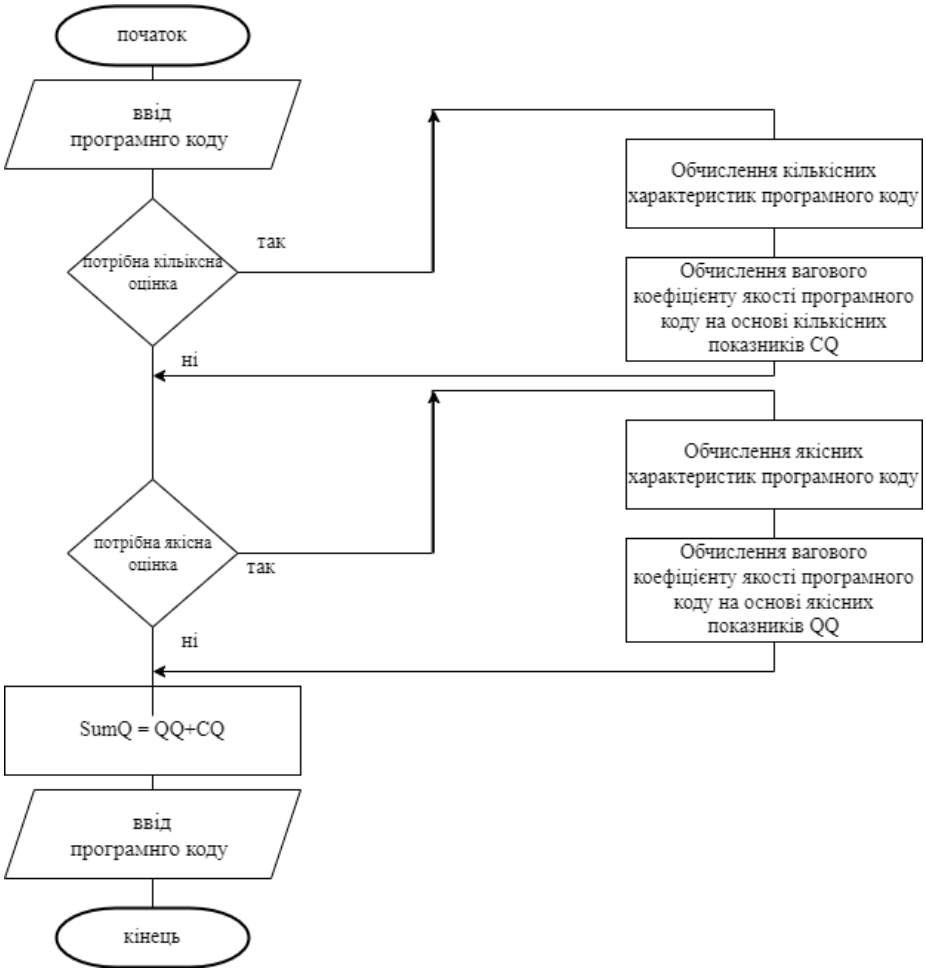

Рисунок 2.5 – Блок-схема алгоритму алгоритму оцінки якості програмного коду на основі кільісних та якісних показників

Для обчислення відповідних значень рівня якості програмного коду користувач має можлисість обрати які саме критерії є для нього більш важливими. Тому в загальному випадку для знаходження рівня якості програмнго коду буде проводитись за формулою:

$$CodeQuality = \alpha*CQ + \beta*QQ,$$

де α – ваговий коефіцієнт ваги кількісних показників;

β– ваговий коефіцієнт ваги якісих показників оцінки якості програмного коду.

На сонові запропонованого критерію оцінювання програмного коду було розроблено алгоритм оцінки рівня якості програмного коду мовою високого рівня. Алгоритм складається з таких кроків:

1) Завантаження програмного коду для аналізу.

2) Якщо при аналізу непотрібно використання кількісних параметрів , то переходимо на крок 5.

3) Аналіз програмного коду та отримання кількісних характеристик вхідного програмного коду.

4) Обчислення загальної оцінки програмного коду на основі кількісних показників *CQ*.

5) Якщо при аналізу непотрібно використання якісних параметрів, то переходимо на крок 8.

6) Аналіз програмного коду та отримання якісних характеристик вхідного програмного коду.

7) Обчислення загальної оцінки програмного коду на основі якісних показників *QQ*.

8) Визначення сумарної оцінки якості програмного коду на основі додавання сумарного кількісного коефіцієнтак та сумарного якісного коефіцієнтак з відповідними ваговими коефіцієнтами.

9) Вивід результатів роботи алгоритму з визначення рівння якості програмного коду.

Запропонований підхуд дозволяє провести оцінку проогамрного коду в залежності від потреб користувача, при цьому алгоритм може налаштовуватись для більш точного аналізу. Перевагами запропонованого алгоритму можна вважати наступні:

- в алгоритмі передбачено використання як субєктивних (якісних) так і обєктивних (кількісних) параметрів для отримання підсумкової оцінки програмного коду;

- обрані кількісні параметри обчислюються швидко, що дозволяє проводити дослідження з мінімальними часовими затримками;
- обрані якісні параметри є простими для розуміння і провести візуальну оцінку на онові обраного масику критеріїв можуть користувачі з різним рівнем підготовки;
- простота реалізації та можливість адаптації дорізних вимог.

Недоліки:

- на рівень визначення параметрів якісної оцінки буде впливати субєктивне враження користувача.

Наведені переваги та недолік дозволяє зробити висновок, що запропонований алгоритм виконуватиме усі поставлені завдання та може бути використанний для проектування та реалізації програмного додатку оцінки якості програмних кодів.

2.4 Висновки до розділу

Проведено аналіз сучасних вимог до якості програмного коду, на основі існуючих метрик, що дозволило провести їх класифікацію та виділити групу критеріїв які дозволяють максимально оцінити структуру та стиль написання програмного коду.

Розроблено алгоритм оцінки якості програмного коду на основі запропонованої формули сумарного оцінювання, що дозволило спроектувати та програмно реалізувати додаток для оцінювання програмного коду на основі кількісних та якісних критеріїв.

3 ПРОГРАМНИЙ ДОДАТОК АНАЛІЗУ ТА ОЦІНКИ ЯКОСТІ ПРОГРАМНОГО КОДУ

3.1 Структура додатку аналізу та оцінки програмного коду

Читабельність вихідного коду є важливою метрикою для характеристики обслуговування програмних систем [21]. Відповідно до [22], технічне обслуговування може становити до 40% витрат, виділених на розробку програмного забезпечення. Серед методів інженерії програмного забезпечення умовність кодування є одним із основних факторів, які мають значний вплив на читабельність вихідного коду. Умови кодування – це набір інструкцій із кодування, яких розробники програмного забезпечення повинні дотримуватися під час написання коду. Конвенції кодування – це правила, що охоплюють усі проблеми щодо покращення якості коду. Ці правила зазвичай класифікуються наступним чином: структура файлу, відступи, коментарі, пробіли, іменування ідентифікаторів, оголошення, оператори та практики, специфічні для мов програмування. Завдяки умовам кодування читачі можуть узгоджено дивитися на вихідний код і швидше розуміти вихідний код. Загалом, перший крок для новачків у більшості компаній, що займаються розробкою програмного забезпечення, передбачає ознайомлення з умовами кодування.

Було доведено, що угода про кодування, яка є частиною Clean Code, покращує продуктивність розробки програмного забезпечення та зменшує збої програм. Оскільки більшість систем програмного забезпечення розробляються та обслуговуються групою інженерів, правила кодування спрощують перегляд і перегляд коду кількома членами команди. Незважаючи на його важливість для індустрії програмного забезпечення, це питання значною мірою ігнорується в курсах програмування на університетському рівні. Однією з перешкод для застосування умов кодування в курсах програмування є складність перегляду програм, щоб визначити, наскільки вони відповідають визначеним умовам кодування.

Інструменти автоматичної перевірки коду порівнюють вихідний код зі списком встановлених найкращих практик, щоб визначити місця, які можна покращити. Але це лише один із багатьох способів покращити якість коду. Важко ігнорувати важливість хорошого коду для створення спеціального програмного забезпечення. Для досягнення цілей якості підтримка якості коду вимагає послідовних зусиль і відданої команди розробників програмного забезпечення. Це важливо для проекту програмного забезпечення, але коли бракує часу, розробники часто йдуть на компроміс щодо якості. Хоча може виникнути спокуса вдатися до кінцевого терміну, дуже важливо писати код, який відповідає найвищим стандартам якості коду. Це те, що може гарантувати, що ваш код витримає вимогливі умови програмного забезпечення.

При проектування структури програмного забезпечення для оцінки якості програмного коду були враховані особливості роботи реалізацій, які вже знаодяться на ринку. Окрім того, попередньо проведений аналіз показав основні структурні модулі, які використовуються розробниками при вирішенні подібних задач. Нижче наведено структуру програмного забезпечення для оцінки якості коду (рисунок 3.1).

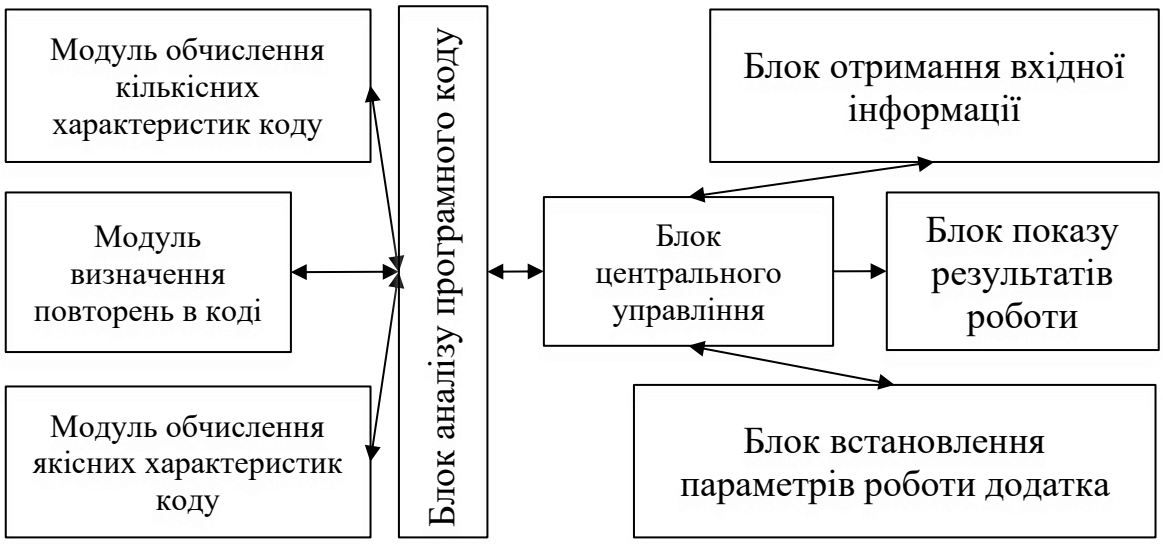

Рисунок 3.1 – Узагальнена структура додатку оцінки якості програмного коду написаного мовою високого рівня

Для більшого розуміння запропонованої архітектури проаналізуємо функціональні можливості кожного з виділених блоків окремо.

Блок центрального управління. Стурктурна одиниця, що є відповідальною за взаємозвязок усіх елементів програмної системи, встановлення початкових параметрів роботи програми, налагодження обмном повідомленнями з користувачами, конвертація та перевірка цілісності вхідних даних. Окрім того даний модуль містить систему допомоги користувача, що дозволяє за допомогою текстових повідомлень підказувати користувачеві його можливі кроки. Додатково система має можливість попередити користувача про критичні ситуації, що можуть виникнути підчас її функціонування. Функції даного модуля, фактично не взаємодіють з користувачами, вони дозволяють активувати відповідні програмні ресурси для надання користувачам можливостей виконати поставлені пред ними завдання.

Блок аналізу програмного коду. Лексичний аналізатор виконує початкове розбиття вихідного коду на лексеми чи токени, мінімальними елементами коду. Даний етап дозволяє створити набір елементарних конструкцій, які потім піддаються синтаксичному аналізу. Синтаксичний аналізатор будує абстрактне синтаксичне дерево (AST), що є ієрархією структур коду. AST є ключовим інструментом статичного аналізу коду. Статичний аналіз призначений для виявлення потенційних проблем та стильових помилок, які можуть впливати на читання, ефективність або безпеку коду. Цей модуль проводить обширний аналіз коду, виявляючи як явні помилки, а й тенденції, які можуть призвести до проблем у майбутньому. В результаті відбувається видача рекомендацій щодо покращення якості коду.

Модуль обчислення кількісних характеристик коду. Цей модуль включає лічильники, які вимірюють різні характеристики коду. Лічильники рядків коду визначають кількість рядків, коментарів та порожніх рядків, надаючи загальну статистику про код. Складність коду вимірюється цикломатичною складністю, яка допомагає визначити, наскільки складний код для розуміння та тестування. Глибина вкладеності вимірює рівень вкладеності циклів та умовних операторів,

виявляючи структурні особливості коду. Метрики розміру оцінюють обсяг коду, включаючи кількість функцій, класів та файлів. Ці метрики надають загальне уявлення про масштаб проекту та його структуру. Вимірювання розміру коду є ключовим компонентом для оцінки складності та обслуговування проекту.

Модуль визначення повторень в коді. Модуль виявлення дублювання коду здійснює аналіз проекту виявлення ідентичних чи схожих ділянок коду. Дублювання коду може призвести до складнощів в обслуговуванні та зміні проекту, тому його виявлення є важливим кроком у забезпеченні якісного коду.

Окремо, методи даного модуля проводят перевірку з можливістю виявити додаткові недодіки в читабельності коду, серед яких довгі методи, велика кількість параметрів, слабка структурованість та інші аспекти, що впливають на читання та ефективність коду.

Модуль обчислення якісних характеристик коду. До завданнь даного модуля відносяться засоби для організації можливості опитування користувачів для отримання оцінки програмного коду шляхом їх опитування. Методи реалізовані в даному модулі візуалізують послідовність запитань які можуть бути задані користувачеві, який повинен обрати одну з запропонованих оцінок. В результаті такого опитування та базуючить на вагових коефіцієнтак програмний додаток дозволить обчислити якісну оцінку програмного коду яку дав вибраний користувач. При потребі вона може скластись з кількісною оцінкою та в результаті отримати узагальнену оцінку.

Блок встановлення параметрів роботи додатка. Блок конфігурації та налаштування надає можливість тонкого налаштування параметрів аналізу та визначення правил, які відповідають конкретним вимогам проекту. Це важливий засіб, що дозволяє адаптувати систему оцінки якості коду до специфіки проекту та його особливостей. Користувачі мають можливість вибирати метрики, які найкраще відповідають цілям проекту. Наприклад, якщо акцент робиться на продуктивності, то можна встановити вагові коефіцієтни саме для відвищення впливу таких характеристик як кількість циклів або розгалужень. У разі

фокусування на чистоті коду та його стилі, більші вагові коефіцієнти встановлюються саме на кількості коментарів зрозумілості назв змінних тощо.

Блок отримання вхідної інформації. Дана структурна одиниця є допоміжною та дозволяє налагодити взаємодію між програмним додатком, файловою системою та користувачем. Функції даного модуля надіють користувачеві можливість вибору файлу для проведення аналізу, а при його відсутності або винекненні помилки (відсутність прав доступу, пошкоджений файл тощо) повідомити про це користувача. Результатами роботи даного модуля є завантажений файл який містить програмний код для проведення процедури аналізу.

Блок показу результатів роботи. Даний блок є допоміжним та дозволяє користувачеві отримани результати роботи програмного додатку у зручному для нього форматі. В загальному випадку програма показує обчислені кільтісні та якісні характеристики та узагальнену оцінку аналізованого програмного коду. Для підвищення зрутності роботи з програмою користувачі через модуль встановлення параметрів робти програмного додатку мають можливість налаштувати деякі параметри виводу інформації, кольор, розмір тексту, тип шрифту.

Обрана за основу, модульна архітектура має ряд переваг, серед яких можливість поетапної розробки окремих модулів не залежно один від одного, що дозволяє зробити процес реалізації програмного коду більш гнучким та менш залежним від сторонніх факторів. Крім того програмні додатки з такою архітетурою простіше піддаються масштабуванню як горизонтальному по додаванні нових алгоритмів у вже існуючі модулі, так і вертикальному. Процес тестування не залежить від наявності цілосної програми, адже модулі функціонують незалежно один від одного, проте при цьому небхідно контрольовати структури даних які передаються на вхід кожного з модулів. В загальному запропонована структура враховує всі основіні тенденції по розробці програмних засобів автоматичної оцінки програмних кодів та дозволяє вирішити усі поставлені перед розробником задачі.

На наступному етапі було проведено моделювання запропонованого алгоритму та структури програмного додатку. Даний етап дозволить оцінити на теоретичному рівні станцілісності запрпонованої структури, наявність усіх необхідних модулів та вияснити чи у структурі відсутні внутрішні конфлікти.

Для проведення процесу моделювання було використано інструментарій універсальної мови моделювання UML. Вибір даних технологій базувався на тому, що UML діаграми надають можливість візуалізації динамічних аспектів системи, таких як взаємодія між об'єктами, зміна станів та послідовність дій. Це особливо корисно при аналізі та проектуванні складних систем, де важливо розуміти внутрішню взаємодію компонентів. Динамічні діаграми, такі як діаграми послідовності та діаграми станів, допомагають краще моделювати та передбачати поведінку системи.

Окрім того, використання UML діаграм підтримує принцип модульності та повторного використання коду. Модель, створена з використанням UML, може бути основою створення документації і автоматичної генерації коду. Це прискорює процес розробки, зменшує ймовірність помилок та сприяє підтримуваності проекту на всіх етапах його життєвого циклу.

Таким чином, використання UML діаграм є потужним інструментом, що сприяє більш ефективній комунікації, абстракції та візуалізації в процесі розробки програмного забезпечення, що в кінцевому підсумку сприяє створенню більш якісних і легко підтримуваних систем.

Для провдення тестування було обрано ряд найбільш часто використовуваних діаграм, а саме діаграму послідовності та діаграму прецедентів. Дані діграми дозволяють провести оцінку структури програмного доадатку та апроаналізувати фактори, що п подальшому дозволять провести корекцію структури розробленої програмної системи на етапі проектування та не витрачати ресурси на виправлення проблем на етапі повної або часткової реалізації. Діаграма прецедетів дозволяє визначити які можливості при роботі з програмною системою отримають різні групи користувачів. Приклад самої діаграми наведено на рисунку 3.2.

Рисунок 3.2 – Діаграма прецедентів програмної системи

В результаті проведеного моделювання було виявлено, що при користуванні програним додатком основною групою користувачів буде група акторів «Користувачі». Дана група людей буде використовувати програмний додаток для автоматизованого оцінювання програмного коду на основі обраного набору параметрів. Параметри налаштування програми можна розділити на дві групи: параметри проведення оцінки коду та параметри візуального відображення результатів роботи. Дані користувачі мають доступ до корегування даних параметрів, що дозволить їх більш точно налаштувати функціонал додатку при вирішення більш складних завдань. Окрім того, можливість налаштувати параметри воводу інформації на екран надає користувачам більш гнучні можливості для отримання кінцевого результату у більш зручному форматі. Доступні функції не є системними і некоректне введення відповідних параметрів не призведе до поломки системи вцілому, тому отримані під час моделювання результати можна вважати відмінними, а запропоновану структуру такою, що підходить для реалізації програмного додатку.

Після дослідження можливостей доступу користувачів до різних функціональних можливостей програмної розробки, неаступним етапом аналізу запропонованої структури є дослідження протікання процесів під час роботи самої системи. Дане дослідження проводиться з метою пошуку накладання процесів під час робтипрограми, а також для перевірки колізій в самій системі. Результат моделювання наведено на рисуноку 3.3.

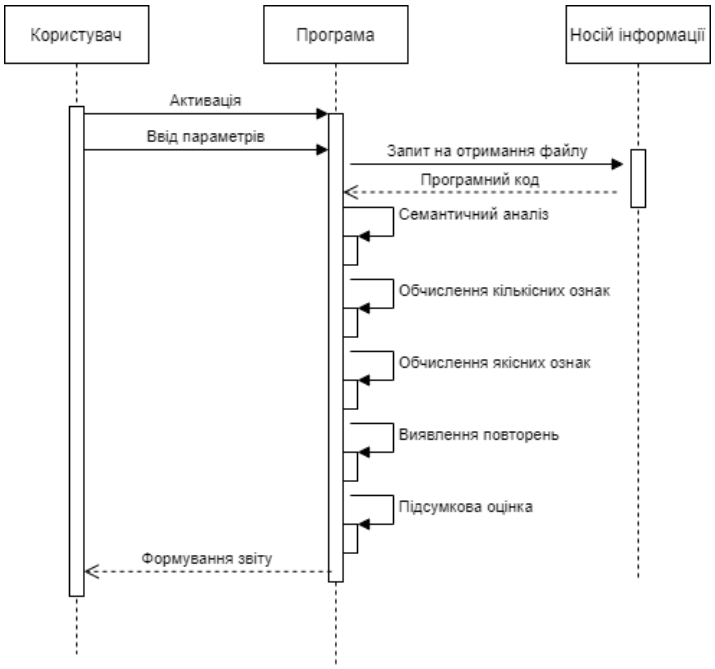

Рисунок 3.3 – Діаграма послідовності програмної системи

Основном завдання моделювання за допомогою діаграми посідовностей є аналізу запуску процесів, що відуваються в середині програмного додатку. Отримані результати підтвердили, що внутрішні процеси програмного додатку запускаються у чіткій послідовності, не перекриваються між собою та не конфліктують через доступ до ресурсів. Основні елементи прооограми діють згідно спроектованого алгоритму, блок керування слідкує за коректністю обміну даними між різними структурними блоками, дані передаються у відповідних форматах.

Основною метою проведення етапу моделювання була перевірка можливостей спроектованої структури програмного забезпечення виконувати

всі поставлені завдання, аналізу запропонованих структур даних для зберігання та обробки вхідних даних та чіткого виконання пунктів запропонованого алгоритму аналіза та оцінки програмного за допомогою узагальнюючої оцінки на основі виділення кількісних та якісних ознак. В результаті моделювання було виявлено, що програмний додаток має цілісну структуру побудовану принципах модульної архітектури, в структурі відсутні конфлікти між окремими модулями і дану структуру можлна використовувати для програмної реалізації додатку оцінки програмного коду.

Для підвищення рівня зручності користування програмним додатком було спроектовано та програмно реалізовано графічний інтерфейс користувача для рботи з програмним додатком. Запропонований дизай складається з трьох основних части, які мають різні функціональні завдання. В верхній лівій частині присутнє системне меню за допомогою якого користувач має можливість взаємодіяти як з операційною системою так снутрішніми можливостіми по налаштування програмного додатку. Додатково там присутня система допомоги користувачу, де ознайомитись з основними функціями робти програми. Центральну частину вікна займає текстовіредактори. За допомогою яких користувач може візуально побачити аналізований код, а також результати роботи програми. В нижній частині розміщена область доступу до швидкиш клавіш керування програмним додатком.

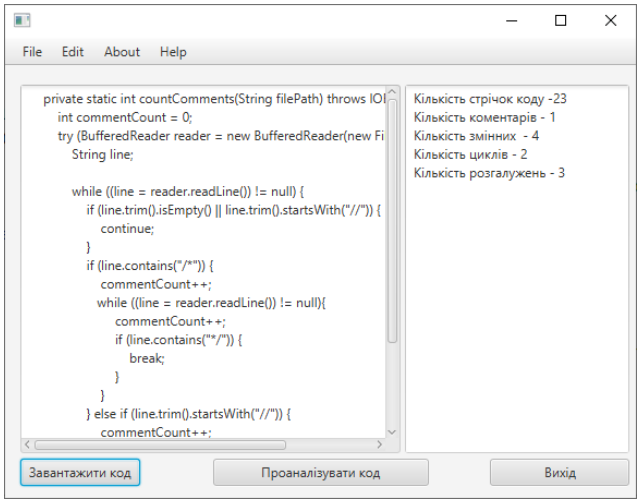

Рисунок 3.4 – Графічний інтерфейс програмного додатку

Запропонований інтерфейс є зручним у користуванні, оскільки основний візуал формується невеликою кількістю інтуїтивно зрозумілих елементів, що значно спрощує робту з програмною системою.

Для більшого розуміння розглянемо дії користувача при роботі з запрпонованою програмною системою. Дії користувача можнарозділити на такі дві гру: основні та другорядні. Послідовність необхідних дій відображено на рисунку 3.5:

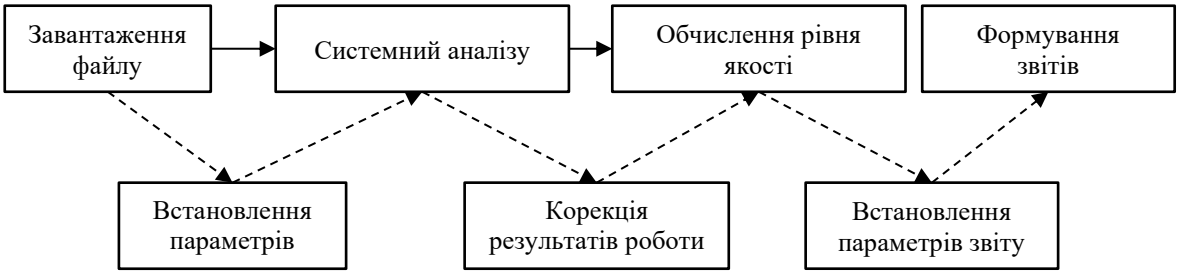

Рисунок 3.5 – Послідовність дій користувача при роботі з програмним додатком оцінки якості програмного коду

Для старту роботи програми користувач повинин завантажити програмний код у відповідну текстову область. Це буде слугувати сигнадом для початку процесу опрацювання програмного коду за допосогою первинного семантичного аналізу з метою виділити основі структурні одиниці. Даний процес проводиться без втручання користувача та на основі запрограмованих внутрішніх алгоритмів. На даному етапі користувач при бажанні може внести корективи в отримані результати шляхом корегування параметрів робти алгоритму. Наступний етап це обчислення кількісних та якісних характерисни програмного коду. Даний процес проводиться в напів автоматичному режимі, при якому кількісні ознаки отримуються на основі аналізу програмного коду шляхом його поелементного аналізу. В той самий час, якісні ознаки отримаються в резулультаті проведення опитування користувача. На останньому етапі проводяться підсумкові обчислення та формування звіту про робту програмного додатку.

3.2 Модулі програмного додатку аналізу та оцінки програмного коду

Для реалізації програмного засобу було обрано мову програмування java, як одну з найбільш поширених мов програмування. Окрім того, мова програмування Java міцно утвердилася у світі інформаційних технологій і стала однією з найпопулярніших мов для розробки програмного забезпечення. Її універсальність та великі можливості роблять її відмінним вибором для реалізації програм аналізу та оцінки програмного коду. Java є кросплатформною мовою програмування, що означає, що програми, написані на Java, можуть виконуватися на різних операційних системах без змін у їхньому вихідному коді. Ця властивість робить мову Java ідеальною для розробки інструментів аналізу коду, які повинні бути доступні на різних платформах, що забезпечує зручність використання та розповсюдження таких інструментів у різноманітних середовищах розробки. Окрім того, Java має велику кількість бібліотек класів і фреймворків. Ці бібліотеки включають безліч інструментів для обробки файлів, роботу з регулярними виразами, маніпуляції рядками, а також для роботи з архітектурою самої мови. Доступ до таких багатих ресурсів спрощує розробку та реалізацію складних алгоритмів аналізу коду, таких як підрахунок змінних, оцінка складності алгоритмів, аналіз структури програм та інші. Також, було враховано інтеграцію Java у різні середовища розробки. Багато сучасних IDE, такі як IntelliJ IDEA, Eclipse, і NetBeans, надають широку підтримку для Java, що спрощує створення, налагодження та тестування програм, у тому числі інструментів аналізу коду. Мова Java надає широкі можливості розробки програм аналізу та оцінки програмного коду. Її кросплатформенність, велика бібліотека класів, підтримка багатопоточності, безпека та підтримка спільнотою роблять її привабливим вибором для створення ефективних та потужних інструментів, здатних покращити якість та структуру програмного коду.

Для реалізації цьогої програмної системи було розроблено ряд допоміжних класів та методів, які довволять реалізувати запропоновані алгоритми та провести повноцінних аналіз файлів з програмними кодами.

Для більш детального аналізу наведемо приклади частини реалізованих класів та методів. На етапі тестування розроблений програмний додаток виконувався в консольному режиміта дозволяв опрацьовувати файли, які містили програмний код написаний мовою Java. Даний пыдхыд був використаний з метою зменшення часу на написання та відлагодження коду програми. Розробка графічного інтерфейсу та можливості аналізувати інші мови програмування була реалізована на наступних етапах.

Для проведення стартових налаштування та отримання програмного коду був реалізований клас CodeMetrics (рисунок 3.6). Даний клас є стартовим для роботи з файлами. Для початку він завантажує файл для аналізу та запускає відповідні методи для продовження його обробки. Додатково здійснений захист від ситуації коли обраний файл буде пошкоджений або відсутній. При використання механізму виключень прграмний додаток буде продовжувати функціонувати незалежно від того чи відбудеться збій в системі завантаження файлу

```
public class CodeMetrics {

    public static void main(String[] args) {
        String filePath = "testJavaCode.java";
        try {
            int linesOfCode = countLinesOfCode(filePath);
            System.out.println("Кількість рядків коду: " +
linesOfCode);
        } catch (IOException e) {
            System.err.println("Помилка при зчитуванні файлу:
" + e.getMessage());
        }
    }
```

Рисунок 3.6 – Приклад реалізації класу аналізу файла

Для проведення процедури оцінки програмного коду необхідно отриманити кількісні характеристики вхідного файлу. Для цього було

реалізована ряд методів які дозволяють на основі вмісту вхідного файла обчислити різні характеристики. Першою характиристикою, що дозволить описати отриманий програмний код буде параметр “кількість стрічок коду”. Даний парметр дозволить оцінити загальний об’єм коду та молиість його швидкого прочитання та розуміння. Згідно з рекомендаціями кількість стрічок програмного коду в одному файлі не повинна первишувати 2000. Якщо кількість більше, то необхідно провести декомпозицію програмних завдань з метоб розбиття задач на менші підзадачі. Метод countLinesOfCode() реалізований алгоритм підрахунку кількості не порожніх стрічок, що містять корисну інформацію (рисунок 3.7).

```
private static int countLinesOfCode(String filePath) throws
IOException {
    int lines = 0;
    try (BufferedReader reader = new BufferedReader(new
FileReader(filePath))) {
    String line;
    while ((line = reader.readLine()) != null) {
    if(!line.trim().isEmpty()&&!line.trim().startsWith("//"))
    {lines++;}
            }}
        return lines;
    }}
```

Рисунок 3.7 – Приклад реалізації методу класу аналізу програмного коду

Використання циклу з складною умовою дозволяє виділити тільки ті стрічки які є не порожніми та не розпочинаються з символа зворотній слеш. Друга умова використовується для того, щоб під час аналізу автоматично відкидались стрічки які містять коментарі, адже для оцінки кількості коментарів реалізований інший метод.

```
while ((line = reader.readLine()) != null) {
if(!line.trim().isEmpty()&&!line.trim().startsWith("//"))
```

для отримання іншої характеристичної ознаки, а саме кількості стрічок з коментарями реалізований метод countComments(String filePath) (рисунок 3.8). Назву файла для аналіза даний метод отримує у вигляді параметра. Даний метод

шляхом пострічкового аналізу вмісту вхідного файлу проводить пошук стрічок, що розпочинають з символа зворотнього слеш. Таким символом прийнято виділяти стрічки з програмним кодом від стрічок з коментарями розробників.

```
private static int countComments(String filePath) throws
IOException {
        int commentCount = 0;
        try (BufferedReader reader = new BufferedReader(new
FileReader(filePath))) {
            String line;
            while ((line = reader.readLine()) != null) {
     if(line.trim().isEmpty()||line.trim().startsWith("//")){
continue; }
            if (line.contains("/*")) {
            commentCount++;
            while ((line = reader.readLine()) != null) {
                    commentCount++;
                    if (line.contains("*/")) {break;}
                    }
                } else if (line.trim().startsWith("//")) {
                    commentCount++;
                }}}
        return commentCount;
    }
```

Рисунок 3.8 – Приклад реалізації методу класу аналізу програмного коду

Окрім перевірки на зворотній слеш відбувається перевірка на послідовність симовлів “*/”. Дана послідовість завершує багатострічковий коментар та може знаходитись в програмному коді в окремій стрічці, що може спотворити отримані результати аналізу програмного коду.

Ще одним методом, який був реалізований є метод countVariables() (рисунок 3.9) У розробленому методі реалізований функціонал підрахунку змінних програмного коду. Це завдання важливе при аналізі кодової бази, оскільки кількість змінних може бути одним із показників складності програми, а також може бути корисною при плануванні рефакторингу. У метод countVariables(), відбувається покрокове читання вмісту файлу, рядок за рядком. При цьому цикл зупиниться у випадку досягнення зацінчення файлу, що в свою чергу дозволить коректно призупинити процес аналізу.

```
while ((line = reader.readLine()) != null){}
```

```
    private static int countVariables(String filePath) throws
IOException {
        Set<String> variables = new HashSet<>();
        try (BufferedReader reader = new BufferedReader(new
FileReader(filePath))) {
            String line;
            Pattern variablePattern =
Pattern.compile("\\b(?:int|float|double|char|boolean)\\s+([a-
zA-Z_][a-zA-Z0-9_]*)");
            while ((line = reader.readLine()) != null) {
            Matcher matcher = variablePattern.matcher(line);
                while (matcher.find()) {
                    variables.add(matcher.group(1));
                }
            }
        }
        return variables.size();
    }
```

Рисунок 3.9 – Приклад реалізації методу класу аналізу програмного коду

Для пошуку оголошень змінних використовується регулярний вираз, який виражає сигнатуру оголошення змінної типу int, float, double, char або boolean.

```
"\\b(?:int|float|double|char|boolean)\\s+([a-zA-Z_][a-zA-Z0-
9_]*)"
```

Знайдені змінні додаються до множини для забезпечення унікальності, і в кінці програма виводить загальну кількість унікальних змінних.

```
Matcher matcher = variablePattern.matcher(line);
```

Цей програмний підхід має кілька ключових переваг. По-перше, використання регулярних виразів дозволяє гнучкіше і точно визначити місця, де оголошуються змінні. По-друге, використання множини для зберігання змінних гарантує, що кожна змінна враховується лише один раз, що особливо корисно при підрахунку унікальних елементів. Даний метод є одним з складових елементів при аналізі кодової бази для оцінки її складності та можливих напрямів оптимізації. Однак важливо розуміти, що ця реалізація фокусується виключно на базових типах даних, і для повного аналізу змінних у складніших структурах коду можуть знадобитися складніші методи та інструменти.

3.3 Тестування та аналіз реалізованого програмного додатку

Автоматизовані системи оцінювання все частіше використовуються на на різноманітних фірмах та під час освітнього процесу для оцінювання розв’язаних програмістами завдань з програмування. Автоматизована стсема зазвичай виконує статичний і динамічний аналіз програмного коду. Крім того, прості форми аналітики навчання часто можна створити досить легко. Проте структурний аналіз і порівняння рішень для більших наборів програмних кодів у багатьох випадках є складними та трудомісткими.

Для проведення тестування автоматизованого оцінювання програмного коду було використано робочу станцію з усередненими параметрами офічного ПК. Вибір був зубомовлений поширеністю такий рботчих станцій в навчальних закладах та фірмах, що спеціалізуються на розробці невеликих програмних систем, а програмісти як правило є початківцями з невеликим досвідом розробки. Технічні параметри наведено на рисунку 3.10.

Назва параметра	Значення
Процесор (CPU):	Intel Core i5-9400 (6 ядер, 2.9 ГГц, до 4.1 ГГц Turbo Boost, 9 МБ кешу)
Оперативна пам'ять (RA	8 ГБ DDR4
Жорсткий диск:	SSD 512 ГБ
Графічний процесор (GF	Вбудована графіка Intel UHD Graphics 630
Операційна система:	Windows 10 Professional (64-біт)
Материнська плата:	MicroATX
Бездротові технології:	Wi-Fi 802.11ac, Bluetooth 5.0
Порти та роз'єми:	USB 3.0/3.1, HDMI, DisplayPort, Ethernet (RJ-45)
Корпус:	Mid Tower
Блок живлення:	400 Вт

Рисунок 3.10 – Технічні параметри робочої станції для тестування

Ця конфігурація надає достатню продуктивність для офісних завдань, таких як робота з текстовими документами, електронними таблицями, інтернет-браузингом та використанням офісних програм. SSD забезпечує швидкий доступ

до даних, а процесор та оперативна пам'ять дозволяють легко справлятися із повсякденними завданнями. Вбудована графіка, що забезпечує достатню продуктивність для офісних додатків, але не призначена для ігор або графічних завдань високого рівня.

Для отримання більш об’єктивних результатів тестування для різих типів програмних кодів було обрано ряд критеріїв яким повинні відповідати аналізовані тестові програмні коди. Серед основних критеріїв кількісних показників це:

- довжина коду;
- кількість змінних в коді програми по відношенню до кількості стрічок;
- кількість циклічних операторів по відношенню до кількості стрічок;
- кількість операторів розгалуження по відношенню до кількості стрічок.
- кількість коментарів.

Проведені тести показали наступні результати, що оптимальною довжиною одного лістингу програмного коду є довжина 200-400 інформативних стрічок (рисунок 3.11). При такій кількості інформації програмісти може легко та швидко проаналізувати написаний код ті вона викликає найменший дискомфорт. Програмні коди нелеликого розміру мають нижчий рівень якості, оскільки розбиття програмного коду не невеличні частини заставляє програміста запамятовувати не програмний код, а структуру файлів.

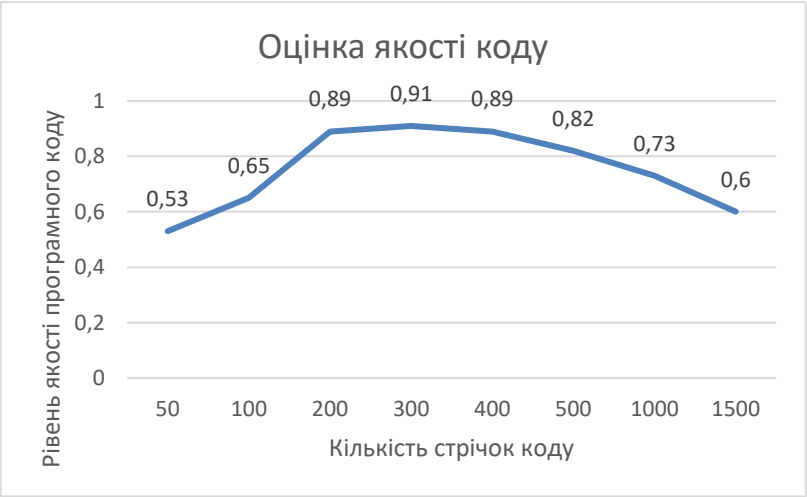

Рисунок 3.11 – Порівняння рівня читабельності коду відносно кількості стрічок

Для підсумовування отриманих результатів тана основі попередньо обраних критеріїв було виділено три групи програмних кодів, які будуть узагальнювати типові приклади:

1) Програмні коди невеликого розміру (до 500 стрічок), кількість змінних – мінімальна. Основний тип алгоритмів, що реалізуються – лінійні. Кількість коментарів – мінімальна. Назви змінних середньої довжини максимум 3 змістових слова.

2) Програмні коди середнього розміру (500 - 1500 стрічок), кількість змінних – середня. В модулі реалізуються усі типи алгоритмів, але переважно лінійні. Кількість коментарів – присутня незначна надлишковість. Назви змінних середньої довжини максимум 4 змістових слова.

3) Програмні коди великого розміру (понад 1500 стрічок), кількість змінних – велика (присутні невикористовувані змінні). В модулі присутні усі типи алгоритмів, переважно циклічні та розгалуження. Кількість коментарів – надлишкова. Назви змінних не стандартизовані (як великі так і маленькі).

В результаті учі отримані результати були погруповані відносно запропонованих груп та представлені в таблиці 3.1

Таблиця 3.1 – Узагальнена таблиця результатів тестування

Тестові групи	Оцінка рівня читабельності	Швидкість розуміння нового програмного коду	Можливість внесення модифікацій в існуючий програний код	Можливість інтеграції у власний проект	Загальне враження від кодів
Перша група	Високий рівень зручності читабельності	Високий рівень розуміння	Швидка адаптація	Проста інтеграція	Позитивне
Друга група	Достатній рівень зручності читабельності	Високий рівень розуміння	Необхідний час на аналіз	Проста інтеграція	Скоріш позитивне
Третя група	Важко читати програмний код	Необхідний час на аналіз	Необхідний значний час на аналіз	Необхідний додатковий час	Скоріш негативне

На основі отриманий результатів можна зробити висновки, що запропонований алгоритм оцінки якості програмного коду на основі використання підходів структурного аналізу та поєднанні кількісний та якісних критеріїв оцінки програмного коду є достатньо надійним та може використовуватись з ріних систмах оцінки знань.

Якщо до фрагмента коду легко додавати та видаляти функції, він вважається «обслуговуваним». Підтримка неякісного коду може зайняти більше часу, ніж зазвичай. Крім того, очікування на покращення може бути величезною тратою часу та грошей. Загальна вартість і якість коду тісно пов'язані. Функціональні тести якості, на відміну від керівництв зі стилю та лінтерів, визначають, чи дійсно ваш код працює.

В загальному можна виділити наступні фактори, що сприяють отриманню якісного програмного коду:

- Додайте коментарі до кожного реалізованого методу, функції та частини логіки.
- Використання правильних типів даних і імен для змінних, класів та інших елементів даних.
- Переконайтеся, що використовуєте правильний стиль і мову коду.
- Впровадження автоматизованого тестування як передумови гарантує, що програмне забезпечення відповідає всім критеріям і реагує на різні вхідні дані.
- Досягніть мети зменшення складності класу шляхом рефакторингу та вибору належного шаблону проектування.
- Спробуйте змінити призначення коду загального призначення.

Виконання модульного тесту є одним із найкращих методів забезпечення високоякісного коду. Завдяки моделюванню поведінки зовнішніх залежностей це дає змогу перевірити окрему частину програмного забезпечення. Тоді як інтеграційне тестування розглядає, як різні компоненти працюють разом, наскрізне тестування досліджує весь цикл клієнт-сервер.

Дотримуватися суворих стандартів якості коду так само важливо, як і писати сам код. Будь-коли на етапі розробки спеціального програмного забезпечення якість коду можна покращити. Це дає змогу враховувати основні характеристики якості коду та використовувати стандарти кодування, щоб отримати від цього користь.

3.4 Висновки до розділу

Розроблено узагальнену структуру та проведено її моделювання на основі використання інструментації uml діаграм, що дозволило програмно реалізувати додаток аналізу та оцінки програмного коду на основі структурного аналізу.

Здійснено практичне тестування розробленого додатку з використання виділених тестових груп, що показало коректність роботи запропонованого алгоритму та структури додатоку аналізу та оцінки програмного коду на основі структурного аналізу.

ВИСНОВКИ

На основі аналізу існуючих систем аналізу та оцінки програмних кодів мовою високого рівня, а також запропонованому алгоритмові та структурі програмної системи можна зробити такі висновки:

1. Проведено дослідження мов програмування на основі їх структурних особливостей, функціональних можливостей та сфер застосування, що дозволило провести їх класифікацію та визначити групу програмних мов які найчастіше використовуються при створенні програмних продуктів.

2. Проведено аналіз використання технологій та інструментів структурного аналізу під час створення програмного коду, що дозволило виділити основні елементи в структурі програмного коду та описати ситуації які сприяють погіршенню оцінки програмної реалізації.

3. Проведено дослідження інтегрованих середовищ розробки, що дозволило виділити та додатково проаналізувати основні функціональні складові які виконують основні задачі програмних систем даного типу.

4. Проведено аналіз сучасних вимог до якості програмного коду, на основі існуючих метрик, що дозволило провести їх класифікацію та виділити групу критеріїв які дозволяють максимально оцінити структуру та стиль написання програмного коду.

5. Розроблено алгоритм оцінки якості програмного коду на основі запропонованої формули сумарного оцінювання, що дозволило спроектувати та програмно реалізувати додаток для оцінювання програмного коду на основі кількісних та якісних критеріїв.

6. Здійснено практичне тестування розробленого додатку з використання виділених тестових груп, що показало коректність роботи запропонованого алгоритму та структури додатоку аналізу та оцінки програмного коду на основі структурного аналізу.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Тимчук Є.І., Далекий М.Р. Оцінка якості програмного коду на основі аналізу рівня його читабельності. Збірник тез VIII Науково-практична конференція молодих вчених і студентів «Інтелектуальні комп'ютерні системи та мережі»., Тернопіль, 05 грудня 2023 р. с. 15.

2. Далекий М.Р., Тимчук Є.І. Оцінка рівня агресії на основі аналізу текстових повідомлень. Збірник тез VIII Науково-практична конференція молодих вчених і студентів «Інтелектуальні комп'ютерні системи та мережі»., Тернопіль, 05 грудня 2023 р. с. 17.

3. Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating Sequences from Structured Representations of Code. In Proceedings of International Conference on Learning Representations. – 2018 – p. 24-31.

4. Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. Proceedings of the ACM on Programming Languages 3, POPL (2019), 1–29.

5. Nadezhda Chirkova and Sergey Troshin. 2021. Empirical study of transformers for source code. In Proceedings of 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 703–715.

6. Kevin Clark, Urvashi Khandelwal, Omer Levy, and Christopher D. Manning. What Does BERT Look at? An Analysis of BERT's Attention. In Proceedings of the 2019 ACL Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP. 2019. - 276–286.

7. Kevin Clark, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning. / ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators. In Proceedings of International Conference on Learning Representations. 2020.

8. Alexis Conneau, Germán Kruszewski, Guillaume Lample, Loïc Barrault, and Marco Baroni. What you can cram into a single vector: Probing sentence embeddings

for linguistic properties. In Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics. 2018.- 2126–2136.

9. Gregory W. Corder and Dale I. Foreman. Nonparametric statistics: A stepby-step approach. John Wiley & Sons. - 2014.

10. Jacob Devlin, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. 2019. - 4171–4186.

11. Li Dong, Nan Yang, Wenhui Wang, Furu Wei, Xiaodong Liu, Yu Wang, Jianfeng Gao, Ming Zhou, and Hsiao-Wuen Hon. Unified Language Model Pretraining for Natural Language Understanding and Generation. In Proceedings of Advances in Neural Information Processing Systems. 2019. - 1342–1354.

12. Kawin Ethayarajh. How Contextual are Contextualized Word Representations? Comparing the Geometry of BERT, ELMo, and GPT-2 Embeddings. Processing and the 9th International Joint Conference on Natural Language Processing. 2019. - 55–65.

13. Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In Proceedings of Findings of the Association for Computational Linguistics: EMNLP 2020. 1536–1547.

14. Xiaodong Gu, Hongyu Zhang. Deep code search. In Proceedings of 40th International Conference on Software Engineering. 2018. 933–944.

15. Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou. GraphCodeBERT: Pre-training Code Representations with Data Flow. In Proceedings of 9th International Conference on Learning Representations. 2019

16. John Hewitt and Christopher D. Manning. A Structural Probe for Finding Syntax in Word Representations. In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. 2019 – 429–438.

17. Benjamin Hoover, Hendrik Strobelt, and Sebastian Gehrmann. exBERT: A Visual Analysis Tool to Explore Learned Representations in Transformer Models. In Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics: System Demonstrations. 2020. - 187–196.

18. Phu Mon Htut, Jason Phang, Shikha Bordia, and Samuel R. Bowman. Do Attention Heads in BERT Track Syntactic Dependencies? CoRR abs/1911.12246 (2019).

19. Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. CoRR abs/1909.09436 – 2019.

20. Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and Evaluating Contextual Embedding of Source Code. In Proceedings of the 37th International Conference on Machine Learning, Vol. 119. 5110–5121.

21. Hong Jin Kang, Tegawendé F. Bissyandé, and David Lo. Assessing the Generalizability of Code2vec Token Embeddings. In Proceedings of 34th IEEE/ACM International Conference on Automated Software Engineering. IEEE, 1–12.

22. Taeuk Kim, Jihun Choi, Daniel Edmiston, and Sang-goo Lee. Are Pretrained Language Models Aware of Phrases? Simple but Strong Baselines for Grammar Induction. In Proceedings of 8th International Conference on Learning Representations. 2021.

23. Olga Kovaleva, Alexey Romanov, Anna Rogers, and Anna Rumshisky. 2019. Revealing the Dark Secrets of BERT. In Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing. 4364–4373.

24. Lucien Le Cam and Grace Lo Yang. Asymptotics in statistics: some basic concepts. Springer Science & Business Media. 2012

25. Nelson F. Liu, Matt Gardner, Yonatan Belinkov, Matthew E. Peters, and Noah A. Smith. Linguistic Knowledge and Transferability of Contextual Representations. In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language

Technologies, NAACL-HLT. 1073–1094. [24] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. CoRR abs/1907.11692 (2019).

26. Christopher Manning and Hinrich Schutze. 1999. Foundations of statistical natural language processing. MIT press.

27. Timothee Mickus, Denis Paperno, Mathieu Constant, and Kees van Deemter. 2019. What do you mean, BERT? Assessing BERT as a Distributional Semantics Model. CoRR abs/1911.05758 (2019).

28. Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In 1st International Conference on Learning Representations, Workshop Track Proceedings.

29. Tomás Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In Proceedings of Advances in Neural Information Processing Systems. 311–319.

30. Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep Contextualized Word Representations. In Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. 227–237.

31. Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. 419–428.

32. Anna Rogers, Olga Kovaleva, and Anna Rumshisky.. A primer in bertology: What we know about how bert works. Transactions of the Association for Computational Linguistics 8 (2020), 842–866.

33. Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural Machine Translation of Rare Words with Subword Units. In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics.

34. Yikang Shen, Zhouhan Lin, Athul Paul Jacob, Alessandro Sordoni, Aaron C. Courville, and Yoshua Bengio. 2018. Straight to the Tree: Constituency Parsing with Neural Syntactic Distance. In Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics. 1171–1180.

35. Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. 2019. MASS: Masked Sequence to Sequence Pre-training for Language Generation. In Proceedings of the 36th International Conference on Machine Learning, Vol. 97. PMLR, 526–536.

36. Yulei Sui, Xiao Cheng, Guanqin Zhang, and Haoyu Wang. 2020. Flow2vec: Valueflow-based precise code embedding. Proceedings of the ACM on Programming Languages 4, OOPSLA (2020), 1–27.

37. Yu Sun, Shuohuan Wang, Yu-Kun Li, Shikun Feng, Xuyi Chen, Han Zhang, Xin Tian, Danxiang Zhu, Hao Tian, and Hua Wu. 2019. ERNIE: Enhanced Representation through Knowledge Integration. CoRR abs/1904.09223 (2019).

38. Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to sequence learning with neural networks. In Proceedings of Advances in neural information processing systems. 3104–3112.

39. Ian Tenney, Dipanjan Das, and Ellie Pavlick. 2019. BERT Rediscovers the Classical NLP Pipeline. In Proceedings of the 57th Conference of the Association for Computational Linguistics. 4593–4601.

40. Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In Proceedings of Advances in neural information processing systems. 5998–6008.

41. Jesse Vig and Yonatan Belinkov. 2019. Analyzing the Structure of Attention in a Transformer Language Model. In Proceedings of the 2019 ACL Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP. 63–76.

42. Jesse Vig, Ali Madani, Lav R. Varshney, Caiming Xiong, Richard Socher, and Nazneen Fatema Rajani. 2021. BERTology Meets Biology: Interpreting Attention in Protein Language Models. In Proceedings of 9th International Conference on Learning Representations.

43. Yao Wan, Yang He, Zhangqian Bi, Jianguo Zhang, Yulei Sui, Hongyu Zhang, Kazuma Hashimoto, Hai Jin, Guandong Xu, Caiming Xiong, and Philip S. Yu. 2022. NaturalCC: An Open-Source Toolkit for Code Intelligence. In Proceedings of 44th International Conference on Software Engineering, Companion Volume. ACM.

44. Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. ACM, 397–407.

45. Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based neural source code summarization. In Proceedings of 42nd International Conference on Software Engineering. ACM, 1385–1397.

46. Google, "Google Java style guide," https://google.github.io/styleguide/javaguide.html, 2017.

47. X. Li and C. Prasad, "Effectively teaching coding standards in programming," in Proceedings of the 6th Conference on Information Technology Education, 2005, pp. 239-244.

48. X. Li, "Using peer review to assess coding standards a case study," in Proceedings of the 36th Annual Conference on Frontiers in Education, 2006, pp. 9-14.

49. Y. Wang, L. Yijun, M. Collins, and P. Liu, "Process improvement of peer code review and behavior analysis of its participants," in Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education, 2008, pp. 107-111.

50.. J. Lawrance, S. Jung, and C. Wiseman, "Git on the cloud in the classroom," in Proceeding of the 44th ACM Technical Symposium on Computer Science Education, 2013, pp. 639-644.

51. C. Z. Kertsz, "Using github in the classroom a collaborative learning experience," in Proceedings of the 21st IEEE International Symposium for Design and Technology in Electronic Packaging, 2015, pp. 381-386.

52. J. Rumbaugh, I. Jacobson, and G. Booch, Unified Modeling Language Reference Manual, Pearson Higher Education, US, 2004.

53. Y. Kats and Y. Kats, Learning Management Systems and Instructional Design: Best Practices in Online Education, IGI Global, PA, 2013.

54. S. S. Nash and M. Moore, Moodle Course Design Best Practices, Packt Publishing, UK, 2014.

55. J. van Baarsene, GitLab Cookbook, Packt Publishing, UK, 2014.

56. R. Kulkarni, Java EE Development with Eclipse, Packt Publishing, UK, 2015.

57. J. Krochmalski, IntelliJ IDEA Essentials, Packt Publishing, UK, 2014.

58. G. Wielenga, Beginning NetBeans IDE: For Java Developers, Apress, NY, 2015.

59. Березький О.М., Дубчак Л.О., Мельник Г.М. Методичні рекомендації до виконання кваліфікаційної роботи з освітнього ступеня “Магістр”. Спеціальність: 123 - Комп’ютерна інженерія. Магістерська програма - Комп’ютерна інженерія". Тернопіль: ЗУНУ, 2022. 32 с.

60. Гураль І.В., Дубчак Л.О. Методичні вказівки до оформлення курсових проектів, звітів про проходження практики, випускних кваліфікаційних робіт для студентів спеціальності «Комп’ютерна інженерія» Тернопіль: ТНЕУ, 2019. 33 с.