Хмельницький національний університет

Факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

# КВАЛІФІКАЦІЙНА РОБОТА

Удосконалення методу оцінки цикломатичної складності коду для підвищення ефективності тестування програмного забезпечення

Рівень вищої освіти Другий (магістерський)

Галузь знань 12 «Інформаційні технології»

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

Шифр КвРІПЗ.180106.01.03.00.ПЗ

| | | |
|---|---|---|
| Виконав студент 2 курсу, група ІПЗм-22-1 | ________ Підпис | Олег ЛУЧИЦЬКИЙ Ім'я, ПРІЗВИЩЕ |
| Керівник канд. техн. наук, доцент (Науковий ступінь, звання) | ________ Підпис | Юрій ФОРКУН Ім'я, ПРІЗВИЩЕ |
| Нормоконтролер канд. техн. наук, доцент | ________ Підпис | Галина РАДЕЛЬЧУК Ім'я, ПРІЗВИЩЕ |

**До захисту допускаю**:
Завідувач кафедри
інженерії програмного забезпечення ________ Підпис Леонід БЕДРАТЮК Ім'я, ПРІЗВИЩЕ

____ ______________ 2023 р.

Хмельницький 2023

## ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет Інформаційних технологій

Кафедра Інженерії програмного забезпечення

Рівень вищої освіти Другий (магістерський)

Галузь знань 12 «Інформаційні технології»

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ

Завідувач кафедри ІПЗ

Л. П. Бедратюк

01.09.2023 р.

## ЗАВДАННЯ
## НА КВАЛІФІКАЦІЙНУ РОБОТУ

Лучицькому Олегу Юрійовичу

Прізвище, ім'я, по батькові здобувача

1. Тема роботи Удосконалення методу оцінки цикломатичної складності коду для підвищення ефективності тестування програмного забезпечення

Керівник роботи Форкун Юрій Вікторович, канд. техн. наук, доцент

Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 15.08.2023 р. № 30

2. Строк подання студентом роботи на кафедру 01.12.2023 р.

3. Вихідні дані до роботи Матеріали науково-дослідної практики

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)

1. Дослідження структури процесу розробки програмного забезпечення
2. Існуючі засоби оцінки складності алгоритмів програмного забезпечення
3. Метрика диференеційованої цикломатичної складності
4. Практична реалізація нової метрики

5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень)

Презентаційні матеріали (слайди)

6. Консультанти розділів кваліфікаційної роботи

| Розділ | Прізвище, ініціали та посада консультанта | Підпис, дата | |
|---|---|---|---|
| | | завдання видав | завдання прийняв |
| Антиплагіат | Форкун Ю.В., канд. техн. наук, доцент | 24.11.2023 | 11.12.2023 |
| Нормоконтроль | Радельчук Г.І., канд. техн. наук, доцент | 24.11.2023р. | 11.12.2023р. |

7. Дата видачі завдання «01» вересня 2023 р.

КАЛЕНДАРНИЙ ПЛАН

| Назва етапів (розділів) кваліфікаційної роботи | Строк виконання етапів роботи | Примітка |
|---|---|---|
| 1 Вибір напряму дослідження та узгодження тематики кваліфікаційної роботи (КвР) з керівником | 01.09-10.09.2023 | |
| 2 Підготовка розділу 1 – ознайомлення зі структурою процесу розробки програмного забезпечення, висновки до розділу | 11.09-25.09.2023 | |
| 3 Робота на розділом 2 – аналіз існуючих засобів оцінки складності програмних застосунків, висновки до розділу | 26.09-10.10.2023 | |
| 4 Робота над науковими статтями | 11.10-10.10.2023 | |
| 5 Робота на розділом 3 – розробка метрики диференційованої цикломатичної складності, висновки до розділу | 11.10-26.10.2023 | |
| 6 Робота на розділом 4 – програмна реалізація спроектованих рішень, результати експериментів та їх аналіз, висновки до розділу | 27.10-17.11.2023 | |
| 7 Попередній захист кваліфікаційної роботи | Листопад (згідно графіку) | |
| 8 Узгодження постановки задачі, отриманих результатів та висновків; написання вступу, загальних висновків, оформлення джерел посилання та додатків; оформлення пояснювальної записки згідно вимог | 18.11-30.11.2023 | |
| 9 Брошурування пояснювальної записки; підготовка супровідних документів | 01.12-04.12.2023 | |
| 10 Підготовка до захисту КвР | з 01.12.2023 р. | |

Студент ____________ Олег ЛУЧИЦЬКИЙ
Підпис Ім'я, ПРІЗВИЩЕ

Керівник роботи ____________ Юрій ФОРКУН
Підпис Ім'я, ПРІЗВИЩЕ

# РЕФЕРАТ

Тема кваліфікаційної роботи: «Удосконалення методу оцінки цикломатичної складності коду для підвищення ефективності тестування програмного забезпечення»

Автор роботи: Лучицький Олег Юрійович.

Керівник роботи: Форкун Юрій Вікторович.

Пояснювальна записка: 112 с., 12 рис., 2 дод., 27 джерел.

АНАЛІЗ ПРОГРАМНОГО КОДУ, ЦИКЛОМАТИЧНА СКЛАДНІСТЬ, МЕТРИКА ЧЕПІНА, ІНЖЕНЕРІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, ЯКІСТЬ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ.

Об'єкт дослідження – процес розробки програмного забезпечення.

Предмет дослідження – метрики складності коду програмного додатку, зокрема – метрика цикломатичної складності коду.

Метою проекту є розробка нової метрики диференційованої цикломатичної складності на основі удосконалення існуючою метрики цикломатичної складності та іншої метрики складності коду для підвищення ефективності тестування програмного забезпечення.

У роботі використані наступні методи дослідження та апаратура:

– спостереження, експеримент, абстрагування, аналіз та синтез, формалізація;

– інструментальні засоби проектування, програмування та тестування;

– персональний комп'ютер.

У процесі виконання кваліфікаційної роботи досліджено процес розробки програмного забезпечення, розглянуті існуючі методи оцінки складності алгоритмів програмного забезпечення, розглянуті їх переваги та недоліки.

Нова метрика диференційованої цикломатичної складності дозволяє отримувати точнішу інформацію щодо складності алгоритмів програмного забезпечення та нівелює недолік метрики цикломатичної складності, пов'язаний з

отриманням приблизно однакових показників при різній об'єктивній складності аналізованих програмних додатків.

Для реалізації програмного забезпечення використано мову програмування C# , платформу розробки .NET Framework, інтегроване середовище розробки Microsoft Visual Studio 2019.

Проведені емпіричні дослідження доводять адекватність та ефективність розробленої метрики диференційованої цикломатичної складності.

Апробація отриманих результатів дослідження показала більший обсяг інформації, яку отримано при обчисленні метрики, за допомогою чого можливе точніша оцінка складності програмного коду.

Нову метрику складності коду можна використовувати в процесі розробки програмного забезпечення на всіх етапах, де доречне проведення аналізу коду додаку.

Підпис — 08.12.2023 Дата

# ABSTRACT

Master’s thesis: "Improving the Method of Evaluating Cyclomatic Complexity for Enhancing Software Testing Efficiency"

Author: Oleg Luchytskyi.

Supervisor: Yuri Forkun.

Master’s thesis consists of: 112 p., 12 pc., 2 add., 27 srs.

SOURCE CODE ANALYSIS, CYCLOMATIC COMPLEXITY, CHAPIN METRIC, SOFTWARE ENGINEERING, SOFTWARE QUALITY.

The object of research is the software development process.

The subject of research is code complexity metrics, in particular, the metric of cyclomatic complexity.

The project's goal is to develop a new metric of differentiated cyclomatic complexity based on the improvement of the existing cyclomatic complexity metric and other code complexity metrics to enhance the efficiency of software testing.

The following research methods and equipment were used in the work:

- Observation, experiment, abstraction, analysis and synthesis, formalization;
- Instrumental tools for design, programming, and testing;
- Personal computer.

During the execution of the qualification work, the software development process was investigated, existing methods for assessing the complexity of software algorithm were considered, along with their advantages and disadvantages.

The new metric of differentiated cyclomatic complexity allows obtaining more accurate information about the complexity of software algorithms and eliminates the drawback of cyclomatic complexity metric, associated with obtaining approximately the same indicators for different objective complexities of analyzed software applications.

For the implementation of the software, the C# programming language, the .NET Framework development platform, and the integrated development environment Microsoft Visual Studio 2019 were used.

Empirical studies prove the adequacy and effectiveness of the developed metric of differentiated cyclomatic complexity. The validation of the obtained research results showed a greater amount of information obtained during metric calculation, allowing for a more accurate assessment of the complexity of the software code.

The new code complexity metric can be used in the software development process at all stages where the analysis of application code is relevant.

O. Луц — Signature

08.12.2023 — Date

# ЗМІСТ

Вступ ........ 8
1 Дослідження структури процесу розробки програмного забезпечення ........ 11
2 Існуючі засоби оцінки складності алгоритмів програмного забезпечення ...... 27
2.1 Часова складність ........ 28
2.2 Просторова складність ........ 30
2.3 Розмірно-орієнтовані метрики ........ 32
2.4 Метрики складності потоку керування програми ........ 37
2.5 Метрики складності потоку керування даними ........ 39
2.6 Метрики складності потоку керування даними та програми ........ 43
2.7 Об’єктно-орієнтовані метрики ........ 46
2.8 Гібридні метрики ........ 50
3 Метрика диференційованої цикломатичної складності ........ 52
3.1 Переваги та недоліки існуючих метрик ........ 53
3.2 Метрика диференційованої цикломатичної складності ........ 68
4 Практична реалізація нової метрики ........ 73
4.1 Структура та призначення модулів програми, їх взаємозв’язок ........ 73
4.2 Аналіз ти вибір технологій для реалізації програмної системи ........ 78
4.3 Розробка програмних модулів ........ 82
4.4 Вибір та обґрунтування методів тестування додатку ........ 87
4.5 Оцінка методу диференційованої цикломатичної складності ........ 89
4.6 Інтеграція та налаштування програмного засобу ........ 91
Висновки ........ 92
Перелік джерел посилання ........ 94
Додаток А Копії наукових публікацій ........ 97
Додаток Б Презентаційні матеріали ........ 104

# ВСТУП

Поняття інженерії програмного забезпечення (англ. "Software engineering") існує протягом понад півстоліття. Ця галузь визначає розвиток програмного забезпечення і включає весь цикл його створення: від початкового аналізу вимог і проектування до тестування, впровадження та підтримки. Зараз програмне забезпечення використовується в практично всіх сферах життя завдяки розвитку інженерії програмного забезпечення та наукових досліджень в цій галузі.

Однією з ключових проблем, яку вивчає інженерія програмного забезпечення, є якість розроблюваного програмного забезпечення. Якість програмного коду грає важливу роль в цьому процесі. Розробники стикаються з численними питаннями: "Як забезпечити надійність коду?", "Як досягти продуктивності та ефективності?", "Як здійснювати підтримку та розвиток програми з плином часу?", "Як гарантувати коректну та безпечну роботу програми?".

Аналіз вихідного коду є важливою частиною вирішення цих питань, дозволяючи ретельно вивчати кожен рядок програмного коду, визначати його характеристики та виявляти можливі проблеми. Якість та надійність програми залежать від якості аналізу вихідного коду. Сучасні програми стають складнішими, вимагаючи постійних оновлень і підтримки, тому аналіз вихідного коду стає обов'язковим етапом розробки. Зростання обсягу коду, різноманітні технології та бібліотеки, а також високі вимоги до продуктивності та безпеки роблять аналіз вихідного коду надзвичайно важливим. Для цього використовуються різні метрики складності коду, які надають об'єктивну інформацію про якість програмного забезпечення.

Актуальність роботи пов’язана з тим, що аналіз вихідного коду комп’ютерних програм є одним з основних елементів процесу розробки програмного забезпечення. Проведення аналізу є доречним на більшості етапів процесу розробки, оскільки отримані відомості про структуру коду дозволяє визначити помилки, вразливості та можливості для оптимізації алгоритмів. А

також тим, що метрика цикломатичної складності досі є одним з основних засобів аналізу коду за рахунок зрозумілості та простоти використання. Тому спроби удосконалити цю метрику залишаються актуальним й на сьогоднішній день.

Об'єкт дослідження – процес розробки програмного забезпечення.

Предмет дослідження – метрики аналізу складності коду програмного додатку, зокрема – метрика цикломатичної складності коду.

Метою проекту є розробка нової метрики диференційованої цикломатичної складності на основі удосконалення існуючою метрики цикломатичної складності та іншої метрики складності коду для підвищення ефективності тестування програмного забезпечення.

Тому завданням магістерської роботи є:

– дослідити предметну область – розробка програмного забезпечення;

– проаналізувати існуючі дослідження в області використання метрик складності коду;

– розробити теоретичні основи для впровадження нової техніки;

– визначити структуру програмного додатку для реалізації нової метрики;

– вибрати технології для подальшої розробки програмного додатку;

– розробити та протестувати програмне забезпечення на відповідність вимогам та наявність недоліків.

Для досягнення мети використано теоретичні та емпіричні та методи дослідження, а саме:

а) теоретичні методи:

– абстрагування – один з важливих методів, який дозволяє відкинути несуттєві параметри (від абстрагування напряму залежить ефективність моделі);

– аналіз та синтез – декомпозиція моделі на прості складові, виявлення зв'язків між компонентами і, відповідно, синтез цих структурних елементів у єдине ціле;

– формалізація – представлення моделі у вигляді програмного коду;

б) емпіричні методи:

– спостереження (темою роботи є створення метрики диференційованої цикломатичної складності на основі існуючих алгоритмів, тому слід провести спостереження існуючих рішень, визначити властивості та зв’язки між ними);

– експеримент (метод необхідний на етапі перевірки ефективності нової метрики).

Наукова новизна отриманих результатів:

– отримала подальший розвиток метрика цикломатичної складності у напрямку відображення складності алгоритмів програмного додатку більш об’єктивно;

– розроблено метрику диференційованої цикломатичної складності, яка за рахунок поєднання алгоритмів оцінки цикломатичної складності та метрики Чепіна, дозволяє точніше визначати складність алгоритмів програмного додатку.

Практична цінність отриманих результатів полягає у вдосконаленні метрики цикломатичної складності, що дозволяє ефективніше оцінювати складність алгоритмів. Удосконалена метрика може бути використана при роботі з будь-яким програмним кодом на будь-якому етапі життєвого циклу цільового програмного забезпечення.

Достовірність та обґрунтованість отриманих результатів підтверджується використанням у процесі дослідження таких прийомів:

– перевірка нових та удосконалених рішень експериментальними дослідженнями;

– наявність наукової публікації у рецензованому виданні.

За результатами дослідження опублікована одна стаття у фаховому науковому виданні.

# 1 ДОСЛІДЖЕННЯ СТРУКТУРИ ПРОЦЕСУ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Метрики складності коду є елементом процесу аналізу коду програмних застосунків, який охоплює значну частину процесу розробки програмного забезпечення. Тому необхідно розглянути структуру цього процесу. Розробка програмного забезпечення представляє собою ключовий аспект в інженерії програмного забезпечення, яка є важливою галуззю науки та професійної діяльності. Ця область досліджує методи та процеси розробки, проектування, створення, тестування, впровадження та підтримки програмного забезпечення. Основна мета інженерії програмного забезпечення полягає в створенні високоякісних програмних продуктів, які відповідають вимогам користувачів та бізнесу, а також є надійними та ефективними[1-3].

Однією з ключових концепцій в інженерії програмного забезпечення є життєвий цикл програмного забезпечення, пов'язаний із процесом розробки. Життєвий цикл програмного забезпечення – це послідовність чітко визначених етапів, через які програмний застосунок проходить під час свого створення та впровадження[4-12].

Кожен етап включає низку процесів, кожен з яких призводить до конкретних результатів, використовуючи необхідні ресурси. Стандарт ISO/IEC 12207 визначає структуру життєвого циклу, охоплюючи процеси, дії та завдання, обов'язкові під час розробки програмного забезпечення.

Згідно зі стандартом, програмне забезпечення розглядається як набір комп'ютерних програм, процедур і супутньої документації та даних. Процес визначається як послідовність взаємопов'язаних дій, які перетворюють вхідні дані в вихідні. Процес розкладається на дії, а дії – на набір завдань. Ці елементи можуть бути ініційовані та виконані в будь-якому порядку, забезпечуючи гнучкість у виконанні.

Усі продукти інженерії програмного забезпечення включають описи, такі як вимоги до розробки, угода, тексти програм, інструкції з експлуатації тощо. Ключові ресурси цієї інженерії – час і вартість, які визначають ефективність розробки.

Згідно із стандартом ISO/IEC 12207 всі процеси життєвого циклу програмного забезпечення розділяються на три групи:

– основні процеси (придбання, поставка, розробка, експлуатація, підтримка);

– організаційні процеси (управління, поліпшення, навчання);

– допоміжні процеси (документація, забезпечення якості, верифікація, атестація, аудит, загальна оцінка і багато інших).

До основних процесів відносяться наступні:

– етап придбання, який запускає життєвий цикл системи і визначає організацію-покупця автоматизованої системи, програмного продукту або сервісу;

– етап розробки, який охоплює дії організації - розробника програмного продукту;

– етап постачання, який включає дії під час передачі розробленого продукту замовнику;

– етап експлуатації, який означає обслуговування системи під час її використання – консультування користувачів, вивчення їхніх побажань і т.д.;

– етап супроводження, який включає дії з управління модифікаціями, підтримки актуального стану та функціональної придатності, інсталяції та вилучення версій програмного забезпечення у користувача.

Процес розробки програмного забезпечення повинен забезпечити шлях від розуміння потреби замовника до передачі йому готового продукту. Він включає такі етапи:

– визначення вимог;

– проектування;

– реалізація;

– документація;

– тестування;

– експлуатація та супроводження готової системи.

На етапі визначення вимог головною метою є конкретизація докладних вимог до системи. Крім того, важливо переконатися, що всі учасники правильно зрозуміли поставлені завдання і як саме кожна вимога буде втілена на практиці. Часто у обговоренні також беруть участь фахівці з тестування, які вже на етапі розробки вимог можуть внести свої побажання і, за необхідності, виправити процес. Залежно від обраної моделі розробки, підходи до визначення моменту переходу від одного етапу до іншого можуть відрізнятися. Наприклад, у каскадній або аналізу вимог фіксується в документі – специфікації вимог до програмного забезпечення (Software Requirement Specification, SRS), оформлення якого повинно бути завершено до переходу на наступний етап.

Проектування інформаційних систем виконується командою програмістів та системних архітекторів, які керуються вимогами та розробляють високорівневий дизайн системи. У процесі проектування розв'язуються різноманітні технічні питання, які обговорюються з усіма зацікавленими сторонами, включаючи замовника. Визначаються технології проекту, завантаженість команди, обмеження, часові рамки та бюджет. З уточненими вимогами обираються найбільш відповідні проектні рішення. Схвалений дизайн системи включає перелік програмних компонентів, які необхідно розробити, взаємодію з третіми сторонами, функціональні характеристики програми, бази даних, що будуть використовуватися та інші деталі. Зазвичай, дизайн фіксується окремим документом, відомим як дизайн-специфікація (DSD). На цьому етапі спрощення процесу візуалізації застосовуються різні нотації, такі як блок-схеми, ER-діаграми, UML-діаграми та макети, наприклад, намальований в графічному редакторі прототип сайту.

Під реалізацією розуміється процес розробки та програмування розпочинається після узгодження вимог і дизайну продукту. На цьому етапі відбувається перехід до наступного етапу життєвого циклу – безпосередньої розробки. Команда програмістів розпочинає написання коду програми відповідно до раніше визначених вимог. Системні адміністратори конфігурують програмне

оточення, front-end програмісти створюють користувальницький інтерфейс програми та розробляють логіку взаємодії з сервером. Паралельно програмісти створюють Unit-тести для перевірки правильності роботи коду кожного компонента системи, проводять рев'ю написаного коду, а також створюють білди та розгортають готове програмне забезпечення в програмному середовищі. Цей цикл повторюється до моменту повної реалізації всіх вимог. Процес програмування включає чотири основні стадії: розробка алгоритмів, написання вихідного коду, компіляція, тестування та відладка.

Розвиток документації визначається умовно, оскільки різноманітні документи генеруються на кожному етапі життєвого циклу програмного продукту. Окрім проектної документації та записів, що відносяться до розробки, існують інші текстові матеріали, що деталізують функціонал програми та її використання. В загальному розгляді, існують чотири рівні документації: Архітектурний (проектний) рівень, такий як дизайн-специфікація. Ці документи містять відомості про моделі, методології, інструменти та ресурси розробки, вибрані для конкретного проекту. Технічний рівень охоплює всю документацію, яка супроводжує розробку. Сюди включаються різноманітні матеріали, що пояснюють розробку системи на рівні окремих модулів. Зазвичай це представлено у вигляді коментарів до вихідного коду, які потім структуруються у формі HTML-документів. Користувацький рівень включає в себе довідкові та пояснюючі матеріали, необхідні кінцевому користувачеві для роботи з системою. Це може бути, наприклад, Readme та Userguide, або розділ довідки по програмі. Маркетинговий рівень забезпечує рекламні матеріали, що висвітлюють випуск продукту. Основною метою є яскраве представлення функціоналу та конкурентних переваг продукту у привабливій формі.

Перевірка кожного з модулів та способів їх інтеграції; тестування програмного продукту в цілому (так звана верифікація); тестування відповідності функцій працюючої програмної системи вимогам, що були до неї поставлені замовником (так звана валідація).

Завершивши тестування програми та виправивши всі серйозні дефекти, надходить момент випуску і початку передачі її кінцевим користувачам. Після введення нової версії програми в експлуатацію, вступає в дію відділ технічної підтримки. Його фахівці взаємодіють із користувачами, надаючи консультації та технічну підтримку. У випадку виявлення користувачами будь-яких помітних помилок після випуску, інформація про них передається у вигляді звітів про недоліки розробницькій команді. Згідно з серйозністю проблеми, команда видає виправлення (найчастіше, "гарячий" фікс, відомий як hot-fix), або відкладає його до майбутніх версій програмного продукту. Крім того, спеціалісти технічної підтримки допомагають збирати та аналізувати різноманітні метрики, що вказують на продуктивність програми в реальних умовах.

Підготовча фаза розпочинається із відбору моделі для життєвого циклу програмного забезпечення, яка належно враховує масштаб, значимість і складність конкретного проекту. Важливо, щоб процес розроблення відповідав обраній моделі. Розробник повинен вибрати відповідні стандарти, методи і засоби розроблення, а також адаптувати їх до конкретних умов проекту, забезпечуючи згідність з вимогами замовника. Крім того, необхідно скласти детальний план виконання робіт.

Існує різноманіття реалізацій життєвого циклу програмного забезпечення, кожна з яких акцентує увагу на певних аспектах розробки та призводить до відмінних результатів. Найбільш поширеними є:

- каскадна (водоспадна) модель;
- ітеративні і інкрементально-еволюційні моделі;
- спіральна модель (відома як модель Боема);

В каскадній моделі перехід від одної фази проекту до іншої передбачає повну правильність результату попередньої фази. Однак навіть невеликі невірності у вимогах чи некоректне їх тлумачення можуть призвести до необхідності повертатися до попередньої фази проекту. Це не тільки порушує графік, але і може спричинити істотне зростання витрат, а навіть призвести до перегляду концепції проекту. Крім того, ця модель не завжди може гарантувати достатню швидкість

реагування на швидкозмінні потреби користувачів, для яких програмна система є важливим інструментом у сфері бізнесу. Існує багато прикладів проблем, що виникають з самої природи цієї моделі, які можуть послужити вагомими аргументами відмови від каскадного підходу до життєвого циклу. До переваг каскадної моделі можна віднести:

– стабільність вимог протягом всього життєвого циклу розробки;

– можливість послідовного вирішення виникаючих труднощів;

– чіткість та зрозумілість кроків моделі, що спрощує її використання;

– покращена можливість планування, контролю та управління проектом;

– доступність для розуміння замовниками;

– ефективність для проектів із чіткими, але складними вимогами;

– ефективність для проектів із високими вимогами до якості, якщо відсутні обмеження витрат і графіку.

Недоліки каскадної моделі життєвого циклу включають:

– складність чіткого формулювання вимог на початку життєвого циклу та неможливість їх динамічної зміни;

– послідовність лінійної структури процесу розробки, яка може призвести до повернення на попередні кроки та збільшення витрат;

– непридатність проміжного продукту для використання;

– неможливість гнучкого моделювання систем, що не мають аналогів;

– пізнє виявлення проблем через інтеграцію всіх результатів в кінці розробки;

– обмежена участь користувача у створенні системи;

– неможливість попередньої оцінки якості системи користувачем;

– проблеми із фінансуванням проекту через складність розподілу великих коштів.

Каскадна модель має свої обмеження, які визначають її придатність. Ефективне використання цієї моделі рекомендується у таких випадках:

– розробка проектів з чіткими вимогами, які залишаються сталими протягом усього циклу життя проекту. Ці вимоги повинні бути ясними, із задокументованою реалізацією та технічними методиками;

– створення проекту, орієнтованого на побудову системи або продукту, аналогічного тим, які розроблялися раніше розробниками;

– розробка проекту, пов'язаного з випуском нової версії існуючого продукту або системи;

– перенесення існуючого продукту на нову платформу – це також область, де каскадна модель може бути використана;

– використання при великих проектах, де декілька обширних розробницьких команд задіяно одночасно.

Зазначені критерії визначають сфери застосування каскадної моделі, де вона проявляє найбільшу ефективність.

Кожен етап представлений блоком, і процес рухається вниз по блоках, і кожен новий етап починається після завершення попереднього. Графічне представлення може виглядати як сходинки (кожен етап нижче попереднього), драбини, або послідовність фаз, що ілюструє послідовність виконання.

Також, важливо врахувати, що каскадна модель не передбачає повернень назад, тобто перехід до попереднього етапу, що робить її лінійною. Це відрізняється від ітеративних або еволюційних моделей, де можливі повторні ітерації чи коригування на підставі результатів тестування та відгуків.

Приклад графічного представлення моделі подано на Рисунку 1.1.

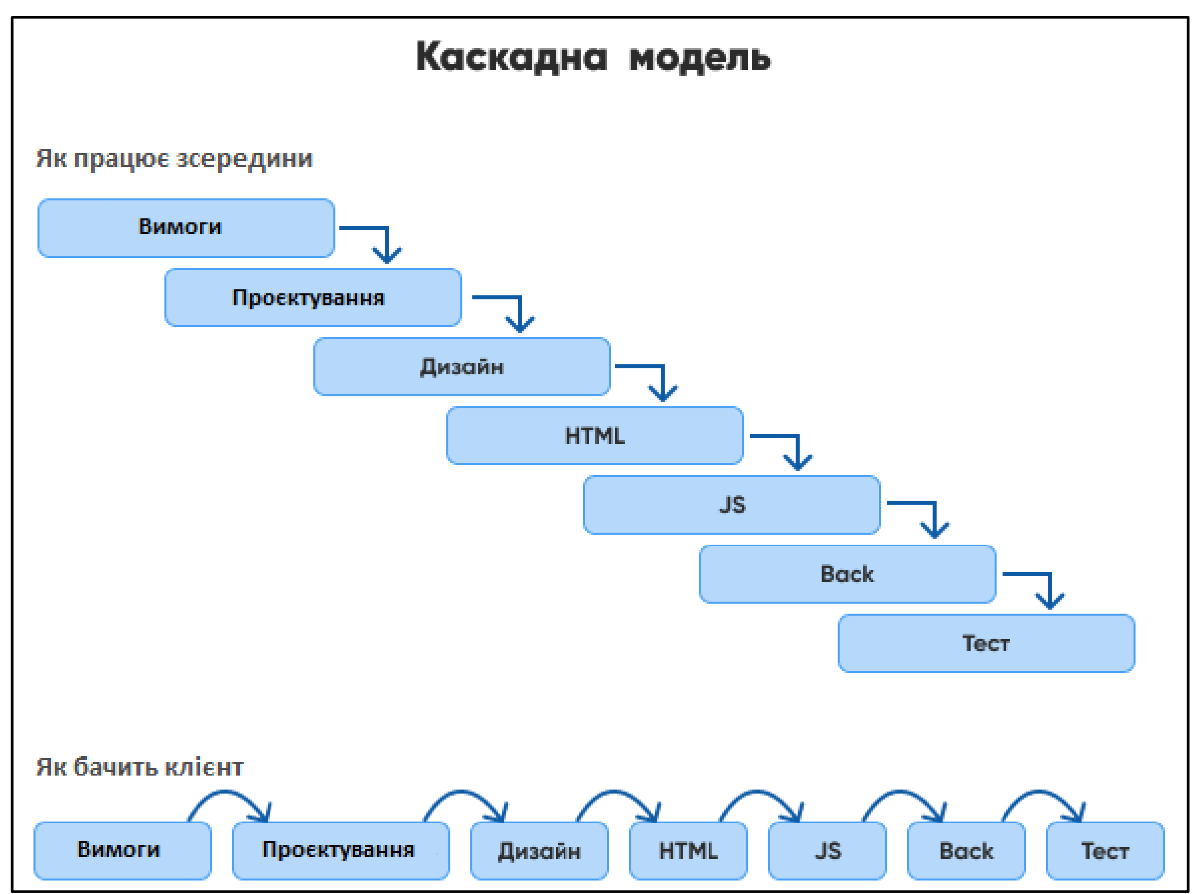



Рисунок 1.1 – Графічне представлення прикладу каскадної моделі

Ітеративна модель враховує розбиття життєвого циклу проекту на послідовні ітерації, кожна з яких є своєрідним "міні-проектом", охоплюючи всі етапи життєвого циклу, але застосовуючи їх для створення менших фрагментів функціональності, порівняно з проектом в цілому.

Головна мета кожної ітерації – отримати працюючу версію програмної системи, яка включає функціональність, визначену в інтегрованому контексті попередніх і поточних ітерацій. Результат фінальної ітерації містить всю необхідну функціональність продукту. Отже, кожна ітерація призводить до інкрементального розвитку продукту.

Структурно цю модель можна назвати ітеративною, а з точки зору розвитку продукту – інкрементальною. Справжній досвід показує, що ці підходи не можна розглядати ізольовано. Такий комбінований еволюційний підхід часто називають просто ітеративним, коли йдеться про процес, або інкрементальним, коли йдеться про нарощування функціональності продукту.

Еволюційна модель передбачає не тільки створення працюючої версії системи, але й її впровадження в реальні умови з подальшим аналізом відгуків користувачів для визначення напрямку та планування наступної ітерації. Відмінною особливістю є те, що "чистий" інкрементальний підхід передбачає відсутність проміжних релізів, і всі ітерації проводяться згідно з попередньо визначеним планом нарощування функціональності. Таким чином, замовник отримує результати лише після завершення фінальної ітерації як повноцінну версію системи.

Значущість еволюційного підходу на основі ітерацій виявляється у зменшенні невизначеності на кожному етапі. Це, в свою чергу, дозволяє зменшити ризики. Рисунок 1.2 ілюструє деякі концепції еволюційного підходу, показуючи, що ітеративне розбиття може стосуватися не тільки загального життєвого циклу, включаючи фази – визначення вимог, проектування, конструювання і т.д., але й розгалуження кожної фази на додаткові ітерації, пов'язані, наприклад, з деталізацією структури декомпозиції проекту, такої як архітектура модулів системи.

Основні переваги ітеративної моделі розробки

– зниження ризиків – вчасне виявлення конфліктів між вимогами, моделями та реалізацією проекту; глибоке фокусування на основних завданнях; постійне формування та управління вимогами;

– ефективний зворотний зв'язок від проектної команди зі споживачем, що дозволяє створити продукт, ідеально відповідний його потребам;

– швидкий випуск мінімально цінного продукту (mvp) та можливість виведення продукту на ринок раніше ніж завершення всього проекту.

Основні недоліки ітеративної моделі розробки:

– проблеми з архітектурою та зайві витрати – при роботі з хаотичними вимогами без опрацьованого глобального плану, архітектура додатка може страждати, адже для її виправлення можуть знадобитися додаткові ресурси;

– відсутність фіксованого бюджету та термінів, а також необхідна активна участь замовника в процесі – для деяких замовників ці умови є неприйнятними, і вони віддають перевагу моделі "водоспад".

Цю модель рекомендовано застосовувати для систем, де ключовими є функціональні можливості і де необхідно швидко продемонструвати їх за допомогою CASE-засобів.

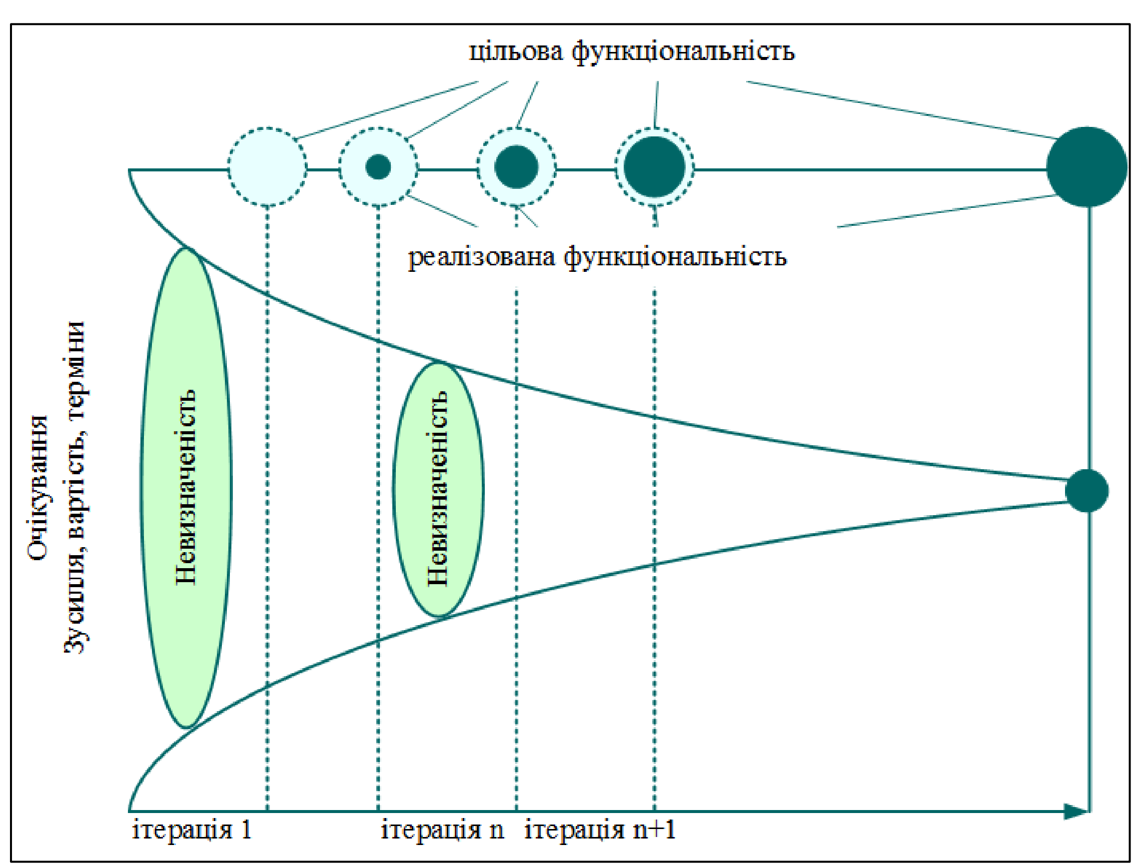



Рисунок 1.2 – Зниження невизначеності та інкрементальне розширення функціональності при ітеративній організації життєвого циклу

На графічному зображенні ітераційної моделі життєвого циклу програмного забезпечення можна спостерігати циклічність та поетапність процесу розробки. Зазвичай таке зображення представляє собою кілька повторюваних блоків або кіл, кожен з яких відображає окремий етап ітерації (рисунок 1.3).

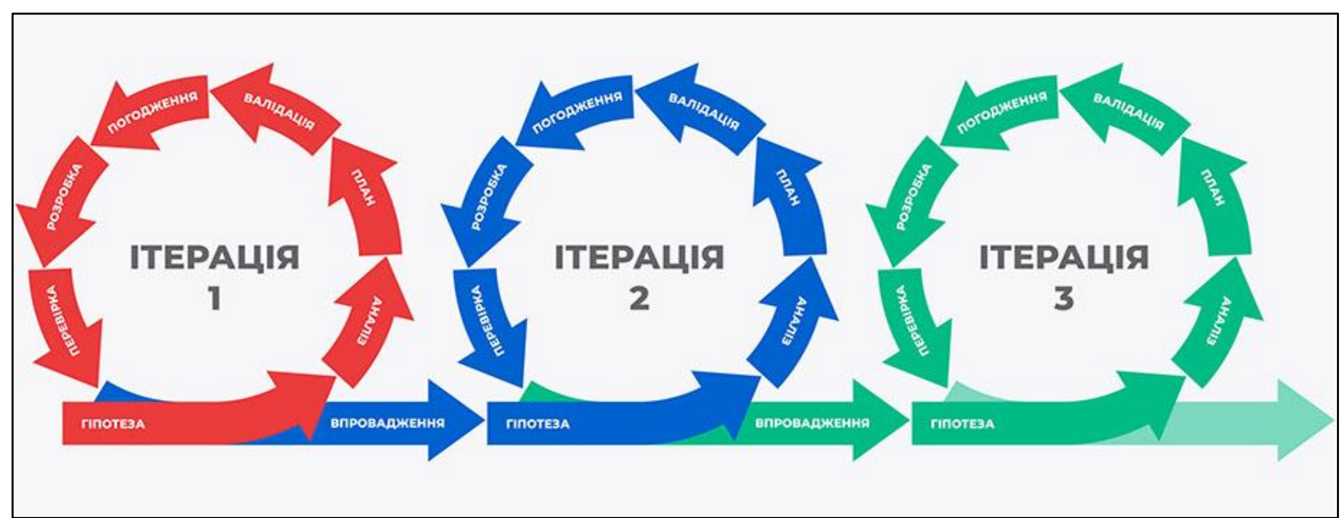



Рисунок 1.3 – Графічне представлення прикладу ітераційної моделі

Спіральна модель розробки програмного забезпечення вперше визначена Баррі Боєм у 1988 році. Однією з ключових особливостей цієї моделі є особлива увага до ризиків, що впливає на організацію життєвого циклу.

Головний внесок спіральної моделі полягає у можливості адаптації вдалих аспектів існуючих моделей процесів життєвого циклу.

Ця модель часто застосовується при розробці новаторських (нетипових) систем. Початково у замовника та розробника немає чіткого уявлення про кінцевий продукт (вимоги не можуть бути чітко визначені), або їх впевненість у успішному завершенні проекту є низькою (ризики значні). У зв'язку з цим приймається рішення розробляти систему частинами з можливістю зміни вимог чи відмови від подальшого розвитку.

Переваги моделі:

– швидше демонструє користувачам працездатний продукт, що активізує процес уточнення та доповнення вимог;

– дозволяє зміну вимог у процесі розробки інформаційної системи, що характерно для більшості розробок, включаючи типові;

– забезпечує більшу гнучкість в управлінні проектом;

– допомагає створити надійну та стійку систему, оскільки помилки виявляються та виправляються на кожній ітерації;

– удосконалює процес розробки через аналіз, проведений на кожній ітерації, що дозволяє вдосконалювати організацію розробки.

Недоліки моделі:

– збільшується невизначеність у розробника щодо перспектив розвитку проекту;

– ускладнюються операції тимчасового і ресурсного планування всього проекту в цілому. Для вирішення цього можливо вводити тимчасові обмеження на кожну стадію життєвого циклу, навіть якщо не вся запланована робота виконана. план складається на основі статистичних даних, отриманих з попередніх проектів та особистого досвіду розробників.

На рисунку 1.4 схематично зображена спіральна модель. Окрім розділення на етапи на моделі процес розробки розділений на чотири великих етапи.

На етапі визначення цілей, альтернативних варіантів та обмежень проводиться аналіз та формулювання цілей, можливих альтернатив реалізації, і обмежень. Встановлюються робочі характеристики, функціональні вимоги, фактори успіху та параметри інтерфейсу. Альтернативні шляхи включають конструювання, повторне використання, чи покупку рішень. Описуються обмеження, такі як витрати, графік, інтерфейс та інші середовищні обмеження. Розробляється документація, що фіксує ризики, пов'язані з нестачею досвіду, використанням нових технологій та інші аспекти.

На етапі оцінки альтернативних варіантів, ідентифікації та вирішення ризиків проводиться оцінка альтернатив в контексті визначених цілей та обмежень. Здійснюється ідентифікація та управління ризиками, включаючи стратегії управління ризиками, методи економічно обґрунтованого вибору та оцінку фінансових ризиків. Приймаються рішення про припинення або продовження розробки проекту в залежності від отриманих результатів;

На етапі розробки продукту наступного рівня проводяться дії, такі як створення проекту, програмування, кодування, тестування та компонування продукту. Перша версія продукту показується замовнику, і після отримання відгуку

розпочинається фаза планування, спрямована на врахування реакцій клієнта. Кожна наступна версія точніше відображає вимоги замовника;

На етапі планування наступної фази розроблюється план проекту, менеджменту конфігурацій, тестування та установки програмного продукту для підготовки до наступної фази розробки..

Для кожної ітерації потрібно виділити цілі, альтернативні варіанти та обмеження, встановити та вирішити ризики; дати оцінку альтернативним варіантам; розробити дані для даної ітерації та підтвердити їх правильність; спланувати наступну ітерацію. Потім необхідно вибрати метод здійснення наступної ітерації у випадку її необхідності.

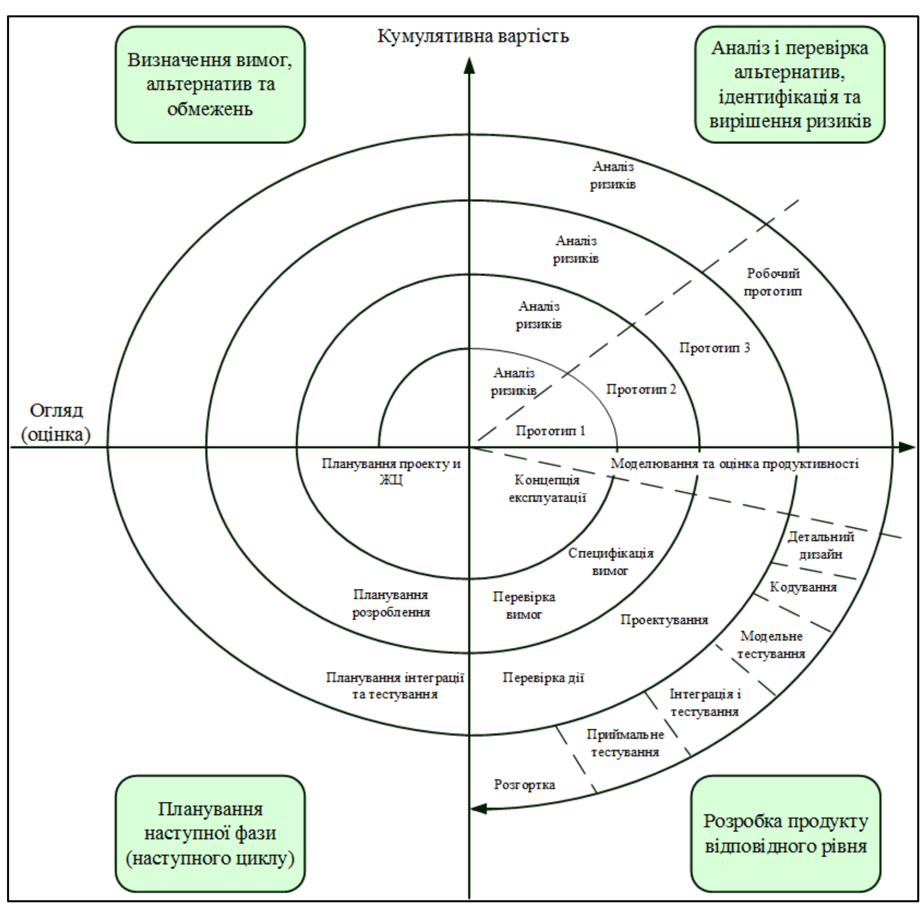



Рисунок 1.4 – Графічне представлення прикладу спіральної моделі

Також існує ряд новіших принципів розробки програмного забезпечення, таких, як Agile, Lean та інші.

Agile – це набір принципів гнучкої розробки, який складається з 12 ідей. Основні пункти Agile визначають, що люди та їх взаємодія є важливішими за процеси та інструменти, працюючий продукт має більше значення, ніж вичерпна документація, і співпраця з клієнтом переважає узгодження умов контракту. Також, готовність до змін має важливість перед проходженням попереднім планом.

Принцип взаємодії підкреслює важливість обміну досвідом між командою та клієнтом. Це дозволяє кожній стороні впливати на процес прийняття рішень, знижує ризики втрати часу та грошей і підвищує здатність команди вирішувати складні завдання з високим рівнем невизначеності.

Але необхідно розуміти обмеження, оскільки взаємодія всіх з усіма може призвести до хаосу. Таким чином, використовуючи Agile, важливо дотримуватися обмежень: невеликі команди, компетентні та мотивовані учасники, короткі ітерації з чіткими цілями, обмеження за часом та очевидні кінцеві результати.

Agile добре використовується з невизначеністю, скорочуючи період прогнозування. Чим вища невизначеність, тим коротша ітерація, можливо, навіть до 24 годин. На кожному етапі ітерації проводиться контроль, ретроспективний аналіз, оцінка та планування наступної ітерації.

У внутрішньому плануванні та розробці продукту Agile виявляється необхідним принципом для уникнення хаосу.

Основна ідея Lean підходу полягає у раціональному використанні ресурсів, включаючи час, та вирішенні завдань найбільш простим способом. Наприклад, замість того, щоб створювати весь продукт, експериментуючи з лендінгом і формою підписки, ми розміщуємо рекламу, щоб визначити зацікавленість клієнтів та прийняти усвідомлене рішення щодо подальшого розвитку.

У випадку розробки продукту чи внесення поліпшень, будь то виробничі чи інженерні, ми спочатку створюємо його MVP (мінімально життєздатний продукт). Термін MVP широко використовується, але виник саме завдяки Lean підходу. MVP – це версія продукту, яка виконує основну функцію і залишається прийнятною для клієнтів, визнаючи її корисність. Цей продукт можна поступово вдосконалювати,

але він повинен бути корисним і зрозумілим на етапі MVP, дозволяючи прийняти рішення щодо подальших вдосконалень чи визнати експеримент неуспішним.

Оголошений Lean підхід базується на тестуванні потреб та цінностей з мінімальними ресурсами. Він не обмежується вирішенням технічних проблем, але передбачає підприємницький підхід до завдань. Усе спрощується, щоб знайти найефективніше рішення: технічно, організаційно тощо, уникати зайвого складного.

Проте, важливо усвідомити ризики Lean підходу, особливо в прагненні до повної спрощеності, що може призвести до втрати важливих функцій і зробити продукт малоцінним для користувача.

Загалом, Lean – це не лише відмова від послуги на користь лендінгу. Це широкий спектр методик, які варто детально вивчити, бо вони пропонують багато інших можливостей та інструментів.

Таким чином у першому розділі була розглянута структура процесу розробки програмного забезпечення. Це процес створення програмних продуктів, які включають в себе всі аспекти життєвого циклу програми, від концепції і розробки до тестування, удосконалення та підтримки. Процес складається з певного набору етапів, виконання яких визначається певним підходом, вибір якого з великого числа варіантів визначається характером проекту.

Не дивлячись на все це різноманіття, для розробки програмного забезпечення в будь-якій формі важливою складовою є процес аналізу коду. А одним з найпоширеніших засобів аналізу є метрика цикломатичної складності, оскільки вона досить простою і доступною для використання.

Однак, ця метрика має серйозний недолік. Її показник враховує шляхи виконання алгоритму, але не складність операцій в цьому алгоритмі, через що може виникнути ситуація, коли два алгоритми с однаковим показником цієї метрики можуть насправді мати різну складність.

Тому завданням магістерської роботи постає:

– аналіз існуючих метрик та їх відбір для розгляду можливості використання їх методик для вдосконалення метрики цикломатичної складності:

– розробити метрику диференційованої цикломатичної складності на основі поєднання методик двох метрик та подолання головного недоліку цикломатичної складності.

Актуальність роботи пов’язана з тим, що аналіз вихідного коду комп’ютерних програм є одним з основних елементів процесу розробки програмного забезпечення. Проведення аналізу є доречним на більшості етапів процесу розробки, оскільки отримані відомості про структуру коду дозволяє визначити помилки, вразливості та можливості для оптимізації алгоритмів. А також тим, що метрика цикломатичної складності досі є одним з основних засобів аналізу коду за рахунок зрозумілості та простоти використання. Тому спроби удосконалити цю метрику залишаються актуальним й на сьогоднішній день.

Об’єкт дослідження – процес розробки програмного забезпечення.

Предмет дослідження – метрики аналізу складності коду програмного додатку, зокрема – метрика цикломатичної складності коду.

Метою проекту є розробка нової метрики диференційованої цикломатичної складності на основі удосконалення існуючою метрики цикломатичної складності та іншої метрики складності коду для підвищення ефективності тестування програмного забезпечення.

Тому завданням магістерської роботи є:

– дослідити предметну область – розробка програмного забезпечення;

– проаналізувати існуючі дослідження в області використання метрик складності коду;

– розробити теоретичні основи для впровадження нової техніки;

– визначити структуру програмного додатку для реалізації нової метрики;

– вибрати технології для подальшої розробки програмного додатку;

– розробити та протестувати програмне забезпечення на відповідність вимогам та наявність недоліків.

# 2 ІСНУЮЧІ ЗАСОБИ ОЦІНКИ СКЛАДНОСТІ АЛГОРИТМІВ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Оцінка якості програмного продукту є ключовим етапом в розробці програмного забезпечення. Цей процес визначає ефективність та продуктивність програми або системи. Оцінка включає ряд аспектів, які визначають її загальну якість та функціональність[13]:

– функціональність: перевірка того, чи виконує програма всі функції, які зазначені у специфікаціях та очікуваннях користувачів;

– надійність: оцінка того, наскільки надійно програма працює, включаючи виявлення та виправлення помилок та вразливостей;

– продуктивність: визначення швидкості та ефективності виконання функцій програми, включаючи аналіз продуктивності коду та ресурсів;

– складність коду: вимірювання та оцінка складності програмного коду, використовуючи метрики, такі як кількість операторів та цикломатична складність;

– зручність використання: оцінка легкості взаємодії користувачів з програмою;

– забезпечення безпеки: аналіз вразливостей, атак та інших аспектів забезпечення безпеки програми;

– документація та коментарі: оцінка наявності та якості супровідної документації та коментарів у вихідному коді;

– портабельність: оцінка можливості використання програми на різних платформах.

Зосередження на складності коду є особливо важливим залежно від тематики проекту. Сучасні підходи включають різні методи вимірювання складності алгоритмів, такі як показники часової та просторової складності [14-17].

## 2.1 Часова складність

Оцінка часової складності алгоритму є одним із основних типів обчислювальної складності та описує час, необхідний для виконання певного алгоритму.

Щоб оцінити часову складність алгоритму, припускається, що кожна окрема операція, яка виконується в алгоритмі, займає константний проміжок часу під час виконання, а потім підраховується кількість всіх таких елементарних функцій, що необхідні для вирішення задачі. Під час виконання такого розрахунку сталий час для однієї елементарної операції позначається як Big-O notation та оцінюється як $O(1)$.

Оскільки час роботи одного алгоритму може бути різним при його реалізації на різних поколіннях обчислювальної техніки, то зазвичай для обчислень часової складності враховують час роботи для гіршого випадку, який позначається як $T(n)$. Тобто, є найбільшим значенням часу роботи алгоритму для всіх вхідних даних розміру $n$. Лише в деяких випадках розраховується та використовується середнє значення часу роботи алгоритму, що є математичним очікуванням часу роботи для всіх можливих вхідних даних.

Алгоритм демонструє сталий час, позначений як $O(1)$, коли кількість базових операцій і час, необхідний для їх виконання, є відомими та незмінними. У цьому випадку час виконання $T(n)$ не залежить від розміру вхідних даних.

Наприклад, розглянемо алгоритм вибору одного елемента з масиву даних. Тут виконується лише одна операція, яка потребує фіксованого проміжку часу.

Інший приклад константного часу - це завдання "обміняти значення $a$ і $b$, якщо потрібно, щоб результат був $a \leq b$". Хоча цей алгоритм має властивість константного часу, важливо зауважити, що це не пов'язано з виконанням вказаної нерівності. У будь-якому випадку існує константа $t$, для якої час виконання завжди перевищує $t$.

Знову один із варіантів часу, який виділяється при класифікації, – це логарифмічний. Алгоритм вважається логарифмічним, якщо його час виконання $T(n) = O(\log n)$. Використання логарифму з основою 2 ($\log_2 n$) обумовлене тим, що комп'ютер використовує двійкову систему числення. Незважаючи на це, у записі як О-велике при різних основах логарифму, наприклад, $\log_a n$ і $\log_b n$, вони відрізняються лише постійним множником $\log_a b$. Таким чином, $O(\log n)$ є стандартним записом для алгоритмів логарифмічного часу.

Використання алгоритмів логарифмічного часу є найбільш ефективним і поширеним при вирішенні завдань з двійковими деревами чи використанні двійкового пошуку. Також вони виявляються ефективними при операціях з масивами даних розміру n. У випадку останніх особливо важливо, що співвідношення між часом виконання однієї операції та розміром масиву зменшується із збільшенням цього розміру.

Вивчення апаратної та програмної реалізації алгоритмів дозволяє класифікувати різновиди часу та розробляти алгоритми з лінійним або навіть більш ефективним часом виконання. Лінійний час виявляється важливим для виконання алгоритму та вирішення конкретних завдань. З точки зору математики та стандартних моделей обчислень існує невелика ймовірність досягнення лінійного часу виконання для деяких алгоритмів, але в той самий час можливе їх використання в лінійний час. Це стає реальністю завдяки паралельному використанню потоків виконання алгоритму та вхідних даних.

Характеристика часу роботи алгоритму визначається лінійна, коли його складність дорівнює $O(n)$. По суті, при великому обсязі вхідних даних час виконання алгоритму збільшується пропорційно цьому обсягу. Наприклад, алгоритм, що обчислює суму всіх елементів списку, працює в часовому проміжку, пропорційному довжині списку. Проте у будь-якому випадку час роботи може не бути точно пропорційним, особливо якщо значення n виявляється невеликим.

Алгоритми, які працюють у межах поліноміального часу, характеризуються верхньою межею часу виконання, яка обмежена многочленом від розміру вхідних даних, а саме $T(n)=O(n^k)$ для певної константи k. Серед прикладів поліноміальних

алгоритмів можна вказати алгоритм швидкого сортування n цілих чисел (з максимумом в $An^2$ операцій для деякої константи A), базові арифметичні операції (додавання, віднімання, множення, ділення і порівняння) та утворення максимальної кількості пар в графах.

Алгоритми поліноміального часу в теорії складності обчислень також визначають певні класи складності, пов'язані з їхнім використанням:

– P: клас складності завдань, які можуть бути розв'язані за поліноміальний час на детермінованій машині Тьюрінга;

– NP: клас складності завдань, які можуть бути розв'язані за поліноміальний час на недетермінованій машині Тьюрінга;

– ZPP: клас складності завдань, які можуть бути розв'язані з нульовою помилкою на ймовірнісній машині Тьюрінга за поліноміальний час;

– RP: клас складності завдань, які можуть бути розв'язані з односторонніми помилками на ймовірнісній машині Тьюрінга за поліноміальний час;

– BPP: клас складності завдань, які можуть бути розв'язані з двосторонніми помилками на ймовірнісній машині Тьюрінга за поліноміальний час;

– BQP: клас складності завдань, які можуть бути розв'язані з двосторонніми помилками на квантовій машині Тьюрінга за поліноміальний час.

Ці класи складності визначають області застосування алгоритмів та їхню здатність до розв'язання завдань у визначений час.

## 2.2 Просторова складність

Просторова складність алгоритмів визначає загальний об'єм пам'яті, необхідний для їх виконання відповідно до розміру вхідних даних.

Існує безліч варіантів опису просторової складності, але найпоширеніший - використання Big-O нотації. Вона надає уявлення про масштаби та продуктивність алгоритму, описуючи його верхню асимптотичну межу. Таким чином, ми можемо визначити найгірший сценарій щодо зростання швидкості виконання алгоритму.

Давайте детальніше розглянемо кілька прикладів функціонального опису просторової складності за допомогою Big-O нотації. Розглядатимемо різні випадки, починаючи від найкращого, коли об'єм необхідної пам'яті зростає найповільніше, до найгіршого, коли збільшення об'єму пам'яті відбувається найшвидше.

– константна складність (O(1)): займає постійний об'єм пам'яті незалежно від розміру вхідних даних.

– логарифмічна складність (O(log n)): використовує об'єм пам'яті, пропорційний логарифму розміру вхідних даних.

– лінійна складність (O(n)): використовує об'єм пам'яті, прямо пропорційний розміру вхідних даних.

– лінеаритмічна складність (O(n log n)): збільшується пропорційно розміру вхідних даних та логарифмічному коефіцієнту.

– квадратична складність (O(n^2)): збільшується пропорційно квадрату вхідного розміру.

Для вираження верхньої асимптотичної межі просторової складності алгоритму використовується позначення Омега (Ω). Це дозволяє відобразити математично найкращий сценарій складності алгоритму, протилежний Big-O нотації. Таким чином, ми можемо визначити, що "об'єм пам'яті, необхідний для виконання цього алгоритму, буде зростати не повільніше, ніж це значення, але може зростати ще швидше".

У контексті опису значення, що лежить в межах нижньої та верхньої границі, використовується символ θ. У таких випадках можна стверджувати, що "обсяг пам'яті, необхідний для виконання алгоритму, займає принаймні той обсяг простору (визначений функцією нижньої границі) і не перевищує значення функції, що представляє верхню межу".

## 2.3 Розмірно-орієнтовані метрики

Окрім цих показників, отримали розповсюдженні різноманітні метрики для оцінки і вимірювання різних характеристик програмного коду та процесу розробки. Так, розрізняють велику кількість метрик, кожна з яких розглядає певний аспект алгоритму програмного додатку [18-22].

Метрика SLOC (рядки вихідного коду) є простою метрикою, що вимірює кількість рядків коду у програмі. Цей метод оцінки був вперше запропонований у 50-х роках XX століття як один із критеріїв вимірювання трудовитрат на проєкт. Тоді популярні були мови програмування, такі як Фортран, Асемблер та Кобол.

Аналіз за допомогою цієї метрики включає два основних показники.

Кількість "фізичних" SLOC (LOC, SLOC, KLOC, KSLOC, DSLOC) визначається як загальна кількість рядків вихідного коду, включаючи коментарі та порожні рядки. При обчисленні цього показника порожні рядки обмежуються до 25% загальної кількості рядків у вимірюваному блоку коду;

Кількість "логічних" SLOC (LSI, DSI, KDSI, де SI - Source Instructions) визначається як кількість команд і залежить від мови програмування. Якщо мова не дозволяє розміщення кількох команд на одному рядку, "логічні" SLOC відповідають "фізичним" SLOC, за винятком порожніх рядків і рядків коментарів. У випадку, коли мова підтримує розміщення декількох команд на одному рядку, один "фізичний" рядок враховується як кілька "логічних", якщо він містить більше однієї команди.

Щодо обчислення SLOC, слід зауважити, що не існує єдиного загальновизнаного підходу для різних мов програмування. Найчастіше SLOC визначається як загальна кількість рядків коду за винятком порожніх рядків і коментарів. У більшості випадків враховується саме число "фізичних" SLOC, і випадки, коли на одному рядку є кілька операторів, не враховуються.

Для метрики SLOC існує багато похідних для отримання окремих показників проекту:

– число порожніх рядків (BLOC);

– число рядків, що містять коментарі (CLOC);

– число рядків з кодом і коментарями (C & SLOC);

– число рядків з декларативним вихідним кодом;

– число рядків з імперативним вихідним кодом;

– відсоток коментарів (кількість рядків коментарів, помножена на 100 і поділена на кількість рядків коду);

– середня кількість рядків для функцій (методів);

– середня кількість рядків з кодом для функцій (методів);

– середня кількість рядків для модулів;

– середня кількість рядків для класів.

Однак, оскільки ця метрика обмежується двома показниками, було розроблено багато похідних метрик, які враховують додаткові фактори, такі як кількість пустих рядків, рядків з коментарями, та інші.

Перевагою цієї метрики є її простота для розуміння та реалізації підрахунку. Але, як і будь-який метод оцінки, у неї є свої недоліки. Вона не є достатньо гнучкою і залежить від мови програмування та стилю розробника, що робить її менш ефективною для оцінки трудовитрат та складності коду.

Аналіз, що базується на SLOC-метриці, не завжди є повним для прийняття рішень щодо розробки програм та внесення змін, оскільки він обмежений кількістю критеріїв та залежить від зовнішніх факторів, таких як мова програмування. Зазвичай вона використовується як додатковий показник складності алгоритму.

Внаслідок розвитку науки та техніки виникла потреба у способі вимірювання складності алгоритмів. Це необхідно для адаптації рішень до можливостей обчислювальної техніки. У 1977 році Говард Холстед у своєму трактаті щодо розробки програмного забезпечення представив власний підхід до цієї проблеми, і його метрики стали одними з ключових у використанні до цього дня.

Холстед використовує аналогії із фізичними величинами, такими як вага, об'єм, тиск, для оцінки програмного коду. Його характеристики базуються на

практичних та теоретичних елементах коду і включають в себе різні параметри, до яких входять:

– n1 – кількість унікальних операторів програми, враховуючи символи-роздільники та знаки операцій;

– n2 – кількість унікальних операндів програми;

– N1 – загальна кількість операторів в програмі;

– N2 – кількість операндів програми;

– n1’ – теоретична кількість унікальних операторів програми;

– n2’ – теоретична кількість унікальних операндів програми.

На основі цих показників виконується розрахунок більш складніших параметрів оцінки коду:

– словник програми:

$$n = n_1 + n_2; \tag{2.1}$$

– довжина програми:

$$N = N_1 + N_2; \tag{2.2}$$

– теоретичний словник програми:

$$n' = n_1' + n_2'; \tag{2.3}$$

– теоретична довжина програми, де для стилістично коректних програм відхилення фактичної довжини від теоретичної не має перевищувати 10%:

$$N' = n_1 \log_2(n_1) + n_2 \log_2(n_2)\,; \tag{2.4}$$

– об’єм програми:

$$V = N \log_2 n\,; \tag{2.5}$$

– теоретичний об’єм програми:

$$V' = N' \log_2 n' ; \qquad (2.6)$$

– рівень якості програмування для ідеальної програми відношення фактичного і теоретичного об’єму становить одиницю:

$$L = \frac{V}{V'} ; \qquad (2.7)$$

– рівень якості програмування без врахування теоретичних параметрів:

$$L' = \frac{2n_2}{n_1 N_2} ; \qquad (2.8)$$

– складність розуміння програми:

$$E_c = \frac{V}{L'} ; \qquad (2.9)$$

– трудовитрати для написання програми:

$$D = \frac{1}{L'} ; \qquad (2.10)$$

– рівень мови виразу:

$$y' = L * V'; \qquad (2.11)$$

– інформаційний склад програми, що дозволяє зрозуміти розумові витрати на створення програми:

$$I = \frac{V}{D} ; \qquad (2.12)$$

– оцінка необхідних розумових зусиль, що характеризує число необхідних елементарних рішень для написання програми:

$$E = N' \log_2 (\frac{n}{L}) ; \qquad (2.13)$$

Ці метрики Холстеда вирівнюють певні недоліки, пов'язані з неоднозначністю відповідності між логічним рядком коду та його фізичним вираженням. Вони враховують якісні та кількісні показники, а також особливості мов програмування та стилів написання коду. Метрики Холстеда прості у використанні і частково компенсують недоліки інших методик, забезпечуючи більше параметрів для оцінки програмного коду.

Завдяки врахуванню різних аспектів написання коду, вони враховують різні мови програмування та стилі, що робить їх більш адаптованими та універсальними. Проте, варто відзначити, що деякі характеристики, такі як "складність розуміння програми" чи "рівень мови виразу", можуть бути суб'єктивними та абстрактними. Також, певні показники можуть бути важкими для порівняння між різними алгоритмами.

Ще одну метрику запропонував Томас Джилбі. Метрика логічної складності програми Джилба визначається як насиченість програми умовними операторами типу IF-THEN-ELSE та операторами циклу. Важливо враховувати, що запис умов і циклів може відрізнятися в різних мовах програмування, але при цьому сенс операторів залишається незмінним.

Вводяться такі характеристики програмного забезпечення:

– CL – абсолютна складність програми, визначена кількістю операторів умов та циклів;

– cl – відносна складність програми, яка визначає насиченість програми операторами умов та циклів. Вона обчислюється як відношення абсолютної складності CL до загальної кількості операторів n.

## 2.4 Метрики складності потоку керування програми

Дані метрики не зосереджені на кількості операторів, функцій, класів чи рядків коду. Замість цього, вони враховують різноманіття можливих сценаріїв виконання програми. Факторами, що впливають на потік виконання, є умовні оператори, виклики інших програм та оператори передчасного завершення.

Для отримання цих метрик ми розпочинаємо з побудови графа виконання програми. Важливим є використання орієнтованого графа з однією початковою та однією кінцевою вершиною. Кожна вершина графа представляє блок операцій та умов, які розділяють потік керування програмою. Дуги у графі відображають переходи між цими блоками за певних умов.

Важливим вимогам є те, щоб кожна вершина була досяжною з початкової вершини і з кожної вершини можна було дійти до кінцевої вершини.

Найпоширенішим методом визначення складності управління програмою є використання цикломатичної складності, також відомої як метрика Мак-Кейба. Критерії, за якими вона визначає складність виконання програми, обчислюються на основі графу потоку управління, що був описаний вище. Шляхом розрахунків цикломатичну складність можна визначити за допомогою формули:

$$V(G) = e - n + 2p, \qquad (2.14)$$

де $e$ – кількість дуг;

$n$ – кількість вершин;

$p$ – компонент зв’язності.

Значення компонента зв’язності в програмному графі визначається як кількість додаткових з'єднань (дуг), необхідних для того, щоб зробити граф потоку виконання програми сильно зв’язаним. Граф вважається сильно зв’язаним, якщо будь-які дві його вершини є взаємно досяжними, тобто існує шлях від будь-якої вершини до будь-якої іншої. У випадку графів, що представляють коректні

програми – тобто ті, що не містять недосяжних вершин від точки входу і в яких з будь-якої вершини можна дістатись до кінцевої – сильна зв'язаність досягається додаванням дуги між кінцевою та початковою вершиною.

Можна стверджувати, що параметр V(G) визначає кількість лінійно незалежних контурів у сильно зв'язаному графі.

Якщо розглядати коректні програми (де p=1), то вищезгадана формула набуває спрощеного вигляду:

$$V(G) = e - n + 2. \tag{2.15}$$

Метрика Мак-Кейба використовує кількість дуг графа та вершин як основні компоненти для визначення складності потоку. Цей підхід дозволяє враховувати складність незалежно від мови програмування, використаної для вирішення завдання. Оцінка, отримана за допомогою цієї метрики, легко розуміється.

Для подолання недоліків Метрики Мак-Кейба було запропоновано кілька методів, включаючи модифікації оригінального методу. Один із таких методів, який був представлений Маєрсом, полягає в оцінці конкретного інтервалу, що залежить від складності операторів у виразі:

$$[V(G), V(G) + h], \tag{2.16}$$

де $h$ = 0 для простих умовних виразів;

$h$ = $n$-1 для n-мірних виразів.

Хоча ця модифікація дозволяє подолати один із недоліків оригінального цикломатичного методу і враховує складність операторів, на практиці її використання виявляється вкрай обмеженим. Іншим способом вирішення цього недоліку подав Хансен, який пропонує оцінювати складність за допомогою пари чисел – цикломатичної складності та кількості операторів. Цей підхід вигідно вирізняється чітким врахуванням складності операцій незалежно від алгоритму чи мови програмування.

Ще однією модифікацією, запропонованою Ченом, є вираження складності програми через кількість перетинів підграфів у створеному графі потоку керування. Такий підхід зазвичай застосовується до структурованих програм, і його важливою особливістю є можливість лише послідовних переходів між блоками команд.

Ще однією модифікацією методу розрахунку цикломатичної складності є метод Пивоварського. Ця метрика дозволяє розуміти відмінність між послідовними та логічними конструкціями, а також між структурованими та неструктурованими програмами. Виразом для цього методу є:

$$N(G) = V'(G) + \sum P_i, \tag{2.17}$$

де $V'(G)$ – це модифікована цикломатична складність, де оператор CASE рахується як один логічний оператор, а не як n-1 логічний оператор;

$P_i$ – це глибина вкладеності і-ої логічної вершини.

Для визначення глибини вкладеності використовується термін «сфера впливу» логічної вершини графу. Ця сфера включає всі інші сфер логічних вершин, де оператори перетинаються з операторами даної вершини або повністю входять в неї. Зростання глибини сфери впливу відбувається не через вкладеність логічних операторів, а через вкладеність самих областей.

Метрика Пивоварського збільшується при переході від послідовних програм до вкладених і, в подальшому, до неструктурованих. Це є головною перевагою перед іншими методами та модифікаціями, що аналізують граф потоку виконання програми, оскільки вона враховує різницю між цими типами програм та визначає складність на основі рівня вкладеності.

## 2.5 Метрики складності потоку керування даними

Оцінка складності алгоритму здійснюється через аналіз потоку керування даними. Частота використання вхідних даних впливає на загальну складність алгоритму: чим частіше використовуються вхідні дані, тим складніше може бути

алгоритм. Метрики оцінки здебільшого спрямовані на аналіз вихідного коду програм, написаних на мовах високого рівня, та структур високорівневих даних.

Почнему з метрики Чепіна. Суть методу Чепіна полягає в оцінці інформаційної міцності окремих програмних модулів через аналіз характеру використання змінних зі списку вводу-виводу. Усі змінні, що становлять список вводу-виводу, розподіляються за 4 функціональні групи:

– $P$ – змінні для введення, використовувані для розрахунків та забезпечення виведення;

– $M$ – модифіковані або створювані всередині програми змінні;

– $C$ – змінні, що беруть участь в управлінні програмним модулем (керуючі змінні);

– $T$ – змінні, які не використовуються в програмі («паразитні»).

Оскільки одна змінна може виконувати кілька функцій одночасно, важливо враховувати це при її класифікації у відповідні функціональні групи. Таким чином, Метрика Чепіна виражається наступним чином:

$$Q = a_1 \times P + a_2 \times M + a_3 \times C + a_4 \times T, \quad (2.18)$$

де $a_1$, $a_2$, $a_3$, $a_4$ – вагові коефіцієнти.

В даному контексті використання вагових коефіцієнтів призначене для відображення різноманітного впливу на складність програми для кожної функціональної групи.

За думкою автора метрик, найважливішою є функціональна група С, якій призначено максимальний коефіцієнт 3, оскільки вона взаємодіє з потоком управління програмою. Коефіцієнти для інших груп розподілені так: $a1$=1, $a2$=2, $a4$=0.5. Групі T також призначено ненульовий ваговий коефіцієнт, оскільки "паразитні" змінні, хоча і не збільшують складність потоку даних програми, іноді ускладнюють її розуміння. Після врахування цих вагових коефіцієнтів, метрику Чепіна можна виразити наступним чином:

$$Q = P + 2M + 3C + 0{,}5T. \quad (2.19)$$

З метою отримання більш детального уявлення про методи оцінки складності алгоритмів, ми також проаналізуємо метрику, що пов'язує складність програм з використанням глобальних змінних.

Два компоненти, які ми позначимо як (*p, r*), вказують на пару "модуль-глобальна змінна", де $p$ – це модуль, що має доступ до глобальної змінної *r*. Враховуючи факт або можливість звернення до змінної r, можна виокремити два типи пар "модуль-глобальна змінна": фактичні та можливі. Припустиме звернення до $r$ за допомогою $p$ вказує на те, що область існування $r$ включає $p$. Ця характеристика позначається як $A_{up}$ і відображає, скільки разів модулі $\text{U}_p$ фактично мали доступ до глобальних змінних, тоді як число $P_{up}$ вказує, скільки разів вони могли б отримати такий доступ.

Відношення кількості фактичних звернень до можливих визначається як:

$$R_{up} = \frac{A_{up}}{P_{up}}. \qquad (2.20)$$

За допомогою цієї формули можна показати наближену можливість посилання довільного модуля на довільну глобальну змінну.

Очевидно, що чим вище ця ймовірність, тим вища ймовірність «несанкціонованої» зміни будь-якої змінної, що може суттєво ускладнити роботи, пов'язані з модифікацією програми.

Засновуючись на концепції потоків інформації, було розроблено ще один метод для оцінки складності потоку, відомий як метрика Кафура. Цей підхід включає в себе визначення локального та глобального потоків. Локальний потік інформації від A до B вважається наявним у таких випадках:

– модуль A викликає модуль B (прямий локальний потік);

– модуль B викликає модуль A, і A повертає значення, яке використовується в B (непрямий локальний потік);

– модуль C викликає модулі A та B і передає результат виконання модуля A в B.

Глобальний потік інформації від A до B через глобальну структуру даних D вважається наявним, якщо модуль A розміщує інформацію в D, а модуль B використовує цю інформацію з D.

На основі введених понять визначається величина I, що відображає інформаційну складність процедури:

$$I = lenght * (fan_in * fan_out)^2, \quad (2.21)$$

де *length* – складність тексту процедури, яка вимірюється за допомогою певної метрики об'єму, такої як метрики Холстеда, Маккейба, LOC та інші;

*fan_in* – кількість локальних потоків процедури, які входять всередину, плюс кількість структур даних, з яких процедура отримує інформацію;

*fan_out* – кількість локальних потоків, які виходять із процедури, плюс кількість структур даних, які оновлюються процедурою.

Модуль може бути характеризований інформаційною складністю, яка представляє собою суму інформаційних складностей процедур, включених до його структури.

Наступним етапом є аналіз інформаційної складності модуля в контексті конкретної структури даних. Міра інформаційної складності модуля, що відноситься до структури даних, може бути визначений так:

$$J = NW * NW + NW * NRW + \mathrm{NRW} * \mathrm{NR} + \mathrm{NRW} * (\mathrm{NRW} - 1), \quad (2.22)$$

де $W$ – кількість процедур, які тільки оновлюють структуру даних;

$R$ – кількість процедур, які тільки читають інформацію із структури даних;

$RW$ – кількість процедур, які як читають, так і оновлюють інформацію в структурі даних.

Також важливо відзначити, що розглянуті метрики складності програм базуються на аналізі вихідних текстів програм та їх графів, що забезпечує єдиний підхід до автоматизації їх розрахунку.

Ще однією з метрик, що визначають складність управління потоком даних, є метрика Овієдо. Основна ідея полягає в розбитті програми на неперетинаючі лінійні ділянки – промені операторів, які утворюють керуючий граф програми.

Позначимо R(i) як безліч визначальних входжень змінних у промені і, розташованих в радіусі його дії. Тут визначальне входження змінної вважається таким, що знаходиться в радіусі дії променя, або ця змінна локальна для променя і має визначальне входження, або вона має визначальне входження в попередньому промені, але не має локального визначення.

Також визначимо V(i) як безліч змінних, використовуючи входження яких утворюється промінь і. Міра складності і-го променя визначається як:

$$DF(i) = \sum(DEF(v_j)), j = i \ldots \|V(i)\|, \quad (2.23)$$

де $DEF(v_j)$ – число визначальних входження змінної $v_j$ з множини $R(i)$;

$//V(i)//$ – потужність множини $V(i)$.

## 2.6 Метрики складності потоку керування даними та програми

Четвертий клас метрик охоплює параметри, що визначають складність як потоку управління програмою, так і потоку управління даними. З одного боку, це включає кількісні показники, пов'язані із складністю потоку управління програмою. З іншого боку, враховуються аспекти складності потоку управління даними. Важливо відзначити, що цей клас метрик, хоча і називається топологічними метриками, об'єднує в собі два аспекти - метрики кількісних показників та метрики складності потоку управління програмою. Виділення їх у відповідному контексті сприяє більшому розумінню. Ці метрики дозволяють визначити складність структури програми на основі якісного та кількісного аналізу управлінських структур.

Однією з метрик є тестуюча М-Міра. Під тестуючою мірою М розуміється характеристика складності, що відповідає умовам, описаним нижче. Рівень цієї

міри збільшується із збільшенням глибини вкладеності та враховує протяжність програми. Також існує міра, базована на регулярних вкладеннях, яка тісно пов'язана з тестуючою мірою М. Основна ідея цієї метрики складності програм полягає в підрахунку сумарної кількості символів (операндів, операторів, дужок) у регулярному виразі з мінімально необхідною кількістю дужок, яка описує керуючий граф програми. Всі виміри цієї групи реагують на вкладеність управляючих конструкцій та протяжність програми, але вони також призводять до збільшення обчислювальної складності.

Додатково, важливо враховувати міру взаємодії модулів програмного забезпечення як показник якості. Якщо модулі сильно взаємопов'язані, програма стає складнішою для модифікації та розуміння. Цю метрику можна чисельно виразити. Розрізняють різні типи взаємодії модулів:

– зв'язок за даними: виникає, коли модулі взаємодіють, передаючи параметри, при цьому кожен параметр є елементарним інформаційним об'єктом. Цей вид зв'язку є найоптимальнішим, що стосується зчеплення;

– зв'язок за структурою даних: виникає, коли один модуль передає іншому складовий інформаційний об'єкт (структуру) для обміну даними;

– зв'язок за управлінням: виникає, коли один модуль посилає іншому інформаційний об'єкт – прапор, призначений для управління внутрішньою логікою;

– загальна область: модулі пов'язані, якщо вони звертаються до однієї і тієї ж глобальної області даних. Такий вид зчеплення небажаний, бо помилка в одному модулі може виявитися в будь-якому іншому, а також такі програми важкі для розуміння;

– зв'язок за вмістом: виникає, коли один модуль звертається до вмісту іншого, порушуючи принцип модульності;

– зовнішні дані: два модулі використовують зовнішні дані, такі як комунікаційний протокол;

– повідомлення: модулі взаємодіють через повідомлення без параметрів;

– відсутність зв'язаності: модулі не взаємодіють;

– підкласова пов'язаність: виникає відношення між класом-батьком і класом-нащадком, де нащадок пов'язаний з батьком, але не навпаки;

– зв'язаність за часом: дві дії групуються в одному модулі лише через одночасність в окремих обставинах.

Наступна метрика із цього класу – Метрика Мак-Клура. Процес обчислення цієї метрики включає три етапи:

– розрахунок значень складних функцій C(i) для кожної керуючої змінної i виконується за допомогою наступної формули:

$$\mathrm{C}(i) = \frac{(D(i) \times J(i))}{n}, \tag{2.24}$$

де $D(i)$ – сфера дії змінної i;

$J(i)$ – складність взаємодії модулів через змінну i;

$n$ – загальну кількість окремих модулів у схемі розбиття.

– для всіх модулів, які входять у сферу розбиття, визначається значення їх складних функцій M(P) за формулою:

$$M(P) = f_p \times X(P) + g_p \times Y(P), \tag{2.25}$$

де $fp$ і $gp$ – відповідно, число модулів, що безпосередньо передують і безпосередньо наступають за модулем P;

$X(P)$ – складність звернення до модуля P;

$Y(P)$ – складність управління викликом із модуля P інших модулів.

– загальна складність MP ієрархічної схеми розбиття програми на модулі задається наступною формулою:

$$MP = \sum (M(P)), \tag{2.26}$$

де по всіх можливих значень $P$ – модулям програми.

Дана метрика призначена для програм, які мають добре структурований характер і складаються з ієрархічних модулів, що визначають функціональну специфікацію та структуру управління. Також важливо враховувати, що в кожному модулі існує одна точка входу і одна точка виходу. Кожен модуль виконує рівно одну функцію, і виклик модулів здійснюється відповідно до ієрархічної системи управління, що визначає взаємозв'язок виклику між безліччю модулів програми. Таким чином, організація програми базується на чітко визначених модулях, які виконують конкретні функції відповідно до ієрархічної структури управління.

## 2.7 Об’єктно-орієнтовані метрики

Клас метрик, пов'язаних із об'єктно-орієнтованою парадигмою програмування, виник у період активного розвитку об'єктно-орієнтованих мов. Цей напрямок став важливим в програмуванні та розробці програмного забезпечення. Першими серед таких метрик стали набори, представлені Мартіном, а також Чидамбером і Кемерером. У подальших розділах розглянемо їх детальніше, розпочинаючи із метрик Мартіна.

Перед тим як розпочати аналіз метрик Мартіна, слід встановити термін "категорія класів". Суттєвою є те, що класи рідко бувають використані ізольовано від інших класів. Практично кожен клас функціонує в синергії з групою класів, з якою він взаємодіє, і від якої він не може бути легко відокремлений. Повторне використання таких класів передбачає використання всієї групи класів, яка має високий рівень внутрішнього зв'язку та отримала назву "категорія класів". Умови існування категорії класів включають:

– класи всередині категорії класів ізольовані від будь-яких спроб спільних змін. Це означає, що зміни в одному класі майже невідмінно призведуть до змін у всіх класах цієї категорії. Якщо будь-який клас відкритий для конкретного виду змін, то всі класи в цій категорії стають вразливими перед цим видом змін;

– класи в категорії використовуються разом та невіддільно. Вони так взаємозалежні, що не можуть бути використані окремо один від одного. Таким чином, будь-яка спроба використання одного класу категорії вимагає повторного використання всіх інших класів разом із ним;

– класи в категорії виконують спільну функцію або досягають спільної мети.

Відповідальність, незалежність та стабільність категорії можуть бути виміряні через аналіз взаємодій, які наявні у цьому контексті. Таким чином, були визначені три показники:

– $C_a$ – параметр центрошвидкого зчеплення. Кількість класів за межами цієї категорії, які мають взаємозв'язок із класами всередині цієї категорії.

– $C_e$ – параметр відцентрового зчеплення. Кількість класів всередині цієї категорії, які залежать від класів за її межами.

– $I$ – півень нестабільності. Ступінь коливань та невпевненості, що характеризує стійкість цієї категорії.

$$I = \frac{C_e}{(C_a + C_e)}, \tag{2.27}$$

Ця метрика представляє собою діапазон значень від 0 до 1, де:

– $I$ = 0 вказує на найвищий рівень стабільності в категорії.

– $I$ = 1 вказує на найвищий рівень нестабільності в категорії.

Додатково, можна врахувати метрику абстракції, яка визначає ступінь абстрактності категорії. Якщо категорія є абстрактною, то вона є гнучкою і може легко розширюватися.

$$A = \frac{n_A}{n_{All}}, \tag{2.28}$$

де $n_A$ – кількість абстрактних класів в категорії;

$n_{All}$ – загальна кількість класів в категорії.

Значення цієї метрики коливаються у межах інтервалу [0,1]:

– 0 відповідає категорії, що є абсолютно конкретною,

– 1 відповідає категорії, яка є абсолютно абстрактною.

На основі представлених метрик Мартіна можна побудувати графік, на якому відображено взаємозв'язок між ступенем абстракції та рівнем нестабільності. Якщо побудувати пряму, що описується формулою I + A = 1, то на цій лінії розташуються категорії, які мають оптимальний баланс між ступенем абстракції та рівнем нестабільності. Цю лінію відомо як головна послідовність.

Додатково можна включити ще дві характеристики:

– відстань від основної послідовності:

$$D = \left|\frac{A + I - 1}{\sqrt{2}}\right|; \tag{2.29}$$

– нормалізована відстань від основної послідовності:

$$D_n = |A + I - 2|. \tag{2.31}$$

Практично для будь-яких категорій важливо, що чим ближче вони розташовані до основної послідовності, тим краще.

Наступною йде підгрупа метрик є Чідамбера та Кемерера, які базуються на оцінці методів класів, дерев успадкування та інших аспектів.

– WMC (Weighted methods per class, вагована кількість методів на клас) визначає загальну складність методів у класі:

$$WMC = \sum ci\,, i = 1 \dots n. \tag{2.32}$$

де $ci$ – складність кожного окремого методу, розраховану за певною метрикою (наприклад, за Холстедом або іншою, залежно від обраного критерію). У випадку, коли у всіх методів складність однакова, WMC = $n$.

– глибина дерева успадкування (DIT, Depth of Inheritance tree): визначається як найбільший шлях ієрархією класів до даного класу від класу-предка. Більша глибина сприяє вищій абстракції даних та зменшує насиченість класу методами.

Проте, при надто великій глибині може виникнути складність у розумінні та написанні програм.

– кількість нащадків (NOC, Number of children): визначає кількість безпосередніх нащадків класу. Збільшення цього числа сприяє вищій абстракції даних.

– зчеплення між класами (CBO, Coupling between object classes): вказує на кількість класів, з якими пов'язаний даний клас. Високий рівень CBO може зменшити абстракцію даних і ускладнити повторне використання класу.

– відгук за клас (RFC, Response for a class): RFC=|RS|, RS розраховується як кількість методів класу у відповідь на дані, отримані об'єктом класу. Тобто:

$$RS = (\{M\}\{Ri\}), i = 1 \dots n, \tag{2.33}$$

де $M$ – всі можливі методи, що належать класу;

$Ri$ - всі можливі методи, які можуть бути викликані класом під номером і.

Отже, RFC представляє собою потужність даної множини. Чим вищий RFC, то тестування та налагодження стають більш складними.

– нестача зчеплення методів (LCOM, Lack of cohesion in Methods) Для визначення цього параметра розглянемо клас C з n методами M1, M2, ..., Mn. Позначимо {I1}, {I2}, ..., {In} як набори змінних, які використовуються в даних методах. В цьому випадку:

$$LCOM = |P| - |Q|. \tag{2.34}$$

де $P$ – кількість пар методів, в яких відсутні спільні змінні;

$Q$ – кількість пар методів, в яких наявні спільні змінні.

Позначення низького рівня зчеплення може служити індикатором того, що можна розглянути можливість розбиття даного класу на кілька інших класів або підкласів. Це сприятиме підвищенню рівня інкапсуляції даних та скороченню складності як класів, так і методів. Таким чином, для досягнення більшої

ефективності рекомендується акцентувати увагу на взаємодії між класами та їх підкласами, щоб покращити загальну організацію програмного коду.

## 2.8 Гібридні метрики

Необхідно вказати на інший тип метрик, які відомі як гібридні. Цей клас метрик базується на менш складних характеристиках і є їхньою зваженою комбінацією. Прикладом цього підходу є метрика, відома як Кокола, що визначається як вагова сума простіших метрик.

$$HM = \frac{M + R1 \times M(M1\,) + \cdots + Rn \times M(Mn)}{1 + R1 + \cdots + Rn}, \tag{2.35}$$

де $M$ – базова метрика;

$Mi$ – інші цікаві міри;

$Ri$ – коректно підібрані коефіцієнти;

$M(Mi)$ – функції.

Функції $M(Mi)$ та коефіцієнти $Ri$ розраховуються за допомогою методів регресійного аналізу чи аналізу конкретної програмної задачі. Визначено три моделі для оцінки складності: Маккейба, Холстеда та SLOC, з метрикою Холстеда як базовою. Ці моделі класифікуються як "найкраща", "випадкова" та "лінійна".

Метрика Зольновського, Сіммонса та Тейєра також представляють собою зважену суму різноманітних показників. Існують дві версії даної метрики:

$$\sum(a, b, c, d)\,, \tag{2.36}$$

де $a$ – структура;

$b$ – взаємодія;

$c$ – об’єм;

$d$ – дані.

А також:

$$\sum(x, y, z, p)\ , \tag{2.37}$$

де $a$ – складність інтерфейсу;

$b$ – обчислювальна складність;

$c$ – складність вводу/виводу;

$d$ – читабельність.

Вибір конкретних метрик у кожному випадку залежить від поставленої задачі, а коефіцієнти обчислюються відповідно до значень метрик для прийняття рішення у конкретному контексті.

Отже, в другому розділі були розглянуті існуючі засоби обчислення складності алгоритмів програмного застосунку. Є велика кількість категорій, по яким розподілені ці засоби обчислення, і які вказують на можливість досліджувати код алгоритму з різни точок зору. А також це вказує на те, що жодна з метрик не абсолютною, і що в кожній наявні як переваги, так і недоліки.

# 3 МЕТРИКА ДИФЕРЕНЦІЙОВАНОЇ ЦИКЛОМАТИЧНОЇ СКЛАДНОСТІ

Для досягнення мети, зазначеної для даної кваліфікаційної роботи, було прийнято рішення проведення модифікації метрики цикломатичної складності шляхом комбінування її алгоритму з алгоритмом іншої існуючої метрики. Взагалі метрика цикломатичної складності була обрана за наступними рисами:

– простота розуміння: метрика проста та легко розуміється. вона відображає кількість можливих шляхів у коді, що полегшує її інтерпретацію;

– виявлення складних функцій: допомагає виділити складні функції, які можуть бути важкими для тестування та розуміння;

– керованість: ця метрика може служити інструментом керування для розробників, допомагаючи їм уникати надмірно складних конструкцій;

– високий рівень доступності: завдяки характеру використання цієї метрики, вона майже не обмежена конкретною мовою програмування чи інструментарієм. Вона може бути застосована до програмного коду, написаного на різних мовах програмування, таких як Java, C++, Python, і багатьох інших. Це стандартна метрика, яка оцінює структурну складність коду, незалежно від конкретної мови програмування;

– універсальність: завдяки власному алгоритму використання, цикломатична складність може бути використана на всіх етапах розробки програмного застосунку, де має місце код або інший засіб опису алгоритму.

Однак, через власну ж простоту, ця метрика має значні недоліки у власному використанні. Найбільшим недоліком була визначена ускладненість при розрахунку об’єктивної складності програмного застосунку. Точніше кажучи, може виникати ситуація, коли цикломатична складність двох алгоритмів демонструє приблизно однакові показники, коли операції всередині цих алгоритмів, насправді, неоднаково складні.

Для здійснення модифікації метрики цикломатичної складності було виконано огляд існуючих метрик, серед яких було виконано відбір за наступними критеріями:

– простота: відповідно до цикломатичної складності, шукана метрика має так само демонструвати легкість у використанні та розумінні;

– доступність: так само, як і цикломатична складність, шукана метрика має бути придатної до використання незалежно від мови програмування та інших аналогічних чинників;

– сумісність: новий алгоритм не має конфліктувати з перевагами цикломатичної складності, а також має продовжувати ідею використання даної метрики – визначення шляхів тестування. Визначення додаткового параметра має без додаткових складнощів виконуватися з алгоритмами в межах певного програмного модуля, так само як і метрика цикломатичної складності. Бажано, щоб додатковий параметр можна було комбінувати з методикою використанням графа потоку управління.

## 3.1 Переваги та недоліки існуючих метрик

Для пошуку необхідно визначити переваги та недоліки розглянутих засобів оцінки складності алгоритмів програмних застосунків.

Часова складність є метрикою, яка виражає обчислювальну складність алгоритму в термінах часу його виконання в залежності від розміру введених даних. В той час як часова складність є цінним критерієм для розуміння ефективності алгоритмів, вона має свої плюси та мінуси:

Переваги:

– оцінка ефективності алгоритму: часова складність надає можливість аналізу ефективності алгоритмів з точки зору часу виконання відносно розміру вводу. Це критично для вибору відповідних алгоритмів для вирішення конкретних завдань.

– об'єктивна міра: часова складність є об'єктивною метрикою, яку можна виводити з самого алгоритму, що дозволяє стандартизований та послідовний спосіб оцінювати алгоритми на різних платформах та в середовищах.

– порівняльний аналіз: вона дозволяє легко порівнювати різні алгоритми, що розв'язують одну й ту ж проблему. Розробники можуть обирати найефективніший алгоритм для конкретного випадку в залежності від його часової складності.

– прогностична сила: часова складність може надавати уявлення про те, як змінюється продуктивність алгоритму при збільшенні розміру вводу. Ця прогностична сила цінна для оцінки того, як алгоритм буде себе вести на великих наборах даних.

– фокус на критичних операціях: виражаючи часову складність у термінах "великого О," розробники можуть акцентувати увагу на найважливіших операціях, що впливають на продуктивність, допомагаючи виявляти та оптимізовувати проблеми.

Недоліки:

– спрощена модель: часова складність надає високорівневу абстракцію та спрощення реального часу виконання. Вона може не враховувати всі фактори, що впливають на продуктивність в реальному світі, такі як постійні фактори, конкретності апаратного забезпечення та умови навколишнього середовища.

– залежить від платформи та мови програмування: Точний час виконання алгоритму може відрізнятися в залежності від мови програмування, оптимізацій компілятора, архітектури апаратного забезпечення та інших факторів, залежних від платформи. Часова складність - це теоретична міра, яка не враховує цих варіацій.

– ігнорує постійні фактори: Часова складність фокусується на темпі зростання часу роботи при збільшенні розміру вводу, але ігнорує постійні фактори. Два алгоритми з однаковою часовою складністю можуть мати різний фактичний час роботи через приховані константи.

– не завжди відображає реальну продуктивність: алгоритм із меншою часовою складністю не завжди є кращим вибором на практиці. Реальна

продуктивність може залежати від факторів, які не враховуються при аналізі часової складності, такими як розподіл даних, використання пам'яті та конкретні вимоги програмного застосування.

Просторова складність - це метрика, яка використовується для вимірювання обсягу пам'яті або просторового простору, необхідного для виконання алгоритму чи програми як функції розміру вхідних даних. Як і будь-яка метрика, у просторової складності є свої переваги та недоліки:

Переваги просторової складності:

– прогнозування використання ресурсів: просторова складність допомагає розробникам оцінювати та передбачати обсяг пам'яті, який програма буде потребувати для різних розмірів вхідних даних. Це важливо для планування та розподілу ресурсів.

– оптимізація: аналізуючи просторову складність, розробники можуть виявити можливості оптимізації використання пам'яті, що призводить до більш ефективних алгоритмів та програм.

– порівняння алгоритмів: просторова складність дозволяє порівнювати різні алгоритми, які вирішують ту ж саму проблему. Розробники можуть вибирати алгоритми на основі їх ефективності щодо простору, залежно від вимог системи.

– виявлення витоків пам'яті: аналіз просторової складності може допомогти виявити потенційні витоки пам'яті чи області з надмірним споживанням пам'яті в програмі.

– розуміння компромісів: аналіз просторової складності спонукає розробників розуміти компроміси між часом та простором. У деяких випадках покращення часової складності може призвести до збільшення просторової складності і навпаки.

Недоліки просторової складності:

– не завжди критична: у деяких застосунках чи середовищах, де пам'ять є вдосталь, просторова складність може не бути настільки критичною проблемою. У таких випадках оптимізація часової складності може бути вищим пріоритетом.

– залежить від реалізації: просторова складність залежить не лише від самого алгоритму, але й від конкретної реалізації. Різні мови програмування та компілятори можуть внести варіації в вимогах щодо пам'яті.

– складність уточненого вимірювання: точне вимірювання просторової складності може бути складним через фактори, такі як фрагментація пам'яті та накладні витрати на виділення. Системи реального світу можуть мати інші просторові складності, ніж теоретичний аналіз вказує.

– перебільшення низькорівневих деталей: занадто сильний фокус на просторовій складності може призвести до мікрооптимізацій на шкоду зрозумілості та підтримуваності коду. Важливо знайти баланс між ефективністю та читабельністю коду.

– непридатність для деяких проблем: для певних типів проблем просторова складність може бути не найбільш відповідною метрикою. Наприклад, в деяких вбудованих системах основною проблемою може бути споживання енергії, а не використання пам'яті.

SLOC (Source Lines of Code) – це метрика, яка вимірює кількість рядків програмного коду у програмному продукті. Ось деякі переваги та недоліки використання SLOC метрик:

Переваги SLOC метрик:

– простота вимірювання: вимірювання кількості рядків коду – це відносно простий і швидкий спосіб отримання величини для аналізу.

– зручність порівняння: SLOC легко порівнювати між різними проектами або версіями програми, дозволяючи оцінити розмір кодової бази.

– визначення прогресу розробки: зміни в SLOC можуть служити показником прогресу розробки або масштабу завдання.

– база для інших метрик: SLOC може служити вихідною точкою для розрахунку інших метрик, таких як завантаження роботи, витрати на тестування та інші.

– простота в розрахунках оцінок: SLOC може використовуватися для оцінювання часу та зусиль, необхідних для завершення проекту.

– використання у контексті проміжних оцінок: SLOC може слугувати для оцінки проміжних результатів, які можна використовувати для прийняття рішень у реальному часі.

– використання у формулюванні конфігураційних змін: зміни в SLOC можуть служити підставою для формулювання конфігураційних змін або для визначення, які частини програми важливі для певних аспектів розробки.

Недоліки SLOC метрик:

– непередбачуваність стандартів коду: різні команди можуть використовувати різні стандарти форматування коду, що робить порівняння SLOC менш надійним.

– необхідність урахування якості коду: SLOC не враховує якість коду або його читабельність, що може бути критичним для загальної оцінки проекту.

– неспроможність розрізняти обсяг функціоналу: однакова кількість SLOC може представляти різні обсяги функціональності, залежно від реалізації.

– неспроможність розрізняти ефективність: кількість рядків коду не завжди є індикатором ефективності або продуктивності розробника.

– вартість обслуговування: зростання SLOC може вказувати на збільшення вартості обслуговування кодової бази без забезпечення нової функціональності.

– неспроможність розрізняти мов програмування: різні мови програмування можуть мати різні вимоги до SLOC для вираження одного і того ж функціоналу.

– вплив автоматизації: автоматизація інструментів розробки може значно впливати на кількість SLOC і зробити її менш значущою метрикою.

Метрика Холстеда базується на основі виконанні обчислень ряду параметрів, беручи за основу такі базові елементи, як кількість операторів та операндів.

Переваги метрики Холстеда:

– простота в обчисленні: метрика Холстеда використовує прості математичні операції, такі як додавання та логарифмування, для обчислення різних параметрів, що робить її досить простою у реалізації.

– висока рівність до мов програмування: метрика не залежить від конкретної мови програмування і може бути використана для оцінки програм незалежно від їхньої реалізації.

– фокус на обсягу програми: метрика враховує обсяг програми (кількість операторів та операндів), що дозволяє оцінити загальну складність коду.

– використання для приблизної оцінки продуктивності програміста: Холстед вважав, що його метрика може слугувати інструментом для оцінки продуктивності програміста, хоча це досить спірне використання.

– можливість порівняння програм: метрика надає засіб порівняння різних версій програми або програм, написаних на різних мовах програмування.

– помічник у визначенні рівня складності: Холстед допомагає визначити рівень складності програми на основі різних параметрів, таких як довжина коду, обсяг словникових та операторних термінів.

– використання для приблизного прогнозу тривалості розробки: деякі параметри метрики можуть слугувати орієнтовними індикаторами тривалості розробки програми.

Недоліки метрики Холстеда:

– відсутність урахування якості коду: метрика Холстеда не враховує якість коду чи його структуру, що може призводити до неправильних висновків щодо його складності та ефективності.

– нечутливість до реальних складнощів: метрика може бути нечутливою до деяких реальних складнощів програмного проекту, таких як взаємодія з базою даних чи мережею.

– не враховує контексту використання: метрика не розглядає контекст використання програми, тобто вона не враховує, наскільки важливою є програма для свого власного середовища.

– відсутність врахування різниці в складності між операціями: метрика розглядає всі операції однаково, не враховуючи можливі різниці в їхній складності.

– залежність від обсягу коду: часто обсяг програми може залежати від індивідуальних стилів програмування, що може впливати на об'єктивність оцінки.

– важкість визначення "розуміння коду": один із параметрів - "розуміння коду" - важко об'єктивно виміряти.

– використання засобу для вимірювання продуктивності програміста: використання метрики Холстеда для вимірювання продуктивності програміста є спірним і може призвести до неправильних висновків.

Метрика Джилба використовує розрахунки наповненості коду операторами умови та операторами циклів.

Переваги метрики Джилба:

– простота: метрика використовує кількісні показники операторів циклів та умов, а також їх відношення до загальної кількості операторів;

– інформативність: цикли та умови є основним способом відображення складності структури програми, завдяки чому можна визначити трудомісткість та рівень складності розуміння програмного алгоритму.

Недоліки метрики Джилба:

– відсутність урахування якості коду: метрика не враховує якість коду чи його структуру, що може призводити до неправильних висновків щодо його складності та ефективності;

– нечутливість до реальних складнощів: метрика може бути нечутливою до деяких реальних складнощів програмного проекту, таких як взаємодія з базою даних чи мережею;

– відсутність врахування різниці в складності між операціями: метрика розглядає лише самі оператори умов та циклу, всі інші оператори для цієї метрики не суттєві, не зважаючи на їхній рівень складності.

З розглянутих метрик жодна не була визначена, як підходяща для використання в процесі модифікації цикломатичної складності. Часова складність

використовується для кожного окремого методу і потребує значної переробки для використанні в межах більшого програмного модуля разом з цикломатичної складністю. Просторова складність досить залежна від технічних засобів реалізації алгоритму, через що погано поєднується з цикломатичною складністю. SLOC метрики не дозволять визначити складність операцій всередині коду. Метрика Холстеда за рахунок великої кількості власних показників може досить ефективно описувати код, через використання з цикломатичною складністю просто недоречне. Метрика Джилба, так само як і цикломатична складність, вказує на розгалуженість алгоритму, але не на складність операцій.

Використання інших метрик складності потоку керування програми, очевидно, не принесе результатів. Усі розглянуті метрики є модифікаціями методики обчислення метрики цикломатичної складності, але так само зосереджені більше на структурі алгоритму, його розгалуженості. Ці модифікації поділяють і ті самі недоліки.

Метрика Чепіна базується на використанні змінних введення-виведення. Її переваги:

– загальна інформаційність: метрика враховує різні типи змінних (p, m, c, t), що надає більш всебічний погляд на потік інформації всередині програми;

– налагоджувана вага: метрика дозволяє налаштовувати вагу за допомогою коефіцієнтів (a1, a2, a3, a4), надаючи гнучкість для адаптації метрики під конкретні характеристики проекту;

– чутливість до різних змінних: розрізняючи між вхідними, модифікованими, керуючими та невикористаними змінними, метрика може вражати різні аспекти потоку інформації та складності програми;

– виділення складності керуючого потоку: включення керуючих змінних (c) дозволяє метриці враховувати складність, пов'язану з прийняттям рішень та керуючим потоком в програмі;

– кількісне вимірювання: метрика забезпечує кількісне вимірювання інформаційного змісту, що може бути корисним для порівняння та оцінки різних програм з точки зору складності;

– простота: методика оцінки метрики Чепіна не вимагає значних витрат у часі або труді. До того ж, її можна використовувати незалежно від мови програмування.

Недоліки метрики Чепіна:

– суб'єктивність установлення ваги: вибір коефіцієнтів ваги (a1, a2, a3, a4) може бути суб'єктивним, і різний вибір може призвести до різних тлумачень складності програми;

– не враховує семантичну складність: метрика в основному фокусується на синтаксичних аспектах (типи змінних та їх потік), і може не враховувати семантичної складності програми, такої як логічні відносини між змінними;

– залежність від найменування змінних: метрика покладається на імена змінних для їх класифікації в різні групи. неспівпадіння у конвенціях найменування змінних може вплинути на точність метрики;

– може надто акцентувати невикористовані змінні: включення терміну для невикористаних змінних (t) в рівняння може призвести до ситуації, коли невикористані змінні сприяють загальній складності, що може не відповідати реальній поведінці програми.

Метрика Кафура використовує інформаційні потоки між модулями. Ця метрика дозволяє визначити інформаційну складність модуля на основі кількості елементів та структур даних, з яких модуль бере інформацію і які він оновлює. Однак має кілька недоліків:

– потреба введення концепцій локальних та глобальних потоків у програмне забезпечення, на яких залежить оцінка інформаційної складності.

– визначення ваг, які розраховуються на основі досвіду попередніх проектів. невідповідне використання при швидкій зміні технологій, методів та засобів проектування (показує невірні оцінки).

Суть метрики Овієдо полягає в тому, що програма розбивається на лінійні ділянки, що не перетинаються – промені операторів, які утворюють керуючий граф програми.

Переваги метрики:

– дозволяє враховувати потік управління в різних базових блоках програми;

– дозволяє оцінювати взаємозв'язок елементів у межах програми;

– дозволяє виявляти приховані залежності в програмі та перетворювати їх у явну форму, що спрощує логіку програми;

Недоліки Метрики:

– суб'єктивність деяких критеріїв вибору шляхів;

– потреба в шаблонах для структур управління;

– спотворення потоку управління, що призводить до збільшення складності вершин графа потоку управління, що не дозволяє ефективно використовувати метрику.

– контрольні змінні також можуть впливати на потік управління програмою.

Метрика Чепіна досить проста та доступна, що дозволяє її розглянути для використання з цикломатичною складністю. Метрики Кафура та Овієдо ж, навпаки, мають досить складну методику.

Метрика Мак-Клура та тестуюча М-Міра, як метрики складності потоку керування даними та програми вже демонструють себе як комбіновані методики, які основані на принципах двох категорій метрик, до однієї з яких входить цикломатична складність. Комбінування в такому випадку недоречне.

Метрики Мартіна використовує поняття класів та досліджує взаємозв’язки між ними.

Переваги метрик Мартина:

– направлення на проектування: метрики Мартина надають напрямок для проектування об'єктно-орієнтованих систем, які є абстрактними та стабільними. Це відповідає принципам гарного дизайну, таким як Принцип Інверсії Залежностей.

– виявлення проблем дизайну: високі значення еферентного або аферентного зв'язку можуть вказувати на можливі проблеми дизайну. Наприклад, високий еферентний зв'язок може свідчити про те, що модуль занадто залежний від інших модулів, що впливає на його стабільність.

– кількісне вимірювання: метрики надають кількісні значення, які можна відстежувати з часом, що дозволяє проводити постійну оцінку якості дизайну системи.

– візуальне представлення: площина нестабільності-абстракції надає візуальне представлення, яке може допомогти розробникам та архітекторам швидко оцінити загальні характеристики дизайну модуля.

Недоліки метрик Мартина:

– чутливість до змін: метрики Мартина можуть бути чутливими до невеликих змін у коді або відносинах класів. Ця чутливість іноді може призводити до коливань у значеннях метрик, які можуть не точно відображати загальний дизайн системи.

– обмежений обсяг: метрики в основному фокусуються на абстракції та стабільності класів всередині модуля. Вони можуть не враховувати всі аспекти якості дизайну, такі як інкапсуляція, коесія або конкретні вимоги домену.

– залежність від ієрархій класів: метрика абстракції ґрунтується на наявності абстрактних класів, що може бути не застосовано в усіх сценаріях дизайну. В системах, які широко використовують інтерфейси або інші механізми для абстракції, метрика може не надавати повної картини.

– не підходить для всіх проектів: метрики розроблені з урахуванням об'єктно-орієнтованих систем. Для проектів, що використовують інші парадигми або мають унікальні вимоги дизайну, ці метрики можуть бути менш відповідними.

– складнощі в інтерпретації: інтерпретація конкретних значень метрик може бути складною. Наприклад, визначення того, що вважається "добрим" або "поганим" значенням для нестабільності чи абстракції може змінюватися в залежності від конкретного контексту проекту.

– обмеження статичного аналізу: метрики Мартина базуються на статичному аналізі, що означає, що вони не враховують динамічних аспектів системи. Це обмеження може впливати на точність оцінки, особливо в системах з складними поведінковими властивостями в час

Метрики Чідамбера та Кемерера ґрунтуються на аналізі методів класу, дерева успадкування.

Переваги:

– кількісне вимірювання: метрики надають кількісні показники для різних аспектів об'єктно-орієнтованого проектування програмного забезпечення, що дозволяє розробникам та керівникам проекту оцінювати якість коду та вести його моніторинг;

– виявлення поганих практик кодування: ці метрики можуть допомагати виявляти потенційні проблеми з поганими практиками кодування або архітектурними проблемами. наприклад, велике значення wmc може вказувати на потребу в рефакторингу для спрощення класу;

– акцент на ключових аспектах: метрики акцентують на важливих аспектах об'єктно-орієнтованого дизайну, таких як складність, глибина успадкування, зчеплення та співпраця, які є критичними для зручності підтримки та зрозуміння коду;

– бенчмаркінг: метрики можуть використовуватися для бенчмаркінгу та порівнянь між різними класами або різними версіями програмної системи.

– попередження про можливі проблеми: високі значення деяких метрик, таких як велике зчеплення чи низька співпраця, можуть служити попередженням про можливі проблеми з обслуговуваністю коду.

Недоліки метрик:

– чутливість до деталей реалізації: деякі метрики ck, особливо wmc та lcom, можуть бути чутливими до деталей реалізації. зміни в реалізації, які не впливають на функціональність, все ще можуть впливати на значення метрик.

– обмежений охоплює: метрики ck в основному фокусуються на метриках рівня класу та можуть не враховувати всі аспекти якості чи дизайну програмного забезпечення. вони можуть не забезпечувати повну картину загальної архітектури системи.

– залежність від інструментів: розрахунок деяких метрик ck, особливо тих, що стосуються зчеплення та співпраці, може вимагати використання складних інструментів статичного аналізу, і точність метрик залежить від точності цих інструментів.

– суб'єктивність порогів: визначення того, які значення вважаються прийнятними чи проблематичними для цих метрик, може бути суб'єктивним та залежати від контексту. немає універсально прийнятих порогів для "добрих" чи "поганих" значень.

– відсутність контексту: метрики самі по собі можуть не надавати повного розуміння контексту програмного забезпечення, його призначення чи конкретних викликів, з якими зіткнулася команда розробників. тому вони повинні використовуватися разом із іншими якісними оцінками.

– складність в інтерпретації метрик: Інтерпретація метрик вимагає глибокого розуміння архітектури програмного забезпечення та принципів дизайну. Неправильна інтерпретація або перевелика залежність від метрик без розуміння основних цілей дизайну може призвести до неоптимальних рішень.

Однією з головних проблем об’єктно-орієнтованих метрик є те, що вони зосереджені на взаємодії з аспектами, які характерні, власне для об’єктно-орієнтованого програмування, що суперечить універсальності метрики цикломатичної складності.

Гібридні метрики основані на використані простіших метрик та комбінуванні їх показників.

Переваги гібридних метрик:

– комплексний аналіз: гібридні метрики дозволяють проводити більш комплексний аналіз, враховуючи кілька вимірів системи. це може забезпечити голістичний погляд на якість програмного забезпечення;

– збалансована перспектива: комбінування різних метрик дозволяє отримати збалансовану перспективу, яка враховує різні аспекти розробки програмного забезпечення, такі як складність коду, підтримка та продуктивність.

– адаптабельність: гібридні метрики можуть бути адаптовані до конкретних вимог проекту, дозволяючи розробникам налаштовувати оцінку під характеристики та цілі конкретної системи;

– підвищена точність: комбінування різних метрик може підвищити точність оцінки, компенсуючи обмеження чи упередження, пов'язані з окремими метриками;

– кастомізація: розробники можуть налаштовувати гібридні метрики для віддачі пріоритету конкретним факторам на основі пріоритетів проекту, стандартів галузі чи цілей організації;

– чутливість до контексту: гібридні метрики можуть бути розроблені з урахуванням контексту, враховуючи унікальні характеристики проекту чи області застосування;

– ідентифікація проблем: вони можуть допомагати в ідентифікації проблем на різних рівнях процесу розробки програмного забезпечення, від проблем коду до архітектурних недоліків;

– гнучкість: гібридні метрики надають гнучкість у поєднанні кількісних та якісних оцінок, що дозволяє отримати більш нюансоване розуміння якості програмного забезпечення;

– інтеграція інструментів: вони можуть бути інтегровані в існуючі інструменти розробки та процеси, що спрощує включення метричних даних в робочий процес розробника;

– бенчмаркінг: гібридні метрики можуть сприяти порівнянню зі стандартами галузі чи кращими практиками, допомагаючи в оцінці відносної продуктивності системи.

Недоліки гібридних метрик:

– складність: комбінування кількох метрик може призвести до складних моделей, які важко розуміти та інтерпретувати;

– суб'єктивність: гібридні метрики можуть містити суб'єктивні елементи, особливо при включенні якісних оцінок, що може призвести до потенційних упереджень;

– витрати ресурсів: реалізація та підтримка гібридних метрик може вимагати значних ресурсів, включаючи час та експертизу, що може бути не завжди доцільним для всіх проектів;

– відсутність консистентності: якщо не ретельно розроблені, гібридні метрики можуть не мати консистентності між різними проектами чи розробницькими командами;

– складність вагових коефіцієнтів: надання відповідних вагових коефіцієнтів різним метрикам може бути складним і внести певний рівень суб'єктивності в процес оцінки;

– проблеми інтеграції даних: комбінування метрик з різних джерел може зіткнутися з проблемами інтеграції даних, особливо якщо дані не стандартизовані чи не сумісні;

– обмежена універсальність: гібридні метрики можуть бути менш універсальними, оскільки їх ефективність може залежати від конкретного контексту чи типу розроблюваного програмного забезпечення;

– стійкість до змін: розробники та учасники процесу можуть відчувати опір до впровадження гібридних метрик, якщо вони звикли до традиційних одновимірних метрик;

– складнощі у валідації: валідація ефективності гібридних метрик може бути складною, вимагаючи обширного тестування та перевірки в реальних умовах;

– занадто сильне покладання на метрики, навіть у гібридній формі, може призвести до тунельного погляду, знехтування іншими важливими аспектами розробки програмного забезпечення, такими як креативність та інновації.

Загалом гібридні метрики більше використовуються для поточних ситуаціях, до того, вони більше описуються як узагальнений підхід, ніж специфічна методика.

Вкупі з самим характером гібридності, використання гібридних метрик для модифікації метрики цикломатичної складності було визначено, як недоречне.

## 3.2 Метрика диференційованої цикломатичної складності

Згідно ретельного огляду існуючих рішень в області оцінки складності алгоритмів програмних застосунків, було прийнято рішення створювати нову метрику через модифікацію методики метрики цикломатичної складності та метрики Чепіна.

Реалізацію метрики можна представити у двох варіантах:

– вручну;

– програмно.

Програмно реалізацію буде розглянуто в наступному розділі. Виконання вручну по суті поєднує в собі обчислення двох окремих метрик.

Критерії, що використовуються для оцінки цикломатичної складності програми, визначаються через граф потоку керування, зразок якого представлено на Рисунку 3.1.

Шляхом проведення розрахунків цикломатичну складність можна визначити за формулою. Шляхом розрахунків цикломатичну складність можна визначити за допомогою формули:

$$V(G) = e - n + 2p, \tag{3.1}$$

де $e$ – кількість дуг,

$n$ – кількість вершин,

$p$ – компонент зв’язності.

Значення компоненти зв'язності можна інтерпретувати як кількість ребер, які необхідно додати для того, щоб граф потоку виконання програми став сильно зв'язаним. Граф називається сильно зв'язаним, якщо будь-які дві його вершини є взаємно досяжними.

Для графів коректних програм, тобто таких, що не мають недосяжних вершин від точки входу та з кожної вершини якої можна дістатись до кінцевої, сильна зв'язаність досягається шляхом додавання ребра між кінцевою вершиною та початковою.

Число V(G) визначає кількість лінійно незалежних контурів у сильно зв'язаному графі, що стосується коректних програм p=1. Таким чином, формула спрощується до:

$$V(G) = e - n + 2, \quad (3.2)$$

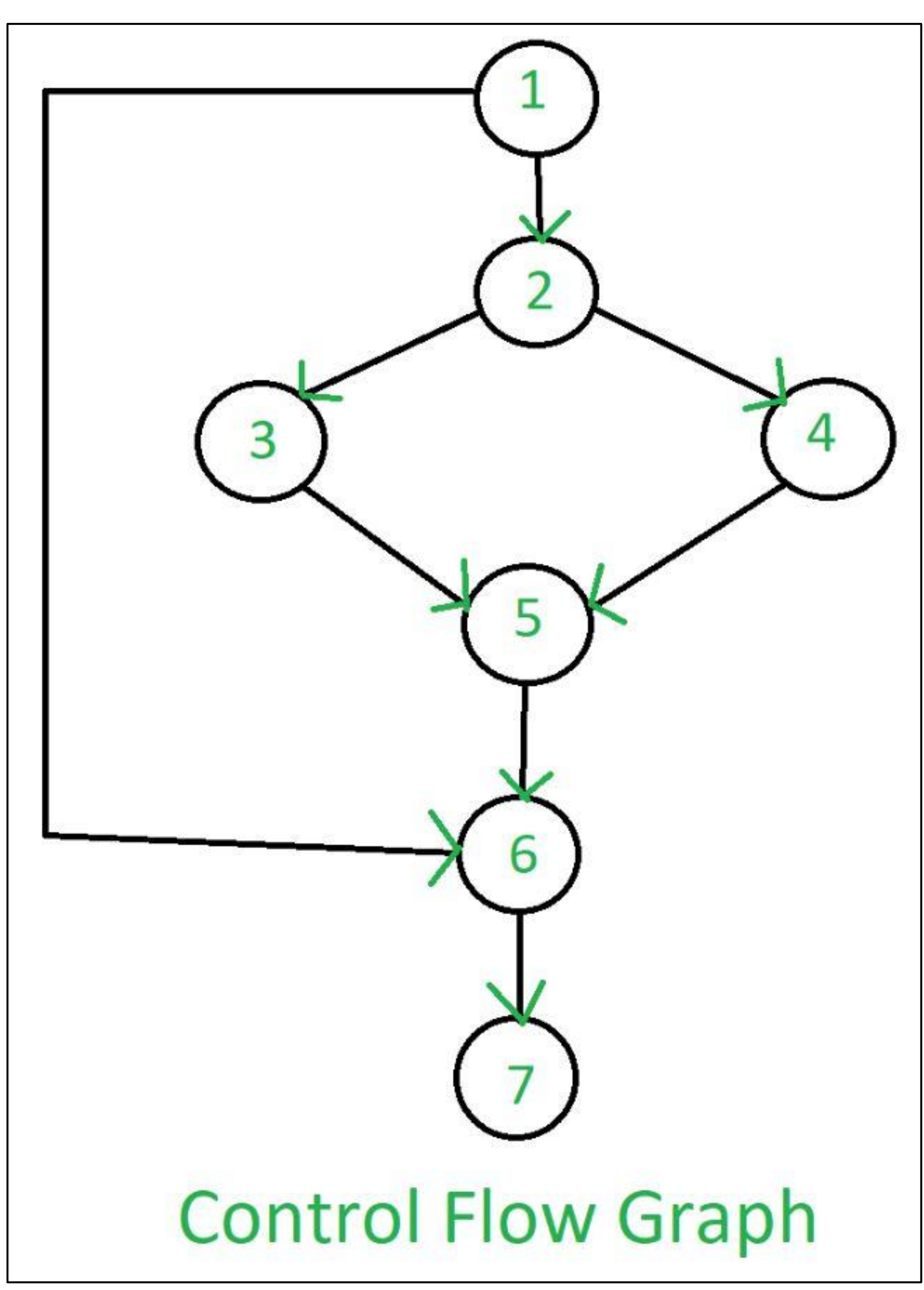



Рисунок 3.1 – Приклад потоку керування

Однак, слід зауважити, що традиційно граф потоку керування описує усі інструкції в коді. Для метрики Чепіна пропонується відображення графу, де вершинами вказуються контролюючі інструкції (умови, цикли) та методи.

Сама по собі метрика Чепіна обчислюються за формулою:

$$Q = a_1 \times P + a_2 \times M + a_3 \times C + a_4 \times T, \quad (3.3)$$

де $a_1$, $a_2$, $a_3$, $a_4$ – вагові коефіцієнти,

$P$ – кількість змінних, що вводяться, для розрахунків і для забезпечення виведення,

$M$ – кількість змінних, які модифіковані або створювані всередині програми,

$C$ – кількість змінних, що беруть участь в управлінні програмним модулем (керуючі змінні),

$T$ – кількість змінних, які не використовуються в програмі («паразитні»).

Згідно думки автора метрики, вагові коефіцієнти, становлять: $a_1$=1, $a_2$=2, $a_3$=3, $a_4$=0,5.

Таким чином, запропонована метрика диференційованої цикломатичної складності визначається наступним чином:

$$QD(G) = [V(G), Q_a], \quad (3.4)$$

де $V(G)$ – цикломатична складність,

$Q_a$ – значення загальної середньої складність програмних модулів, яка визначається за формулою:

$$Q_a = \sum_{0}^{i} Q_i, \quad (3.5)$$

де $Q_i$ – середня складність програмних модулів на $i$-тому шляху виконання програми:

$$Q_i = \frac{\sum_0^n Q_{ij}}{n}, \tag{3.6}$$

де $Q_{ij}$ – значення метрики Чепіна для $j$-того програмного модуля на $i$-тому шляху виконання програмного додатку,

$n$ – кількість програмних модулів, з яких складається шлях виконання програмного додатку.

З урахуванням того, що кількісна характеристика отримується в процесі обчислення, можливо і розширене представлення показника диференційованої цикломатичної складності:

$$\mathrm{Q}D(G) = [V(G), n, Q_a\ ], \tag{3.7}$$

Використання метрики Чепіна дозволяє розширено використати методику графу потоку керування. В той час, коли звичайний граф демонструє шлях, по якому можна виконати алгоритм програмного модуля, через метрику Чепіна можна вичислити складність алгоритму в кожній вершині графа. В якості прикладу подано рисунок 3.2:

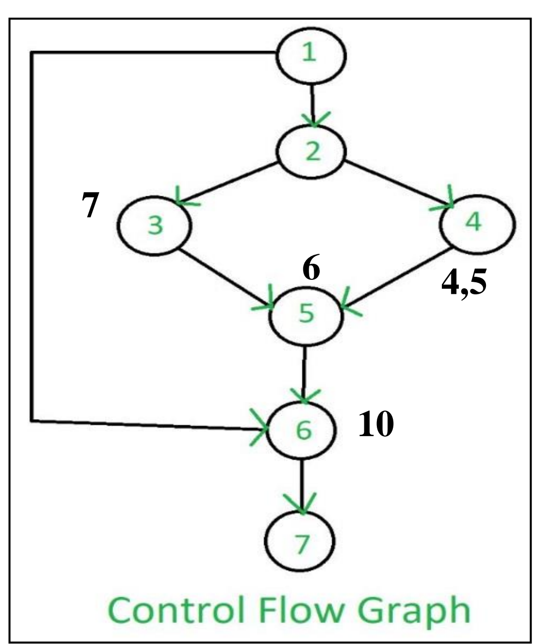



Рисунок 3.2 – Приклад модифікованого графу потоку керування. Число всередині верши – порядкове, біля неї – показник міри Чепіна

Як видно, граф потоку керування представлений вже в модифікованій формі, яку описано вище. Оскільки метрика Чепіна охоплює саме методи алгоритму, оцінки методу не позначаються для всіх вершин.

Така відображення надає можливість наочно не тільки оцінити структуру алгоритму і шляхи його виконання, але й побачити, у яких випадках алгоритм зустрічає найбільш складним.

Диференційована цикломатична складність отримала в спадок і декілька підходів обчислення від звичайної цикломатичної складності:

– «модифікована» цикломатична складність – розглядає не кожне розгалуження оператора множинного вибору (switch, case), а весь оператор як єдине ціле. В такому випадку показник Чепіна на вершині грфау розраховується як сумарний для всього змісту розгалуження;

– «сувора» цикломатична складність – містить логічні оператори у цьому випадку. На графі це буде виглядати як розгалуження на кількість гілок, рівну кількості логічних операторів, для кожної з яких надається показник Чепіна;

– «актуальна» складність – визначається як кількість незалежних шляхів, що проходить код при тестуванні;

У третьому розділі було обґрунтовано використання метрики цикломатичної складності в якості основи для утворення нової метрики. Були докладно розглянуті переваги та недоліки метрик оцінки складності алгоритмів програмних додатків. На основі цього була розглянута можливість використання кожної з метрик в якості елементу для модифікації цикломатичної складності. В результаті була обрана метрика Чепіна, на основі якої була модифікована метрика цикломатичної складності, що дозволило створити метрику диференційованої цикломатичної складності.

# 4 ПРАКТИЧНА РЕАЛІЗАЦІЯ НОВОЇ МЕТРИКИ

## 4.1 Структура та призначення модулів програми, їх взаємозв'язок

Практична реалізація метрики диференційованої цикломатичної складності програмним методом також успадкована від звичайної цикломатичної складності. Згідно теорії цикломатичної складності, як правило, при обчисленні цикломатичної складності логічні оператори не приймаються до уваги, допускається також спрощений підхід, згідно з яким власне побудова графа не проводиться, а показник визначається на підставі підрахунку кількісті операторів керуючої логіки (if, switch і т. д.) і можливої кількості шляхів виконання програми.

Таким чином може бути два варіанти реалізації метрики:

– повна;

– спрощена.

При повній реалізації від програми очікується наступний функціонал:

– формування модифікованого графу потоку керування;

– можливість відокремленого відображення кожного шляху проходження алгоритму з оцінкою Чепіна;

– власне розрахунок метрики диференційованої цикломатичної складності на основі визначення інструкцій коду в якості вершин графу потоку керування, а також шлях переходу між ним в якості дуг.

Згідно спрощеного підходу, робота алгоритму розділена на два елементи:

– підрахунок операторів керуючої логіки: умов та циклів;

– підрахунок змінних за методикою Чепіна: розділених на чотири групи з власними критеріями;

– розрахунок середнього значення метрики Чепіна серед класів: сумарне значення метрики розділене на кількість методів в класі.

Визначимо структуру програмного додатку програмного забезпечення визначено різними способами. Але їх можна звести до двох категорій:

– структура взаємопов'язаних елементів;

– певний набір рішень та принципів, за якими виконується створення програмного забезпечення.

Для подальшої роботи використано об’єднане поняття. Архітектура програмного забезпечення – це комплекс структурних елементів, їх взаємозв’язків та уявлень, що  визначають основні властивості розроблюваного програмного забезпечення.

Існує немала кількість архітектур:

– монолітна архітектура: усі компоненти програми розглядаються як єдина, велика система, зазвичай використовується для невеликих або середніх застосунків.

– рівнева архітектура: розділяє програму на рівні (наприклад, представлення, бізнес-логіка, доступ до даних), кожен рівень може взаємодіяти тільки з певними іншими рівнями.

– клієнт-серверна архітектура: розділення програми на клієнтські та серверні частини, які взаємодіють через мережу, дозволяє забезпечити розподілену обробку та покращену масштабованість.

– мікросервісна архітектура: розбиває програму на невеликі, незалежні мікросервіси, які взаємодіють між собою, сприяє гнучкості, швидкості розробки та розгортанню.

– схема "події-реакція": компоненти системи реагують на події, які вони отримують, а не чекають на запити, дозволяє створювати реактивні та асинхронні системи.

– інтерфейс-орієнтована архітектура: визначається основними інтерфейсами між компонентами, а не структурою чи логікою. забезпечує взаємозамінність компонентів, що реалізують однакові інтерфейси.

– архітектура "чиста або доменно-орієнтована": акцентує увагу на доменній логіці та високорівневих концепціях, забезпечує легку розширюваність і підтримує чистоту коду.

– середовище виконання архітектура: фокусується на взаємодії компонентів під час виконання програми, зокрема важлива для систем, які мають різноманітні розподілені компоненти.

Враховуючи характер реалізованої програми, очевидно, що буде розроблюватися додаток з монолітною архітектурою. В такій архітектурі одна програма, одна кодова база. Розробка, тестування та розгортання можуть бути більш простими порівняно з розподіленою системою.

Взаємодія з додатком представлена двома діями:

– підключення файлу з кодом;

– отримання спрощеної метрики диференційованої цикломатичної складності.

У якості інтерфейсу буде використовано консольний текстовий інтерфейс для взаємодії з користувачем. В цьому інтерфейсі користувач використовує командний рядок для введення команд та отримання виводу у текстовому форматі.

Давайте розглянемо взаємодію користувача з розроблюваним програмним забезпеченням за допомогою діаграм послідовності, які графічно відображають взаємодію об'єктів системи у часі.

Почнемо з процесу вибору файлу. Користувач запускає програму, яка автоматично відкриває консоль і взаємодіє з програмою «Провідник». Ця програма надає доступ до файлової системи комп'ютера, де користувач обирає файл з кодом. Обраний файл надсилається в програму у вигляді тексту. Важливо, щоб цей файл мав розширення .sln. Якщо файл не відповідає необхідному формату, виводиться відповідне повідомлення. Цей процес можна ілюструвати на діаграмі послідовності, представленій на Рисунку 4.1.

Наступною дією є власне виконання обчислень. Зчитуючи отриманий код, додаток автоматично отримує з коду окремі методи та кількість екземплярів, після чого з коду отримуються екземпляри змінних. На основі цих даних, програмний додаток самостійно, без участі користувача, виконує розрахунок показників метрики диференційованої цикломатичної складності. Цей процес можна ілюструвати на діаграмі послідовності, представленій на рисунку 4.2.

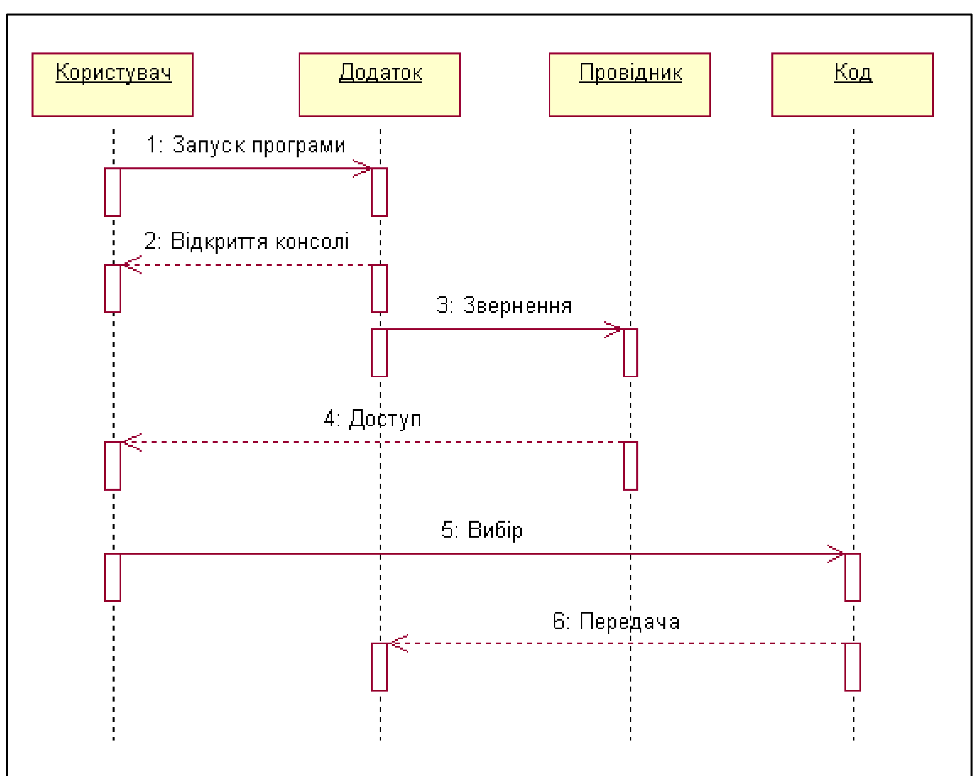



Рисунок 4.1 – Діаграма послідовності вибору файлу

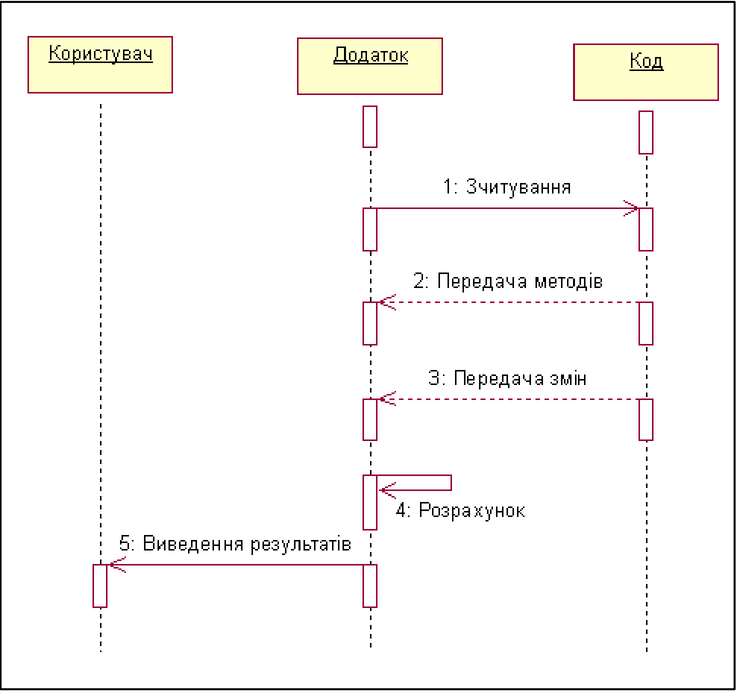



Рисунок 4.2 – Діаграма послідовності виконання обчислень

Проводячи проектування модулів програмної системи, будуть описані способи роботи модулів і способи взаємодії між собою.

Вже були розглянуті функції розроблюваного програмного забезпечення, а також структура графічного інтерфейсу і функціональна логіка його елементів. В результаті можна виділити наступні модулі:

– запуск програми;

– вибір цільового файлу;

– обчислення метрики.

Далі буде продемонстрована структуру даних всередині розроблюваного програмного забезпечення засобами діаграми класів. Слід зазначити, як згадувалось вище, розроблюваний додаток представлений монолітною структурою. Враховуючи поставлену задачу, весь функціонал буде розміщений в єдиному модулі – файлі класу. Тому на діаграмі класів в якості модулів представлені методи коду. Утворена діаграма класів представлена на рисунку 4.3.

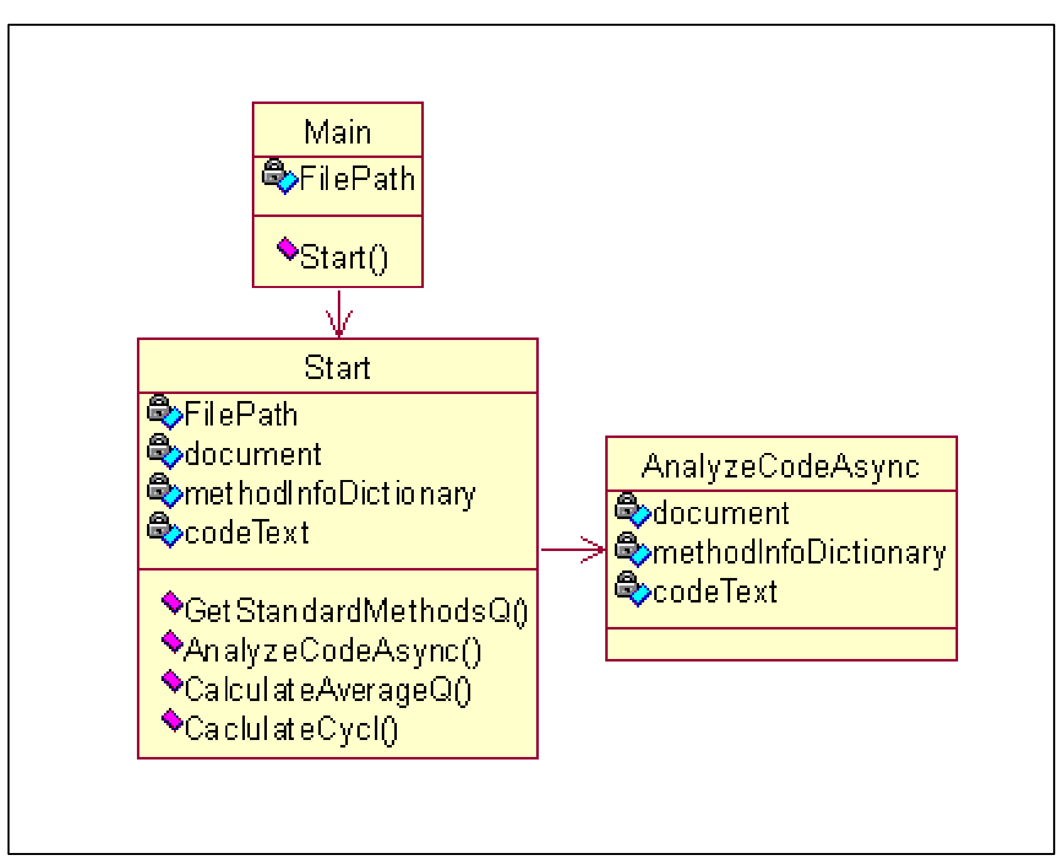



Рисунок 4.3 – Діаграма взаємодії модулів програмного додатку

## 4.2 Аналіз ти вибір технологій для реалізації програмної системи

Як вже було зазначено, нова метрика позиціонує себе, як нечутлива до мови програмування. Тому вибір засобів практичної реалізації характеризується особистими уподобаннями.

Всі популярні мови програмування можуть бути використані для розробки програмних додатків. Здебільшого, в якості засобу зв’язку коду програми та графічного інтерфейсу користувача виступають платформи та середовища розробки, які надають необхідні бібліотеки.

В якості мови програмування обрано мову програмування C# (Сі-Шарп) [23], яка є об'єктно-орієнтованою мовою з безпечною системою типізації для платформи .NET. Ця мова є потужним інструментом для розробки різноманітних програмних додатків, розвиваючись від мови C++ та Java.

Дана мова програмування має строгу статичну типізацію, підтримує поліморфізм, перевантаження операторів, вказівники на функції-члени класів, атрибути, події, властивості, винятки, коментарі.

Об'єктно-орієнтований підхід C# дозволяє використовувати готові конструкції, полегшуючи розробку та покращуючи читабельність коду, хоча ця універсальність може впливати на швидкодію додатків.

Окрім того, це – продукт компанії Microsoft, з чого можна зробити висновки щодо поширеності та актуальності мови програмування. А також про наявність великої спільноти розробників, до якої можливо звернутися у процесі праці с даною мовою програмування.

Інші немалозначні переваги мови пов’язані з далі вказаними інструментами, з якими зазвичай Сі-Шарп і застосовується.

C# працює на базі платформи Microsoft .NET. .NET Framework – це програмна платформа від компанії Microsoft, яка підтримує розробку та виконання програмних додатків різних типів: від простих додатків до веб-служб [24-25]. Архітектура платформи ґрунтується на використанні середовища виконання

Common Language Runtime (CLR), де код мов програмування перетворюється в проміжний універсальний код, який потім компілюється у машинний код.

.NET Framework складається з двох ключових компонентів. Перша частина включає заздалегідь написаний код (офіційно відомий як SDK, Dev Packs або "Пакети розробника"). Друга частина включає середовище виконання, яке інтерпретує код .NET Framework в команди для операційної системи, дозволяючи запускати програми, написані з використанням цієї платформи.

Основні характеристики платформи включають:

– кросплатформенність – платформа здатна підтримувати більшість операційних систем лінійки Windows, підтримувати розробку під Linux та MacOS, та навіть розробку мобільних додатків;

– мультимовність – за рахунок трансформацій у проміжний код Common Intermediate Language (CIL). Розробник отримує свободу у виборі мови програмування. А, за необхідності, така структура реалізації коду дозволяє створювати проекти на кількох мовах. Найбільш популярними мовами, що підтримуються, є C#, VB.NET, C++, F#. У разі необхідності є окремі проекти, завдяки яким, стає можлива розробка під цю платформу на інших мовах;

– велика бібліотека класів та технологій – платформа підтримує величезну кількість готових бібліотек, за допомогою яких можливе виконання широкого спектру задач. Завдяки спеціальному менеджеру пакетів nuget користувач позбавляється необхідності реалізовувати логіку роботи програми на низькому рівні.

Головним недоліком відносно платформи можна вважати те, що без встановлення .NET Framework, фактично, більшість сучасних програм просто не будуть працювати на комп’ютері.

Проект Roslyn є частиною .NET Foundation разом з іншими проектами, такими як .NET Runtime [26]. Roslyn є відкритою реалізацією компіляторів C# та Visual Basic, розробленою Microsoft, з багатими API аналізу коду. Roslyn є просто кодовою назвою для .NET Compiler Platform. Roslyn може генерувати попередження у вашому коді ще до того, як ви завершите рядок, тому його доречно

називають аналізатором. Roslyn також може запропонувати автоматичне виправлення помилок.

Аналізатори Roslyn використовуються для перевірки вашого коду на C# чи Visual Basic на стиль, якість, дизайн, підтримку та інші аспекти. Обробка вихідного коду складається з трьох основних етапів: аналіз синтаксису, семантичний аналіз та генерація коду. Результат кожної фази діє як вхід для наступної, але ці проміжні результати та внутрішні структури даних не видимі/доступні для користувача. Таким чином, розробники розглядають компілятор як чорну скриньку.

Roslyn API використовує традиційну для компілятора архітектуру конвеєра, надаючи доступ до кожного кроку аналізу вихідного коду та обробки:

– API синтаксичного дерева показує лексичну та синтаксичну структуру вашого коду, включаючи форматування та коментарі. Останні два є важливими, оскільки вони допомагають маніпулювати кодом та зберігати форматування та коментарі коду в незмінному вигляді;

– Символьне API показує вам таблицю символів, що містить імена, оголошені в вашому вихідному коді, а також ті, що походять від зазначених збірок без відповідного вихідного коду;

– API аналізу прив'язки та потоку розкриває повну семантичну модель вашого коду, яка стає доступною після етапу прив'язки. Ці API містять всю інформацію про код, яка необхідна для генерації бінарних файлів, включаючи помилки та попередження, виявлені в коді;

– API видачі надає доступ до служб для формування байткоду IL.

Обсяг Roslyn йде далеко за межі просто API компілятора. Кодова база Roslyn зараз використовується в двох сценаріях з майже протилежними вимогами. З точки зору компілятора, велика пропускна здатність та правильність - це найважливіші фактори, тоді як при розгляданні інтерактивного редактора, відзивчивість та виявлення помилок є вищими пріоритетами. Roslyn вдалося досягти порівняної, якщо не кращої, продуктивності, ніж Visual Studio 2013, в обох наведених випадках. Проте це супроводжується деякими недоліками, а саме всі структури даних є незмінними, тобто їх неможливо змінити.

Кожен раз, коли розробник вносить зміну у файл, створюється нова копія всіх структур даних, залишаючи попередню версію незмінною. Це показує великий рівень паралельності та конкурентноспроможності в двигуні Roslyn, запобігаючи будь-яким умовам гонки, які могли виникнути у будь-якому випадку. Це вважається величезною перевагою аналізатора Roslyn. З метою продуктивності ці операції дуже оптимізовані та можуть бути повторно використані в межах існуючих структур даних наскільки це можливо.

Нарешті, в якості середовища розробки була обрана Microsoft Visual Studio 2019, у якій об’єднані всі засоби розробки, описані вище. Visual Studio – це лінійка продуктів, започаткована наприкінці минулого століття, як засіб полегшення роботи з мовами, на кшталт C++. Згодом продукт набув функціоналу та поширення і зараз розповсюджується у різних версіях для різних типів користувачів (хоча фактична різниця між версіями мінімальна).

Microsoft Visual Studio – це середовище програмування, розроблене компанією Microsoft. MVS дозволяє створювати кросплатформені проекти різними мовами програмування, такими як Visual Basic, Visual C#, Visual C++, Visual F# та інші. У Visual Studio розробник може проводити розробку веб-сайтів, веб-служб, писати консольні програми, а також програми з графічним інтерфейсом. Інтерфейс середовища розробки представлено на рисунку 4.4.

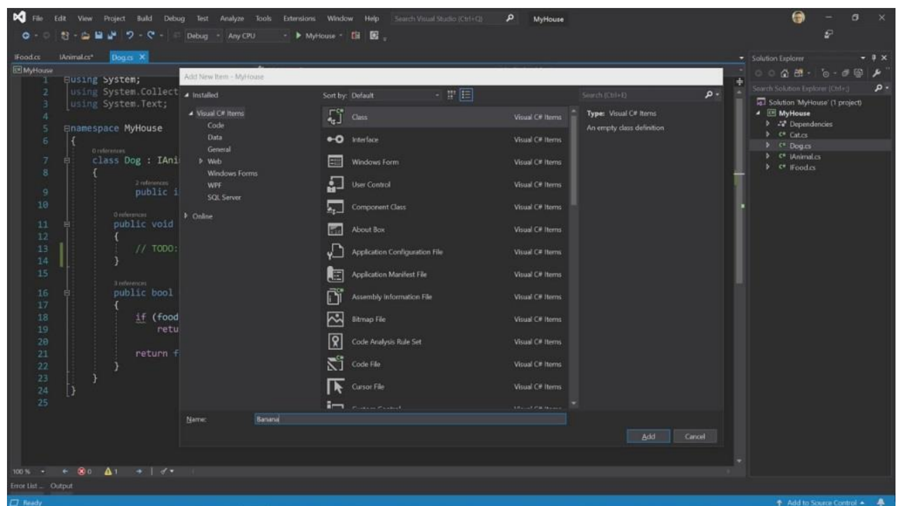


Рисунок 4.4. – Зразок інтерфейсу Microsoft Visual Studio

Також MVS підтримує різноманітні доповнення. Наприклад, згаданий до платформи .NET Framework менеджер пакетів nuget стає доступний через інтерфейс середовища розробки, завдяки чому стає можливе доповнення стандартних бібліотек новими, необхідними для конкретного проекту.

Як і будь-яке середовище розробки, до складу Visual Studio входять:

– редактор коду, який здатен підсвічувати синтаксис, помилки, типи даних та інше;

– компілятор та інтерпретатор для перетворення мов програмування у машинний код. Visual Studio використовує обидва в залежності від проекту;

– засоби автоматизації складання;

– дебаггер, або відладник для дослідження коду на предмет помилок

Перевагами середовища розробки MVS вважаються:

– підтримка великої кількості мов програмування;

– вбудований веб-сервер, що полегшує розробку веб-додатків;

– автоматичне додавання коду до стандартних елементів керування;

– автоматичне форматування коду під час його написання (підсвічування, вставка відступів);

– велика спільнота, до якої можна звернутися по допомогу;

– програма-інсталятор, який надає можливість встановлення, окрім самого середовища розробки, додаткового програмного забезпечення, необхідного для роботи.

До значних недоліків виділяють MVS відносять:

– масивність – середовище розробки разом з додатковим програмним забезпеченням та інструментарієм, навіть для однієї області розробки, займає немало місця, особливо на системному диску, через що можуть виникнути проблеми з працездатністю робочої машини;

обмеженість офіційними робочими платформами Windows та Mac.

## 4.3 Розробка програмних модулів

Як вже було зазначено, призначення програмного додатку та монолітна архітектура призвели до створення додатку в якості єдиного класу. Тому в якості модулів розглядаються великі методи.

Все починається з запуску додатку. Після стандартного виведення консолі, додатку автоматично звертається до файлової системи, через яку очікується вибір користувачем файлу рішення Visual Studio формату .sln.

```
[STAThread]
static void Main()
{
  var openFileDialog = new OpenFileDialog
  {
      Title = "Виберіть файл рішення Visual Studio",
      Filter = "Файли рішень Visual Studio (*.sln)|*.sln",
      Multiselect = false
  };

      if (openFileDialog.ShowDialog() == DialogResult.OK)
      {

            string filePath = openFileDialog.FileName;
            Start(filePath).Wait();
      }
      else
      {
      Console.WriteLine("Документ не знайдено");
      }
      Console.ReadKey();
}
```

Далі ініціалізується асинхронний метод, який приймає шлях до файлу рішення Visual Studio. Основна функціональність цього методу включає в себе:

– реєстрацію MSBuildLocator (реєстрація робочого простору MSBuild);

– створення робочого простору MSBuild з використанням MSBuildWorkspace;

– відкриття рішення за вказаним шляхом;

– зчитування тексту з файлу рішення;

– перебір усіх проектів та документів у рішенні та аналіз коду за допомогою методу AnalyzeCodeAsync;

– розрахунок середнього значення Q та числа цикломатичної складності для кожного документа та виведення результатів на основі словника методів.

```
static async Task Start(string filePath)
{
            MSBuildLocator.RegisterDefaults();

            using var workspace = MSBuildWorkspace.Create();

            var solution = await workspace.OpenSolutionAsync(filePath);

            List<double> allQValues = new List<double>();

            string codeText = File.ReadAllText(filePath);

            foreach (var project in solution.Projects)
            {
                foreach (var document in project.Documents)
                {
                    Dictionary<string, double> methodInfoDictionary = 
GetStandardMethodsQ();
                    Console.WriteLine($"Документ: {document.Name}");
                    var methodInfo = await AnalyzeCodeAsync(document, 
methodInfoDictionary, codeText);
                    double avgQ = CalculateAverageQ(methodInfo);
                    int cycl = CalculateCycl(filePath);
                    Console.WriteLine($"Середнє значення Q: {avgQ:F2}");
                    Console.WriteLine($"Цикломатична складність: {cycl}");
                    Console.WriteLine("----------------------------");
                    allQValues.Add(avgQ);
                }
            }

            static double CalculateAverageQ(Dictionary<string, double> 
methodInfoDictionary)
            {
                return methodInfoDictionary.Any() ? 
methodInfoDictionary.Values.Average() : 0;
            }

            static int CalculateCycl(string filePath)
            {
                int num = 0;
                using (StreamReader reader = new StreamReader(new 
FileStream(filePath, FileMode.Open, FileAccess.Read)))
                {
                    while (reader.Peek() >= 0)
                    {
                        string str = reader.ReadLine();
                        if (str.Contains("if(") || str.Contains("if ("))
                        {
                            num++;
                        }
                        else if (str.Contains("else if(") || str.Contains("else(") 
||
                        str.Contains("else if (") || str.Contains("else ("))
                        {
                            num++;
                        }
                        else if (str.Contains("case"))
```

```
                        {
                            num++;
                        }
                        else if (str.Contains("while (") ||
str.Contains("while("))
                        {
                            num++;
                        }
                        else if (str.Contains("for(") || str.Contains("for ("))
                        {
                            num++;
                        }
                        else if (str.Contains("foreach (") ||
str.Contains("foreach("))
                        {
                            num++;
                        }
                        else continue;
                    }
                }
                return num;
            }


            static Dictionary<string, double> GetStandardMethodsQ()
            {
                var standardMethodsQ = new Dictionary<string, double>
                {
                    {"Console.WriteLine", 1.0},
                    {"Console.ReadLine", 1.0},
                    {"Console.ReadKey", 1.0},

                };
                return standardMethodsQ;
            }
}
```

Далі викликається ще одни асинхронний метод, який приймає об'єкт Document, словник methodInfoDictionary та текст коду. Цей метод:

– аналізує використання та модифікацію змінних у методах коду.

– використовує регулярні вирази та інші умови для визначення параметрів P, M, C, T для кожного методу.

– обчислює значення Q за формулою, яку ви надали в попередньому запитанні.

– оновлює значення словника methodInfoDictionary для відповідного методу із розрахованим значенням Q.

– повертає оновлений словник.

```
static async Task<Dictionary<string, double>> AnalyzeCodeAsync(Document document,
Dictionary<string, double> methodInfoDictionary, string codeText)
```

```
        {
            var referencedLibraries = new List<string>();
            var usedMethods = new Dictionary<string, int>();

            var syntaxRoot = await document.GetSyntaxRootAsync();

            var usingDirectives =
syntaxRoot.DescendantNodes().OfType<UsingDirectiveSyntax>();

            var methodDeclarations =
syntaxRoot.DescendantNodes().OfType<MethodDeclarationSyntax>();

            foreach (var methodDeclaration in methodDeclarations)
            {
                var methodName = methodDeclaration.Identifier.ToString();
                var methodBody = methodDeclaration.Body;

                if (methodInfoDictionary.ContainsKey(methodName))
                {
                    continue;
                }

                var variables =
methodBody.DescendantNodes().OfType<VariableDeclarationSyntax>();

                foreach (var variable in variables)
                {
                    var variableName =
variable.Variables.First().Identifier.ToString();

                    int P = 0, M = 0, C = 0, T = 0;

                    if (codeText.Contains($"({variableName})") ||
codeText.Contains($"return {variableName}"))
                        P++;

                    if (codeText.Contains($"{variableName} = "))
                        M++;

                    if (IsInsideConditionalStatement(codeText))
                        C++;

                    if (!IsUsedOrModified(variableName, codeText))
                        T++;

                    double Q = P + 2 * M + 3 * C + 0.5 * T;

                    methodInfoDictionary[methodName] = Q;
                }

                static bool IsInsideConditionalStatement(string code)
                {
                    return code.Contains("if") || code.Contains("while") ||
code.Contains("switch") || code.Contains("for") || code.Contains("foreach");
                }

                static bool IsUsedOrModified(string variableName, string code)
                {
                    return Regex.IsMatch(code, @"\b" + variableName + @"\b");
                }
            }
            return methodInfoDictionary;
}
```

## 4.4 Вибір та обґрунтування методів тестування додатку

Програмне тестування - це метод визначення відповідності фактичного результату роботи розроблюваного програмного забезпечення передбаченим вимогам. Цей метод надає інформацію про наявність дефектів у програмі і включає в себе використання ручних або автоматизованих інструментів, які дотримуються завчасно встановлених алгоритмів для оцінювання критеріїв програми.

Основна мета тестування - знаходження дефектів програмного продукту в межах стандартного процесу розробки програмного забезпечення. Крім виявлення дефектів, тестування також включає коригування програми для усунення недоліків.

Завдання тестування включають в себе пошук дефектів на різних етапах життєвого циклу додатку, перевірку ефективності усунення виявлених дефектів, а також дослідження на предмет виявлення нових дефектів, що можуть виникнути внаслідок виправлення вже відомих.

Задачі тестування включають в себе пошук дефектів моделі системи, дефектів кодування, помилок взаємодії системи з оточенням, дефектів інтеграції програмного забезпечення, а також оцінку продуктивності системи та її стійкість до перевантажень.

Тестування є невід'ємною частиною життєвого циклу програмного забезпечення, включаючи проектування, розробку і етап експлуатації. Воно також пов'язане з управлінням вимогами і змінами для переконання у відповідності програм заявленим вимогам.

Програмне забезпечення вважається коректним, якщо воно відповідає наступним критеріям:

– при коректних даних програма генерує правильну відповідь;

– відхилення від некоректних вхідних даних;

– відсутність «зависань» та неочікуваного завершення роботи програми незалежно від вхідних даних;

– функціонування програми не обмежено часовими рамками;

– програма працює без збоїв і виконує всі необхідні функції в повному обсязі.

Переваги тестування програмного забезпечення включають:

– зменшення витрат на програмний продукт за рахунок вчасного тестування та усунення дефектів;

– нивелювання ризиків витоку інформації;

– забезпечення користі користувачам як головної мети проекту;

– дотримання програмного продукту визначеним специфікаціям, бізнес-вимогам та вимогам до функціоналу;

– забезпечення кращого користувацького досвіду через ui/ux тестування.

Тестування проводиться за методиками "чорного ящика" та "білого ящика". Різноманіття підходів до тестування призводить до різноманітної класифікації, такої як функціональне тестування та нефункціональне тестування.

Функціональне тестування включає модульне, інтеграційне, системне та приймальне тестування. Модульне тестування перевіряє найменші функціональні елементи. Інтеграційне тестування перевіряє взаємодію об'єктів системи. Системне тестування проводиться на готовому продукті, а приймальне тестування - кінцевим користувачем.

Інтегроване середовище розробки Microsoft Visual Studio 2019 надає можливості автоматизації тестів за допомогою платформ MSTest, NUnit та xUnit, які підтримують технології Microsoft та Java/C#.

Усі ці платформи так чи інакше надають інструменти для проведення функціонального тестування, тому вибір для даного програмного додатку досить вільний. Специфікою тестування під час виконання даного проекту є те, що процес тестування виконувався вручну за допомогою виділення фрагментів коду та використання тимчасових графічних форм, без застосування засобів автоматизації.

З огляду структуру та розміри програмного застосунку, розробка окремих сценаріїв тестування не проводилося.

## 4.5 Оцінка методу диференційованої цикломатичної складності

У дипломній роботі було розроблено метрика диференційованої цикломатичної складності на основі модифікації метрики цикломатичної складності.

Практична апробація отриманих результатів показала, що удосконалений алгоритм дає змогу отримувати більше інформації про складність алгоритму програмного застосунку.

Завдяки нового підходу до цикломатичної складності можна уникати ситуацій, коли два алгоритми можуть мати однакову цикломатичну складність при різній фактичній, яка невідома, що показано на рисунку 4.5.

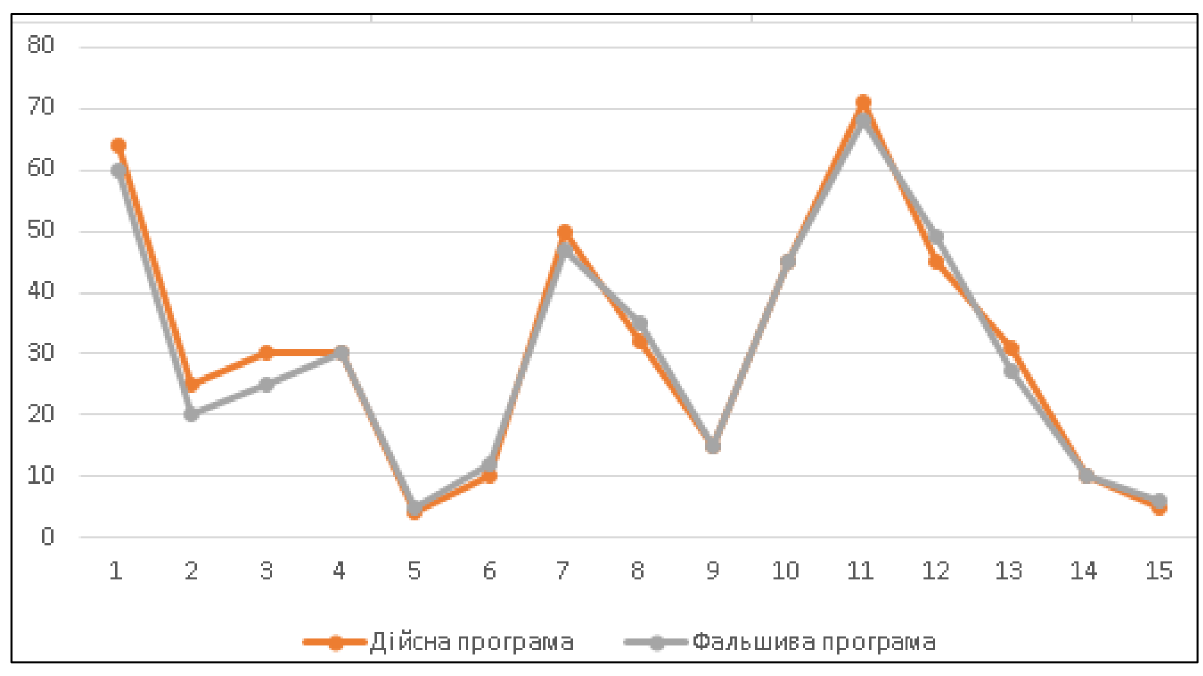



Рисунок 4.5 – Графік цикломатичних складностей довільних та «фальшивих» алгоритмів.

Оскільки міра Чепіна дозволяє вказати на складнішу структуру використання даних алгоритмом на рисунку 4.6 можна побачити більш об’єктивну різницю між алгоритмами. Ця властивість продемонстрована при використанні довільних алгоритмів, та спеціально написаних алгоритмів, які б штучно імітували

цикломатичну складність. Оскільки метрика оперує програмними модулями, в якості алгоритмів обиралися не повні додатки, а їх окремі класи.

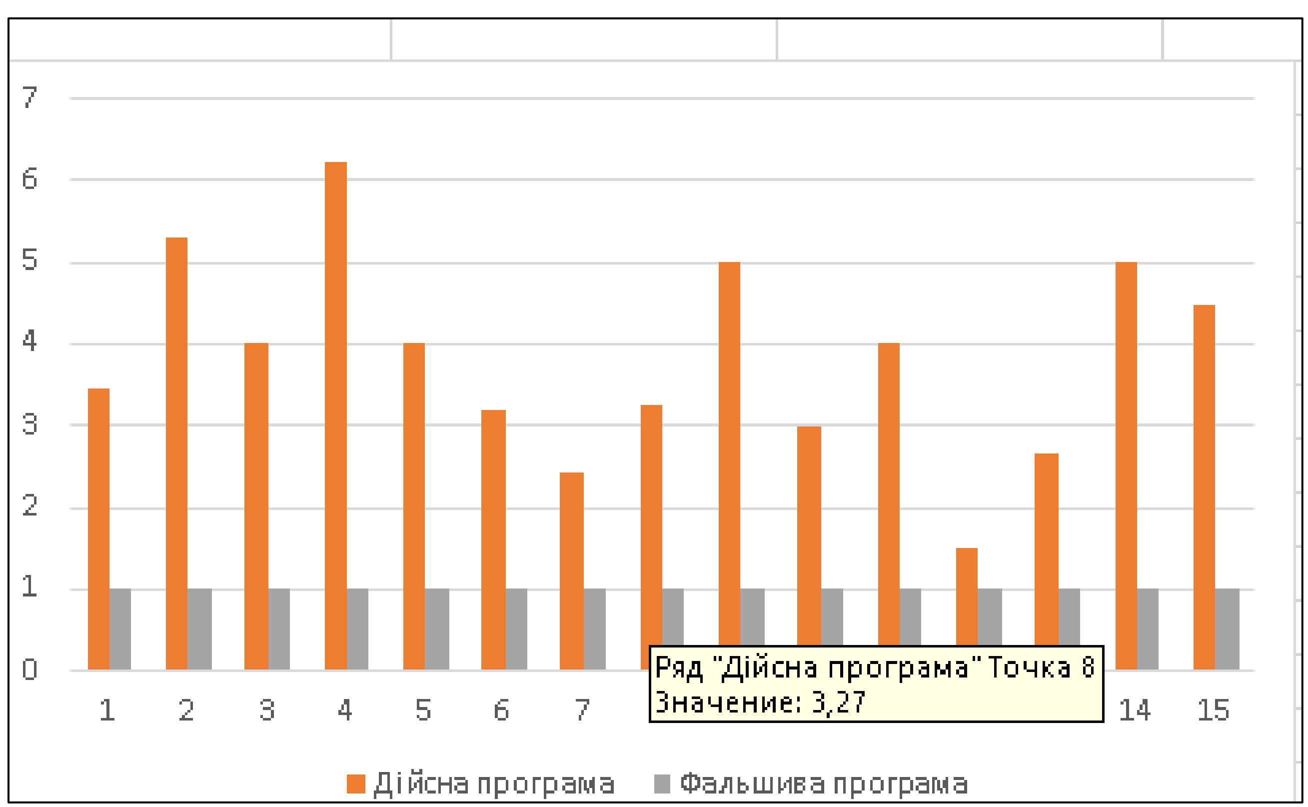



Рисунок 4.6 – Графік середніх мір Чепіна для довільних та «фальшивих» алгоритмів.

При використання метрики диференційованої цикломатичної складності отримується на 100% більше інформації, що дозволило при апробації покращити розпізнання різних за складністю алгоритмів у 53% випадків. Тобто, загальна результативність нової метрики диференційованої цикломатичної складності перевищує звичайну цикломатичну складність на 77%.

## 4.5 Інтеграція та налаштування програмного засобу

Для використання програмного забезпечення необхідно:

– операційна система Windows;

– .NET 5.0 або новіше.

Мінімальні системні вимоги залежать від програмного забезпечення, яке використовується при розробці алгоритмів програмних застосунків, але в будь-якому випадку необхідним є використання периферійних пристроїв: монітор, клавіатура, комп’ютерна миша.

За умови наявності додаткового програмного забезпечення, вказаного у вимогах, достатньо копіювання файлів системи на дисковий простір комп’ютера.

В четвертому розділі було проведено проектування модулів розроблюваного програмного забезпечення, визначена їх структура та взаємодії між ними. Також відбулася реалізація логіки додатку з використанням мови програмування, код якої описали виконання основних функцій. Були визначені вимоги до технічних та програмних засобів для запуску системи, а також описаний процес розгортання та встановлення системи. Була проведена апробація нової метрики, яка на практиці продемонструвала покращення в процесі визначення складності алгоритму.

Головними перевагами нової метрики є простота та доступність у використанні, а також – вирішення проблеми метрики цикломатичної складності, яка пов’язана з неможливістю об’єктивно розрізнити складність алгоритмів з однаковим показником метрики.

Основним недоліком методики є нечутливість до структури коду у тому сенсі, що метрики не розрізняє конструкції циклів та конструкції умов. Також програмний застосунок, представлений для роботи несе більше демонстративний характер, внаслідок чого алгоритм має обмеження при знаходженні цикломатичного числа (може помилково звернутися до конструкцій керування в текстових рядках) та обмежений словник методів (сам словник по замовчуванню потребує подальшого заповнення методами, які представлені в стандартних бібліотеках).

Програмний застосунок, як і метрика, можуть використовуватися будь-ким в процесі виконання розробки програмного продукту на всіх етапах, де фігурує опис алгоритму застосунку.

# ВИСНОВКИ

У даній кваліфікаційній роботі проаналізовано процес розробки програмного забезпечення, були дослідженні існуючі засоби оцінки складності алгоритмів програмних застосунків, розглянуті детально їх переваги та недоліки, на основі чого розглянута можливість модифікації метрики цикломатичної складності за допомогою методик однієї з цих метрик. Була запропонована метрика диференційованої цикломатичної складності, на основі опису якої була виконана практична реалізація.

Таким чином у першому розділі була розглянута структура процесу розробки програмного забезпечення. Це процес створення програмних продуктів, які включають в себе всі аспекти життєвого циклу програми, від концепції і розробки до тестування, удосконалення та підтримки. Процес складається з певного набору етапів, виконання яких визначається певним підходом, вибір якого з великого числа варіантів визначається характером проекту.

Отже, в другому розділі були розглянуті існуючі засоби обчислення складності алгоритмів програмного застосунку:

– часова складність;
– просторова складність;
– розмірно-орієнтовані метрики;
– метрики складності потоку керування даними;
– метрики складності потоку керування програмами;
– метрики складності потоку керування даними та програми;
– об'єктно-орієнтовані метрики;
– гібридні метрики.

Є велика кількість категорій, по яким розподілені ці засоби обчислення, і які вказують на можливість досліджувати код алгоритму з різни точок зору, і що вказує на те, що жодна з метрик не абсолютною, і що в кожній наявні як переваги, так і недоліки.

У третьому розділі було обґрунтовано використання метрики цикломатичної складності в якості основи для утворення нової метрики. Були докладно розглянуті переваги та недоліки метрик оцінки складності алгоритмів програмних додатків. На основі цього була розглянута можливість використання кожної з метрик в якості елементу для модифікації цикломатичної складності. В результаті була обрана метрика Чепіна, на основі якої була модифікована метрика цикломатичної складності, що дозволило створити метрику диференційованої цикломатичної складності.

В четвертому розділі було проведено проектування модулів розроблюваного програмного забезпечення, визначена їх структура та взаємодії між ними. Також відбулася реалізація логіки додатку з використанням мови програмування, код якої описали виконання основних функцій. Були визначені вимоги до технічних та програмних засобів для запуску системи, а також описаний процес розгортання та встановлення системи. Була проведена апробація нової метрики, яка на практиці продемонструвала покращення в процесі визначення складності алгоритму.

Головними перевагами нової метрики є простота та доступність у використанні, а також – вирішення проблеми метрики цикломатичної складності, яка пов’язана з неможливістю об’єктивно розрізнити складність алгоритмів з однаковим показником метрики.

Основним недоліком методики є нечутливість до структури коду у тому сенсі, що метрики не розрізняє конструкції циклів та конструкції умов. Також програмний застосунок, представлений для роботи несе більше демонстративний характер, внаслідок чого алгоритм має обмеження при знаходженні цикломатичного числа (може помилково звернутися до конструкцій керування в текстових рядках) та обмежений словник методів (сам словник по замовчуванню потребує подальшого заповнення методами, які представлені в стандартних бібліотеках).

Програмний застосунок, як і метрика, можуть використовуватися будь-ким в процесі виконання розробки програмного продукту на всіх етапах, де фігурує опис алгоритму застосунку.

Результати дослідження опубліковані у фаховому науковому виданні [27]. Копії наукових публікацій подані у додатку Б.

# ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ


1. Deep Learning Approach To Predict Software Development Life Cycle Model / J. Dhami та ін. *2021 International Conference on Advances in Computing, Communication, and Control (ICAC3)*, м. Mumbai, India, 3–4 груд. 2021 р. 2021. URL: https://doi.org/10.1109/icac353642.2021.9697271 (дата звернення: 11.12.2023).

2. Bhujang R. K., Dean S. V. Propagation of Risk across the Phases of Software Development. *2018 2nd International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)*, м. Palladam, India, 30–31 серп. 2018 р. 2018. URL: https://doi.org/10.1109/i-smac.2018.8653647 (дата звернення: 11.12.2023).

3. What is the Need of Software Engineering? - GeeksforGeeks. *GeeksforGeeks*. URL: https://www.geeksforgeeks.org/what-is-the-need-of-software-engineering/ (дата звернення: 11.12.2023).

4. ДСТУ ISO/IEC 12207:2016 Інженерія систем і програмного забезпечення. Процеси життєвого циклу програмного забезпечення (ISO/IEC 12207:2008, IDT) URL: http://online.budstandart.com/ua/catalog/doc-page?id_doc=71824 (дата звернення: 11.12.2023).

5. Етапи життєвого циклу розробки ПЗ. *Веб-студія розробки програмного забезпечення - IC Studio, Україна*. URL: https://icstudio.online/post/etapi-zhittyevogo-ciklu-rozrobki-pz (дата звернення: 11.12.2023).

6. Розробка ПЗ: моделі життєвого циклу, методи та принципи. *Evergreen - web розробка і діджиталізація бізнесу за допомогою AI продуктів*. URL: https://evergreens.com.ua/ua/articles/software-development-metodologies.html (дата звернення: 11.12.2023).

7. Черткова Е. А. Комп\'ютерні технології навчання. *Stud*. URL: https://stud.com.ua/174105/informatika/kompyuterni_tehnologiyi_navchannya.

8. What is SDLC? - Software Development Lifecycle Explained - AWS. *Amazon Web Services, Inc*. URL: https://aws.amazon.com/what-is/sdlc/?nc1=h_ls (дата звернення: 11.12.2023).

9. Software Development Life Cycle (SDLC) - GeeksforGeeks. *GeeksforGeeks*. URL: https://www.geeksforgeeks.org/software-development-life-cycle-sdlc/ (дата звернення: 11.12.2023).

10. What Is SDLC? Understand the Software Development Life Cycle. *Stackify*. URL: https://stackify.com/what-is-sdlc/ (дата звернення: 11.12.2023).

11. Популярні життєві цикли розробки ПЗ | Онлайн-курси від компанії QATestLab. *Онлайн-курси від компанії QATestLab | Головна сторінка*. URL: https://training.qatestlab.com/blog/technical-articles/popular-software-development-life-cycles / (дата звернення: 11.12.2023).

12. Стадії циклу розробки ПЗ. *QALight*. URL: https://qalight.ua/baza-znaniy/stadiyi-tsiklu-rozrobki-pz/ (дата звернення: 11.12.2023).

13. Gilbert M. 13 Code Quality Metrics That You Must Track | Opsera. *Opsera | CI/CD Orchestration Platform and DevOps Intelligence*. URL: https://www.opsera.io/blog/13-code-quality-metrics-that-you-must-track (date of access: 11.12.2023).

14. Olawanle J. Big O Cheat Sheet – Time Complexity Chart. *freeCodeCamp.org*. URL: https://www.freecodecamp.org/news/big-o-cheat-sheet-time-complexity-chart/ (date of access: 11.12.2023).

15. Jena B. K. One-Stop Solution Guide to Understand Data Structure and Algorithm Complexity. *Simplilearn*. URL: https://www.simplilearn.com/tutorials/data-structure-tutorial/algorithm-complexity-in-data-structure. (date of access: 11.12.2023).

16. Савчук В. Big O: Складність алгоритмів. *The Code*. URL: https://www.the-code.com.ua/skladnist-alghoritmiv/ (дата звернення: 11.12.2023).

17. P, NP, CoNP, NP hard and NP complete | Complexity Classes - GeeksforGeeks. *GeeksforGeeks*. URL: https://www.geeksforgeeks.org/types-of-complexity-classes-p-np-conp-np-hard-and-np-complete/ (date of access: 11.12.2023).

18. Exterman D. Top Software Metrics for Dev Leaders - Incredibuild. *Incredibuild*. URL: https://www.incredibuild.com/blog/top-software-metrics-for-dev-leaders (date of access: 11.12.2023).

19. Size Oriented Metrics. *www.javatpoint.com*. URL: https://www.javatpoint.com/software-engineering-size-

orihttps://www.javatpoint.com/software-engineering-size-oriented-metricsented-metrics (date of access: 11.12.2023).

20. Казарін О. В., Шубинський І. Б. Надійність і безпека програмного забезпечення. М : Юрайт, 2018. 342 с.

21. Software code metrics. *PVS-Studio*. URL: https://pvs-studio.com/en/blog/posts/a0045/ (date of access: 11.12.2023).

22. Mathews S. Getting The Most Out Of Code Quality Metrics. *Medium*. URL: https://medium.com/codex/when-code-quality-metrics-can-be-of-real-value-to-software-engineers-7fc2de96afc4 (date of access: 11.12.2023).

23. A tour of C# - Overview - C#. *Microsoft Learn: Build skills that open doors in your career*. URL: https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/ (date of access: 11.12.2023).

24. Що таке .NET і чим займаються .NET-розробники?. *EPAM*. URL: https://training.epam.ua/ua/blog/301 (дата звернення: 11.12.2023).

25. Overview of .NET Framework - .NET Framework. *Microsoft Learn: Build skills that open doors in your career*. URL: https://docs.microsoft.com/uk-ua/dotnet/framework/get-started/overview (date of access: 11.12.2023).

26. Roslyn Analyzer. *Sitecore | Kentico | Umbraco | nopCommerce & .NET ontwikkeling en interim consultancy, support en training - ParTech*. URL: https://www.partech.nl/en/publications/2021/01/roslyn-analyzer (date of access: 11.12.2023).

27. МЕТРИКА ДИФЕРЕНЦІЙОВАНОЇ ЦИКЛОМАТИЧНОЇ СКЛАДНОСТІ АНАЛІЗУ ПРОГРАМНГО КОДУ З ВИКОРИСТАННЯМ СИСТЕМ КЕРУВАННЯ БАЗАМИ ДАНИХ / Ю. ФОРКУН та ін. *MEASURING AND COMPUTING DEVICES IN TECHNOLOGICAL PROCESSES*. 2023. № 3. С. 100–105. URL: https://doi.org/10.31891/2219-9365-2023-75-11 (дата звернення: 11.12.2023).

ДОДАТОК А

(обов'язковий)

**КОПІЇ НАУКОВИХ ПУБЛІКАЦІЙ**

*Міжнародний науково-технічний журнал*
***«Вимірювальна та обчислювальна техніка в технологічних процесах»***
**ISSN 2219-9365**

https://doi.org/10.31891/2219-9365-2023-75-11
УДК 004.051

ФОРКУН Юрій
Хмельницький національний університет
https://orcid.org/0000-0002-7906-4191
forkun@ridne.net
МАРТИНЮК Валерій
Хмельницький національний університет
https://orcid.org/0000-0001-5758-4244
martynyuk.valeriy@gmail.com
ПРАВОРСЬКА Наталія
Хмельницький національний університет
https://orcid.org/0000-0001-6001-3311
margana2000007@gmail.com
ЛУЧИЦЬКИЙ Олег
Хмельницький національний університет
oleg.arch999@gmail.com

# МЕТРИКА ДИФЕРЕНЦІЙОВАНОЇ ЦИКЛОМАТИЧНОЇ СКЛАДНОСТІ АНАЛІЗУ ПРОГРАМНГО КОДУ З ВИКОРИСТАННЯМ СИСТЕМ КЕРУВАННЯ БАЗАМИ ДАНИХ

*У даній роботі розглянуті сучасні рішення в області проведення аналізу програмного коду, виділені їх переваги та недоліки. Досліджені існуючі метрики складності коду на предмет їх комбінування і досягнення кращих результатів, у результаті чого була запропонована метрика диференційованої цикломатичної складності, основана на комбінації метрик цикломатичної складності та метрики Чепіна.*
*Ключові слова: аналіз програмного коду, цикломатична складність, метрика Чепіна, база даних, інженерія програмного забезпечення, якість програмного забезпечення.*

FORKUN Yuriy, MARTYNYUK Valeriy, PRAVORSKA Nataliia, LUCHYTSKYI Oleh
Khmelnytskyi National University

# METRICS OF DIFFERENTIATED CYCLOMATIC COMPLEXITY OF SOFTWARE CODE ANALYSIS USING DATABASE MANAGEMENT SYSTEMS

*In the modern world, the level of software development has reached a sufficiently high standard, making the question of quality software development particularly relevant. Specifically, the issue of analyzing the code of a software application allows for a thorough examination of every line of source code to understand its characteristics, quality, and potential issues. The quality and reliability of a program, as well as its future success, depend on the analysis of the source code.*

*This is especially true for today's popular software with a multi-tiered structure, where various algorithmic elements, such as databases, are distributed among different applications, requiring special attention to data exchange implementation.*

*This article explores modern solutions in the field of code analysis, highlighting their advantages and disadvantages. Existing code complexity metrics are examined for possible combinations to achieve better results, resulting in the proposal of a metric called "Differential Cyclomatic Complexity," which is based on the combination of cyclomatic complexity and Chapin's metric.*
*Keywords: software code analysis, cyclomatic complexity, Chapin's metric, database, software engineering, software quality.*

## Постановка проблеми у загальному вигляді та її зв'язок із важливими науковими чи практичними завданнями

Поняття інженерії програмного забезпечення (англ. «Software engineering») існує вже більше півсторіччя. Діяльність в межах цієї дисципліни формує основу, практично, для всієї науки розробки програмного забезпечення, включаючи в себе весь життєвий цикл створення програмних додатків, починаючи від аналізу вимог та проектування і закінчуючи тестуванням, підтримкою та оновленням. Розповсюдження програмного забезпечення в наш час у майже у всі сфери людського життя – результат досконалого вивчення та використання знань, досягнутих в межах цієї дисципліни.

Одним з питань, які досліджує інженерія програмного забезпечення, є питання якості розроблюваного програмного забезпечення. Одним з ведучих чинників є якість програмного коду. Запитання, що виникають перед розробниками програмного забезпечення, завжди були важливими і складними. Як зробити код більш надійним? Як забезпечити його продуктивність та ефективність? Як підтримувати та розвивати застосунок з плином часу? Як переконатися, що вона працює коректно та безпечно?

*Міжнародний науково-технічний журнал*
***«Вимірювальна та обчислювальна техніка в технологічних процесах»***
**ISSN 2219-9365**

Аналіз вихідного коду відіграє ключову роль у відповіді на ці запитання. Це процес, який дозволяє ретельно розглянути кожен рядок програмного коду та зрозуміти його характеристики, якість та можливі проблеми. Від аналізу вихідного коду залежить якість та надійність програми, а також її подальший успіх.

Сучасні програмні застосунки стають все складнішими, а підтримка розроблюваного програмного забезпечення сьогодні супроводжуються створенням оновлень з відносно великою частотою. Не дивлячись на роботу, напрямлену, здебільшого, на покращення роботи вже створеного функціоналу – видалення помилок та додавання нових можливостей – регулярно виникає ситуація, коли чергове оновлення призводить до погіршення роботи алгоритму, яке доводиться неминуче виправляти. У таких обставинах аналіз вихідного коду стає настільки ж важливим елементом процесу розробки, наскільки й складним. Зростання обсягу коду, використання різних технологій та бібліотек, а також вимог до продуктивності та безпеки роблять аналіз вихідного коду необхідністю.

Окремої уваги заслуговує програмного забезпечення, в основі роботи якого лежить принцип розділення алгоритмів роботи між безпосереднім додатком та базою даних. Ця структура ще називається моделлю «клієнт-сервер». В такій архітектурі за реалізацію різних елементів програмного застосунку відповідають різні додатки, між якими відбувається обмін інформацією. В таких застосунках, окрім роботи основного коду, аналізу потребують алгоритми обміну даними між програмними застосунками які працюють з базами даних, системами керування базами даних та програмних застосунків з багаторівневою структурою.

В даній статті будуть розглянуті сучасні пропозиції щодо вирішення питання аналізу програмного коду, визначені їх переваги та недоліки, на основі яких буде запропонована ще одна методика проведення аналізу коду програмних застосунків, яка продемонстрована у вигляді метрики диференційованої цикломатичної складності, що створена на основі показників метрики цикломатичної складності та метрики Чапіна.

### Аналіз досліджень та публікацій

Аналіз програмного коду є багатофакторною областю у розробці програмних застосунків, тобто існують різні варіанти впровадження нових рішень у цю область. Це спостерігається через ознайомлення з різноманітними роботами за даною тематикою, куди входить огляд таких тем:

- безпека програмного забезпечення;
- розробка засобів аналізу коду;
- розробка нових метрик;
- профілювання програмного забезпечення та інші.

Розглянемо частину цих робіт. Почнемо з роботи [1]. У даній статті автори розглядають явище регресії продуктивності роботи програмного застосунку, яка може виникати у випадку значних змін у коді в процесі створення нової версії. Вирішення даної проблеми представлені у вигляді використання або специфічних юніт-тестів, які самі по собі є непопулярним рішенням, або методик, які не користуються кодом та мають високі вимоги для отримання можливості проведення аналізу. Тому авторами була запропоновано прототип власного інструменту, робота якого базується на аналізі коду. Таке впровадження, згідно тексту роботи, зменшує витрати часу у процесі тестування та складність його проведення.

У роботі [2] досліджується аналіз коду на предмет наявності шкідливого коду, який є наслідком поширення тенденції використання відкритого вихідного коду у процесі розробки, що створює вразливості у системі безпеки програмного забезпечення. Для вирішення цієї проблеми, автори пропонують алгоритм, який використовує нейронну мережу з глибоким навчанням для аналізу коду для пошуку аномалій, які вказуватимуть на наявність вразливостей у програмному застосунку. Хоча автори зазначають, що метод має недоліки, але їх робота впроваджує сучасний засіб захисту, пов'язаний з актуальною проблемою.

У ще одній роботі [3] автори також досліджують пошук вразливостей у коді, мотивуючи це сучасними тенденціями (вузькі часові межі та слабо контрольована робота розробника з дому), що погіршують ефективність традиційних методик тестування програмного забезпечення. В цій роботі, технологію штучного інтелекту було об'єднано з технологією блокчейнів, яка дозволить підтримувати взаємодію з розроблюваним програмним забезпеченням та її складовими в умовах дистанційної роботи, підвищуючи рівень безпеки при обміні інформацією через мережу. Разом з автоматизацією процесу аналізу коду штучним інтелекту, нова методика полегшує роботу поза робочими приміщенням компаній-розробників.

Робота [4] пов'язана з добуванням інформації з вихідного коду об'єктно-орієнтованих мов програмування. Автори роботи розглядають існуючі методики для досягнення цих цілей, звертаючи увагу на обмеженість цих засобів в таких моментах, як доступність цих самих засобів для потенційних користувачів, так й обсягу та якості інформації, яка надається в результаті використання цих засобів. Впроваджений авторами новий метод базується на зовсім інших, ніж у традиційних попередників, технологіях, завдяки чому досягаються кращі результати у процесі збору вихідного коду та вилученні

*Міжнародний науково-технічний журнал*
***«Вимірювальна та обчислювальна техніка в технологічних процесах»***
**ISSN 2219-9365**

необхідної розробникам інформації. Враховуючи перехід на нові принципи створення моделі, це робота може бути основою для якісного стрибка серед засобів аналізу коду.

Автори роботи [5], розглядаючи оцінку складності, читабельність та зрозумілості коду, пропонують нову метрику у чотирьох варіаціях, яка дозволяє аналізувати код підвищеної складності, зокрема – код, в якому активно використовуються рекурсії, до яких автори звертають особливу увагу. Очевидно, що використання нової метрики робить зручнішим аналіз розвиненого коду, з яким можуть виникнути складності при використанні більш старіших метрик.

В тексті роботи [6] розглядає аналіз коду з точки зору т. зв. підсумування вихідного коду (англ. «Source Code Summarization»). Звертаючи увагу на можливість покращення якості підтримки програмного забезпечення та його розвитку за допомогою якісного коментування та опису коду, автори вказують на загальну низьку ефективність описування коду традиційними методами: API документація, GitHub та опис вручну. Таким чином, пропонується використання методу на основі принципу доповнення даних (англ. «Data Augmentation»). Згідно тексту роботи, новий метод дозволяє генерувати високоякісний опис програмного коду, що надає перспективи полегшити роботу при підтримці коду.

Огляд останніх досліджень та робіт в області аналізу коду вказує на логічні тенденції: автори робіт використовують досягнення останніх часів (в основному – поширення нейронних мереж) або розробляють власні методики, базуючись на недоліках існуючих рішень.

Таким чином, можна звернути увагу на кілька суперечливих моментів. Як наприклад, створення нових рішень пов'язано зі складною та тривалою розробкою та подальшим супроводом, складність яких напряму пов'язані з загальним рівнем розвитку сучасного програмного забезпечення. Також, звертаю увагу на відсутність спроб розвинути існуючі рішення шляхом їх об'єднання з іншими, що можна виправдати відносною легкістю, адже ці методики те технології вже перевірені часом і в наявності досвід їх використання. Впровадження їх модифікацій в такому плані викликає менші ризики, зокрема і при поєднання метрик складності програмного коду.

**Виклад основного матеріалу**

Сучасна практика розробки та аналізу програмного забезпечення володіє значною кількістю способів виміряти складність створеного алгоритму. Перш за все, використовуються показники часової складності (максимальна кількість часу для всіх можливих даних розміру n) та просторова складність (кількість додаткової пам'яті, яку необхідна при збільшенні об'єму вхідних даних) [7], які можна назвати базовими.

Окрім цих показників, отримали розповсюдженні різноманітні метрики, які розглядають складність програмного застосунку з точки зору різних аспектів. Так, розрізняють метрики трьох груп:

– розмірно-орієнтовані метрики – обчислюються на основі підрахунку певних характеристик вихідного коду. Характерні простотою використання;

– метрики складності потоку керування програми – напрямлені на дослідження різноманіття варіантів сценарії виконання програмного коду;

– метрики складності потоку керування даними – розглядають складність через використання вхідних даних, де складність прямо пропорційна частоті використання даних.

Таким чином, враховуючи такі підходи, ми пропонуємо модифікацію метрики складності потоку керування програми – цикломатична складність (інша назва - цикломатичне число МакКейба). Критерії, за допомогою яких вона оцінює складність потоку виконання програми, вираховуються на основі графу потоку керування. Шляхом розрахунків цикломатичну складність можна визначити за допомогою формули:

$$V(G) = e - n + 2p, \quad (1)$$

де e – кількість дуг, n – кількість вершин, p – компонент зв'язності.

Значення компонента зв'язності можна сприймати як кількість дуг, які необхідно додати для того, щоб граф потоку виконання програми став сильно зв'язаним, тобто коли дві його вершини є взаємно досяжними і з будь-якої вершини є шлях до будь-якої іншої вершини.

Для графів коректних програм, тобто для таких, які не містять недосяжних вершин від точки входу та з кожної вершини якої можна дістатись до кінцевої, сильна зв'язаність досягається шляхом з'єднання дугою кінцевої вершини та початкової. Можна стверджувати, що число V(G) визначає кількість лінійно незалежних контурів у сильно зв'язаному графі. Оскільки для коректної програми p=1, то формула набуває спрощеного виду:

$$V(G) = e - n + 2, \quad (2)$$

Як правило, при обчисленні цикломатичної складності логічні оператори не приймаються до уваги, допускається також спрощений підхід, згідно з яким власне побудова графа не проводиться, а показник

*Міжнародний науково-технічний журнал*
***«Вимірювальна та обчислювальна техніка в технологічних процесах»***
**ISSN 2219-9365**

визначається на підставі підрахунку кількості операторів керуючої логіки (if, switch і т. д.) і можливої кількості шляхів виконання програми. Дана метрика має кілька варіацій обчислення:

– «модифікована» цикломатична складність – розглядає не кожне розгалуження оператора множинного вибору (switch, case), а весь оператор як єдине ціле;

– «сувора» цикломатична складність – містить логічні оператори;

– «спрощена» цикломатична складність – передбачає обчислення на основі не графа, а підрахунку керуючих операторів;

– «актуальна» складність – визначається як кількість незалежних шляхів, що проходить код при тестуванні;

– метрика складності глобальних даних – обчислюється як цикломатична складність модуля та збільшується на кількість взаємозв'язків з глобальними даними.

Метрика цикломатичної складності може бути розрахована для модуля, методу та інших структурних одиниць програмного забезпечення.

Завдяки такій методиці розрахунку, цикломатична складність є метрикою, яка не залежить від мови програмування та результати обчислення котрої легко розуміти. Завдяки цьому, ця метрика стала дуже розповсюдженою і дозволяє легко створювати похідні метрики. Власне, тому ця метрика і була обрана.

Однак, дана метрика має серйозний недолік: цикломатична складність не враховує складність операцій. Таким чином отримати два різних програмних застосунки з приблизно однаковим значенням цикломатичної складності, проте справжня їх складність буде відрізнятися.

Тому нами запропоновано метрика диференційованої цикломатичної складності, яка дозволить здійснити об'єднання цикломатичної складності з метрикою, яка б могла подолати цей недолік.

Для цього нами були розглянуті метрики трьох зазначених груп. Метрики складності потоку керування програми, очевидно, не підходять, оскільки вони входять до однієї групи.

Серед розмірно-орієнтованих метрик розглянуто SLOC-метрику, її аналоги та метрику Холстеда. SLOC-метрика та її аналоги надають показники у вигляді кількісної характеристики коду (кількість рядків, операторів, методів тощо). Самі по собі такі показники вдається використати, але це окремі випадки. Метрика Холстеда представляє собою сукупність кількох показників, які послідовно розраховуються і формують набір показників, по яким вже можна оцінити програму. Для цієї метрики вже цикломатична складність є зайвою.

Серед метрик складності потоку керування даними були розглянуті: метрика Чепіна, пара «модуль–глобальна змінна», метрика Спена, метрика Кафура.

Метрика Чепіна розраховується на основі використовуваних змінних вводу-виводу окремого програмного модуля, розділяючи змінні на 4 групи, кожна з яких має власний ваговий коефіцієнт. Показником метрики виступає сума добутків кількості змінних кожної групи на власні коефіцієнти.

Пара «модуль-глобальна змінна» розраховує ймовірність посилання довільного модуля на довільну глобальну змінну в програмі. Висока вірогідність вказує на ризик «несанкціонованої» зміни будь-якої змінної, що може суттєво ускладнити роботи, пов'язані з модифікацією програмного застосунку.

Метрика Спена обчислює кількість тверджень, які містять один й той самий ідентифікатор. Великі показники спену вказують, що тестування та налагодження проходитиме складніше.

Метрика Кафура використовує поняття локального потоку даних, глобального потоку даних. Ця метрика обчислюється на основі інформаційної складності процедур, що проходять через модуль програмного застосунку, та складності модуля відносно структури даних програмного застосунку.

В ідеалі, модифікація метрики цикломатичної складності має надати лише один додатковий показник, який би дозволив би відрізнити програми різної складності. Серед вказаних метрик була обрана метрика Чепіна через простоту отримання та розуміння показника.

Сама по собі метрика обчислюються за формулою:

$$Q = a_1 \times P + a_2 \times M + a_3 \times C + a_4 \times T, \tag{3}$$

де $a_1$, $a_2$, $a_3$, $a_4$ – вагові коефіцієнти,

P – кількість змінних, що вводяться, для розрахунків і для забезпечення виведення,

M – кількість змінних, які модифіковані або створювані всередині програмного застосунку,

C – кількість змінних, що беруть участь в управлінні програмним модулем (керуючі змінні),

T – кількість змінних, які не використовуються в програмному застосунку («паразитні»).

Згідно думки автора метрики, вагові коефіцієнти, становлять: $a_1$=1, $a_2$=2, $a_3$=3, $a_4$=0,5.

Таким чином, запропонована метрика диференційованої цикломатичної складності визначається наступним чином:

$$D(G) = [V(G), Q_a], \tag{4}$$

*Міжнародний науково-технічний журнал*
***«Вимірювальна та обчислювальна техніка в технологічних процесах»***
**ISSN 2219-9365**

де $V(G)$ – цикломатична складність,

$Q_a$ – значення загальної середньої складність програмних модулів, яка визначається за формулою:

$$Q_a = \sum_{0}^{i} Q_i, \tag{5}$$

де $Q_i$ – середня складність програмних модулів на $i$-тому шляху виконання програми:

$$Q_i = \frac{\sum_0^n Q_{ij}}{n}, \tag{6}$$

де $Q_{ij}$ – значення метрики Чепіна для $j$-того програмного модуля на $i$-тому шляху виконання програмного застосунку,

$n$ – кількість програмних модулів, з яких складається шлях виконання програмного застосунку.

Реалізацію обчислення диференційованої цикломатичної складності слід розглядати двома способами: виконання «вручну» та автоматизоване.

Виконання «вручну» мало чим відрізняється від збору інформації двох окремих метрик і його характер визначається варіацією обчислення цикломатичної складності. Тобто, наприклад, при обчисленні «модифікованої» цикломатичної складності, середня складність програмних модулів буде включати значення метрики Чепіна всіх програмних модулів, які включатимуть в собі оператори множинного вибору.

Для реалізації програмними засобами можна використати два підходи: повний та спрощений, по аналогії з метрикою цикломатичної складності. Повний підхід включає повноцінний аналіз коду з пошуком шляхів його виконання та побудування графу керування, де визначена належність кожного програмного модуля до свого шляху. Спрощений підхід обмежується лише підрахунком кількості операторів керуючої логіки, кількості програмних модулів та їх загальної середньої складності. Перший спосіб доречний у випадку дослідження коду на наявність занадто складних частин, де загальний аналіз нічого не дасть.

Запропонована метрика дозволяє краще визначити рівень складність програмного застосунку, код яких може мати приблизно однакову кількість способів виконання. Нова метрика може бути використана на етапі проектуванні для завчасного визначення теоретичної складності програмного додатку та на етапі тестування для безпосереднього визначення складності розроблюваного програмного забезпечення. Метрика дозволяє проводити аналіз програмних застосунків з різноманітними структурами, зокрема – ефективно себе демонструє при аналізі додатків з багаторівневою структурою, зокрема, які працюють з базами даних та системами керування базами даних.

**Висновки з даного дослідження і перспективи подальших розвідок у даному напрямі**

У цій роботі було розглянуто частино робіт у області аналізу програмного коду. Досліджені роботи демонструють різносторонні підходи до покращення процесу аналізу у межах досягнення різноманітних результатів, але не спостерігалися спроби використання методик, побудованих на основі вже існуючих та перевірених рішень.

Була запропонована метрика диференційованої цикломатичної складності, створена для усунення недоліку оригінальної метрики цикломатичної складності у вигляді нездатності розрізнити складність двох програмних застосунків з однаковою кількістю шляхів. Утворена метрика дозволяє визначити, за допомогою використання показників метрики Чепіна, складність операцій, що відбувається всередині алгоритму, що дозволяє розрізняти два програмних застосунки різної реальної складності з однакової кількістю шляхів виконання. Тобто, диференційованої цикломатична складність демонструє більшу точність у порівнянні з оригінальною метрикою. У тому числі, при аналізі програмних застосунків з різноманітними структурами, зокрема – ефективно себе демонструє при аналізі програмних застосунків, які працюють з базами даних, системами керування базами даних та програмних застосунків з багаторівневою структурою зокрема.

**References**

1. Salma Eid, Soha Makady, Manal Ismail, Detecting software performance problems using source code analysis techniques, Egyptian Informatics Journal, Volume 21, Issue 4, 2020, Pages 219-229, ISSN 1110-8665, https://doi.org/10.1016/j.eij.2020.02.002.
2. Chen Tsfaty, Michael Fire, Malicious source code detection using a translation model, Patterns, Volume 4, Issue 7, 2023, 100773, ISSN 2666-3899, https://doi.org/10.1016/j.patter.2023.100773.
3. Panchanan Nath, Jaya Rani Mushahary, Ujjal Roy, Maharaj Brahma, Pranav Kumar Singh, AI and Blockchain-based source code vulnerability detection and prevention system for multiparty software development, Computers and Electrical Engineering, Volume 106, 2023, 108607, ISSN 0045-7906, https://doi.org/10.1016/j.compeleceng.2023.108607.
4. Gholamali Nejad Hajali Irani, Habib Izadkhah, Sahand, 1.0: A new model for extracting information from source code in object-oriented projects, Computer Standards & Interfaces, 2023, 103797, ISSN 0920-5489, https://doi.org/10.1016/j.csi.2023.103797.
5. Gordana Rakić, Melinda Tóth, Zoran Budimac, Toward recursion aware complexity metrics, Information and Software Technology, Volume 118, 2020, 106203, ISSN 0950-5849, https://doi.org/10.1016/j.infsof.2019.106203.

6. Zixuan Song, Hui Zeng, Xiuwei Shang, Guanxi Li, Hui Li, Shikai Guo, An data augmentation method for source code summarization, Neurocomputing, Volume 549, 2023, 126385, ISSN 0925-2312, https://doi.org/10.1016/j.neucom.2023.126385.
7. Savchuk V. Big O: Skladnist algoritmiv [Internet-resurs]. – Rezhym dostupu: https://www.the-code.com.ua/skladnist-alghoritmiv/vilniy.

ДОДАТОК Б

(обов’язковий)

## ПРЕЗЕНТАЦІЙНІ МАТЕРІАЛИ

Хмельницький національний університет
Факультет інформаційних технологій
Кафедра інженерії програмного забезпечення

# Кваліфікаційна робота

на тему: «Удосконалення методу оцінки цикломатичної складності коду для підвищення ефективності тестування програмного забезпечення»

Керівник:
канд. техн. наук, доцент
Форкун Ю. В.

Виконав:
студент ІІ курсу
групи ІПЗм-22-1
Лучицький О.Ю.

# Актуальність

Аналіз вихідного коду комп'ютерних програм є одним з основних елементів процесу розробки програмного забезпечення. Проведення аналізу є доречним на більшості етапів процесу розробки, оскільки отримані відомості про структуру коду дозволяє визначити помилки, вразливості та можливості для оптимізації алгоритмів.

Метрика цикломатичної складності досі є одним з основних засобів аналізу коду за рахунок зрозумілості та простоти використання. Тому спроби удосконалити цю метрику залишаються актуальним й на сьогоднішній день.

## Об'єкт та предмет дослідження:

**Об'єктом** дослідження є процес розробки програмного забезпечення.

**Предметом** дослідження є метрики аналізу складності коду програмного додатку, зокрема – метрика цикломатичної складності коду.

## Мета та завдання дослідження:

**Метою** проекту є розробка метрики диференційованої цикломатичної складності.

**Завдання** дослідження включають:

- дослідити предметну область – розробка програмного забезпечення;
- проаналізувати існуючі дослідження в області використання метрик складності коду;
- розробити теоретичні основи для впровадження нової техніки;
- визначити структуру програмного додатку для реалізації нової метрики;
- вибрати технології для подальшої розробки програмного додатку;
- розробити та протестувати програмне забезпечення на відповідність вимогам та наявність недоліків.

## Наукова новизна

• отримала подальший розвиток метрика цикломатичної складності у напрямку відображення складності алгоритмів програмного додатку більш об'єктивно;

• розроблено метрику диференційованої цикломатичної складності, яка за рахунок поєднання алгоритмів оцінки цикломатичної складності та метрики Чепіна, дозволяє точніше визначати складність алгоритмів програмного додатку.

## Практична цінність

Практична цінність полягає у можливості враховувати різну фактичну складність програмних застосунків в процесі аналізу коду.

В результаті це покращить результативність процесу тестування, що підвищить якість реалізації алгоритму та зменшить витрати на процес розробки програмного забезпечення.

## Існуючі засоби оцінки складності програмного коду

- Часова складність.
- Просторова складність.
- Розмірно-орієнтовані метрики.
- Метрики складності потоку керування даними.
- Метрики складності потоку керування програмами.
- Метрики складності потоку керування даними та програми.
- Об'єктно-орієнтовані метрики.
- Гібридні метрики.


Існуючі засоби оцінки складності програмного коду

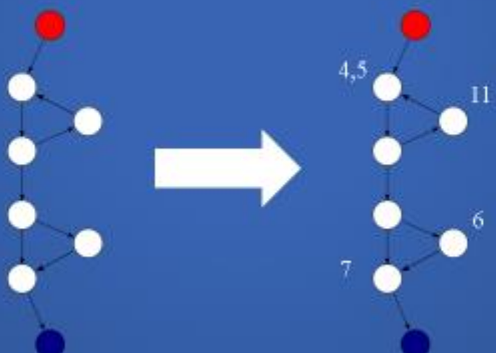
- Часова складність.
- Просторова складність.
- Розмірно-орієнтовані метрики.
- Метрики складності потоку керування даними.
- Метрики складності потоку керування програмами.
- Метрики складності потоку керування даними та програми.
- Об'єктно-орієнтовані метрики.
- Гібридні метрики.

## Диференційована цикломатична складність

- На основі поєднання цикломатичної складності та метрики Чепіна.
- $D(G) = [V(G), Q_a]$
- $Q_a = \sum_0^i Q_i$
- $Q_i = \frac{\sum_0^n Q_{ij}}{n}$
- $Q_{ij} = (\mathrm{P} + 2\mathrm{M} + 3\mathrm{C} + 0.5\mathrm{T})_{ij}$
- Можна додатково зазначити кількість методів: $D(G) = [V(G), n, Q_a]$

## Практична реалізація

- Вручну – найбільше залежить від варіанту обчислення цикломатичної складності.
- Розширений граф потоку керування, де можна визначити ускладнену частину коду:



# Практична реалізація

- Програмно спрощена – підрахунок кількості операторів керуючої логіки, кількості модулів та середньої складності. Обрано цей варіант.

- Програмно повна – аналіз коду з пошуком його виконання та побудову графу керування, де визначається належність кожного модуля до свого шляху та його складність.

# Практична реалізація

**Діаграма послідовності**

# Практична реалізація

# Висновки

Нова метрика диференційованої цикломатичної складності дозволяє отримувати точнішу інформацію щодо складності алгоритмів програмного забезпечення та нівелює недолік метрики цикломатичної складності, пов'язаний з отриманням приблизно однакових показників при різній об'єктивній складності аналізованих програмних додатків.

Нову метрику складності коду можна використовувати в процесі розробки програмного забезпечення на всіх етапах, де доречне проведення аналізу коду додаку.

Як в самій методиці, так і програмній реалізації наявні певні недоліки, вирішення яких дозволить в перспективі покращити показники метрики.

# Публікація

ФОРКУН, Ю. ., МАРТИНЮК, В., ПРАВОРСЬКА, Н., & ЛУЧИЦЬКИЙ, О. (2023). МЕТРИКА ДИФЕРЕНЦІЙОВАНОЇ ЦИКЛОМАТИЧНОЇ СКЛАДНОСТІ АНАЛІЗУ ПРОГРАМНОГО КОДУ З ВИКОРИСТАННЯМ СИСТЕМ КЕРУВАННЯ БАЗАМИ ДАНИХ . *MEASURING AND COMPUTING DEVICES IN TECHNOLOGICAL PROCESSES*, (3), 100–105. https://doi.org/10.31891/2219-9365-2023-75-11

# Доповідь завершено

Завідувачу кафедри інженерії програмного забезпечення проф. Леоніду БЕДРАТЮКУ
здобувача вищої освіти
**Олега ЛУЧИЦЬКОГО**
факультет ІТ, 2 курс, група ІПЗм-22-1

**ЗАЯВА**

З правилами чинного Положення про систему забезпечення академічної доброчесності в Хмельницькому національному університеті, згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту і застосування заходів дисциплінарної та академічної відповідальності, ознайомлений. Про використання програмно-технічних засобів для перевірки кваліфікаційних робіт здобувачів вищої освіти на плагіат оповіщений та надаю свою згоду на обробку й збереження університетом моєї роботи в інституційному репозитарії університету.

Також надаю університету право на передачу моєї роботи для обробки та збереження в базах даних програмно-технічних засобів (Unicheck та Anti-Plagiarism) та використання роботи для виявлення плагіату в інших роботах, які перевіряються програмно-технічними засобами та користувачами, що мають доступ до цих програмно-технічних засобів, виключно в обмежених цілях для виявлення плагіату в текстах робіт.

Робота для перевірки університетом надається в друкованому та електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

05.09.2023 — дата

О. Луч — підпис

Mon Dec 11 09:37:07 EET 2023, Форкун Юрій Вікторович, Хмельницький національний університет, ХНУ

## Anti-Plagiarism v-15.257

**Максимальне співпадіння з одним документом 4.0%**

Словники перевірки: en_US, ru_RU, ua_UA. **Помилок в документах: 7%**

| ID: 122448<br>Назва: Удосконалення методу оцінки цикломатичної складності коду для підвищення ефективності тестування програмного забезпечення<br>Додано в БД: 2023-12-11<br>Автора: Лучицький О. Ю.<br>Керівники: Форкун Ю.В.<br>Консультанти:<br>Опоненти: | Документ | | Сумарний збіг по Базі Даних | |
|---|---|---|---|---|
| | Символи | Лексеми | Символи | Лексеми |
| | 127481 | 991 | 10667 (8%) | 124 (13%) |

Джерело плагіату

| ID | Опис | Наявність плагіату в документі | |
|---|---|---|---|
| | | Символи | Лексеми |

**Ім'я користувача:**
ІПЗ

**ID перевірки:**
1015991231

**Дата перевірки:**
11.12.2023 10:35:57 EET

**Тип перевірки:**
Doc vs Internet + Library

**Дата звіту:**
11.12.2023 10:36:47 EET

**ID користувача:**
100012953

---

**Назва документа:** КвР_Лучицький О.Ю._ІПЗм-22-1

**Кількість сторінок:** 93 **Кількість слів:** 17615 **Кількість символів:** 141048 **Розмір файлу:** 862.26 KB **ID файлу:** 1015673800

---

# 15.9% Схожість

**Найбільша схожість:** 5.55% з джерелом з Бібліотеки (ID файлу: 1011261885)

15.7% Джерела з Інтернету | 760 ........ Сторінка 95

7.24% Джерела з Бібліотеки | 108 ........ Сторінка 99

# 0.4% Цитат

Цитати | 5 ........ Сторінка 100

Не знайдено жодних посилань

# 0% Вилучень

Немає вилучених джерел

# Модифікації

Виявлено модифікації тексту. Детальна інформація доступна в онлайн-звіті.

Замінені символи | 2

# **РІШЕННЯ** ЕКСПЕРНОЇ КОМІСІЇ КАФЕДРИ ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Підтверджуємо ознайомлення з результатами звіту/звітів подібності щодо роботи, продукованими програмно-технічним засобом(ами) перевірки текстів на плагіат.

Назва: « Удосконалення методу оцінки цикломатичної складності коду для підвищення ефективності тестування програмного забезпечення»

Автор: Лучицький Олег Юрійович

Спеціальність: 121 – Інженерія програмного забезпечення

Освітня програма: Освітньо-професійна програма «Інженерія програмного забезпечення»

Науковий керівник: Форкун Юрій Вікторович, кандидат технічних наук, доцент

Після аналізу звіту подібності зроблено такий висновок:

| № | Висновок | Позначка про відповідність |
|---|---|---|
| 1 | Запозичення, виявлені в роботі, є законними і не є плагіатом. Робота приймається до захисту. | **відповідає** |
| 2 | Виявлені запозичення не є плагіатом, розміщені у розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи. Робота приймається до захисту, але має бути відкоригована. Відкоригований варіант має бути поданий на кафедру за два дні до захисту, разом із заявою щодо самостійності виконання письмової роботи та ідентичності друкованої й електронної версії роботи. | |
| 3 | Виявлені запозичення не є плагіатом, але частково розміщені у розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. В зв'язку з цим мета роботи та поставлені завдання не були досягнені. Робота може бути допущена до захисту (наступного року) після того, як буде відкоригована та доопрацьована і успішно пройде повторну перевірку на академічний плагіат. | |
| 4 | Робота містить навмисні текстові спотворення, передбачувані спроби укриття запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту. | |
| 5 | Інше: | |

Підтвердження:

Запозичення, виявлені у роботі, є законними і не є плагіатом, оскільки:

1) у тексті кваліфікаційної роботи системою перевірки на плагіат Unicheck виявлено схожість з деякими документами у частині загальновживаних обов'язкових словосполучень у стандартних бланках (титулка, бланк завдання), у структурі змісту, у назвах розділів/підрозділів, у назвах публікацій переліку джерел посилання тощо;

2) в якості запозичень системою Unicheck було зафіксовано деякі послідовності вихідного коду і посиланання на бібліотеки, які є стандартними мовними конструкціями програмування та не можуть розглядатися як об'єкт авторських прав і, відповідно, їх порушення;

3) запозичення, виявлені в тексті роботи, є фрагментарними або мають належним чином оформленні посилання;

4) виявлені модифікації тексту не впливають на відсоток схожості.

Максимальний обсяг запозичень, визначений системою Anti-Plagiarism, складає 4.0%. Обсяг запозичень, визначений системою Unicheck виявлення збігів ідентичності/схожості, складає 15,9% і адресується до 760 джерел з інтернету і 108 джерела з бібліотеки, що, з урахуванням наведених обґрунтувань, відповідає характеру теми і свідчить на користь кваліфікаційної роботи.

Дата 11.12.2023

Завідувач кафедри ІПЗ ____________ Леонід БЕДРАТЮК

Гарант освітньої програми ____________ Оксана ЯШИНА

Керівник кваліфікаційної роботи ____________ Юрій ФОРКУН

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

# РЕЦЕНЗІЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ
освітнього ступеня «магістр»

Магістр Лучицький Олег Юрійович

Тема Удосконалення методу оцінки цикломатичної складності коду для підвищення ефективності тестування програмного забезпечення

Спеціальність 121 «Інженерія програмного забезпечення»

**Обсяг кваліфікаційної роботи:**

Кількість сторінок кваліфікаційної роботи 112.

1. Короткий зміст роботи та прийнятих рішень У кваліфікаційній роботі здійснено системний аналіз предметної області у сфері розробки програмного забезпечення, розглянуті основні засоби проведення аналізу коду, їх переваги та недоліки. На основі проведеного аналізу удосконалено та програмно реалізовано метрика диференційованої цикломатичної складності на основі модифікування метрики цикломатичної складності. Розроблена метрика дозволяє вирішувати конфліктну ситуацію, коли звичайна цикломатична складність демонструє однакові показники для додатків з різною об'єктивною складністю, покращити якість аналізу коду на різних етапах та, як наслідок, якість процесу розробки програмного продукту.

2. Висновок про відповідність роботи дипломному завданню Кваліфікаційна робота освітнього ступеня «магістр» у повній мірі відповідає поставленому завданню як у теоретичній, так і в практичній її частині.

3. Характеристика виконання кожного розділу роботи, ступінь використання останніх досягнень науки і техніки і передових методів роботи У вступі обґрунтовується актуальність теми роботи, формулюються мета та завдання дослідження, описується наукова новизна та практична цінність отриманих результатів. У першому розділі охарактеризовано структуру процесу розробки програмного забезпечення в якості предметної області. У другому розділі досліджено методи і способи проведення аналізу коду, зокрема метрику цикломатичної складності, визначені переваги та недоліки описаних методів та способів проведення аналізу. У третьому розділі обґрунтовано використання метрики цикломатичної складності як основи для утворення нової метрики, розглянуті інші засоби проведення аналізу коду у контексті можливості комбінування одного з них з цикломатичної складністю, запропоновані теоретичні основи використання нової метрики. У четвертому розділі розглянуто питання, що стосуються реалізації програмного засобу на основі прийнятих рішень, а також її технічні та технологічні характеристики. Також проведено емпіричне дослідження, спрямоване на доведення працездатності розробленої теорії та програмного засобу. Обґрунтована ефективність метрики диференційованої цикломатичної складності та розроблено рекомендації з її застосування.

4. Позитивні сторони роботи В кваліфікаційній роботі обґрунтовано доцільність проведення модифікації існуючих засобів аналізу коду програмних додатків. Запропоновано удосконалення метрики цикломатичної складності, яке дозволяє подолати один з головних її недоліків. Удосконалений алгоритм продемонстрував свою ефективність під час апробації.

5. Негативні сторони роботи У роботі в якості способу покращення використано спосіб поєднання методик вже існуючих метрик. Метрика диференційованої цикломатичної складності, так само, як й звичайна цикломатична складність, не враховує при обчисленні особливості структури досліджуваного коду (наприклад, не визначена різниця між умовним та циклічним розгалуженням в коді).

6. Оцінка графічного оформлення та пояснювальної записки роботи Графічне оформлення виконане відповідно до теми кваліфікаційної роботи з дотриманням вимог стандартів. Пояснювальна записка відповідає вимогам стандартів до її оформлення.

7. Відгук про роботу в цілому В цілому кваліфікаційна робота заслуговує позитивної оцінки. Весь матеріал роботи структурований, чіткий та послідовний. Усі розділи роботи є послідовними та логічними, що дозволяє чітко розуміти викладений матеріал у рамках тематики кваліфікаційної роботи. Графічний матеріал дозволяє наочно побачити доцільність та ефективність рішень, які були прийняті для вирішення поставленої задачі.

8.Інші зауваження

9. Оцінка кваліфікаційної роботи Розглянувши позитивні та негативні сторони представленої кваліфікаційної роботи, можна зробити висновок, що вона заслуговує оцінки «відмінно».

РЕЦЕНЗЕНТ (прізвище, ім'я, по батькові, посада, місце роботи)
Мартинюк Валерій Володимирович, зав. каф. автоматизації, комп'ютерно-інтегрованих технологій в робототехніки

08.12.2023
Дата

(Підпис)