Хмельницький національний університет
Факультет інформаційних технологій
Кафедра інженерії програмного забезпечення

ДИПЛОМНА РОБОТА

Автоматизована система аналізу програмного коду для оцінки ризиків та забезпечення якості 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 ДРІПЗ. 2001111.01.06.ПЗ

Виконав студент 2 курсу група ІПЗм-20-1 _____ В.В. Кудрявцев
Підпис Ініціали, прізвище

Керівник д.т.н. проф. _____ О.В. Бармак
Науковий ступінь, звання Підпис Ініціали, прізвище

Нормоконтролер канд. техн.наук, доцент _____ Ю. В. Форкун
Підпис Ініціали, прізвище

До захисту допускаю:
Завідувач кафедри інженерії програмного забезпечення _____ Л. П. Бедратюк
Підпис Ініціали, прізвище

7 грудня 2021 р.

Хмельницький 2021

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 2021 р.

ЗАВДАННЯ
НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ)

Кудрявцеву Віктору Володимировичу
Прізвище, ім'я, по батькові студента

1. Тема проєкту (роботи) Автоматизована система аналізу програмного коду для оцінки ризиків та забезпечення якості програмного забезпечення

Керівник проєкту (роботи) Бармак Олександр Володимирович д. т. н. проф.
Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 25.08.2021 р. № 102

2. Строк подання студентом проєкту (роботи) на кафедру 01.12.2021 р.

3. Вихідні дані до проєкту (роботи) Матеріали переддипломної практики

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)
1. Теоретичні основи досліджуваної проблеми
2. Аналіз методів забезпечення якості програмних систем
3. Розробка програмного рішення

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

6. Консультанти розділів дипломного проєкту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Антиплагіат	Гурман І. В., доцент	3.12.21	3.12.21
Нормоконтроль	Форкун Ю. В., доцент	30.11.21	3.12.21

7. Дата видачі завдання «_01_» __вересня__ 2021 р.

КАЛЕНДАРНИЙ ПЛАН

Назва етапів (розділів) дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1 Вивчення предметної області; формулювання мети та задач дослідження; визначення об'єкта та предмета дослідження; формування логістичної структури дипломної роботи	01.09-10.09.2021	
2 Робота над розділом 1 дипломної роботи – вивчення літературних та Інтернет-джерел; аналіз відомих моделей, методів та засобів за темою роботи; визначення методологічних підходів до вирішення задачі; висновки до розділу та постановка задач дослідження	11.09-25.09.2021	
3 Робота над розділом 2 дипломної роботи – розробка моделей, методів та алгоритмів вирішення задачі; висновки до розділу	26.09-10.10.2021	
4 Робота над науковими статтями	11.10-30.10.2021	
5 Робота над розділом 3 дипломної роботи – розробка інформаційної технології вирішення задачі (аналіз вимог до програмного засобу та його проектування, аналіз та вибір засобів реалізації програмного засобу тощо); висновки до розділу	11.10-26.10.2021	
7 Попередній захист дипломної роботи	Листопад (згідно графіка)	
8 Узгодження постановки задачі, отриманих результатів та висновків; написання вступу, загальних висновків, оформлення джерел посилання та додатків; оформлення пояснювальної записки та графічних матеріалів згідно вимог чинних стандартів	18.11-30.11.2021	
9 Перевірка роботи на наявність плагіату; нормоконтроль; брошурування пояснювальної записки; підготовка супровідних документів	01.12-04.12.2021	
10 Підготовка до захисту дипломної роботи	з 01.12.2021 р	

Студент _____ В.В. Кудрявцев
Підпис Ініціали, прізвище

Керівник проекту (роботи) _____ _____
Підпис Ініціали, прізвище

РЕФЕРАТ

Тема дипломної роботи: «Автоматизована система аналізу програмного коду для оцінки ризиків та забезпечення якості програмного забезпечення».

Автор роботи: Кудрявцев Віктор Володимирович.

Керівник роботи: Бармак Олександр Володимирович.

Пояснювальна записка: 122 с., 14 рис., 5 табл., 3 дод., 11 джерел.

ОЦІНКА РИЗИКІВ, МЕТРИКА КОДУ, ЗАБЕЗПЕЧЕННЯ ЯКОСТІ, ОПТИМІЗАЦІЯ КОДУ.

Об’єкт дослідження — процеси оцінки ризиків та забезпечення якості при розробці програмних продуктів.

Мета дослідження — удосконалення методів оцінки якості коду, розробка методу оцінки пріорітетів оптимізації програмних продуктів та розробка програмного засобу на основі розробленого методу, що дозволить проводити аналіз пріорітетів оптимізації на основі введених користувачем даних про розроблюваний проект.

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 дипломного проєктування досліджено галузь оцінки якості програмного забезпечення та сучасні методи і засоби визначення рівня якості програмних кодів, визначено можливі напрями вдосконалення існуючих методів та підходи до їх вдосконалення; на базі невирішених проблем розроблено удосконалені методи оцінки якості програмного коду та розроблено метод оцінки пріорітетів оптимізації програмних систем та виконано програмну реалізацію розробленого алгоритму.

Вдосконалені методи дозволяють більш детально оцінювати такі аспекти якості програмних кодів як складність алгоритму, зчеплення модулю, складність

потоку управління даними шляхом врахування додаткових коефіцієнтів при обчисленні цих метрик програмного забезпечення. Розроблений метод оцінки пріорітетів оптимізації програмних проектів дозволяє проводити дослідження з метою визначення основних ключових аспектів розроблюваної програмної системи, що потребують оптимізації.

Для програмної реалізації використано такі технології як C#, .NET, WPF.

Проведені емпіричні дослідження доводять адекватність та ефективність розроблених методів оцінки якості програмних кодів та визначення пріорітетів оптимізації програмних систем, працездатність та функціональну придатність реалізованого на їх основі програмного засобу.

Підпис

01.12.2021
Дата

ABSTRACT

Master's thesis: «Automatized Software Code Analysis System for Risk Assessment and Quality Assurance of Software».

Author: Kudriavtsev Viktor.

Head of research: Barmak Oleksandr.

Master's thesis consists of: 122 p., 14 pc., 5 tb., 3 add., 11 srs.

RISK ASSESSMENT, CODE METRIC, QUALITY ASSURANCE, CODE OPTIMIZATION.

The subject of the research are risk assessment and quality assurance processes in software development.

The aim of the research is improvements of code quality assurance methods, development of the method for assessing the priorities of program product's optimization processes and development of new software on the base of developed method, that would allow to conduct the analysis of optimization priorities for program products based on entered data about the project.

The following approaches of the research are used during the study:

– monitoring, experiment, abstraction, analysis and synthesis, formalization;
– modern instrumental approaches of system design and development;
– personal computer.

During the study the sphere of quality assurance of software and modern methods and tools of quality assessment of software code were researched and investigated, possible directions of improvement of existing methods and the ways of their improvement were determined; based on unsolved problems improved methods of code quality assessment were developed as well as a method for assessment of optimization priorities for program system and its program realization were developed.

Improved methods allow to evaluate in more detail such aspects of software code quality as algorithm complexity, module coupling, complexity of data management flow by taking into account additional coefficients during the calculation of these software metrics. The developed method of estimating the priorities of software project

optimization allows to conduct research to determine the main key aspects of the developed software system that need optimization.

C#, .NET and WPF were used for the implementation of the developed method.

The conducted empirical researches prove adequacy and efficiency of the developed methods of an estimation of quality of program codes and definition of priorities of optimization of software systems, working capacity and functional suitability of the software realized on their basis.

Signature

01.12.2021
Date

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

БД	–	база даних
ПЗ	–	програмний засіб
ПК	–	персональний комп’ютер
ПС	–	програмна система
ООП	–	об’єктно-орієнтоване програмування
IT	–	Information Technology
ISO	–	International Organization for Standardization
IEC	–	International Electrotechnical Commission
MVVM	–	Model-View-ViewModel
UI	–	User Interface
UML	–	Unified Modeling Language
UX	–	User Experience
WPF	–	Windows Presentation Foundation

ВСТУП

В сучасних реаліях основна маса IT проектів має потребу у приділенні уваги написанню якісного коду. Це означає, що програмний код проекту має не лише бути коректним та виконувати поставлену задачу без помилок, але і відповідати визначеним на проекті стандартам та бути легким для розуміння. Виникнення таких вимог спричинене в першу чергу розвитком технологій розробки, що в свою чергу збільшує складність архітектури програмних продуктів та систем.

В наслідок появи таких розширених вимог до написання програмного коду було розроблено велику кількість різноманітних принципів та методів розробки програмного забезпечення та архітектури програмних проектів. Без застосування цих методик на сьогоднішній день не обходиться майже жоден комерційний проект в сфері IT. Основною метою їх застосування є формування набору правил, за якими відбуватиметься розробка програмного коду та архітектури проекту. Необхідність дотримання цим правилам полягає в тому, що однорідний код, написаний за визначеними принципами є легшим для розуміння, а відповідно і легшим для розширення та модифікації. Окрім того, легкий для розуміння код також полегшує процес залучення до проекту нових розробників та зменшує імовірність помилок при подальшій розробці через невірне розуміння ними процесу роботи системи.

Також, для контролю якості розроблюваного проекту часто застосовують метрики прогнозування якості програмних продуктів. Основними видами метрик є метрики процесів, які оцінюють сам процес розробки та метрики коду, які дозволяють здійснювати оцінку написаного програмного коду та його архітектури. Такі метрики часто застосовуються на проектах з метою оцінки продуктивності розробки та визначення слабких місць проекту і шляхів їх покращення.

Впровадження стандартів програмування може дати позитивні результати не одразу. Розробникам може знадобитись певний час для освоєння нових методів розробки та адаптування до нових правил кодування. Проте дослідження [1] показують, що запровадження стандартів покращує продуктивність проекту та зменшує імовірність виникнення помилок в довготривалій перспективі.

Таким чином, потреба в наявності системи стандартів та метрик для аналізу та контролю якості системи існує в майже кожного IT проекту. Проте, варто зазначити, що система стандартів та правил повинна бути збалансована, тобто принципи не повинні суперечити один одному та відповідати загальній парадигмі за якою ведеться розробка проекту.

Також, такі системи повинні відповідати низці вимог, а саме:

– бути універсальними, тобто підтримувати роботу з будь-якими наявними структурами коду, а не бути обмеженими окремими випадками;

– забезпечувати оцінку системи на різних рівнях, мати можливість оцінювати як окремі модулі так і систему в цілому.

Актуальність теми роботи полягає у потребі розробки системи, яка б дозволяла проводити оцінку коду та архітектури програмних проектів.

Об’єктом дослідження є процеси оцінки ризиків та забезпечення якості при розробці програмних продуктів.

Предмет дослідження – методи, механізми та принципи оцінки ризиків та якості коду програмних продуктів.

Завданнями роботи є:

– дослідити існуючі набори принципів програмування та методи оцінки якості коду, виділити можливі проблеми та шляхи їх вирішення;

– охарактеризувати структуру проведення оцінки якості програмних систем та базову модель аналізу якості програмного коду;

– на основі проведених досліджень сформувати основні функціональні та нефункціональні вимоги, визначити функції які система повинна виконувати;

- підвести підсумки про необхідність розробки системи;
- розробити програмний комплекс, який реалізуватиме розроблені алгоритми;
- провести практичне тестування роботи розробленої системи.

1. ТЕОРЕТИЧНІ ОСНОВИ ДОСЛІДЖУВАНОЇ ПРОБЛЕМИ

1.1 Аналіз предметної області і виявлення наявних проблем та завдань

На сьогоднішній день галузь розробки ПЗ є однією з найбільших галузей світової економіки. Необхідність в автоматизації та діджиталізації торкнулась майже всіх сфер людського життя і ця потреба лише продовжує зростати. Відповідно, зростають і вимоги до якості програмного забезпечення.

Такий швидкий розвиток IT галузі мав також і негативні наслідки для індустрії. В сучасних реаліях існує криза сфери розробки програмного забезпечення. Так, велика кількість IT проектів завершується зі значними перевитратами або навіть провалом. Згідно досліджень, близько 53% проектів закінчується з значними перевитратами фінансів та перевищеннями термінів [2]. Окрім того, близько 18% програмних проектів завершуються провалом. Отже, криза IT галузі є досить серйозною проблемою, яка потребує нових рішень для забезпечення зменшення імовірності невдачі проектів.

Однією з ключових причин виходу проекту за відведені рамки бюджету та часу є низька якість написаного коду. Якість коду визначається різними критеріями, багато з яких мають значення лише для людини. Наприклад, зрозумілість формату кодування та структури проекту не мають значення для комп’ютера, але мають велике значення для супроводу та подальшої розробки проекту. Неякісно написаний код збільшує складність всієї подальшої розробки, що і виливається в вище зазначені перевитрати на розробку.

Іншою проблемою, яка виникає в наслідок неякісно написаного коду є складність залучення нових спеціалістів до роботи на проекті. Заплутаність коду збільшує час, необхідний новому члену команди для того щоб розібратись в проекті і повноцінно влитись в процес розробки. Таким чином виникає ситуація, коли розробка іде занадто повільно, і в той же час залучення нових спеціалістів не вирішить проблему повноцінно. Єдиним оптимальним

рішенням може бути повноцінний рефакторинг проекту, що також спричинює додаткові витрати грошей та часу.

Для забезпечення якості коду розроблюваного програмного проекту було розроблено багато методів та принципів написання коду. Основний стандарт моделі якості ПЗ ISO/IEC 25010 [3] виділяє зокрема зручність супроводу як один з елементів моделі якості. Детальніше структуру цього елементу представлено у таблиці 1.1.

Таблиця 1.1 – зручність супроводу у стандарті ISO/IEC 25010

Модульність	Рівень розділення системи на окремі компоненти таким чином, що зміна в одному з них має мінімальний вплив на інші
Повторна використовуваність	Рівень можливості використання компонентів в більше ніж одній системі або для побудови нових компонентів
Аналізованість	Зручність проведення аналізу помилок, дефектів і недоліків, а також зручність аналізу необхідності змін і їх можливих наслідків.
Модифікованість	Ступінь можливості ефективної модифікації без виникнення нових дефектів або погіршення функціональних якостей проекту
Тестованість	Ступінь ефективності встановлення тестових критеріїв та можливості виконання тестів з метою перевірки виконання встановлених критеріїв

Таким чином, існує потреба в методиках які б покращили якість написання коду. На сьогоднішній день розроблено багато методів та принципів покращення якості коду та структури програмних проектів, проте для забезпечення ефективних результатів важливо підходити до запровадження таких засобів з увагою до забезпечення їх гармонійного поєднання та попередження виникнення конфліктів між ними.

1.2 Порівняльний аналіз переваг та недоліків існуючих рішень

Загалом, методи по покращенню якості розробки програмних продуктів можна розділити на наступні категорії

- застосування архітектурних принципів;
- застосування стандартів кодування;
- використання метрик;
- оптимізація та рефакторинг.

Застосування архітектурних принципів є найбільш широким поняттям серед вищенаведених. Архітектурні принципи передбачають застосування якогось визначеного підходу при побудові програмної системи. Класичний приклад таких принципів надає Об’єктно-орієнтована парадигма програмування. Основні принципи об’єктно-орієнтованої парадигми надають вказівки по побудові архітектури системи. Таким чином об’єктно-орієнтована архітектура надає можливість групування сутностей та алгоритмів відповідно до їх призначення у класи та сімейства класів. І хоча об’єктно-орієнтоване програмування не являється універсальним рішенням, оскільки в специфічних випадках ця парадигма може програвати іншим в ефективності реалізації, на сьогоднішній день саме ООП є найбільш поширеною при розробці комерційних проектів.

Проте, саме по собі використання об’єктно-орієнтованої парадигми не означає що розробник застрахований від написання неякісного коду. Тому

було розроблено велику кількість наборів принципів програмування, які спираючись на принципи об’єктно-орієнтованого програмування розширяють їх, та надають більш детальні вказівки по побудові якісної архітектури програмних систем. Найбільш популярний з цих принципів є принцип SOLID, представлений Робертом Мартіном [4]. Основний акцент в цій методології робиться на важливості чіткого розмежування класів та сімейств класів за їх призначеннями, а також забезпечення універсальності в роботі цих класів. В цих ідеях простежуються такі базові принципи об’єктно-орієнтованого програмування як інкапсуляція та поліморфізм.

Користь цього набору методів полягає в формуванні чіткої архітектури проекту, яка є досить зрозумілою та гнучкою. Гнучкість є особливо важливою в сучасних умовах, оскільки на сьогоднішній день розробка найчастіше ведеться за еволюційними моделями життєвого циклу, що передбачає мінливість вимог до розроблюваного продукту. Дослідження показують, що застосування принципів SOLID дозволяє значно зменшити зчеплення та збільшити зв’язність системи [9].

Іншим засобом підтримки якості коду є застосування стандартів кодування. Стандарти кодування – це набори правил, якими керуються програмісти під час форматування коду. Основною задачею стандартів кодування є забезпечення читабельності та зручності розуміння коду. Хоча ці показники не мають значення для комп’ютера, що виконує програму, форматування коду в контексті якості коду розглядається майже на одному рівні з коректністю та оптимальністю алгоритмів. Причиною цього є те, що в основному на проекті працює більше ніж один програміст, а тому для того щоб їм було легше розуміти код один одного доцільно дотримуватись якогось одного стандарту кодування. Навіть якщо над кодом працює лише одна людина, у випадку якщо потрібно буде повернутись до давно написаного коду, може статися так, що через погане форматування програмісту буде важко зрозуміти навіть свій власний код.

Зазвичай, існує певний стандарт кодування, яким користується більшість програмістів, що працюють з певною мовою. Також, існують певні стандарти, які не залежать від мови програмування. Фахівцями [5] відзначаються такі прийоми форматування як відображення спорідненості чи відмінності через вертикальну щільність та відстань або відображення ієрархії коду шляхом використання відступів. Окрім того, в команди розробників можуть існувати свої власні стандарти кодування. Загалом, важливим є те, щоб всі на проекті дотримувались якогось одного стилю форматування коду, для того щоб код був однорідним та легко зрозумілим всім членам команди, а також будь-кому кому в майбутньому доведеться працювати з цим кодом.

Сучасними авторами також відзначається висока ефективність використання метрик в області прогнозування якості програмних продуктів. Метрики поділяють на метрики процесів та метрики кодування [7]. Метрики процесів відносяться до властивостей самого процесу розробки, а метрики кодування – безпосередньо до програмного продукту, який розробляється. Такі метрики часто застосовуються на проектах з метою оцінки продуктивності розробки та визначення слабких місць проекту і шляхів їх покращення та оптимізації.

Метрики кодування дозволяють отримати певні числові значення, які характеризують аналізований програмний модуль. Одним з найстаріших прикладів таких метрик є метрика кількості рядків. Оцінка кількості рядків, незважаючи на свою простоту, також може бути корисною. Одним з варіантів застосування цієї характеристики може бути підтримка оптимального об’єму програмних модулів, щоб вони не були занадто великі або навпаки, занадто роздріблені. Більш складні метрики включають в себе такі метрики як метрика Холстеда та метрика Чепіна. Ці метрики спираються на певні логічні конструкції всередині коду, що дозволяє більш детально оцінити складність програмного модуля.

Варто зазначити, що ці метрики потрібно розглядати не в відриві від оцінюваних модулів та інших метрик, а як частину цілісної картини. Висновок

про модуль на основі використання метрик можна зробити лише шляхом порівняння та співставлення його показників з показниками інших частин програмної системи.

Іншими важливими параметрами програмних систем є зчеплення і зв’язність. Зчеплення – це зовнішня характеристика модуля, вона виражає тип зв’язку модуля з іншими модулями. В ідеальному випадку модулі системи повинні мати мінімальне зчеплення – тобто мати мінімум зв’язків з іншими модулями. Зв’язність – це внутрішня характеристика модуля, яка описує рівень спорідненості елементів модуля. Добрим результатом проектування системи є модулі з низькими показниками зчеплення та високими показниками зв’язності – таким чином зміни одного модуля не спричиняють потреби змін в іншому модулі. Авторами було доведено переваги такого підходу [10] – легкість в розумінні, відлагодженні та внесенні змін системи розроблені з дотриманням таких показників.

Одним із найбільш важливих аспектів якості програмних продуктів та систем, як на сьогоднішній день, так і на протязі всього існування програмної інженерії як галузі є їх продуктивність та швидкодія. Здатність програми обробляти дані різних об’ємів в прийнятний для поставленої цілі час завжди була однією з найголовніших цілей забезпечення якості програмних систем. В сучасних реаліях потреба в забезпеченні прийнятної швидкодії системи зросла ще більше.

Існує декілька основних цілей, для яких може вестись оптимізація коду. Це може бути:

– покращення швидкодії;
– покращення відмовостійкості;
– оптимізація використання ресурсів системи;
– оптимізація об’єму коду.

Найбільш поширені оптимізаційні процеси – це оптимізації швидкодії. Швидкодія важлива майже в будь-яких програмних системах. В системах реального часу швидкодія важлива, оскільки вся система базується на

забезпеченні стабільної роботи програми за чітко визначені проміжки часу, будь які відставання чи провисання в цих випадках недопустимі. В більш традиційних програмах, які розраховані на рядових користувачів, швидкодія хоча і не має такого абсолютного значення як в системах реального часу, проте в сучасних умовах в користувацьких додатках UI-UX відіграє особливо важливу роль, оскільки він є ключовим аспектом за яким здійснюється конкуренція між декількома додатками з подібним призначенням.

Для покращення швидкодії існує декілька прийомів. Найбільш поширеним може бути розбиття однієї великої команди на декілька менших, які сумарно займатимуть менше часу. Наприклад, в деяких випадках може бути доцільно розділити один великий запит до бази даних на декілька менших, які отримуватимуть ті самі дані. Таким способом можна значно пришвидшити роботу додатку, зберігаючи при цьому весь функціонал.

Також, в коді можуть міститись надлишкові команди. Це можуть бути змінні, яким присвоюється значення, яке ніде не використовується, недоцільні логічні перевірки, зайві запити до бази даних, тощо. Загалом, гарним тоном вважається забезпечення чистоти коду від таких випадків, оскільки вони не тільки сповільнюють роботу програми, але і погрішують розуміння коду, відволікаючи від більш важливих компонентів [5].

Під час проведення такого рефакторингу варто мати на увазі також якість коду. Якість коду передбачає не лише його функціональні характеристики, але і такі властивості як читабельність, зрозумілість та складність. Гарним методом оцінки таких характеристик є використання метрик кодування, які вже були описані вище. Варто зазначити, що добре оптимізований код найчастіше вже має принаймні в початковому рівні відповідність більшості стандартів кодування. І навпаки, код, написаний згідно загальноприйнятих методів найчастіше вже є в міру оптимізованим.

Ще одним методом оптимізації є передбачення результатів. Можна передбачати наступні кроки користувача, і відповідно до того проводити наступні обчислення до того як вони стануть потрібні. Іншим варіантом

застосування таких методів може бути побудова лукап-таблиць для частих результатів функцій. Прикладом такого застосування може бути створення таблиць частих значень для функції яка обчислює тригонометричну функцію. Перед тим як проводити обчислення функція співставить аргументи з таблицею, і якщо відповідні дані будуть знайдені, функція одразу поверне значення не проводячи більш складних обчислень.

Окрім того, існують прийоми покращення часу відповіді інтерфейсу. На відміну від звичайних методів оптимізації швидкодії, вони не направлені на безпосереднє пришвидшення виконання операцій, а на зменшення часу неактивності інтерфейсу користувача. Такі прийоми направлені на створення видимості високої швидкодії, що покращує досвід користувача у роботі з додатком. Забезпечення відповідного інтерфейсу з низьким часом неактивності значно покращує використовуваність та дружність інтерфейсу, що є одними з основних вимог при розробці інтерфейсів користувача програмних систем [5].

Іншим прийомом покращення швидкодії є вставка коду на ассемблері, або іншій низькорівневій мові програмування. В цьому випадку команди будуть виконуватись напряму, без обробки їх компілятором. Проте, таких підхід містить також ряд недоліків. Досить важливим недоліком такого підходу є більша складність та неоднорідність отримуваного коду. Такий код може бути важче для розуміння ніж програма, написана на одній мові програмування. Тому, такий прийом застосовують найчастіше лише у випадках крайньої необхідності.

Оптимізація для покращення стабільності передбачає забезпечення стабільної роботи програми в усіх випадках. Основним фокусом цього напряму оптимізації є забезпечення обробки виключних випадків та збереження стабільності роботи при навантаженні. Також, у випадку якщо в системі використовуються певні зовнішні компоненти, наприклад бази даних або зовнішні API, варто передбачити можливість функціонування програми у випадку якщо ці компоненти будуть недоступні з певних причин.

Оптимізація використання ресурсів системи означає забезпечення раціонального використання обчислювальних потужностей пристрою, на якому працюватиме програма. До таких ресурсів належать об’єм оперативної пам’яті, ресурси процесору та відеопам’яті, пропускна здатність мережевого з’єднання та інші. Цей тип оптимізації найбільш актуальний в користувацьких додатках, які встановлюються безпосередньо на пристрій користувача. Зменшення вимог до ресурсів користувача для встановлюваного додатку дозволить більшій кількості користувачів працювати з додатком, що відповідно принесе більше прибутку розробнику цього додатку. Веб додатки також отримують ряд переваг від цього типу оптимізації. Хоча в сучасному світі хмарних технологій залучення більш потужних серверів не є великою проблемою, оптимальне використання ресурсів може дозволити веб додатку працювати на менш потужному сервері, що зменшить витрати на підтримку роботи веб додатку. Окрім того, оптимізація використання ресурсів найчастіше призводить і до двох попередніх типів оптимізації. Чим оптимальніше система використовує ресурси, тим швидше вона буде працювати і тим менше імовірність виникнення помилки в наслідок недостачі тих чи інших ресурсів системи. Проте, цей тип оптимізації є найскладнішим з усіх наведених, оскільки він потребує від розробника глибоких знань про роботу мови програмування та використовуваних фреймворків.

Більш специфічним випадком оптимізації є оптимізація об’єму коду. Оптимізація об’єму коду в основному призначена для програм які запускаються на різноманітних мобільних та смарт пристроях, в яких об’єм внутрішньої пам’яті вкрай обмежений. Такі оптимізаційні процеси часто можуть мати негативний вплив на стабільність та відмовостійкість системи, проте в цих випадках такі кроки можуть бути необхідними.

Таким чином, ми отримуємо три основних напрями оптимізації програмних систем – оптимізація швидкодії, оптимізація відмовостійкості та оптимізація об’єму коду. Усі ці напрями в деякій мірі конфліктують один з іншим – оптимізація об’єму коду може мати негативні наслідки для швидкодії

та відмовостійкості, оптимізація швидкодії може зменшити стабільність і т.д. Тому, при розробці програмного проекту варто визначати пріорітети для кожного з цих напрямів. Для більшої зрозумілості, їх варто визначати в графічному форматі, так як це наведено на рисунку 1.1.

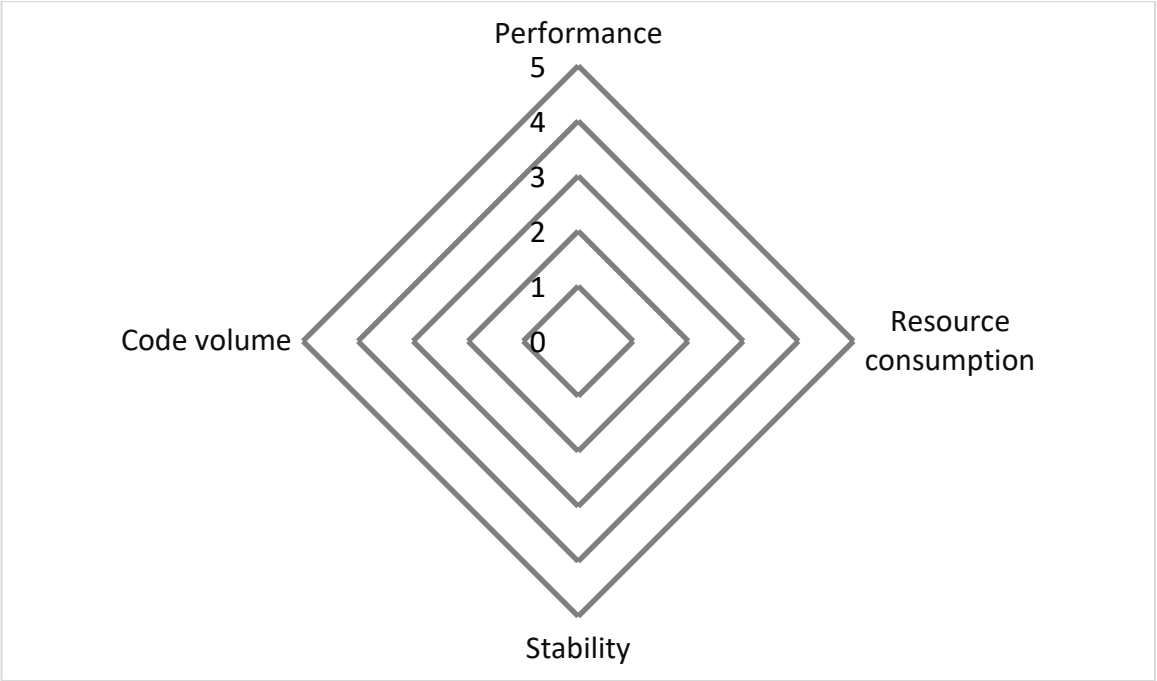


Рисунок 1.1 – Діаграма пріорітетів оптимізації

Отримана діаграма представляє порівняння трьох напрямів оптимізації за п’ятибальною шкалою, де 5 – максимальний пріорітет, а 0 – відсутність пріорітету (оптимізація є недоцільною). Таким чином можна наглядно продемонструвати важливість оптимізаційних процесів за кожним напрямом, і відповідно до того здійснювати подальше планування проекту. Доцільно обмежити максимальне сумарне значення пріорітетів значенням, яке не кратне 3, для того щоб не виникало ситуацій, коли усі пріорітети рівні, і відповідно жоден напрям оптимізації не є пріорітетним. Наприклад, доцільно було б встановити максимальне значення суми пріорітетів 8, таким чином якщо один з напрямів матиме максимальний пріорітет, то інші значення можуть мати лише найнижчі показники пріорітетності.

Хоча на кожному проекті ця діаграма набуватиме різних значень, відповідно до вимог, які представлені до розроблюваної програмної системи, можна виділити основні типи проектів та їх характеристики пріорітетів. Найпоширенішим типом проектів є звичайні користувацькі додатки, розраховані на рядових користувачів. Такі системи зазвичай мають графічні інтерфейси, а тому для забезпечення гарного досівду користувача необхідно подбати про відповідний рівень швидкодії. Окрім того, забезпечення стабільності також є важливим, оскільки більшість користувачів не мають відповідного досвіду та навичок для вирішення складних проблем, які можуть виникати в процесі роботи з додатком. Оптимізація використання ресурсів має помірний пріорітет, оскільки більшість десктопних додатків не виконують надзвичайно складних обчислень, проте потреба в більш раціональному використанні ресурсів системи може виникнути. Оптимізація об’єму коду не має особливого значення в цих випадках, оскільки ці системи зазвичай встановлюються на пристрої в яких дисковий простір не є сильно обмеженим. Таким чином, можна сформувати діаграму пріорітетів оптимізації, яка представлена на рисунку 1.2.

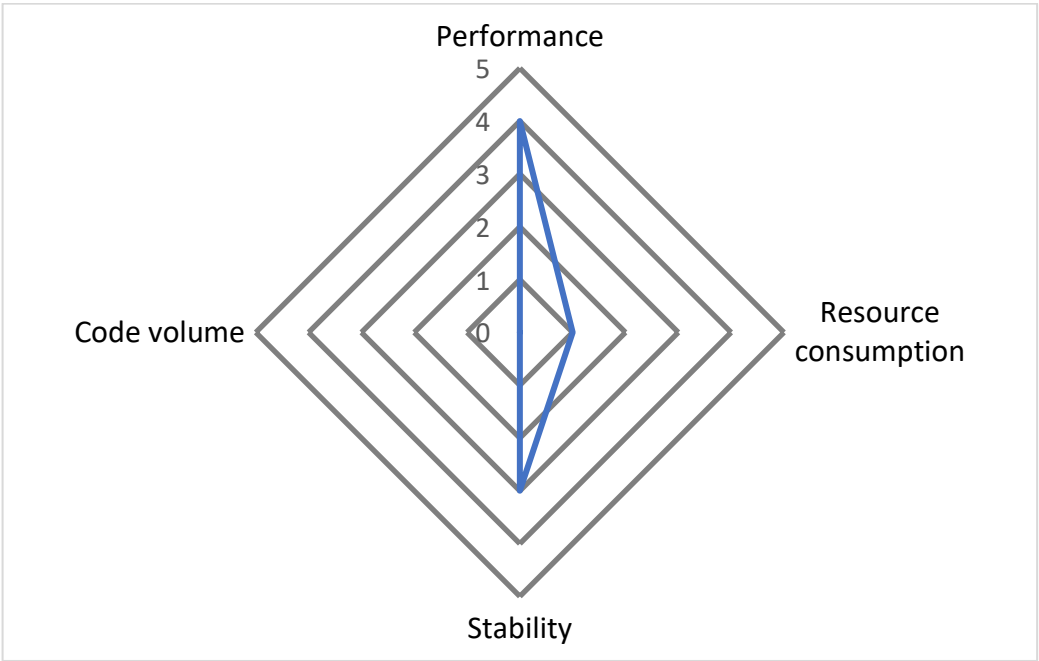


Рисунок 1.2 – Діаграма пріорітетів оптимізації для користувацьких додатків

Іншим прикладом програмних проектів є системи реального часу. В цих випадках більшого значення пріорітету набуватиме оптимізація стабільності. Хоча швидкодія в цих випадках також має важливість, виникнення помилок, які здатні зупинити чи значно погіршити роботу системи зазвичай приносять набагато гірші наслідки. Оптимізація використання ресурсів має нижчий пріорітет, оскільки системи реального часу зазвичай працюють на більш потужних пристроях. Оптимізація об'єму коду також не є доцільною, оскільки вона може призвести до втрат в продуктивності та стабільності, що не є допустимим для цього типу систем. Діаграма пріорітетів для такого типу проектів представлена на рисунку 1.3.

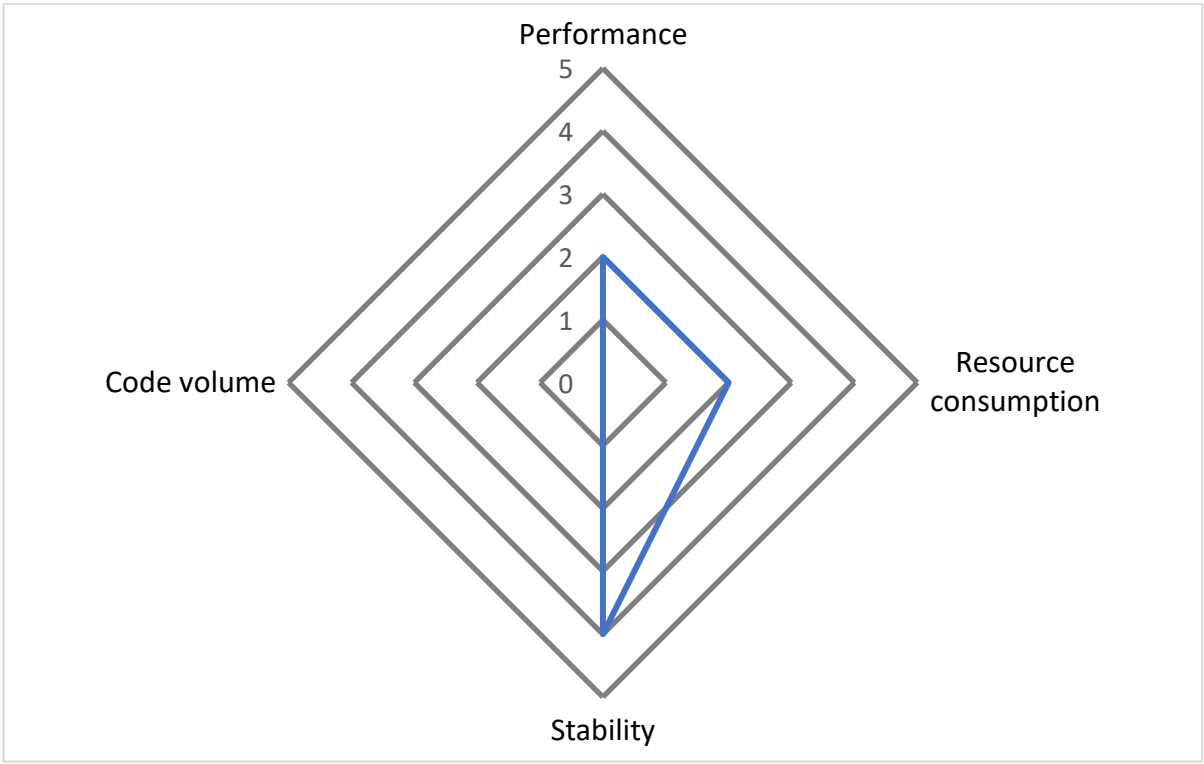


Рисунок 1.3 – Діаграма пріорітетів систем реального часу

І ще одним класичним типом програмних проектів є вбудовані системи. В цих випадках основним пріорітетом є зменшення об'єму коду, оскільки ці системи зазвичай функціонують в умовах обмеженого простору пам'яті. З цих же причин оптимізація використання ресурсів є важливою для цього типу програм. Оптимізація стабільності має помірний пріорітет, оскільки вбудовані

системи не завжди мають можливість коректно відобразити повідомлення про помилку, тому більшість виключних випадків повинні бути опрацьовані самою системою. Продуктивність в цих випадках зазвичай не є надзвичайно важливою, тому що ці системи зазвичай виконують менш складні обчислення ніж програми, призначені для роботи на звичайних комп'ютерах. Проте, в деяких випадках потужність пристроїв, на яких ці програми запускатимуться може бути обмеженою, і в цих випадках варто також приділити увагу оптимізації продуктивності. Загальний приклад діаграми пріорітетів для вбудованих систем представлений на рисунку 1.4.

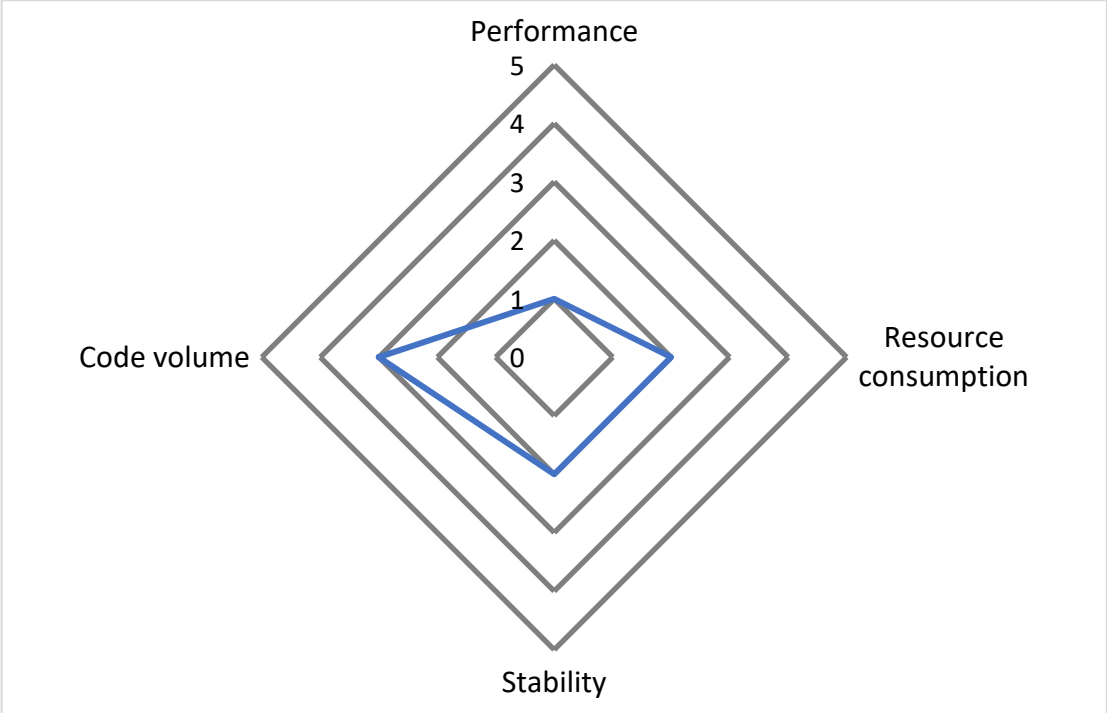


Рисунок 1.4 – Діаграма пріорітетів оптимізації вбудованих систем

Варто зазначити, що для кожного проекту діаграма пріорітетів оптимізації повинна будуватись індивідуально, оскільки в кожного проекту можуть бути свої вимоги, які відрізняються від загальних випадків. Побудову діаграми варто здійснювати на початку проекту, коли відомий тип системи та найбільш базові вимоги до неї. Спираючись на цю діаграму розробники зможуть вести розробку та рефакторинг відповідно до потреб проекту, та

приймати рішення про важливість тієї чи іншої характеристики коли під час розробки виникатимуть конфлікти між ними.

Таким чином, на сьогоднішній день існує багато різних методів покращення якості програмного забезпечення. Ці методи можуть бути застосовані на різних етапах розробки програмної системи. Так, наприклад, під час первинної розробки розробники зазвичай використовують стандарти кодування, що допомагає їм зберігати однорідність та зрозумілість коду та архітектури системи. Після завершення початкових процесів кодування використовують різноманітні метрики для оцінки якості модулів системи. При виявленні недоліків цим методом до проблемних модулів застосовують рефакторинг з метою покращення якості коду та оптимізації роботи системи для усунення наявних проблем. Залежно від моделі життєвого циклу проекту така послідовність дій може відбуватись один або декілька разів. Наприклад, в водоспадній моделі ці дії виконаються один раз строго послідовно. В більш сучасних еволюційних моделях цей процес варто застосовувати до кожної ітерації життєвого циклу проекту.

1.3 Висновок

Відповідно, в сьогоднішніх реаліях індустрії розробки програмного забезпечення існує потреба в розробці нових рішень для забезпечення якості розроблюваних програмних проектів. Знаходження нових рішень в цій сфері дозволить значно покращити існуючі моделі життєвого циклу програмних продуктів і збільшить імовірність успішного завершення проектів.

До можливих варіантів дослідження з метою знаходження нових рішень у цій сфері можна віднести метрики кодування та аналіз з метою оптимізаційних рефакторингів. Метрики кодування можуть бути вдосконалені, що дозволить їм більш точно надавати оцінку якості коду. Альтернативним варіантом розвитку в цьому напрямі може бути також

модифікування існуючих метрик. Це не покращить їх безпосередню ефективність, проте може дозволити використання модифікованих метрик з дещо іншим призначенням.

Оптимізаційні процеси також є перспективним напрямом дослідження. Існує потреба в визначенні пріорітетів різних видів оптимізаційних процесів у відношенні один до одного.

Важливість розробки нових рішень забезпечення якості коду полягає в наявності кризи індустрії розробки програмного забезпечення та необхідності засобів оминання та подолання проблем, з якими стикаються IT проекти різних масштабів.

2. АНАЛІЗ МЕТОДІВ ЗАБЕЗПЕЧЕННЯ ЯКОСТІ ПРОГРАМНИХ СИСТЕМ

2.1 Розробка метрик для оцінки якості програмного коду

Під час всього життєвого циклу розробки програмного продукту є потреба в постійному визначенні та контролі якості написаного програмного коду. Забезпечення контролю якості кодування несе багато переваг для проекту в цілому. По перше, при здійсненні такого нагляду за якістю розроблюваного проекту можливі проблеми, які виникають в процесі виявляються одразу і відповідно, є можливість на них реагувати набагато скоріше, що в свою чергу значно зменшить затрати часу на їх виправлення. По друге, можна відслідковувати зміни якості програмного коду на протязі всього життєвого циклу, що також може бути корисним для менеджменту проекту.

Для цих цілей було розроблено багато метрик програмного забезпечення. Метрика програмного забезпечення – метод визначення певних характеристик розроблюваного програмного проекту шляхом обчислення параметрів проекту за допомогою математичних методів. В цілому, метрики призначені для визначення складності проекту та прогнозування об’ємів робіт.

Метрики є одними з найбільш простих, і в той же час найбільш легкозастосовуваних методів для оцінки якості програмного коду. Тому, основу системи для оцінки коду варто базувати саме на застосуванні метрик кодування. Ключові методи та аспекти застосування метрик можуть бути різні, проте загальним принципом використання значень метрик є співставлення значень метрик усіх модулів досліджуваної області проекту і визначення значних відхилень. Над елементами, що містять відхилення варто здійснити аналіз з метою визначення причини цих відхилень і можливості виправлення проблем, які можуть бути наявні в цих елементах системи.

Найпростішим класом метрик ПЗ є кількісні метрики. Ці метрики призначені для визначення кількісних характеристик вихідного коду.

Класичним представником цього класу є метрика кількості рядків коду (SLOC – Source lines of code). Ця метрика може використовуватись для визначення трудозатрат по проекту. Проте, з появою мов програмування що підтримують запис декількох команд в одному рядку, ця метрика стала застарілою. Їй на зміну прийшла метрика кількості логічних рядків коду, тобто кількість команд. Ще однією можливою альтернативою вимірювання цієї метрики може бути підрахунок кількості операторів в коді. Таким чином, визначається не тільки об’єм коду, але і в певній мірі і об’єм виконуваної програмою роботи. Проте, усі ці метрики значно залежать від особливостей мови програмування, оскільки різні мови можуть потребувати різної кількості рядків та операторів. Тому, при оцінці цих метрик важливо враховувати специфіку використаної мови. В зв’язку з цим, ці метрики погано підходять для порівняння програм, написаних різними мовами.

Також, до кількісних метрик відносять метрики Холстеда [5]. При використанні метрик Холстеда частково компенсуються недоліки, пов’язані з записом однієї і тієї ж операції різною кількістю рядків. Багато авторів відзначають метрику Холстеда як один з найпоширеніших методів прогнозування надійності програм [9].

Метрику Холстеда можна вдосконалити, скориставшись методами з метрики Чепіна [6]. Обчислення метрики Чепіна передбачає врахування вагових коефіцієнтів, які залежать від типів змінних. Метрика Чепіна поділяє змінні на 4 типи - введені змінні для розрахунків та забезпечення виводу; модифіковані, або створені всередині програми змінні; управляючі змінні; невикористані змінні. Вагові коефіцієнти, на думку автора метрики, розподіляються наступним чином — $a_1=1$, $a_2=2$ $a_3=3$, $a_4=0,5$. Ці самі коефіцієнти можна враховувати при обчисленні кількості операндів в метриці Холстеда. Аналогічно, для підрахунку операторів можна також використати вагові коефіцієнти, які будуть рівні кількості аргументів оператора. Таким чином, формули метрики Холстеда приймуть вигляд, як представлено на формулах:

$$n_1 = a_1 \cdot p + a_2 \cdot m + a_3 \cdot c + a_4 \cdot t \quad (1.1)$$

$$n_2 = \sum k_i \cdot i \quad (1.2)$$

$$N_1 = a_1 \cdot P + a_2 \cdot M + a_3 \cdot C + a_4 \cdot T \quad (1.3)$$

$$N_2 = \sum K_i \cdot i \quad (1.4)$$

де p – кількість змінних для розрахунків та забезпечення виводу, m – кількість модифікованих, або створені всередині програми змінних, c – кількість управляючих змінних, t – кількість невикористаних змінних, k_i – кількість операторів з кількістю аргументів і, i – кількість аргументів, P – кількість використань змінних для розрахунків та забезпечення виводу, M – кількість використань модифікованих, або створені всередині програми змінних, C – кількість використань управляючих змінних, T – кількість повторів невикористаних змінних, K_i – кількість використань операторів з і аргументів.

Таким чином, ми отримуємо вдосконалену метрику Холстеда, яка буде враховувати не тільки кількісні характеристики коду, але і складність потоку управління даними. Такий підхід дозволить більш точно оцінити складність класу або модуля програмної системи.

Інший варіант метрики для оцінки якості коду можна базувати на оцінці складності алгоритмів вимірюваного модуля. Складність алгоритмів зазвичай визначають для функцій або модулів. При визначенні складності алгоритму не береться до уваги реальний час виконання операцій на реальних наборах даних, оскільки він залежить від багатьох факторів, таких як потужність процесора, об’єм пам’яті і власне дані в наборі даних. Натомість, визначається більш абстрактна величина, яку можна охарактеризувати як складність при кількості вхідних даних наближених до нескінченності.

Варто зазначити, що сама по собі висока складність алгоритму ще не свідчить про наявність якихось проблем у програмному коді. Проте, якщо в певному модулі або класі спостерігається значення складності яке значно

відрізняється від середнього по всьому проекту або модулю, це може свідчити про те, що алгоритм є занадто обширним, і над ним може бути проведена декомпозиція. Звертати увагу слід саме на алгоритми складність яких є набагато вищою ніж середнє значення. Досить часто прості набори операцій, які мають низьку складність є часто повторюваними, тому їх виносять в окремі функції або модулі. І хоча подібна декомпозмція може іноді здаватись надлишковою, наприклад якщо така ділянка коду використовується лише в одному місці, в певних випадках це може бути використане в подальшому, особливо якщо проект все ще на стадії розробки. Тому основну увагу слід приділяти алгоритмам, складність яких сильно відрізняється від середнього значення в більшу сторону.

Таким чином, метрику можна виразити формулою

$$C_n = O_n - O_c \tag{1.5}$$

де C_n – відносна складність оцінюваного модулю, O_n – складність оцінюваного модулю, O_c – середня складність всіх модулів.

Дану метрику можна застосовувати як до модулів, так і до окремих методів. У випадку з методами, доцільно було б порівнювати їх не з всіма іншими методами, а з методами, що знаходяться в одному класі. Таким чином відбуватиметься більш точна оцінка відносної складності алгоритмів.

Іще одним важливим методом для оцінки якості коду програмної системи є визначення зчеплення та зв’язності програмних модулів. Одним з можливих варіантів покращення визначення рівня зчеплення є визначення не тільки типу зчеплення, але і кількості зв’язків з зовнішніми модулями. Це дозволить виявляти модулі, які мають зчеплення низьких рівнів, проте використовують його занадто часто, що може спричинити потребу в надлишковій роботі при внесенні змін в зчеплені модулі.

Таким чином, формула обчислення зчеплення модуля набуде наступного вигляду:

$$C = k_c * n \quad (1.6)$$

де C – показник зчеплення, k_c – коефіцієнт типу зчеплення, n – кількість посилань на зовнішні модулі.

Значення коефіцієнту зчеплення k_c доцільно буде прийняти згідно таблиці 2.

Таблиця 2.1 – Значення коефіцієнтів зчеплення

Тип зчеплення	k_c
Зчеплення за даними Модуль А здійснює виклик модуля В. Параметри – прості типи даних	1
Зчеплення за зразком Модуль А здійснює виклик модуля В. Параметри – структури даних	3
Зчеплення за управлінням Модуль А здійснює виклик модуля В і передає управляючі аргументи	4
Зчеплення за зовнішнім посиланням Модулі А і В посилаються на одну зовнішню змінну простого типу	5
Зчеплення за пам’яттю Модулі А і В посилаються на одну зовнішню структуру даних	7
Зчеплення за вмістом Один модуль прямо посилається на зміст іншого модуля не через його точку входу	9

Таким чином, коефіцієнт враховуватиме тип зчеплення і значення метрики зчеплення модуля буде змінюватись в залежності від кількості

зв’язків між модулями, що збільшить точність визначення проблемних місць в модульній структурі програмної системи. Відповідно, основну увагу слід приділяти модулям з великими показниками зчеплення, звертаючи увагу на типи зв’язків між модулями.

Проте, застосування подібного підходу для визначення показників зв’язності модулю не буде доцільним. Кількість внутрішніх зв’язків між елементами модуля не мають такого важливого значення як кількість зовнішніх зв’язків між модулями, оскільки для більшості типів зв’язності кількість зв’язків не має такого явного значення.

Тому, доцільно буде застосовувати класичний метод визначення зв’язності, який виражатиметься в вигляді числового значення, яке залежить від характеру внутрішніх зв’язків.

Таким чином, можна оцінювати рівень зчеплення та зв’язності розроблюваної системи. Ці характеристики є одними з основних показників якості побудови архітектури системи та якості написаного коду. Визначення рівня зчеплення та зв’язності системи дозволить отримати оцінку про загальносистемну якість архітектури розроблюваної системи, що в свою чергу, дозволяє проводити моніторинг ключових аспектів розробки, таких як стан архітектури системи, складність створених програмних структур, визначення слабких місць архітектури розроблюваного додатку і багато іншої корисної інформації, яка стане у нагоді команді розробників в процесі розробки та підтримки програмного проекту.

Загалом, визначені метрики дозволяють отримати корисні дані про елементи розроблюваної програмної системи та проект в цілому. Метрики зчеплення та зв’язності дозволяють отримати найбільш ключові характеристики архітектури системи, а вдосконалений метод визначення зчеплення дозволяє більш точно визначати міру зчеплення модулів шляхом врахування кількісних характеристик зовнішніх зв’язків, які наявні в модулі. Кількісні метрики дозволяють отримати більш загальну інформацію про написаний код. Вдосконалена метрика Холстеда забезпечує визначення

кількісних якостей коду з врахуванням потоку управління даними, порівняння цієї метрики з середніми показниками по проекту може допомогти виявити аномалії в написаному коді. Метрики складності алгоритмів можуть допомогти визначити місця в які можуть потребувати оптимізації з точки зору використання ресурсів програмної системи.

Таким чином, розроблений комплекс метрик допомагає визначити та локалізувати найбільш поширені проблеми програмних проектів.

2.2 Алгоритм визначення пріорітету оптимізації проекту

Оптимізаційні процеси є важливою частиною будь-якого проекту. Майже завжди початковий варіант написаного коду не є оптимальним і потребує додаткового рефакторингу. Окрім того, інформація про пріорітетність тих чи інших аспектів оптимізації може стати в нагоді розробникам ще на процесі первинної розробки функціоналу, оскільки ця інформація може підказати на які показники слід звернути увагу в першу чергу при написанні та відлагодженні нового коду. Це в свою чергу може запобігти потребі в глибокому рефакторингу з метою оптимізації, оскільки основні вимоги будуть враховані ще на етапі розробки.

Отже, існує потреба в визначенні пріорітетів напрямів оптимізації. В певних випадках схема пріорітетів можу бути досить очевидною. Наприклад, якщо вимоги до системи чітко вказують важливість якості того чи іншого аспекту оптимізації. Проте, в деяких випадках вимоги можуть бути менш специфічними, і виникатиме потреба в побудові структури пріорітетів виходячи з різноманітних даних про проект.

В таких випадках корисною була б експертна система, яка б допомагала визначити значення пріорітетів оптимізаційних процесів для проекту. На вхід такій системі надавалися б певні дані про проект, такі як тип проекту, параметри середовища роботи системи, очікувана кількість користувачів,

тощо. Виходячи з цих даних система б опрацьовувала параметри та виконувала їх аналіз, результатом якого був би висновок про важливість параметрів оптимізації програмного проекту.

Для того щоб побудувати модель роботи такої експертної системи необхідно спочатку визначити якими будуть вхідні дані, які вводитиме користувач. Можливий набір даних наведений в таблиці 2.2.

Таблиця 2.2 – Вхідні параметри експертної системи

Параметр	Опис значення
Тип проекту	Один з декількох можливих варіантів загального формату проекту
Платформа	Один з декількох можливих варіантів середовища, в якому буде працювати проект
Параметри середовища	Технічні параметри середовища в якому працює проект
Очікувана кількість користувачів	Діапазон очікуваного числа користувачів
Додаткові параметри	Додаткова інформація про проект, наприклад важливі типи даних користувачів, з якими працює додаток (особиста інформація, банківські дані), сфера застосування, потреба в безперебійному підключенні до мережі інтернет, тощо.

Першим та головним вхідним параметром є тип проекту. Як вже було зазначено, різні типи проектів можуть мати різні потреби в оптимізації, як наприклад системи реального часу мають більшу потребу в забезпеченні стабільності роботи, а вбудовані системи – в мінімізації об’єму коду та використанні ресурсів. Тому, цей параметр є ключовим у прийнятті рішення

про пріорітетність оптимізації того чи іншого аспекту якості системи. Можливими варіантами цього параметру будуть:

– десктопний додаток;
– веб-додаток;
– система реального часу;
– вбудована система.

Наступним кроком є визначення типу платформи на якій працюватиме система. Цей параметр також є важливим, оскільки він буде визначати, які параметри будуть прийматись до уваги на наступному кроці – параметри середовища системи. Наприклад, якщо система запускатиметься на хмарному середовищі, важливими будуть такі параметри як обсяг пам’яті для бази даних, а у випадку якщо це смарт-пристрій, то більш важливими будуть вже ресурси оперативної пам’яті та процесору. Можливими варіантами значення цього параметру будуть:

− ПК;
− хмарне середовище;
− смарт-пристрій.

Варто зазначити, що тип платформи «ПК» включатиме в себе як стаціонарні комп’ютери та ноутбуки, так і мобільні пристрої, такі як смартфони або планшети, оскільки в сучасних умовах усі ці пристрої мають відносно схожі параметри в питаннях, які стосуються ресурсів системи.

Визначення параметрів середовища є логічним продовженням наступного кроку. Загалом, цей параметр можна охарактеризувати як параметри наявного хмарного середовища або смарт-пристрою на якому буде запускатись система, або мінімальні системні вимоги, якщо на попередньому кроці була обрана платформа типу «ПК».

Іще одним важливим параметром експертної системи є очікувана кількість користувачів. В поєднанні з такими параметрами як тип та параметри середовища, його можна застосувати для визначення пріорітету важливості оптимізації швидкодії.

Додаткові параметри містять у собі інформацію про те, з якими даними працює додаток. Деякі типи даних, як наприклад банківська інформація або особисті дані користувачів потребують більшого рівня надійності від системи. Іншим типом даних що може бути включеним в додаткові параметри є великі дані. Інформація про те, чи працює додаток з великими даними може значно змінити очікувані пріорітети оптимізації аспектів системи. Так, наприклад, системи які працюють з великими даними зазвичай мають потребу в оптимізації швидкодії, оскільки надзвичайно великі обсяги даних повинні оброблятись за відносно невеликі для таких об’ємів проміжки часу, щоб не втратити свою актуальність.

Таким чином, можна побудувати модель експертної системи, яка визначатиме пріорітети варіантів оптимізації розроблюваного програмного проекту. Схема роботи такої системи у вигляді блок-схеми представлена на рисунку 2.1.

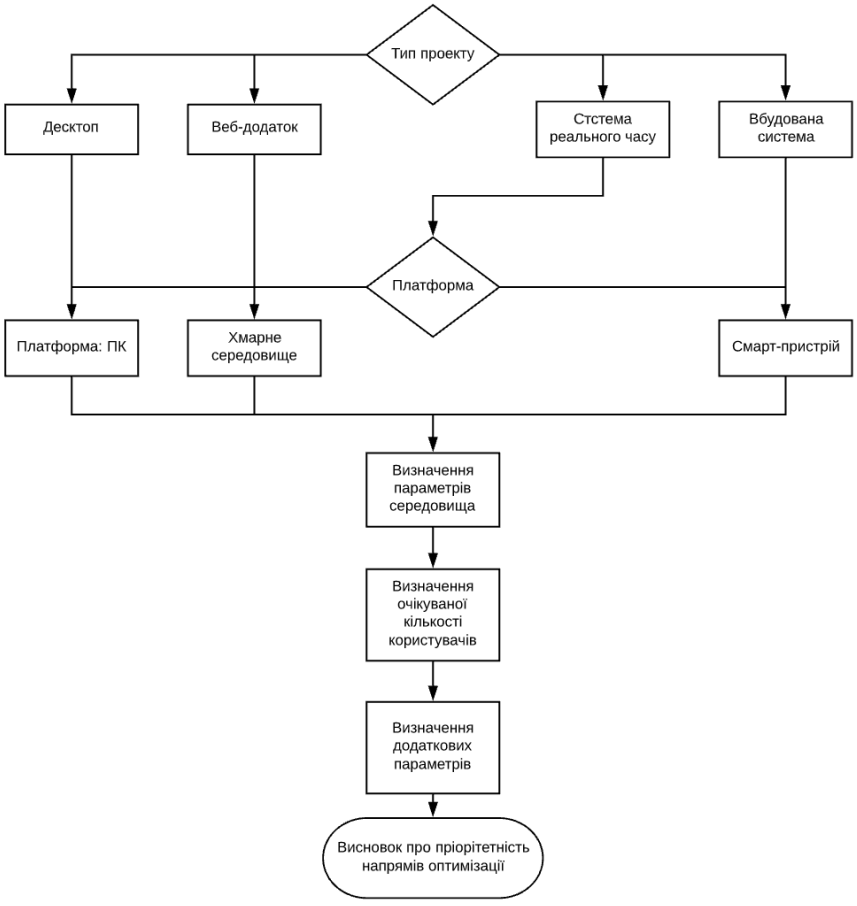


Рисунок 2.1 – Схема роботи експертної системи

Перший крок, крок визначення типу проекту надасть початковий шаблон можливого варіанту пріорітетів оптимізаційних процесів. В усіх наступних кроках цей шаблон прийматиметься за основу, і буде змінюватись відповідно до введених параметрів.

Наступний крок, крок вибору платформи, внесе невеликі корективи до загального шаблону. Тип платформи визначить, який напрям оптимізації матиме перевагу над іншими. Детально ця залежність продемонстрована в таблиці 2.3.

Таблиця 2.3 – типи платформи та їх пріорітетний напрям оптимізації

Тип платформи	Пріорітетний напрям оптимізації
ПК	Швидкодія
Хмарне середовище	Надійність
Смарт пристрій	Об’єм коду

Крок визначення параметрів середовища виконання допомагає визначити пріорітетність оптимізації використання ресурсів. В цілому, цю залежність можна охарактеризувати таким чином – чим більш обмежені ресурси тим більший пріорітет в оптимізації використання ресурсів. Основними видами ресурсів, які розглядатимуться системою будуть об’єм оперативної пам’яті, об’єм статичної пам’яті та кількість потоків процесора. Виходячи з цих даних про проект можна визначити залежність потреби оптимзації використання ресурсів від потужностей системи. Характеристики такої залежності наведені в таблиці 2.4.

Крок визначення очікуваної кількості користувачів допоможе визначити баланс між пріорітетами оптимізацій швидкодії та надійності. Цю залежність можна охарактерзувати наступним чином: для малої кількості користувачів більш важливою є надійність, оскільки такі системи зазвичай виконують відносно прості обчислення, і невелика кількість користувачів не призведе до значних збільшень вимог до продуктивності. Зі збільшенням кількості

користувачів зростатиме важливість швидкодії – система повинна буде обробляти дані значної кількості користувачів, а тому є потреба в збереженні швидкої роботи системи.

Таблиця 2.4 – Залежність пріорітету оптимізації ресурсів від типу системи

Тип потужності системи	Кількість потоків процесора	Об’єм оперативної пам’яті	Об’єм статичної пам’яті	Вплив на показник пріорітету оптимізації ресурсів
Слабка	1	до 4 ГБ	До 128 ГБ	+2
Помірної потужності	2	4 – 8 ГБ	128 – 512 ГБ	+1
Середньої потужності	2 – 4	8 – 12 ГБ	512 ГБ – 1 ТБ	0
Високої потужності	Більше ніж 4	Від 12 ГБ	2 ТБ і більше	0 (-1 якщо інші напрями оптимізації критично необхідні)

Проте, після подолання певної відмітки кількості користувачів важливість оптимізації надійності знову зростатиме. Це спричинене двома факторами. По перше, великі обсяги даних самі по собі потребують специфічних методів для успішної обробки, що в свою чергу забезпечить інший рівень швидкодії. По друге, системи з великою кількістю користувачів потребують додаткової уваги до надійності ще й тому, що такі системи мають більшу імовірність стати ціллю зловмисників, оскільки дані великої кількості користувачів можуть мати велику цінність для них. Таким чином, характер

залежності пріорітетів оптимізації до кількості користувачів можна охарактеризувати, як зображено на рисунку 2.2.

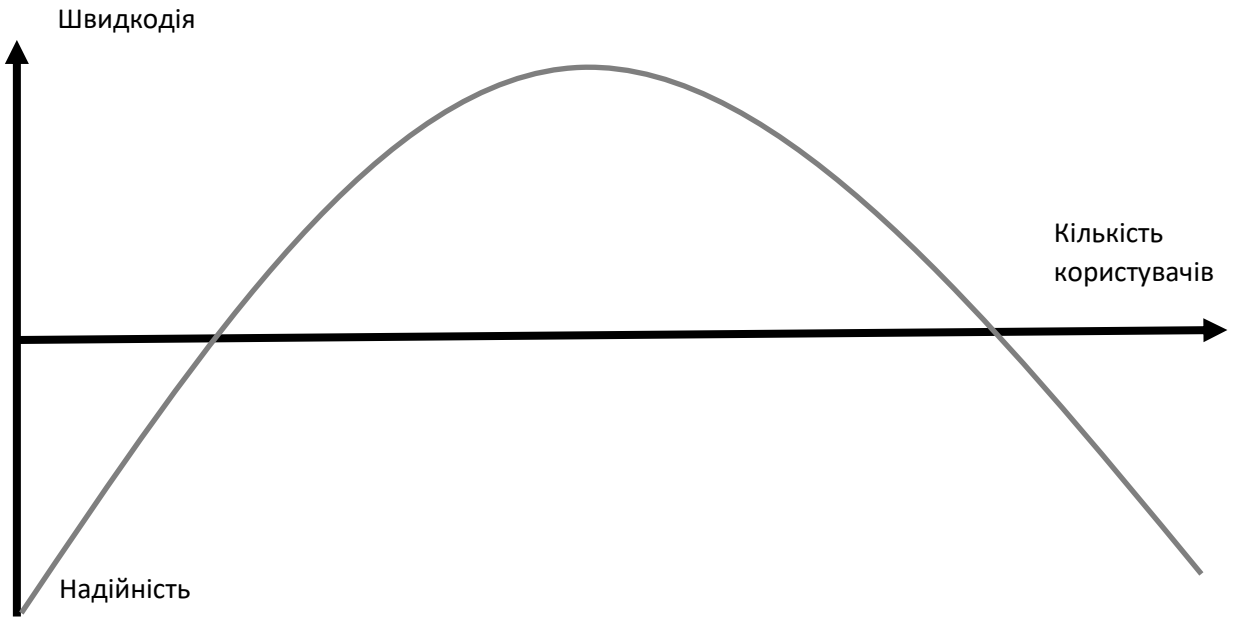


Рисунок 2.2 – Залежність пріорітетів оптимізації від кількості користувачів

Умовно наведену залежність можна розділити на 3 відрізки. Перший відрізок складатимуть системи з кількістю користувачів до 1000. Зазвичай це програмні системи призначені для внутрішнього використання у компаніях, і забезпечення надійності необхідне для уникнення витоків інформації за це обмежене коло користувачів, а обчислення які виконуватиме система зазвичай не є надзвичайно складними, оскільки вона оперуватиме даними меншої кількості користувачів. Тому доцільно буде збільшити показник пріорітету оптимізації надійності на 1 та за потреби зменшити показник пріорітету оптимізації швидкодії на 1, якщо сума всіх показників пріорітетів виходить більшою за 8, як того вимагає розроблений метод. Другий відрізок складатимуть системи з кількістю користувачів від 1000 до 1 000 000. Такі системи мають оперувати з даними великої кількості користувачів, а томув них виникає більша потреба в оптимізації швидкодії. Аналогічним чином, в таких випадках пріорітет оптимізації швидкодії збільшується на 1, а пріорітет

оптимізації надійності за потреби зменшується на 1. Третій відрізок складають системи з кількістю користувачів більше ніж 1 000 000. Як вже було зазначено, такі системи по замовчуванню потребують особливих методів роботи з даними та збільшення надійності через ризик стати жертвою зловмисників. Тому, показники пріорітетів змінюються за такою самою схемою як і для першого відрізку.

Останній крок, крок визначення додаткових параметрів дозволить підкоригувати дані про пріорітетність оптимізаційних процесів та зробити остаточний висновок про оптимізаційні потреби проекту. Кожен з додаткових параметрів має вплив на один з напрямів оптимізації. Так, робота проекту з банківськими даними означає, що проект має додаткову потребу в оптимзації надійності, а робота з великими даними – в оптимізації швидкодії. Кожен з цих параметрів додаватиме до значення пріорітету відповідного параметру оптимізації системи.

Після закінчення роботи цього алгоритму може виникнути ситуація коли сума значень всіх пріорітетів є більшою ніж 8 відповідно до розробленого методу. В таких випадках необхідно провести нормалізацію структури пріорітетів, зменшивши їх показники таким чином, щоб їх порядок посортований по значенню пріорітету не змінився.

Таким чином, сформована система допоможе більш точно визначити потреби оптимізації розроблюваного програмного проекту. Чітка ієрархія ключових аспектів якості програмного проекту є безперечною первагою в різних сферах процесу розробки. Визначення пріорітетів оптимізації допоможе як в безпосередній розробці та супроводі проекту так і в задачах по плануванню робіт та ітерацій на протязі всього життєвого циклу. Спираючись на цю структуру пріорітетів розробники зможуть вести розробку та рефакторинг відповідно до потреб проекту, та приймати рішення про важливість тієї чи іншої характеристики коли під час розробки виникатимуть конфлікти між ними.

Персонал менеджменту проекту також зможе використовувати отриману ієрархію пріорітетів в своїй роботі. До можливих варіантів використання цієї інформації для цілей менеджменту процесів розробки проекту можна віднести формування задач по рефакторингу та визначення пріорітетів виправлення існуючих помилок роботи системи відповідно до проблем, які вони створюють. Усі ці дії сприяють вдалому плануванню проекту як в короткостроковій перспективі, наприклад, в плануванні ітерацій в ітераційних моделях життєвого циклу проекту, так і в довогостроковій перспективі шляхом визначення ключових аспектів якості проекту для визначення цілей робіт по контролю якості програмного продукту.

Таким чином, результати роботи розробленої системи будуть сприяти вдалій розробці проекту на різних рівнях та в різних аспектах.

2.3 Висновок

Таким чином, у розділі проведено дослідження існуючих методів оцінки програмного коду, таких як метрики кодування. На основі досліджених метрик було розроблено вдосконалені метрики, які можуть допомогти в процесі оцінки кодів програмних систем.

Метрику Холстеда було вдосконалено шляхом застосування методів з метрики Чепіна. В формулу було введено додаткові коефіцієнти, які відображають різні типи змінних та їх вплив на складність потоку управління даними. Така модифікація дозволить враховувати ці показники при обчисленні метрики, та забезпечить відображення складності коду при оцінці трудомісткості проекту.

Метрику складності алгоритму також було модифіковано за допомогою врахування середнього значення складності алгоритмів в модулі. Обчислення такої відносної складності дозволить виявляти неоднорідності в структурі

алгоритмів модулю, що нерідко свідчить про проблеми в відповідних ділянках модулів.

Окрім того, було також вдосконалено метрику оцінки зчеплення модулю. Врахування кількості зовнішніх зв’язків модулю дозволить більш точно визначати можливі області проблем, пов’язаних з високими показниками зчеплення.

Також було розроблено метод оцінки пріорітетів оптимізаційних процесів під час розробки програмних систем. Розроблений метод базується на оцінці таких параметрів проекту як платформа, кількість користувачів та особливості даних з якими працює проект. Результатом аналізу є ієрархія пріорітетів напрямів оптимізаційних процесів. Результати роботи цього алгоритму можуть бути застосовані як командою розробки, для вирішення спірних ситуацій при рефакторингу проекту, так і менеджментом проекту для більш успішного планування проекту.

Наступним етапом розробки дипломного проекту є програмна реалізація розробленого алгоритму визначення пріорітетів оптимізації.

3. РОЗРОБКА ПРОГРАМНОГО РІШЕННЯ

3.1 Постановка технічного завдання

Отже, задача для розробки програмного проекту полягає в розробці програмної системи, яка б реалізувала функції експертної системи для визначення пріорітетів оптимізації розроблюваного програмного проекту.

Експертна система повинна виконувати наступні функції:

– забезпечувати введення користувачем параметрів проекту;
– виконувати валідацію введених даних на предмет невідповідності типу проекту та платформи;
– виконувати обробку даних відповідно до розробленого алгоритму визначення пріорітету;
– забезпечувати виведення даних в вигляді діаграми, представленої на рисунку 1.1.

Першочерговим питанням яке виникає при розробці будь-якого додатку є вибір мови програмування програмної реалізації. На сьогоднішній день існує багато різних мов програмування, більшість з яких забезпечують найбільш різноманітні можливості реалізації програмних систем, зокрема шляхом підтримки різних технологій та фреймворків. Для розробки нашої системи доцільно буде обрати мову програмування C#. C# є сучасною об’єктно-орієнтованою мовою програмування, яка підтримує широкий спектр технологій, що забезпечить свободу у виборі засобів програмної реалізації.

Іншим питанням яке виникає при плануванні розробки системи є вибір типу додатку. C# підтримує багато різних типів платформ та типів додатків. Двома найбільш підходящими варіантами типу додатку є веб-додаток та десктопний додаток. Обидва типи мають ряд переваг та недоліків. Веб-додатки призначені для роботи в браузері, тобто не потребують встановлення на комп’ютері користувача. Іншою перевагою веб-додатків є те, що вони виконують основні обчислення на стороні серверу, що може значно зменшити навантаження на пристрій користувача. Проте, в такого підходу також є ряд

недоліків. Найбільш очевидним є те, що такі додатки потребують постійного підключення до мережі інтернет. І хоча в сучасному світі майже всі мають підключення до мережі, цим недоліком все ж не варто нехтувати, оскільки не всі користувачі мають бажання розкривати свої данні в мережі якщо на те немає особливої потреби. Іншим недоліком веб-додатку є більша вартість підтримки. В залежності від розміру додатку вартість його підтримки може бути значно більшою ніж в аналогічного десктопного додатку. Особливо актуальним є цей недолік для додатків що не потребують використання таких технологій як бази даних або хмарні обчислення, оскільки вони не мають елементів які потребували б хостингу, і тому десктопний варіант не потребував би додаткових витрат на підтримку роботоздатності додатку.

Десктопні додатки виконуються на пристрої користувача і не потребують підключення до мережі інтернет, за винятком додатків які взаємодіють з віддаленими ресурсами, проте такі додатки майже аналогічні веб-додаткам. До переваг такого типу додатків можна віднести їх незалежність від мережі інтернет, та у деяких випадках меншу вартість підтримки. Проте, вони також мають ряд недоліків. По перше, на відміну від веб додатків десктопні додатки повністю розміщуються на пристрої користувача, а тому вони потребують значно більше ресурсів, ніж додатки які виконують основну масу обчислень на віддаленому сервері. Другий недолік випливає з першого – десктопні додатки більш важкі в поширенні, оскільки вони потребують повного завантаження, що залежно від розміру додатку та швидкості підключення до мережі може зайняти значний відрізок часу. Окрім того, деякі додатки потребують встановлення, що також займає додатковий час.

Отже, існує два варіанти реалізації системи з відносно рівнозначними перевагами та недоліками. Проте, в нашому випадку доцільно було б обрати реалізацію десктопного додатку. Система не потребуватиме зовнішніх ресурсів, таких як бази даних, і не виконуватиме занадто складних обчислень, які б потребували значних ресурсів від пристрою користувача. Тому, обраним форматом реалізації системи є десктопний додаток.

Таким чином, було сформовано вимоги до системи. Це повинен бути десктопний додаток, який виконує функції експертної системи по визначенню пріорітетів оптимзіації програмного продукту та підтримує наступні функції:

– забезпечує введення користувачем вхідної інформації;
– виконує валідацію введених даних на предмет невідповідності;
– виконує обробку даних відповідно до розробленого алгоритму визначення пріорітету;
– забезпечує виведення даних у графічному форматі.

3.2 Вибір та обґрунтування архітектурних рішень

Як вже було описано, обраним форматом розроблюваної програмної системи є десктопний додаток. В якості засобів розробки було обрано мову програмування C# та технологію WPF для побудови графічного інтерфейсу користувача додатку.

WPF – це система розробки графічних клієнтських додатків, яка може використовувати будь-яку .NET сумісну мову, включаючи і C#. WPF має ряд переваг порівняно з іншими аналогами. В першу чергу це використання одиниць незалежних від розширення екрану для визначення розмірів елементів графічного інтерфейсу. Це дозволяє розробляти графічні інтерфейси, які будуть однаково відображатись на системах з різними розширеннями екрану. Окрім того, WPF дозволяє використовувати апаратне прискорення для рендерингу графічних елементів, що зменшує навантаження на процесор.

Наступним кроком планування розробки програмної системи є вибір типу архітектури. При розробці архітектури нашого додатку доцільно буде скористатись архітектурним паттерном Model-View-ViewModel (MVVM). Основне призначення цього паттерну полягає в розділенні логіки додатку від візуальної частини. Основною цілевою платформою цього паттерну власне

була WPF, проте на сьогоднішній день він також широко використовується і для розробки на інших платформах, таких як Android та iOS.

Як можна здогадатись з назви, паттерн MVVM складається з трьох компонентів – моделі, представлення та моделі представлення. Модель призначена для опису логіки програми. Зазвичай моделі описують певні об’єкти предметної області та містять логіку по обробці та валідації даних. Представлення в паттерні MVVM – це компонент системи, який призначений для відображення через візуальний інтерфейс. У випадку з WPF, ці компоненти зазвичай представлені у вигляді xml файлів, які визначають розмітку певного елементу інтерфейсу. Зазвичай, згідно паттерну MVVM, компоненти представлення не мають містити ніякої логіки окрім ініціалізації елементу інтерфейсу. Уся логіка, пов’язана з відображенням виноситься в компонент моделі представлення. Компонент ViewModel виконує функцію зв’язування моделі та представлення. Модель представлення реалізує логіку отримання та форматування даних моделі для відображення у графічному форматі, а також логіку збереження змін моделі, які були виконані за допомогою графічного інтерфейсу. У WPF модель представлення виконує передаванні інформації у компонент представлення за допомогою функції прив’язки даних, а представлення взаємодіє з моделлю представлення за допомогою команд, оскільки інтерактивні елементи, як наприклад кнопки не використовують події для сповіщення про зміни.

Таким чином, основну ідею архітектури паттерну MVVM можна зобразити за допомогою схеми, зображеної на рисунку 3.1.

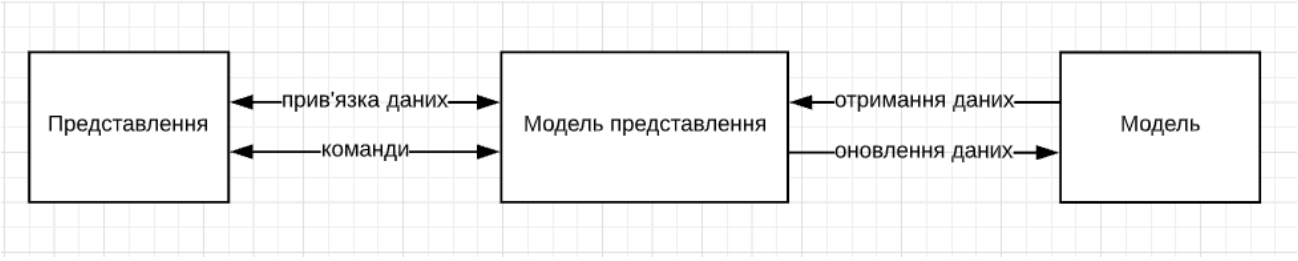


Рисунок 3.1 – Схема архітектури паттерну MVVM

Отже, для розробки програмної системи було обрано мову програмування C# та технологію WPF. Побудова архітектури відбуватиметься з використанням паттерну Model-View-ViewModel для підтримання розділення логіки додатку від візуальної частини додатку.

3.3 Визначення функцій системи

В розділі постановки технічного завдання було визначено наступні функції системи:

- забезпечення введення користувачем вхідної інформації;
- виконання валідації введених даних на предмет невідповідності;
- виконання обробки даних відповідно до розробленого алгоритму визначення пріорітету;
- забезпечення виведення даних у графічному форматі.

Відповідно, виходячи з цих функцій, можна наступні напрями роботи системи. Основним застосування розроблюваної системи є визначення пріорітетів оптимізації програмного проекту. Для забезпечення визначення показників пріорітетів необхідно визначити різноманітні фактори параметрів проекту, відповідно до розробленого алгоритму роботи експертної системи.

Першим кроком цього алгоритму є вибір типу проекту. Вибір здійснюється з чотирьох можливих варіантів – десктопний додаток, веб-додаток, система реального часу та вбудований додаток. Вибір, здійснений на цьому етапі визначить основу системи пріорітетів, яка буде уточнюватись на всіх подальших етапах.

Наступним етапом є вибір платформи. Вибір типу платформи здійснюватиметься з таких варіантів як ПК, хмарне середовище та смарт пристрій. Для таких варіантів типу додатку як десктопний додаток, веб-додаток та вбудованих додаток цей вибір буде автоматичним, це буле ПК, хмарне середовище та смарт пристрій відповідно. У випадку з типом додатку

система реального часу користувач повинен вибрати цей параметр в ручному режимі вибору.

Подальшим етапом є визначення параметрів середовища. На цьому етапі користувач повинен ввести такі характеристики середовища роботи додатку як кількість потоків процесора, об’єм оперативної та статичної пам’яті.

Далі користувач визначає очікувану кількість користувачів системи, а також додаткові параметри розроблюваної системи. Останнім етапом роботи є отримання результатів аналізу введених даних. Результат подаватиметься у графічному вигляді, в вигляді діаграми представленої на рисунку 1.

Таким чином, було визначено основні варіанти використання. Графічно цю структуру можна зобразити за допомогою UML діаграми Use Case, діаграми варіантів використання. Діаграма варіантів використання зображена на рисунку 3.2.

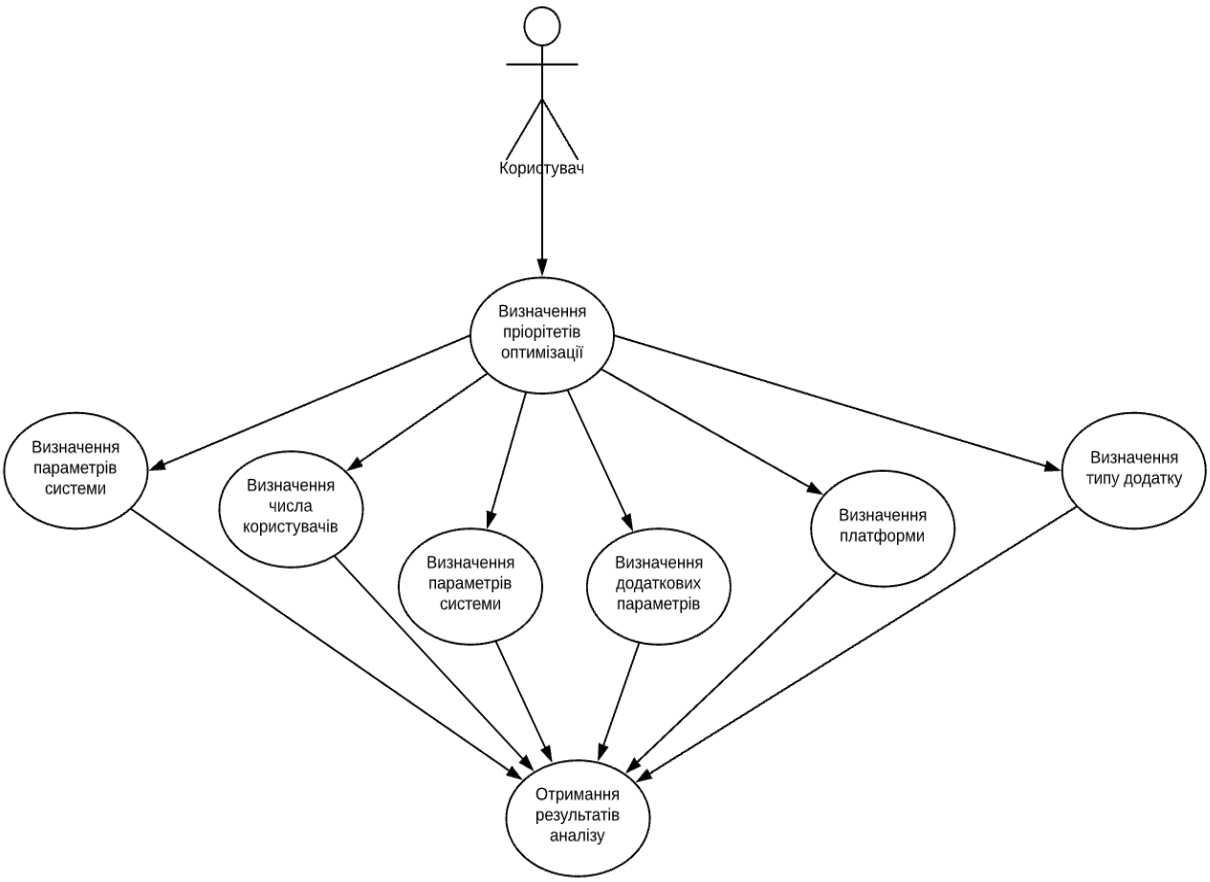


Рисунок 3.2 – Діаграма варіантів використання

Визначені на цьому етапі варіанти використання системи допоможуть в подальшій розробці функційних вимог та модульної структури системи.

Наступним етапом розробки є визначення послідовності дій при роботі з системою. Наведені вище варіанти використання повинні бути виконані послідовно, оскільки попередні етапи можуть містити інформацію, яка впливає на подальші етапи. Першим кроком є визначення типу розроблюваної системи. Тип системи визначатиме можливі варіанти на наступному кроці – кроці вибору платформи розробки. Так, усі три варіанти типу платформи будуть доступні для вибору тільки для типу системи система реального часу. Для решти типів системи тип платформи буде визначений автоматично.

Наступним кроком користувача буде визначення параметрів системи. Параметрами системи, які будуть визначатися користувачем будуть кількість потоків, підтримуваних процесором, об’єм оперативної пам’яті та об’єм статичної пам’яті.

Далі, система потребуватиме введення очікуваного числа користувачів та додаткові параметри розроблюваної системи, такі як специфічні типи даних, з якими взаємодіє додаток.

Останнім етапом роботи з системою є отримання результатів обробки та аналізу введених даних системою. Експертна система повинна оцінити введені користувачем параметри системи та забезпечити вивід результату аналізу даних у вигляді діаграми.

Для графічного відображення послідовності дій роботи з системою доцільно використати UML діаграму послідовності, яка представлена на рисунку 3.3. Ця діаграма знадобиться як на етапі первинної розробки, так і на етапі тестування, оскільки вона описує основну послідовність дій при роботі з додатком, що стане основою більшості тест-кейсів для тестування.

Наступною областю планування є планування станів системи. Початковий стан системи це стан, в якому знаходитиметься система до того як користувач здійснив введення будь-яких параметрів розроблюваної системи.

В цьому стані експертна система приймає лише один параметр – тип розроблюваної системи.

Рисунок 3.3 – Діаграма послідовності

Наступною областю планування є планування станів системи. Початковий стан системи це стан, в якому знаходитиметься система до того як користувач здійснив введення будь-яких параметрів розроблюваної системи. В цьому стані експертна система приймає лише один параметр – тип розроблюваної системи.

Після того як користувач обрав тип проекту, система переходить в стан очікування вибору типу платформи, після якого відбувається перехід у стан введення параметрів середовища. Якщо користувач обрав десктопний, веб або вбудований додаток, то система виконає цей стан самостійно, оскільки вибір типу платформу буде здійснено автоматично, відповідно до типу системи.

В стані введення параметрів середовища система очікуватиме від користувача введення таких параметрів середовища виконання проекту як кількість потоків процесора, об’єм оперативної пам’яті та об’єм статичної пам’яті. Після введення цих параметрів система переходить до наступного стану, стану введення очікуваної кількості користувачів. Після того як користувач завершує цей етап існує два можливих варіанта подальшої роботи. Користувач може також ввести додаткові параметри системи, які визначатимуть додаткові характеристики розроблюваної системи, наприклад специфічні типи даних. Проте цей етап не є обов’язковим, і тому не обов’язково повинен бути виконаний.

В будь-якому випадку, після виконання всіх етапів введення параметрів розроблюваного програмного проекту система переходить в заключний етап, а саме в стан отримання результатів аналізу у вигляді діаграми пріорітетів оптимізаційних процесів.

Графічно ця структура станів системи представлена на рисунку 3.4 у вигляді UML діаграми станів (Statechart).

Наступним етапом проектування є визначення активностей системи. Активності загалом схожі на стани системи, проте активності є певними діями всередині системи, а переходи між ними відбуваються після завершення дій.

Рисунок 3.4 – Діаграма станів

Після вихідного стану системи, першочерговою активністю яка повинна виконатись є вибір типу проекту. Вибір типу здійснюється з чотирьох можливих варіантів, після чого можливі два варіанти подальшої роботи системи. Якщо користувач обрав систему реального часу у якості типу проекту, то наступним кроком є вибір типу платформи. В іншому випадку цей етап здійснюється автоматично, відповідно до обраного типу проекту.

Далі, йде крок введення параметрів середовища. Ця активність передбачає від користувача введення даних про такі параметри середовища, в якому працюватиме розроблюваний проект як кількість потоків процесора, об’єм оперативної та статичної пам’яті. По заверешнню введення цих даних система переходить до наступної активності – введення даних про очікувану кількість користувачів. Після цієї активності слідує розгалуження. Користувач може ввести додаткові параметри системи, які характеризують специфічність даних, з якими працює система. Проте, цей крок не є обов’язковим, і можливий варіант коли після активності введення кількості користувачів слідуватиме фінальна активність.

Останньою активністю є виведення результатів аналізу даних введених користувачем. Ця активність передбачає обробку введених даних та формування результату у вигляді діаграми пріорітетів оптимізаційних процесів. Ця активність є заключною і представляє собою підсумок всього процесу роботи системи.

Активності системи можна зобразити графічно за допомогою UML діаграми активностей, яка представлена на рисунку 3.5.

Діаграми станів та активностей стануть у нагоді при побудові робочого потоку програмної системи при початковому проектуванні та розробці системи. Окрім того, ці діаграми також можуть бути використані на етапі тестування, для забезпечення контролю за коректною послідовністю станів та дій системи.

Усі ці діаграми, що були розроблені для розроблюваної системи є послідовними етапами проектування розробки поведінки програмної системи.

Ці діаграми описують можливі варіанти застосування та їх послідовність, таким чином будуючи схему робочого процесу програмної системи. Під час розробки програмного коду ці знання значно полегшать процес визначення та задання поведінки програмної системи.

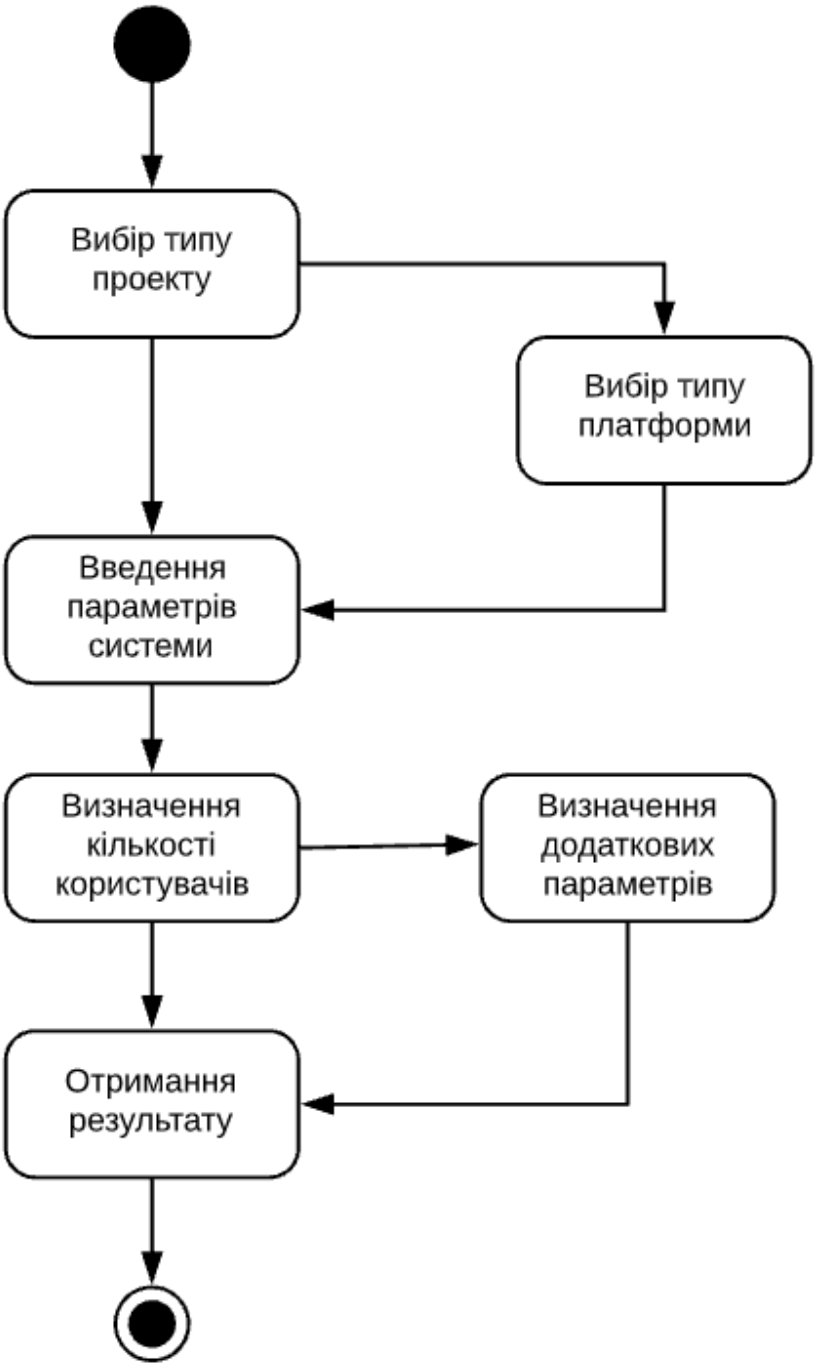


Рисунок 3.5 – Діаграма активностей

Таким чином, було побудовано архітектурні основи розроблюваної програмної системи. Було визначено засоби розробки та побудови графічного інтерфейсу користувача. Окрім того, було обрано архітектурні шаблони, які будуть застосовуватись для побудови системи.

Також, за допомогою UML діаграм було спроектовано такі аспекти системи як варіанти використання, послідовність роботи, структуру станів та активностей системи. В подальшому ці діаграми будуть використані для побудови поведінки системи.

3.4 Проектування інтерфейсу користувача

Інтерфейс користувача – це набір засобів за допомогою яких користувач взаємодіє з системою. Інтерфейси користувача характеризуються двома основними компонентами – засобами введення та засобами виведення інформації системи.

На сьогоднішній день двома найбільш поширеними типами користувацьких інтерфейсів є текстові та графічні. Текстові інтерфейси передбачають взаємодію з користувачем шляхом введення текстових команд та виведення текстової інформації. Проте, на сьогоднішній день такі інтерфейси поступово витісняються інтерфейсами, які використовують графічні засоби для взаємодії з користувачем. Тому, для розробки програмної реалізації експертної системи доцільно буде обрати графічну реалізації інтерфейсу користувача.

Важливим етапом розробки інтерфейсу є побудова макету графічного інтерфейсу. Макетування надає ряд переваг при розробці графічних інтерфейсів користувача. Основною перевагою такого підходу є відносно легке створення зразка представлення того, як має виглядати завершений продукт. Маючи видимий орієнтир, розробнику буде значно легше виконувати безпосередню розробку графічного інтерфейсу. Окрім того, під час побудови

макету можуть більш детально прояснитись певні моменти, які відносяться до функціоналу додатку. Саме тому макетування та прототипування є популярною методикою під час розробки проектів різних масштабів.

Для розробки макету графічного інтерфейсу додатку було обрано онлайн-сервіс Figma. Figma дозволяє будувати макети графічних елементів різної складності, і тому цей сервіс набув широкого застосування для розробки макетів графічних інтерфейсів додатків.

Розроблений макет представлено на рисунку 3.6.

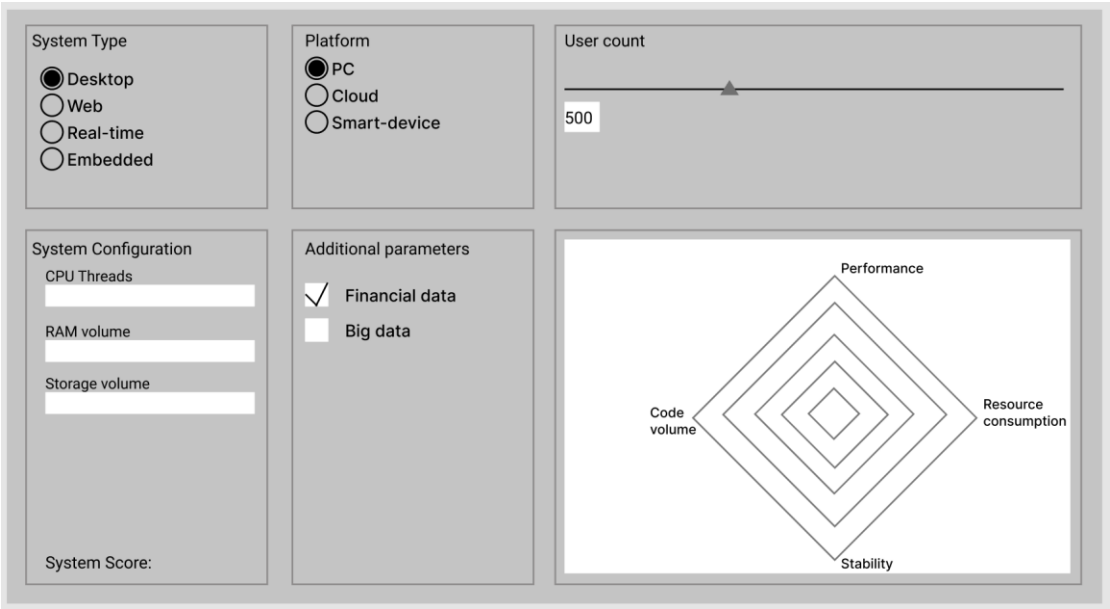


Рисунок 3.6 – Макет графічного інтерфейсу

Графічний інтерфейс повинен відповідати робочому потоку експертної системи. Алгоритм роботи експертної системи складається з послідовних етапів, тому це варто врахувати при побудові інтерфейсу. Доцільно буде побудувати інтерфейс у вигляді блоків, кожен з яких відповідатиме одному етапу алгоритму експертної системи. Окрім того, забезпечення виведення результату роботи системи варто також реалізувати окремим блоком. Таким чином, керування системою буде розділене на окремі, логічно виділені етапи, і в той же час користувач зможе побачити всі аспекти системи в цілому та результат аналізу введених даних в той же час.

Першим кроком роботи системи є визначення типу проекту. Блок визначення типу проекту доцільно буде реалізувати за допомогою перемикачів Radio Button. Перемикачі такого типу дозволяють обрати тільки один з декількох варіантів. Це цілком відповідає поставленій цілі, оскільки на цьому етапі можна обрати тільки один варіант.

Наступний крок, крок вибору типу платформи буде реалізовано подібним чином. Блок вибору типу платформи міститиме три можливих варіанти вибору, представлені перемикачами. Проте, ці перемикачі повинні бути неактивними до моменту коли користувач обирає тип проекту. Після вибору типу проекту, якщо було обрано десктопний додаток, веб додаток або вбудовану систему, на етапі вибору платформи буде доступний тільки один можливий варіант платформи, а тому доцільно буде автоматично обрати його. Якщо ж було обрано систему реального часу як тип проекту, то для вибору доступні будуть усі три варіанти типу платформи, і користувач повинен буде обрати один з них.

Наступним кроком є визначення числа користувачів. Введення цього параметру доцільно реалізувати двома шляхами. Першим є введення за допомогою шкали. Користувач зможе обрати потрібне число переміщуючи повзунок на шкалі. Це буде досить наглядним способом визначення та відображення числового значення. Проте, такий метод може бути недостатньо зручним для точного введення даних. Тому варто також реалізувати введення цього значення за допомогою текстового поля, в якому можна буде вводити лише цілі числа. Таким чином, текстове поле відіграватиме роль основної області введення, а шкала з повзунком реалізуватимуть більш наглядну форму відображення даних. Проте, за бажання, користувач також зможе використовувати повзунок для встановлення значення. Це надасть блоку введення більшу інтерактивність, що сприятиме більш позитивному досвіду користувача. Також, для того щоб відображення шкали було більш зрозумілим, варто зробити блок вибору кількості користувачів довшим за інші блоки введення.

Наступним блоком є блок вибору характеристик середовища. До характеристик які користувач повинен ввести відносяться кількість потоків процесора, об’єм оперативної пам’яті та об’єм статичної пам’яті. Для введення цих даних варто застосувати текстові поля, подібні до того яке було використане в попередньому блоці. Також, на цьому блоці варто відображати інформацію про рівень потужності системи згідно з класифікацією експертної системи. Це надаватимеме користувачу більше інформації про те як здійснюється аналіз.

Останнім кроком є вибір додаткових параметрів. Цей блок може бути реалізований схожим чином до перших двох блоків. Проте, на цьому етапі користувач може обрати декілька параметрів, або не обрати жодного. Тому для реалізації цього блоку доцільно буде використати перемикачі типу Checkbox замість Radio Button. На відміну від Radio Button Checkbox дозволяє вибрати декілька елементів, а також зняти вибір з уже обраних елементів. Це відповідає коректній поведінці для цього етапу.

Останнім блоком є блок відображення результатів. Відображення результату доцільно здійснювати у вигляді діаграми, оскільки це сприяє більш комфортному сприйняттю інформації і збільшує зрозумілість результатів аналізу даних, введених користувачем. Цей блок доцільно буде зробити більшим за інші, оскільки він міститиме ключову інформацію, а тому повинен звертати на себе увагу.

Окрім визначення зовнішнього вигляду додатку на етапі проектування графічного інтерфейсу варто також продумати алгоритм поведінки інтерфейсу при взаємодії користувача з ним. Важливим аспектом будь-якого інтерфейсу користувача є досвід користувача при роботі з ним. Для покращення цього досвіду доцільно збільшувати динамічність та відповідність роботи інтерфейсу користувача.

Так, для збільшення динамічності інтерфейсу варто забезпечити динамічну зміну інтерфейсу при зміні даних введених користувачем. Основною точкою цих змін буде блок відображення результатів, оскільки він

містить в собі інформацію, яка залежить від всіх даних, введених користувачем додатку.

На першому кроці роботи алгоритму експертної системи ми отримуємо базову схему пріорітетів оптимізаційних процесів, а тому відображення результатів можна починати вже не цьому етапі. Всі подальші етапи здійснюватимуть уточнення початкової схеми пріорітетів, і тому після кожної зміни даних введених користувачем доцільно змінювати вигляд результатів аналізу. Окрім того, динамічність інтерфейсу буде досягатись завдяки пов’язаним елементам. Один з таких прикладів вже був наведений на етапі опису вигляду інтерфейсу – коли дані з першого блоку впливають на доступність та значення елементів другого блоку. Іншим прикладом такої взаємодії можна вважати взаємодію елементів блоку вибору кількості користувачів. Зміни значення за допомогою шкали та повзунка автоматично будуть призводити до зміни значення текстового поля. І навпаки, зміна значення в текстовому полі буде призводити до зміни позиції повзунка на шкалі. Такої взаємодії досить легко досягти засобами WPF, шляхом прив’язки значень обох елементів до одного значення об’єкту даних або моделі представлення, згідно паттерну MVVM.

Таким чином, було розроблено макет графічного інтерфейсу користувача додатку. Розробка макету не тільки надає зразок, за яким будуватиметься реальний інтерфейс засобами WPF, але і дозволила продумати певні моменти реалізації до того як почалась їх програмна реалізація.

3.5 Розробка структури класів додатку

Розробка додатку буде відбуватись за технологією об’єктно-орієнтованого програмування. Це означає, що система буде представлена у вигляді множини об’єктів різних класів. Для успішної розробки необхідно чітко визначити необхідні класи, їх властивості методи, та їх призначення

Основним класом у системі буде клас, який представлятиме характеристики проекту. Об’єкт цього класу формуватиметься з даних, які вводитиме користувач за допомогою інтерфейсу. Цей клас буде характеризуватися такими властивостями як:

– тип проекту;
– тип платформи;
– кількість потоків процесора;
– об’єм оперативної пам’яті;
– об’єм статичної пам’яті;
– кількість користувачів;
– список додаткових параметрів.

Як можна побачити з списку властивостей класу, він потребує створення додаткових структур даних. Так, властивості типу проекту, типу платформи та додаткових параметрів неможливо реалізувати за допомогою набору стандартних типів мови програмування C#. Тому, доцільно буде створити додаткові структури даних, і включити їх в основний клас шляхом агрегації.

Для реалізації цих структур даних доцільно буде скористатись таким засобом мови програмування C# як тип перерахування (enum). Створення перерахування дозволить реалізувати набір констант, пов’язаних за областю застосування, що цілком відповідає поставленій задачі.

Отже, для реалізації цих специфічних властивостей необхідно створити набори константних значень, які відповідатимуть різним значенням цих параметрів проекту. Перерахування типу системи може набувати таких значень як:

– десктопний додаток;
– веб-додаток;
– система реального часу;
– вбудована система.

Параметр типу платформи відповідатиме одному з наступних значень:

– ПК;

– хмарне середовище;

– смарт пристрій.

Властивість додаткових параметрів варто реалізувати у якості списку типу перерахування. Це дозволить зберігати декілька значень одночасно, оскільки користувач може обрати один або декілька з додаткових параметрів, або не обрати жодного. Тому, використання списку буде оптимальним рішенням для цього параметру. Перерахування додаткових параметрів зможе набувати одного з таких значень як:

– робота з банківськими даними;

– робота з великими даними.

Отже, цей клас буде реалізацією набору даних, які вводить користувач. Цей клас також повинен містити метод, який виконуватиме аналіз цих даних з метою визначення пріорітетів оптимізаційних процесів. Наступним логічним кроком буде визначення класу, об’єкт якого відповідатиме результату аналізу цих даних. Метод аналізу даних повертатиме об’єкт цього типу.

Клас даних результату аналізу міститиме чотири властивості, які відповідатимуть чотирьом напрямам оптимізації:

– швидкодія;

– стабільність;

– об’єм коду;

– використання ресурсів.

Ці властивості набуватимуть значень цілих невід’ємних чисел. Окрім того, цей клас повинен містити додаткову валідацію, яка забезпечуватиме, що сума значень цих властивостей є меншою ніж 9.

Оскільки розробка ведеться за паттерном програмування View-Model-ViewModel, окрім класів, які реалізуватимуть структури даних необхідно також реалізувати моделі відображення, які забезпечуватимуть прив’язку даних та логіку відображення цих даних. Оскільки додаток оперуватиме одним відображенням для введення даних та виведення результатів аналізу введених даних, модель відображення буде також одна.

Вона міститиме наступні параметри:

– тип проекту;
– тип платформи ПК доступний;
– тип платформи хмарне середовище доступний;
– тип платформи смарт пристрій доступний;
– тип платформи;
– кількість користувачів;
– кількість потоків процесора;
– об’єм оперативної пам’яті;
– об’єм статичної пам’яті;
– обрано додатковий параметр робота з банківськими даними;
– обрано додатковий параметр робота з великими даними;
– структура пріорітетів оптимізації.

Модель представлення також містить методи-сповіщувачі на зміни цих властивостей, тому при зміні параметрів, які вводяться користувачем відбуватиметься і зміна об’єктів даних та проведення аналізу даних з метою одержання структури пріорітетів.

Основне призначення цієї моделі представлення полягає в забезпеченні збереження даних введених користувачем для відображення на інтерфейсі та реалізація реагування системи на зміни в даних, які будуть здійснюватися користувачем за допомогою вищезгаданих методів-сповідщувачів. Модель представлення також міститиме в собі модель даних. За допомогою цієї моделі здійснюватиметься більшість розрахунків, які відносяться до обчислення пріорітетів оптимізації досліджуваної програмної системи. Модель представлення буде звертатись до моделі параметрів системи з метою переобчислення пріорітетів оптимізації кожен раз коли користувач здійснюватиме зміни в введених даних.

Більш детально структура класів описана на рисунку 3.7, за допомогою UML діаграми класів.

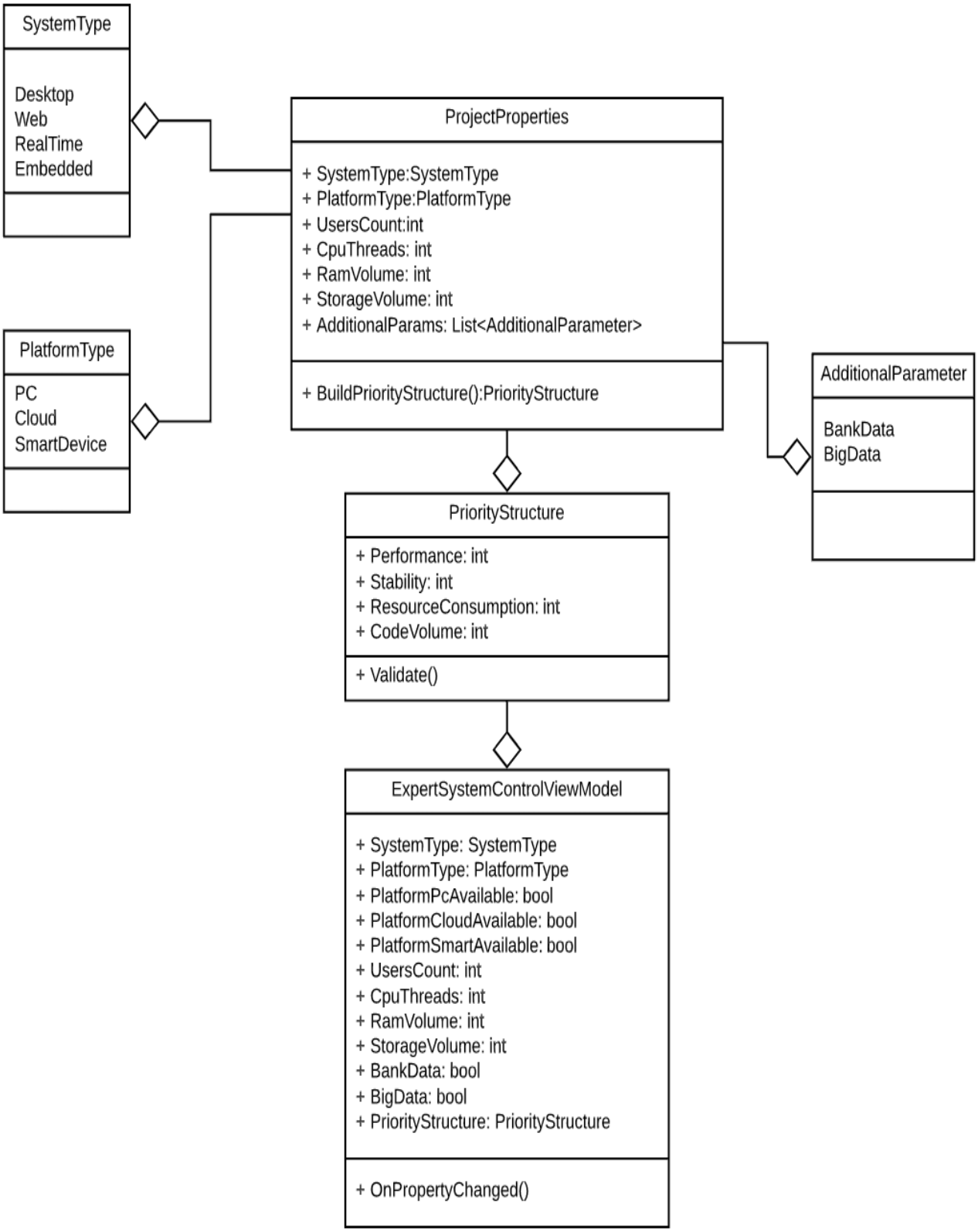


Рисунок 3.7 – Діаграма класів

Отже, було розроблено структуру класів, яка буде застосована для побудови додатку. Структура класів передбачає розділення класів, які

використовуватимуться для роботи з даними та класів, які застосовуватимуться для реалізації логіки відображення, відповідно до паттерну View-Model-ViewModel. Розроблена структура класів дозволить реалізувати роботу експертної системи та забезпечить коректну роботу інтерфейсу користувача.

3.6 Програмна реалізація розробленої системи

Програмна система складається з набору компонентів, і представлятиме собою додаток, який буде складатися з одного вікна, яке буде забезпечувати здійснення контролю експертної системи та реалізуватиме функції виведення результату аналізу даних експертною системою. Інтерфейс буде реалізовано за допомогою таких компонентів WPF як UserControl, які будуть отримувати дані з об’єктів типу ViewModel та здійснювати двусторонню прив’язку до них. ViewModel будуть здійснювати звернення до об’єкту який буде реалізовувати алгоритм експертної системи, передавати їй введені користувачем дані та отримувати результати аналізу експертної системи на кожному кроці.

Першим кроком розробки програмної системи буде створення базового класу, який реалізуватиме інтерфейс INotifyPropertyChanged. Цей клас наслідуватимуть всі класи ViewModel. Реалізація цього інтерфейсу забезпечуватиме сповіщення системи про зміну властивостей моделі представлення, а відповідно і оновлення даних на інтерфейсі користувача. Забезпечення цих властивостей дозволить нам здійснювати обчислення, а оновлення відображення цих даних на інтерфейсі здійснюватиметься в автоматичному режимі.

Реалізація інтерфейсу INotifyPropertyChanged:

```
public abstract class ViewModel : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;
```

```
     public void OnPropertyChanged([CallerMemberName] string
prop = "")
     {
          if (PropertyChanged != null)
          PropertyChanged(this, new
     PropertyChangedEventArgs(prop));
     }
}
```

Цей клас було реалізовано як абстрактний клас, оскільки він призначений лише для наслідування його іншими класами.

Наступний крок розробки програмної системи є побудова класів моделей. На відміну від моделей представлення, вони не повинні реалізувати інтерфейс INotifyPropertyChanged, а тому їх можна реалізувати як звичайні класи. Для роботи системи необхідними будуть моделі двох типів – ProjectProperties, яка представлятиме дані про проект, введені користувачем та PrioritySystem, яка відображатиме результат аналізу даних експертною системою. Ці моделі міститимуть набори властивостей, які будуть містити дані, необхідні для роботи системи. Окрім того, модель ProjectProperties буде містити метод, який буде виконувати обчислення згідно алгоритму роботи експертної системи. Цей метод має виконуватись на кожному етапі заповнення користувачем даних про проект, а також при зміні вже введених даних. Це дозволить відображати, як введені параметри впливають на систему пріорітетів, а також підтримувати актуальність результатів роботи відповідно до введених або змінених даних.

Перший етап роботи цього алгоритму полягає в визначенні типу проекту. Згідно цього типу обиратиметься один з наперед визначених систем пріорітетів оптимізаційних процесів, які будуть основою для всіх подальших обчислень пріорітетів оптимізації. Також, оскільки цей етап є першим на ньому буде і ініціалізуватись значення змінної, яка відповідатиме за збереження даних про пріорітети оптимізації та буде повертатись як результат роботи цього методу.

Фрагмент коду програми з виконанням ініціалізації змінної пріорітетів оптимізації:

```
PrioritySystem prioritySystem = new PrioritySystem(0, 0, 0, 0);

if (ProjectType.HasValue)
{
     switch (ProjectType.Value)
     {
          case SystemType.Desktop:
          prioritySystem = new PrioritySystem(4, 3, 1, 0);
          break;

          case SystemType.Web:
          prioritySystem = new PrioritySystem(2, 4, 2, 0);
          break;

          case SystemType.RealTime:
          prioritySystem = new PrioritySystem(2, 4, 2, 0);
          break;

          case SystemType.Embedded:
          prioritySystem = new PrioritySystem(1, 2, 2, 3);
          break;
     }
}
else
{
     return prioritySystem;
}
```

Далі виконується введення типу платформи. Тип платформи визначає певні корективи в вже визначену систему пріорітетів шляхом збільшення одного з пріорітетів та зменшення іншого. Фрагмент коду, який реалізує виконання цих операцій:

```
if (ProjectPlatformType.HasValue)
{
     switch (ProjectPlatformType.Value)
     {
          case PlatformType.Desktop:
          prioritySystem.Performance++;
          if (prioritySystem.CodeVolume < 1)
          {
               prioritySystem.ResourceConsumption--;
          }
```

```
            else
            {
                 prioritySystem.CodeVolume--;
            }

            break;

            case PlatformType.Cloud:
                 prioritySystem.Stability++;
                 prioritySystem.ResourceConsumption--;

                 break;

            case PlatformType.Smart:
                 prioritySystem.CodeVolume++;
                 prioritySystem.ResourceConsumption++;
                 prioritySystem.Performance--;
                 if          (prioritySystem.Performance          +
           prioritySystem.Stability + prioritySystem.CodeVolume +
prioritySystem.ResourceConsumption > 8)
                 {
                      prioritySystem.Stability--;
                 }

                 break;
      }
}

else
{
      return prioritySystem;
}
```

Варто також звернути увагу на перевірку, яка перевіряє чи параметр типу платформи має значення. Ця перевірка буде застосована для кожного параметру проекту, який був введений користувачем з метою забезпечення можливості застосування одного і того ж методу для обчислення пріорітетів на всіх етапах введення даних користувачем. Завдяки таким перевіркам обчислення буде вестись поетапно до тих пір поки будуть наявні дані введені користувачем, після чого метод поверне результат аналізу.

На наступному етапі роботи алгоритму відбувається аналіз очікуваної кількості користувачів. Розбиття типів кількості користувачів на три етапи реалізовано одним логічним розгалуженням, оскільки перший і третій

інтервали мають однаковий вплив на показники пріорітетів оптимізації. Фрагмент коду з виконанням аналізу кількості користувачів:

```
if (UserCount.HasValue)
{
     if (UserCount.Value < 1000 || UserCount.Value > 1000000)
     {
          prioritySystem.Performance--;
          prioritySystem.Stability++;
     }

     else
     {
          prioritySystem.Performance++;
          prioritySystem.Stability--;
     }
}

else
{
     return prioritySystem;
}
```

Далі йде аналіз параметрів середовища. Цей етап характерний тим, що на ньому можлива різка зміна пріорітету оптимізації використання ресурсів. Такі зміни майже завжди призводитимуть до потреби в перебалансуванні системи пріорітетів, оскільки це призведе до виходу суми пріорітетів оптимізації за відведені рамки. Тому на цьому етапі варто також реалізувати механізм, який зменшуватиме значення іншого пріорітету так, щоб сума всіх пріорітетів відповідала визначеному стандарту.

Фрагмент коду програми, який реалізує аналіз параметрів середовища:

```
if (SystemScore.HasValue)
{
     switch (SystemScore.Value)
     {
          case Enums.SystemScore.SemiLow:
          prioritySystem.ResourceConsumption++;
          break;

          case Enums.SystemScore.Low:
          prioritySystem.ResourceConsumption += 2;
          break;
```

```
     }

     var     unbalance     =     prioritySystem.Performance     +
prioritySystem.ResourceConsumption +
prioritySystem.Stability + prioritySystem.CodeVolume - 8;

     if (unbalance > 0)
     {
          if (prioritySystem.CodeVolume > unbalance)
          {
               prioritySystem.CodeVolume -= unbalance;
          }
          else if (prioritySystem.Performance > unbalance)
          {
               prioritySystem.Performance -= unbalance;
          }
          else if (prioritySystem.Stability > unbalance)
          {
               prioritySystem.Stability -= unbalance;
          }
     }
}

else
{
     return prioritySystem;
}
```

Завершальним етапом роботи алгоритму є аналіз додаткових параметрів. На цьому етапі система перевіряє чи користувачем було обрано якісь з додаткових параметрів та виконує відповідні обчислення, залежно від типу додаткового параметру.

Фрагмент коду, який реалізує цей етап:

```
if (AdditionalParameters.Any())
{

     if
(AdditionalParameters.Contains(AdditionalParameter.BankData))
     {
          prioritySystem.Stability++;
          prioritySystem.ResourceConsumption--;
     }

     if
(AdditionalParameters.Contains(AdditionalParameter.BigData))
```

```
     {
          prioritySystem.Performance++;
          if (prioritySystem.CodeVolume > 0)
          {
               prioritySystem.CodeVolume--;
          }
          else if (prioritySystem.ResourceConsumption > 0)
          {
               prioritySystem.ResourceConsumption--;
          }
          else if (prioritySystem.Stability > 0)
          {
               prioritySystem.Stability--;
          }
     }

}

return prioritySystem;
```

Після розробки моделей, які реалізуватимуть дані системи та роботу з ними необхідно також розробити frontend-логіку програмної системи. Цей етап характеризуватиметься розробкою представлення та моделі представлення додатку.

Представлення буде реалізовано як елемент UserControl. Розмітка сторінки за допомогою мови розмітки XAML буде вестись згідно розробленого макету. Також, для реалізації логіки опрацювання введених даних необхідно розробити клас моделі представлення, який буде наслідуватись від абстрактного класу, розробленого раніше.

Для кожного елементу керування, який буде використовуватись користувачем для введення даних потрібно реалізувати прив’язку даних для забезпечення зв’язку з моделлю представлення. Реалізація прив’язки виконується наступним чином:

```
     <RadioButton FontSize="18"
     GroupName="SystemType"  IsChecked="{Binding  Path=IsDesktop,
Mode=TwoWay}">
     Desktop
     </RadioButton>
```

Тут також варто звернути увагу на тип прив’язки. Для елементів керування, які будуть використовуватись для введення інформації необхідно реалізувати двосторонню прив’язку даних. В WPF цей зв’язок досягається за допомогою встановлення параметру Mode=TwoWay при здійсненні прив’язки. Окрім елементів керування інтерфейс також буде містити елементи відображення даних. До таких належать елементи, які відображатимуть кінцевий результат та проміжні результати, як наприклад тип потужності системи на етапі вибору параметрів середовища. У цьому випадку достатньо буде односторонньої прив’язки, яка є значенням параметру Mode за замовчуванням, а тому ніяких додаткових параметрів у таких випадках вказувати не потрібно.

Останнім кроком роботи програми є формування діаграми пріорітетів оптимізації. Оскільки ця діаграма завжди буде мати наперед визначену кількість категорій та кроків, її можна реалізувати вбудованими засобами WPF. Основою діаграми може бути елемент Grid з однією коміркою. Це дозволить накладати елементи один на інший і побудувати область даних діаграми. Для створення ліній, які відображатимуть рівні даних на діаграмі варто застосувати елемент Path. Цей елемент дозволяє будувати фігури різних видів, в нашому випадку – ромб. Для реалізації області даних діаграми нам потрібні будуть п’ять таких елементів, кожен з яких буде меншим за попередній на сталу величину.

Фрагмент коду з реалізацією області даних діаграми:

```
<Path Width="450"
Height="450"
HorizontalAlignment="Center"
VerticalAlignment="Center"
Data="M50,0L100,50 50,100 0,50z"
Fill="White"
Stretch="Fill"
Stroke="Black"
StrokeThickness="2" />

<Path Width="350"
Height="350"
```

```
HorizontalAlignment="Center"
VerticalAlignment="Center"
Data="M50,0L100,50 50,100 0,50z"
Fill="White"
Stretch="Fill"
Stroke="Black"
StrokeThickness="2" />

<Path Width="250"
Height="250"
HorizontalAlignment="Center"
VerticalAlignment="Center"
Data="M50,0L100,50 50,100 0,50z"
Fill="White"
Stretch="Fill"
Stroke="Black"
StrokeThickness="2" />

<Path Width="150"
Height="150"
HorizontalAlignment="Center"
VerticalAlignment="Center"
Data="M50,0L100,50 50,100 0,50z"
Fill="White"
Stretch="Fill"
Stroke="Black"
StrokeThickness="2" />

<Path Width="50"
Height="50"
HorizontalAlignment="Center"
VerticalAlignment="Center"
Data="M50,0L100,50 50,100 0,50z"
Fill="White"
Stretch="Fill"
Stroke="Black"
StrokeThickness="2" />
```

Для забезпечення зрозумілості діаграми варто також додати такі позначки як назви категорій даних та відмітки з значеннями кожного кроку даних. Ці позначки можна реалізувати у вигляді елементів Label розташованих відповідним чином.

Останнім елементом, необхідним для реалізації цієї діаграми є елемент, який відображатиме показники значень пріорітетів оптимізації на області даних. Його можна відобразити за допомогою елементу WPF під назвою Polygon, який відображає багатокутник за заданими вершинами. Тому, для

реалізації цього методу необхідно розробити метод, який би забезпечував обчислення цих вершин відповідно до обчислених даних про пріорітети оптимізації. Фрагмент коду, який виконує ці обчислення:

```
private void updatePoints()
{

var performancePointsX = 225;
var performancePointsY = 225;

if (ProjectPrioritySystem.Performance > 0)
{
performancePointsY -= 20 + 50 *
(ProjectPrioritySystem.Performance - 1);
}

var stabilityPointsX = 225;
var stabilityPointsY = 225;

if (ProjectPrioritySystem.Stability > 0)
{
stabilityPointsX += 20 + 50 * (ProjectPrioritySystem.Stability -
1);
}

var codeVolumePointsX = 225;
var codeVolumePointsY = 225;

if (ProjectPrioritySystem.CodeVolume > 0)
{
codeVolumePointsY += 20 + 50 * (ProjectPrioritySystem.CodeVolume
- 1);
}

var resourceConsumptionPointsX = 225;
var resourceConsumptionPointsY = 225;

if (ProjectPrioritySystem.ResourceConsumption> 0)
{
resourceConsumptionPointsX -= 20 + 50 *
(ProjectPrioritySystem.ResourceConsumption - 1);
}

Points = $"{performancePointsX},{performancePointsY}
{stabilityPointsX},{stabilityPointsY}
{codeVolumePointsX},{codeVolumePointsY}
{resourceConsumptionPointsX},{resourceConsumptionPointsY}";
}

}
```

Повний програмний код розробленої програмної системи наведено в додатку А. Приклад роботи розробленого додатку та демонстрація його інтерфейсу представлені на рисунку 3.8.

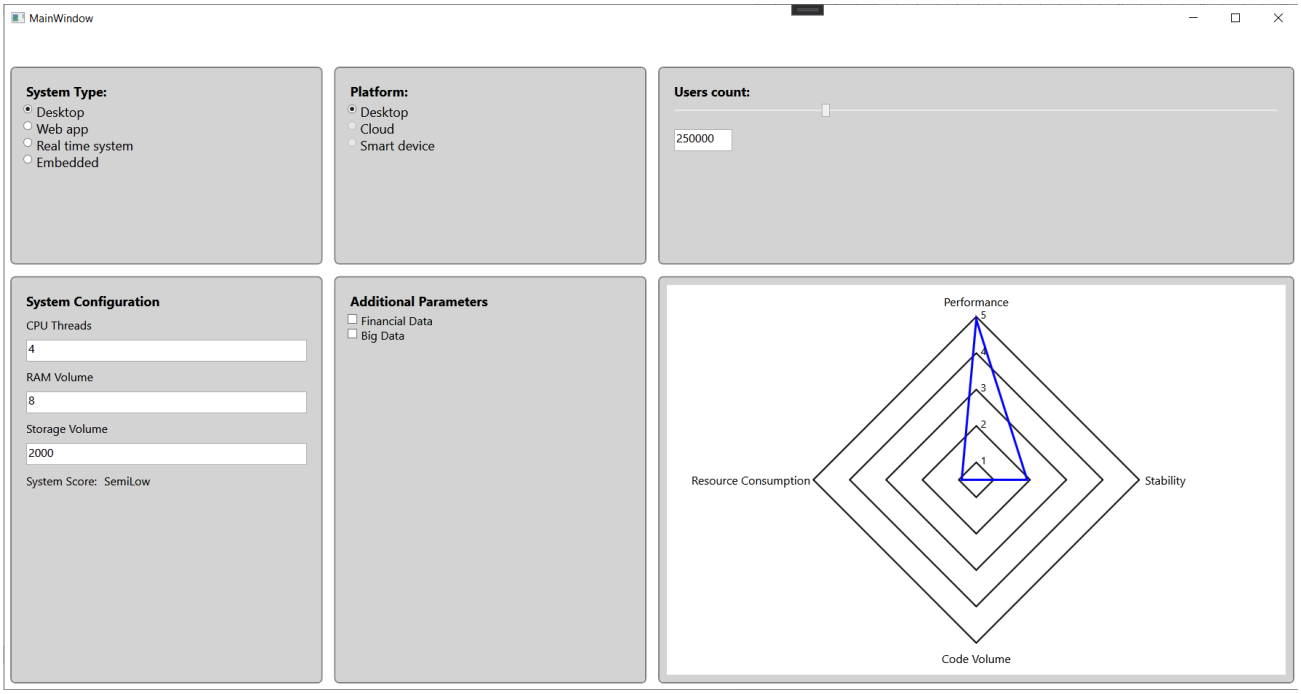


Рисунок 3.8 – Демонстрація роботи програми

Таким чином, було розроблено програмну реалізацію експертної системи для визначення пріорітетів оптимізації програмних продуктів. Розроблений додаток дозволить користувачам здійснювати оцінку пріорітетів оптимізації шляхом введення невеликої кількості даних про проект.

3.7 Висновок

Було детально досліджено можливі шляхи програмної реалізації експертної системи та проведено детальне планування різних аспектів програмної реалізації та було здійснено розробку програмного продукту, який забезпечуватиме здійснення оцінку пріорітетів оптимізаційних процесів.

В першу чергу було обрано інструменти програмної реалізації. Мова програмування C# є сучасною та гнучкою, що дозволить забезпечити

належний рівень програмної реалізації, а технологія WPF дозволить реалізувати сучасний графічний інтерфейс користувача.

Також було визначено основні варіанти використання системи та основний потік робочого процесу експертної системи. Визначено такі етапи процесу роботи експертної системи як:

– визначення типу проекту;
– визначення типу платформи;
– визначення параметрів системи;
– визначення кількості користувачів;
– визначення додаткових параметрів.

Усі ці варіанти використання знайшли своє відображення в спроектованому інтерфейсі користувача. Кожному етапу роботи системи було виділено окремий блок інтерфейсу, та визначено за допомогою яких засобів графічного інтерфейсу здійснюватиметься введення та виведення даних.

Окрім того, було побудовано структуру класів, яка буде застосовуватись в програмній системі. Було визначено типи даних та їх взаємодію для забезпечення роботи експертної системи. Також, було реалізовано паттерн View-Model-ViewModel, який передбачає розділення логіки відображення та логіки даних. Отже, було сформовано структуру класів, яка реалізувала б логіку введення-виведення та її взаємодію з класами, які реалізують логіку експертної системи. Далі, було реалізовано розроблені плани шляхом побудови практичного застосунку згідно обраних архітектурних та функціональних рішень.

Таким чином, було створено програмну систему, яка реалізує на практиці розроблений алгоритм експертної системи для визначення пріорітетів оптимізаційних процесів програмного продукту.

ВИСНОВОК

В результаті виконання дипломної роботи було розроблено систему методів оцінки якості програмних кодів та пріорітетів типів оптимізаційних процесів при розробці програмного продукту.

В першому розділі було досліджено основні проблеми предметної області та обґрунтовано актуальність досліджуваної тематики. Також, було розглянуто основні типи рішень та досліджено найбільш поширені методики, які застосовуються в галузі оцінки якості програмних кодів. На основі цих досліджень було визначено список проблем, які потребують вирішення та основні напрями, в яких варто проводити подальші дослідження.

У другому розділі розглядаються розроблені рішення. Було вдосконалено ряд існуючих метрик оцінки якості програмного коду. Вдосконалено метрику Холстеда шляхом застосування в ній коефіцієнтів типу змінних з метрики Чепіна, що дозволить враховувати в ній складність потоку управління даними. Також було вдосконалено метрику оцінки складності алгоритмів за допомогою врахування середніх значень складності алгоритмів в досліджуваному модулі. Така модифікація забезпечить звернення уваги на можливі проблемні області всередині модулів. Окрім того, було вдосконалено метрику зчеплення модулю шляхом врахування кількості зовнішніх зв’язків. Це забезпечить більшу чутливість метрики до модулів, які мають зовнішні зв’язки та можуть становити проблему для однорідності та зрозумілості програмної системи в цілому. Також, в другому розділі було розроблено метод визначення пріорітетів оптимізаційних процесів в розробці програмних продуктів. Алгоритм роботи цього методу полягає в аналізі різних аспектів розроблюваної програмної системи і визначенні системи пріорітетів оптимізаційних процесів при розробці такого програмного продукту.

Третій розділ був присвячений практичній реалізації алгоритму оцінки пріорітетів оптимізації у вигляді програмного додатку, який би реалізовував функції експертної системи. Зокрема, було розроблено детальний план, який включав в себе варіанти використання системи, алгоритм роботи та робочий

потік програмного додатку, визначення структури класів та проектування графічного інтерфейсу користувача. Згідно розробленого плану було здійснено практичну реалізацію автоматизованої системи, яка виконує обчислення згідно розробленого алгоритму визначення пріорітетів оптимізації на основі даних про проект.

Розроблений програмний продукт дозволяє здійснювати оцінку важливості різних аспектів системи, а відповідно і важливість їх оптимізації і дозволяє отримувати результати такого аналізу у вигляді зручної для розуміння діаграми. Застосування такого додатку доцільно для проектів різних масштабів, оскільки будь-який програмний проект має потребу в забезпеченні оптимізації системи.

Таким чином, в результаті виконання дипломної роботи було розроблено ряд методів оцінки якості коду програмного забезпечення та реалізовано і втілено в життя алгоритм оцінки пріорітетів оптимізації різних аспектів програмних проектів.

За результатами дослідження було опубліковано дві наукові статті: одна стаття – у фаховому науковому виданні [11] та одна стаття – у збірнику матеріалів наукової конференції [12].

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Srđan Popić, Gordana Velikic, Hlavač Jaroslav, Zvjezdan Spasic Pavkovic. The Benefits of the Coding Standards Enforcement and its Impact on the Developers Coding Behaviour-A Case Study on Two Small Projects: тези конф. (м. Белград, листопад. 2018 р.). Белград, 2018. – Режим доступу до ресурсу: https://www.researchgate.net/publication/328912784_The_Benefits_of_the_Coding_Standards_Enforcement_and_its_Impact_on_the_Developers_Coding_Behaviour-A_Case_Study_on_Two_Small_Projects.

2. Говорущенко Т. О. Аналіз галузі оцінювання якості програмного забезпечення / Т. О. Говорущенко // Вісник Національного університету "Львівська політехніка". Комп’ютерні системи та мережі. - 2013. - № 773. - С. 41-48.

3. Systems and software engineering -- Systems and software Quality Requirements and Evaluation (SQuaRE) -- Guide to SQuaRE — Режим доступу до ресурсу: https://iso25000.com/index.php/en/iso-25000-standards

4. Martin R. Design Principles and Design Patterns / Robert Martin – Pearson, 2000. – 32 с

5. Martin R. Clean code – A handbook of Agile Software Craftsmanship / Robert Martin. – Pearson, 2009

6. Singh G., Singh D., Singh V. A Study of Software Metrics/ Singh G., Singh D., Singh V. // IJCEM International Journal of Computational Engineering & Management - 2011. – Vol. 11. - P. 22-27

7. Матросова Н. М. Профіль інформаційної системи та інтерфейс користувача як складові методичного забезпечення інформаційної систем / Н. М. Матросова // Інформаційні технології і засоби навчання – 2010, №3 (17)

8. Harmeet Singh, Syed Imtiyaz Hassan. Effect of SOLID Design Principles on Quality of Software: An Empirical Assessment / Harmeet Singh, Syed Imtiyaz Hassan // International Journal of Scientific & Engineering Research, Volume 6, Issue 4, April-2015

9. Edward Y., Larry L. C. Structured Design / Edward Yourdon and Larry L. Constantine.

10. Кудрявцев В. В., Бармак О. В., Форкун Ю. В., Яшина О. М. Підхід до аналізу програмного коду з використанням метрик Холстеда / В. В. Кудрявцев, О. В. Бармак, Ю. В. Форкун, О. М. Яшина // Вісник Хмельницького національного університету – 2021, №3 – 25-29 с.

11. Кудрявцев В. В., Форкун Ю. В. Аналіз та застосування методів оптимізації швидкодії та відмовостійкості програмних продуктів: тези конф. (Збірник наукових праць Конференції АПКН-2021).

ДОДАТОК А

(обов’язковий)

ПРОГРАМНИЙ КОД

А. 1 Програмний код файлу MainWindow.xml

```
<Window x:Class="Analyzer.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation
"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:local="clr-namespace:Analyzer"
        xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
        xmlns:views="clr-namespace:Analyzer.Views"
        Title="MainWindow"
        Width="1800"
        Height="950"
        mc:Ignorable="d">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>

        <Grid.RowDefinitions>
            <RowDefinition Height="40" />
            <RowDefinition />
        </Grid.RowDefinitions>

        <views:ExpertSystemControl Grid.Row="1" />
    </Grid>
</Window>
```

А. 2 Програмний код файлу ExpertSystemControl.xaml

```
<UserControl x:Class="Analyzer.Views.ExpertSystemControl"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation
"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             xmlns:local="clr-namespace:Analyzer.Views"
```

```
             xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
             xmlns:viewModels="clr-
namespace:Analyzer.ViewModels"
             d:DesignHeight="950"
             d:DesignWidth="1800"
             mc:Ignorable="d">
    <UserControl.DataContext>
        <viewModels:ExpertSystemControlViewModel />
    </UserControl.DataContext>

    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition />
            <RowDefinition />
            <RowDefinition />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition />
            <ColumnDefinition />
            <ColumnDefinition />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>

        <DockPanel Margin="8">
            <Border Background="LightGray"
                    BorderBrush="Gray"
                    BorderThickness="2"
                    CornerRadius="6"
                    DockPanel.Dock="Top">
                <StackPanel Orientation="Horizontal">
                    <StackPanel Margin="15">
                        <Label                          FontSize="18"
FontWeight="Bold">System Type:</Label>
                        <RadioButton FontSize="18"
                                     GroupName="SystemType"
                                     IsChecked="{Binding
Path=IsDesktop, Mode=TwoWay}">
                            Desktop
                        </RadioButton>
                        <RadioButton FontSize="18"
                                     GroupName="SystemType"
                                     IsChecked="{Binding
Path=IsWeb, Mode=TwoWay}">
                            Web app
                        </RadioButton>
                        <RadioButton FontSize="18"
                                     GroupName="SystemType"
                                     IsChecked="{Binding
Path=IsRealTime, Mode=TwoWay}">
                            Real time system
                        </RadioButton>
                        <RadioButton FontSize="18"
```

```
                                        GroupName="SystemType"
                                        IsChecked="{Binding
Path=IsEmbedded, Mode=TwoWay}">
                                Embedded
                            </RadioButton>
                        </StackPanel>
                    </StackPanel>
                </Border>
            </DockPanel>

            <DockPanel Grid.Row="0"
                       Grid.Column="1"
                       Margin="8">
                <Border Background="LightGray"
                        BorderBrush="Gray"
                        BorderThickness="2"
                        CornerRadius="6"
                        DockPanel.Dock="Top">
                    <StackPanel Orientation="Horizontal">
                        <StackPanel Margin="15">
                            <Label                          FontSize="18"
FontWeight="Bold">Platform:</Label>
                            <RadioButton FontSize="18"
                                         GroupName="PlatformType"
                                         IsChecked="{Binding
Path=IsDesktopPlatform, Mode=TwoWay}"
                                         IsEnabled="{Binding
Path=IsDesktopPlatformEnabled}">
                                Desktop
                            </RadioButton>
                            <RadioButton FontSize="18"
                                         GroupName="PlatformType"
                                         IsChecked="{Binding
Path=IsCloudPlatform, Mode=TwoWay}"
                                         IsEnabled="{Binding
Path=IsCloudPlatformEnabled}">
                                Cloud
                            </RadioButton>
                            <RadioButton FontSize="18"
                                         GroupName="PlatformType"
                                         IsChecked="{Binding
Path=IsSmartPlatform, Mode=TwoWay}"
                                         IsEnabled="{Binding
Path=IsSmartPlatformEnabled}">
                                Smart device
                            </RadioButton>
                        </StackPanel>
                    </StackPanel>
                </Border>
            </DockPanel>

            <DockPanel Grid.Row="0"
                       Grid.Column="2"
```

```
                    Grid.ColumnSpan="2"
                    Margin="8">
            <Border Background="LightGray"
                    BorderBrush="Gray"
                    BorderThickness="2"
                    CornerRadius="6"
                    DockPanel.Dock="Top">
                <StackPanel Margin="15">
                    <Label  FontSize="18"  FontWeight="Bold">Users
count:</Label>
                    <Slider Height="30"
                            IsSelectionRangeEnabled="True"
                            Maximum="1000000"
                            Minimum="0"
                            SelectionStart="0"
                            Value="{Binding           UsersCount,
Mode=TwoWay}" />

                    <TextBox Name="UsersCountTextbox"
                             Width="80"
                             Height="30"
                             Margin="5"
                             HorizontalAlignment="Left"
                             FontSize="16"
                             Text="{Binding           UsersCount,
Mode=TwoWay}" />
                </StackPanel>
            </Border>
        </DockPanel>

        <DockPanel Grid.Row="1"
                   Grid.RowSpan="2"
                   Grid.Column="0"
                   Margin="8">
            <Border Background="LightGray"
                    BorderBrush="Gray"
                    BorderThickness="2"
                    CornerRadius="6"
                    DockPanel.Dock="Top">
                <StackPanel Margin="15">
                    <Label FontSize="18" FontWeight="Bold">System
Configuration</Label>

                    <Label FontSize="16">CPU Threads</Label>
                    <TextBox Height="30"
                             Margin="5"
                             HorizontalAlignment="Stretch"
                             FontSize="16"
                             Text="{Binding           CpuThreads,
Mode=TwoWay}" />

                    <Label FontSize="16">RAM Volume</Label>
                    <TextBox Height="30"
```

```
                                Margin="5"
                                HorizontalAlignment="Stretch"
                                FontSize="16"
                                Text="{Binding            RamVolume,
Mode=TwoWay}" />

                    <Label FontSize="16">Storage Volume</Label>
                    <TextBox Height="30"
                                Margin="5"
                                HorizontalAlignment="Stretch"
                                FontSize="16"
                                Text="{Binding        StorageVolume,
Mode=TwoWay}" />

                    <StackPanel        VerticalAlignment="Bottom"
Orientation="Horizontal">
                        <Label         HorizontalAlignment="Left"
FontSize="16">System Score:</Label>
                        <Label HorizontalAlignment="Right"
                               Content="{Binding
SystemCapacityScore}"
                               FontSize="16" />
                    </StackPanel>
                </StackPanel>
            </Border>
        </DockPanel>

        <DockPanel Grid.Row="1"
                   Grid.RowSpan="2"
                   Grid.Column="1"
                   Margin="8">
            <Border Background="LightGray"
                    BorderBrush="Gray"
                    BorderThickness="2"
                    CornerRadius="6"
                    DockPanel.Dock="Top">
                <StackPanel Margin="15">
                    <Label                          FontSize="18"
FontWeight="Bold">Additional Parameters</Label>

                    <CheckBox x:Name="financialDataCheckbox"
                              Content="Financial Data"
                              FontSize="16"
                              IsChecked="{Binding       BankData,
Mode=TwoWay}" />
                    <CheckBox x:Name="bigDataCheckbox"
                              Content="Big Data"
                              FontSize="16"
                              IsChecked="{Binding        BigData,
Mode=TwoWay}" />
                </StackPanel>
            </Border>
        </DockPanel>
```

```
<DockPanel Grid.Row="1"
           Grid.RowSpan="2"
           Grid.Column="2"
           Grid.ColumnSpan="2"
           Margin="8">
    <Border Background="LightGray"
            BorderBrush="Gray"
            BorderThickness="2"
            CornerRadius="6"
            DockPanel.Dock="Top">
        <Grid Margin="10" Background="White">
            <Path Width="450"
                  Height="450"
                  HorizontalAlignment="Center"
                  VerticalAlignment="Center"
                  Data="M50,0L100,50 50,100 0,50z"
                  Fill="White"
                  Stretch="Fill"
                  Stroke="Black"
                  StrokeThickness="2" />
            <Path Width="350"
                  Height="350"
                  HorizontalAlignment="Center"
                  VerticalAlignment="Center"
                  Data="M50,0L100,50 50,100 0,50z"
                  Fill="White"
                  Stretch="Fill"
                  Stroke="Black"
                  StrokeThickness="2" />
            <Path Width="250"
                  Height="250"
                  HorizontalAlignment="Center"
                  VerticalAlignment="Center"
                  Data="M50,0L100,50 50,100 0,50z"
                  Fill="White"
                  Stretch="Fill"
                  Stroke="Black"
                  StrokeThickness="2" />
            <Path Width="150"
                  Height="150"
                  HorizontalAlignment="Center"
                  VerticalAlignment="Center"
                  Data="M50,0L100,50 50,100 0,50z"
                  Fill="White"
                  Stretch="Fill"
                  Stroke="Black"
                  StrokeThickness="2" />
            <Path Width="50"
                  Height="50"
                  HorizontalAlignment="Center"
```

```
      VerticalAlignment="Center"
      Data="M50,0L100,50 50,100 0,50z"
      Fill="White"
      Stretch="Fill"
      Stroke="Black"
      StrokeThickness="2" />

<Label Margin="0,0,0,490"
       HorizontalAlignment="Center"
       VerticalAlignment="Center"
       FontSize="16">
    Performance
</Label>
<Label Margin="520,0,0,0"
       HorizontalAlignment="Center"
       VerticalAlignment="Center"
       FontSize="16">
    Stability
</Label>
<Label Margin="0,490,0,0"
       HorizontalAlignment="Center"
       VerticalAlignment="Center"
       FontSize="16">
    Code Volume
</Label>
<Label Margin="0,0,620,0"
       HorizontalAlignment="Center"
       VerticalAlignment="Center"
       FontSize="16">
    Resource Consumption
</Label>

<Label Margin="20,0,0,55"
       HorizontalAlignment="Center"
       VerticalAlignment="Center"
       FontSize="14">
    1
</Label>
<Label Margin="20,0,0,155"
       HorizontalAlignment="Center"
       VerticalAlignment="Center"
       FontSize="14">
    2
</Label>
<Label Margin="20,0,0,255"
       HorizontalAlignment="Center"
       VerticalAlignment="Center"
       FontSize="14">
    3
</Label>
<Label Margin="20,0,0,355"
       HorizontalAlignment="Center"
       VerticalAlignment="Center"
```

```
                        FontSize="14">
                    4
                </Label>
                <Label Margin="20,0,0,455"
                       HorizontalAlignment="Center"
                       VerticalAlignment="Center"
                       FontSize="14">
                    5
                </Label>

                <Polygon Name="poligon"
                         Width="450"
                         Height="450"
                         HorizontalAlignment="Center"
                         VerticalAlignment="Center"
                         Points="{Binding Points}"
                         Stroke="Blue"
                         StrokeThickness="3" />
            </Grid>
        </Border>
    </DockPanel>

</Grid>
</UserControl>
```

А. 3 – Програмний код файлу ViewModel.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Runtime.CompilerServices;
using System.Text;

namespace Analyzer.ViewModels
{
    public abstract class ViewModel : INotifyPropertyChanged
    {
        public       event       PropertyChangedEventHandler
PropertyChanged;
        public   void   OnPropertyChanged([CallerMemberName]
string prop = "")
        {
            if (PropertyChanged != null)
                PropertyChanged(this,                    new
PropertyChangedEventArgs(prop));
        }
    }
}
```

А. 3 – Програмний код файлу AdditionalParameter.cs

```
namespace Analyzer.Enums
{
    public enum AdditionalParameter
    {
        Undefined = 0,
        BankData,
        BigData
    }
}
```

А. 4 – Програмний код файлу PlatformType.cs

```
namespace Analyzer.Enums
{
    public enum PlatformType
    {
        Undefined = 0,
        Desktop,
        Cloud,
        Smart
    }
}
```

А. 5 – Програмний код файлу SystemScore.cs

```
namespace Analyzer.Enums
{
    public enum SystemScore
    {
        Undefined = 0,
        Low,
        SemiLow,
        Medium,
        High
    }
}
```

А. 6 – Програмний код файлу SystemType.cs

```
namespace Analyzer.Enums
{
    public enum SystemType
```

```
    {
        Undefined = 0,
        Desktop,
        Web,
        RealTime,
        Embedded
    }
}
```

А. 7 – Програмний код файлу FileModel.cs

```
namespace Analyzer.Models
{
    public class FileModel
    {
        public string FileName { get; set; }
        public string Path { get; set; }
        public string Content { get; set; }
        public List<FileModel> Children { get; set; } = new
List<FileModel>();
    }
}
```

А. 8 – Програмний код файлу PrioritySystem.cs

```
    public class PrioritySystem
    {
        public int Performance { get; set; }
        public int Stability { get; set; }
        public int ResourceConsumption { get; set; }
        public int CodeVolume { get; set; }

        public PrioritySystem(int preformance, int stability, int
resourceConsumption, int codeVolume)
        {
            Performance = preformance;
            Stability = stability;
            ResourceConsumption = resourceConsumption;
            CodeVolume = codeVolume;
        }
    }
```

А. 9 – Програмний код файлу ProjectProperties.cs

```
namespace Analyzer.Models
{
```

```
    public class ProjectProperties
    {
        public SystemType? ProjectType { get; set; }
        public PlatformType? ProjectPlatformType { get; set; }
        public int? UserCount { get; set; }
        public SystemScore? SystemScore { get; set; }
        public  List<AdditionalParameter>  AdditionalParameters  {
get; set; } = new List<AdditionalParameter>();

        public PrioritySystem CalculatePriorities()
        {
            PrioritySystem prioritySystem = new PrioritySystem(0,
0, 0, 0);

            if (ProjectType.HasValue)
            {
                switch (ProjectType.Value)
                {
                    case SystemType.Desktop:
                        prioritySystem = new PrioritySystem(4, 3,
1, 0);
                        break;
                    case SystemType.Web:
                        prioritySystem = new PrioritySystem(2, 4,
2, 0);
                        break;
                    case SystemType.RealTime:
                        prioritySystem = new PrioritySystem(2, 4,
2, 0);
                        break;
                    case SystemType.Embedded:
                        prioritySystem = new PrioritySystem(1, 2,
2, 3);
                        break;
                }
            }
            else
            {
                return prioritySystem;
            }

            if (ProjectPlatformType.HasValue)
            {
                switch (ProjectPlatformType.Value)
                {
                    case PlatformType.Desktop:
                        prioritySystem.Performance++;
                        if (prioritySystem.CodeVolume < 1)
                        {
                            prioritySystem.ResourceConsumption--
;
                        }
                        else
```

```
                        {
                            prioritySystem.CodeVolume--;
                        }
                        break;
                    case PlatformType.Cloud:
                        prioritySystem.Stability++;
                        prioritySystem.ResourceConsumption--;
                        break;
                    case PlatformType.Smart:
                        prioritySystem.CodeVolume++;
                        prioritySystem.ResourceConsumption++;
                        prioritySystem.Performance--;
                        if      (prioritySystem.Performance      +
prioritySystem.Stability + prioritySystem.CodeVolume +
                            prioritySystem.ResourceConsumption  >
8)
                        {
                            prioritySystem.Stability--;
                        }

                        break;
                }
            }
            else
            {
                normalize(prioritySystem);
                return prioritySystem;
            }

            if (UserCount.HasValue)
            {
                if (UserCount.Value < 1000 || UserCount.Value >
1000000)
                {
                    prioritySystem.Performance--;
                    prioritySystem.Stability++;
                }
                else
                {
                    prioritySystem.Performance++;
                    prioritySystem.Stability--;
                }
            }
            else
            {
                normalize(prioritySystem);
                return prioritySystem;
            }

            if (SystemScore.HasValue)
            {
                switch (SystemScore.Value)
                {
```

```
                    case Enums.SystemScore.SemiLow:
                        prioritySystem.ResourceConsumption++;
                        break;
                    case Enums.SystemScore.Low:
                        prioritySystem.ResourceConsumption += 2;
                        break;
                }

                var unbalance = prioritySystem.Performance +
prioritySystem.ResourceConsumption +
                                prioritySystem.Stability +
prioritySystem.CodeVolume - 8;

                if (unbalance > 0)
                {
                    if (prioritySystem.CodeVolume > unbalance)
                    {
                        prioritySystem.CodeVolume -= unbalance;
                    }
                    else if (prioritySystem.Performance >
unbalance)
                    {
                        prioritySystem.Performance -= unbalance;
                    }
                    else if (prioritySystem.Stability >
unbalance)
                    {
                        prioritySystem.Stability -= unbalance;
                    }
                }
            }
            else
            {
                normalize(prioritySystem);
                return prioritySystem;
            }

            if (AdditionalParameters.Any())
            {
                if
(AdditionalParameters.Contains(AdditionalParameter.BankData))
                {
                    prioritySystem.Stability++;
                    prioritySystem.ResourceConsumption--;
                }

                if
(AdditionalParameters.Contains(AdditionalParameter.BigData))
                {
                    prioritySystem.Performance++;
                    if (prioritySystem.CodeVolume > 0)
                    {
                        prioritySystem.CodeVolume--;
```

```
                    }
                    else if (prioritySystem.ResourceConsumption >
0)
                    {
                        prioritySystem.ResourceConsumption--;
                    }
                    else if (prioritySystem.Stability > 0)
                    {
                        prioritySystem.Stability--;
                    }
                }
            }

            normalize(prioritySystem);

            return prioritySystem;
        }

        private void normalize(PrioritySystem prioritySystem)
        {
            if (prioritySystem.Stability > 5)
            {
                prioritySystem.Stability = 5;
            }

            if (prioritySystem.Performance > 5)
            {
                prioritySystem.Performance = 5;
            }

            if (prioritySystem.CodeVolume > 5)
            {
                prioritySystem.CodeVolume = 5;
            }

            if (prioritySystem.ResourceConsumption > 5)
            {
                prioritySystem.ResourceConsumption = 5;
            }
        }
    }
}
```

А. 9 – Програмний код файлу ExpertSystemControlViewModel.cs

```
namespace Analyzer.ViewModels
{
    public class ExpertSystemControlViewModel: ViewModel
    {
        private bool _isDesktop;
        public bool IsDesktop
        {
```

```
    get { return _isDesktop; }

    set
    {
        _isDesktop = value;
        OnPropertyChanged(nameof(IsDesktop));
        if (value)
        {
            updateBase(SystemType.Desktop);
        }
    }
}


private bool _isWeb;
public bool IsWeb
{
    get { return _isWeb; }

    set
    {
        _isWeb = value;
        OnPropertyChanged(nameof(IsWeb));
        if (value)
        {
            updateBase(SystemType.Web);
        }
    }
}


private bool _isRealTime;
public bool IsRealTime
{
    get { return _isRealTime; }

    set
    {
        _isRealTime = value;
        OnPropertyChanged(nameof(IsRealTime));
        if (value)
        {
            updateBase(SystemType.RealTime);
        }
    }
}


private bool _isEmbedded;
public bool IsEmbedded
{
    get { return _isEmbedded; }
```

```
            set
            {
                _isEmbedded = value;
                OnPropertyChanged(nameof(IsEmbedded));
                if (value)
                {
                    updateBase(SystemType.Embedded);
                }
            }
        }



        private bool _isDesktopPlatform;
        public bool IsDesktopPlatform
        {
            get { return _isDesktopPlatform; }

            set
            {
                _isDesktopPlatform = value;
                OnPropertyChanged(nameof(IsDesktopPlatform));
                if (value)
                {
                    updatePlatform(PlatformType.Desktop);
                }
            }
        }
        private bool _isDesktopPlatformEnabled;
        public bool IsDesktopPlatformEnabled
        {
            get { return _isDesktopPlatformEnabled; }

            set
            {
                _isDesktopPlatformEnabled = value;

OnPropertyChanged(nameof(IsDesktopPlatformEnabled));
            }
        }



        private bool _isCloudPlatform;
        public bool IsCloudPlatform
        {
            get { return _isCloudPlatform; }

            set
            {
                _isCloudPlatform = value;
```

```
                OnPropertyChanged(nameof(IsCloudPlatform));
                if (value)
                {
                    updatePlatform(PlatformType.Cloud);
                }
            }
        }

        private bool _isCloudPlatformEnabled;
        public bool IsCloudPlatformEnabled
        {
            get { return _isCloudPlatformEnabled; }

            set
            {
                _isCloudPlatformEnabled = value;

OnPropertyChanged(nameof(IsCloudPlatformEnabled));
            }
        }




        private bool _isSmartPlatform;
        public bool IsSmartPlatform
        {
            get { return _isSmartPlatform; }

            set
            {
                _isSmartPlatform = value;
                OnPropertyChanged(nameof(IsSmartPlatform));
                if (value)
                {
                    updatePlatform(PlatformType.Smart);
                }
            }
        }

        private bool _isSmartPlatformEnabled;
        public bool IsSmartPlatformEnabled
        {
            get { return _isSmartPlatformEnabled; }

            set
            {
                _isSmartPlatformEnabled = value;

OnPropertyChanged(nameof(IsSmartPlatformEnabled));
            }
        }
```

```
        private int _usersCount;
        public int UsersCount
        {
            get { return _usersCount; }

            set
            {
                _usersCount = value;
                updateUserCount(value);
                OnPropertyChanged(nameof(UsersCount));
            }
        }



        private int? _cpuThreads;
        public int? CpuThreads
        {
            get { return _cpuThreads; }

            set
            {
                _cpuThreads = value;
                updateSystemParameters(_cpuThreads,   _ramVolume,
_storageVolume);
                OnPropertyChanged(nameof(CpuThreads));
            }
        }

        private int? _ramVolume;
        public int? RamVolume
        {
            get { return _ramVolume; }

            set
            {
                _ramVolume = value;
                updateSystemParameters(_cpuThreads,   _ramVolume,
_storageVolume);
                OnPropertyChanged(nameof(RamVolume));
            }
        }

        private int? _storageVolume;
        public int? StorageVolume
        {
            get { return _storageVolume; }

            set
            {
                _storageVolume = value;
```

```
                updateSystemParameters(_cpuThreads,   _ramVolume,
_storageVolume);
                OnPropertyChanged(nameof(StorageVolume));
            }
        }

        private string _systemCapacityScore;
        public string SystemCapacityScore
        {
            get { return _systemCapacityScore; }

            set
            {
                _systemCapacityScore = value;
                OnPropertyChanged(nameof(SystemCapacityScore));
            }
        }



        private bool _bankData;
        public bool BankData
        {
            get { return _bankData; }

            set
            {
                _bankData = value;
                updateAdditionalParams(_bankData, _bigData);
                OnPropertyChanged(nameof(BankData));
            }
        }

        private bool _bigData;
        public bool BigData
        {
            get { return _bigData; }

            set
            {
                _bigData = value;
                updateAdditionalParams(_bankData, _bigData);
                OnPropertyChanged(nameof(BigData));
            }
        }



        private string _points;
        public string Points
        {
            get { return _points; }
```

```
            set
            {
                _points = value;
                OnPropertyChanged(nameof(Points));
            }
        }


        private PrioritySystem _prioritySystem;
        public PrioritySystem ProjectPrioritySystem
        {
            get { return _prioritySystem; }

            set
            {
                _prioritySystem = value;
                updatePoints();

OnPropertyChanged(nameof(ProjectPrioritySystem));
            }
        }

        private ProjectProperties projectProperties;

        public ExpertSystemControlViewModel()
        {
            projectProperties = new ProjectProperties();
            Points = "225,225 225,225 225,225 225,225";
        }

        private void updateBase(SystemType systemType)
        {
            switch (systemType)
            {
                case SystemType.Desktop:
                    IsDesktopPlatform = true;
                    IsDesktopPlatformEnabled = true;

                    IsCloudPlatform = false;
                    IsCloudPlatformEnabled = false;

                    IsSmartPlatform = false;
                    IsSmartPlatformEnabled = false;
                    break;
                case SystemType.Web:
                    IsDesktopPlatform = false;
                    IsDesktopPlatformEnabled = false;

                    IsCloudPlatform = true;
                    IsCloudPlatformEnabled = true;

                    IsSmartPlatform = false;
                    IsSmartPlatformEnabled = false;
```

```
                    break;
                case SystemType.RealTime:
                    IsDesktopPlatformEnabled = true;
                    IsCloudPlatformEnabled = true;
                    IsSmartPlatformEnabled = true;
                    break;
                case SystemType.Embedded:
                    IsDesktopPlatform = false;
                    IsDesktopPlatformEnabled = false;

                    IsCloudPlatform = false;
                    IsCloudPlatformEnabled = false;

                    IsSmartPlatform = true;
                    IsSmartPlatformEnabled = true;
                    break;
            }

            projectProperties.ProjectType = systemType;
            ProjectPrioritySystem                                    =
projectProperties.CalculatePriorities();
        }

        private void updatePlatform(PlatformType platformType)
        {
            projectProperties.ProjectPlatformType = platformType;
            ProjectPrioritySystem                                    =
projectProperties.CalculatePriorities();
        }

        private void updateUserCount(int value)
        {
            projectProperties.UserCount = value;
            ProjectPrioritySystem                                    =
projectProperties.CalculatePriorities();
        }

        private void updateSystemParameters(int? cpu, int? ram,
int? storage)
        {
            if (cpu.HasValue && ram.HasValue && storage.HasValue)
            {
                SystemScore cpuScore;
                SystemScore ramScore;
                SystemScore storageScore;

                if (cpu.Value >= 4)
                {
                    cpuScore = SystemScore.High;
                }
                else if (cpu.Value > 2)
                {
                    cpuScore = SystemScore.Medium;
```

```
                }
                else if (cpu.Value == 2)
                {
                    cpuScore = SystemScore.SemiLow;
                }
                else
                {
                    cpuScore = SystemScore.Low;
                }


                if (ram.Value >= 12)
                {
                    ramScore = SystemScore.High;
                }
                else if (ram.Value > 8)
                {
                    ramScore = SystemScore.Medium;
                }
                else if (ram.Value > 4)
                {
                    ramScore = SystemScore.SemiLow;
                }
                else
                {
                    ramScore = SystemScore.Low;
                }


                if (storage.Value >= 2000)
                {
                    storageScore = SystemScore.High;
                }
                else if (storage.Value > 512)
                {
                    storageScore = SystemScore.Medium;
                }
                else if (storage.Value > 128)
                {
                    storageScore = SystemScore.SemiLow;
                }
                else
                {
                    storageScore = SystemScore.Low;
                }

                var scoreValues = new List<SystemScore> {cpuScore,
ramScore, storageScore};

                projectProperties.SystemScore                    =
scoreValues.Min();
                SystemCapacityScore                              =
projectProperties.SystemScore.ToString();
```

```
                ProjectPrioritySystem                                   =
projectProperties.CalculatePriorities();
            }
        }

        private void updateAdditionalParams(bool isBankData, bool
isBigData)
        {
            var list = new List<AdditionalParameter>();
            if (isBankData)
            {
                list.Add(AdditionalParameter.BankData);
            }
            if (isBigData)
            {
                list.Add(AdditionalParameter.BigData);
            }

            projectProperties.AdditionalParameters = list;
            ProjectPrioritySystem                                       =
projectProperties.CalculatePriorities();
        }

        private void updatePoints()
        {
            var performancePointsX = 225;
            var performancePointsY = 225;

            if (ProjectPrioritySystem.Performance > 0)
            {
                performancePointsY    -=    20    +    50    *
(ProjectPrioritySystem.Performance - 1);
            }


            var stabilityPointsX = 225;
            var stabilityPointsY = 225;

            if (ProjectPrioritySystem.Stability > 0)
            {
                stabilityPointsX    +=    20    +    50    *
(ProjectPrioritySystem.Stability - 1);
            }

            var codeVolumePointsX = 225;
            var codeVolumePointsY = 225;

            if (ProjectPrioritySystem.CodeVolume > 0)
            {
                codeVolumePointsY    +=    20    +    50    *
(ProjectPrioritySystem.CodeVolume - 1);
            }
```

```
            var resourceConsumptionPointsX = 225;
            var resourceConsumptionPointsY = 225;

            if (ProjectPrioritySystem.ResourceConsumption> 0)
            {
                resourceConsumptionPointsX -= 20 + 50 *
(ProjectPrioritySystem.ResourceConsumption - 1);
            }

            Points = $"{performancePointsX},{performancePointsY}
{stabilityPointsX},{stabilityPointsY}
{codeVolumePointsX},{codeVolumePointsY}
{resourceConsumptionPointsX},{resourceConsumptionPointsY}";
        }
    }
}
```

ДОДОАТОК Б

(обов’язковий)

КОПІЇ НАУКОВИХ ПУБЛІКАЦІЙ

DOI 10.31891/2307-5732-2021-297-3-25-29
УДК 004.053:004.054:004.052.2

О. В. БАРМАК, В. В. КУДРЯВЦЕВ, Ю. В. ФОРКУН, О. М. ЯШИНА
Хмельницький національний університет

ПІДХІД ДО АНАЛІЗУ ПРОГРАМНОГО КОДУ З ВИКОРИСТАННЯМ МЕТРИК ХОЛСТЕДА

У роботі наведено результати досліджень різних стандартів, правил та методик написання програмного коду та аналізу їх впливу на якість ПЗ й імовірність виникнення технічних ризиків, пов'язаних з інформаційними процесами всередині системи.

Ключові слова: розробка програмного забезпечення, оцінка ризиків, стандарти в програмуванні, метрики коду, забезпечення якості.

A. V. BARMAK, V. KUDRIAVTSEV, Y. V. FORKUN, O. M. YASHYNA
Khmelnytskyi National University

SOFTWARE CODE ANALYSIS SYSTEM FOR RISK ASSESSMENT AND QUALITY ASSURANCE OF SOFTWARE

The paper presents the results of research of various standards, rules and methods of writing software code and analysis of their impact on software quality and the likelihood of technical risks associated with information processes within the system. Most of the risks that arise while developing software products are due to errors in building the system architecture or writing code. As a solution for such problems, it is proposed to apply the developed set of rules and methods to build the system architecture and assess the quality of writing software objects. Metrics have been developed to estimate the size and complexity of the module by combining elements of Halsted and Chepin metrics. Also, a set of principles for optimizing the structure of the system, also known as SOLID principles, was presented. The application of these principles for system construction and analysis was substantiated in order to minimize risks, ensure the quality of the software system and provide opportunities for easy extensibility of the project. Using these methods will optimize the project both for use and for further development. The need for such optimization processes in terms of risk management is that the clearer the system and the easier it is to expand, the less likely it is that errors will occur in the future when adding new functionality.

Keywords: Software development, risk assessment, standards in programming, code metrics, quality assurance.

Постановка проблеми

На сьогодні у більшості ІТ-проектів все більше приділяється увага якісному написанню коду. Поняття якісного коду у сучасному його розумінні означає не лише програмний код, який виконує поставлену задачу, але і відповідає певним стандартам, а також є легким для розуміння.

Причиною такої еволюції поняття є, в першу чергу, розвиток проблем, які ставляться перед ІТ-галуззю, а відповідно і збільшення об'ємів ІТ-проектів. Також великі проекти вимагають залучення більшої кількості людей, що призводить до зростання важливості збереження єдиного стилю написання коду.

Як наслідок, було розроблено багато різноманітних стандартів програмування, які надають список правил та рекомендацій стосовно написання коду та побудови архітектури програмних систем. Сучасні автори [1] наводять такі переваги стандартизації в програмуванні, як збільшення ефективності, полегшення підтримки та зменшення складності коду. В той же час, відхилення від загальноприйнятих стандартів програмування може вилитись у серйозні проблеми з продуктивністю та захищеністю програмного забезпечення. Впровадження стандартів програмування може не дати позитивних результатів одразу. При впровадженні нового стандарту в проект розробникам може знадобитись певний час, щоб звикнути до нього та адаптувати свій стиль коду до нових правил. Проте, авторами роботи [2] було встановлено що запровадження стандартів з часом зменшує імовірність виникнення помилки у проекті, а відповідно покращує ефективність роботи системи.

Сучасними авторами також відзначається висока ефективність використання метрик в області прогнозування якості програмних продуктів [9]. Метрики поділяють на метрики процесів та метрики кодування [11]. Метрики процесів відносяться до властивостей самого процесу розробки, а метрики кодування – безпосередньо до програмного продукту, який розробляється. Метрики кодування, такі як метрики Холстеда, на сьогоднішній день часто використовуються для аналізу якості програмних систем. Такі метрики застосовуються, зокрема і для розробки аналізаторів коду, які оцінюють якість інших програм [8]. До переваг подібних метрик відносять їх низьку залежність від мов програмування – вони оперують найбільш базовими елементами, які наявні майже в усіх сучасних мовах [12]. Отже, існує простір для покращення ефективності впровадження стандартів програмування, особливо на початкових етапах. Одним з можливих рішень для такої оптимізації є розробка системи методів та правил для аналізу програмного коду. Застосування такої системи дозволить оцінювати якість складових елементів проекту, що забезпечить можливість прийняття рішення про оптимізацію програмного коду та процесів розробки.

Метою роботи є розробка системи методів та правил для аналізу програмного коду з метою забезпечення якості програмного забезпечення (ПЗ) та оцінки ризиків проекту.

Результати досліджень

В результаті проведеного аналізу можна стверджувати, що на сьогоднішній день існує багато різноманітних стандартів та принципів програмування, які дозволяють створити певну систему правил написання коду та побудови архітектури системи, відповідно до потреб проекту.

Аналіз програмного коду для оцінки ризиків і забезпечення якості програмного забезпечення слід починати з проектування програмних систем з дотриманням принципів SOLID [4]. Об'єктно-орієнтоване програмування вводить в розробку ПЗ ряд специфічних підходів до проектування програмних систем. Однією з переваг об'єктно-орієнтованого програмування (ООП) є те, що розробник має змогу виділити сутності та алгоритми, призначені для розв'язання однієї задачі в окремі класи. Проте, саме по собі застосування об'єктно-орієнтованої парадигми розробки не означає, що розробник застрахований від написання заплутаного і незрозумілого коду. Для вирішення такої проблеми Робертом Мартіном було сформульовано п'ять принципів об'єктно-орієнтованого програмування та проектування, які на сьогодні відомі під акронімом SOLID, який можна розшифрувати наступним чином:

– Single responsibility: принцип єдиної відповідальності;
– Open-closed: принцип відкритої закритості;
– Liskov substitution: принцип підстановки Барбари-Лісков;
– Interface segregation: принцип розділення інтерфейсів;
– Dependency inversion: принцип інверсії залежності.

Принцип єдиної відповідальності в системі аналізу програмного коду полягає в тому, що клас повинен бути призначений для чогось одного. Збереження цього принципу призведе до поділу системи на модулі з чітко визначеним функціоналом. Наявність чітко розмежованої системи зробить архітектуру проекту більш зрозумілою та легшою до розширення.

Принцип відкритої закритості в системі аналізу також дозволить зменшити імовірність помилок. Розширення функціоналу класу, замість його кардинальної зміни забезпечує збереження правильної роботи вже існуючого функціоналу.

Принцип підстановки Лісков теж відіграє важливу роль в системі забезпечення якісного коду. Цей підхід також призначений для безпечного розширення функціоналу системи. Перевагою дотримання цього принципу є збереження ключової функціональності в сімействах класів, що покращує розуміння системи в цілому.

Впровадження принципу розділення інтерфейсів в систему аналізу коду дозволить підтримувати ієрархію класів програмної системи у логічному і оптимізованому вигляді. Порушення цього принципу буде сприяти перевантаженню класів непотрібним, або некорисним функціоналом, що буде негативно відбиватися на зрозумілості коду та може сприяти виникненню непередбачуваних помилок.

Принцип інверсії залежностей також відіграватиме значну роль в системі. Впровадження такого принципу має на меті підвищення модульності програмної системи. Програмний продукт, який вдало розділений на модулі значно легше підтримувати та модифікувати ніж той, який представляє собою велику монолітну систему.

Хоча визначення принципів SOLID оперують багатьма термінами з об'єктно-орієнтованої парадигми програмування, вони також можуть бути застосовані і для інших парадигм. Наприклад, багато експертів в області програмування погоджуються, що ці принципи можуть бути застосовані і в області функційного програмування. Також варто зазначити, що принцип єдиної відповідальності був відомий ще за часів імперативної парадигми програмування. Тому, можна стверджувати, що принципи SOLID можна застосовувати не тільки при використанні об'єктно-орієнтованої парадигми, якщо того потребує проект, хоча і основною областю їх застосування залишаються системи з об'єктно-орієнтованою архітектурою.

На сьогоднішній день SOLID є одним із найбільш поширених принципів для об'єктно-орієнтованого програмування. Хоча застосування цих принципів може збільшити об'єм коду та складність розробки, проте слідування SOLID значно покращує якість коду та спрощує підтримку системи. Дослідження показали, що застосування цих принципів може зменшити зчеплення у проекті на 69 % і збільшити зв'язність до 29 % [3].

Таким чином, першим кроком алгоритму використання системи аналізу ризиків і забезпечення якості програмного коду буде застосування принципів SOLID при розробці програмної системи. Це забезпечить якість та зрозумілість програмного коду та поділить проект на легко розширювані модулі.

Кількість отриманих модулів залежить від масштабів розроблюваного проекту. Найпростіші проекти можуть складатись з невеликої кількості модулів. Проекти більшого масштабу можуть складатись з великої кількості модулів. В деяких випадках проект може складатись з декількох підпроектів різної величини – такий підхід називається мікросервісною архітектурою [13].

Після розроблення певної кількості модулів проекту, виникає потреба в оцінці їх якості. Для цих цілей було розроблено багато метрик програмного забезпечення. Метрика програмного забезпечення – це міра, що дозволяє отримати числове значення певних властивостей програмного модуля. В цілому, метрики призначені для визначення складності проекту та прогнозування об'ємів робіт.

Найпростішим класом метрик ПЗ є кількісні метрики. Ці метрики призначені для визначення кількісних характеристик вихідного коду. Класичним представником цього класу є метрика кількості рядків коду (SLOC – Source lines of code). Ця метрика може використовуватись для визначення трудозатрат по проекту. Проте, з появою мов програмування що підтримують запис декількох команд в одному рядку, ця метрика стала застарілою. Їй на зміну прийшла метрика кількості логічних рядків коду, тобто кількість команд. Ще однією можливою альтернативою вимірювання цієї метрики може бути підрахунок кількості операторів в коді. Таким чином, визначається не тільки об'єм коду, але і в певній мірі і об'єм виконуваної програмою роботи. Проте, усі ці метрики значно залежать від особливостей мови програмування, оскільки

різні мови можуть потребувати різної кількості рядків та операторів. Тому, при оцінці цих метрик важливо враховувати специфіку використаної мови. В зв'язку з цим, ці метрики погано підходять для порівняння програм, написаних різними мовами.

Також, до кількісних метрик відносять метрики Холстеда [5]. При використанні метрик Холстеда частково компенсуються недоліки, пов'язані з записом однієї і тієї ж операції різною кількістю рядків. Багато авторів відзначають метрику Холстеда як один з найпоширеніших методів прогнозування надійності програм [9]. Метрику Холстеда можна вдосконалити, скориставшись методами з метрики Чепіна [6]. Обчислення метрики Чепіна передбачає врахування вагових коефіцієнтів, які залежать від типів змінних. Метрика Чепіна поділяє змінні на 4 типи – введені змінні для розрахунків та забезпечення виводу; модифіковані, або створені всередині програми змінні; управляючі змінні; невикористані змінні. Вагові коефіцієнти, на думку автора метрики, розподіляються наступним чином – $a_1 = 1$, $a_2 = 2$ $a_3 = 3$, $a_4 = 0.5$. Ці самі коефіцієнти можна враховувати при обчисленні кількості операндів в метриці Холстеда. Аналогічно, для підрахунку операторів можна також використати вагові коефіцієнти, які будуть рівні кількості аргументів оператора. Таким чином, формули метрики Холстеда приймуть наступний вигляд:

$$n_1 = a_1 p + a_2 m + a_3 c + a_4 t \quad (1)$$

$$n_2 = \sum k_i i \quad (2)$$

$$N_1 = a_1 P + a_2 M + a_3 C + a_4 T \quad (3)$$

$$N_2 = \sum K_i i \quad (4)$$

де p – кількість змінних для розрахунків та забезпечення виводу, m – кількість модифікованих, або створені всередині програми змінних, c – кількість управляючих змінних, t – кількість невикористаних змінних, k_i – кількість операторів з кількістю аргументів i, i – кількість аргументів, p – кількість використань змінних для розрахунків та забезпечення виводу, m – кількість використань модифікованих, або створені всередині програми змінних, c – кількість використань управляючих змінних, t – кількість повторів невикористаних змінних, k_i – кількість використань операторів з i аргументів.

Таким чином, метрика Холстеда буде враховувати не тільки кількісні характеристики коду, але і складність потоку управління даними. Такий підхід дозволить більш точно оцінити складність класу або модуля програмної системи. Варто зазначити, що єдиної універсальної метрики не існує. Будь-які метричні характеристики програми повинні контролюватись в залежності від поставленої задачі та інших метрик. Будь-яка метрика – це лише показник, який в певній мірі може служити індикатором якості програмного коду, проте не варто зводити одну метрику в абсолют і приймати рішення опираючись лише на неї.

Таким чином, можна побудувати систему принципів написання та оцінки програмного коду для забезпечення якості та розширюваності проекту. В якості вихідної точки прийматиметься об'єктно-орієнтована парадигма, оскільки на сьогоднішній день вона найбільш розповсюджена в програмних проектах різних масштабів. За основу слід взяти набір принципів SOLID. Слідування цим принципам забезпечить зрозумілість проекту як на рівні архітектури, так і на рівні окремих модулів та класів. У якості певного мірила складності системи варто взяти метрики коду, наприклад розширену версію метрики Холстеда, яка була представлена вище. Метрики не варто розглядати в контексті лише одного, окремо взятого модуля або класу [7]. Коректним застосуванням значень метрик буде порівняння їх з метриками інших класів та модулів системи. Наприклад, якщо метрики якогось модуля значно більше ніж в середньому по проекту, то це може свідчити про потребу в реорганізації алгоритмів в цьому модулі, або проведенні загального рефакторингу, якщо цей модуль тісно пов'язаний з іншим функціоналом.

Роботу розробленої системи можна зобразити за допомогою схеми, наведеної на рис. 1.

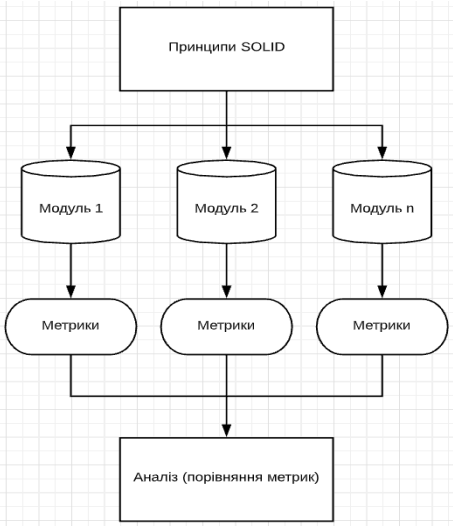


Рис. 1. Схема роботи системи

Розробка програмних модулів проекту ведеться з дотриманням принципів SOLID. Далі, для кожного модуля вираховуються метрики, і на основі порівняння цих метрик проводиться аналіз та приймається рішення про якість модулів та потребу в рефакторингу. Варто зазначити, що ця схема роботи має рекурсивний характер, тобто вона можу бути застосована як на рівні модулів системи, так і на більш низькому рівні всередині самих модулів, по відношенню до підсистем та класів, які належать цьому модулю.

Загалом, така система принципів може бути розширена, відповідно до потреб проекту. Наприклад, можуть бути додані інші метрики для більш детального контролю якості. Але, в загальному, наведеної сукупності принципів і метрик повинно бути достатньо для підтримання якості проектів різних масштабів.

Висновки

Таким чином, в роботі було проведено аналіз сучасних стандартів та метрик програмування. В тому числі, було запропоновано варіант вдосконалення метрики Холстеда, що дозволяє метриці враховувати не тільки кількісні параметри коду, але і складність потоку даних. Також, розроблено систему аналізу програмного коду для забезпечення якості програмних продуктів та оцінки ризиків.

Представлена система фактично базується на використанні відомих принципів та методів і основною її перевагою є те, що вона не тільки наводить основні вказівки з написання програмного коду, але і дозволяє динамічно оцінювати структуру модулів розроблюваної програмної системи, що дозволяє вчасно приймати рішення про необхідність проведення оптимізаційних процесів та рефакторингів над модулями програмної системи. Отже, застосування такої системи аналізу є досить доцільним для проектів з великою кількістю модулів з метою збереження якості програмного продукту та мінімізації ризиків, пов'язаних з помилками в коді або архітектурі системи.

Література

1. Importance of Code Quality and Coding Standard in Software Development. multidots.com. 2020. URL: https://www.multidots.com/importance-of-code-quality-and-coding-standard-in-software-development/.

2. Srđan Popić, Gordana Velikic, Hlavač Jaroslav, Zvjezdan Spasic Pavkovic. The Benefits of the Coding Standards Enforcement and its Impact on the Developers Coding Behaviour-A Case Study on Two Small Projects : тези конф. (м. Белград, листопад 2018 р.). Белград, 2018. – URL : https://www.researchgate.net/publication/328912784_The_Benefits_of_the_Coding_Standards_Enforcement_and_its_Impact_on_the_Developers_Coding_Behaviour-A_Case_Study_on_Two_Small_Projects.

3. Effect of SOLID Design Principles on Quality of Software: An Empirical Assessment. International Journal of Scientific & Engineering Research. 2015.

4. Martin R. Design Principles and Design Patterns. 2000. 32 p.

5. Halstead, Maurice H. Elements of Software Science. Amsterdam: Elsevier North-Holland, Inc. ISBN 0-444-00205-7 1977.

6. Chapin N. An entropy metric for software maintainability. In System Sciences. Vol. II: Software Track, Proceedings of the Twenty-Second Annual Hawaii International Conference on. Vol. 2, pp. 522–523.

7. Fedasyuk D., Yakovyna V., Serdyuk P., Nytrebych O. Variables state-based software usage model. ECONTECHMOD. AN INTERNATIONAL QUARTERLY JOURNAL. 2014. Vol. 3. No. 2. P. 15–20.

8. Афанасова А.И. Программа по оценке качества академических программных продуктов на основе методики Холстеда / А.И. Афанасова // Программные продукты и системы. – 2015. – № 4 (112). – С. 256–260.

9. Аверьянов А. В. Применение метрик Холстеда для количественного оценивания характеристик программ ЭВМ / А. В. Аверьянов, И. Н. Кошель, В. В. Кузнецов // Известия высших учебных заведений «Приборостроение». – 2019. – № 11.

10. Цветков В.Я. Метрики сложной детерминированной системы / В.Я. Цветков, А.В. Буравцев // Онтология проектирования. – 2017. – Т. 7, № 3(25). – С. 334–346.

11. Singh G., Singh D., Singh V. A Study of Software Metrics. IJCEM International Journal of Computational Engineering & Management. 2011. Vol. 11. P. 22–27.

12. Thirumalai C., Thirunavukkarasu H., Vidhyagaran G., Seenu K. Software Complexity Analysis Using Halstead Metrics. International Conference on Trends in Electronics and Informatics – ICEI 2017.

13. Клапчук Р. Г. Монолітні веб-сервіси та мікросервіси: порівняння та вибір / Р. Г. Клапчук, В. С. Харченко // Радіоелектронні і комп'ютерні системи. – 2017. – № 1 (81).

References

1. Importance of Code Quality and Coding Standard in Software Development. multidots.com. 2020. URL: https://www./multidots.com/importance-of-code-quality-and-coding-standard-in-software-development/.

2. Srđan Popić, Gordana Velikic, Hlavač Jaroslav, Zvjezdan Spasic Pavkovic. The Benefits of the Coding Standards Enforcement and its Impact on the Developers Coding Behaviour-A Case Study on Two Small Projects : tezy konf. (m. Belhrad, lystopad 2018 r.). Belhrad, 2018. – URL : https://www.researchgate.net/publication/328912784_The_Benefits_of_the_Coding_Standards_Enforcement_and_its_Impact_on_the_Developers_Coding_Behaviour-A_Case_Study_on_Two_Small_Projects.

3. Effect of SOLID Design Principles on Quality of Software: An Empirical Assessment. International Journal of Scientific & Engineering Research. 2015.

4. Martin R. Design Principles and Design Patterns. 2000. 32 p.

5. Halstead, Maurice H. Elements of Software Science. Amsterdam: Elsevier North-Holland, Inc. ISBN 0-444-00205-7 1977.

1. Chapin N. An entropy metric for software maintainability. In System Sciences. Vol. II: Software Track, Proceedings of the Twenty-Second Annual Hawaii International Conference on. Vol. 2, pp. 522–523.

2. Fedasyuk D., Yakovyna V., Serdyuk P., Nytrebych O. Variables state-based software usage model. ECONTECHMOD. AN INTERNATIONAL QUARTERLY JOURNAL. 2014. Vol. 3. No. 2. P. 15–20.

3. Afanasova A.I. Programma po ocenke kachestva akademicheskih programmnyh produktov na osnove metodiki Holsteda / A.I. Afanasova // Programmnye produkty i sistemy. – 2015. – № 4 (112). – S. 256–260.

4. Averyanov A. V. Primenenie metrik Holsteda dlya kolichestvennogo ocenivaniya harakteristik programm EVM / A. V. Averyanov, I. N. Koshel, V. V. Kuznecov // Izvestiya vysshih uchebnyh zavedenij «Priborostroenie». – 2019. – № 11.

5. Cvetkov V.Ya. Metriki slozhnoj determinirovannoj sistemy / V.Ya. Cvetkov, A.V. Buravcev // Ontologiya proektirovaniya. – 2017. – T. 7, № 3(25). – S. 334–346.

6. Singh G., Singh D., Singh V. A Study of Software Metrics. IJCEM International Journal of Computational Engineering & Management. 2011. Vol. 11. P. 22–27.

7. Thirumalai S., Thirunavukkarasu H., Vidhyagaran G., Seenu K. Software Complexity Analysis Using Halstead Metrics. International Conference on Trends in Electronics and Informatics - ICEI 2017.

8. Klapchuk R. H. Monolitni veb-servisy ta mikroservisy: porivniannia ta vybir / R. H. Klapchuk, V. S. Kharchenko // Radioelektronni i kompiuterni systemy. – 2017. – № 1 (81).

О. В. БАРМАК ORCID ID: 0000-0003-0739-9678 alexander.barmak@gmail.com
В. В. КУДРЯВЦЕВ vivern07@gmail.com
Ю. В. ФОРКУН ORCID ID: 0000-0002-7906-4191 forkun@ridne.net
О. М. ЯШИНА ORCID ID: 0000-0001-7816-1662 ksusha.ja@gmail.com

Рецензія/Peer review : 18.04.2021 р. Надрукована/Printed :30.06.2021 р.

УДК 004.051:004.052.2:004.052.3

Кудрявцев В.В., Форкун Ю.В.

Хмельницький національний університет

АНАЛІЗ ТА ЗАСТОСУВАННЯ МЕТОДІВ ОПТИМІЗАЦІЇ ШВИДКОДІЇ ТА ВІДМОВОСТІЙКОСТІ ПРОГРАМНИХ ПРОДУКТІВ

Розглянуто та пропонуються до розробки метод визначення пріорітетів оптимізаційних процесів при розробці програмних систем. В процесі розробки сучасних програмних систем неможливо уникнути процесів оптимізації та рефакторингу. Тому, це питання є одним з основних в рамках галузі забезпечення якості програмного забезпечення. Розроблений метод дозволяє визначати пріорітети різних напрямів оптимізаційних процесів залежно від характеристик проекту.

The method of determining the priorities of optimization processes in the development of software systems is considered and proposed for development. In the process of developing modern software systems, it is impossible to avoid the processes of optimization and refactoring. Therefore, this issue is a major one in the scope of software quality assurance. The developed method allows to determine the priorities of different directions of optimization processes depending on the properties of the project.

Одним із найбільш важливих аспектів якості програмних продуктів є їх продуктивність та швидкодія. Здатність програми обробляти дані в прийнятний для поставленої цілі час завжди була однією з найголовніших цілей забезпечення якості програмних систем. В сучасних реаліях потреба в забезпеченні прийнятної швидкодії системи лише зросла. Причиною цього є збільшення конкуренції на ринку програмних продуктів та зростання важливості такої категорії як великі дані.

Великі дані – це великі масиви даних, які виникають в наслідок використання людьми мережі Інтернет. Їх обробка та розуміння можливі лише з допомогою спеціалізованих інструментів та методів [1]. Дослідження показують, що використання великих даних допомагає зрозуміти тенденції та настрої як людського суспільства в цілому, так і окремих його частин [2].

Метою роботи є аналіз можливих рішень по оптимізації коду та оцінка їх переваг та можливих недоліків.

Найбільш поширені оптимізаційні процеси – це оптимізації швидкодії. В системах реального часу швидкодія важлива, оскільки вся система базується на забезпеченні стабільної роботи програми за чітко визначені проміжки часу. В більш традиційних програмах UI-UX відіграє особливо важливу роль, оскільки він є ключовим аспектом за яким здійснюється конкуренція між декількома додатками з подібним призначенням. Одним з методів рефакторингу є вилучення надлишкових команд, які можуть призвести до сповільнення системи та ускладнення процесів розробки [3].

Оптимізація для покращення стабільності передбачає забезпечення стабільної роботи програми в усіх випадках. Основним фокусом цього напряму оптимізації є забезпечення обробки виключних випадків та збереження стабільності роботи при навантаженні.

Більш специфічним випадком оптимізації є оптимізація об'єму коду. Оптимізація об'єму коду в основному призначена для програм які запускаються на різноманітних мобільних та смарт пристроях, в яких об'єм внутрішньої пам'яті вкрай обмежений.

Таким чином, ми отримуємо три основних напрями оптимізації програмних систем – оптимізація швидкодії, оптимізація відмовостійкості та оптимізація об'єму коду. Усі ці напрями в деякій мірі конфліктують один з іншим – оптимізація об'єму коду може мати негативні наслідки для швидкодії та відмовостійкості, оптимізація швидкодії може зменшити стабільність і т.д. Тому, при розробці програмного проекту варто визначати пріорітети для кожного з цих напрямів.

Перелік посилань

1. Cambridge Dictionary [онлайн ресурс] - https://dictionary.cambridge.org/dictionary/english/big-data

2. Кислова О. М. Великі дані в контексті дослідження проблем сучасного суспільства / О. М. Кислова // Вісник Харківського національного університету імені В.Н. Каразіна – 2019, №42

3. Martin R. Clean code – A handbook of Agile Software Craftsmanship / Robert Martin., 2009

Дані про авторів:

ПІБ автора	Телефон	Email
Кудрявцев Віктор Володимирович	+380982718334	vivern07@gmail.com
Форкун Юрій Вікторович	+380503764019	forkun@ridne.net

ДОДОАТОК В

(обов’язковий)

ПРЕЗЕНТАЦІЙНІ МАТЕРІАЛИ

Автоматизована система аналізу програмного коду для оцінки ризиків і забезпечення якості програмного забезпечення

Кафедра інженерії програмного забезпечення
Розробив: Кудрявцев Віктор Володимирович
Керівник: д.т.н., проф. Бармак О.В.

Постановка задачі

- Об'єкт дослідження: процеси оцінки ризиків та забезпечення якості при розробці програмних продуктів;
- Предмет дослідження: методи, механізми та принципи оцінки ризиків та якості коду програмних продуктів

Завдання роботи

- Завданнями роботи є:

– дослідити існуючі набори принципів програмування та методи оцінки якості коду, виділити можливі проблеми та шляхи їх вирішення;

– охарактеризувати структуру проведення оцінки якості програмних систем та базову модель аналізу якості програмного коду;

– вдосконалити існуючі методи оцінки якості програмного коду;

– розробити метод оцінки важливості аспектів оптимізації програмної системи.

Актуальність задачі

- Основна маса IT проектів має потребу у приділенні уваги написанню якісного коду.
- Швидкий розвиток IT галузі мав також і негативні наслідки для індустрії. В сучасних реаліях існує криза сфери розробки програмного забезпечення. Так, велика кількість IT проектів завершується зі значними перевитратами або навіть провалом.
  - близько 53% проектів закінчується з значними перевитратами фінансів та перевищеннями термінів;
  - близько 18% програмних проектів завершуються провалом.
- Однією з ключових причин виходу проекту за відведені рамки бюджету та часу є низька якість написаного коду.

Актуальність задачі

- Для забезпечення якісного коду було розроблено багато методів та принципів написання коду.
- Основний стандарт моделі якості ПЗ ISO/IEC 25010 виділяє зокрема зручність супроводу як один з елементів моделі якості.
- Таким чином, існує потреба в методиках які б покращили якість написання коду. На сьогоднішній день розроблено багато методів та принципів покращення якості коду та структури програмних проектів, проте для забезпечення ефективних результатів важливо підходити до запровадження таких засобів з увагою до забезпечення їх гармонійного поєднання та попередження виникнення конфліктів між ними.

Наукова новизна

- Удосконалено метод використання метрики Холстеда шляхом врахування параметрів складності потоку управління даними.
- Удосконалено метод використання метрики складності алгоритмів шляхом врахування середнього значення складності алгоритму в модулі.
- Удосконалено метод використання метрики зчеплення модуля шляхом врахування кількості зовнішніх зв'язків.
- Розроблено метод визначення пріорітетів оптимізаційних процесів проекту.

Існуючі рішення

- Архітектурні принципи – надають визначений підхід до побудови архітектури додатку, перевагами їх застосування є збереження однорідної структури проекту та збереження гнучкості системи;
- Метрики – надають набір характеристик, згідно яких можна оцінювати якість коду проекту;
- Рефакторинг – оптимізаційні процеси, призначені для забезпечення підтримки якості розроблюваного програмного продукту.

Вдосконалена метрика холстеда

- Метрика Холстеда – одна з числових метрик кодування, яка обчислюється на основі аналізу числа рядків та синтаксичних елементів коду програми.
- Метрику можна вдосконалити, скориставшись засобами метрики Чепіна, яка враховує призначення змінних в коді програми.
- Таким чином, метрика буде враховувати не тільки кількісні характеристики коду, але і складність потоку управління даними.

$$n1 = a1*p + a2*m + a3*c + a4*t$$

$$n2 = \Sigma ki*i$$

$$N1 = a1*P + a2*M + a3*C + a4*T$$

$$N2 = \Sigma Ki*i$$

p, m, c, t – кількість змінних різних типів

P, M, C, T – кількість повторів змінних різних типів

ki – кількість операторів з кількістю аргументів і

Ki – кількість використань операторів з і аргументів

a1-4 – коефіцієнти типу змінної

Метрика складності алгоритмів

- Визначення складності алгоритмів широко застосовується в розробці програмного забезпечення
- Високі показники складності самі по собі не свідчать про проблеми у коді. Слід приділяти увагу тим алгоритмам, значення яких сильно відрізняється від середньої складності алгоритмів модулю
- Тому доцільно буде враховувати середнє значення складності при розрахунку метрики складності алгоритму

$$C_n = O_n - O_c$$

C_n – відносна складність оцінюваного модулю, O_n – складність оцінюваного модулю, O_c – середня складність всіх модулів.

Метрика зчеплення модулів

- Важливим методом для оцінки якості коду програмної системи є визначення зчеплення та зв'язності програмних модулів.
- Одним з можливих варіантів покращення визначення рівня зчеплення є визначення не тільки типу зчеплення, але і кількості зв'язків з зовнішніми модулями.
- Це дозволить виявляти модулі, які мають зчеплення низьких рівнів, проте використовують його занадто часто, що може спричинити потребу в надлишковій роботі при внесенні змін в зчеплені модулі.

$$C = k_c * n$$

C – показник зчеплення, k_c – коефіцієнт типу зчеплення, n – кількість посилань на зовнішні модулі.

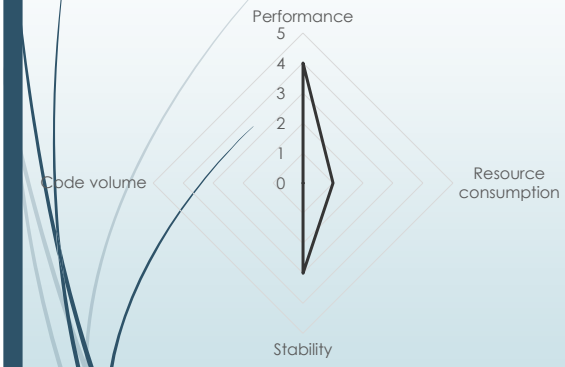
Метрика зчеплення модулів

Тип зчеплення	k_c
Зчеплення за даними **Модуль А здійснює виклик модуля В. Параметри – прості типи даних**	1
Зчеплення за зразком **Модуль А здійснює виклик модуля В. Параметри – структури даних**	3
Зчеплення за управлінням **Модуль А здійснює виклик модуля В і передає управляючі аргументи**	4
Зчеплення за зовнішнім посиланням **Модулі А і В посилаються на одну зовнішню змінну простого типу**	5
Зчеплення за пам'яттю **Модулі А і В посилаються на одну зовнішню структуру даних**	7
Зчеплення за вмістом **Один модуль прямо посилається на зміст іншого модуля не через його точку входу**	9

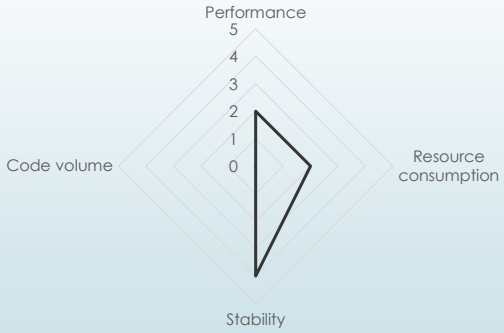
Визначення пріорітету оптимізації

- Оптимізаційні процеси є важливою частиною будь-якого проекту. Майже завжди початковий варіант написаного коду не є оптимальним і потребує додаткового рефакторингу.
- Існує потреба в визначенні пріорітетів напрямів оптимізації.
- В деяких випадках вимоги можуть бути менш специфічними, і виникатиме потреба в побудові структури пріорітетів виходячи з різноманітних даних про проект.

Визначення пріорітету оптимізації



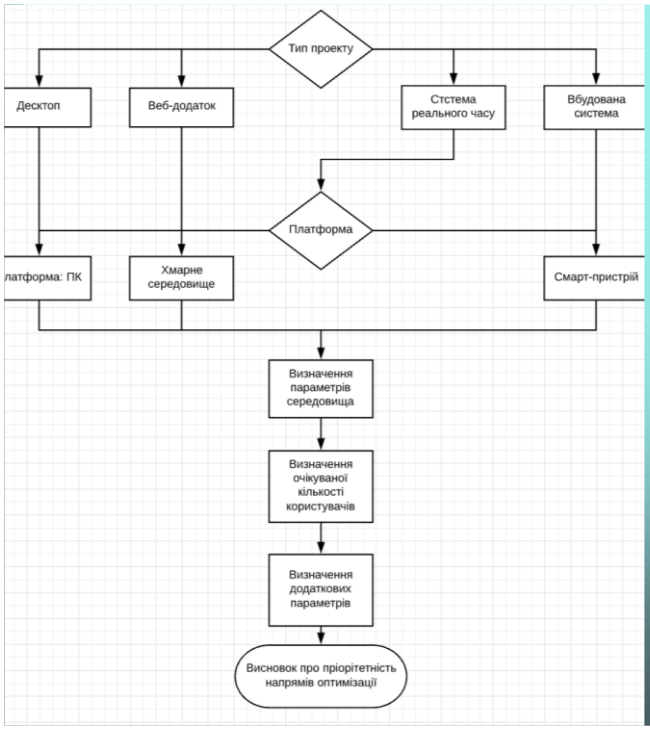
Діаграма пріорітетів оптимізації для користувацьких додатків



Діаграма пріорітетів систем реального часу

Визначення пріорітету оптимізації

Параметр	Опис значення
Тип проекту	Один з декількох можливих варіантів загального формату проекту
Платформа	Один з декількох можливих варіантів середовища, в якому буде працювати проект
Параметри середовища	Технічні параметри середовища в якому працює проект
Очікувана кількість користувачів	Діапазон очікуваного числа користувачів
Додаткові параметри	Додаткова інформація про проект, наприклад важливі типи даних користувачів, з якими працює додаток (особиста інформація, банківські дані), сфера застосування, потреба в безперебійному підключенні до мережі інтернет, тощо.



Визначення пріорітету оптимізації

- Сформована система допоможе більш точно визначити потреби оптимізації розроблюваного програмного проекту.
- Визначення пріорітетів оптимізації допоможе як в безпосередній розробці та супроводі проекту так і в задачах по плануванню робіт та ітерацій на протязі всього життєвого циклу.

Практичне значення

Практична цінність розроблених методів полягає в забезпеченні визначення та контролю якості коду розроблюваного проекту. Підтримка написання якісного коду допоможе проектам в довгостроковій перспективі, адже основними характеристиками якісного коду є однорідність та зрозумілість, які також мають позитивний вплив на легкість підтримки та модифікації програмного продукту з часом.

Висновки

- Було розроблено систему метрик, яка допоможе визначати проблемні місця розроблюваного проекту.
- Також, було розроблено експертну систему для визначення пріорітетів оптимізації проекту.
- В цілому, розроблені системи можуть допомогти в процесі розробки будь-якого програмного проекту.

Список публікацій

- Кудрявцев В. В., Бармак О. В., Форкун Ю. В., Яшина О. М. Підхід до аналізу програмного коду з використанням метрик Холстеда / В. В. Кудрявцев, О. В. Бармак, Ю. В. Форкун, О. М. Яшина // Вісник Хмельницького національного університету – 2021, №3 – 25-29 с.
- Кудрявцев В. В., Форкун Ю. В. Аналіз та застосування методів оптимізації швидкодії та відмовостійкості програмних продуктів: тези конф. (Збірник наукових праць Конференції АПКН-2021).

Завідувачу кафедри інженерії програмного
забезпечення проф. Бедратюку Л. П.
здобувача вищої освіти
Кудрявцева Віктора Володимировича
факультет ІТ, 2 курс, група ІПЗм-20-1

ЗАЯВА

З правилами чинного Положення «Про дотримання академічної доброчесності в Хмельницькому національному університеті» від 26.09.2020 (зі змінами від 26.11.2020), згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та 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 та 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.

23.11.2021 р.
дата

підпис

Fri Dec 03 10:59:51 EET 2021, Хіврич Володимир Русланович, Хмельницький національний університет, ХНУ

Anti-Plagiarism v-15.257

Максимальне співпадіння з одним документом 12.0%

Словники перевірки: en_US, ru_RU, ua_UA. **Помилок в документах: 7%**

ID: 97963 Назва: Автоматизована система аналізу програмного коду для оцінки ризиків та забезпечення якості програмного забезпечення Додано в БД: 2021-12-03 Автора: В.В. Кудрявцев Керівники: О.В. Бармак Консультанти: Опоненти:	Документ		Сумарний збіг по Базі Даних	
	Символи	Лексеми	Символи	Лексеми
	101365	831	15922 (16%)	140 (17%)

Джерело плагіату

ID	Опис	Наявність плагіату в документі	
		Символи	Лексеми
95870	Назва: Звіт з переддипломної практики Додано в БД: 2021-09-27 Автора: В. В. Кудрявцев Керівники: Бармак О. В. Консультанти: Опоненти:	12552 (12.0%)	98 (12.0%)

UNICHECK

Ім'я користувача:
Кафедра ІПЗ

ID перевірки:
1009502371

Дата перевірки:
03.12.2021 12:15:18 EET

Тип перевірки:
Doc vs Internet + Library

Дата звіту:
03.12.2021 12:37:48 EET

ID користувача:
100005589

Назва документа: diplom_Кудрявцев_bez_dod

Кількість сторінок: 85 Кількість слів: 14367 Кількість символів: 113075 Розмір файлу: 641.78 KB ID файлу: 1009513555

8.46%
Схожість

Найбільша схожість: 3.81% з Інтернет-джерелом (http://elar.khnu.km.ua/jspui/bitstream/123456789/10620/1/6-2.pdf)

5.15% Джерела з Інтернету	176	Сторінка 87
3.95% Джерела з Бібліотеки	85	Сторінка 88

0% Цитат

Вилучення цитат вимкнене

Вилучення списку бібліографічних посилань вимкнене

0%
Вилучень

Немає вилучених джерел

Модифікації

Виявлено модифікації тексту. Детальна інформація доступна в онлайн-звіті.

Замінені символи	3

РІШЕННЯ ЕКСПЕРНОЇ КОМІСІЇ
КАФЕДРИ ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ
ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Підтверджуємо ознайомлення з результатом звіту подібності щодо роботи, генерованого системою виявлення текстових збігів/ідентичності/схожості:

Назва: «Автоматизована система аналізу програмного коду для оцінки ризиків та забезпечення якості програмного забезпечення»
Автор: Кудрявцев Віктор Володимирович
Спеціальність: 121 – Інженерія програмного забезпечення
Освітня програма: Освітньо-професійна програма «Інженерія програмного забезпечення»
Науковий керівник: Бармак Олександр Володимирович, доктор технічних наук, професор
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 в 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 2 дні до захисту, разом із заявою щодо самостійності виконання письмової роботи та ідентичності друкованої та електронної версії роботи	
3	Виявлені запозичення не є плагіатом, але частково розміщені в 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 допрацьована 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 в роботі, є законними і не є плагіатом, оскільки:

1) у тексті дипломної роботи системами перевірки на плагіат виявлено схожість з деякими документами в частині загальновживаних обов'язкових словосполучень у стандартних бланках (титулка, бланк завдання), у структурі змісту, назвах розділів/підрозділів тощо та в назвах публікацій у переліку джерел посилання;

2) в якості запозичень системою 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) усі запозичення є фрагментарними або мають належним чином оформленні посилання;

4) виявлені модифікації тексту не впливають на відсоток схожості.

Сумарний обсяг всіх запозичень, визначений системою виявлення збігів ідентичності/ схожості, складає 8,46% і адресується до 176 джерел, що, з урахуванням наведених обгрунтувань, відповідає характеру теми і свідчить на користь дипломної роботи.

Керівник _____ О. В. Бармак

Гарант ОП _____ О. М. Яшина

Завідувач кафедри _____ Л. П. Бедратюк

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА ДИПЛОМНУ РОБОТУ

Дипломник Кудрявцев Віктор Володимирович

Тема Автоматизована система аналізу програмного коду для оцінки ризиків та забезпечення якості програмного забезпечення

Спеціальність 121 – Інженерія програмного забезпечення

Обсяг дипломної роботи:

Кількість листів креслень _____; кількість сторінок записки 122

1.Короткий зміст ДР та прийнятих рішень У магістерській роботі проведено аналіз існуючих підходів та методів оцінки якості коду програмного забезпечення. Запропоновано удосконалені методи оцінки якості коду засновані на метриках Холстеда, зчеплення програмного модулю та оцінки складності алгоритму. Розроблено метод оцінки пріорітетів оптимізації програмної системи. Спроектовано та розроблено програмну реалізацію алгоритму оцінки пріорітетів оптимізації програмних систем.

2. Висновок про відповідність ДР поставленому завданню Дипломна робота освітнього ступеня «магістр» у повній мірі відповідає поставленому завданню як у теоретичній так і 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 цілі і завдання дослідження, описується наукова новизна та практична значимість отриманих результатів. У першому розділі охарактеризовано структуру предметної області, існуючі методи та метрики, які застосовуються для оцінки якості програмних коді, виконана розгорнута постановка задачі. У другому розділі досліджено методи та підходи вирішення поставлених задач. Модифіковано підхід з використанням метрики Холстеда шляхом використання коефіцієнтів типів змінних, які застосовуються в метриці Чепіна, вдосконалено підхід використання метрики оцінки зчеплення модулю шляхом врахування кількості зовнішніх зв’язків та метрики оцінки складності алгоритмів за допомогою врахування відносної складності. Також було розроблено метод для оцінки пріорітетів оптимізації при розробці програмної системи шляхом аналізу інформації про проект. В третьому розділі було виконане всебічне проектування та програмна реалізація методу оцінки пріорітетів оптимізаційних процесів, розробленого в другому розділі. Обґрунтована доцільність використання розробленого методу та наведені рекомендації з подальшого застосування.

4. Позитивні сторони роботи Дипломна робота містить низку інноваційних рішень, зокрема, були вдосконалені підходи з використанням відомих метрик, що дозволить застосовувати їх з метою більш чутливої оцінки програмного коду, а також запровадження нового методу оцінки пріорітетів оптимізації, який забезпечить отримання додаткової інформації про потреби розроблюваної програмної системи ще на етапі початкового проектування.

5. Негативні сторони роботи В роботі не наведено результати експериментальної частини запропонованих підходів

6. Оцінка графічного оформлення та пояснювальної записки роботи Графічне оформлення виконано відповідно до теми дипломної роботи з дотриманням вимог та стандартів. У загальному графічне оформлення виконано на достатньому рівні. Пояснювальна записка відповідає вимогам стандартів до її оформлення.

7. Відгук про роботу в цілому В цілому дипломна робота заслуговує позитивної оцінки. Весь матеріал дипломної роботи структурований, чіткий та послідовний. Усі розділи роботи є послідовними та логічними, що дозволяє чітко зрозуміти викладений матеріал у рамках тематики дипломної роботи. Графічний матеріал дозволяє наочно побачити доцільність та ефективність рішень, які були прийняті для вирішення поставленої задачі.

8. Інші зауваження

9. Оцінка дипломної роботи Розглянувши позитивні та негативні сторони представленої дипломної роботи, можна зробити висновок, що вона заслуговує оцінки «відмінно».

РЕЦЕНЗЕНТ (прізвище, ім'я, по-батькові, посада, місце роботи)
Мартенюк Валерій Володимирович,
зав. кафедри "Автоматизації та комп'ютерно-інтегрованих технологій",
д.т.н., професор

"____" _____ 202__ р. _____
(підпис)