МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ КОМП'ЮТЕРНИХ НАУК ТА ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНИХ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач випускової кафедри
_____ Аліна САВЧЕНКО
«___»_____2023 р.

КВАЛІФІКАЦІЙНА РОБОТА
(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСКНИКА ОСВІТНЬОГО СТУПЕНЯ МАГІСТР
ЗА ОСВІТНЬО-ПРОФЕСІЙНОЮ ПРОГРАМОЮ
«ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ ПРОЕКТУВАННЯ»

Тема: «Програмний засіб оцінювання якості вебзастосунків»

Виконавець: Рустам ПЕРСТЕНЬОВ

Керівник: к.пед.н., доцент Юрій СІНЬКО

Нормоконтролер: к.т.н., доцент Олена ТОЛСТІКОВА

КИЇВ 2023

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет *комп'ютерних наук та технологій*
Кафедра *комп'ютерних інформаційних технологій*
Спеціальність *122 «Комп'ютерні науки»*
Освітньо-професійна програма *«Інформаційні технології проектування»*

ЗАТВЕРДЖУЮ:
завідувач кафедри КІТ
_____Аліна САВЧЕНКО
(підпис)

«____»_____2023 р.

ЗАВДАННЯ
на виконання кваліфікаційної роботи
Перстеньова Рустама Теюб огли
(ПІБ випускника)

1. Тема роботи: «Програмний засіб оцінювання якості вебзастосунків» затверджена наказом ректора № 1976/ст від 29.09.2023р.

2. Термін виконання роботи: з 02 жовтня 2023 року по 31 грудня 2023 року.

3. Вихідні дані до роботи: засіб створено на мові програмування Java, з використанням бази даних MySQL.

4. Зміст пояснювальної записки: 1. Аналіз та поняття процесу визначення якості програмного коду. 2. Методи та моделі оцінки якості програмного коду. 3. Вимоги до програмного засобу. 4. Розробка програмного засобу.

5. Перелік обов'язкового ілюстративного матеріалу: 1. Ієрархія показників якості згідно стандартів. 2. Еталонна модель якості ПЗ. 3. Модель характеристик якості ПЗ. 4. Діаграми послідовностей. 5. Архітектура засобу. 6. Скріншоти роботи програмного засобу.

6. Календарний план-графік

№ з/п	Завдання	Термін виконання	Підпис керівника
1.	Аналіз предметної області та огляд аналогів. Написання 1 розділу, аналіз та поняття технології	02.10.2023- 15.10.2023	
2.	Вибір та опис використаних технологій. Написання 2 розділу, оцінювання коректності розроблюваного засобу	16.10.2023- 29.10.2023	
3.	Написання 3 розділу, архітектура програмної системи	30.10.2023- 12.11.2023	
	Написання 4 розділу, застосування засобу для оцінювання якості вебзастосунків	13.11.2023- 03.12.2023	
4.	Загальне редагування та друк пояснювальної записки	04.12.2023- 10.12.2023	
5.	Проходження нормоконтролю, перепліт пояснювальної записки.	11.12.2023- 18.12.2023	
6.	Розробка тексту доповіді. Оформлення графічного матеріалу для презентації	19.12.2023- 22.12.2023	

7. Дата видачі завдання 02.10.2023 р.

Керівник кваліфікаційної роботи _____ Юрій СІНЬКО
(підпис керівника)

Завдання прийняв до виконання _____ Рустам ПЕРСТЕНЬОВ
(підпис випускника)

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи на тему: «Програмний засіб оцінювання якості вебзастосунків» містить: 87 сторінок, 29 рисунків, 6 таблиць, 39 інформаційних джерела.

Об'єкт дослідження – процес визначення якості програмного коду.

Предмет дослідження – оцінювання вимог коректності коду розроблюваного програмного забезпечення.

Мета кваліфікаційної роботи – створення програмного засобу для оцінювання якості програмного коду вебзастосунку.

Методи дослідження – середовище розробки Visual Studio 2019, платформу .NET Framework 4.8, мову програмування C#, СУБД Microsoft SQL Server 2016.

Задачі дослідження:

- Провести аналіз та розкрито поняття процесу визначення якості програмного коду;
- Описати методи та моделі оцінки якості програмного коду;
- Сформулювати вимоги до програмного засобу;
- Створити програмний засіб оцінювання якості вебзастосунків.

Результати кваліфікаційної роботи рекомендується використовувати при розробці програмного забезпечення для визначення оцінювання коректності вебзастосунків.

ВЕБЗАСТОСУНОК, ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ, ЯКІСТЬ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, НЕФУНКЦІОНАЛЬНІ ВИМОГИ, КОРЕКТНІСТЬ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ, МЕТРИКИ

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

ЖЦ – життєвий цикл.

ТЗ – технічне завдання.

ПЗ – програмне забезпечення.

ПП – програмний продукт.

ПС – програмна система.

СУБД – система управління базами даних.

ВСТУП

Якість програмного забезпечення (ПЗ) є основною його характеристикою в різних сферах використання інформаційних технологій. Вона вказує на ступінь відповідності ПЗ встановленим вимогам та міжнародним стандартам якості [1]. Тому через постійне зростання різноманітного програмного забезпечення у світі, усе більш актуальною стає задача оцінювання якості цього програмного забезпечення, причому на всіх етапах його розробки.

По суті, кожний програмний засіб є перетворювачем між користувачем і комп'ютером, функцією якого є певна обробка даних і виведення отриманих результатів.

Актуальність роботи. На сьогодні розробка програмного забезпечення є однією з найбільших галузей в світовій економіці, в якій задіяно приблизно 3-х мільйонів високоякісних фахівців (програмістів, розробників програмного забезпечення тощо), і безпосередньо від їхньої успішної діяльності залежать ще 5 мільярдів осіб. За багато років еволюції сфери розробки програмного забезпечення, рівень її складності та форми представлення результатів істотно змінилися. Однак, навіть на сьогоднішній день, розробка якісних програмних продуктів залишається проблематичною задачею. У галузі забезпечення якості програмного забезпечення існує криза: великі проекти виконуються з відставанням від графіка або з перевищенням бюджету, розроблені продукти часто не володіють необхідними функціональними можливостями, продуктивність програмного забезпечення низька, а якість не відповідає очікуванням користувачів. За оцінками Standish Group International, витрати на розробку програмного забезпечення складають приблизно 275 мільярдів доларів, проте лише 72% програмних проектів досягають етапу впровадження, і лише 26% завершуються успіхом. Тобто лише 71,5 мільярда доларів витрачаються на успішні проекти, а решта 200 мільярдів доларів витрачаються на невдачі або незавершені проекти.

Знизити ці показники можна лише приділяючи більшу увагу задачі оцінювання якості ПЗ на усіх етапах його ЖЦ.

Перед створенням будь-якого програмного засобу, насамперед, повинні формуватися вимоги до умов виконання його функцій та обробки його даних. Ці вимоги є предметом практичної домовленості між замовником і розробником засобу. Прикладами таких функцій можуть бути бізнес-функції, функції документообігу, управління даними та структурою інформації, необхідною для прийняття системних рішень тощо.

У кваліфікаційній роботі розглядаються методи та засоби визначення показників якості програмного забезпечення при оцінці його коду, а також використання цих показників для його удосконалення і, як наслідок, отримання найкращого кінцевого продукту.

Метою кваліфікаційної роботи є створення програмного засобу для оцінювання якості програмного коду вебзастосунку.

Розроблений програмний засіб повинен визначати якість програмного коду за допомогою показників, отриманих у результаті попереднього аналізу з врахуванням заданих стандартів. Засіб повинен оцінювати вихідний код на основі таких показників якості, як час кодування, цикломатична складність, приховані показники, щільність коментарів і, перш за все, послідовність.

Об'єктом дослідження роботи є процес визначення якості програмного коду.

Предметом дослідження є оцінювання вимог коректності коду розроблюваного програмного забезпечення.

Для досягнення поставленої у роботі мети необхідно виконати наступні **задачі дослідження**:

- Провести аналіз та розкрито поняття процесу визначення якості програмного коду;
- Описати методи та моделі оцінки якості програмного коду;
- Сформулювати вимоги до програмного засобу;
- Створити програмний засіб оцінювання якості вебзастосунків.

РОЗДІЛ 1
АНАЛІЗ ТА ПОНЯТТЯ ПРОЦЕСУ ВИЗНАЧЕННЯ ЯКОСТІ ПРОГРАМНОГО КОДУ

1.1. Поняття якості програмного коду

Показники якості являються однією з найголовніших частин ефективного управління якістю в кожному процесі. Простіше кажучи, метрика якості коду – це міра перетворення функції програмного забезпечення в кількісні показники, що дозволяють вимірювати продуктивність. А саме завдяки можливості вимірювання продуктивності розробка програмного забезпечення може стати більш ефективною [2].

Багато показників можуть бути оцінені лише програмістами, які взаємодіють із програмним забезпеченням або його прототипом, а більшість можна визначити лише шляхом аналізу вихідного коду. Це може значити, що методи оцінювання якості програмного забезпечення можна розділити на дві великі категорії, а саме кількісні та якісні.

Кількісні методи використовують числа. Ці числа вимірюють показники, які можна обчислити. Так як показники числові, то вони можуть підлягати статистичному аналізу, прогресії або порівнянню з подібними показниками. Основна перевага кількісних даних полягає в тому, що їх можна підготувати автоматично, виміряти на будь-якому етапі завершення програмного продукту та легко інтегрувати. Важливо, що для покращення ефективності ці методи повинні мати визначену мету або порівняльну модель [2].

Якісні методи не використовують числа при визначенні показників. Дані можна отримати лише за допомогою опитувань або спостережень.

Такий підхід ускладнює процес впровадження, і може зробити отримані результати менш об’єктивними та складними для обробки, але може бути більш насиченим та інформативним. Якісні методи в більшості випадків вимагають закінчений і готовий до впровадження продук або його прототип і можуть допомогти при подальшому вдосконаленні використовуваних вимірювальних структур [2].

Загалом це означає, що обидві категорії доповнюють одну одну та дозволяють отримати кращий результат, але під час розробки чітко визначені кількісні показники усеж таки матимуть наступні переваги:

1. Показники, які піддаються кількісному вимірюванню, можна відстежувати протягом усього циклу. Щоб побачити результати, не обов’язково треба чекати кінця розробки, виявляючи слабкі сторони та ризики.

2. Продуктивність залежить від часу, витраченого на завдання. Кількісні показники потрібні для визначення пріоритетності завдань при відстежуванні продуктивності і внесення покращення.

3. Кількісні показники можна використовувати як засіб комунікації. Підтримання метрик на цільовому рівні означає, що при відхиленні від цільового прогресу про це, за допомогою встановлених метрик, можна повідомити керівництву, і при подальшому корегуванню це приведе до покращення і обізнаності робочого процесу.

Але щоб визначити, які показники необхідні для покращення та вимірювання якості програмного забезпечення, спочатку необхідно визначити, якими характеристиками повинен володіти кінцевий програмний продукт.

Такі характеристики були визначені та описані в стандарті ISO 9126 – Оцінка програмного забезпечення – Характеристики якості та рекомендації щодо їх використання [3], який був вперше опублікований в 1991 році у зв'язку з тим, що комп'ютер вже тоді став невід'ємною частиною багатьох програм, а його коректна та правильна робота стала мати велике значення в багатьох сферах діяльності людини.

Стандарт розділений на чотири частини:

- якісна модель,
- зовнішні метрики,
- внутрішні метрики,
- показники якості у використанні.

А сама модель якості складається з 6 ключових характеристик: супроводжуваність, ефективність, портативність, надійність, функціональність, зручність використання (рис. 1.1) [4].

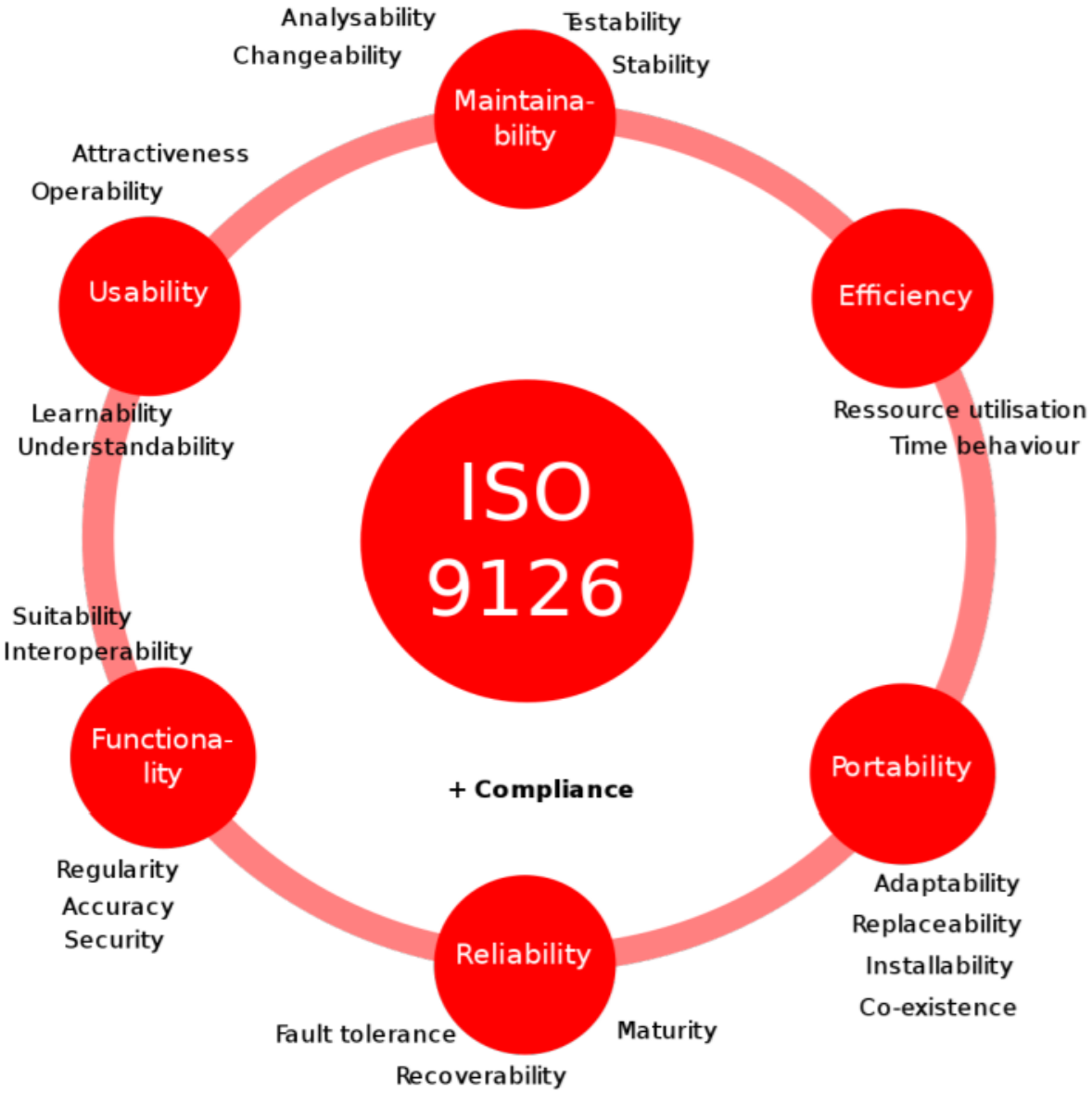

Рис. 1.1. Якість ПЗ відповідно до ISO 9126

Із зростанням кількості ПЗ 01 березня 2011 року ISO/IEC 9126 був замінений на ISO/IEC 25010:2011 Системи та програмна інженерія – Вимоги

та оцінка якості систем та програмного забезпечення (SQuaRE) – Моделі якості системи та програмного забезпечення [5]. Відповідно до нього, якість програмного забезпечення стала ключовим фактором у розробці ПЗ, оскільки вона впливає на задоволення клієнта та визначає успіх проекту. Прийнято рішення, що для того, щоб краще задовольнити сучасні вимоги, необхідно внести низку змін. Тому до цього списку увійшли два додакові пункти (рис. 1.2).

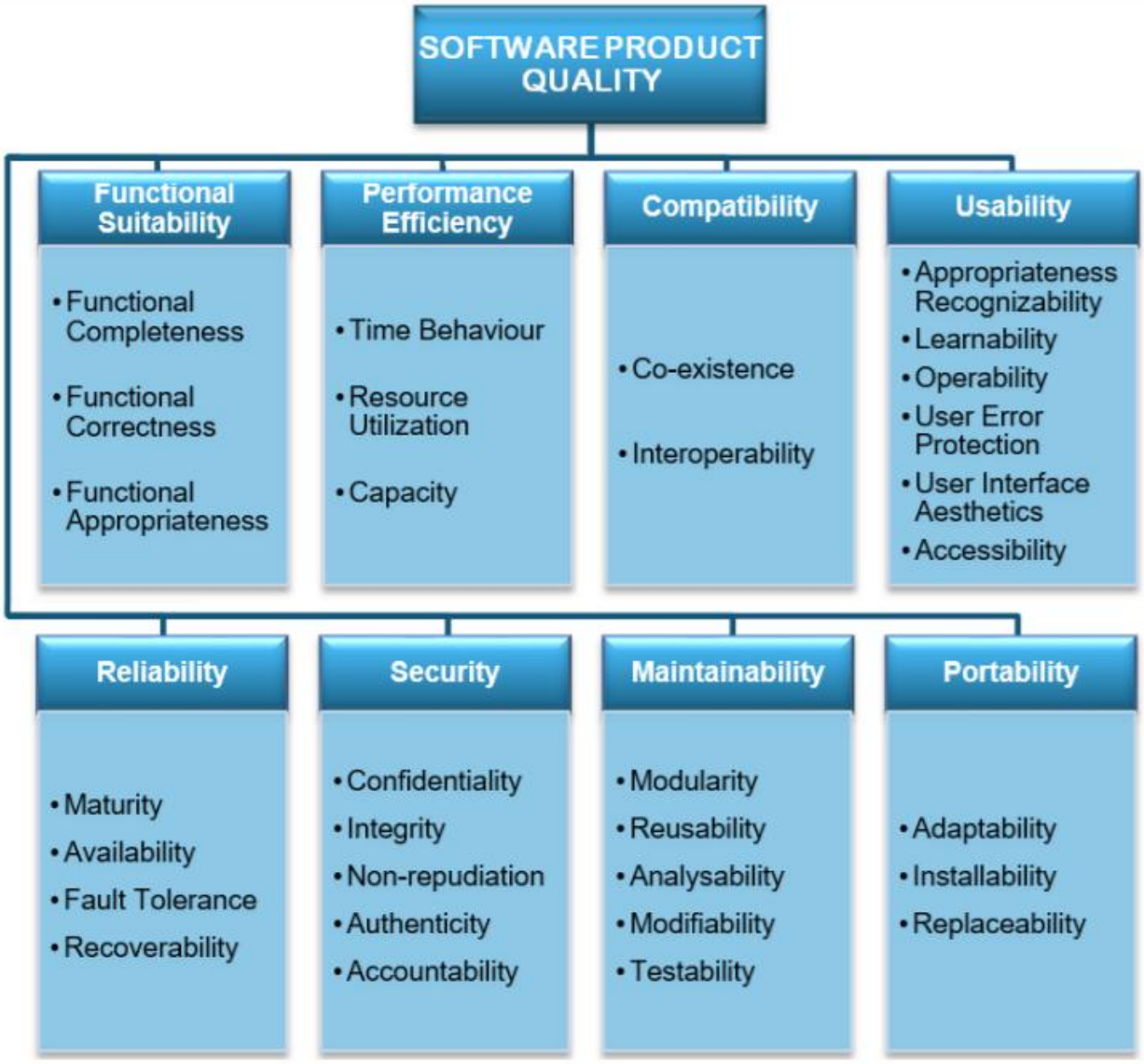

Рис. 1.2. Якість ПЗ відповідно до ISO/IEC 25010:2011

Поняття «якість програмного продукту» включає не тільки повноту і коректність реалізації необхідного функціоналу, але також враховує простоту

підтримки та модифікації програми. Як уникнути труднощів за підтримки нечитаного коду, як для себе, так і для колег?

При створенні програми розробники в першу чергу піклуються про її працездатність, оскільки невідповідність вимогам замовника може негативно вплинути на успіх проекту. Однак завершення розробки та випуск працюючої версії не є кінцевою точкою у життєвому циклі програми. У процесі експлуатації можуть виникнути помилки, що вимагають виправлення, а також нові потреби користувачів, які потребують доопрацювання та випуску нових версій.

Проблема полягає в тому, наскільки скрутним може бути внесення змін до коду, який не редагувався протягом місяців або навіть років. Виправлення помилок та внесення змін до програми «за гарячими слідами», коли її структура та схема роботи всім учасникам розробки зрозумілі, відрізняється від модифікації коду, з яким ще належить ознайомитися. Часто програмісти стикаються з труднощами при аналізі навіть власних кодів, написаних кілька років тому, не кажучи вже про чужі проекти.

Загальна характеристика коду, що впливає на складність його сприйняття та зміни, називається якістю коду. Чим вища ця якість, тим простіше підтримувати та оновлювати програму. В результаті розробники можуть фокусуватися на створенні нового функціоналу, а не витрачати час на виправлення проблем, що виникли через невдалі рішення в минулому.

1.2. Аспекти що впливають на ступінь якості програмного коду

Розглянемо ключові фактори, що впливають на якість програмного коду, та узагальнемо рекомендації щодо його покращення, яке застосовуються у різних проектах та компаніях.

1.2.1. Форматування та оформлення коду

Один з основних аспектів якості коду – його візуальне оформлення, яке суттєво підвищує читання програми та спрощує її підтримку та доопрацювання. Першим важливим моментом, визначеним згідно з

правилами оформлення коду, є стиль відступів виділення структурних блоків програми, як-от тіла функцій і циклів. Важливо уточнювати, чи використовуються символи табуляції або пробіли для відступів, і, у разі прогалин, скільки (звичайно вибір робиться між значеннями 2, 4 і 8).

Хоча здається очевидним використання однорідного стилю відступів хоча б у межах одного файлу, у реальній практиці часто зустрічається змішування стилів, що ускладнює читання коду. Наприклад, якщо в одних місцях для відступів використовуються чотири пробіли, а в інших – табуляція, то в текстовому редакторі із шириною табуляції в чотири пробіли все може виглядати нормально. Однак у редакторі з табуляцією у вісім пробілів код може виглядати заплутаним. Тому важливо забезпечити читання коду незалежно від налаштувань редактора.

Правила форматування коду також включають низку інших аспектів, і їхня однаковість впливає на сприйняття програми:

− використання пробілів поруч із дужками та операторами в арифметичних та логічних виразах (наприклад, чи відокремлювати в конструкції «i=1» знак рівності пробілами чи ні);

− правила розміщення фігурних дужок, для обмеження структурних блоків програми в багатьох мовах (наприклад, чи слід у програмі мовою C ставити відкриваючу фігурну дужку на тому ж рядку, де розташовується логічна умова «if», або потрібно переносити її на новий рядок);

− правила розбиття логічного висловлювання з багатьма умовами на кілька рядків;

− вимоги до вертикального вирівнювання, щоб кілька поспіль подібних інструкцій виглядали красиво:

```
int son = 0;
int daughter = 1;
int father = 2;
```

− інші моменти, часто специфічні для певних мов програмування [6].

1.2.2. Правила іменування

Крім форматування, легкість сприйняття коду значною мірою залежить від способу іменування змінних, функцій, членів класів та інших компонентів програми. Всім відомо принципове значення «імен, що говорять», але в практиці розробки часто виникають неоднозначності в їх застосуванні. Почнемо із запитання: наскільки «розмовляючими» слід робити імена?

У ранні періоди програмування, а також на початкових етапах розвитку мов високого рівня програмування переважно давали короткі імена. Це пояснюється тим, що екрани моніторів мали обмежену горизонтальну роздільну здатність (і в даний час зустрічається обмеження на довжину рядка коду в 80 символів), і вводити імена потрібно було вручну. Інструменти розробки також не були налаштовані на використання довгих ідентифікаторів. Наприклад, розглянемо список функцій POSIX [7] – більшість імен у ньому складаються з менш ніж десяти символів, і ті, що довші, з’явилися наприкінці XX – на початку XXI століття.

Тим не менш, сучасні тенденції дозволяють збільшувати довжину імен, оскільки екрани моніторів розробників зазвичай підтримують більше 80 символів (навіть якщо код пишеться на смартфоні), а засоби розробки надають зручні функції, такі як індексування та автодоповнення. Не дивно, що довжина імен збільшується і ідентифікатори стають більш інформативними. Порівняйте POSIX з іменами класів у .NET [8] або Java [9].

Тим не менш, не варто надто старатися: імена не повинні перетворюватися на розповідь. Зазвичай довжину ідентифікаторів обмежують двома-трьома десятками символів. Функції та глобальні змінні, як правило, мають довші імена, у той час як локальні змінні – більш короткі.

Важливо уникати кількох довгих імен, що відрізняються лише парою букв у середині, оскільки при швидкому перегляді коду вони можуть бути важко помітні.

Необхідно також пам'ятати про класичні назви, наприклад, використання «i» як лічильника циклу або префіксів «get» і «set» для методів, що повертають або встановлюють значення поля.

Якщо розглянути приклади за посиланнями вище, можна побачити, що багато імен складаються з кількох слів. Хоча використання багатослівних імен цілком логічно (особливо для функцій і методів), програмісти-початківці часто стикаються з питанням: як розділяти слова в іменах?

Приклади POSIX, .NET і Java, демонструють популярні методи розподілу слів: використання підкреслення і так званого верблюжого стилю («CamelCase»), коли кожне слово в ідентифікаторі починається з великої літери (можливо, крім першого – відповідні варіанти отримали назву «UpperCamelCase» та «lowerCamelCase»). Також іноді зустрічається використання дефісу як роздільник, хоча у багатьох мовах цей символ не допускається в іменах ідентифікаторів.

Варто зазначити, що у деяких проектах застосовується угорська нотація, яка передбачає додавання префікса до імені змінної для вказівки її типу чи семантичного значення.

Наприклад, рядки можуть мати префікс «s» (sMyName), покажчики – префікс «p» (pMyVariable) і таке інше [6].

1.2.3. Використання констант

Крім змінних, більшість програм активно використовує константні значення, які змінюються під час виконання. Проте слід бути обережними під час використання їх у коді. Рекомендується натомість оголошувати константи зі зрозумілими іменами (якщо мова програмування не підтримує явне визначення констант, то можна скористатися простою змінною) та використовувати ці імена.

Під час перегляду коду людина може запитати себе: чому вибрано саме це число, рядок або інший вираз, і звідки воно взято? Хоча це можна пояснити у коментарях, якщо фіксовані значення використовуються у багатьох місцях, коментувати їх усе може бути недоцільним.

Одним із аргументів на користь визначених констант часто є зручність зміни їх значень. У разі потреби зміни значення, достатньо внести зміни лише в одному місці коду, замість шукати та оновлювати їх у всьому проекті. Навіть якщо ви впевнені, що значення константи ніколи не зміниться, є сенс все одно використовувати для неї певний ідентифікатор.

Прикладом може бути число Пі. Єдиним випадком, коли можливе «перевизначення» його значення – зміна точності (наприклад, якщо буде потрібна більша точність, ніж «3.14»). Проте, навіть якщо передбачається, що значення Пі залишиться незмінним, однаково має сенс використовувати йому визначений ідентифікатор.

Той, хто переглядає програму і розуміє, що вона робить, скажімо, тригонометричні розрахунки, напевно здогадається, що ховається за числом «3.14». Однак чи зможе він розпізнати, що за «1.047» ховається Пі/3? Можна, звісно, писати «3.14/3», але «Pi» та «Pi/3» коротші, зручніші та наочніші [6].

1.2.4. Розмір структурних блоків

Розробники давно дійшли висновку, що для зручності керування та розуміння коду програми краще структурувати його на відносно незалежні блоки. У процедурній парадигмі програмування це можуть бути процедури та функції, а в об'єктно-орієнтованому програмуванні – класи та їх методи. Однак питання про те, наскільки дрібними чи великими слід робити тіла функцій та методів, залишається актуальним.

Часткову відповідь на це питання надають самі програмні парадигми, багато з яких рекомендують створювати модулі, які є самодостатніми, але простими. Наприклад, філософія UNIX заохочує створення програм, які спеціалізовані для виконання конкретного завдання, але гарного його виконання.

Цей підхід також застосовується до функцій та методів класів, при яких кожна функція повинна виконувати одне завдання, а її поведінка має визначатися виключно її аргументами, а не зовнішніми умовами, такими як глобальні змінні.

З погляду візуального сприйняття традиційна рекомендація у тому, щоб тіло підпрограми вміщалося на екрані монітора, щоб його можна було охопити поглядом. Однак ця вимога відносно умовна, оскільки розміри моніторів різні у різних програмістів, і обсяг коду з однією і тією ж функціональністю може змінюватись в залежності від мови програмування.

Наприклад, у мові Perl, яка відома своєю «важкістю читання», можна помістити на пів-екрана логіку, яка в Java може зайняти весь вертикальний розмір монітора (при дотриманні правил форматування та відсутності кількох операторів в одному рядку). Проте сприйняття коду на Perl може бути складнішим.

Також слід враховувати інші показники складності функцій та методів, такі як кількість локальних змінних. Відомо, що середньостатистична людина здатна утримувати увагу близько семи сутностей одночасно. Отже, якщо функція використовує десятки змінних, більшість людей може відчувати труднощі швидкому розумінні її роботи.

Те ж саме стосується рівня вкладеності структурних блоків – занадто багато вкладених умов робить функцію складною до читання і важкою для розуміння [6].

1.2.5. Документованість коду

Широко відомо твердження, що добре написаний код сам собою не вимагає коментарів, оскільки його суть очевидна. Але навіть при ідеальному написанні та положенні коду, коли він читається без допоміжних пояснень, далеко не завжди вдається досягти його розумілості, і в деяких випадках текстові підказки «людською» мовою залишаються необхідними.

Однак слід уникати надмірного коментування, оскільки вивчення коду, який попередньо супроводжується десятками рядків коментарів до кожного другого рядка коду, може бути незручним.

При написанні коментарів часто дотримуються принципу, що більшість з них повинні описувати «що» робить область програми, а не «як» вона це робить. Зазвичай високорівневі структурні елементи програми, такі як

функції та методи класів, мають докладні коментарі, тоді як усередині функцій уникають зайве докладних коментарів. Якщо якісь ділянки вимагають детального опису, їх краще виділити в окремі функції.

Необхідно також приділяти увагу опису обмежень на вхідні параметри функцій та середовища, у яких функція має працювати. Коментарі також корисні під час роботи з ділянками коду, у яких можливі сумніви за певних умов.

У сучасному програмуванні коментарі стають невід'ємною частиною документації до програми. Використовувані системи, такі як Javadoc та Doxygen, автоматично генерують документацію на основі коду та відповідних коментарів.

Автоматична генерація документації може бути застосована, наприклад, до створення опису API бібліотеки, де коментарі до функцій, класів та змінних виділяються як їх опис.

Ефективне використання таких інструментів передбачає строгий стиль оформлення коду, що у свою чергу сприяє однаковості стилю коментарів і, отже, покращує читання коду.

Однак однією з важливих вимог до коментарів є їх відповідність коду. Не завжди після значних змін у коді розробники оновлюють відповідні коментарі. Це може створювати труднощі для людей, які вивчають програму, і питання про те, чи актуальний цей коментар чи містить він помилку в реалізації. Тому відсутність коментаря – це не найкраща практика, але наявність невірного коментаря може бути ще гіршою.

1.2.6. Загальноприйняті стилі оформлення коду

Раніше в роботі розглядалися ключові аспекти, які зазвичай відображаються у стандартах форматування коду. Але для багатьох немає єдиного правильного підходу, проте є кілька альтернатив. Кожен із цих варіантів має свої плюси та мінуси, і вибір конкретного методу часто залежить від естетичних переваг кожного програміста.

Наприклад, угорська нотація, що спочатку запропонована і активно застосовується в Microsoft (особливо в MFC), викликає обурення у творця ядра Linux, Лінуса Торвальдса, який вважає її надмірною і шкідливою.

Також вплив на вибір стилю форматування може надавати і сама мова програмування. Деякі мови, наприклад Python, зменшують кількість спірних моментів, пропонуючи свій підхід до форматування без явних роздільників блоків коду. Однак навіть у таких випадках можливі дискусії, наприклад, щодо використання прогалин або табуляції для відступів.

Автори багатьох сучасних мов часто надають рекомендації щодо стилю коду. Наприклад, для Java [10], C# [11], Perl [12] та інших мов є офіційні рекомендації. У межах цих стандартів можуть існувати деякі варіації між різними командами, але зазвичай мінімальні і легко засвоюються.

Для мов із довшою історією, таких як C, ситуація складніша. Для них є кілька наборів стандартів форматування. Наприклад, у світі широко використовується стиль K&R, заснований на класичній книзі Кернігана і Рітчі.

Також існують альтернативні стилі, такі як стиль Олмана, що застосовується в BSD та багатьох версіях MS Visual Studio, або стиль GNU, що використовується у продуктах однойменного проекту [13].

Нерідко прихильники кожного стилю виражають крайнє обурення стосовно альтернативних підходів. Треба зважати на це, приєднуючись до нової команди.

У багатьох компаніях стиль коду входить до загальних угод про кодування (Coding conventions). Вони можуть включати не тільки правила форматування, але й організацію файлів у проекті, рекомендації щодо використання парадигм програмування, алгоритмів, шаблонів проектування та інше.

Ці угоди спрямовані на підвищення якості програмних продуктів і, найімовірніше, доведеться вивчити їх перед початком роботи у новому середовищі [6].

1.2.7. Інструментальна підтримка

Одним з ключових етапів у забезпеченні високої якості коду є налаштування середовища розробки (IDE) або текстового редактора. Сучасні інструменти розробки та просунуті текстові редактори надають безліч опцій, що впливають на форматування та візуальне оформлення програмного коду.

Тим не менш, редактор не здатний виправити всі недоліки, такі як вибір осмислених імен функцій або встановлення оптимальної довжини змінних імен. Також варто зазначити, що створення коду часто відбувається нерівномірно, з повторним переписуванням та копіюванням із різних джерел. Не всі текстові редактори здатні обробляти всі ці зміни «на льоту».

У проектах часто виникають ситуації, коли змінюються вимоги до коду, і необхідно переформатувати великий обсяг вихідних файлів. У таких випадках можуть допомогти утиліти, що здатні провести аналіз програмного коду на відповідність певним вимогам і, при необхідності, внести корективи.

Інструменти для перевірки стилю кодування мають своє коріння в утиліті «lint», розробленої в кінці 1970-х для аналізу програм мовою C. Сьогодні подібні утиліти існують для багатьох мов програмування, часто включаючи слово «lint» у свою назву.

Наприклад, pylint використовується для перевірки програм Python, а cpplint – мовою C++. Незважаючи на це, існує безліч інших інструментів із зрозумілими назвами, таких як checkstyle для Java.

Зазначимо, термін «lint» став загальноприйнятим для позначення різних статичних перевірок на коректність у різних галузях IT. Наприклад, dlint використовується для виявлення помилок DNS, а fslint – для пошуку непотрібних файлів у файловій системі [6].

При розробці дистрибутивів Linux для контролю якості використовуються інструменти lintian (Debian, Ubuntu та інших) і rpmlint (у системах на основі RPM, таких як Red Hat/Fedora, OpenSUSE, ROSA, ALT Linux та інших).

Інструменти типу lint, як правило, виявляють порушення правил форматування коду, але не вносять зміни. У багатьох випадках виправлення

неможливе без участі людини, наприклад, з некоректними іменами функцій чи змінних. Проте при корекції відступів або розміщення фігурних дужок, автоматизація можлива.

На сьогодні існують відповідні інструменти для багатьох мов програмування, такі як C++ Beautifier, indent (для мови C), PerlTidy, PHP Beautifier, JavaScript Beautifier та багато інших.

Більшість утиліт також можуть інтегруватися в середовища розробки, такі як MS Visual Studio або Eclipse (рис. 1.3).

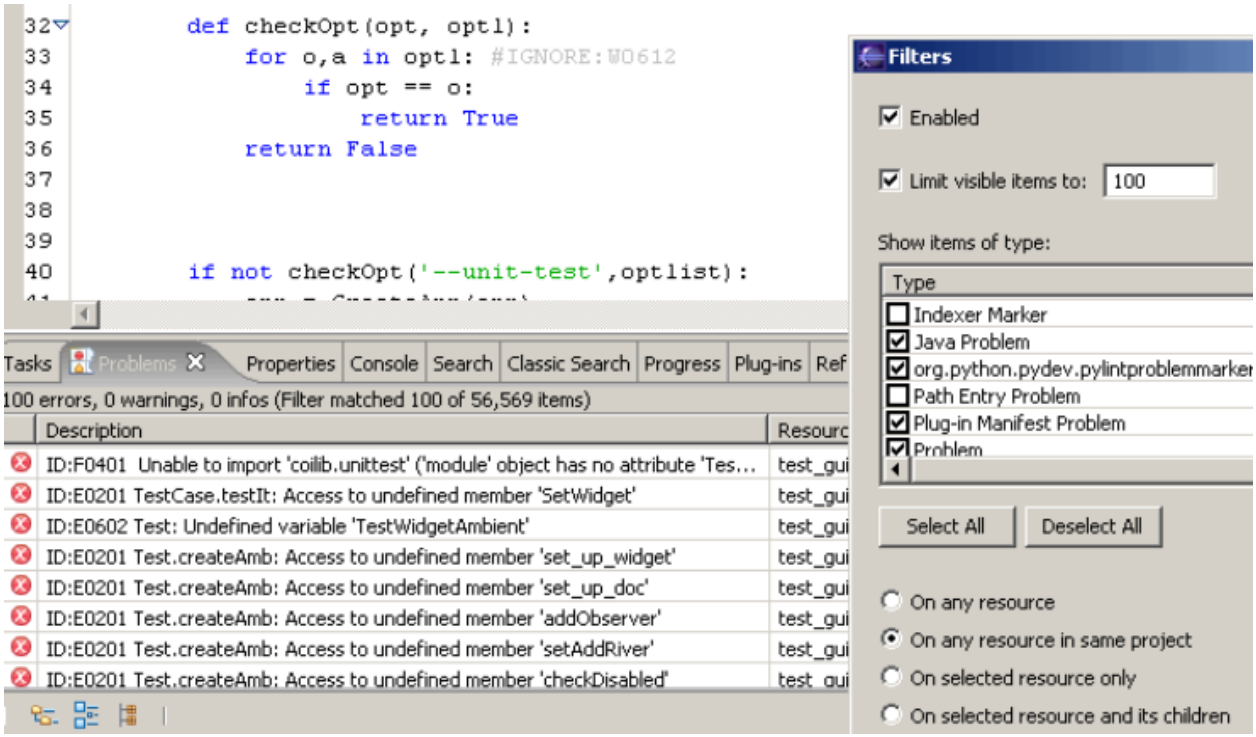

Рис. 1.3. Запуск pylint на проект в Eclipse

Проте запуск різноманітних аналізаторів сам по собі представляє рутинну роботу. Як правило, у великих проектах він здійснюється автоматично. Зокрема, існують інструменти безперервної інтеграції, такі як Hudson та Jenkins, які надають плагіни для запуску різних утиліт аналізу коду та подальшого зручного представлення їх результатів (рис. 1.4) [6].

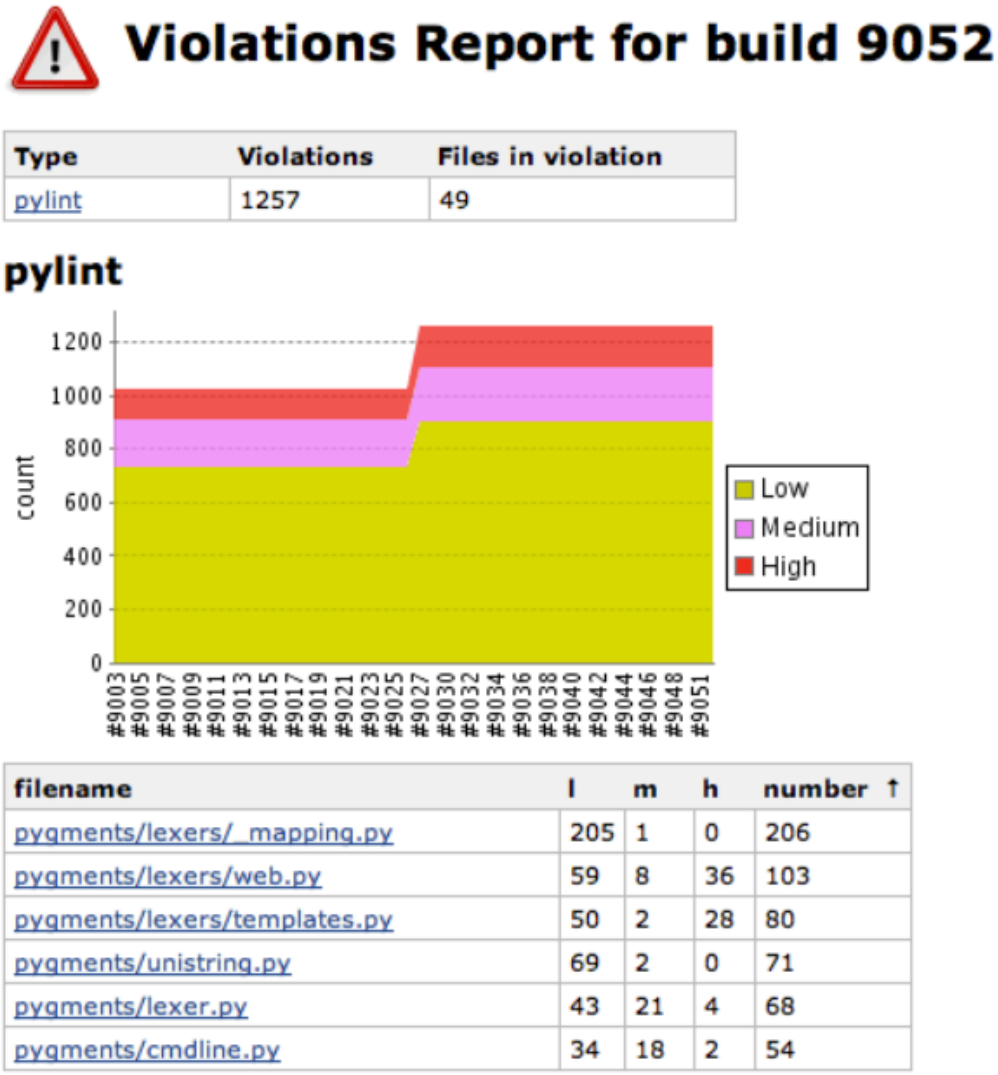

Рис. 1.4. Звіт pylint у Jenkins

1.3. Аналіз програмних засобів для дослідження якості програмного коду

Помилки, які допускаються при розробці програм, часто призводять до наявності вразливостей у програмному забезпеченні. В результаті, нормальна робота програми порушується, що може призвести до змін та пошкоджень даних, зупинки програми або навіть системи в цілому. Більшість вразливостей пов'язані з неправильною обробкою зовнішніх даних або недостатньо суворою їх перевіркою.

Для виявлення таких вразливостей використовуються різні інструменти, наприклад, статичні аналізатори вихідного коду програми [14].

Коли вимога коректної роботи програми порушується для всіх можливих вхідних даних, може виникнути проблема вразливісті безпеки. Вразливість безпеки може призвести до того, що одна програма може бути використаною для обходу обмежень безпеки всієї системи в цілому.

Класифікація вразливостей захисту залежно від програмних помилок включає наступні категорії:

1. Переповнення буфера (buffer overflow): виникає через відсутність контролю над виходом межі масиву у пам'яті під час виконання програми. Коли масив обмеженого розміру переповнюється через великий обсяг даних, вміст суміжних областей пам'яті перезаписується, що призводить до збою та аварійного завершення програми. Переповнення буфера може відбуватися у стеку (stack buffer overflow), купі (heap buffer overflow) та області статичних даних (bss buffer overflow).

2. Вразливості «зіпсованого введення» (tainted input vulnerability): виникають, коли дані, які користувач вводить, передаються інтерпретатору зовнішньої мови без достатнього контролю. Це стосується мов, таких як Unix shell або SQL. Зловмисник може ввести дані таким чином, що інтерпретатор виконає не передбачену команду, що може призвести до вразливості програми.

3. Помилки форматних рядків (format string vulnerability): є підтипом «зіпсованого введення» і походить з недостатнього контролю параметрів при використанні функцій форматного вводу-виводу, таких як printf, fprintf, scanf і т. д. у стандартній бібліотеці мови С. Зловмисник може використовувати цю вразливість, якщо може контролювати форматування рядків, що передаються цими функціями.

4. Уразливості, спричинені помилками синхронізації (race conditions): пов'язані з багатозадачністю, можуть призвести до «стану гонки», коли програма, не призначена до виконання в багатозадачному середовищі, неправильно вважає, що її ресурси можуть бути недоступні до змін іншими програмами. Зловмисник, втручаючись у вміст робочих файлів у потрібний момент, може змусити програму виконувати небажані дії [14].

Крім зазначених вище, існують інші класи вразливостей захисту.

1.4. Огляд популярних аналізаторів коду

Для виявлення вразливостей у програмному забезпеченні застосовуються такі інструменти:

1. Динамічні відладчики: дозволяють здійснювати налагодження програми під час її виконання, що забезпечує можливість виявлення та виправлення помилок у реальному часі.

2. Статичні аналізатори (статичні налагоджувачі): використовують інформацію, одержану в результаті статичного аналізу програми. Статичні аналізатори виявляють потенційні місця помилок у коді програми, вказуючи на фрагменти, де можливі проблеми. Однак ці підозрілі ділянки коду можуть як містити помилку, так і виявитися абсолютно безпечними.

Надалі наведено огляд популярних аналізаторів:

iPlasma

iPlasma є інтегрованим середовищем для аналізу якості об’єктно-орієнтованого ПЗ. iPlasma успішно використовувався для аналізу об’єктно-орієнтованих систем, включаючи великі системи з відкритим кодом (> 1 MLOC), наприклад, Mozilla (C + +, 2560000 LOC) та Eclipse, (Java, 1360000 LOC). iPlasma використовувався у якості консультаційного інструменту для розробників ПЗ, більшість з яких беруть участь в розробці великих програмних додатків для систем телекомунікацій (рис. 1.5.)

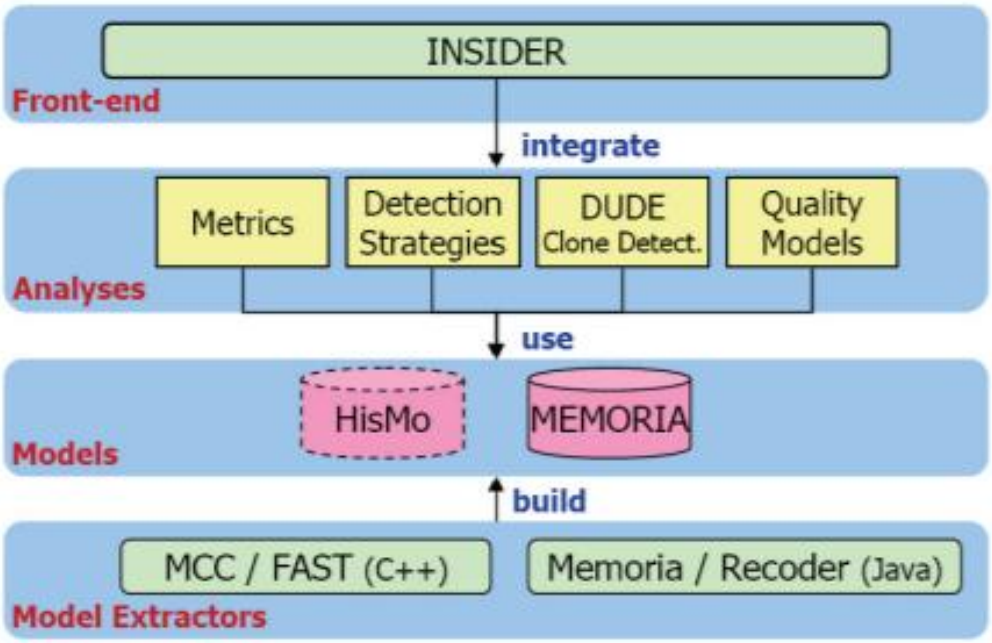

Рис. 1.5. Платформа для аналізу програмного коду iPlasma

BOON

BOON – це інструмент, який автоматизує процес сканування вихідних текстів програм мовою С з використанням глибокого семантичного аналізу. Його мета – виявлення вразливих місць, здатних викликати переповнення буфера. BOON визначає потенційні дефекти, припускаючи, що певні значення є частиною неявного типу з розміром буфера.

CQual

CQual – це інструмент аналізу, призначений для виявлення помилок у програмах на мові С. Цей інструмент розширює мову С з використанням додаткових специфікаторів що призначаються користувачем. Програміст анотує свій код відповідними специфікаторами і CQual перевіряє програму на наявність помилок. Неправильні інструкції вказують на потенційні проблеми, включаючи вразливість форматного рядка.

MOPS

MOPS (MOdel checking Programs for Security) – це інструмент, призначений для пошуку вразливостей у програмах мовою С. Його завдання – динамічне коригування, що забезпечує відповідність програми С статичної моделі. MOPS використовує модель аудиту програмного забезпечення, яка

допомагає визначити, чи програма відповідає набору правил, призначених для створення безпечних програм.

ITS4, RATS, PScan, Flawfinder

Для виявлення помилок переповнення буфера та помилок форматних рядків використовуються статичні аналізатори, такі як ITS4, RATS, PScan та Flawfinder. ITS4 проводить поверхневий семантичний аналіз, виявляючи потенційно небезпечні функції, такі як strcpy/memcpy. RATS поєднує семантичні перевірки від ITS4 з глибоким аналізом у пошуку дефектів, що призводять до переповнення буфера. PScan сканує вихідні тексти на C, виявляючи некоректне використання функцій, аналогічних printf. Flawfinder також є статичним сканером, призначаючи коефіцієнти ризику функцій, що використовуються некоректно.

Всі ці інструменти схожі і використовують лише лексичний та найпростіший синтаксичний аналіз. Тому результати, видані цими програмами, можуть містити до 100% неправдивих повідомлень [14].

Bunch

Bunch – це інструмент аналізу та візуалізації програм мовою C. Він створює граф залежностей, який допомагає аудитору зрозуміти модульну структуру програми.

UNO

UNO – це простий аналізатор вихідного коду, розроблений для виявлення помилок, таких як неініціалізовані змінні, нульові покажчики та вихід за межі масиву. UNO виконує аналіз потоку управління та потоків даних, підтримує як внутрішньо так і міжпроцедурний аналіз, але на даний момент не забезпечує аналіз більш складних програм.

FlexeLint (PC-Lint)

FlexeLint (PC-Lint) – це аналізатор, який призначений для аналізу вихідного коду з метою виявлення різних типів помилок. Програма виконує семантичний аналіз вихідного коду, а також аналіз потоків даних та управління. В кінці процесу аналізу генеруються повідомлення різних типів, такі як можливий нульовий покажчик, проблеми з виділенням пам’яті,

проблеми з потоком управління, можливе переповнення буфера та попередження про поганий стиль коду.

Viva64

Viva64 – призначений для відстеження у вихідному коді C/C++ програм потенційно небезпечних фрагментів, пов'язаних із переходом від 32-бітових систем до 64-бітних. Вбудовуючись у середу Microsoft Visual Studio, Viva64 полегшує зручність роботи з цим інструментом. Цей аналізатор допомагає створювати коректний та оптимізований код для 64-бітових систем.

Parasoft C++test

Parasoft C++test – це спеціалізований інструмент для Windows, призначений для автоматизації аналізу якості коду мовою C++. Пакет C++test проводить аналіз проекту, генерує код для перевірки компонентів у проекті та здійснює важливу роботу з аналізу класів C++. Після завантаження проекту налаштовуються методи тестування, і програма вивчає кожен аргумент методу, значення, що повертаються, і тестові дані. C++test підтримує як внутрішньо так і міжпроцедурний аналіз, і навіть має можливість тестувати незавершений код.

Coverity

Coverity застосовуються для виявлення та усунення дефектів безпеки та якості у критично важливих програмах. Технологія компанії Coverity спрощує процес написання та впровадження складного програмного забезпечення, автоматизуючи пошук та виправлення критичних програмних помилок та недоліків безпеки у процесі розробки. Інструмент Coverity здатний обробляти десятки мільйонів рядків коду із мінімальною позитивною похибкою, забезпечуючи повне покриття траси.

Klocwork K7

Klocwork – це цілий клас продуктів ПЗ розроблених для автоматизованого статичного аналізу коду, виявлення та запобігання дефектам програмного забезпечення та проблемам безпеки. Ці інструменти

допомагають виявляти основні причини дефектів якості та безпеки програмного забезпечення, а також запобігати їх на всіх етапах розробки.

Frama-C

Frama-C – це відкритий набір інструментів для аналізу вихідного коду мовою C. Склад включає ACSL (ANSI/ISO C Specification Language) – спеціальну мову для докладного опису специфікацій функцій C, таких як діапазони допустимих вхідних і вихідних значень. Frama-C забезпечує формальну перевірку коду, пошук потенційних помилок виконання, аудит та рецензування коду, реверс-інжиніринг для кращого розуміння структури та генерацію формальної документації.

CodeSurfer

CodeSurfer – інструмент аналізу програм, не спрямований безпосередньо на пошук вразливостей захисту. Його переваги включають аналіз покажчиків, різні аспекти аналізу потоку даних та використання скриптової мови. CodeSurfer може застосовуватися для пошуку помилок, покращення розуміння коду та реінжинірингу програм. Всередині середовища CodeSurfer було створено прототипний інструмент виявлення вразливостей захисту, але він використовується лише всередині організації розробників.

FxCop

FxCop надає засоби автоматичної перевірки .NET-збірок на відповідність правилам Microsoft .NET Framework Design Guidelines. Використовуючи механізми рефлексії, парсингу MSIL та аналізу графа викликів, FxCop виявляє понад 200 недоліків у галузях архітектури бібліотек, локалізації, правил іменування, продуктивності та безпеки. FxCop також дозволяє створювати власні правила з використанням спеціального SDK і може працювати як у графічному інтерфейсі, так і командному рядку.

JavaChecker

JavaChecker – статичний аналізатор Java програм, заснований на технології TermWare. Цей інструмент виявляє дефекти коду, такі як недбала обробка винятків, приховування імен, порушення стилю, порушення

стандартних контрактів використання та порушення синхронізації. Управління перевірками здійснюється за допомогою керуючих коментарів, а виклик JavaChecker може виконуватися з ANT скрипта.

Simian

Simian – аналізатор подібності, який шукає синтаксис, що повторюється, в декількох файлах. Підтримуючи різні мови програмування, Simian дозволяє налаштовувати правила пошуку коду, що дублюється. Хоча він не має графічного інтерфейсу, Simian може бути запущений з командного рядка або програмно. Результати відображаються в текстовому форматі і можуть бути представлені в XML. Simian допомагає підтримувати цілісність та ефективність проектів, виявляючи повторення коду у великих та маленьких проектах.

ВИСНОВКИ ДО РОЗДІЛУ 1

У даному розділі було проведено аналіз предметної області якості програмного коду. Були розглянуті стандарти та елементи, які відносяться до моделей якості програмного забезпечення. Крім цього, були розглянуті програмні засоби, які можуть бути використані для аналізу вихідного програмного коду.

РОЗДІЛ 2
МЕТОДИ ТА МОДЕЛІ ОЦІНКИ ЯКОСТІ ПРОГРАМНОГО КОДУ

2.1. Моделі якості програмного забезпечення

На сьогоднішній день користувачі програмного забезпечення мають потребу у створенні таких моделей якості програмного забезпечення які б могли оцінити їх як якісно, так і кількісно. Більшість існуючих моделей якості є ієрархічними, заснованими на критеріях якості та пов'язаних з ними показників (метрик). Усі ці моделі можна розподілити на три категорії в залежності від методів, за якими вони були створені. До першого типу належать теоретичні моделі, які базуються на гіпотезах щодо взаємозв'язків між змінними якості. Другий тип охоплює моделі «управління даними», які ґрунтуються на статистичному аналізі. І, нарешті, існує комбінована модель, в якій інтуїція програміста використовується для визначення потрібного типу моделі, а аналіз даних використовується для визначення констант якості моделі.

Модель МакКола

Перша модель якості була представлена МакКолом [15–17]. Ця запропонована модель призначалася для визначення повної характеристики якості програмного продукту через різні його аспекти. Модель якості МакКола визначає три основні напрямки для оцінки та ідентифікації якості програмного забезпечення:

1. Використання (коректність, надійність, ефективність, цілісність, практичність).

2. Модифікація (тестованість, гнучкість, супроводжуваність – фактори якості, що важливі для розробки нової версії програмного забезпечення).

3. Переносимість (мобільність, можливість багаторазового використання, функціональна сумісність – фактори якості, що важливі для переносимості програмного продукту на інші апаратні та програмні платформи).

Модель Боема

Другою основною моделлю якості є модель Боема (рис. 2.1.) [18].

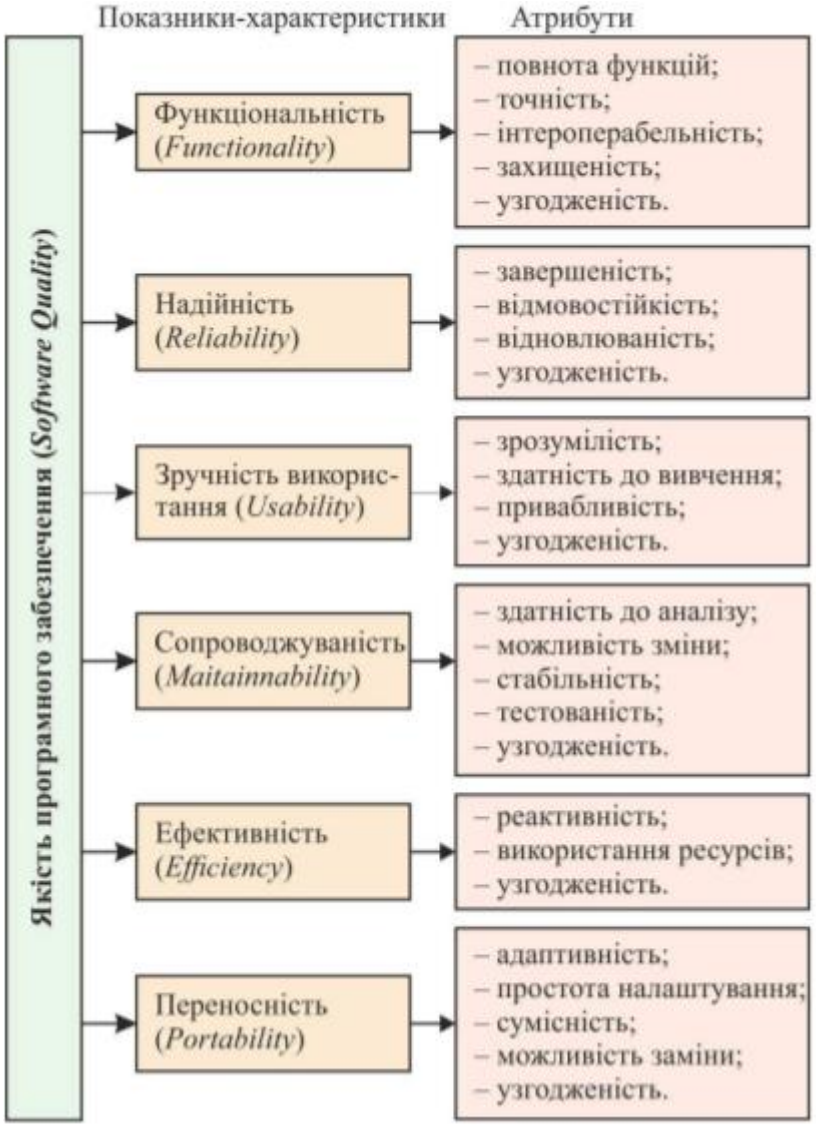

Рис. 2.1. Модель характеристик якості ПЗ

В порівнянні зі сучасними моделями, які автоматично та якісно оцінюють якість програмного забезпечення, модель Боема має свої недоліки. В основному, модель Боема намагається якісно визначити якість програмного забезпечення за допомогою заданого набору показників та метрик. Модель якості Боема охоплює характеристики програмного

забезпечення в більшому масштабі, ніж модель МакКола. Хоча вона подібна до моделі МакКола у тому, що є ієрархічною, структурованою навколо високорівневих, проміжних та примітивних характеристик, кожна з яких вносить свій внесок у рівень якості програмного забезпечення.

Модель FURPS/FURPS+

Акронім FURPS, використовуваний у цій моделі, означає ряд категорій вимог до якості програмного забезпечення:

− Functionality (Функціональність) – особливості, можливості, безпека;

− Usability (Практичність) – людський фактор, ергономічність, документація користувача;

− Reliability (Надійність) – частота відмов, відновлення інформації, прогнозованість;

− Performance (Продуктивність) – час відгуку, пропускна спроможність, точність, доступність, використання ресурсів;

− Supportability (Експлуатаційна придатність) – тестованість, розширюваність, адаптованість, супроводжуваність, сумісність, конфігурованість, обслуговування, вимоги до встановлення, локалізованість [19].

Символ «+» розширює модель FURPS, додаючи до неї:

1. Обмеження проекту – обмеження за ресурсами, вимоги до мов та засобів розробки, вимоги до апаратного забезпечення.
2. Інтерфейс – обмеження, що накладаються на взаємодію із зовнішніми системами.
3. Вимоги до виконання.
4. Фізичні вимоги.
5. Вимоги до ліцензування.

Модель якості FURPS [20], запропонована Грейді та Hewlett Packard, будується подібно до моделей МакКола і Боема, проте вона відрізняється двома шарами: перший визначає характеристики, а другий пов'язаний з атрибутами. Основна концепція FURPS полягає в декомпозиції характеристик програмного забезпечення на дві категорії вимог:

функціональні (F) і нефункціональні (URPS). Ці категорії вимог можна використовувати як вимоги до програмного продукту, так і для оцінки якості програмного продукту. В даний час FURPS+ широко використовується в розробці програмного забезпечення і служить універсальним контрольним списком характеристик ПЗ.

Модель Гецці

Модель Гецці виділяє два аспекти: якість продукту та якість процесу. У моделі Гецці якість програмного забезпечення визначається наступними характеристиками: цілісність, надійність і стійкість, продуктивність, практичність, верифікованість, супроводжуваність, можливість багаторазового використання, мобільність, зрозумілість, можливість взаємодії, ефективність, своєчасність реагування, видимість процесу розробки [20].

Модель Дромі

Модель якості Дромі базується на критеріях оцінки. Мета моделі Дромі – оцінити якість системи, враховуючи унікальні особливості кожного програмного продукту. Вона допомагає передбачити дефекти ПЗ, а також вказує на ті аспекти ПЗ, які, якщо їх проігнорувати, можуть призвести до появи дефектів. Модель основана на взаємозв'язках між характеристиками якості та підхарактеристиками, а також між властивостями програмного забезпечення та характеристиками якості ПЗ [21].

Модель SATC

SATC (Software Assurance Technology Center) була створена у Центрі забезпечення якості програмного забезпечення NASA, та являє собою програму метрик, що дозволяє оцінити ризики проекту, якість продукції та ефективність процесів. Програма SATC рекомендує відслідковувати якість вимог, якість програмного забезпечення, інших продуктів (документації), якість тестування та якість виконання процесів окремо. Модель якості SATC визначає набір цілей, пов'язаних із програмним продуктом та атрибутами процесів, відповідно до структури моделі якості програмного забезпечення ISO 9126-1 [22].

Модель ISO 9126

Стандарт ISO 9126-1 визначає якість програмного забезпечення як будь-яку сукупність його характеристик, що може задовольняти виражені потреби або потреби всіх зацікавлених сторін [23].

Модель якості ISO 9126-1 розрізняє наступні концепції:

− внутрішньої якості, яка стосується характеристик самого програмного забезпечення, незалежно від його поведінки;

− зовнішньої якості, що описує ПЗ з точки зору його поведінки;

− якості ПЗ під час використання в різних контекстах – тобто ту якість, яку користувачі відчувають за конкретних сценаріїв використання програмного забезпечення.

Для всіх цих аспектів якості визначено метрики, які дозволяють їх оцінювати. Крім того, для створення надійного програмного забезпечення важлива якість технологічних процесів його розробки. Взаємозв'язки між цими аспектами якості, як визначено в ISO/IEC 9126 (ISO/IEC 9126-1:2001 [23], ISO/IEC TR 91262:2003 [24], ISO/IEC TR 9126-3:2003 [25], ISO/IEC TR 9126-4:2004 [26]), представлено на рис. 2.2 – 2.3.

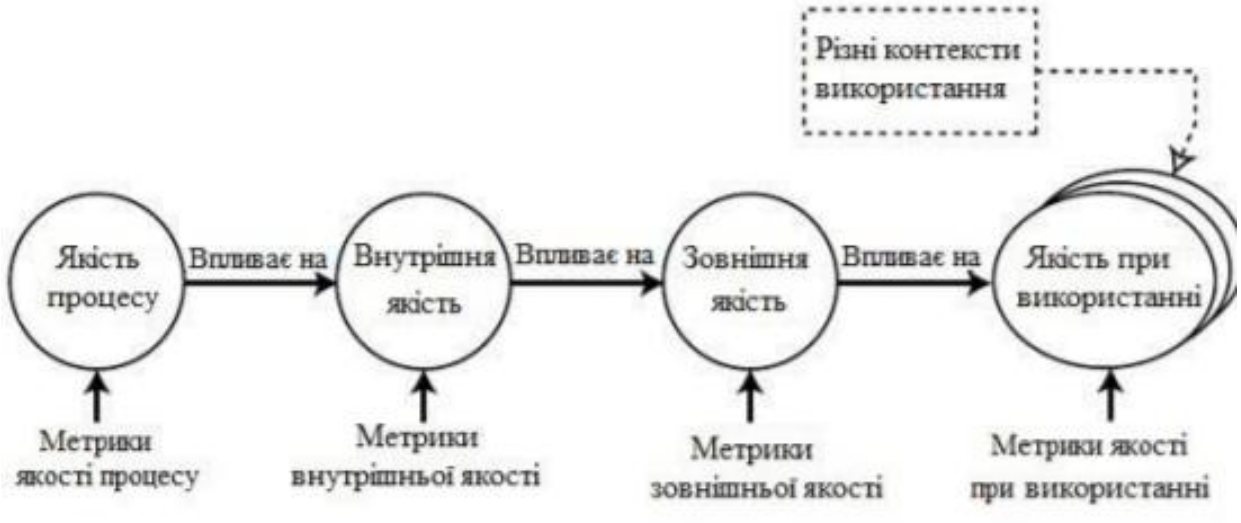

Рис. 2.2. – Основні аспекти якості ПЗ по ISO 9126.

Рис. 2.3. – Характеристики й атрибути якості ПЗ відповідно до ISO 9126.

Модель QMOOD

Ієрархічна модель якості для об'єктно-орієнтованого проекту (QMOOD) була представлена Джагдіш Банзієм та Карлом Девісом. Вона розширила методологію моделі якості Дромі та включила у себе чотири рівні:

1. Визначення показників якості проекту: набір атрибутів якості проекту, які використовуються в QMOOD для опису характеристик об'єктно-орієнтованих систем (функціональність, ефективність, зрозумілість, розширюваність, можливість багаторазового використання та гнучкість).

2. Визначення об'єктно-орієнтованих властивостей проекту: властивості проекту можуть бути визначені у процесі дослідження внутрішньої та зовнішньої структури, функціональності компонентів проекту, атрибутів, методів та класів. Структурною та об'єктно-орієнтованою безліччю властивостей проекту, що використовуються в QMOOD (розмір

проекту, ієрархічна структура, інкапсуляція, пов'язаність, склад проекту, успадкування, поліморфізм, обмін інформацією, складність).

3. Визначення об'єктно-орієнтованих метрик проекту: різні об'єктно-орієнтовані метрики проекту.

4. Визначення об'єктно-орієнтованих властивостей проекту: компоненти проекту визначено за допомогою визначення архітектури об'єктно-орієнтованого проекту. Ця модель визначає парадигму та вводить новий набір об'єктно-орієнтованих метрик [27].

Модель Хосраві

Згідно з моделлю якості Хосраві К. процес оцінки якості включає дві основні задачі:

− вибір глобальних показників;

− вибір підхарактеристик, пов'язаних із глобальними характеристиками.

Ця модель якості акцентується на багаторазовому використанні програмного забезпечення як глобальній характеристиці та наголошує на можливостях багаторазового використання, зрозумілості, гнучкості, модульності, надійності, масштабованості та зручності використання. Модель якості Хосраві та інших авторів пов'язує показники якості та підхарактеристики, користуючись визначеннями IEEE, ISO/IEC та інших моделей якості [28].

Модель Чанга

У моделі Чанга, оцінка якості програмного забезпечення забезпечується на основі теорії нечітких множин та методу аналізу ієрархій. Чанг та ін. визначили керівні принципи і використали цей підхід для моделі якості ISO 9126-1. Оцінки якості програмного забезпечення у моделі Чанга ґрунтуються на характеристиках та підхарактеристиках моделі ISO 9126-1 [29].

Компонентно-орієнтована модель якості

Шарма А. та ін. представили компонентно-орієнтовану модель якості розробки програмного забезпечення, яка включає всі характеристики та

підхарактеристики моделі якості ISO 9126-1, а також пропонує нові підхарактеристики, такі як придатність до повторного використання, гнучкість, складність, простежуваність, масштабованість. У цій моделі для оцінки якості проекту використовується метод аналізу ієрархій [30].

Інші моделі якості

Існують і інші моделі якості, які представляють два відмінні підходи до визначення показників якості протягом усього життєвого циклу програмного забезпечення. Характеристики якості у них діляться на дві групи:

− ефективність, безпека, доступність та функціональність;

− модифікованість, мобільність, можливість багаторазового використання, успадкованість та тестованість [31].

2.2. Оцінювання коректності розроблюваного програмного забезпечення

Згідно до багатьох описаних вище стандартів, контроль якості програмного забезпечення на етапах життєвого циклу передбачає:

1. Перевірку відповідності вимог до проектованого продукту та критеріїв їхнього досягнення.

2. Верифікацію й атестацію (валідацію) проміжних результатів ПЗ на етапах життєвого циклу та вимірювання ступеня відповідності певним показникам, які досягаються.

3. Тестування готового ПЗ зі збором даних про відмови, дефекти та інші помилки в системі.

4. Підбір моделей надійності для оцінювання надійності на підставі результатів тестування (дефекти, відмови і т. д.).

5. Оцінку показників якості, визначених у вимогах до розроблення ПЗ.

Якість ПЗ є відносним поняттям і набуває сенсу лише враховуючи реальні умови його використання. Таким чином, вимоги до якості визначаються відповідно до умов і конкретної сфери його застосування. Якість розглядається з трьох аспектів: якість програмного продукту, якість процесів життєвого циклу та якість супроводу чи впровадження (рис. 2.4).

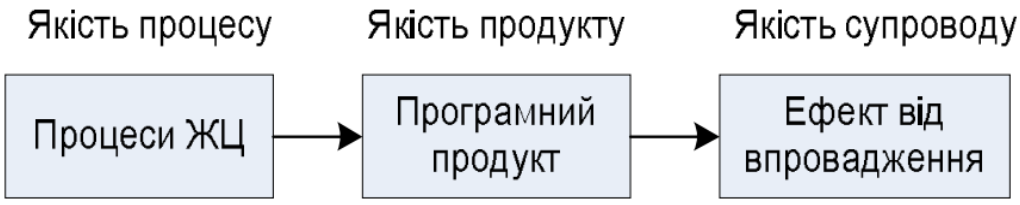

Рис. 2.4. Основні аспекти якості ПС

В аспекті, пов'язаному з процесами життєвого циклу (ЖЦ), визначається ступінь формалізації та вірогідність процесів ЖЦ, пов'язаних із розробкою програмного забезпечення, а також верифікація й валідація проміжних і кінцевих результатів цих процесів. Пошук і усунення помилок у готовому ПЗ виконуються за допомогою методів тестування, що призводить до зменшення кількості помилок і підвищення якості продукту.

Якість продукту досягається за допомогою процедур контролю проміжних продуктів під час процесів ЖЦ, перевірки їх на досягнення необхідної якості та методів супроводу продукту. Впровадження ПЗ суттєво залежить від знань обслуговуючого персоналу щодо функцій продукту та виконання відповідних правил. Згідно з ГОСТу 28195-89 модель якості програмного забезпечення включає три рівні представлення: фактор, критерій, метрика (рис. 2.5).

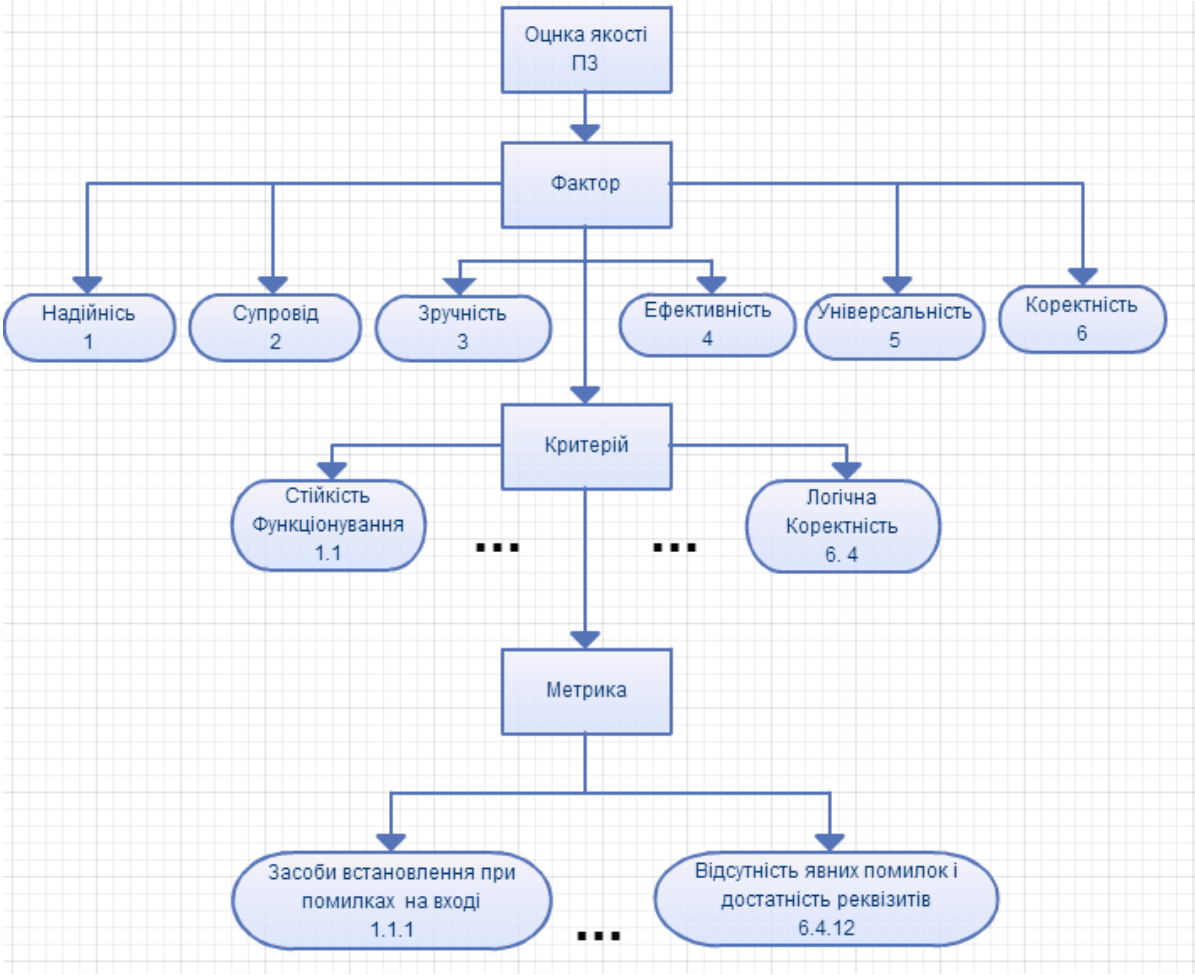

Рис. 2.5. Ієрархія показників якості згідно з ГОСТу 28195-89

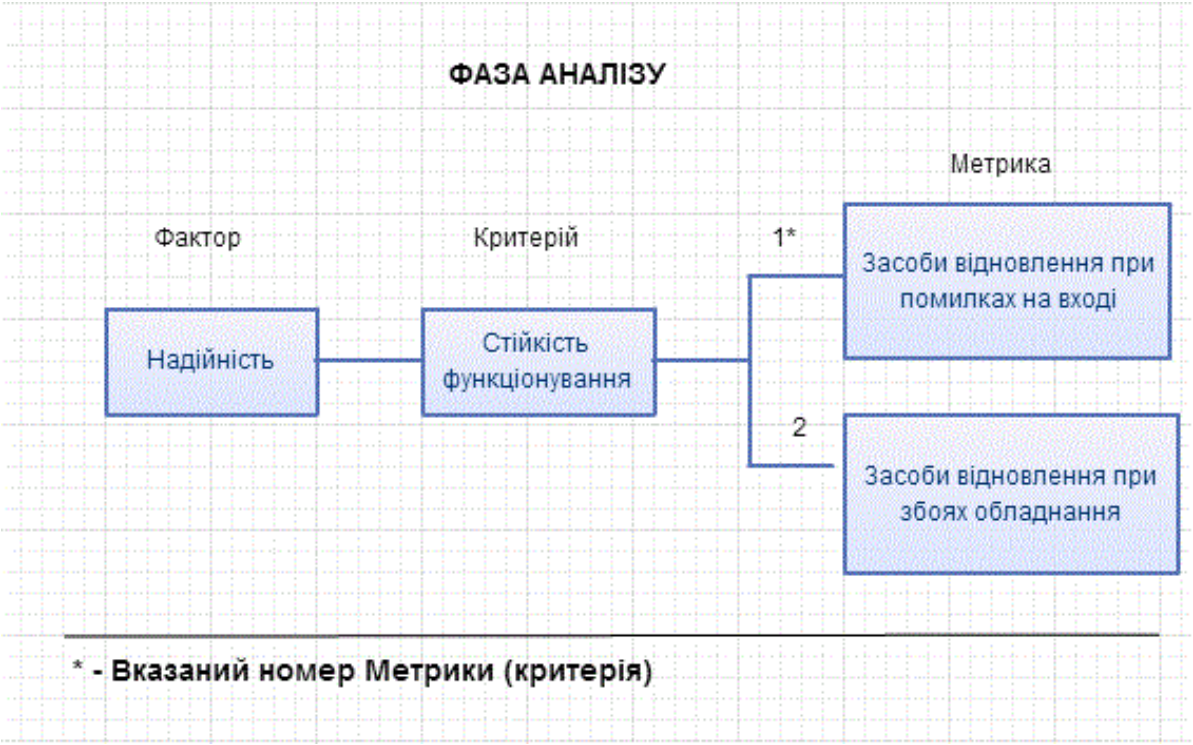

Рис. 2.6. Ієрархія Фази Аналізу

Перший рівень подання відповідає визначенню характеристик (факторів) якості ПЗ, кожна з яких відображає окреме уявлення користувача про якість. Відповідно до стандарту ГОСТ 28195-89 у модель якості входить шість характеристик або шість показників якості [32].

На рис. 2.5. – 2.6. зображена ієрархія стандарту для оцінки якості.

У табл. 2.1. – 2.6 наведено фактори якості та їх критерії та короткі характеристики по кожному з них [32].

Таблиця 2.1.

Показники універсальності

Найменування груп і показників якості	**Позначення показника**	**Характерна властивість**
5. Показники універсальності		Характеризують адаптованість ПЗ до нових функціональних вимог, які виникають внаслідок зміни області застосування або інших умов функціонування
5.1. Гнучкість	У1	Можливість використання ПЗ в різних областях застосування
5.2. Мобільність	У2	Можливість застосування ПЗ без істотних додаткових трудовитрат на комп'ютерах аналогічного класу
5.3. Модифікова-ність	У3	Забезпечення простоти внесення необхідних змін і доробок в програму в процесі експлуатації

Таблиця 2.2.

Показники надійності

Найменування груп і показників якості	**Позначення показника**	**Характерна властивість**
1. Показники властивості ПЗ		Характеризують здатність ПЗ в конкретних областях використання виконувати завдання функції в відповідності з програмними документами в умовах виникнення відхилень в середовищі функціонування, викликаних збоями технічних засобів, помилками в вхідних даних, помилками обслуговування та іншими дестабілізуючими впливами.
1.1. Стійкість функціонування	H1	Здатність забезпечувати продовження роботи програми після виникнення відхилень, викликаних збоями технічних засобів , помилками в вхідних даних і помилками обслуговування.
1.2. Працездат-ність	H2	Здатність програми функціонувати в заданих режимах і обсягах оброблюваної інформації відповідно з програмними документами за відсутності збоїв технічних засобів

Таблиця 2.3.

Показники супроводження

Найменування груп і показників якості	**Позначення показника**	**Характерна властивість**
2. Показники супроводу		Характеризують технологічні аспекти, що забезпечують простоту усунення помилок у програмі і програмних документах і підтримки ПЗ в актуальному стані
2.1 Структурованість	С1	Організація всіх взаємопов'язаних частин програми в єдине ціле з використанням логічних структур «послідовність», «вибір», «повторення»
2.2 Простота конструкції	С2	Побудова модульної структури програми найбільш раціональним з точки зору сприйняття і розуміння
2.3 Наочність	С3	Наявність і подання до найбільш легко сприйманого вигляду вихідних модулів ПЗ, повний їх опис у відповідних програмних документах
2.4 Повторюваність	С4	Ступінь використання типових проектних рішень або компонентів, що входять в ПЗ

Таблиця 2.4.

Показники зручності

Найменування груп і показників якості	**Позначення показника**	**Характерна властивість**
3. Показник зручності		Характеризують властивості ПЗ, що сприяють швидкому освоєнню, застосування та експлуатації ПЗ з мінімальними трудовитратами з урахуванням характеру вирішуваних завдань і вимог до кваліфікації обслуговуючого персоналу
3.1. Легкість засвоєння	З1	Представлення програмних документів і програм у вигляді, який сприяє розумінню логіки функціонування програми в цілому та її частин
3.2. Доступність експлуатаційних програмних документів	З2	Зрозумілість, наочність і повнота опису взаємодії користувача з програмою в експлуатаційних програмних документах
3.3 Зручність експлуатації та обслуговування	З3	Відповідність процесу обробки даних і форм представлення результатів характером вирішуваних завдань

Таблиця 2.5.

Показники ефективності

Найменування груп і показників якості	**Позначення показника**	**Характерна властивість**
4. Показник ефективності		Характеризують ступінь задоволення потреби користувача в обробці даних з урахуванням економічних, обчислювальних та людських ресурсів
4.1. Рівень автоматизації	E1	Рівень автоматизації функцій процесу обробки даних з урахуванням раціональності функціональної структури програми з точки зору взаємодії з нею користувача та використання обчислювальних ресурсів
4.2. Тимчасова ефективність	E2	Здатність програми виконувати задані дії в інтервал часу, що відповідає заданим вимогам
4.3. Ресурсомі-сткість	E3	Мінімально необхідні обчислювальні ресурси і число обслуговуючого персоналу для експлуатації ПЗ

Таблиця 2.6.

Показники коректності

Найменування груп і показників якості	Позначення показника	Характерна властивість
6. Показники коректності		Характеризують ступінь відповідності ПЗ вимогам встановленим в технічному завданні
6.1. Повнота реалізації	К1	Повнота реалізації заданих функцій ПЗ і достатність їх опису в програмній документації
6.2. Узгодженість	К2	Однозначний, несуперечливий опис і використання тотожних об'єктів, функцій, термінів, визначень, ідентифікаторів і т.д
6.3. Логічна коректність	К3	Функціональна та програмна відповідність процесу обробки даних при виконанні завдання загальносистемним вимогам

2.3. Оцінювання якості програмного забезпечення

Для досягнення поставленої мети при оцінюванні якості ПЗ потрібно виконати наступні основні завдання:

1. Розглянути особливості процесу оцінки якості програмного забезпечення, проаналізувати його як об'єкт стандартизації та визначити рівні представлення моделі якості ПЗ.

2. Визначити особливості використання метричного аналізу для визначення якості програмного забезпечення, виявивши причини його неефективного використання, а також різні інтерпретації значень цих показників.

3. Розробити програмне забезпечення для визначення якості ПЗ, що дозволить спрогнозувати подальшу ефективність процесу розробки та створити відповідний набір даних для визначення якості комплексного програмного показника.

4. Зробити відповідні висновки та надати рекомендації щодо практичного використання засобів визначення якості ПЗ за допомогою метричного аналізу.

Сучасні технології розробки ПЗ досягли такого рівня розвитку, що вимагають використання інженерних методів для оцінки результатів їх проектування на всіх етапах програмного проекту.

Рис. 2.7. Статичні технології оцінювання якості ПЗ

Важливо контролювати ступінь виконання запланованих показників якості та проводити метричний аналіз для управління ризиками. Оцінка та використання збірних компонентів можуть сприяти зниженню вартості нового проекту. Технічні методи оцінки якості програмного забезпечення

базуються на можливості їх вдосконалення через формулювання відповідних вимог до критеріїв оцінки якості та поліпшення моделей метричного аналізу та методів кількісного вимірювання на всіх етапах проекту. Статичні технології оцінки якості ПЗ представлені на рис. 2.7. і, одночасно, вони є динамічними у контексті прямого тестування.

ISO/IEC 25000 характеризує модель якості як чітко визначений набір функцій та їх взаємозв'язків, які формують основу специфікації. Вимоги до якості та оцінка якості визначаються у цьому стандарті.

У цьому стандарті використовується тестова модель метричних факторів, що включає:

1. Фактори – описують зовнішній вигляд програмного забезпечення з точки зору користувача.

2. Критерії – надають опис внутрішнього вигляду програмного забезпечення з точки зору розробника.

3. Показники – визначаються та використовуються для забезпечення методу вимірювання.

Метрики є функціями, де входи - це дані програми, а виходи – числові значення, які інтерпретуються як ступінь обробки програмним забезпеченням певних атрибутів, які впливають на його якість [33].

Кожному фактору призначаються критерії та показники, необхідні для вимірювання цих критеріїв. Початкові значення для функцій масштабування представляють собою числові значення, які додаються для визначення обсягу певних критеріїв.

Ці стандарти регламентують ієрархічну модель якості ПЗ з наступною структурою (рис. 2.8.)

Властивості якості представляють собою невід'ємні атрибути програмного забезпечення, які сприяють підвищенню його якості. Характеристики якості можуть бути розподілені на одну або кілька загальних характеристик або часткових властивостей.

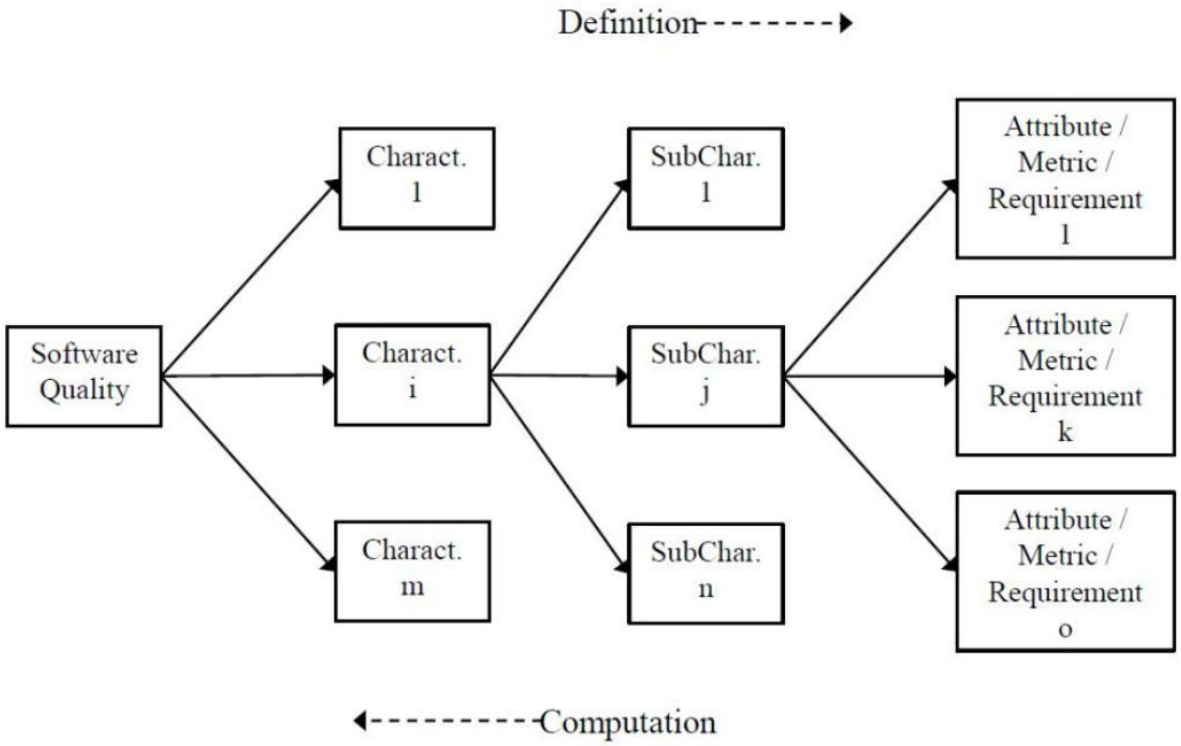

Рис. 2.8. Структура моделі оцінки якості ПЗ

Еталонна модель вимірювання якості програмного забезпечення зображена на рис. 2.9 визначає взаємозв'язок між моделлю якості та пов'язаними програмними функціями, підфункціями та атрибутами, які визначають вимірювання якості. Це виражено через функції вимірювання, елементи показників якості та методи вимірювання.

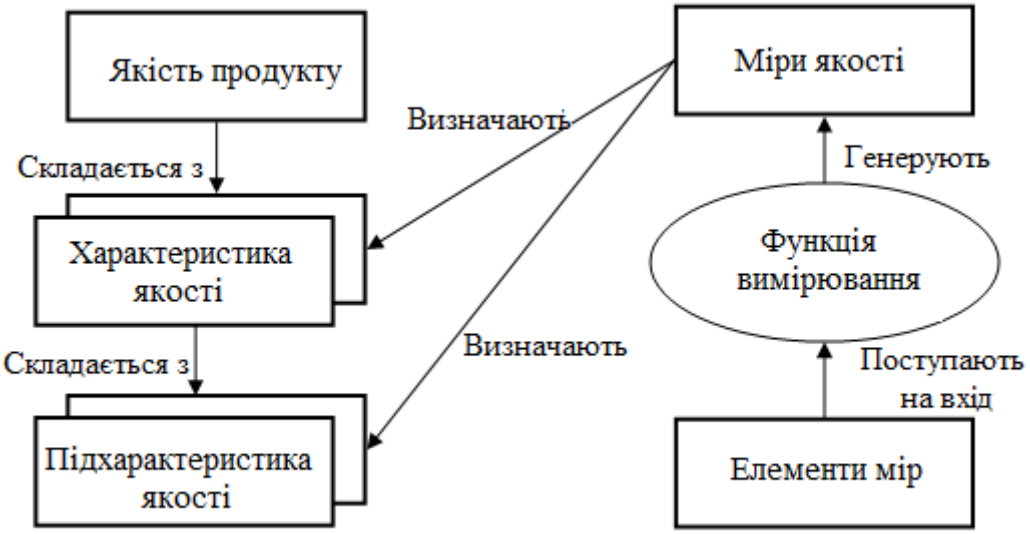

Рис. 2.9. Еталонна модель якості ПЗ

Функція вимірювання є алгоритмом, який використовується для поєднання елементів показників якості. Елементи одиниці – це одиниці вимірювання, які виникають в результаті математичних функцій або є вхідними значеннями для одиниць вимірювання. Показник якості визначає характеристики об'єкта та методи його вимірювання.

Метод вимірювання представляє собою послідовність операцій, що використовуються для кількісної оцінки значення певної властивості на визначеній шкалі. Процес вимірювання реалізується через застосування конкретного методу вимірювання. Результат, отриманий за допомогою цього методу, входить у загальний обсяг вимірювання якості [33].

Функція вимірювання дозволяє кількісно оцінити якісні та особливі характеристики. Окремі характеристики якості потім оцінюються на основі індивідуальних ознак, а оцінки цих ознак використовуються для комплексної оцінки якості. Для вимірювання якісних та особливих характеристик може використовуватися більше однієї міри.

2.4. Метрики якості програмного коду

Метрика – це кількісний масштаб і метод, який може використовуватися для вимірювання [34]. Введення і використання метрик необхідно для поліпшення контролю над процесом розробки, зокрема над процесом тестування [35].

Метрики складності програм прийнято розділяти на 3 основні групи:

1. Метрики розміру програм – базуються на визначенні кількісних характеристик, пов'язаних з розміром програми, і відрізняються відносною простотою. Ця група метрик орієнтована на аналіз вихідного тексту програм, тому вони можуть використовуватись для оцінки складності проміжних продуктів розробки. Серед відомих метрик цієї групи можна виділити кількість операторів програми, кількість рядків вихідного тексту та набір метрик Холстеда [34–35].

2. Метрики складності потоку управління програм – ґрунтуються на аналізі графів управління програмою і також можуть бути використані для

оцінки складності проміжних продуктів розробки. У цю групу входить метрика МакКейба [35].

3. Метрики складності потоків даних програм – ґрунтуються на оцінці використання, конфігурації та розташування даних у програмі, особливо для глобальних змінних. Сюди належать метрики Чепіна [35].

Метрики, орієнтовані на розмір (індикатори оцінки обсягу), безпосередньо вимірюють програмний продукт. Ці метрики ґрунтуються на рядках LOC (Lines Of Code). Кількість рядків вихідного коду (LOC) є найпростішим та найпоширенішим способом оцінити робоче навантаження проекту. Важливо враховувати, що ці метрики не є універсальними, особливо для LOC, який сильно залежить від мови програмування.

Є два основних показники LOC:

1. Кількість «фізичних» рядків коду (LOC, SLOC, KLOC, KSLOC, DSLOC) – сума рядків вихідного коду, включаючи коментарі та порожні рядки.

2. Кількість «логічних» рядків коду (LSI, DSI, KDSI, де SI вихідні команди) – кількість команд, які залежать від мови програмування. Ці метрики йдуть в парі, де кількість «логічних» LOC дорівнює кількості «фізичних», за винятком порожніх рядків та рядків з коментарями.

Метрики Холстеда – це показники, розраховані на основі аналізу кількості рядків та синтаксичних елементів вихідного коду програми. Метрика Холстеда ґрунтується на чотирьох вимірюваних характеристиках програми:

- NUOprtr – кількість унікальних операторів);
- NUOprnd – кількість унікальних операндів;
- NOprtr – загальна кількість операторів;
- NOpnd – загальна кількість операндів [35].

Виходячи з цих характеристик, розраховуються наступні оцінки:

- HPVoc – словник програми: $HPVoc = NUOprtr + NUOprnd$;
- HPLen – довжина програми: $HPLen = Noprtr + Noprnd$;
- HPVol – обсяг програми: $HPVol = HPLen \log2 HPVoc$.

Далі обчислюється оцінка складності програми (Halstead Difficulty, HDiff) [35]:

$$HDiff = (NUOprtr/2) \times (NOprnd/NUOprnd).$$

Показник цикломатичної складності Маккейба є одним з широко використовуваних показників для оцінки складності програмних проектів. Цей показник вказує на кількість потрібних проходів для покриття всіх контурів сильнозв'язного графа або кількість тестових прогонів програми для повного тестування за принципом «працює кожна гілка програми». Цей показник розраховується для різних структурних одиниць програми, таких як модуль чи метод.

Показник цикломатичної складності не лише дозволяє оцінювати трудомісткість реалізації окремих елементів програмного проекту і коригувати загальні показники тривалості та вартості проекту, але також надає можливість оцінити пов'язані ризики і приймати відповідні управлінські рішення [35].

Існує кілька модифікацій метрик Чепіна, проте простий та ефективний метод оцінки полягає у визначенні інформаційної міцності окремого програмного модуля за допомогою аналізу характеру використання змінних зі списку введення-виведення. Змінні розбиваються на чотири функціональні групи: множина «P» – змінні для введення та виведення; множина «M» – модифіковані або створювані всередині програми змінні; множина «C» – змінні, які використовуються для управління роботою програмного модуля; множина «T» – не використовувані (паразитні) змінні.

Оцінка якості за метрикою Чепіна обчислюється за формулою

$$Q = P + 2* M + 3*C + 0.5* T .$$

Метрика Альбрехта. Метрика дефектів якості програмних засобів. Існує два різновиди цієї метрики:

- Заснована на рядках коду;
- Заснована на функціональних покажчиках.

Метрика, заснована на функціональних покажчиках використовується для непрямої оцінки рівня якості процедурно-орієнтованого ПЗ. Перевага цієї метрики – легкість обчислення. Недолік – результати ґрунтуються на суб'єктивних даних.

Замість прямих вимірювань використовуються не прямі.

Формула метрики:

$$DQ = \frac{\text{кількість дефектів}}{FP}$$

де FP – функціональні показчики

$$FP = F * (0{,}65 + 0{,}001 * \sum_{i=1}^{14} k_i)$$

де F – загальна кількість функціональних покажчиків

$$F = \sum_{i=1}^{5} f_i$$

Значення коефіцієнтів регулювання складності k залежать від відповідей на 14 запитань, що стосуються впливу певних факторів на виконання функцій програмного забезпечення, на які відповідає людина. Саме через суб’єктивність значення даного виду метрики, була обрана метрика, заснована на рядках коду.

У цьому випадку

$$DQ = \frac{\text{кількість дефектів}}{SLOC}$$

де SLOC – кількість рядків коду,

$$SLOC = \sum_{i=1}^{N} S_i$$

де *N* – кількість структурних блоків у коді, *Si* – обсяг *i*- го блоку.

Для оцінки складності програми використовуються й інші метрики [33]:

− метрики Джилба – кількість операторів циклу, кількість операторів умови, кількість модулів або підсистем, відношення кількості зв’язків між модулями до кількості модулів;

− метрика Шнадевида – кількість шляхів в графі керування;

− метрика Майєрса – інтервальна міра;

− метрика Хансена – пара (цикломатична кількість, кількість операторів);

− метрика Чена – топологічна міра;

− метрика Вудворда – кількість вузлів передач управління;

− метрика Кулика – кількість найпростіших циклів;

− метрика Хура – цикломатична кількість мереж Петрі;

− метрики Вітворфа, Зулевського – міра складності потоку керування, міра складності потоку даних;

− метрика Петерсона – кількість багатовходових циклів;

− метрики Харрісона, Мейджела – функційна кількість, функційне відношення, регулярні вирази;

− метрика Пивоварського – модифікована цкломатична міра складності;

− метрика Пратта – тестуюча міра;

− метрика Кантоне – характеристичні числа поліномів графу програми;

− метрика Мак-Клура – міра складності, заснована на кількості можливих шляхів виконання програми, кількості керуючих конструкцій та змінних;

− метрика Кафура – міра на основі концепції інформаційних потоків;

− метрика Схуттса, Моханті – ентропійні міри;

− метрика Коллофело – міра логічної стабільності програм;

− метрика Зольновського, Сімонса, Тейєра – зважена сума різних індикаторів;

− метрика Берлінгера – інформаційна міра;

− метрика Шумана – складність з позиції статистичної теорії мови;

− метрика Янгера – логічна складність з врахуванням історії обчислень;

− метрика Тая – покращення метрики Маккейба;

− метрика Кокола – комплексна метрика, заснована на більш простих;

− метрики зв’зності (зчеплення) – ступінь залежності кожного модуля від кожного з інших модулів за даними, за зразком, за управлінням, за зовнішніми посиланнями, за загальною областю, за змістом (кількісний показник – ступінь зчеплення) [33].

2.5. Методи визначення показників якості програмного забезпечення

Методи визначення показників якості програмного забезпечення розрізняються:

− за способами отримання інформації про ПЗ (вимірювальний, реєстраційний, органолептичний, розрахунковий);

− за джерелами отримання інформації (експертний, соціологічний) [36].

Вимірювальні методи базуються на отриманні інформації про властивості та характеристики програмного забезпечення за допомогою інструментальних засобів. Наприклад, ці методи включають визначення обсягу ПЗ, такого як кількість рядків вихідного тексту програми і кількість коментарів, а також різні інші показники, такі як кількість операторів і операндів, виконуваних операторів, кількість гілок в програмі, час реакції і інші.

Реєстраційні методи полягають в отриманні інформації під час досліджень чи функціонування ПЗ, коли фіксуються та підраховуються певні події, такі як час і кількість збоїв і відмов, час передачі управління іншим модулям, час початку та закінчення роботи.

Органолептичні методи використовують інформацію, яка отримана в результаті аналізу сприйняття органів відчуття, таких як зір і слух, і

застосовується для визначення таких показників, як зручність використання та ефективність.

Розрахункові методи ґрунтуються на використанні теоретичних і емпіричних залежностей (на ранніх етапах розробки), статистичних даних, що накопичені під час досліджень, експлуатації та супроводження ПЗ. Ці методи визначають такі параметри, як тривалість і точність розрахунків, час реакції та необхідні ресурси.

Експертні методи використовується для визначення значень показників якості ПЗ групою експертів-спеціалістів, компетентних у вирішенні даної задачі. Ці методи базується на їхньому досвіді та інтуїції і використовується в тих випадках, коли інші методи не ефективні або трудомісткі. Експертні методи рекомендуються застосовувати при визначенні показників наочності, повноти та доступності програмної документації, легкості освоєння та структурності ПЗ.

Соціологічні методи основані на обробці спеціальних анкет-опитувальників.

Найбільш розповсюдженими методами оцінки експертами значень атрибутів є пряма оцінка, ранжування та попарне порівняння [36]. Ефективність використання конкретного методу визначається характером аналізованої інформації. В даній роботі буде використовуватися метод ранжирування.

Перевага ранжування як методу експертної оцінки полягає в простоті процесу оцінювання. У процесі ранжування експерти повинні встановити взаємозв'язок між усіма об'єктами і розглядати їх як єдине ціле.

У цьому методі вся група об'єктів подається експерту для ранжування, і пропонується вказати найважливіший об'єкт серед них. Цей об'єкт виключається з подальшого розгляду, оскільки його ранг вважається визначеним, і після цього з решти обирається об'єкт з найвищим пріоритетом. Порівняння всіх об'єктів створює впорядковану послідовність $a_1 > a_2 > ... > a_N$, , де об'єкт з першим номером має найвищий пріоритет, другий – менший пріоритет, ніж перший, але важливіший за інші (рис. 2.10.).

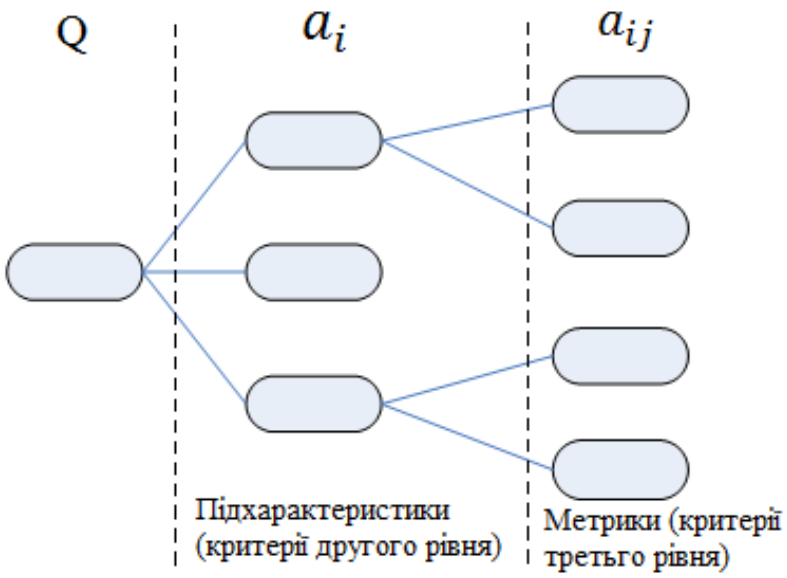

Рис. 2.10. Ієрархія атрибутів

У випадку неможливості визначення пріоритетність між двома чи більше атрибутами, використовується однаковий ранг, чиє значення є середнім арифметичним суми позначених місць за допомогою цих атрибутів.

Вагові коефіцієнти критеріїв можуть бути визначені на основі рейтингу, сформованого експертами відповідно до процесу ранжування. Припустимо, що думки експертів є узгодженими, оскільки детальний аналіз даної теми виходить за рамки даної роботи.

За допомогою підсумкового ранжування можна використовувати шкалу Фішберна для оцінки значущості кожного критерію нефункціональних вимог [36].

$$p_i = \frac{2(n-i+1)}{n(n+1)},$$

де p_i – коефіцієнт значимості i-го критерію;

i – ранг поточного критерію в підсумковому ранжируванні;

n – кількість критеріїв.

Для вирішення задач отримання оцінки нефункціональних вимог до програмного забезпечення, необхідно розглядати її як адитивну функцію корисності та застосовувати адитивну згортку [36]. Загальний вигляд

адитивної функції корисності можна виразити наступною узагальненою формулою:

$$Q(x) = \sum_{i=1}^{n} p_i \cdot \hat{Q}_i(x),$$

де Q – функція оцінки варіанта x;

p_i – вага критерію i;

$\hat{Q}_i(x)$ – оцінка варіанта x за критерієм i.

Тоді модель оцінки нефункціональних вимог можна описати функцією адитивної скалярної згортки:

$$Q(x) = \sum_{i=1}^{n} p_i a_i(x) = \sum_{i=1}^{n} p_i \sum_{j=1}^{m} q_j a_{ij}(x),$$

де $Q(x)$ – загальний критерій для оцінок користувачів $x \in X$;

$\{a_i(x)\}_1^n$, $\{a_{ij}(x)\}_1^m$ – набори складових атрибутів відповідних рівнів ієрархії;

n, m – кількість атрибутів на рівнях;

p_i, q_j - вага складових атрибутів a_i, a_{ij}.

Для важливості ваг виконується умова нормування:

$$\sum_{i=1}^{n} p_i = \sum_{j=1}^{m} q_j = 1.$$

Для атрибутів на всіх рівнях використовується єдина шкала оцінки від 0 до 1 [36].

ВИСНОВОК ДО РОЗДІЛУ 2

У розділі розглянуті основні моделі якості, які використовуються для оцінки програмного забезпечення, здійснено порівняльний аналіз цих моделей. Представлений всебічний аналіз допомагає зрозуміти, чому важливо використовувати певні характеристики та параметри якості програмного забезпечення при їх оцінці. Важливо відзначити, що для комплексної оцінки якості можна одночасно використовувати різні моделі.

РОЗДІЛ 3
ВИМОГИ ДО ПРОГРАМНОГО ЗАСОБУ

3.1. Функціональні вимоги до розроблюваного програмного засобу

Функціональна вимога представляє собою опис того, що програмне забезпечення повинно робити. Або іншими словами це опис самого ПЗ або його компонентів, включаючи вхідні дані, поведінку та вихідні дані. Функція може виявлятися у формі обчислень, обробки даних, бізнес-процесів, взаємодії з користувачем або будь-якої іншої конкретної функціональності, що визначає, які завдання система може виконувати. В програмній інженерії функціональні вимоги також відомі як функціональні специфікації.

При розробці програмного забезпечення функціональні вимоги можуть бути висловлені як абстрактними високорівневими твердженнями так і деталізованими математичними функціональними вимогами. Вимоги до функціонального програмного забезпечення допомагають узяти до уваги очікувану поведінку системи.

Функціональні вимоги не слід плутати з іншими типами вимог при розробці ПЗ, таких як:

1. Бізнес-вимоги – описують високорівневі бізнес-потреби, такі як збільшення частки ринку, зменшення відтоку клієнтів або підвищення цінності клієнтів.

2. Вимоги користувачів – охоплюють різноманітні цілі які користувачі можуть досягти за допомогою продукту. Зазвичай документується у вигляді історій користувачів, способів використання та сценаріїв.

Вимоги до ПЗ описують, як система повинна працювати, щоб задовольняти бізнес-вимогам та вимогам користувачів, і включають як функціональні, так і нефункціональні вимоги.

В контексті розроблюваного програмного засобу можна почати з функціональних вимог до інтерфейсу засобу. Інтерфейс повинен бути зручним, простим і зрозумілим, щоб дозволити користувачеві ефективно виконувати потрібні завдання. Всі елементи розроблюваного програмного засобу повинні відповідати стандартам юзабіліті. Система має включати довідку для користувача та надавати повідомлення про некоректні дії.

Вхідними даними засобу є інформація, яку користувач вводить у систему у вигляді тексту програмного коду, і запити на виконання конкретних функцій (наприклад, обчислення метрик, побудова діаграм, перегляд статистики, тощо).

Опис функціональних можливостей та вимог розроблюваного програмного засобу, який призначений для автоматизації обчислення якості програмного коду, представлений у вигляді діаграм варіантів використання (рис. 3.1.)

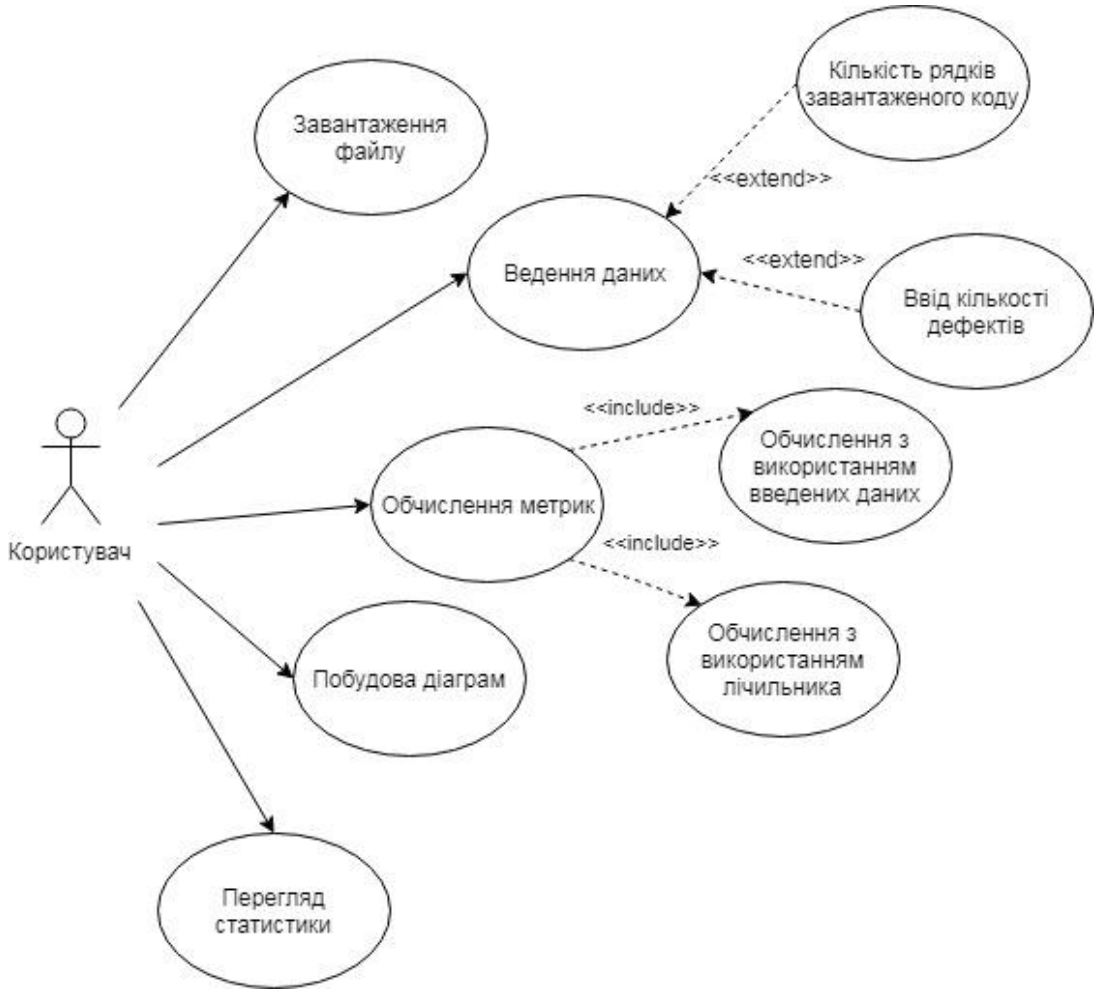

Рис. 3.1. Use case діаграма застосунку

3.2. Загальний опис нефункціональних вимог до програмного засобу

Нефункціональні вимоги визначають характеристики та обмеження експлуатації системи, які покращують її функціональність, швидкодію, безпеку та надійність. Раніше розглядалися різні типи вимог до програмного забезпечення, але зараз акцент буде зроблено на нефункціональних вимогах та способах їх документування.

Термінологія для нефункціональних вимог може варіюватися залежно від джерела, проте найбільш поширений наразі, та найбільш часто використовується саме термін «нефункціональні вимоги» (NFR) викладений у BABOK, ключовому джерелі для бізнес-аналітиків. У ньому нефункціональні вимоги описують саме якість експлуатації, а не поведінку продукту.

Продуктивність визначає, наскільки швидко програмна система або окрема її частина реагує на дії конкретних користувачів при певному робочому навантаженні. У більшості випадків ця характеристика пояснює, як довго користувач повинен очікувати виконання певної операції (наприклад, завантаження сторінки або обробка транзакції), враховуючи загальну кількість користувачів на той момент. Однак це правило не завжди вірне. Вимоги до продуктивності також можуть включати фонові процеси, які залишаються непомітними для користувача, такі як резервне копіювання.

Масштабованість визначає, як ефективно система може впоратися з найвищим рівнем робочого навантаження, при якому вона все ще відповідає вимогам щодо продуктивності.

Переносність визначає, наскільки система чи її компоненти можуть бути запущені у різних середовищах. Зазвичай вона включає в себе специфікації обладнання, програмного забезпечення та інших використовуваних платформ. Переносність визначає, наскільки ефективно операції або дії, які виконуються на одній платформі, можуть бути відтворені на іншій. Також враховується доступність елементів системи та можливість їх взаємодії у двох різних середовищах. Додатковий аспект переносності,

відомий як сумісність, оцінює можливість системи співіснувати з іншими системами у тому ж самому середовищі, наприклад, якщо програмне забезпечення має бути сумісним із брандмауером чи антивірусом.

Оцінка переносності і сумісності вирішується з урахуванням операційних систем, апаратних пристроїв, браузерів, програмного забезпечення та їх версій. Сучасні стандарти для веб-додатків розглядають рішення для крос-платформних, мобільних платформ та перехресного перегляду.

Нефункціональні вимоги щодо переносності зазвичай базуються на ринкових та користувацьких дослідженнях та аналітичних звітах щодо видів програмного забезпечення та пристроїв, які використовують користувачі. У корпоративному середовищі, де доступ до програмного забезпечення регламентується списками документованих пристроїв і операційних систем, визначення сумісності і переносності здійснюється досить легко.

Надійність – це міра ймовірності того, що система або її компоненти будуть працювати без збоїв протягом певного періоду за попередньо визначеними умовами. Вона зазвичай виражається у відсотках ймовірності. Наприклад, якщо система має 85% надійності протягом місяця, це означає, що за нормальних умов експлуатації є 85% ймовірності того, що система не вийде з ладу. Вимірювання надійності можна провести різними способами, такими як підрахунок кількості критичних помилок протягом певного часу або обчислення середнього часу до відмови. Існує три способи вимірювання надійності: у відсотках ймовірності за певний час; у кількості критичних відмов за певний час; у середньому часі між збоями.

Супроводжуваність – визначає час, який необхідний для виправлення цілого рішення або його окремої компоненти, час для зміни та підвищення продуктивності тих чи інших характеристик ПЗ або час для його адаптації до іншого середовища. Подібно до надійності, супроводжуваність можна виразити як ймовірність проведення доробок протягом певного часу. Наприклад, якщо система має 75% супроводжуваності протягом 24 годин, це

означає, що є 75% ймовірність виправлення компоненти системи протягом 24 годин.

Доступність – визначає, наскільки ймовірно те, що система буде доступною для користувача у певний момент часу. Хоча її можна виразити у відсотках ймовірності, доступність також можна визначити як відсоток часу, протягом якого система буде готовою до роботи у певний період. Наприклад, система може бути доступною 98% часу протягом місяця. Можна сказати, що доступність є найважливішою вимогою для бізнесу, тому що для її визначення необхідно мати оцінки щодо надійності та супроводжуваності.

3.3. Діаграма послідовності

Діаграми послідовностей в UML вважаються популярним інструментом для динамічного моделювання, оскільки вони спеціально акцентують увагу на лініях життя ПЗ, його процесах та об'єктах, які існують одночасно, і повідомленнях, якими обмінюються процеси при виконанні певних функцій.

Діаграму послідовностей можна будувати як для усього процесу, так і для його окремих функцій. На рисунку 3.2 зображено діаграму, що графічно представляє виконання обчислення метрик програмного коду, розташовану на вкладці «Metrics»: запуск програмного коду, завантаження файлу з програмним кодом та обчислення метрик з подальшим представленням результатів у числовій та графічній формі.

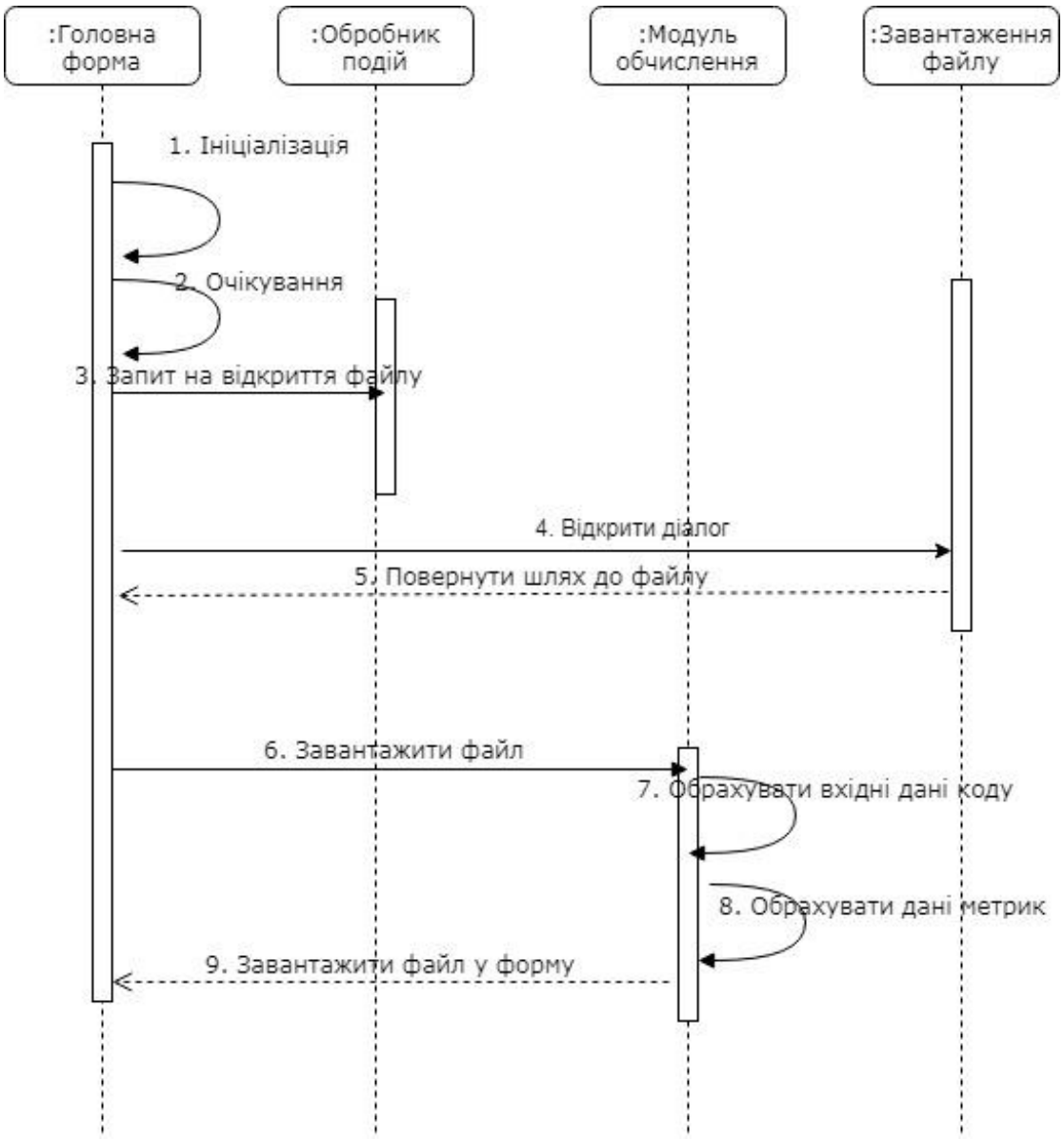

Рис. 3.2. Діаграма послідовностей виконання обрахунку метрик програмного коду

На рис 3.3. наведено відповідну діаграму для вкладки «Diagram»: завантаження даних, графічне зображення за допомогою стовпчастих діаграм.

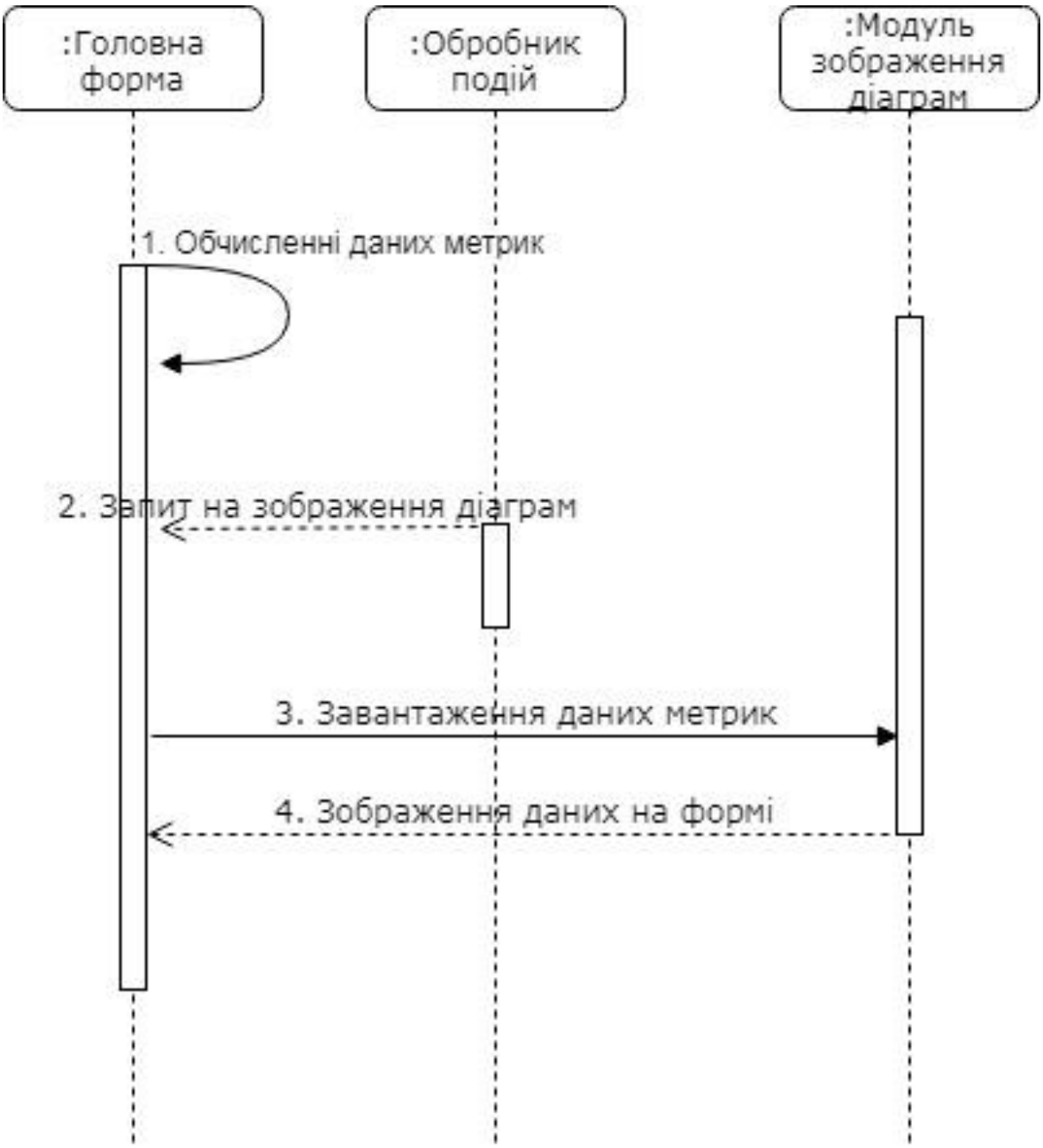

Рис. 3.3. Діаграма послідовностей виконання для відображення діаграм

3.4. Системні вимоги до розроблюваного програмного засобу

Мінімальні системні вимоги:

- Операційна система: Windows 7 Service Pack 1;
- Процесор: Pentium 4 1.5 GHz or Athlon XP 1500+;
- Оперативна пам'ять: 512 MB;
- Місце на жорсткому диску: 1 GB;
- Графічний адаптер: з підтримкою DirectX 9.0c;
- Передустановлення: .NET Framework 4.7.

Рекомендовані системні вимоги:

- Операційна система: Windows 10;
- Процесор: Core 2 Duo або Athlon X2 2.4 GHz;
- Оперативна пам'ять: 2 GB;
- Місце на жорсткому диску: 1 GB;
- Графічний адаптер: з підтримкою DirectX 10;
- Передустановлення: .NET Framework 4.9.

ВИСНОВКИ ДО РОЗДІЛУ 3

В даному розділі були розглянуті функціональні та нефункціональні вимоги для розроблюваного програмного засобу оцінювання якості програмного коду, наведені діаграми послідовностей окремих його функцій, надані мінімальні та рекомендовані системні вимоги.

РОЗДІЛ 4
РОЗРОБКА ПРОГРАМНОГО ЗАСОБУ

4.1. Інструментальні засоби розробки засобу

Для створення програмного засобу було використане середовище розробки Visual Studio 2019, платформу .NET Framework 4.8, мову програмування C#, СУБД Microsoft SQL Server 2016.

Вибір .NET Framework зумовлений тим, що ця програмна технологія, надана Microsoft як платформа для створення як звичайних застосунків, так і вебзастосунків. Багато в чому ця технологія продовжила ідеї і принципипи технології Java. Однією з ідей .NET Framework є сумісність служб, написаних різними мовами. Хоча ця функція рекламується Microsoft як перевага .NET, платформа Java пропонує ті самі функції [37].

Кожна бібліотека (колекція) у .NET має свою власну версію, яку можна використовувати для вирішення можливих конфліктів між різними колекціями версій [37].

.NET є кросплатформною технологією, на даний момент в DotGNU існують варіанти технології Microsoft Windows, FreeBSD (від Microsoft), Mono-Projekt (згідно з угодою між Microsoft і Novell) і Linux.

Захист авторських прав стосується створення середовища виконання (CLR) для програм .NET. Багато компаній можуть безкоштовно надати компілятор .NET для різних мов [37].

.NET ділиться на дві основні частини: середовище виконання (по суті, віртуальна машина) та засоби розробки.

Середовище розробки .NET: Visual Studio .NET (C++, C#, J#), SharpDevelop, Borland Developer Studio (Delphi, C#) тощо.

Середовище Eclipse має додаток для розробки програм .NET. Також можна використовувати консольний компілятор для розробки програм у текстовому редакторі.

Так само, як і у технології Java, середовище розробки .NET створює байт-код для виконання віртуальною машиною. Мова, яку використовує ця віртуальна машина в .NET, отримала назву CIL (Common Intermediate Language), також відому як MSIL (Microsoft Intermediate Language) або просто IL. Використання байт-коду дозволяє працювати між платформами на рівні скомпільованого проекту (.NET: лише асамблея), а не на рівні вихідного тексту (наприклад, на C). Байт-код часу виконання (CLR) вбудовується в середовище компіляції «точно вчасно» (JIT) в механізмі процесора Codicil перед виконанням збірки. Важливо відзначити, що один із перших компіляторів Java JIT також був розроблений Microsoft (в наш час Java використовує більш розвинену багаторівневу компіляцію – Sun HotSpot). Сучасна технологія динамічної компіляції дозволяє досягти рівня продуктивності, схожого на рівень продуктивності традиційних «статичних» компіляторів (наприклад, C++), а проблеми з продуктивністю зазвичай пов'язані лише з якістю компілятора [37].

.NET Framework є додатковим компонентом операційної системи і може бути будь-якою версією Windows.

Він містить:

− мови програмування: C#, VB.NET, Managed C++ і JScript.NET;

− об'єктно-орієнтоване середовище CLR (Common Language Runtime), яке спільно використовується мовою програмування для створення Windows та Інтернет-додатків;

− кілька взаємопов'язаних бібліотек класів під загальною назвою FCL (Framework Class Library) [38].

З концептуальної точки зору взаємозв’язок між цими архітектурними компонентами .NET Framework показаний на рис. 4.1.

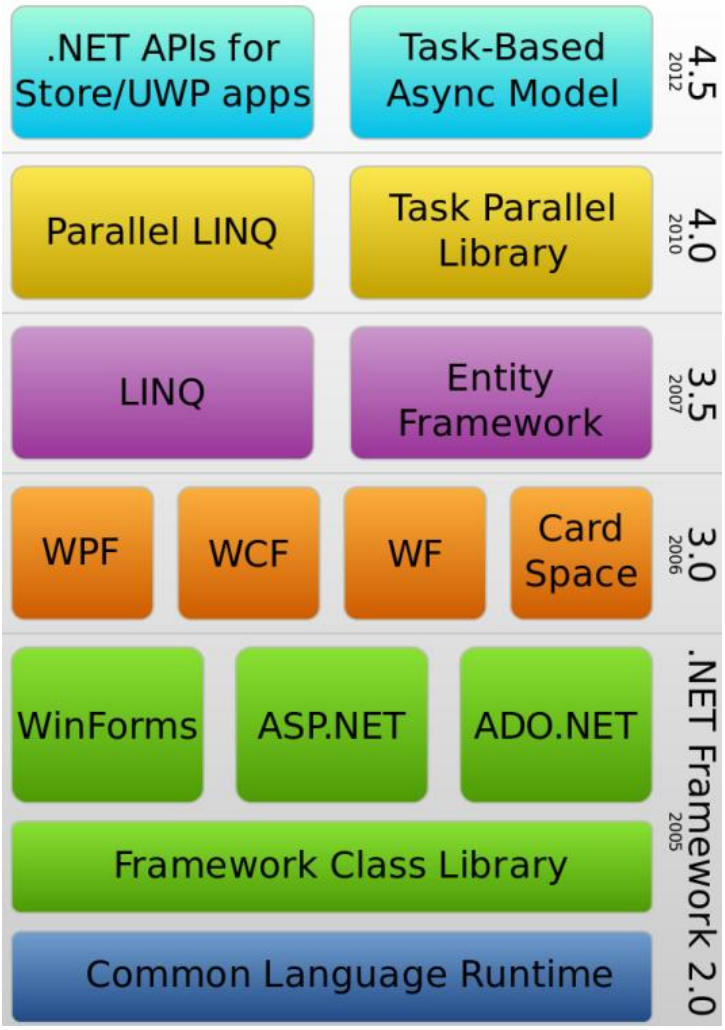

Рис. 4.1. Архітектура .NET Framework

Найбільш важливим компонентом .NET Framework є середовище CLR, яке забезпечує середовище для роботи програми. Його основна функція – виявляти та завантажувати типи .NET та керувати ними відповідно до отриманих команд.

CTS (Common Type System) – це стандартна система типів, вбудована у CLR. Вона визначає всі типи даних, що підтримуються середовищем виконання.

CLS (Common Language Specification) – специфікація загальної мови, яка визначає підмножину загальних типів та структур програмного забезпечення, які можуть бути зрозумілі всіма мовами програмування [38].

Вище рівня CLR знаходиться ряд базових класів платформи (бібліотека класів FCL), над ним рівень класів даних і XML, а також шар для створення веб-служб, веб-додатків і додатків Windows (веб-форм і форм Windows).

Ряд базових класів платформи низькорівневого FCL не лише приховують звичайні низькорівневі операції, такі як файлове введення-виведення, обробка графіки та взаємодія з комп'ютерним обладнанням, але також надають безліч сервісів безпеки, що підтримують їх у сучасних додатках. Управління, підтримка мережного зв'язку, управління обчислювальним потоком, використання зіставлення та збору тощо [38].

Вище цього рівня знаходиться рівень класу, який розширює базовий клас для забезпечення керування даними та XML. Для управління інформацією можна використовувати класи даних, які зберігається в базі даних сервера. Ці класи включають класи мови структурованих запитів (SQL), які дозволяють програмістам отримати доступ до довгострокових сховищ даних за допомогою стандартних інтерфейсів SQL. Можна також використовувати набір класів ADO.NET для управління постійними даними. Платформа .NET Framework також підтримує безліч класів, які можна використовувати для керування даними XML, а також їх пошуком та перетворенням [38].

4.2 Експлуатація розробленого програмного забезпечення

Розроблений програмний засіб DAnalyz (аналізатор дефектів) призначений для оцінки якості програмного коду. Засіб дозволить автоматизуючи процес тестування, спростити моніторинг помилок програмного коду та скоротити час, необхідний для розрахунку показників.

Засіб запускається за допомогою виконуваного файлу DAnalyz.exe.

Після запуску програми, на екрані з'являється головне вікно програми (рис. 4.2).

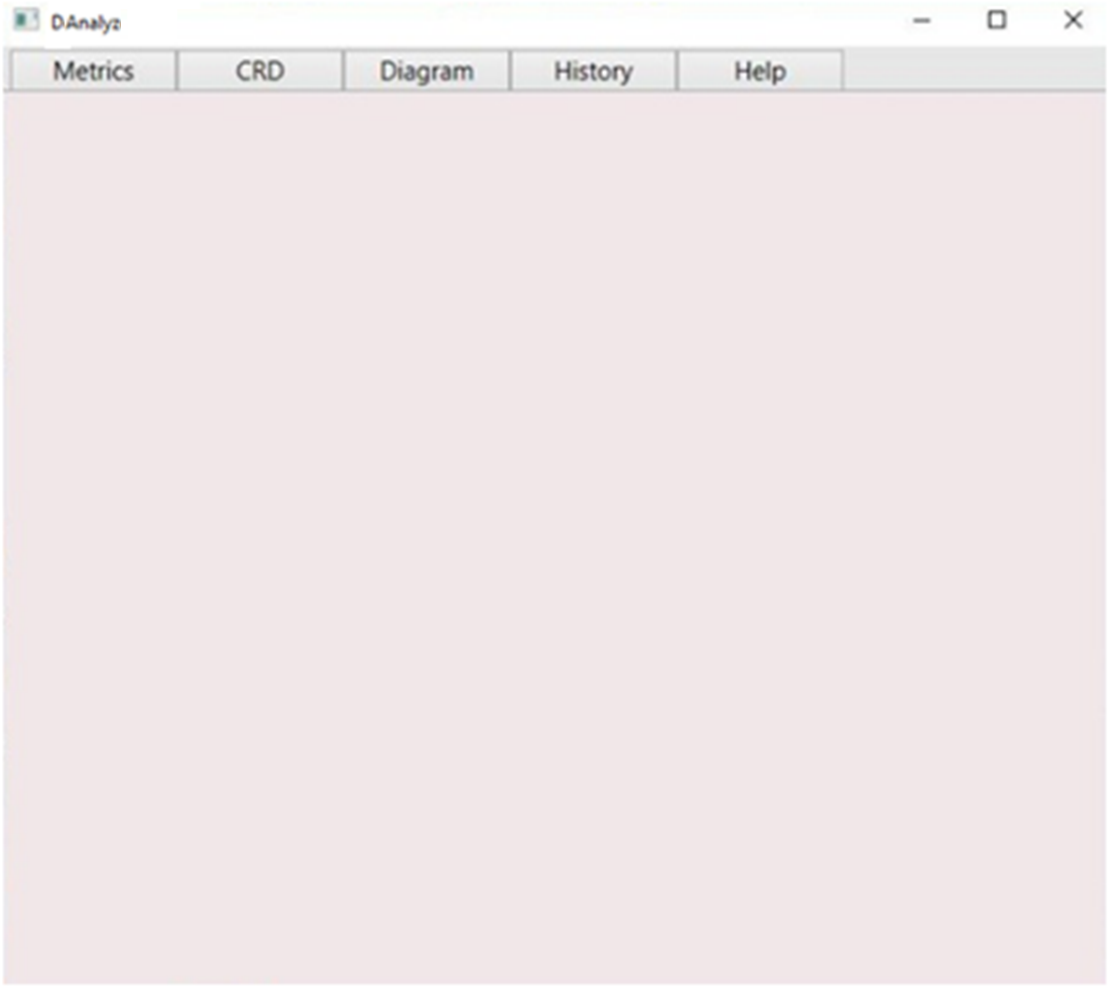

Рис. 4.2. Головне вікно програми

На головному вікні користувачу відкривається можливість натиснути і подивитися п’ять вкладок: «Індикатори», «CRD», «Графіки», «Історія» та «Довідка».

Натиснувши першу вкладку «Метрики», відкриється вікно з параметрами по яким будуть шукатися задані метрики, та кнопка «Вибору файлу», натиснувши на яку, відкриється вікно пошуку потрібного файлу (рис. 4.3).

Знайшовши потрібний файл з кодом, можна завантажити його у програму для подальшого аналізу. У верхній частині вікна також буде відображатися шлях до файлу, ліворуч текстовий блок з усіма виділеними файлами, а праворуч – код (рис. 4.4).

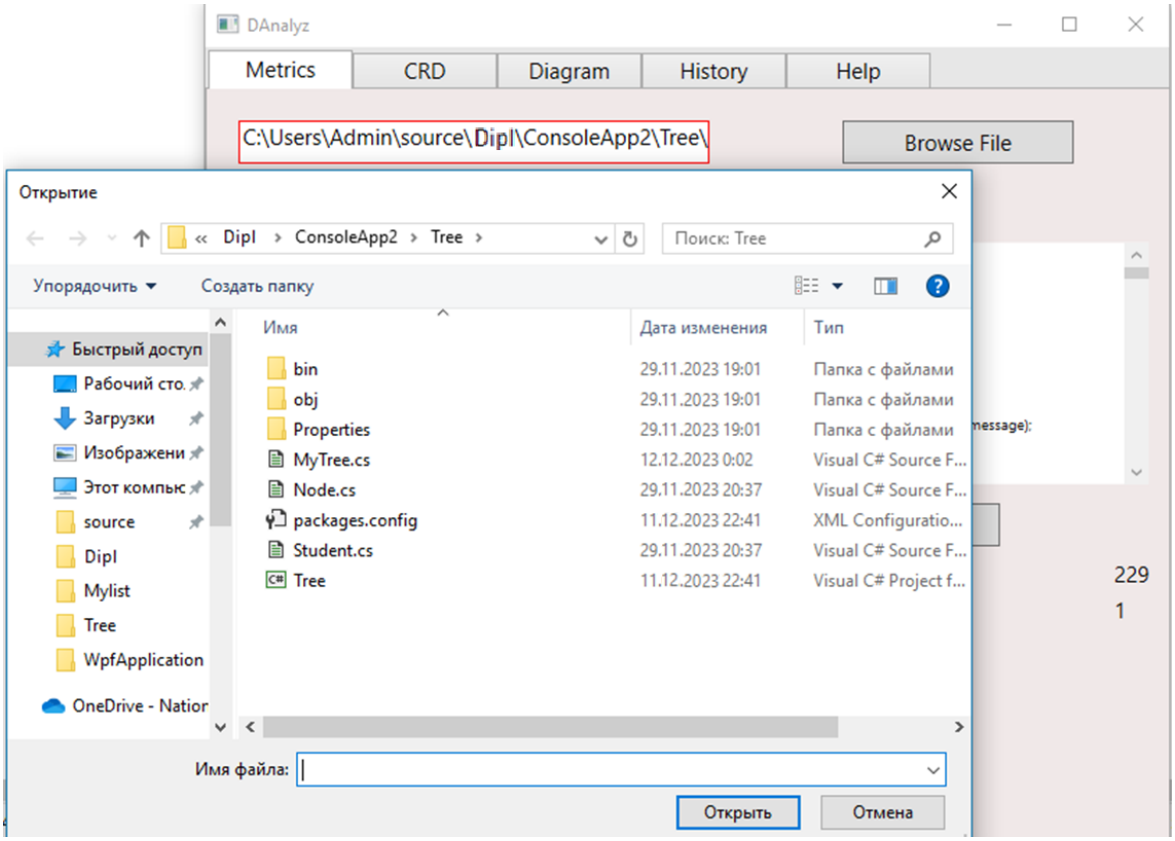

Рис. 4.3. Пошук файлу

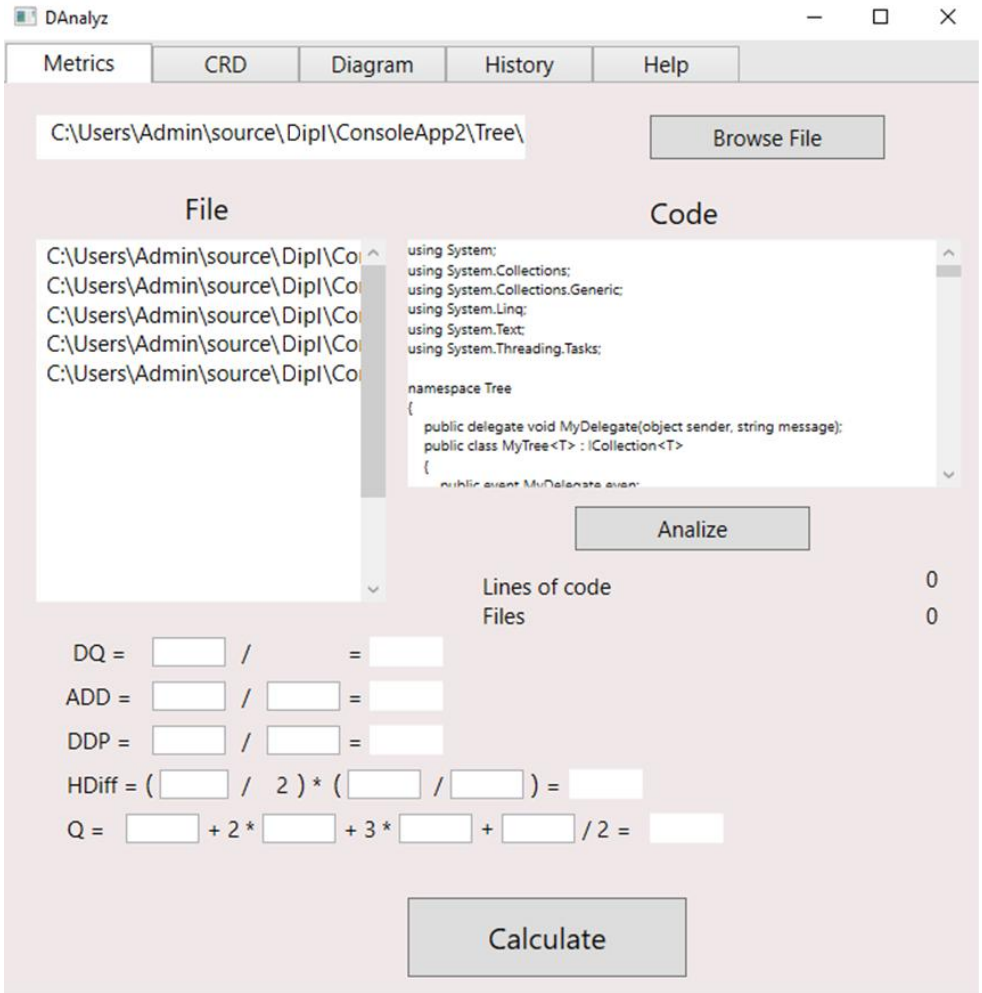

Рис. 4.4. Інформація про обраний файл

Також у вкладці «Метрики» користувачу доступні для виконання ще дві кнопки: «Analyze» та «Calculate». Після обрання потрібного файлу з кодом необхідно підрахувати кількість рядків у ньому. Цей параметр є дуже важливим, так як використовується для підрахунку майже усіх метрик. Для підрахунку кількості рядків коду файлу необхідно натиснути кнопку «Analyze». Потрібна інформація відобразиться під відповідною областю вікна (рис. 4.5).

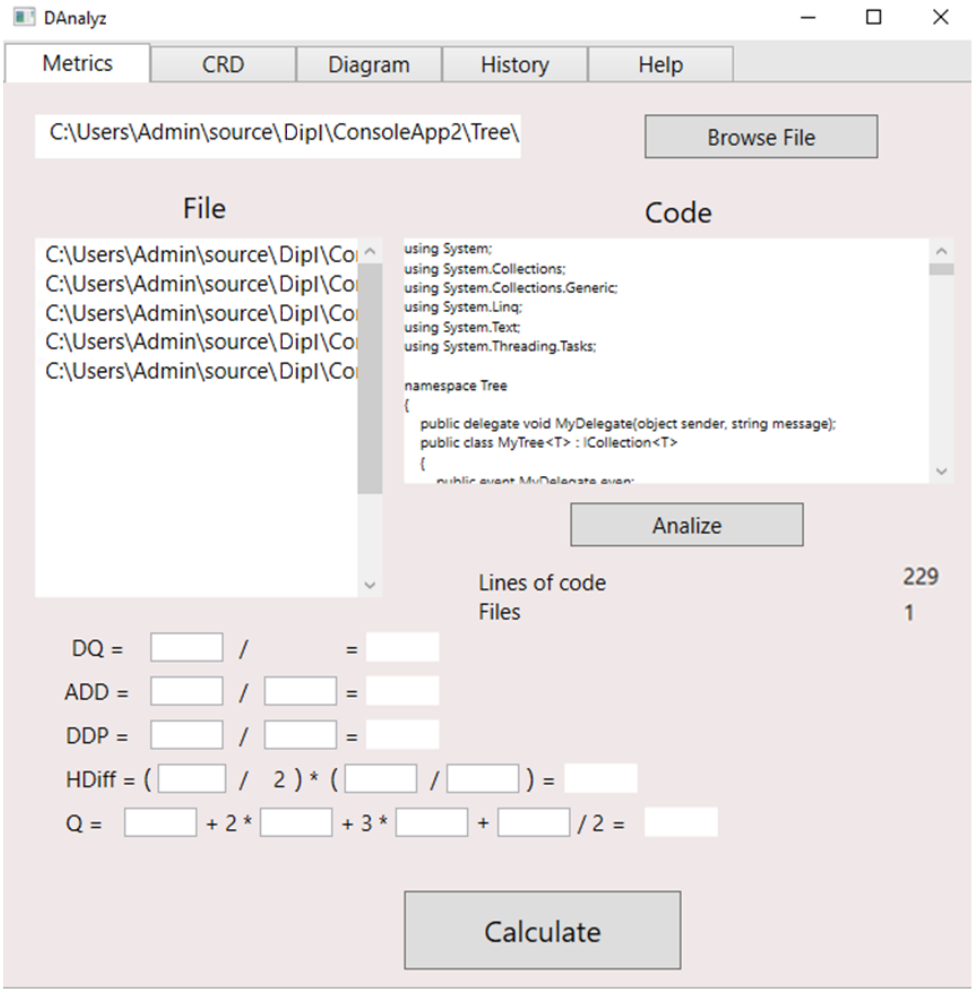

Рис. 4.5. Підрахунок кількості рядків коду файлу

Далі можна розраховувати значення метрик. Засіб робить розрахунок наступних заданих метрик:

1. DQ = Кількість помилок/Кількість рядків коду (DQ – щільність дефектів) (Метрика Альбрехта). Показник щільності дефектів розраховується як відношення загальної кількості виявлених дефектів до кількості тестових

процедур, які були виконані для конкретної функціональності або сценарію використання системи. Таким чином, у випадку високої інтенсивності помилок у певній функціональності слід провести аналіз причин та наслідків. Чи можливо, що функціональність є надто складною, що може впливати на високу щільність помилок? Чи є проблеми, пов'язані з проектуванням чи реалізацією цієї функціональності? Чи можливо, що виділення ресурсів для реалізації цієї функціональності було неправильно оцінене з причини неправильної оцінки ризику? Також може виникнути висновок, що розробникам, відповідальним за цю функціональність, потрібне додаткове навчання [39].

2. ADD = Кількість дефектів/ Кількість рядків доданого коду (ADD – Alternative defect density) (Альтернативна щільність помилок). Оцінка якості програмного застосунку та ефективності доданого коду. Типовим вважається кількість дефектів 1-5 на 1000 рядків коду [39].

3. DDP = Кількість помилок/ Кількість рядків коду певного автора (Defect density person). Аналіз цієї метрики виявити проблемні частини коду, оцінити ефективність роботи конкретного програміста, передбачити потенційні ризики змін, раціонально розподілити ресурси для тестування та вибрати оптимальну дату релізу. Типовим вважається кількість дефектів 1-5 на 1000 рядків коду [39].

4. HDiff – Оцінка складності програми (Halstead Difficulty) (Метрика Холстеда) Складається з багатьох параметрів побудованих на аналізі кількості рядків і синтаксичних елементів вихідного коду програми. Усі ці параметри обчислюються окремо. Ця метрика частково дозволяє врахувати можливість реалізації однієї і тієї ж функціональності різним числом рядків та операторів коду. Також її використовують для запобігання надмірного ускладнення окремих структурних елементів програмного проекту та зростання пов'язаних з цим ризиків.

5. Q – метрика складності потоку управління даних (Метрика Чепина). Краще відслідковувати зміни даної метрики на одному проекті. У разі якщо

вона раптом сильно змінилася, то необхідно вивчати код для виявлення з чим пов'язана ця зміна.

При натисканні кнопки «Calculate» користувач зможе отримати значення усіх вищеописаних метрик для подальшого аналізу програмного коду (рис. 4.6).

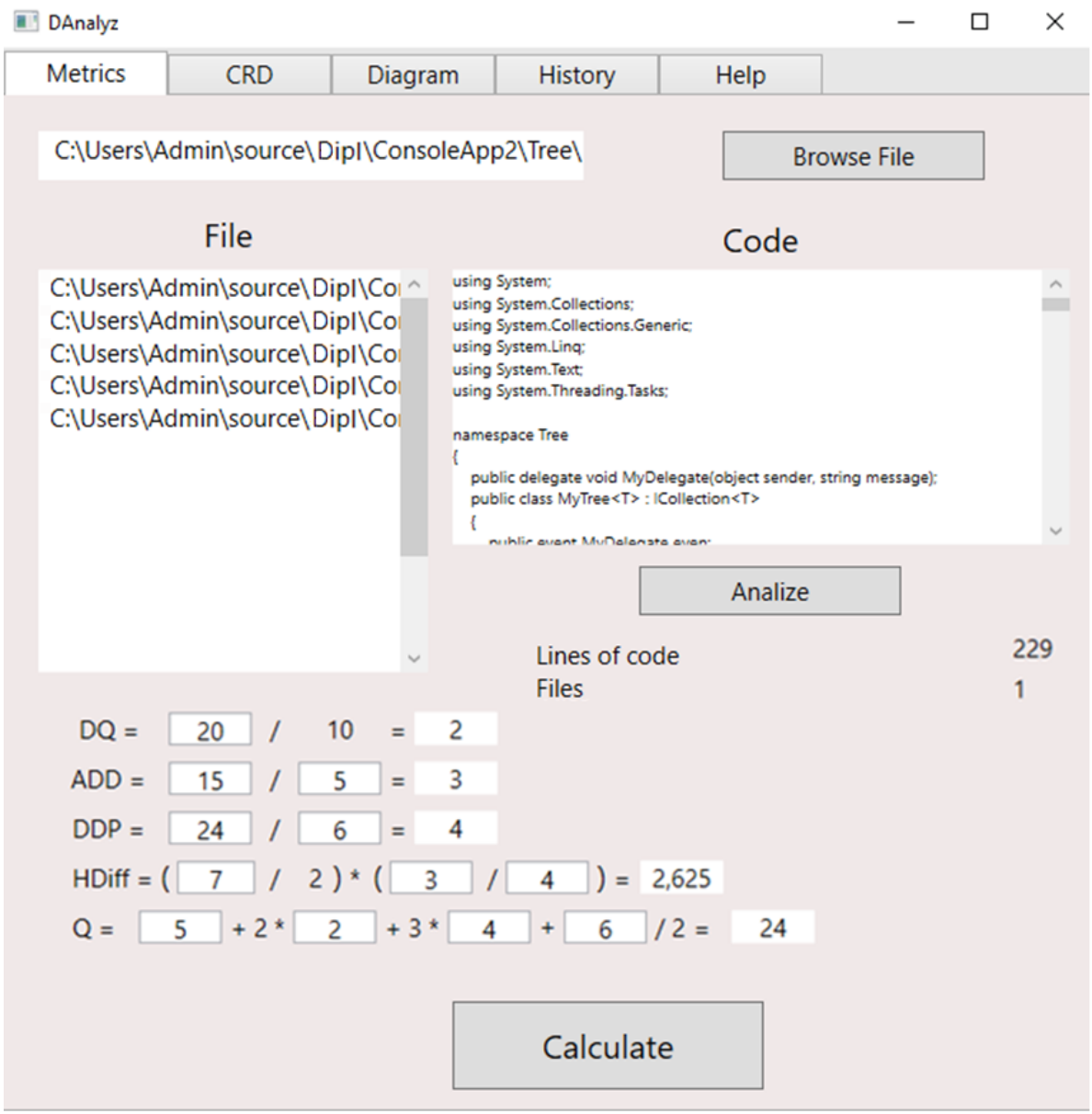

Рис. 4.6. Вікно результату знайдених метрик

Поряд з вкладкою «Метрики» знаходиться вкладка «CRD», вікно якої має наступний вигляд (рис. 4.7).

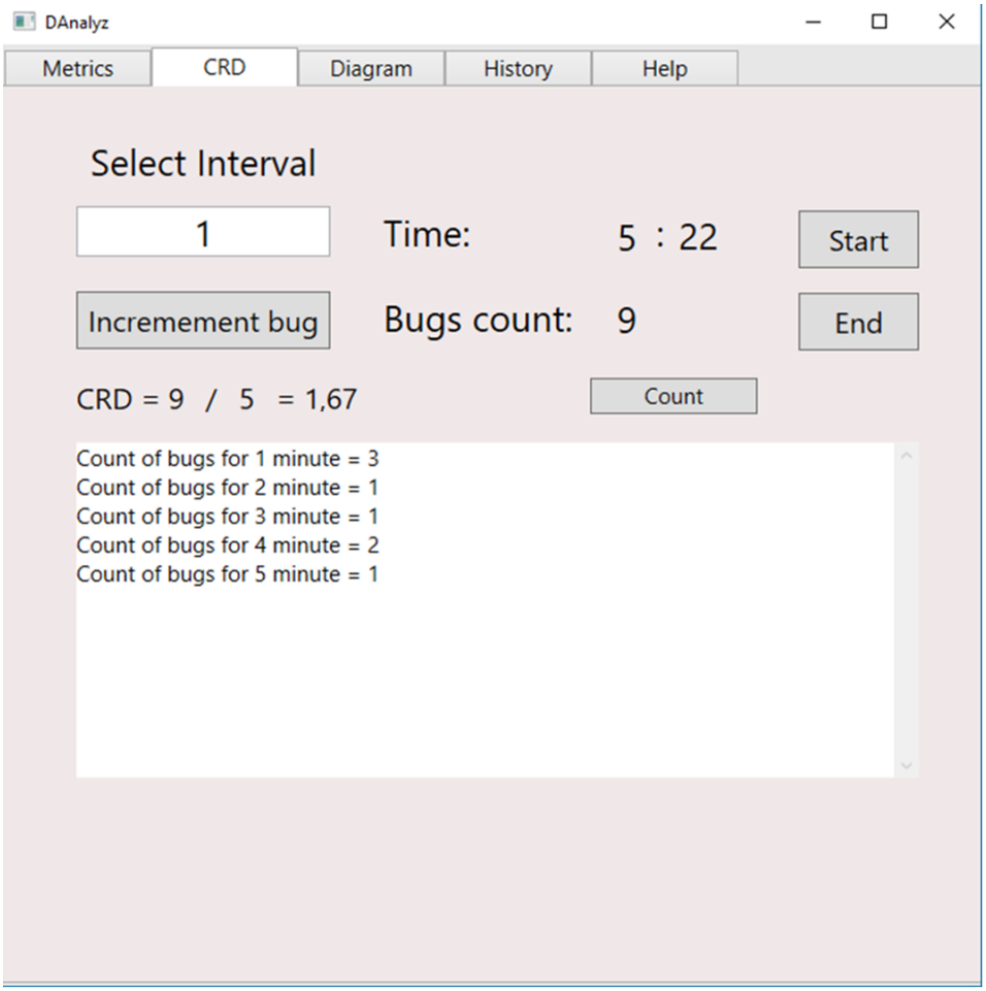

Рис. 4.7. Вікно вкладки «CRD»

CRD це окрема метрика, яка через відмінність підходів її обрахунку винесена у окрему вкладку.

CRD = Кількість дефектів/хвилина, (CRD – code review defects). (Універсальна метрика якості коду). В ідеалі ця метрика вимірюється під час проведення процесу перегляду коду. Перегляд – це діяльність, спрямована на визначення придатності, адекватності та ефективності коду. Щоб переконатися, що код короткий, простий і відповідає меті, «переглядач» запускає секундомір і розпочинає огляд коду. Кожен раз, коли він виявляє помилку в програмі, він додає +1 до свого лічильника. Таким чином, поділивши значення лічильника на час, отримаємо метрику якості коду [39].

В розробленому за стосунку даний процес автоматизовано. Програма сама шукає дефекти за заданим наперед таймером.

Щоб знайти CRD, необхідно спочатку задати параметр для інтервалу лічильника «Вибір інтервалу», а потім натиснути кнопку «Пуск», щоб запустити таймер. Для завершення необхідно натиснути кнопку «Кінець».

В розробленому за стосунку також можна побудувати та подивитися на графічну інтерпретацію отриманих результатів за допомогою вкладки «Діаграма» (рис. 4.8).

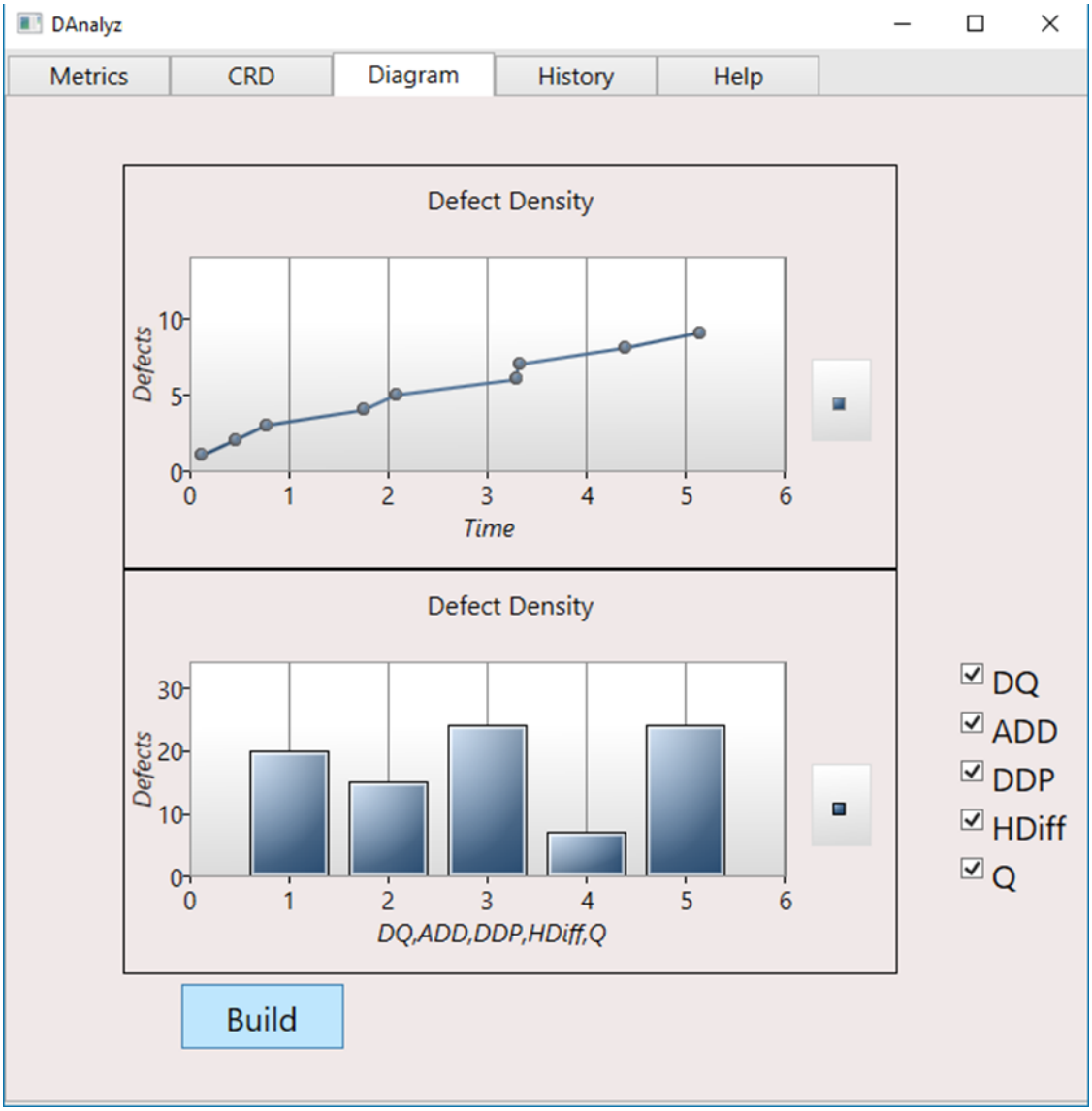

Рис. 4.8. Вікно «Diagram»

У вікні вкладки користувач отримає відображення двох діаграм. Верхня діаграма являє собою плавну діаграму розсіювання, яка відображає

значення метрик, розрахованих у додатку. По осі абсцис відкладається час у хвилинах, а по ординаті – кількість знайдених дефектів. Нижня діаграма являє собою традиційну стовпчасту діаграму, яка за вибором користувача показує кілька розрахункових метрик. По осі абсцис відображається індикатор, а по осі ординат – кількість виявлених помилок (рис. 4.8).

У вікні наступної вкладки «History», користувач може переглянути статистику та інформацію про досліджуваний код зі значеннями отриманих результатів (рис. 4.9).

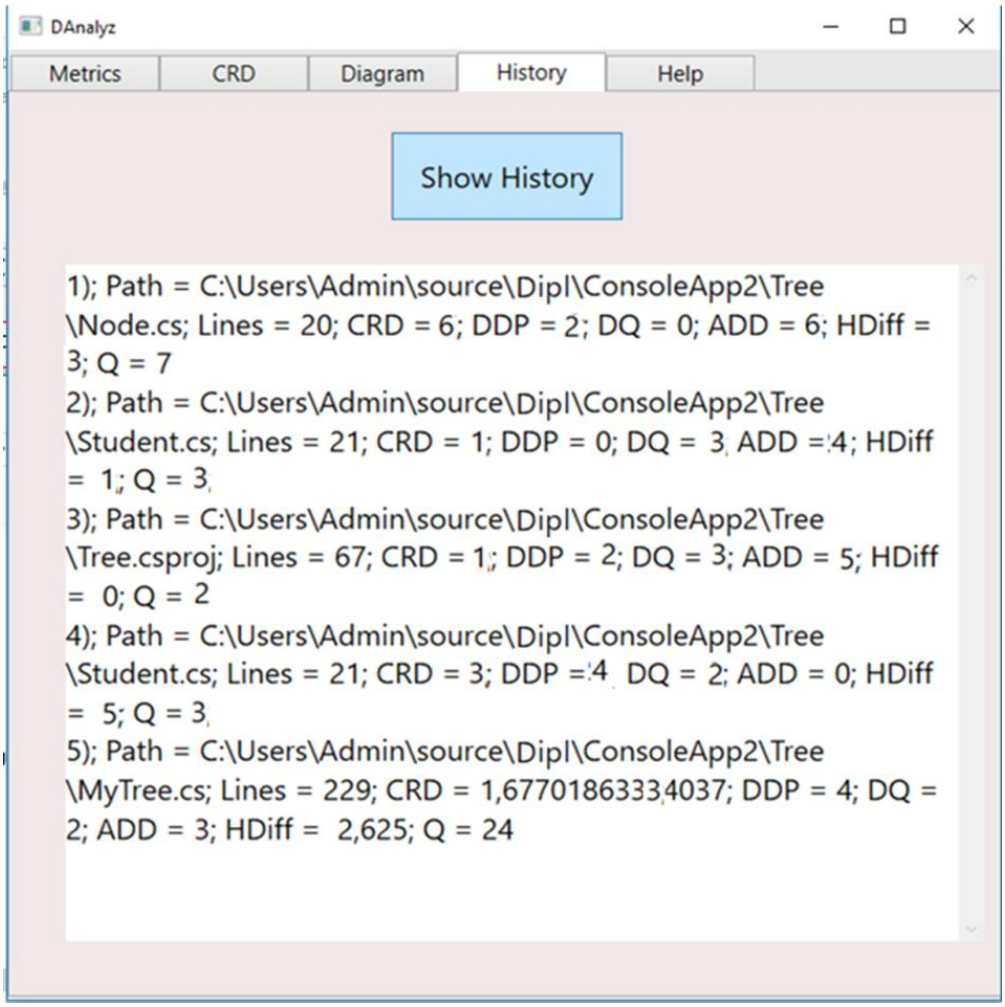

Рис. 4.9. Вікно «History»

Останньою вкладкою за стосунку є вкладка «Help» (рис. 4.10 – 4.11), яка дає можливість ознайомитися як з усією розрахунковою інформацією про метрики, які обчислюються у за стосунку, так і описом кожної з метри, які використовуються для оцінювання якості програмного коду.

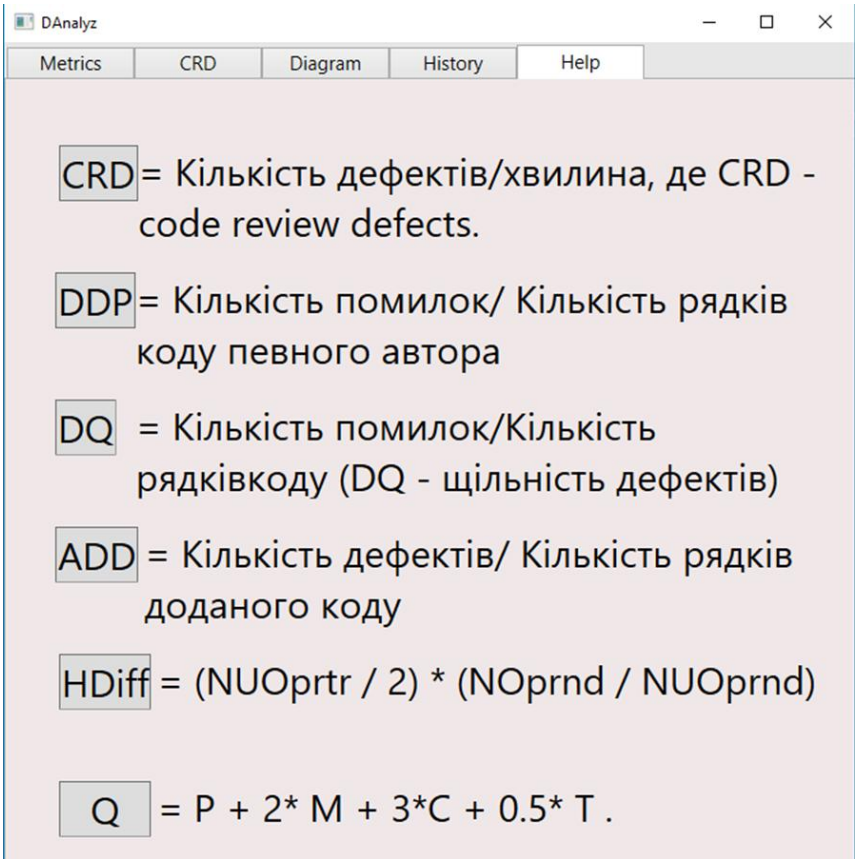

Рис. 4.10. Вікно «Help» з описом метрик

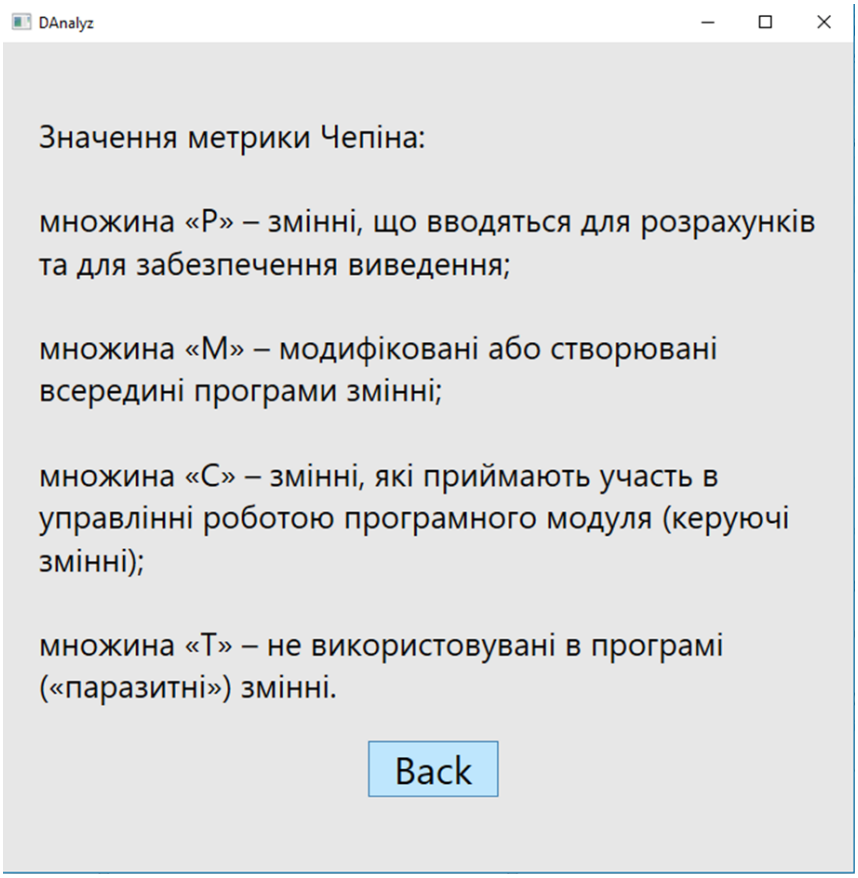

Рис. 4.11. Вікно «Help» з розрахунковою інформацією

ВИСНОВКИ ДО РОЗДІЛУ 4

В даному розділі було описано роботу та архітектуру розробленого програмного засобу для оцінки якості програмного коду. Описані рекомендації по використанню системи для користувача.

Були проаналізовані основні вкладки та кнопки, що беруть участь при оцінці якості коду.

Програма не тільки чисельно вимірює метрики, але також дає можливість переглянути деякі з них у вигляді графічної інтерпретації результатів. Також користувачу пропонується можливість порівняння отриманих результатів в залежності від версій коду одного проекту.

ВИСНОВКИ

Програмний код, який є добре оформленим і легко читається, являється невід'ємним елементом забезпечення простоти підтримки програмного продукту, що значно спрощує роботу розробників. Так як професія програміста передбачає високу рухливість, фахівцям часто доводиться змінювати команди і переходити з одного проекту на інший. Одна і та сама людина за свою кар'єру може брати участь у безлічі проектів, а над кодом одного і того ж проекту протягом його життєвого циклу можуть працювати різні програмісти.

Дотримання встановлених стандартів суттєво полегшує ці процеси, дозволяючи людям зосереджуватись на роботі, а не витрачати час для вивчення коду, та вирівнювання його стилю з попередниками та колегами по команді.

У командній роботі важливо дотримуватись єдиного стилю, навіть якщо це суперечить власним уподобанням програміста. При вступі програміста до нового проекту чи компанії йому часто не доводиться вибирати або залишатися при своєму стилі, йому просто повідомлять, які правила треба використовувати.

Причому при виконанні групових проектів «з нуля», перед початком написання коду, програмістам доводиться розробляти правила самостійно. І тут важливо не брати вимоги до ПЗ навмання – для більшості популярних мов програмування вже існують стандартні угоди, і на них варто орієнтуватися.

Аналіз, проведений у роботі, показав на наявність численних технологій та методологій, які враховують життєвий цикл ПЗ при оцінці його якості. Однак при цьому частина проблем залишається актуальною, і впровадження підходів до оцінки якості ПЗ не демонструє значного зменшення невдач. Навіть, не зважаючи на велику кількість наявних технологій та методологій, процес розробки ПЗ залишається непередбачуваним, а його результат завжди залишається невідомим. Такий стан речей в галузі програмування пояснюється двома основними факторами,

пов'язаними із технологіями та методологіями: або їх недостатньою вдосконаленістю, або невірністю їх вибору.

У роботі показано, що ПЗ володіє рядом характеристик, які різною мірою впливають на вибір технології та методології. На сьогодні процес оцінювання та вибору лишається питанням індивідуального суб'єктивного характеру, оскільки немає жодних чітко визначених критеріїв вибору, жодних рекомендацій, жодних систем підтримки прийняття рішень. Весь цей процес покладено на розробників, які не завжди можуть зробити оптимальний вибір, через нерозуміння всієї складності розробки ПЗ та вплив змін вимог на цей процес.

Таким чином вимірюваність та визначення якості коду – це важлива тема у світі програмування. Фахівці, які мають досвід роботи з великими проектами, не сумніваються в необхідності підтримувати код у якісному стані. Проте у них не завжди вистачає часу для того, щоб з'ясувати, які характеристики важливі саме в цьому проекті.

Насамперед це стосується метрик, за якими треба визначати якість коду. Як правило, для визначення метрики достатньо наперед визначити важливі характеристики коду, такі які відповідність правилам, складність коду, наявність дублікатів, коментування, покриття тестами.

Для кожного критерію та метрики коректності має бути визначене кількісне значення, а для деяких метрик ваговий коефіцієнт. Останній вимірюється в результаті присвоєння деякому атрибуту рангу, що визначає його значимість серед інших атрибутів.

Розроблений у роботі програмний засіб реалізує методику оцінки якості програмного забезпечення за допомогою факторного аналізу. Засіб оцінки коректності дозволяє автоматизувати процес вимірювання та може використовуватися як на ранніх етапах створення ПЗ, так і для оцінки вже існуючих систем, а також в навчальному процесі для спеціальностей, які пов’язані з розробкою та експлуатацією програмного забезпечення.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Стандарт ISO/IEC 12207:1995. Informationtechnologies. Softwarelifecycleprocesses. // ISO/IEC. – 1995. – 61 p.
2. Кендалл С. Уніфікований процес. Основні концепції. – Київ. – 2022. – 157 с.
3. Стандарт ISO/IEC 9126:1991. Software engineering – Product quality.
4. ISO/IEC 9126 [Електронний ресурс] – Режим доступу: https://en.wikipedia.org/wiki/ISO/IEC_9126 (дата звернення 19.12.2023).
5. Стандарт ISO/IEC 25010:2011 : Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models.
6. Денис Сілаков. Якість програмного коду. Подбайте про довге життя ваших програмних продуктів. Журнал системний адміністратор. [Електронний ресурс] – Режим доступу: http://samag.ru/archive/article/2628 (дата звернення 19.12.2023).
7. Список функцій POSIX [Електронний ресурс] – Режим доступу: http://www.opennet.ru/man.shtml?list=3 (дата звернення 19.12.2023).
8. Імена класів в .NET [Електронний ресурс] – Режим доступу: http://msdn.microsoft.com/en-us/library/System.aspx (дата звернення 19.12.2023).
9. Класи Java [Електронний ресурс] – Режим доступу: http://docs.oracle.com/javase/7/docs/api/allclasses-frame.html (дата звернення 19.12.2023).
10. Рекомендації по оформленню коду написаних для Java [Електронний ресурс] – Режим доступу: http://www.oracle.com/technetwork/java/codeconv-138413.html (дата звернення 19.12.2023).
11. Рекомендації по оформленню коду написаних для C# [Електронний ресурс] – Режим доступу: http://msdn.microsoft.com/en-us/library/ff926074.aspx html (дата звернення 19.12.2023).

12. Рекомендації по оформленню коду написаних для Perl [Електронний ресурс] – Режим доступу: http://perldoc.perl.org/perlstyle.html (дата звернення 19.12.2023).

13. Відмінності стилів [Електронний ресурс] – Режим доступу: http://en.wikipedia.org/wiki/Indent_style (дата звернення 19.12.2023).

14. Olena Harytonova. Detection of vulnerabilities in programs with the help of code analyzers [Електронний ресурс] – Режим доступу: https://pvs-studio.com/en/blog/posts/a0028/ (дата звернення 19.12.2023).

15. McCall J. A., Richards P. K., Walters, G. F. Factors in Software Quality: Concept and Definitions of Software Quality. Final Technical Report. Vol. 1. National Technical Information Service, Springfield. [Електронний ресурс]: – Режим доступу: https://apps.dtic.mil/sti/pdfs/ADA049014.pdf (дата звернення 19.12.2023).

16. Mc Call J. A., Richards P. K., Walters, G. F. Factors in Software Quality: Metric Data Collection and Validation. Final Technical Report. Vol. 2. National Technical Information Service, Springfield.

17. Mc Call J. A., Richards P. K., Walters, G. F. Factors in Software Quality: Preliminary Handbook on Software Quality for an Acquisition Manager. Final Technical Report. Vol. 3. National Technical Information Service, Springfield.

18. Boehm B.W., Brown J.R., Kaspar H., Lipow M., MacLeod G.J., Merritt M.J.. Characteristics of Software Quality, TRW Series of Software Technology, Amsterdam, North Holland, 166 p.

19. Grady R.B., Caswell D.L. Software Metrics: Establishing a Company-Wide Program. Prentice-Hall, 275 p.

20. Ghezzi C., Jazayeri M., Mandrioli D. Fundamental of Software Engineering, Prentice–Hall, NJ, USA.

21. Dromey G.R. A model for software product quality // Transactions of Software Engineering. Vol. 21, No. 2. P. 146-162.

22. Hyatt L.E., Rosenberg L.H. A Software Quality Model and Metrics for Identifying Project Risks and Assessing Software Quality // Proceedings of

Product Assurance Symposium and Software Product Assurance Workshop. Noordwijk, P. 209-212.

23. Стандарт ISO/IEC 9126-1:2001. Software engineering – Software product quality – Part 1: Quality model.

24. Стандарт ISO/IEC TR 9126-2:2003 Software engineering – Product quality – Part 2: External metrics.

25. Стандарт ISO/IEC TR 9126-3:2003 Software engineering – Product quality – Part 3: Internal metrics.

26. Стандарт ISO/IEC TR 9126-4:2004 Software engineering – Product quality – Part 4: Quality in use metrics.

27. Bansiya J., Davis C. A Hierarchical Model for Object-Oriented Quality Assessment // IEEE Transactions on Software Engineering. 2022. Vol. 28, No. 1. P. 4-17.

28. Khosravi K., Gueheneuc Y. On Issues with Software Quality Models // Proceedings of 9th ECOOP workshop on Quantitative Approaches in Object-Oriented Software Engineering. 2021. P. 70-83.

29. Chang C., Wu C., Lin H. 2020. Integrating Fuzzy Theory and Hierarchy Concepts to Evaluate Software Quality // Software Quality Control. 2008. Vol. 16, No. 2. P. 263-267.

30. Sharma A., Kumar R., Grover P.S. Estimation of Quality for software components: an empirical approach // ACM SIGSOFT Software Engineering Notes. 2020. Vol. 33, No. 6. P. 1-10.

31. Bass L., Clements P., Kazman R. Software Architecture in Practice. 2Ed. Addison Wesley. 2019. 528 p.

32. ГОСТ 28195-89 [Електронний ресурс]: – Режим доступу: https://pdf.standartgost.ru/catalog/Data2/1/4294826/4294826593.pdf (дата звернення 19.12.2023).

33. Поморова О. В. Аналіз та опрацювання метрик якості програмного забезпечення на етапі проектування / О. В. Поморова, Т. О. Говорущенко, С. Я. Тарасек // Вісник Хмельницького національного університету. – 2020. – № 1. – С. 54–62.

34. Інформаційні технології. Оцінювання програмного продукту. Частина 1. Загальний огляд (ISO/IEC 14598-1:1999, IDT): ДСТУ ISO 7000:2004. – [Чинний від 2006–04–01] . – Київ , 19 с. – (Державний Стандарт України – видано ISO).

35. Michele Lanza, Radu Marinescu and S. Ducasse. Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems. Springer-Verlag Berlin Heilderbeg, 2021. – 213 pages.

36. Тараненко К.Г. Автоматизований аналіз та оцінка зручності використання програмних систем / К. Г. Тараненко, І. В. Гученко // SAIT 2010 – Системний аналіз та інформаційні технології: 12-а Міжнародна наук.-техн. конф., 25–29 травня. 2010 р.: тези доп. – К., 2020. – С. 322.

37. Troelsen A. Japikse Ph. Pro C# 7: With .NET and .NET Core 9Ed. Apress Media. 2019. 1300 p.

38. .NET_Framework [Електронний ресурс]: – Режим доступу: https://uk.wikipedia.org/wiki/.NET_Framework (дата звернення 19.12.2023).

39. Дишлевий О.П., Драпушко Ю.В. Засіб моніторингу та аналіз метрик дефектів якості програмного коду. [Електронний ресурс]: – Режим доступу: https://www.researchgate.net/publication/334399133_ZASIB_MONITORINGU_TA_ANALIZ_METRIK_DEFEKTIV_AKOSTI_PROGRAMNOGO_KODU (дата звернення 19.12.2023)