МІНІСТЕРСТВО ФІНАНСІВ УКРАЇНИ
ДЕРЖАВНИЙ ПОДАТКОВИЙ УНІВЕРСИТЕТ

ФАКУЛЬТЕТ ФІНАНСІВ ТА ЦИФРОВИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНИХ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ І СИСТЕМ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: Інструменти забезпечення якості програмних продуктів

Здобувача вищої освіти другого (магістерського) рівня,
ІІ курсу, групи КМІ-24-2, спеціальності 122 «Комп'ютерні науки»,
освітня програма «Інформаційні управляючі системи і технології в економіці»



Корнутяка Павла Миколайовича

(підпис) (прізвище, ім'я, по батькові)

Науковий керівник 

Горбовий Артур Юліанович
д-р.техн.наук, професор

(підпис) (прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Керівник освітньо-професійної програми 

Омельчук Антон Анатолійович, к.т.н

(підпис) (прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Допущено до захисту за рішенням кафедри протокол від «02» січня 2026 № 10

Завідувач кафедри 

Філоненко Михайло Миколайович, к.ф.-м.н., доцент

(підпис)

(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Ірпінь – 2026

Реєстрація № 5 16.01.2026 (дата) (підпис відповідальної особи)

Результати перевірки до захисту (до захисту, на доопрацювання) 02.01.2026 (дата) (підпис)

Результат захисту 4 (добре) (оцінка) 04.02.2026 (дата) (підпис)

Голова комісії: (підпис) О.Є. Дюбанов

Члени комісії:

(підпис) А.Ю. Горбовий

(підпис) М.М. Філоненко

(підпис) І.Д. Погорєловська

БЛАНК ОЦІНЮВАННЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Об'єкти контролю	Діапазон оцінки в балах	Фактична кількість балів	Примітки	Дата, П.І.Б.: керівника, голови комісії членів комісії,секретаря, (підписи)
ОЦІНЮВАННЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ КОМІСІЄЮ ПО ЗАХИСТУ				
1. Зміст кваліфікаційної роботи (якість теоретичного матеріалу, його зв'язок з практичним, вміння представляти матеріал в логічній послідовності, наявність висновків та пропозицій)	від 0 до 30	21		04.02.2026 Дюбанов О.С. Горбовий А.Ю.
2. Дотримання вимог щодо оформлення	від 0 до 10	8		Філоненко М.М.
3. Вчасне виконання роботи на всіх етапах у встановлені кафедрою терміни	від 0 до 5	3		Погорєловська І.Д. Суховецька Л.П.
4. Рецензент	від 0 до 5	4		
РАЗОМ БАЛІВ	від 0 до 50	36		

Оцінювання кваліфікаційної роботи науковим керівником - 35 балів.

ПЕРЕВЕДЕННЯ ОЦІНКИ ЗА 100-БАЛЬНОЮ ШКАЛОЮ ДО НАЦІОНАЛЬНОЇ ШКАЛИ ТА ШКАЛИ ЄКТС

Оцінка за 100-бальною шкалою	Оцінка за шкалоюЄКТС	Оцінка за національною шкалою	Дата	П.І.Б.: голови, членів комісії, секретаря, (підпис)
71	C	4	04.02.2026	Дюбанов О.С. Горбовий А.Ю. Філоненко М.М. Погорєловська І.Д. Суховецька Л.П.

ПРАВИЛА ПЕРЕВЕДЕННЯ ОЦІНОК ДО НАЦІОНАЛЬНОЇ ШКАЛИ ТА ШКАЛИ ЄКТС

Оцінка за 100-бальною шкалою	Оцінка за шкалою ЄКТС	Оцінка за національною шкалою
90-100	A	5 (відмінно)
80-89	B	4 (добре)
70-79	C	
60-69	D	3 (задовільно)
50-59	E	
35-49	FX	2 (незадовільно)

Факультет фінансів та цифрових технологій
Кафедра комп'ютерних та інформаційних технологій і систем
Галузь знань 12 «Інформаційні технології»
Спеціальність 122 «Комп'ютерні науки»
Освітня програма «Інформаційні управляючі системи і технології в економіці»

Затверджую
Завідувач кафедри _____ Філоненко М.М.
«11» грудня 2024

Завдання на кваліфікаційну роботу

здобувачу вищої освіти

Корнутяку Павлу Миколайовичу

(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи **Інструменти забезпечення якості програмних продуктів**
Затверджена наказом ректора від «10» грудня 2024 № 1852
2. Строк здачі закінченої роботи «02» січня 2026
3. Мета роботи обґрунтування та визначення комплексу методів і програмних інструментів, спрямованих на аналіз якісних показників програмного забезпечення
4. Об'єкт дослідження якісні характеристики програмного коду
Предмет дослідження методичний та програмний інструментарій оцінювання якості ПЗ.

5. Вихідні дані до роботи програмне забезпечення та методики, технології, інструменти оцінювання його якості Перелік графічного матеріалу схема моделі якості ПЗ, копії екранів, лістинг з програмним кодом, презентація роботи(.ppt).
6. Зміст кваліфікаційної роботи (перелік питань за кожним розділом)

Розділ 1. Теоретичні аспекти оцінювання якості програмного забезпечення. 1.1. Якість ПЗ як предмет стандартизації. 1.2. ***Інструменти оцінювання якості програмного забезпечення.***

Розділ 2. Застосування метричного аналізу у визначенні якості програмного забезпечення 2.1. Застосування метрики Холстеда 2.2. Застосування метрики Маккейба – цикломатична складність 2.3. Застосування метрик визначення складності програмного продукту (метрика Чепіна, Джілбі Т.).

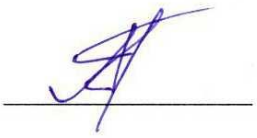
Розділ 3. Засоби автоматизованого оцінювання якості програмного забезпечення 3.1. Передумови автоматизованого тестування 3.2. Види автоматизованих тестів. 3.3. Основні інструменти для автоматизованого тестування.

7. Календарний план виконання роботи

№ п/п	Назва етапів кваліфікаційноїроботи	Строк виконання етапів роботи	
		За планом	Фактично
1	2	3	4
1	Вибір теми, підбір літературних джерел, формування структури кваліфікаційної роботи	До 15.12.2024	До 15.12.2024
2	**І розділ** (розкрити сутність і стан дослідження наукової проблематики роботи, після чого навести її загальну характеристику; огляд літератури за темою і вибір конкретних напрямів досліджень)	30.05.2025	30.05.2025
3	**ІІ розділ** (аналіз експериментальної частини про методики досліджень)	10.09.2025	10.09.2025
4	**ІІІ розділ** (проведені теоретичні і (або) експериментальні дослідження та проектні рішення)	20.11.2025	20.11.2025
5	**Висновки**	11.12.2025	11.12.2025
6	Доопрацювання роботи з урахуванням зауважень наукового керівника	25.12.2025	25.12.2025
7	Отримання відгуку керівника	28.01.2026	28.01.2026
8	Отримання зовнішньої рецензії	30.01.2026	30.01.2026

8. Дата видачі завдання «11» грудня 2024

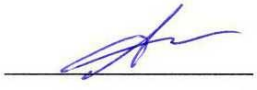
9. Керівник кваліфікаційної роботи



Горбовий А.Ю.

(підпис) (прізвище, ініціали)

10. Керівник освітньо-професійної програми



Омельчук А.

(підпис) (прізвище, ініціали)

11. Завдання до виконання отримав здобувач вищої освіти



Корнутяк П.М

(підпис) (прізвище, ініціали)

АНОТАЦІЯ

Мета кваліфікаційної роботи: визначення методів та програмних засобів аналізу якості програмного забезпечення

Завдання, що були вирішені при виконанні роботи:

- визначено сутність поняття “якість програмного забезпечення” як предмету стандартизації;

- проаналізована модель якості ПЗ, яка визначена стандаром ДСТУ ISO/IEC 25010;

- досліджено існуючі інструменти оцінювання якості програмного забезпечення — метрики, статичне та динамічне тестування;

- визначено якість програмного коду за результатами метричного аналізу, з використанням метрики Холстеда, Мак-Кейба, Чепіна, Джілбі;

- визначено передумови автоматизованого тестування;

- проаналізовано види автоматизованих тестів, продемонстровано застосування інструментів автоматизованого тестування

Об’єкт дослідження: якість програмного коду.

Предмет дослідження: засоби оцінювання якості програмного забезпечення.

Інформаційною базою дослідження є стадарти, вимоги яких спрямовані на забезпечення якості програмного забезпечення, метрики оцінювання складності ПЗ, методи та інструменти тестування ПЗ.

За результатами дослідження було сформульовано висновки про те, що якість ПЗ є багатогранною характеристикою, яка включає різні аспекти, такі як ефективність, надійність, зручність у використанні та інші. Оцінювання якості вимагає комплексного застосування метричного аналізу, статичного та динамічного аналізу коду, автоматизованих тестів, є підґрунтям створення високоякісного програмного продукту, який відповідає вимогам замовника та забезпечує йому надійність та ефективність у використанні.

Одержані результати можуть бути використані при визначені якості програмного коду, а також в освітньому процесі при викладанні навчальних

дисциплін “Моделі та методи забезпечення якості ІУС”, “Інноваційне управління ІТ-проєктами”.

Кваліфікаційної роботи містить 58 сторінок до яких входять вступ, основна частина, яка складається з трьох розділів, до роботи входять 4 рисунки, 6 таблиць, висновки, перелік використаної літератури, що містить 35 джерела, 11 додатків.

Ключові слова: якість програмного забезпечення, метрики, статистичний аналіз, динамічний аналіз, тестування.

Рік виконання дипломної роботи 2025-2026р.

Рік захисту роботи 2026р.

ANNOTATION

Software quality assessment tools.

The purpose of the qualification work: definition of software quality analysis methods and tools.

The tasks that were solved during the performance of the work: - the essence of the concept of "software quality" as a subject of standardization is determined; - the analyzed software quality model, which is defined by the DSTU ISO/IEC 25010 standard; - existing software quality assessment tools — metrics, static and dynamic testing — were investigated; - the quality of the software code was determined based on the results of metric analysis, using Halstead, McCabe, Chapin, Gilby metrics; - prerequisites for automated testing are defined; - analyzed the types of automated tests, demonstrated the use of automated testing tools Object of research: quality of software code.

The subject of research: software quality assessment tools.

The information base of the research is the standards, the requirements of which are aimed at ensuring the quality of the software, metrics for evaluating the complexity of the software, methods and tools for testing the software.

Based on the results of the research, the conclusions were formulated that software quality is a multifaceted characteristic that includes various aspects, such as efficiency, reliability, ease of use, and others.

Quality assessment requires the complex application of metric analysis, static and dynamic code analysis, automated tests, and is the basis for creating a high-quality software product that meets the customer's requirements and ensures its reliability and efficiency in use.

The obtained results can be used to determine the quality of the software code, as well as in the educational process when teaching the educational disciplines "Models and methods of quality assurance of IUS", "Innovative management of IT projects".

The qualification work contains 58 pages, which include an introduction, the main part, which consists of three sections, the work includes 4 figures, 6 tables, conclusions, a list of used literature, containing 35 sources, and 11 appendices.

Keywords: software quality, metrics, statistical analysis, dynamic analysis, testing.

Year of completion of the thesis 2025-2026.

Year of work protection 2026.

ЗМІСТ

ВСТУП 3

РОЗДІЛ 1. ТЕОРЕТИКО-МЕТОДОЛОГІЧНІ ОСНОВИ ОЦІНКИ ЯКОСТІ ПРОГРАМНИХ ПРОДУКТІВ……. 6

1.1. Стандартизація як базис визначення якості ПЗ……..... 6

1.2. Огляд сучасного інструментарію для аналізу якісних характеристик програм..…… 9

РОЗДІЛ 2. ПРАКТИЧНЕ ВИКОРИСТАННЯ МЕТРИЧНОГО ПІДХОДУ ДЛЯ ВИМІРЮВАННЯ ПАРАМЕТРІВ ПЗ …… 14

2.1. Оцінювання програмного коду за допомогою метричного апарату Холстеда …….....…… 14

2.2. Дослідження структурної складності алгоритмів за методом Маккейба…….…… 18

2.3. Аналіз складності програмних систем на основі альтернативних метрик (Чепіна та Т. Джілба)…….…… 20

2.3.1. Особливості та функціональне призначення метрики Чепіна…….... 20

2.3.2. Методика обчислення показників складності заТ. Джілбі……... 22

РОЗДІЛ 3.ТЕХНОЛОГІЇТА ІНСТРУМЕНТИ АВТОМАТИЗАЦІЇ П РОЦЕСІВ ОЦІНЮВАННЯ ЯКОСТІ..……..…… 23

3.1. Концептуальні засади та передумови впровадження автоматизації в тестування..……...…… 23

3.2. Класифікація сучасних типів автоматизованих тестів …….. 31

3.3. Порівняльний аналіз провідних інструментів автоматизованого контролю якості…… 37

ВИСНОВКИ……...……...…… 46

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ…… 48

ДОДАТКИ…… 52

ВСТУП

Актуальність теми. Динамічний розвиток глобального інформаційного простору та цифрова трансформація економічних процесів висувають підвищені вимоги до надійності та функціональності програмних рішень. У сучасних умовах якість програмного забезпечення (ПЗ) трансформується з технічного показника у стратегічний фактор безпеки та стабільності цифрової інфраструктури. Необхідність мінімізації ризиків при експлуатації критично важливих систем обумовлює пошук новітніх методологічних підходів до вимірювання якісних атрибутів коду. Актуальність дослідження підсилюється потребою в інтеграції класичних метричних моделей із сучасними автоматизованими технологіями контролю якості.

Стан та рівень наукового опрацювання теми. Теоретичний фундамент та практичні засади управління якістю ПЗ закладені у працях провідних науковців. Проблематику стандартизації та оцінювання досліджували О. Димо, В. Ліпаєв, К. Вігерс, Н. Фентон. Значний внесок у розвиток інструментарію метричного аналізу та методології тестування зробили Ю. Грицюк, О. Андрущакевич, В. Харченко, Г. Майєрс, С. Макконнелл та І. Соммервилл. Попри значну кількість публікацій, питання синергії статичних метрик та засобів автоматизації у межах єдиного циклу розробки потребують подальшої деталізації.

Мета дослідження полягає у науковому обґрунтуванні та практичній апробації сукупності методологічних підходів і програмних засобів, призначених для комплексної діагностики якісних параметрів програмного забезпечення.

Для досягнення поставленої мети визначено такі завдання:

- дослідити сутність терміну «якість програмного забезпечення» у контексті сучасних стандартів;
- проаналізувати структурну модель якості ПЗ згідно з положеннями ДСТУ ISO/IEC 25010;
- систематизувати методичний інструментарій оцінювання, включаючи стратегії статичного та динамічного аналізу;

- здійснити кількісне оцінювання програмного коду за допомогою метрик М. Холстеда, Т. Маккейба, В. Чепіна та Т. Джілба;
- визначити функціональні та організаційні передумови переходу до автоматизованого тестування;
- здійснити класифікацію автоматизованих перевірок та продемонструвати ефективність їх застосування на практиці.

Об’єкт дослідження: сукупність якісних характеристик та параметрів програмного коду.

Предмет дослідження: методики, математичні моделі та спеціалізоване програмне забезпечення для оцінювання рівня якості ПЗ.

Методи дослідження. Методологічну основу роботи становлять: системний аналіз (для узагальнення теоретичної бази); порівняльно-аналітичний метод (при зіставленні метрик складності); методи математичного моделювання та оцінювання (при розрахунку параметрів коду); методи класифікації та групування (для диференціації програмних структур).

Інформаційна база дослідження. Підґрунтям роботи послужили міжнародні та державні стандарти (ISO/IEC), технічна документація метричних систем, а також сучасні методології забезпечення якості та верифікації ПЗ.

Структура кваліфікаційної роботи. Робота включає вступ, три розділи основної частини, загальні висновки, список використаних джерел (35 найменувань) та 10 додатків. Обсяг пояснювальної записки ілюстровано 3 рисунками та 6 таблицями.

Стислий опис результатів дослідження. У роботі доведено, що якість програмного продукту є інтегральною характеристикою, яка поєднує стійкість, адаптивність та зручність використання. Встановлено, що найбільш достовірні результати забезпечує комбінований підхід, який поєднує метричний аналіз внутрішньої структури коду з динамічними тестами та інструментами автоматизації.

Науково-практичне значення результатів. Сформульовані в роботі пропозиції та результати розрахунків можуть бути використані ІТ-підрозділами

при проведенні внутрішнього аудиту якості коду, а також в освітньому процесі Державного податкового університету при вивченні фахових дисциплін з управління ІТ-проєктами та забезпечення якості інформаційних систем.

РОЗДІЛ 1. ТЕОРЕТИКО-МЕТОДОЛОГІЧНІ ОСНОВИ ОЦІНКИ ЯКОСТІ ПРОГРАМНИХ ПРОДУКТІВ

1.1. Стандартизація як базис визначення якості ПЗ

Функціонування та розвиток сучасних економічних галузей безпосередньо залежить від впровадження високоякісних програмних рішень. Ефективність процесів прийняття управлінських рішень визначається рівнем досконалості відповідного програмного продукту. В контексті інформаційних технологій якість програмного забезпечення (ПЗ) розглядається як його фундаментальна характеристика, що відображає рівень відповідності встановленим технічним і функціональним критеріям. Інтерпретація даного поняття може варіюватися залежно від специфіки конкретної системи, проте у загальному розумінні якість ПЗ трактують як сукупність властивостей, що забезпечують здатність продукту задовольняти сформульовані або передбачувані потреби замовника, визначені на початкових стадіях розробки [8, с.159]. Принципи забезпечення та оцінювання якості регулюються комплексом міжнародних та національних стандартів.

Сучасна методологія базується на серії стандартів ISO/IEC 25000, відомій як SQuaRE (Software Quality Requirements and Evaluation). Класифікація атрибутів та метрик якості регламентована нормативом ДСТУ ISO/IEC 25023:2019. Відповідно до цього документа, якість ПЗ визначається як міра здатності програмного продукту задовольняти встановлені потреби за умови його експлуатації у визначеному середовищі [1].

Стандарт ДСТУ ISO/IEC 25022:2019 вводить категорію «якість при використанні», що дозволяє оцінювати продукт з позиції кінцевого користувача. Цей термін описує ступінь, до якого система забезпечує досягнення цілей користувачів з необхідним рівнем результативності, економічної ефективності та безпеки, мінімізуючи ризики у конкретних сценаріях використання [2]. Організаційні та інженерні аспекти управління якістю протягом усього життєвого циклу розробки визначаються стандартом ДСТУ ISO/IEC/IEEE 12207:2018 [4]. Згідно з його положеннями, контроль якості має бути

безперервним і охоплювати: зіставлення вимог замовника з встановленими індикаторами якості; проведення верифікації та валідації проміжних результатів на кожному етапі реалізації проєкту; тестування фінальної версії продукту, систематизацію даних про дефекти та відмови, виявлені під час експлуатації; обґрунтування вибору моделей надійності на основі отриманих результатів випробувань; підтвердження досягнення цільових показників якості, зафіксованих у технічному завданні [4].

Враховуючи вищевикладене, якість ПЗ є відносною категорією, де нормативні вимоги корелюють із умовами функціонування продукту в певній галузі. Як зазначається у джерелі [8], якість ПЗ формується через якість процесів розробки, якість отриманих продуктів та якість супроводу. Процесна компонента визначає рівень формалізації та достовірності інженерних операцій, що тісно пов’язано з підтвердженням правильності проміжних результатів. Виявлення та корекція помилок у готовому коді здійснюється через методи тестування, що мінімізують кількість дефектів. Висока якість продукту досягається завдяки контролю проміжних артефактів та використанню прогресивних засобів підтримки.

Результативність впровадження ПЗ також обумовлена кваліфікацією персоналу та чіткістю дотримання правил експлуатації [8, с.162]. Для підтвердження відповідності ПЗ узгодженим із замовником вимогам необхідно використовувати строгі методи та моделі. Модель якості формується для визначення системи критеріїв, що всебічно описують властивості програмного продукту. Стандарт ДСТУ ISO/IEC 25010:2016 [3] пропонує ієрархічну модель якості, представлену на чотирьох рівнях.

Перший рівень визначає базові характеристики якості, кожна з яких відображає специфічний погляд користувача. Модель включає шість фундаментальних показників (рис. 1.1)

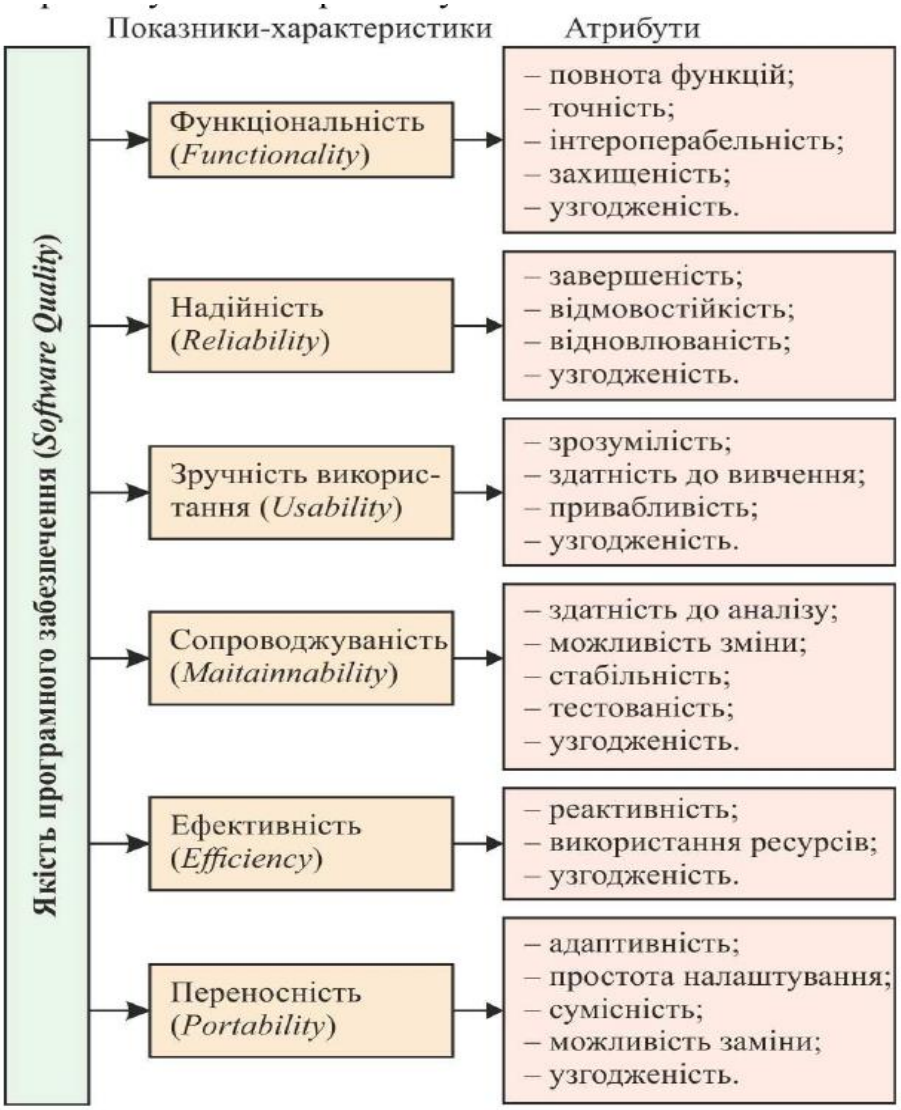


Рис. 1.1. Модель якості програмного забезпечення (ISO/IEC 25010)

Джерело[8, с. 162]

Перший рівень ієрархії моделі охоплює такі фундаментальні показники:

1. Функціональність (functionality) — визначає спроможність системи виконувати завдання, що повністю відповідають запитам споживача (зокрема, оцінюються точність, повнота, захищеність та інтероперабельність).

2. Надійність (reliability) — характеризує здатність програмного виробу підтримувати працездатний стан у заданих експлуатаційних умовах протягом визначеного часового інтервалу.

3. Зручність використання (usability) — властивість системи, що забезпечує користувачеві можливість ефективної взаємодії з програмним продуктом при мінімальних інтелектуальних витратах.

4. Ефективність (efficiency) — відображає раціональність співвідношення між рівнем наданих послуг та обсягом системних ресурсів, що залучаються для їх реалізації.

5. Зручність супроводу (maintainability) — визначає легкість здійснення модифікацій, виправлення виявлених дефектів та адаптації ПЗ до нових вимог.

6. Мобільність (portability) — здатність програмного забезпечення до успішного перенесення між різними апаратно-програмними платформами без деградації його функціональних можливостей.

На другому рівні кожна з наведених характеристик деталізується через сукупність атрибутів, які виступають об'єктами подальшого метричного аналізу.

Третій рівень ієрархії фокусується на безпосередньому вимірюванні атрибутів за допомогою метрик, що інтегрують методики оцінювання та відповідні шкали величин. Процедури вимірювання проводяться на різних стадіях життєвого циклу, охоплюючи також етап фінального тестування продукту.

Четвертий рівень присвячений інтерпретації накопичених кількісних та якісних даних. Важливо зауважити, що пріоритетність окремих характеристик встановлюється залежно від функціонального призначення та специфіки експлуатації програмної системи. Таким чином, модель якості постає як структурований механізм для проведення об’єктивної експертизи програмного продукту [6].

Отже, у межах даного підрозділу було досліджено сутність якості програмного забезпечення як ключового об’єкта стандартизації. Також ідентифіковано специфічні властивості процесів оцінювання та розкрито зміст рівнів ієрархічної моделі представлення якості ПЗ.

1.2. Інструменти оцінювання якості програмного забезпечення

На сучасному етапі розвитку ІТ-галузі сформовано значний арсенал інструментарію, що дозволяє проводити комплексну оцінку виробничих та

експлуатаційних характеристик ПЗ [12]. Відповідно до положень стандарту ISO 25010, процедура оцінювання базується на атрибутах якості, визначених у ISO 25023:2019 [1]. При цьому здійснюється аналіз як окремих підхарактеристик, так і загальних показників якості, що в сукупності надають цілісне уявлення про продукт. Згідно з термінологією ISO 25010, «характеристика» розглядається як набір властивостей, що дають змогу описати та верифікувати якість програмного продукту [3]. Кількісне вираження підхарактеристик зазвичай обчислюється як середньозважене значення відповідних атрибутів із урахуванням коефіцієнтів їхньої вагомості [12].

Основним інструментом кількісного вимірювання є метрики якості. У стандарті ISO/IEC 25022:2016 метрику визначено як модель вимірювання атрибута, що корелює з певною характеристикою якості. Вона являє собою поєднання конкретної методики отримання значень та шкали їх оцінювання [2]. Згідно з ISO 24765:2010, метрика — це чисельна міра ступеня володіння об’єкта певною властивістю. Отже, метрика ПЗ — це індикатор, що дозволяє отримати кількісну оцінку характеристик системи або її компонентів для перевірки їх відповідності нормативним вимогам [7].

Впровадження метричного аналізу дозволяє вирішувати ряд практичних завдань:

- апріорне прогнозування щільності дефектів у ПЗ;
- оцінювання складності продукту та витрат на його подальший супровід;
- визначення обсягу необхідного тестування та прогнозування кількості прихованих помилок;
- розрахунок фінального розміру коду на основі архітектурних рішень;
- аналіз впливу структурних особливостей коду на підсумкову якість продукту;
- моніторинг процесів реалізації проєкту та порівняння ефективності різних технологій розробки [9].

На етапі проєктування застосовують групу прогнозних метрик, зокрема:

1. Експертну оцінку LOC (кількість рядків коду).
2. Метрику Холстеда (аналіз лексичного складу програми).
3. Цикломатичну складність Маккейба.
4. Метрику Джилба (логічна насиченість умовними переходами).
5. Прогнозну кількість операторів та оцінку складності інтерфейсів.
6. Метрики процесів розробки (тривалість, вартість, трудовитрати за моделлю Боема).
7. Функціональний розмір FP (Function Points) [12]. Детальний розгляд цих показників наведено у другому розділі.

Залежно від стану об’єкта оцінювання, метрики поділяють на внутрішні, зовнішні та експлуатаційні [13]. Внутрішні метрики фокусуються на структурних властивостях коду, що визначають потенційну здатність системи задовольняти вимоги. Зовнішні метрики оцінюють поведінку ПЗ під час його функціонування. Експлуатаційні метрики (quality in use) відображають рівень задоволеності користувача в реальному середовищі, безпеку та ефективність вирішення прикладних завдань. Найбільш поширеними є метрики зв’язності, зчеплення та прогнозування ресурсних витрат [13].

Ключовою технікою верифікації якості є тестування — методологія контролю, що базується на зіставленні фактичної та очікуваної поведінки системи на обмеженому наборі тестових сценаріїв [23]. Воно охоплює перевірку коду в статичному режимі (рецензування, інспекції) та в динаміці (виконання на тестових даних).

Процедури тестування регламентуються серією стандартів ISO/IEC/IEEE 29119, що визначають концепції, процеси, документацію та техніки випробувань [5]. Модульне тестування передує інтеграційному, під час якого перевіряється взаємодія компонентів та їх інтерфейсів. Отримані дані про збої та відмови стають основою для фінального оцінювання характеристик якості.

Статичний аналіз коду передбачає ідентифікацію дефектів та аномалій без безпосереднього запуску програми. Це дозволяє не лише знаходити помилки, а

й контролювати дотримання корпоративних стандартів кодування [15]. Основні засоби статичного аналізу включають [18]:

- Лінтери (Linters): контроль відповідності стилістичним правилам (наприклад, ESLint, Pylint).
- Статичні аналізатори: глибокий пошук логічних помилок та проблем безпеки (SonarQube, Checkmarx).
- Аналізатори вразливостей: фокусуються на виявленні критичних прогалин, таких як SQL-ін'єкції (Veracode).
- Аналізатори покриття коду: визначення частки коду, що була активована під час тестів (JaCoCo).
- Аналізатори залежностей: моніторинг безпеки сторонніх бібліотек (Snyk, Dependency-Check). Динамічний аналіз коду полягає у дослідженні системи безпосередньо під час її виконання. Його результативність критично залежить від якості вхідних даних та повноти тестового покриття. Цей підхід є визначальним для систем, де критичними є час відгуку, надійність та ресурсна ефективність [34]. До засобів динамічного аналізу належать:
- Дебагери: покрокове виконання коду та аналіз стану змінних.
- Профілювальники: моніторинг використання ресурсів та продуктивності у реальному часі.
- Юніт-тести та автоматизовані фреймворки: перевірка коректності окремих функцій.
- Інструменти навантажувального тестування: аналіз реакції системи на інтенсивні запити [25].

Важливим складником оцінювання є аналіз покриття коду, мета якого — ідентифікувати ділянки програми, що не були задіяні під час виконання тестів. Якщо повнота покриття не відповідає встановленому рівню (особливо для систем критичного призначення класу А), необхідно розробляти додаткові сценарії [6]. Причинами недостатнього покриття можуть бути:

- прогалини у формуванні тестових випадків на основі вимог;
- неадекватність самих специфікацій;

- наявність «мертвого» коду;
- дезактивований код (фрагменти для специфічних конфігурацій);
- надмірність логічних умов (наприклад, логічні суперечності у виразах);
- захисний код (обробники виняткових ситуацій, що не виникають у тестовому середовищі, як-от гілки default в операторах switch) [30, 34].

Отже, сучасний інструментарій оцінювання якості ПЗ включає розгалужену систему метричного аналізу, а також методи статичного та динамічного тестування, що в сукупності забезпечують надійність та функціональну відповідність програмних продуктів.

РОЗДІЛ 2. ПРАКТИЧНЕ ВИКОРИСТАННЯ МЕТРИЧНОГО ПІДХОДУ ДЛЯ ВИМІРЮВАННЯ ПАРАМЕТРІВ ПЗ

2.1. Оцінювання програмного коду за допомогою метричного апарату Холстеда

Використання метричного підходу М. Холстеда дозволяє певною мірою нівелювати недоліки, притаманні оцінюванню за кількістю рядків коду, оскільки цей метод базується на лексичному складі програми незалежно від індивідуального стилю її оформлення розробником. Розрахунки за цією методикою базуються на таких параметрах:

n1 — кількість унікальних операторів;

n2 — кількість унікальних операндів;

N1 — загальна кількість використаних операторів;

N2 — загальна кількість використаних операндів.

На основі вказаних дескрипторів визначаються такі характеристики, як словник програми (n):

$$n = n1 + n2 \tag{2.1}$$

де n1 — кількість унікальних операторів;

n2 — кількість унікальних операндів.

Довжина програми (N) розраховується за формулою:

$$N = N1 + N2 \tag{2.2}$$

де N1 — загальна кількість операторів;

N2 — загальна кількість операндів.

Обсяг програми (V) обчислюється наступним чином:

$$V = (N1+N2) * \log2(n1+n2) \tag{2.3}$$

де n1, n2, N1, N2 — відповідні параметри словника та довжини.

Додатково Холстед вводить концепцію теоретичного словника (n*), що відображає мінімально необхідний лексичний набір для реалізації алгоритму. На основі цього показника визначається теоретичний обсяг (V’):

$$V’ = n * \log 2n \quad (2.4)$$

де n — теоретичний словник програми.

Теоретична довжина програми (N') визначається за залежністю:

$$N'= n1 * \log2 (n1) + n2 * \log2(n2) \quad (2.5)$$

де n1, n2 — кількість унікальних операторів та операндів.

Рівень якості програмування (L):

$$L = V'/V \quad (2.6)$$

де V’ — теоретичний обсяг;

V — фактичний обсяг програми.

Апроксимований рівень якості (L'), що базується на параметрах реального коду:

$$L' = (2 * n2) / (n1 * N2) \quad (2.7)$$

де n1, n2 — унікальні оператори та операнди;

N2 — загальна кількість операндів.

Складність у розумінні програми (EC):

$$EC = (V) / (L') * 2 \quad (2.8)$$

де V — обсяг програми; L' — апроксимований рівень якості.

Трудомісткість кодування (D):

$$D = 1 / L' \quad (2.9)$$

де L' — апроксимований рівень якості.

Інформаційний зміст (I):

$$I = V / D \quad (2.10)$$

де V — обсяг програми;

D — складність програми.

Оцінка інтелектуальних зусиль (E):

$$E = N' * \log2 (n / L) \quad (2.11)$$

де N' — теоретична довжина;

n — словник програми;

L — рівень якості.

Для проведення аналізу було використано програмні коди з додатків: лінійна програма (Додаток А), програма з вказівниками (Додаток Б) та модульна програма (Додаток В). Словники програм (Додатки Г, Д, Е) розраховано за формулою (2.1): n1 = 52 (для лінійної), n2 = 55 (з вказівниками), n3 = 51 (модульна). Найвищий показник словника виявлено у програмі з вказівниками.

Довжина програм (N) за формулою (2.2) склала:

N1 = 222;

N2 = 237;

N3 = 298.

Максимальну довжину має модульна програма (N3 = 298). Теоретична довжина (N') за виразом (2.5) становить:

$$N'1 = 43 * \log2(43) + 9 * \log2(9) = 261,59 \quad (2.12)$$

$$N'2 = 46 * \log2(46) + 9 * \log2(9) = 282,45 \quad (2.13)$$

$$N'3 = 28 * \log2(28) + 23 * \log2(23) = 238,96 \quad (2.14)$$

Найбільше значення відповідає програмі із вказівниками (N'2 = 282,45). Інформаційний обсяг (V) згідно з формулою (2.3):

$$V1 = 222 * \log2(52) = 1265,4 \quad (2.15)$$

$$V2 = 237 * \log2(55) = 1370,097 \quad (2.16)$$

$$V3 = 298 * \log2(51) = 1690,256 \quad (2.17)$$

Найбільший обсяг притаманний модульній програмі (V3 = 1690,256). Теоретичний обсяг (V’) за формулою (2.4):

V’1 = 134,4;

V’2 = 179,515;

V’3 = 310,77.

Максимальне значення демонструє модульна програма (V’3 = 310,77).

Рівень якості (L) за виразом (2.6):

L1 = 0,1;

L2 = 0,13;

L3 = 0,18.

Найвищий рівень зафіксовано у модульної програми. Апроксимований рівень якості (L') за формулою (2.7):

L’1 = 0,01;

L’2 = 0,009;

L’3 = 0,02.

Найнижчий показник виявлено у коду з вказівниками.

Складність (D) за виразом (2.9):

D1 = 10;

D2 = 7,69;

D3 = 5,55.

Найбільш складною є лінійна програма (D1 = 10). Інформаційний зміст (I) за формулою (2.10):

I1 = 126,54;

I2 = 178,16;

I3 = 304,55.

Найбільш насиченою інтелектуальним змістом є модульна програма (I3 = 304,55).

Оцінка інтелектуальних зусиль (E) за формулою (2.11) становить:

E1 = 2354,3;

E2 = 2462,8;

E3 = 1947,5.

Таким чином, найменших зусиль потребує реалізація модульної програми (E3 = 1947,5).

Складність програми (D) за формулою (2.9):

D 1 = 10;

D 2 = 7,69;

D 3 = 5,55.

Згідно з отриманими даними, найбільш складною для сприйняття виявилася D 1 = 10 лінійна програма.

Інформаційний зміст (I) згідно з формулою (2.10):

I1 = 1265,4 / 10 = 126,54;

I2 = 1370,09 / 7,69 = 178,16;

I3 = 1690,256 / 5,55 = 304,55.

Найбільш насиченою інтелектуальним змістом є програма з модулями I3=304,55.

2.2. Дослідження структурної складності алгоритмів за методом Маккейба

Окрему категорію оціночних засобів складають метрики, що базуються на аналізі топології потоків керування програмою. Дані методи дають змогу оцінити складність логічної структури через аналіз щільності та взаємозв’язків керуючих переходів всередині програмного коду. Традиційним підходом у межах цього аналізу є подання програми у вигляді орієнтованого керуючого графа G(V, E), де множина вершин V відображає оператори, а множина дуг E — логічні переходи між ними. У кожній дузі (U, V) вершина V розглядається як вихідна, а U — як кінцева.

Для формалізації структури коду використовується потоковий граф, характерними особливостями якого є:

- побудова графа шляхом прямого відображення керуючих структур алгоритму;
- трактування закриваючих конструкцій умовних операторів та циклів як окремих фіктивних операторів;
- відповідність вершин графа лінійним фрагментам програми;
- відображення дугами передачі керування між відповідними операційними блоками;
- диференціація вузлів на операторні та предикатні.

Методика графічного аналізу, запропонована Т. Маккейбом, використовує цикломатичну складність як ключовий індикатор, що визначає трудовитрати на тестування та супровід продукту. Обчислення цикломатичної складності на основі топологічних характеристик графа здійснюється за формулою:

$$Z(G) = m - n + 2 \quad (2.12)$$

де m — кількість дуг; n — кількість вузлів у потоковому графі.

Також розрахунок може бути виконаний через кількість предикатних вузлів:

$$Z(G) = p + 1 \quad (2.13)$$

де p — кількість предикатних вузлів у потоковому графі G.

У загальному вигляді для графів з декількома компонентами зв’язності використовується формула:

$$Z(G) = m - n + 2p \quad (2.14)$$

де m — кількість дуг орієнтованого графа G; n — кількість вершин; p — кількість компонентів зв’язності графа.

Цикломатичне число Маккейба фактично визначає кількість незалежних контурів у повнозв’язному графі. Для орієнтованого графа (Додаток Ж) розрахунок цикломатичної складності за виразом (2.12) показав наступне:

$$Z(G) = 20 - 14 + 2 = 8 \quad (2.15)$$

Перевірка графа G на кількість предикатних вузлів (P = 7) згідно з формулою (2.13) підтверджує результат:

$$Z(G) = 7 + 1 = 8 \quad (2.16)$$

Аналогічно розрахунок за формулою (2.14) дає такий самий показник:

$$Z(G) = 20 - 14 + 2 * 1 = 8 \quad (2.17)$$

При аналізі складності програм застосовується порогове значення: якщо Z > 10, структура вважається надмірно складною. У розглянутому випадку показник Z = 8 свідчить про те, що програма має прийнятну складність і не потребує рефакторингу.

2.3. Аналіз складності програмних систем на основі альтернативних метрик (Чепіна та Т. Джілбі)

2.3.1. Особливості та функціональне призначення метрики Чепіна

Методологія, запропонована Чепіним, базується на оцінюванні інформаційної міцності програмного модуля шляхом дослідження функціонального призначення змінних, що входять до списку вводу-виводу.

Аналіз передбачає диференціацію всіх ідентифікаторів на чотири основні категорії залежно від характеру їх використання:

P (Input variables) — змінні, що вводяться для виконання обчислень або забезпечення процедур виведення результатів;

M (Modified variables) — ідентифікатори, що підлягають модифікації або ініціалізуються безпосередньо всередині програмного блоку;

C (Control variables) — змінні, що керують логікою виконання модуля (керуючі конструкції);

T (Parasitic variables) — змінні, що оголошені, але фактично не задіяні в алгоритмі.

Оскільки одна й та сама змінна може виконувати декілька функцій, її враховують у кожній відповідній категорії[7]. Підсумкова оцінка складності розраховується за формулою:

$$Q = a1 * P + a2 * M + a3 * C + a4 * T \quad (2.18)$$

де a1, a2, a3, a4 — коефіцієнти ваги[8].

Найбільш вагомою є категорія C (a3 = 3), для інших груп встановлено значення: a1 = 1; a2 = 2; a4 = 0,5[9]. З урахуванням ваг формула набуває вигляду:

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

Для досліджуваних програмних реалізацій було визначено кількісний склад змінних (табл. 2.1).

Таблиця 2.1

Змінні значення для розрахунку метрики Чепіна

	Лінійна програма	**Вказівники для програм**	**Модулями для програма**
P	4	4	4
M	3	3	3
C	1	1	1
T	0	0	0

На основі цих даних проведено розрахунок метрики Чепіна за формулою (2.19):

Q1 = 4 + 2 * 3 + 3 * 1 + 0,5 * 0 = 13 (для лінійної програми);

Q2 = 4 + 2 * 3 + 3 * 1 + 0,5 * 0 = 13 (для програми з вказівниками);

Q3 = 4 + 2 * 3 + 3 * 1 + 0,5 * 0 = 13 (для модульної програми).

2.3.2. Методика обчислення показників складності за Т. Джілбі

Метрика Тома Джілбі характеризує складність програмного забезпечення через насиченість коду конструкціями розгалуження та циклами. Логічна складність у межах цієї моделі визначається інтенсивністю використання виразів типу IF-THEN-ELSE. Аналіз передбачає обчислення відносної складності (cl):

$$cl = CL / N \quad (2.20)$$

де CL — абсолютна складність (кількість операторів умови та циклів);

N — загальна кількість операторів у програмі17.

Для досліджуваних фрагментів коду було встановлено сталу кількість керуючих структур:

CL1 = 7,

CL2 = 7,

CL3 = 7.

Проте через різний обсяг операторів значення відносної складності за формулою (2.20) різняться:

$$cl1 = 7 / 34 = 0,20 \quad (2.21)$$

$$cl2 = 7 / 38 = 0,18 \quad (2.22)$$

$$cl3 = 7 / 52 = 0,13 \quad (2.23)$$

де cl1, cl2, cl3 — показники відносної складності для відповідних типів програм.

Результати свідчать, що найвищий рівень відносної логічної складності притаманний лінійній програмі (cl1 = 0,20), тоді як модульна організація коду демонструє найкращу оптимізацію.

РОЗДІЛ 3. ТЕХНОЛОГІЇ ТА ІНСТРУМЕНТИ АВТОМАТИЗАЦІЇ ПРОЦЕСІВ ОЦІНЮВАННЯ ЯКОСТІ

3.1. Концептуальні засади та передумови впровадження автоматизації в тестування

Впровадження автоматизованого тестування та застосування відповідних програмних інструментів є фундаментальним складником сучасного життєвого циклу розроблення програмного забезпечення. Під автоматизованими тестами розуміють спеціалізовані скрипти або програмні модулі, що зініціюються автономно з метою верифікації функціональної придатності та якісних показників системи. Їх використання спрямоване на досягнення високих стандартів надійності продукту, підвищення загальної продуктивності команди та інтенсифікацію процесів контролю якості на всіх етапах виробництва.

Серед ключових переваг впровадження автоматизації у виробничий процес слід виокремити такі фундаментальні чинники як ефективність, відтворюваність та повнота покриття. Ефективність проявляється у тому, що автономні сценарії забезпечують значно вищу швидкість обробки даних порівняно з мануальними методами, що дозволяє суттєво оптимізувати часові та інтелектуальні ресурси проєкту. Відтворюваність гарантує можливість багаторазового запуску тестів з ідентичними вхідними параметрами без ризику втрати точності, що є критично важливим за умов ітераційних випусків програмного продукту.

Повнота покриття досягається завдяки здатності автоматизованих засобів опрацьовувати значні обсяги сценаріїв та масиви даних, що сприяє глибшій ідентифікації прихованих дефектів. Важливою перевагою є стабільність результатів: тестування здійснюється суворо відповідно до заздалегідь визначених алгоритмів, що повністю нівелює вплив людського фактору на об'єктивність оцінювання. Крім того, автономність дозволяє реалізовувати стратегію «нічного тестування», коли скрипти виконуються без безпосередньої

участі фахівця, забезпечуючи безперервність циклу перевірки та оперативність виявлення дефектів у коді.

Водночас необхідно враховувати, що автоматизація не є універсальним інструментом для всіх видів перевірок. В окремих сценаріях ручне тестування залишається більш доцільним та результативним методом оцінювання. Ефективність автоматизованих засобів прямо корелює з рівнем стратегічного планування та методологічно коректним використанням обраних інструментів. Раціональне управління тестовими даними та підтримка стабільного робочого середовища є базовими аспектами успішної інтеграції автоматизації. Слід розуміти, що програмні скрипти не здатні повноцінно замінити креативну складову мануального аналізу, зокрема в аспектах оцінювання користувацького досвіду (UX) або верифікації принципово нових функціональних блоків.

Реалізація стратегії автоматизації вимагає належної фахової підготовки команди розробників та інженерів з контролю якості. Необхідною умовою є регулярне оновлення тестової бази для забезпечення її актуальності та повної відповідності поточним змінам в архітектурі системи. Тільки за умови постійної адаптації скриптів до нових функціональних можливостей можна досягти стабільної роботи системи контролю.

Отже, інтеграція автоматизованих рішень здатна істотно спростити процедури верифікації, скоротити часові витрати на корекцію дефектів та підвищити загальні якісні параметри програмного продукту. Проте результативність таких заходів безпосередньо зумовлена виваженим стратегічним плануванням та адаптацією обраного інструментарію до специфічних вимог конкретного проєкту.

Належна підтримка інфраструктури для автоматизації також є критичним фактором, оскільки вона гарантує стабільність та достовірність отриманих результатів. Формування структурованих аналітичних звітів та постійний моніторинг результатів виконання дозволяють оперативно вирішувати виявлені технічні суперечності та мінімізувати ризики пропуску критичних помилок.

Варто наголосити на необхідності перманентної оптимізації та вдосконалення тестових сценаріїв. Такий підхід дозволяє запобігти надмірним витратам ресурсів на підтримку та виконання застарілих або неефективних скриптів, які втратили свою актуальність у процесі еволюції продукту. Регулярний рефакторинг тестового коду є обов’язковою умовою підтримки високої швидкості тестування.

Загалом, методологічно обґрунтована автоматизація стає невід’ємним елементом сучасного виробничого циклу, що сприяє прискоренню випуску нових функцій при збереженні високих стандартів надійності. Подальший розвиток цього напряму передбачає тісну інтеграцію автоматизованих перевірок з екосистемами безперервної інтеграції та доставки (CI/CD). Це дозволяє активувати перевірки при кожній модифікації коду, що забезпечує ранню ідентифікацію помилок та прискорює реалізаційну політику компанії.

Впровадження інноваційних підходів, зокрема на основі машинного навчання, здатне суттєво розширити аналітичні можливості автоматизованих перевірок. Використання інтелектуальних алгоритмів дозволяє моделювати складні сценарії для дефектування неочевидних помилок, які складно виявити стандартними методами. Це забезпечує глибший аналіз поведінки системи та адаптацію тестових скриптів до динамічних змін інтерфейсу.

Окрему увагу слід приділити автоматизації контролю безпеки, що є критичним для захисту даних користувачів. Використання спеціалізованих засобів дозволяє ідентифікувати потенційні вразливості та усувати загрози ще на стадіях проєктування та активної розробки. Такий підхід значно знижує ризики експлуатації продукту та запобігає виникненню критичних помилок у системі безпеки на ранніх етапах.

Сукупність цих факторів забезпечує формування цілісної та надійної системи автоматизованого контролю. Автоматизовані тести гарантують об'єктивність та ідентичність умов при кожному запуску, повністю мінімізуючи суб'єктивний вплив фахівця на результати випробувань. Це створює стабільний

фундамент для верифікації якісних характеристик програмного забезпечення незалежно від людського фактору.

Важливим вектором розвитку є забезпечення масштабованості тестів для різних технологічних платформ. Автоматизація дозволяє оперативно проводити крос-платформне тестування, охоплюючи різні операційні системи, типи пристроїв та браузери. Така універсальність є визначальною для забезпечення коректної роботи продукту серед широкої цільової аудиторії та збереження конкурентоспроможності на ринку.

Фінальним етапом циклу є генерація розгорнутої звітності та детальних логів, які допомагають технічній команді швидко локалізувати та нівелювати виявлені проблеми. Наявність прозорих аналітичних даних позитивно впливає на загальну стабільність системи та суттєво скорочує час, необхідний на рефакторинг коду.

Також автоматизація є оптимальним рішенням для регресійного тестування. Це дозволяє переконатися, що нові функціональні надбудови не порушують роботу існуючих блоків системи, що суттєво знижує ймовірність деградації якості коду при інтенсивному розвитку продукту. Таким чином, методологічно коректна автоматизація виступає ключовим чинником сучасного софтверного виробництва, забезпечуючи швидку адаптацію та підтримку високонадійних програмних систем.

Загалом, автоматизовані сценарії є незамінним інструментом для розробників, що забезпечує високу швидкість обробки даних, стабільність програмного середовища та суттєво спрощує процеси розгортання систем. Позитивний ефект автоматизації поширюється на загальну продуктивність команди та сприяє значному скороченню циклів випуску програмного забезпечення (Time-to-Market). Своєчасна ідентифікація дефектів на ранніх етапах дозволяє мінімізувати часові та фінансові витрати на їх подальше усунення.

Додатково автоматизація полегшує перевірку сумісності продукту з різними технологічними стеками. Можливість оперативного виконання тестів у

варіативних конфігураціях підвищує впевненість у стабільності функціонування системи в диференційованих середовищах експлуатації. Вказані аспекти визначають автоматизоване тестування як критичний елемент життєвого циклу ПЗ, що гармонійно інтегрує показники якості, безпеки та ресурсної ефективності.

Такий підхід дозволяє оперативно реагувати на архітектурні зміни та функціональні доповнення, оскільки тестові сценарії можуть бути швидко модифіковані відповідно до нових вимог. Така гнучкість є стратегічно важливою у динамічних умовах сучасного ринку програмних рішень. Окрім цього, автоматизація суттєво знижує ймовірність допущення випадкових помилок, зумовлених людським фактором, оскільки алгоритми виконують завдання з незмінною точністю при кожній ітерації.

Це сприяє виявленню дефектів на стадіях, коли їх виправлення потребує мінімальних ресурсних витрат, що відповідає концепції раннього тестування. Отже, автоматизоване тестування трактується як високоефективний метод вдосконалення процесів розробки, що гарантує надійність та якість протягом усього життєвого циклу. Використання автоматизації також полегшує масштабування перевірок на різні платформи, що є критично важливим для продуктів, які функціонують у гетерогенних середовищах.

Окремо слід виділити можливість проведення комплексних випробувань у неробочий час без безпосереднього залучення персоналу. Це дозволяє оптимально використовувати наявні обчислювальні ресурси та інтенсифікує графік розробки, забезпечуючи команду актуальними результатами перевірок на початок кожного робочого дня.

Загалом, впровадження автоматизованих рішень є стратегічно обґрунтованим кроком для забезпечення високих стандартів якості та підвищення загальної рентабельності процесу розроблення програмного забезпечення. Проте варто усвідомлювати, що автоматизація не є універсальною панацеєю для всіх технічних викликів. Певні сегменти, зокрема тестування складних інтерфейсів користувача або специфічні аспекти естетичного дизайну,

досі вимагають залучення ручних методів контролю для отримання адекватної оцінки зручності експлуатації.

Також необхідно враховувати витрати на розроблення та подальшу підтримку скриптів, які можуть бути значними на початкових фазах проєкту. Однак у довгостроковій перспективі ці інвестиції повністю нівелюються за рахунок суттєвого зростання ефективності, швидкості виконання та глибини тестування. У підсумку, результативне використання автоматизації потребує ретельного стратегічного планування, раціонального вибору інструментарію та постійної актуалізації тестової бази відповідно до динаміки проєкту.

Максимальна віддача від автоматизації досягається за умови безперервного моніторингу змін у ПЗ та адаптації сценаріїв до нових функціональних вимог. Тісна синергія між розробниками та інженерами з тестування є запорукою вчасного виявлення та нейтралізації технічних проблем. Додатково автоматизовані засоби стають невід'ємною частиною стратегій безперервної інтеграції, що дозволяє мінімізувати ризики при впровадженні нових компонентів у стабільну систему.

У масштабних проєктах із великими обсягами коду автоматизація перетворюється на ключовий елемент загальної стратегії забезпечення якості. При цьому важливо пам'ятати, що її ефективність безпосередньо залежить від системного управління та грамотного планування архітектури тестів. Для оптимізації процесів необхідно чітко визначити критерії відбору сценаріїв для автоматизації, надаючи пріоритет повторюваним випадкам зі стабільною бізнес-логікою та високою частотою виконання.

У процесі експлуатації автоматизованих систем особливу увагу слід приділяти їхній стабільності, оскільки будь-які зміни в основному коді можуть критично вплинути на результативність тестів. Глибоке розуміння принципів автоматизації та методично правильний вибір технологічного стеку є базовою запорукою успіху при створенні надійних та відтворюваних систем контролю якості.

Для підтримки актуальності тестів слід забезпечити їхню регулярну ревізію та адаптацію відповідно до еволюції програмного продукту. Це включає систематичне оновлення скриптів, корекцію логічних помилок та врахування нових бізнес-вимог, що виникають у процесі розроблення. Важливо наголосити, що автоматизація не замінює мануальний підхід, а гармонійно доповнює його. Ручне тестування залишається критичним для аналізу аспектів, які важко формалізувати, наприклад, емоційне сприйняття інтерфейсу користувачем або оцінювання складної візуальної естетики продукту.

За умови системного впровадження автоматизації можна очікувати суттєвого скорочення часових витрат на регресійні перевірки, підвищення прецизійності виявлення помилок та досягнення високої якості фінального програмного забезпечення. Ключовим етапом цього процесу є аналіз та інтерпретація отриманих результатів. При виявленні аномалій необхідно проводити детальний розбір причин їх виникнення та вносити відповідні корективи як у вихідний код системи, так і безпосередньо в тестові сценарії.

Для забезпечення високої ефективності процесів доцільно використовувати системи контролю версій для зберігання та відстеження змін у тестовому коді. Це гарантує цілісність тестової бази, можливість паралельної роботи декількох інженерів та швидкого відкату до попередніх стабільних станів у разі виникнення збоїв. Узагальнюючи, автоматизація тестування потребує комплексного методологічного підходу, де постійне вдосконалення скриптів, безперервний моніторинг результативності та злагоджена комунікація всередині команди є визначальними чинниками забезпечення якості.

Для підтримки стабільності тестів необхідно враховувати варіативність вхідних даних. Використання сучасних технік параметризації дозволяє суттєво розширити тестове покриття, використовуючи один і той самий сценарій для обробки різних наборів даних. Інтеграція цих процесів у CI/CD конвеєри дозволяє автоматизувати запуск перевірок при кожній модифікації коду, що забезпечує розробникам швидкий зворотний зв’язок та мінімізує ризик потрапляння дефектів у релізну версію. Загалом, успішне впровадження

автоматизованих методів вимагає постійної адаптації до технологічних змін та активної співпраці всієї команди розробки.

Значну увагу слід приділяти підтримці та розвитку тестової інфраструктури. Це охоплює не лише технічне забезпечення, а й комплексне налаштування систем моніторингу, звітності та централізованого зберігання результатів для подальшої ретроспективної аналітики. Паралельне виконання тестів на диференційованих конфігураціях дозволяє суттєво розширити тестове покриття, а використання хмарних технологій відкриває додаткові можливості для крос-платформної автоматизації без необхідності утримання значного парку власного обладнання.

Ведення чіткої та структурованої документації щодо архітектури та призначення тестів є обов’язковою умовою, що полегшує онбординг нових фахівців та забезпечує прозорість процесів підтримки коду. Успішна експлуатація автоматизованих систем передбачає постійне вдосконалення їхньої внутрішньої структури та повну інтеграцію в загальний ландшафт розробки. Зі зростанням складності проєкту критично важливо оптимізувати стратегію вибору сценаріїв, фокусуючись на найбільш вразливих ділянках системи, що дозволяє раціонально використовувати ресурси та прискорювати отримання результатів.

Підтримка механізмів автоматичного аналізу логів та трасування винятків суттєво спрощує процес налагодження системи та локалізації дефектів. Інтеграція в CI/CD ландшафт забезпечує автоматичну верифікацію коду при кожному оновленні, що гарантує поставку виключно стабільних версій продукту кінцевому користувачу. Усі ці фактори у сукупності сприяють підвищенню надійності та результативності автоматизованого контролю якості. При цьому необхідно постійно розширювати перелік конфігурацій та середовищ, гарантуючи коректну роботу продукту в різноманітних умовах експлуатації.

Автоматизація збору метрик та генерації аналітичних звітів дозволяє відстежувати статус дефектів протягом усього циклу розробки, забезпечуючи повну прозорість показників якості. Адаптація стратегій тестування до бізнес-

вимог та функціональних змін є ключовим етапом у забезпеченні актуальності контрольних заходів. Постійне вдосконалення методів автоматизації є невід’ємною частиною успішного розроблення ПЗ у сучасному динамічному середовищі. Слід також інтегрувати спеціалізовані інструменти безпеки (DevSecOps), що дозволяє запобігати потенційним атакам та захищати конфіденційні дані користувачів на ранніх стадіях виробництва.

Розвиток практик тестування на витривалість забезпечує стабільність ПЗ при тривалих навантаженнях у реальних умовах, включаючи аналіз великих масивів даних та моделювання стресових сценаріїв. Налаштування ефективних CI/CD процесів прискорює цикл випуску нових функцій, забезпечуючи миттєву реакцію на виявлені помилки. Отже, комплексне впровадження автоматизації, фокус на надійності та безпеці, а також використання сучасних методологій є запорукою успішного функціонування програмних систем.

Необхідно активно розвивати засоби юніт-тестування, інтеграційні перевірки та тести на відповідність бізнес-вимогам. Повністю автоматизовані набори тестів сприяють оперативності впровадження змін та мінімізують ризики деградації коду. Питання безпеки та регулярний аудит вразливостей мають бути невід'ємною частиною загальної стратегії контролю якості для надійного запобігання кіберзагрозам. Окрім технічних перевірок, слід впроваджувати тести на рівні користувацького інтерфейсу (UI Testing), що дозволяє ідентифікувати проблеми взаємодії у реальних сценаріях використання. Важливим є постійне підвищення кваліфікації команди та впровадження культури DevOps, що сприяє синергії розробки та тестування, забезпечуючи безперервне покращення якості програмного забезпечення.

3.2. Класифікація сучасних типів автоматизованих тестів

Автоматизація тестування охоплює широкий спектр типів незалежних перевірок, кожна з яких орієнтована на дослідження специфічних аспектів програмного продукту. Систематизація цих перевірок дозволяє сформувати багаторівневу структуру контролю якості, що охоплює систему від окремих

рядків коду до комплексних користувацьких сценаріїв. До фундаментальних категорій автоматизованих тестів належать модульні, інтеграційні, функціональні та спеціалізовані типи тестування.

Юніт-тести (модульне тестування) спрямовані на верифікацію коректності функціонування мінімальних програмних одиниць, компонентів або окремих модулів у повній ізоляції. Головним цільовим призначенням цього виду є ідентифікація логічних помилок на початкових етапах розроблення та підтвердження цілісності внутрішньої архітектури модулів, що дозволяє виявляти дефекти ще до моменту їх потрапляння у загальну збірку системи.

Наступним рівнем є інтеграційні тести, що зосереджені на дослідженні механізмів взаємодії та обміну даними між різними компонентами або програмними підсистемами. Вони дозволяють виявляти дефекти, що виникають саме при поєднанні модулів, та перевіряти коректність роботи міжкомпонентних інтерфейсів, забезпечуючи цілісність складних архітектурних рішень.

Функціональні тести забезпечують контроль відповідності поведінки системи встановленим вимогам та специфікаціям. Їх впровадження необхідне для підтвердження того, що програмне забезпечення повною мірою реалізує завдання, визначені замовником, та коректно реагує на варіативні вхідні дані в межах робочих сценаріїв. Паралельно з цим застосовуються приймальні та регресійні тести, які аналізують стабільність системи після внесення архітектурних змін або виправлення дефектів, гарантуючи якість при випуску оновлень та запобігаючи повторній появі помилок у раніше стабільному коді.

Спеціалізованим вектором є тести безпеки та тести взаємодії з користувачем. Тестування безпеки включає перевірки, спрямовані на дефектування вразливостей та оцінювання стійкості системи до зовнішніх загроз, що формує надійний захисний контур продукту. У свою чергу, аналіз реакції системи на дії користувача через графічну оболонку (UI/GUI) дозволяє верифікувати коректність інтерфейсних логік та забезпечити високу зручність експлуатації продукту.

Для досягнення цільових показників надійності впроваджуються також навантажувальні перевірки та тести на відновлюваність, які дозволяють ідентифікувати потенційні слабкі місця та архітектурних прогалин при роботі в екстремальних умовах. Систематичне та диференційоване використання всіх зазначених категорій є запорукою високої якості програмного забезпечення та оперативного реагування на сучасні технічні виклики.

Комплексна стратегія автоматизації також передбачає обов'язкове тестування сумісності, що фокусується на забезпеченні коректності роботи програмного забезпечення у гетерогенних середовищах. Тести локалізації гарантують адаптивність продукту до мовних та регіональних стандартів, що є критичним для глобального розповсюдження систем. Важливим етапом перевірки є аналіз стійкості, що демонструє здатність системи оперативно відновлювати свою функціональність після критичних технічних збоїв. Синхронне використання цих методів гарантує багатогранну надійність програмного продукту та його готовність до експлуатації в реальних умовах.

Окрему увагу в межах автоматизації приділяють регресійним тестам, які мінімізують ризики деградації якості після модифікації вихідного коду або додавання нових функцій. Тести взаємодії аналізують синергію системи із зовнішніми сервісами та сторонніми компонентами, перевіряючи цілісність екосистеми. Одночасно з цим тестування продуктивності дає змогу оцінити швидкість та загальну ефективність обробки операцій при різних рівнях інтенсивності запитів, що дозволяє виявити «вузькі місця» в архітектурі.

Паралельно з суто технічними перевірками вагоме значення мають автоматизовані тести користувацького досвіду (UI/UX). Вони дозволяють підтвердити відповідність графічного інтерфейсу вимогам ергономіки, естетики та інтуїтивності, що є визначальним фактором для задоволеності кінцевого споживача. У межах автоматизованих циклів контроль кібербезпеки дозволяє превентивно виявляти критичні вразливості та забезпечувати стабільний захист від кібератак ще на етапі розроблення.

Тестування сумісності підтверджує повну працездатність програми на диференційованих апаратних платформах, що суттєво розширює потенційну базу користувачів. Слід також відзначити роль автоматизованих тестів у підтримці стратегій безперервної інтеграції та доставки (CI/CD), що значно прискорює впровадження інновацій та операційне усунення виявлених помилок. Використання широкої палітри тестів корелює з передовими практиками софтверної інженерії, сприяючи створенню надійних, безпечних та конкурентоспроможних програмних продуктів.

На завершення, автоматизація може включати тестування на відмовостійкість, що моделює поведінку системи у кризових ситуаціях для оцінювання її здатності до самовідновлення. Продуктивні випробування аналізують роботу системи з масивами великих даних, гарантуючи її масштабованість та стабільність під високим навантаженням. Ключовим залишається контроль інтерфейсу для забезпечення задоволеності користувачів, а тести відхилень дозволяють чітко відстежувати вплив будь-яких виправлень на загальний стан продукту. Отже, інтеграція варіативних видів автоматизованих перевірок забезпечує стабільно високий рівень якості та ефективності програмного забезпечення, інтенсифікуючи процеси його розроблення та супроводу.

3.3. Сучасний інструментарій автоматизації процесів тестування

Реалізація стратегії автоматизованого тестування базується на використанні широкого спектра програмних засобів, що забезпечують інтенсифікацію та результативність контрольних процесів. Вибір конкретного інструментарію зумовлений технологічним стеком проєкту, архітектурними особливостями системи та специфікою бізнес-вимог. До ключового інструментарію сучасної розробки належать такі рішення:

Selenium виступає провідним засобом для автоматизації перевірок веб-орієнтованих систем. Він дозволяє проектувати та виконувати складні сценарії взаємодії з браузером, підтримуючи широкий перелік мов програмування та

операційних середовищ. Для сегмента мобільних додатків на базі платформ Android та iOS спеціалізованим рішенням є Appium, який забезпечує крос-платформну автоматизацію без необхідності модифікації вихідного коду програми.

Структурування та виконання тестових сценаріїв, зокрема для систем на мові Java, забезпечується фреймворками JUnit та TestNG. Вони пропонують потужні механізми анотацій та звітності, що спрощує підтримку великих тестових наборів. Платформа Jenkins відіграє критичну роль в автоматизації процесів безперервної інтеграції та доставки (CI/CD), ініціюючи виконання тестів при кожній зміні коду та забезпечуючи швидкий зворотний зв’язок.

Для верифікації прикладних програмних інтерфейсів ефективно використовується Postman. Цей інструмент дозволяє перевіряти коректність обміну даними між сервісами, аналізувати HTTP-запити та автоматизувати тестування API на різних рівнях. Cucumber, у свою чергу, реалізує концепцію тестування на основі специфікацій природною мовою (BDD), що сприяє продуктивному порозумінню між розробниками, тестувальниками та бізнес-аналітиками.

Оцінка продуктивності та масштабованості складних архітектур здійснюється за допомогою LoadRunner. Цей інструмент дозволяє проводити високонавантажувальні випробування, моделюючи роботу тисяч одночасних користувачів. Також значного поширення набула універсальна інтегрована платформа TestComplete, яка забезпечує автоматизацію на різних рівнях, включаючи веб-застосунки, мобільні та десктопні рішення в межах єдиного середовища.

Впровадження вказаних засобів дозволяє суттєво оптимізувати процедури контролю якості, гарантуючи високу надійність програмних систем та їхню відповідність установленим стандартам. Процес верифікації програмного забезпечення традиційно поділяється на мануальний та автоматизований підходи, де останній передбачає часткову або повну заміну людської участі у рутинних та трудомістких операціях.

Сучасні технології автоматизації класифікуються за методологічними підходами до побудови сценаріїв. До них належать приватні архітектурні рішення, розроблені під конкретні потреби проєкту, а також тестування на основі даних (Data Driven Testing — DDT), що дозволяє використовувати зовнішні джерела інформації для ініціації перевірок. Важливе місце посідає тестування під керуванням ключовими словами (Keyword Driven Testing — KDT) та використання комплексних тестових фреймворків, які структурують процес розробки скриптів. Окремо виділяють технологію запису та відтворення сценаріїв (Record & Playback) та поведінкове тестування (Behaviour Driven Testing — BDT/BBT), що фокусується на функціональній логіці з позиції користувача.

Варто зауважити, що окремі інструменти можуть ефективно комбінувати кілька підходів. Наприклад, Selenium IDE поєднує механізм запису-відтворення з моделлю KDT, що робить його гнучким інструментом для швидкої розробки. Використання готових фреймворків суттєво полегшує адаптацію фахівців до робочих процесів, а методики запису прискорюють створення первинних сценаріїв навіть за умов обмеженого досвіду команди в автоматизації.

Екосистема Selenium, як один із найпоширеніших інструментів, включає також Selenium WebDriver та Selenium Server. WebDriver являє собою набір спеціалізованих бібліотек для програмного керування браузерами за допомогою різних мов програмування, забезпечуючи пряму взаємодію з елементами інтерфейсу. Selenium Server надає можливість віддаленого виконання команд, а конфігурація Selenium Grid дозволяє масштабувати тестове середовище шляхом створення розподілених мереж, що забезпечує одночасне виконання тестів у різних браузерах та операційних системах.

Командний інтерфейс Selenium диференціюється на три функціональні категорії, що визначають логіку взаємодії зі сценарієм. Перша категорія — це дії (actions), які безпосередньо керують станом системи, наприклад, виконання кліку або вибір операції; важливо, що при виникненні критичної помилки під час виконання дії тест автоматично переривається. Друга категорія — зчитувачі

(accessors), які аналізують поточний стан додатка та зберігають отримані дані у змінних для подальшого використання. Третя категорія — перевірки (assertions), що виконують зіставлення фактичного стану програмного продукту з очікуваним еталоном, формуючи остаточний вердикт щодо успішності проходження тесту.

3.3. Порівняльний аналіз провідних інструментів автоматизованого контролю якості

Реалізація стратегії автоматизованого тестування базується на використанні широкого спектра програмних засобів, що забезпечують інтенсифікацію та результативність контрольних процесів. Вибір конкретного інструментарію зумовлений технологічним стеком проєкту, архітектурними особливостями системи та специфікою бізнес-вимог. До ключового інструментарію сучасної розробки належать такі рішення:

Selenium виступає провідним засобом для автоматизації перевірок веб-орієнтованих систем. Він дозволяє проектувати та виконувати складні сценарії взаємодії з браузером, підтримуючи широкий перелік мов програмування та операційних середовищ. Для сегмента мобільних додатків на базі платформ Android та iOS спеціалізованим рішенням є Appium, який забезпечує крос-платформну автоматизацію без необхідності модифікації вихідного коду програми.

Структурування та виконання тестових сценаріїв, зокрема для систем на мові Java, забезпечується фреймворками JUnit та TestNG. Вони пропонують потужні механізми анотацій та звітності, що спрощує підтримку великих тестових наборів. Платформа Jenkins відіграє критичну роль в автоматизації процесів безперервної інтеграції та доставки (CI/CD), ініціюючи виконання тестів при кожній зміні коду та забезпечуючи швидкий зворотний зв’язок.

Для верифікації прикладних програмних інтерфейсів ефективно використовується Postman. Цей інструмент дозволяє перевіряти коректність обміну даними між сервісами, аналізувати HTTP-запити та автоматизувати

тестування API на різних рівнях. Cucumber, у свою чергу, реалізує концепцію тестування на основі специфікацій природною мовою (BDD), що сприяє продуктивному порозумінню між розробниками, тестувальниками та бізнес-аналітиками.

Оцінка продуктивності та масштабованості складних архітектур здійснюється за допомогою LoadRunner. Цей інструмент дозволяє проводити високонавантажувальні випробування, моделюючи роботу тисяч одночасних користувачів. Також значного поширення набула універсальна інтегрована платформа TestComplete, яка забезпечує автоматизацію на різних рівнях, включаючи веб-застосунки, мобільні та десктопні рішення в межах єдиного середовища.

Впровадження вказаних засобів дозволяє суттєво оптимізувати процедури контролю якості, гарантуючи високу надійність програмних систем та їхню відповідність установленим стандартам. Процес верифікації програмного забезпечення традиційно поділяється на мануальний та автоматизований підходи, де останній передбачає часткову або повну заміну людської участі у рутинних та трудомістких операціях.

Сучасні технології автоматизації класифікуються за методологічними підходами до побудови сценаріїв. До них належать приватні архітектурні рішення, розроблені під конкретні потреби проєкту, а також тестування на основі даних (Data Driven Testing — DDT), що дозволяє використовувати зовнішні джерела інформації для ініціації перевірок. Важливе місце посідає тестування під керуванням ключовими словами (Keyword Driven Testing — KDT) та використання комплексних тестових фреймворків, які структурують процес розробки скриптів. Окремо виділяють технологію запису та відтворення сценаріїв (Record & Playback) та поведінкове тестування (Behaviour Driven Testing — BDT/BBT), що фокусується на функціональній логіці з позиції користувача.

Варто зауважити, що окремі інструменти можуть ефективно комбінувати кілька підходів. Наприклад, Selenium IDE поєднує механізм запису-відтворення

з моделлю KDT, що робить його гнучким інструментом для швидкої розробки. Використання готових фреймворків суттєво полегшує адаптацію фахівців до робочих процесів, а методики запису прискорюють створення первинних сценаріїв навіть за умов обмеженого досвіду команди в автоматизації.

Екосистема Selenium, як один із найпоширеніших інструментів, включає також Selenium WebDriver та Selenium Server. WebDriver являє собою набір спеціалізованих бібліотек для програмного керування браузерами за допомогою різних мов програмування, забезпечуючи пряму взаємодію з елементами інтерфейсу. Selenium Server надає можливість віддаленого виконання команд, а конфігурація Selenium Grid дозволяє масштабувати тестове середовище шляхом створення розподілених мереж, що забезпечує одночасне виконання тестів у різних браузерах та операційних системах.

Командний інтерфейс Selenium диференціюється на три функціональні категорії, що визначають логіку взаємодії зі сценарієм. Перша категорія — це дії (actions), які безпосередньо керують станом системи, наприклад, виконання кліку або вибір операції; важливо, що при виникненні критичної помилки під час виконання дії тест автоматично переривається. Друга категорія — зчитувачі (accessors), які аналізують поточний стан додатка та зберігають отримані дані у змінних для подальшого використання. Третя категорія — перевірки (assertions), що виконують зіставлення фактичного стану програмного продукту з очікуваним еталоном, формуючи остаточний вердикт щодо успішності проходження тесту.

Надсилайте наступну сторінку тексту. Я буду продовжувати з урахуванням усіх зауважень щодо змісту та оформлення рисунків.

Модульне тестування фокусується на ізольованій перевірці об'єктів та класів, зазвичай використовуючи для цього інструментарій родини XUnit. Інтеграційне тестування аналізує синергію окремих компонентів, системне — взаємодію програмного продукту з апаратним рівнем, а приймальне — повну відповідність системи встановленим бізнес-вимогам замовника.

Більшість сучасних технологій фокусуються переважно на технічних аспектах реалізації, що суттєво ускладнює перевірку високорівневих сценаріїв. Цю проблему ефективно вирішує методологія поведінкового тестування (BDD), де основний акцент зміщується на загальну працездатність системи при використанні її кінцевим користувачем. Методологія базується на логічній формулі «Given – When – Then» (Дано – Коли – Тоді), що дозволяє описувати сценарії зрозумілою для всіх учасників проєкту мовою. Прикладом успішної реалізації цього підходу є фреймворк Serenity BDD, який дозволяє не лише автоматизувати перевірки, а й генерувати наочні аналітичні звіти про результати проходження тестів (рис. 3.1, 3.2).

```
@Test
public void verifySubCategory() throws InterruptedException {
    // GIVEN
    stepsForSerenity.a_user_visits_a_page(siteHomePage);
    // WHEN
    stepsForSerenity.the_user_chooses_category_Compurer();
    // THEN
    stepsForSerenity.the_user_can_see_subcategory_ITService("IT услуги");
}
```

Рис. 3.1. Зразок коду тесту для Serenity BDD [28]

Нижче (див. рис. 3.2.) проілюстровано зразок реалізації конкретного кроку

```
@Step("Given the user visits a page {0}")
public void a_user_visits_a_page(String homePage) {
    this.siteHomePage = homePage;
}
```

з використанням інструментарію фреймворку Serenity BDD:Рис. 3.2. Фрагмент реалізації тесту засобами інструментарію Serenity BDD

Результати тестування у формі сформованого звіту представлено на рис. 3.3.

Steps	Outcome	Duration
Given the user visits a page https://pn.com.ua/	SUCCESS	0,05s
When the user chooses category Compurer	SUCCESS	37,49s
Then the user sees IT услуги subcategory	SUCCESS	1,62s
	SUCCESS	*39,51s*

Рис. 3.3. Візуалізація підсумкового звіту після завершення тестів Serenity BDD

Окрему групу в системі забезпечення якості складають методи оцінювання продуктивності, які дозволяють визначити межі працездатності системи під впливом зовнішніх факторів:

Навантажувальне тестування (load testing) передбачає перевірку стабільності та часу відгуку системи при номінальних та очікуваних пікових навантаженнях. Стрес-тестування (stress testing) спрямоване на аналіз поведінки програмного продукту при критичних змінах навантаження, що виходять за межі нормальної експлуатації. Об'ємне тестування (volume testing) фокусується на оцінці ефективності обробки системою великих масивів даних.

Важливим аспектом є перевірка масштабованості (scalability testing), яка підтверджує здатність системи до лінійного підвищення продуктивності при додаванні обчислювальних ресурсів. Конкурентне тестування (concurrency testing) дозволяє провести глибокий аналіз обробки множинних одночасних запитів та вчасно ідентифікувати конфлікти за спільні ресурси (race conditions). Окремо виділяють перевірки на надійність та відновлюваність, що засвідчують стабільність системи протягом тривалого часу та її здатність до коректного самовідновлення після виникнення технічних збоїв.

Для реалізації таких випробувань часто використовується інструментарій JMeter, архітектура якого включає групи потоків (Thread groups), контролери логіки, слухачі (Listeners) для візуалізації результатів та таймери для моделювання реальних затримок. Показник Ramp-Up Period у налаштуваннях інструмента визначає динаміку наростання навантаження на систему, а Loop count — інтенсивність циклічних повторів обраного сценарію.

У практичній частині дослідження було розглянуто програмну одиницю (процедуру), структура якої відтворює синтаксис основної програми та може містити внутрішні функції. В межах проектування було сформовано специфікації, що визначають вхідні потоки даних та ключові алгоритмічні особливості. Для оптимізації загальної кількості тестів та забезпечення максимального покриття було проведено розподіл вхідних даних на відповідні групи еквівалентності (див. табл. 3.1).

Таблиця 3.1.

Вхідні параметри для методу еквівалентного розділення

Вхідні умови	**Правильні класи еквівалентності**	**Неправильні класи Еквівалентності**
N	2<= n<=50	n<2; n>50 Символ
M	2<= m<=50	m<2; m>50 Символ
Елемент матриці	дійсне число	Символ

Для верифікації правильності роботи підпрограм розроблено тестові набори на основі специфікацій та безпосереднього аналізу коду (див. табл. 3.2.). Програма реалізована на мові C++ у середовищі MS Visual Studio та призначена для пошуку максимального середнього арифметичного в упорядкованих рядках матриці.

Таблиця 3.2.

Опис тестових сценаріїв правильних класів еквівалентності для підтвердження

№ Тесту	**№ Класу еквівалентності**	**Вхідні умови**	**Результат**
	1	3	Правильні
1	5	3	Дані
	9	7	

Детальний опис тест-кейсу для виявлення реакції програми на некоректні дані систематизовано в таблиці 3.3. Ці сценарії дозволяють перевірити стійкість програмного забезпечення та коректність роботи механізмів обробки виняткових ситуацій при введенні значень, що виходять за межі встановлених норм.

Таблиця 3.3.

Опис тестових сценаріїв неправильних класів еквівалентності для підтвердження

№ Тесту	№ Класу еквівалентності	Вхідні умови	Результат
	2	1	M and N must be >= 2
1	3	51	M and N must be < 50
	4	A	M and N must be >= 2
	6	0	M and N must be >= 2
	7	55	M and N must be < 50
	8	B	M and N must be >= 2
	10	T	Float or Integer ONLY!

Результати верифікації методами «Білого» та «Чорного» ящиків представлені в таблицях 3.4 та 3.5. Ці методи дозволяють комплексно оцінити якість програмного продукту як з точки зору внутрішньої логіки коду, так і з позиції відповідності функціональним вимогам.

Таблиця 3.4.

Оцінка результатів ефективності тестових сценаріїв «Білого ящика»

№ тесту	№ класу еквівалентності	Вхідні умови	Результат
1	**2**	**3**	**4**
1	1	3	Результат представлено в табл 5.2
	5	3	
	9	7	
2	2	0	M and N must be >= 2
3	3	220	M and N must be < 100
4	4	a	M and N must be >= 2

Продовження табл. 3.4.

1	2	3	4
5	6	b	M and N must be >= 2
6	7	115	M and N must be < 100
7	8	-3	M and N must be >= 2
8	10	pi	Float or Integer ONLY!
У випробуванні №1 виконалися всі умови модуля.			
У випробуванні №2 не виповнилося жодна умова.			
У випробуванні №3 умова 1 виповнилося, не виконуючи умови 2,3,4,5			
У випробуванні №4 не виповнилося жодна умова.			
У випробуванні №5 не виконалось жодна умова.			
У випробуванні №6 умова 1 виповнилося, не виконуючи умови 2,3,4,5			
У випробуванні №7 не виповнилося жодна умова.			
У випробуванні №8 умови 1,2 виконалися, що не виконалися умови 3,4,5			

Аналіз показав, що при введенні коректних даних (випробування №1) виконуються всі логічні гілки модуля, що підтверджує повноту тестового покриття внутрішньої структури алгоритму. У випадках використання некоректних параметрів (випробування №2–№8) система адекватно ідентифікує помилки введення, активуючи відповідні захисні механізми та виводячи інформативні повідомлення.

Це свідчить про високу якість реалізації програмного коду та надійність обробки виняткових ситуацій. Такий підхід до верифікації дозволяє гарантувати стабільну роботу програми не лише в стандартних умовах, а й при виникненні непередбачуваних помилок користувача, забезпечуючи цілісність даних та безперервність обчислювального процесу.

Таблиця 3.5.

Оцінка результатів ефективності тестових сценаріїв «Чорного ящика»

Номер тесту	**Вхідні дані**	**Очікуваний результат**	**Результат тестування**
1	Спроба ввести невірну матрицю	Програма повинна надавати помітку про помилку	Тест пройдений

Практичне дослідження підтвердило повну функціональну придатність розробленого програмного коду. Програма коректно обробляє вхідні матриці, безпомилково ідентифікує впорядковані за зростанням або спаданням рядки та підтримує стабільну інтерактивну взаємодію з користувачем. Завдяки впровадженим механізмам контролю, система забезпечує точність обчислень та логічну цілісність результатів.

Для подальшого вдосконалення показників якості програмного продукту доцільно розширити обсяг автоматизованих тестів, зосередивши увагу на аналізі граничних станів (Boundary Value Analysis). Також перспективним напрямком є поглиблення методів обробки виняткових ситуацій при роботі з надвеликими обсягами даних, що дозволить гарантувати масштабованість та високу відмовостійкість системи у складних експлуатаційних умовах.

Описане вище завершує розгляд практичної частини верифікації та тестування розробленого програмного забезпечення.

ВИСНОВКИ

За результатами проведеного дослідження теоретичних засад оцінювання якості програмного забезпечення було встановлено, що цю категорію слід розглядати як сукупність атрибутів продукту, що визначають його здатність задовольняти регламентовані або очікувані потреби замовника. Це багатогранна характеристика, яка інтегрує такі аспекти, як функціональна придатність, надійність, продуктивність та зручність експлуатації. Основні результати роботи були представлені на міжнародній науково-практичній конференції [24] (Додаток К).

У процесі формування вимог до якісних показників ПЗ та вибору методів їх оцінювання критично важливо орієнтуватися на актуальні нормативні стандарти, а також враховувати специфіку функціонування продукту в конкретній прикладній галузі. Модель якості виступає інструментом для побудови ієрархічної системи критеріїв, що всебічно характеризують програмний продукт. Згідно з положеннями стандарту ISO/IEC 25010, до базових показників якості належать: функціональність, надійність, практичність, ефективність, супроводжуваність та мобільність.

Методологічна база оцінювання охоплює використання метричних систем, методи статичного та динамічного аналізу, автоматизоване тестування, а також оцінку повноти тестового покриття. Метрика якості ПЗ трактується як формалізована міра, що дозволяє отримати чисельний індикатор певної властивості системи або її специфікацій. Отримані значення дають змогу встановити рівень відповідності архітектури та процесів встановленим нормативам.

Зокрема, цикломатичне число Маккейба визначає кількість лінійно незалежних контурів у графі програми, що корелює з обсягом випробувань, необхідних для вичерпної перевірки всіх логічних гілок. Метрика Т. Джілбі фокусується на логічній насиченості коду операторами умови та циклу. Аналіз за методом Чепіна дозволяє оцінити інформаційну міцність модуля через дослідження характеру використання змінних різних функціональних груп.

Комплексне використання метрик Холстеда дає можливість оцінити складність програмного виробу в цілому, базуючись на аналізі кількості операторів та операндів.

Додатковими інструментами контролю виступають різні види тестування, а також статичне та динамічне дослідження коду. Статичний аналіз забезпечує ідентифікацію потенційних дефектів на етапах розроблення без виконання коду, тоді як динамічний підхід дозволяє верифікувати поведінку системи у реальному часі. Аналіз покриття коду тестами є критичним для оцінки результативності тестових сценаріїв, оскільки високий рівень покриття мінімізує ймовірність критичних помилок та підвищує стабільність ПЗ. Засоби автоматизації тестування суттєво інтенсифікують процеси дефектування та усунення дефектів.

Підсумовуючи, можна стверджувати, що лише системна інтеграція метричного аналізу, сучасних технік тестування та автоматизованих засобів контролю є фундаментом для розроблення високоякісного програмного продукту, який гарантує надійність, ефективність та повну відповідність вимогам кінцевого споживача.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. ДСТУ ISO/IEC 25023:2019 (ISO/IEC 25023:2016, IDT). Інженерія систем і програмних засобів. Вимоги до якості систем програмних засобів та їхнього оцінювання (SQuaRE). Вимірювання якості систем та програмних продуктів. — [Чинний від 2019-12-01]. — Київ : ДП «УкрНДНЦ», 2019.

2. ДСТУ ISO/IEC 25022:2019 (ISO/IEC 25022:2016, IDT). Інженерія систем і програмних засобів. Вимоги до якості систем програмних засобів та їхнього оцінювання (SQuaRE). Вимірювання якості під час застосування. — [Чинний від 2019-12-01]. — Київ : ДП «УкрНДНЦ», 2019.

3. ДСТУ ISO/IEC 25010:2016 (ISO/IEC 25010:2011, IDT). Інженерія систем і програмних засобів. Вимоги до якості систем і програмних засобів та її оцінювання (SQuaRE). Моделі якості системи та програмних засобів. — [Чинний від 2016-11-01]. — Київ : ДП «УкрНДНЦ», 2016.

4. ДСТУ ISO/IEC/IEEE 12207:2018 (ISO/IEC/IEEE 12207:2017, IDT). Інженерія систем і програмних засобів. Процеси життєвого циклу програмних засобів. — [Чинний від 2018-11-01]. — Київ : ДП «УкрНДНЦ», 2018.

5. ISO/IEC/IEEE 29119 Software and systems engineering — Software testing. — URL: https://en.wikipedia.org/wiki/ISO/IEC_29119

6. Волочій Б. Ю. Технологія моделювання алгоритмів поведінки інформаційних систем. Львів : Вид-во НУ «Львівська політехніка», 2022. 220 с.

7. Говорущенко Т. О., Питлик Є. В. Визначення ефективності метрик якості на етапі проєктування програмного забезпечення. — URL: http://journals.khnu.km.ua/vestnik/pdf/tech/2012_2/42gov.pdf

8. Грицюк Ю. І., Андрущакевич О. Т. Засіб для визначення якості програмного забезпечення методами метричного аналізу. Науковий вісник НЛТУ України. 2018. Т. 28. № 6. С. 159–171. — URL: https://nv.nltu.edu.ua/Archive/2018/28_6/33.pdf

9. Lavrishcheva K. M. Prohramna inzheneriia: pidruchnyk. Kyiv: Akademperiodyka, 2018. 320 p. [In Ukrainian].

10. Колдовский В. В. Актуальні проблеми оцінки якості в управлінні програмними проектами з автоматизації економіки. — URL: http://www.economy.nayka.com.ua/?op=1&z=547

11. Поморова О. В., Говорущенко Т. О. Аналіз методів та засобів оцінки якості програмних систем. Радіоелектронні і комп'ютерні системи. 2009. № 6(40). С. 148–158.

12. Pomorova O. V., Hovorushchenko T. O. Suchasni problemy otsiniuvannia yakosti prohramnoho zabezpechennia. Radioelektronni i kompiuterni systemy. 2013. № 5. С. 319–327.

13. Pomorova O. V., Hovorushchenko T. O. Intelligent Assessment and Prediction of Software Characteristics at the Design Stage. American Journal of Software Engineering and Applications (AJSEA). 2013. Vol. 2(2). P. 25–31.

14. Zalewski J., Kornecki A. J., Pfister H. L. Numerical Assessment of Software Development Tools in RealTime Safety Critical Systems Using Bayesian Belief Networks. — URL: http://www.proceedings2006.imcsit.org/pliks/194.pdf

15. Інструменти з відкритим вихідним кодом для аналізу коду. — URL: https://hackyourmom.com/servisy/instrumenty-z-vidkrytym-vyhidnym-kodom-dlya-analizu-kodu/

16. Arnon Axelrod. Complete Guide to Test Automation: Techniques, Practices, and Patterns for Building and Maintaining Effective Software Projects. Print2print, 2018. 588 p.

17. Automation Testing Tutorial: What is Automated Testing. — URL: https://www.guru99.com/automation-testing.html

18. Software testing principle. — URL: https://qalight.ua/baza-znaniy/printsipi-testuvannya

19. Automation Testing Tutorial: What is Automated Testing. — URL: https://www.guru99.com/automation-testing.html

20. Волочій Б. Ю. Технологія моделювання алгоритмів поведінки інформаційних систем. Львів : Вид-во НУ «Львівська політехніка», 2022. 220 с.

21. Клієнт-серверна архітектура. Вікіпедія. — URL: https://uk.wikipedia.org/wiki/Клієнт-серверна_архітектура

22. Огляд протоколу HTTP. MDN Web Docs. — URL: https://developer.mozilla.org/ru/docs/Web/HTTP/Overview

23. Тестування програмного забезпечення. Вікіпедія. — URL: https://uk.wikipedia.org/wiki/Тестування_програмного_забезпечення

24. Badiru A. B., Rusnock C. F., Valencia V. V. Project Management for Research: A Guide for Graduate Students. CRC Press, 2019.

25. Axelrod A. Complete Guide to Test Automation: Techniques, Practices, and Patterns for Building and Maintaining Effective Software Projects. Apress, 2018. — URL: http://tisten.ir/wp-content/uploads/2019/01/Complete-Guide-to-TestAutomation.pdf

26. Driver requirements Maven. Selenium Documentation. — URL: https://www.selenium.dev/documentation/webdriver/driver_requirements/

27. Getting Started with Selenium for Automated Website Testing. Sauce Labs Wiki. — URL: https://wiki.saucelabs.com/display/DOCS/Getting+Started+with+Selenium

28. Installing Selenium libraries. Selenium Documentation. — URL: https://www.selenium.dev/documentation/selenium_installation/

29. IntelliJ IDEA. JetBrains. — URL: https://www.jetbrains.com/idea/

30. Oracle Java. Oracle. — URL: https://www.oracle.com/java/

31. Selenium Webdriver. — URL: https://drive.google.com/file/d/0B7TBmsv_w76nQ1FfQVdyWWRXYk0/view

32. Selenium with Java. BrowserStack Guide. — URL: https://www.browserstack.com/guide/selenium-with-java

33. Strategies for Testing Web Applications від Client Side. Florida Institute of Technology. — URL: https://cs.fit.edu/media/TechnicalReports/cs-2003-11.pdf

34. Top 10 Automation Testing Tools in 2021. Net Solutions. — URL: https://www.netsolutions.com/insights/top-10-automation-testing-tools/

35. Корнутяк П. М. Засоби оцінювання якості програмного забезпечення. Економіка і бізнес: нові реалії, технологічні тренди, цифрові інновації : зб. мат. Міжн. наук.-практ. конф. (Львів-Ірпінь, 15 травня 2025 р.). Львів : СПДФО Омельченко В. Г., 2025. С. 169–175. ISBN 978-617-8683-03-0.

ДОДАТКИ

Додаток А

Лінійна програма

```
 #include <conio.h>
#include <stdio.h>
#include <iostream.h>
#include <fstream.h>
typedef struct _STUDENT
{
	char fio[20];
	char group[10];
	unsigned int grant;
}STUDENT, *PSTUDENT;
char path[] = "students.dat";
STUDENT st;
void main()
{
	char choice;
	fstream file;
	unsigned int num;
	do
	{
		clrscr();
		cout<<"1-add new record\n";
		cout<<"2-search record\n";
			switch(choice = getch()) {
				case '1':
				{
					file.open(path,ios::app);
					if(!file)
```

Продовження додатка А

```
                {
                    cout<<"File could not be opened!!!\n";
                }
                else
                {
                    cout<<"fio=";cin>>st.fio;
                    cout<<"group=";cin>>st.group;
                    cout<<"grant=";cin>>st.grant;

        file.write((char*)&st,sizeof(STUDENT));
                    file.close();
                }
                getch();
                break;
            }
            case '2':
            {
                file.open(path,ios::in);
                if(!file)
                {
                    cout<<"File could not be opened!!!\n";
                }
                else
                    {
                        cout<<"Enter number of
record:";cin>>num;
        file.seekp(num*sizeof(STUDENT));
```

Продовження додатка А

```
                    file.read((char*)&st,sizeof(STUDENT));
                        file.close();
                                cout<<"fio="<<st.fio<<endl;
                                 cout<<"group="<<st.group<<endl;

        cout<<"grant="<<st.grant<<endl;
                                    }
                        getch();
                        break;
                    }
                }
            }
            while(choice != 27);
}
```

Додаток Б

Програма із вказівниками

```
#include <conio.h>
#include <stdio.h>
#include <iostream.h>
#include <fstream.h>
typedef struct _STUDENT
{
	char fio[20];
	char group[10];
	unsigned int grant;
}STUDENT, *PSTUDENT;
char path[] = "students.dat";
PSTUDENT st;
void main()
{
	char choice;
	unsigned int num;
	fstream *file;
	st = new STUDENT;
	file = new fstream();
	do
	{
		clrscr();
		cout<<"1-add new record\n";
		cout<<"2-search record\n";
		switch(choice = getch())
		{
			case '1':
```

Продовження додатку Б

```
{
        file→open(path,ios::app);
        if(!file)
        {
        cout<<"File could not be opened!!!\n";
        }
        else
        {
                cout<<"fio=";cin>>st->fio;
                cout<<"group=";cin>>st→group;
                cout<<"grant=";cin>>st->grant;
                file->write((char*)st,sizeof(STUDENT));
                file->close();
        }
        getch();
        break;
}

case '2':
{
        file→open(path,ios::in);
        if(!file)
        {
                cout<<"File could not be opened!!!\n";
        }
        else
        {
                cout<<"Enter number of record:";cin>>num;
```

Продовження додатку Б

```
                    file->seekp(num*sizeof(STUDENT));
                    file->read((char*)st,sizeof(STUDENT));
                    file->close();
                    cout<<"fio="<<st->fio<<endl;
                    cout<<"group="<<st->group<<endl;
                    cout<<"grant="<<st->grant<<endl;
                }
                getch();
                break;
            }
        }
    }
    while(choice != 27);
    delete file; delete st;
}
```

Додаток В

Програма з модулями

```
#pragma argsused
#include "type.h"
#include <iostream.h>
#include "Unit1.h"
 #include "Unit2.h"
#include "Unit3.h"
#include "Unit4.h"
char path[] = "students.dat";
STUDENT st;
void main()
{
        char choice;
        unsigned int num;
        do
        {
                clrscr();
                switch(choice=menu())
                {
                        case '1':
                        {
                                cout<<"fio=";cin>>st.fio;
                                cout<<"group=";cin>>st.group;
                                cout<<"grant=";cin>>st.grant;
                                add(path, st); break;
                        }
                        case '2':
                        {
```

```
                        cout<<"Enter number of record:";cin>>num;
                        st = search(path,num);
                        print(st);
                        break;
                    }
                }
            }
            while(choice != 27);
}
 //---------------------------------------------------------------------------
#include "Unit1.h"
//---------------------------------------------------------------------------
char menu()
{
        cout<<"1-add new record\n";
        cout<<"2-search record\n";
        return getch();
}
#pragma package(smart_init)
 //---------------------------------------------------------------------------
#include "Unit2.h"
//---------------------------------------------------------------------------
void add(char *path, STUDENT st)
{
        fstream file;
        file.open(path,ios::app);
        if(!file)
        {
```

```
            cout<<"File could not be opened!!!\n";
        }
        else
        {
            file.write((char*)&st,sizeof(STUDENT));
            file.close();
        }
}
//---------------------------------------------------------------------------
#include "Unit3.h"
//---------------------------------------------------------------------------
STUDENT search(char *path, int num)
{
        fstream file;
        STUDENT st;
        file.open(path,ios::in);
        if(!file)
        {
            cout<<"File could not be opened!!!\n";
        }
        else
{
            file.seekp(num*sizeof(STUDENT));
            file.read((char*)&st,sizeof(STUDENT));
            file.close();
}
return st;
}
```

Продовження додатку В

```
//---------------------------------------------------------------------------
#include "Unit4.h"
//---------------------------------------------------------------------------
void print(STUDENT st)
{
cout<<"fio="<<st.fio<<endl;
cout<<"group="<<st.group<<endl;
cout<<"grant="<<st.grant<<endl;
getch();
}
```

Додаток Г

Метрика Холстеда для лінійної програми

Словник операторів (оператори, кількість)		Словник операндів (операнди, кількість)	
#include	4	fio	3
<	4	group	3
>	4	grant	3
Typedef	1	“”	12
Char	6	‘’	2
Unsigned	2	Choice	3
Int	2	file	3
struct_STUDENT	1	num	3
STUDENT	5	st	9
*PSTUDENT	1		
;	37		
[	3		
]	3		
Void	1		
Fstream	1		
Do	1		
While	1		
Switch	1		
Case	2		
Clrscr	1		
Cout	11		
Cin	4		
<<	17		
>>	4		
\n	4		

Продовження додатку В

=	3		
!	3		
()	13		
:	2		
If	2		
Else	2		
Fstream.open	2		
Fstream.seekp	1		
Fstream.read	1		
Fstream.close	2		
Fstream.write	1		
Getch	3		
Break	2		
*	4		
&	2		
Main	1		
Sizeof	3		
.	13		

Додаток Д

Метрика Холстеда для програми з вказівниками

Словник операторів (оператори, кількість)		Словник операндів (операнди, кількість)	
#include	4	Fio	3
<	4	Group	3
>	4	Grant	3
Typedef	1	“”	12
Char	6	‘’	2
Unsigned	2	Choice	3
Int	2	File	3
struct_STUDENT	1	Num	3
STUDENT	5	St	9
*PSTUDENT	2		
;	41		
[	3		
]	3		
Void	1		
Fstream	1		
Do	1		
While	1		
Switch	1		
Case	2		
Clrscr	1		
Cout	11		
Cin	4		
<<	17		
>>	4		
\n	4		

Продовження додатку Д

=	3		
!	3		
()	13		
:	2		
If	2		
Else	2		
Fstream.open	2		
Fstream.seekp	1		
Fstream.read	1		
Fstream.close	2		
Fstream.write	1		
Getch	3		
Break	2		
*	5		
&	2		
Main	1		
Sizeof	3		
.	10		
->	8		
New	2		
Delete	2		

Додаток Е

Метрика Холстеда для програми з модулями

Словник операторів (оператори, кількість)		Словник операндів (операнди, кількість)	
#pragma	2	argsused	1
#include	10	type.h	1
Char	7	<iostream.h>	1
STUDENT	8	Unit1.h	2
Void	3	Unit2.h	2
Unsigned	1	Unit3.h	2
Int	2	Unit4.h	2
Do	1	Path	7
While	1	St	16
()	8	Main	1
Switch	1	Choice	3
Case	2	Num	5
‘	4	fio	4
“	42	group	4
:	7	grant	4
=	10	path	7
!=	1	st	16
Cout	11	add	3
Cin	4	print	2
<<	17	getch	2
>>	4	menu	2
Endl	3	package	1
\n	4	Smart_init	1
;	39		

Продовження додатку Е

Break	2		
*	5		
&	2		
,	8		

Додаток Ж

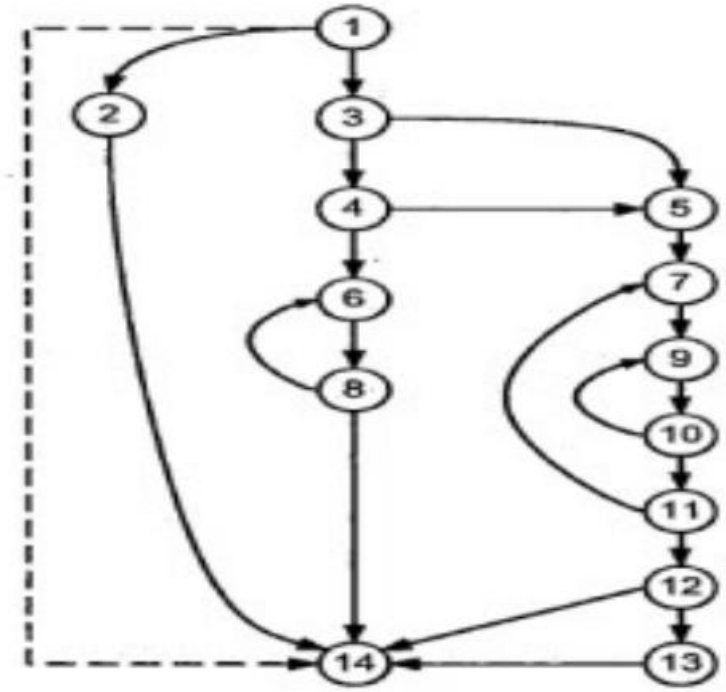


Рис. Ж.2.1. Орієнтований граф

Додаток З

Програмний код знаходження максимального значення середнього арифметичного усіх елементів тих рядків вказаної матриці, які впорядковані за зростанням або спаданням

```
#include <iostream>
#include <conio.h>
using namespace std;
// Перевіряє, чи впорядкований рядок row
bool isSorted(long n, long *row);
// Повертає максимальний елемент рядка
long GetMax(long n, long *row);
int main() {
   long i, j, m, n, val, maxv;
   long **arr;  // Нехай це буде масив із довгими цілими числами
   bool bSorted = false;  // Прапорець, що вказує на те, чи є хоча б один впорядкований рядок у масиві
   do {
      // Введення кількості рядків та стовпців матриці
      cout << "Rows in matrix: ";
      cin >> m;
      cout << "Cols in matrix: ";
      cin >> n;
      arr = new long *[m];  // Виділення пам'яті під вказівники на рядки матриці
      for (i = 0; i < m; i++) {
         arr[i] = new long[n];  // Виділення пам'яті під елементи рядка
         for (j = 0; j < n; j++) {
cout << "arr[" << i + 1 << "][" << j + 1 << "] = ";
            cin >> arr[i][j];  // Введення елементів масиву
```

Продовження додатку З

```
            }
        }
// Тепер шукаємо впорядкований рядок, якщо такі існують.
// Вирішив заощадити на коді та одночасно знаходжу
// перший максимальний елемент серед впорядкованих рядків.
// Коли рядок isSorted вже буде містити maxv як
// перший максимальний елемент впорядкованих рядків,
// подальший пошук рядків веду з i-го рядка - це зручно.
while (0 < (i--)) {
    if ((bSorted = isSorted(n, arr[i]))) {
        maxv = GetMax(n, arr[i]);
        break;
    }
}

if (!bSorted) {
    // Якщо впорядкованих рядків у масиві немає, bSorted = false,
    // виводимо відповідне повідомлення
    cout << "Input array does not contain sorted rows!\r\n";
} else {
// Завершуємо перегляд масиву
while (0 < (i--)) {
   if (isSorted(n, arr[i])) {  // Працюємо тільки з впорядкованими рядками
        if (maxv < (val = GetMax(n, arr[i])))
            maxv = val;
    }
```

Продовження додатку З

```
}
cout << "Max value in sorted rows: " << maxv << "\r\n";
}
cout << "Press Y for new input\r\n";
}  // Якщо натиснули Y, можемо повторити алгоритм для нової матриці
while (toupper(getch()) == 'Y');
return 0;
}
bool isSorted(long n, long *row) {
   bool bRet = true;  // Полагаємо на початку, що рядок впорядкований
   long i;
   // Тепер порівнюємо елементи рядка row
   // Перевіряємо впорядкованість за зростанням
   for (i = 0; i < n - 1 && bRet; i++) {
      if (row[i] > row[i + 1])
         bRet = false;
   }
   // Якщо рядок не впорядкований за зростанням,
   // перевіряємо, чи він впорядкований за спаданням

  if (!bRet) {
      bRet = true;
      for (i = n - 1; 0 <= i && bRet; i--) {
         if (row[i] < row[n - i - 1])
            bRet = false;
      }
   }
```

Продовження додатку З

```
    return bRet;
}
long GetMax(long n, long *row) {
    long ret = row[0];
    for (long i = 1; i < n; i++) {
        if (ret < row[i])
            ret = row[i];
    }
    return ret;
}
```

Додаток И

Тест на введення правильних даних

```
#include <gtest/gtest.h>
#include <gmock/gmock.h>
#include <iostream>
#include <sstream>
// Додаємо метод, який приймає введення від користувача та повертає його як рядок
std::string GetUserInput(const std::vector<std::vector<long>>& matrix) {
    std::ostringstream input;
    input << matrix.size() << "\n" << matrix[0].size() << "\n";
    for (const auto& row : matrix) {
        for (long element : row) {
            input << element << "\n";
        }
    }
    return input.str();
}
// Тест для введення правильних даних
TEST(MainTest, CorrectInput) {
    // Підготовка даних для тесту
    std::vector<std::vector<long>> matrix = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };
    std::string userInput = GetUserInput(matrix);
```

Продовження додатку И

```
 // Перенаправлення введення користувача
    std::istringstream inputStream(userInput);
    std::streambuf* originalCin = std::cin.rdbuf();
    std::cin.rdbuf(inputStream.rdbuf());
    // Виклик головної функції
    testing::internal::CaptureStdout();
    main();
    std::string output = testing::internal::GetCapturedStdout();
    // Відновлення введення користувача
    std::cin.rdbuf(originalCin);
    // Перевірка результату виведення
    ASSERT_THAT(output, testing::HasSubstr("Max value in sorted rows: 9"));
}
int main(int argc, char **argv) {
    testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

Додаток К

Сертифікат учасника міжнародної конференції



МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
МІНІСТЕРСТВО ФІНАНСІВ УКРАЇНИ

Сертифікат

Павло КОРНУТЯК

взяв(ла) участь у роботі

І Міжнародної науково-практичної конференції здобувачів вищої освіти та молодих вчених
«ЕКОНОМІКА І БІЗНЕС: НОВІ РЕАЛІЇ, ТЕХНОЛОГІЧНІ ТРЕНДИ, ЦИФРОВІ ІННОВАЦІЇ»
ECONOMY AND BUSINESS: NEW REALITIES, TECHNOLOGICAL TRENDS, DIGITAL INNOVATIONS

Андрій ГУКАЛЮК
Перший проректор

15 травня 2025 р.

Олександра СМІРНОВА
Проректор з наукової роботи