

МІНІСТЕРСТВО ФІНАНСІВ УКРАЇНИ
ДЕРЖАВНИЙ ПОДАТКОВИЙ УНІВЕРСИТЕТ

Факультет фінансів та цифрових технологій
Кафедра комп'ютерних та інформаційних технологій і систем

КВАЛІФІКАЦІЙНА РОБОТА

на тему:

ЗАСОБИ ОЦІНЮВАННЯ ЯКОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Здобувача вищої освіти другого (магістерського) рівня,
II курсу, групи КМІ-22-1,
спеціальності 122 «Комп'ютерні науки»,
освітня програма «Інформаційні управляючі системи і технології в економіці»


(підпис)

Федорука Романа Івановича

Науковий керівник


(підпис)

Погоршловська І. Д., к.е.н., доцент

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

Керівник освітньо-професійної
програми


(підпис)

Горбовий А. Ю., д.т.н., професор

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

Допущено до захисту за рішенням кафедри комп'ютерних та інформаційних технологій і систем, протокол від «28 » грудня 2023 № 8

Завідувач кафедри


(підпис)

Філоненко М. М., к.ф.-м.н., доцент

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

Ірпінь – 2024

Реєстрація № 12

29.01.2024

(дата)



(підпис відповідальної особи)

Результати перевірки

до захисту

(до захисту, на доопрацювання)

29.01.2024

(дата)



(підпис)

Результат захисту

4 (добре)

(оцінка)

12.02.2024

(дата)



(підпис)

Голова комісії:



(підпис)

В.М. Франчук

Члени комісії:



(підпис)

А.Ю. Горбовий



(підпис)

О.В. Редич



(підпис)

В.О. Ніжегородцев

БЛАНК ОЦІНЮВАННЯ ВИПУСКНОЇ РОБОТИ

Об'єкти контролю	Діапазон оцінки в балах	Фактична кількість балів	Примітки	Дата, ПІБ: керівника, голови комісії, членів комісії, секретаря, (підписи)
ОЦІНЮВАННЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ КОМІСІЄЮ ПО ЗАХИСТУ				
1.Зміст випускної роботи (якість теоретичного матеріалу, його зв'язок з практичним, вміння представляти матеріал в логічній послідовності, наявність висновків та пропозицій)	від 0 до 30	23		Франчук В.М. <i>В.М.</i> Горбовий А.Ю. <i>А.Ю.</i> Редич О.В. <i>О.В.</i> Ніжегородцев В.О. <i>В.О.</i> Суховерхівка Л.П.
2.Дотримання вимог щодо оформлення	від 0 до 10	5		
3.Вчасне виконання роботи на всіх етапах у встановлені кафедрою терміни	від 0 до 5	2		
4. Рецензент	від 0 до 5	4		
РАЗОМ БАЛІВ*	від 0 до 50	34		

Оцінювання кваліфікаційної роботи науковим керівником - 41 балів.

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

Оцінка за 100-бальною шкалою	Оцінка за шкалою ЄКТС	Оцінка за національною шкалою	Дата	ПІБ: голови, членів комісії, секретаря, (підпис)
75	C	4 (добре)	12.02.2024	Франчук В.М. <i>В.М.</i> Горбовий А.Ю. <i>А.Ю.</i> Редич О.В. <i>О.В.</i> Ніжегородцев В.О. <i>В.О.</i> Суховерхівка 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

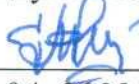
Галузь знань: 12 «Інформаційні технології»

Спеціальність 122 «Комп'ютерні науки»

Освітньо-професійна програма «Інформаційні управляючі системи та технології в економіці»

Затверджую

Завідувач кафедри

 Філоненко М.М.
04. 04. 2023

**Завдання на кваліфікаційну роботу
здобувача вищої освіти**

Федорука Романа Івановича

1.Тема кваліфікаційної роботи: Засоби оцінювання якості програмного забезпечення

затверджена наказом ректора від 04.04.2023 №362

2.Строк здачі закінченої роботи: 20.12.2023

3.Цільова установка: застосування набутих знань, умінь при оцінюванні якості програмного забезпечення.

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

Об'єкт дослідження: якість програмного коду

Предмет дослідження: засоби оцінювання якості програмного забезпечення

4.Вихідні дані до роботи: програмне забезпечення та методики, технології, інструменти оцінювання його якості

5.Перелік графічного матеріалу: схема моделі якості ПЗ, копії екранів, лістинг з програмним кодом, презентація роботи(.ppt).

6. Зміст випускної кваліфікаційної (дипломної) роботи: Розділ 1. Теоретичні аспекти оцінювання якості ПЗ. 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 випускної кваліфікаційної (дипломної) роботи	Строк виконання етапів роботи	
		За планом	Фактично
1	2	3	4
1	Вибір теми, підбір літературних джерел, формування структури ДР	До 27.12.2022	27.12.2022
2	1. Вступ та I розділ (розкрити сутність і стан дослідження наукової проблематики роботи, після чого навести її загальну характеристику; огляд літератури за темою і вибір конкретних напрямів досліджень)	26.06.2023	26.06.2023
3	II розділ (аналіз експериментальної частини і методики досліджень)	23.10.2023	23.10.2023
4	III розділ (проведені теоретичні і (або) експериментальні дослідження та проектні рішення)	30.11.2023	30.11.2023
5	Висновки	10.12.2023	10.12.2023
6	Доопрацювання роботи з урахуванням зауважень наукового керівника	20.12.2023	20.12.2023
7	Отримання відгуку керівника	29.01.2024	29.01.2024
8	Отримання зовнішньої рецензії	01.02.2024	01.02.2024

8. Дата видачі завдання 04.04.2023 р.

9. Керівник кваліфікаційної роботи  Погореловська І.Д.

10. Гарант
освітньо-професійної програми  Горбовий А.Ю.

11. Завдання до виконання отримав
здобувач вищої освіти  Федорук Р.І.

АНОТАЦІЯ

Засоби оцінювання якості програмного забезпечення

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 стандартом 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 стандарти, вимоги яких спрямовані на забезпечення якості програмного забезпечення, метрики оцінювання складності ПЗ, методи та інструменти тестування ПЗ.

Структура дипломної роботи містить вступ, основну частину, яка складається з трьох розділів, до якої входять 3 рисунки, 6 таблиць, висновки, перелік використаної літератури, що містить 34 джерела.

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 визначенні якості програмного коду, а також в освітньому процесі при викладанні навчальних дисциплін “Моделі та методи забезпечення якості ІУС”, “Інноваційне управління ІТ-проєктами”.

Дипломна робота містить 55 сторінок, 3 рисунки, 6 таблиць, список використаних джерел з 34 найменувань, 9 додатків.

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

Рік виконання дипломної роботи: 2023-2024

Рік захисту роботи: 2024.

ABSTRACT

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.

The structure of the thesis contains an introduction, the main part, which consists of three sections, which includes 3 figures, 6 tables, conclusions, a list of used literature, which contains 34 sources.

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 thesis contains 50 pages, 3 figures, 6 tables and a list of used sources from 34 items, 9 appendices.

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

The year of completion of the thesis: 2023-2024

Year of job protection: 2024.

ЗМІСТ	
ВСТУП.....	3
РОЗДІЛ 1. ТЕОРЕТИЧНІ АСПЕКТИ ОЦІНЮВАННЯ ЯКОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ.....	6
1.1. Якість ПЗ як предмет стандартизації.	6
1.2. Інструменти оцінювання якості програмного забезпечення.....	11
РОЗДІЛ 2. ЗАСТОСУВАННЯ МЕТРИЧНОГО АНАЛІЗУ У ВИЗНАЧЕННІ ЯКОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ...	19
2.1. Застосування метрики Холстеда.....	19
2.2. Застосування метрики Маккейба	22
2.3. Застосування метрик визначення складності програмного продукту (метрика Чепіна, Джілбі Т.).....	25
РОЗДІЛ 3. ЗАСОБИ АВТОМАТИЗОВАНОГО ОЦІНЮВАННЯ ЯКОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ.....	28
3.1. Передумови автоматизованого тестування.....	28
3.2. Види автоматизованих тестів.....	38
3.3. Основні інструменти для автоматизованого тестування.....	42
ВИСНОВКИ.....	53
СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ.....	55
ДОДАТКИ	58

ВСТУП

Актуальність теми. Сучасний розвиток інформаційних технологій вимагає постійного вдосконалення якості програмного забезпечення, що використовується в різних сферах економіки. Оцінка якості програмного забезпечення стає ключовою проблемою, оскільки ефективність та надійність використання програмного забезпечення залежать від його якісних характеристик, що потребує пошуку нових та вдосконалення існуючих методів та моделей метричного аналізу та тестування.

Стан і ступінь дослідження теми в літературі. Проблемі якості ПЗ присвячено ряд робіт українських та іноземних учених: О. Димо, Л. Омельчука, В. Ліпаєва, К. Вігерса, Н. Фентона, А. Чена, А. Фатванто, Т. Рехмана. Питання оцінювання та забезпечення якості ПЗ одержали свій розвиток в роботах українських та іноземних учених: Ю. Грицюк, О. Андрущакевич, В. Сеньківського, В. Харченка, Б. Конорєва, Д. Маєвського, В. Яковини, В. Міщенко, О. Поморової, К. Лавріщевої, О. Харченка, Г. Майєрса, С. Макконнелла, Р. Фатрелла, І. Соммервилла, О. Медче, С. Джонса.

Мета дослідження: визначення методів та програмних засобів аналізу якості програмного забезпечення

Для досягнення визначеної мети були виконанні наступні завдання:

- визначено сутність поняття “якість програмного забезпечення” як предмету стандартизації;
- проаналізована модель якості ПЗ, яка визначена стандартом 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, Маккейба, Чепіна, Джілбі Т.;
- 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.

Методи дослідження: при написанні роботи були використані наступні методи: системний аналіз, який застосовувався при роботі з інформаційними ресурсами за темою дослідження; методи порівняння та аналогій були застосовані при дослідженні метрик оцінювання якості програмного коду та інструментів автоматизованого тестування; методи оцінювання застосовувалися при дослідженні методів статичного та динамічного тестування ПЗ; методи групування – при визначені якості лінійної програми, програма із вказівниками та програми з модулями; за допомогою методу узагальнення були сформувані висновки.

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

Структура дипломної роботи містить вступ, основну частину, яка складається з трьох розділів, до якої входять 3 рисунки, 6 таблиць, висновки, перелік використаної літератури, що містить 34 джерела, 9 додатків.

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.

Науково-практичне значення одержаних результатів: 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”.

РОЗДІЛ 1. ТЕОРЕТИЧНІ АСПЕКТИ ОЦІНЮВАННЯ ЯКОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1. Якість програмного забезпечення як предмет стандартизації

Високоякісне програмне забезпечення є однією з важливих складових розвитку всіх галузей економіки. Від того, наскільки вдало розроблено програмний продукт, залежить в кінцевому результаті ефективність прийняття управлінських рішень. Якість програмного забезпечення (ПЗ) є основною його характеристикою в різних сферах використання інформаційних технологій, яка вказує на ступінь його відповідності встановленим вимогам. Поняття якості має різні інтерпретації в залежності від вимог до конкретної прикладної системи чи програмного продукту. Здебільшого, під якістю програмного забезпечення розуміють набір властивостей програмного продукту, що характеризують його здатність задовольнити встановлені або передбачувані *потреби* замовника, які він висловив у вигляді користувацьких *вимог* на початкових етапах його розроблення [8, с.159].

Водночас, існує набір стандартів, який регулює принципи забезпечення якості програмного забезпечення. Зокрема, сучасний підхід до визначення якості ПЗ відображений в серії стандартів ISO/IEC 25000 (Вимоги до якості систем програмних засобів та їхнього оцінювання) або SQuaRE (Software Quality Requirements and Evaluation). Номенклатура, атрибути і метрики вимог якості програмного забезпечення визначені стандартом ДСТУ ISO/IEC 25023:2019. Згідно цього стандарту, «якість програмного забезпечення (software quality) – ступінь задоволення програмним продуктом заявленим та передбачуваним потребам при використанні в зазначених умовах» [1].

У стандарті ДСТУ ISO/IEC 25022:2019 визначено поняття «якість при використанні», що забезпечує можливість оцінювання якості з точки зору користувачів: «якість при використанні (quality in use) – ступінь, з якою продукція або система можуть бути застосовані певними

користувачами для задоволення їх вимог в досягненні цілей ефективності (у т. ч., і економічної), уникаючи ризику, задоволеності і охоплення контексту в заданих умовах використання» [2].

Основні процеси етапів розробки ПЗ, організаційні та додаткові процеси, які регламентують інженерію, планування й управління якістю програмного забезпечення визначає стандарт ДСТУ ISO/IEC/IEEE 12207:2018 [4]. Згідно цього стандарту, на всіх етапах розробки ПЗ повинен здійснюватися контроль його якості, що включає: «перевірку відповідності користувацьких вимог до ПЗ з критеріями і показниками його якості; перевірку (верифікація) та атестацію (валідація) проміжних результатів розроблення ПЗ на кожному з етапів реалізації програмного проєкту і визначення ступеня задоволення досягнутих критеріїв і показників його якості; тестування готового ПЗ, збирання даних про відмови, дефекти та інші помилки, які було виявлено в ПЗ під час його безпосередньої експлуатації; підбір моделей надійності ПЗ за отриманими результатами його тестування (дефекти, відмови та ін.); перевірка досяжності критеріїв і показників якості ПЗ, заданих у вимогах до нього» [4].

Отже, якість ПЗ є відносним поняттям, і вимоги, що висувуються до якості ПЗ відповідно стандартів, враховують також умови його функціонування у конкретній сфері господарювання.

Якість ПЗ, як зазначено у [8], характеризується: якістю процесів розробки ПЗ, якістю продуктів програмного проєкту і якістю супроводу або впровадження ПЗ. «Компонента, що пов'язана з процесами розробки ПЗ, визначає ступінь формалізації, достовірності самих процесів на кожному етапі його розробки, а також тісно пов'язана з верифікацією (перевіркою) та валідацією (затвердженням) проміжних результатів цих процесів. Пошук і усунення помилок у готовому ПЗ проводять методами тестування, які знижують їх кількість і підвищують якість майбутнього програмного продукту. Якість продуктів програмного проєкту

досягається за рахунок використання процедур контролю проміжних продуктів проекту на всіх етапах розробки ПЗ, перевіркою їх на досягнення необхідної якості, а також завдяки використанню сучасних методів і засобів супроводу програмного продукту. Ефект від впровадження ПЗ значною мірою залежить від знань обслуговуючого персоналу, функціональних можливостей програмного продукту і чітких правил їх виконання» [8, с.162].

Для того, щоб переконатися у тому, що розроблене ПЗ відповідає тим вимогам, які було узгоджено з замовником, з урахуванням вимог стандартів, необхідно визначитися з методами та моделями, застосування яких доведе правильність роботи ПЗ. Таке доведення має бути формальним (строгим). Для визначення комплексу критеріїв, що належним чином відображають властивості та характеристики програмного забезпечення з різних сторін, формується модель якості.

Стандартом ДСТУ ISO/IEC 25010:2016 [3] визначено модель якості програмного забезпечення, яка має чотири рівня представлення.

Перший рівень відповідає визначенню характеристик якості програмного забезпечення, кожна з яких відображає окрему точку зору користувача на якість. Згідно стандарту в модель якості входить шість характеристик або показників якості (рис.1.1):

функціональність (functionality) – здатність ПЗ виконувати певний набір функцій, які задовольняють потреби користувачів (функціональна повнота, правильність (точність), інтероперабельність, захищеність);

надійність (reliability) – сукупність атрибутів, що визначають здатність ПЗ підтримувати визначену працездатність у заданих умовах;

зручність використання або практичність (usability) – характеристика ПЗ, яка дозволяє мінімізувати зусилля користувача по підготовці вхідних даних, і оцінці отриманих результатів;

продуктивність або ефективність (efficiency) – відношення рівня

послуг, які надає ПЗ користувачу при заданих умовах, до обсягу використаних усіх типів ресурсів;

зручність супроводу (*maintainability*) – зручність проведення всіх видів діяльності, пов'язаних із супроводом програм і дозволяє мінімізувати зусилля по внесенню змін для усунення в ньому помилок та його модифікації відповідно до потреби користувача;

переносність або мобільність (*portability*) – здатність ПЗ зберігати працездатність при перенесенні з одного середовища в інше, включаючи організаційні, апаратні й програмні аспекти оточення.

Ці аспекти якості взаємодіють та визначають загальний успіх програмного продукту на ринку та задоволення його користувачів.

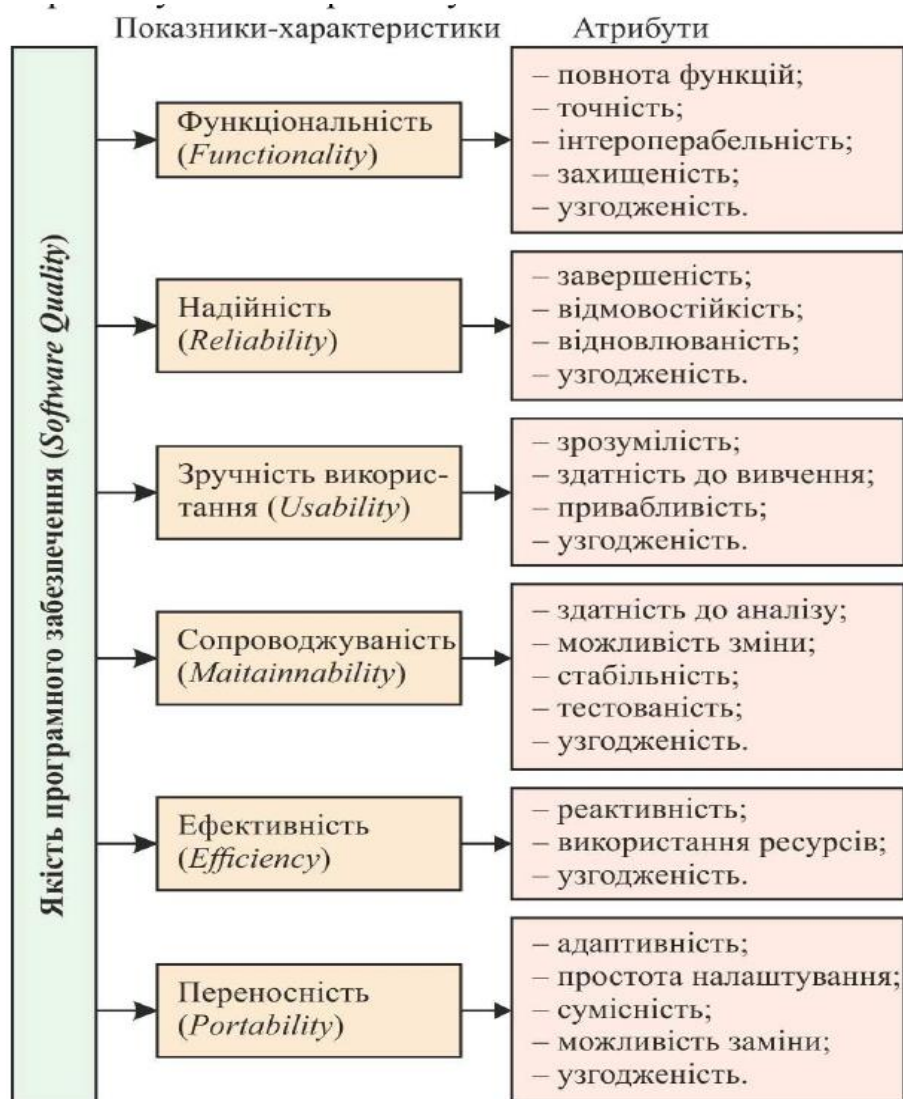


Рис. 1.1. Модель якості програмного забезпечення (ISO/IEC 25010) [8, с.

Другому рівню відповідають атрибути для кожної характеристики якості, які деталізують різні аспекти конкретної характеристики (рис.1.1). Набір атрибутів характеристик якості використовують в метричному аналізі якості ПЗ.

Третій рівень призначений для вимірювання якості за допомогою метрик, кожна з яких згідно стандарту [3] визначається як комбінація методу вимірювання атрибуту і шкали встановлення значень атрибутів. Атрибут якості визначається за допомогою однієї або декількох методик оцінки на етапах життєвого циклу і на завершальному етапі розробки – тестуванні ПЗ.

На *четвертому рівні* для оцінювання кількісного або якісного значення окремих атрибутів ПЗ використовують результати вимірювання якості за допомогою метрик, що отримані на третьому рівні моделі. Залежно від призначення, особливостей експлуатації та умов супроводу ПЗ вибирають найбільш важливі характеристики і атрибути його якості (див. рис. 1.1).

Описана модель якості ПЗ - це структурований підхід до визначення та вимірювання якості програмного забезпечення [6].

Отже, проаналізовано поняття якості програмного продукту як предмету стандартизації, з'ясовано особливості процесу оцінювання якості ПЗ та рівні представлення моделі якості ПЗ.

1.2. Інструменти оцінювання якості програмного забезпечення

Сучасна ІТ-індустрія нагромадила достатньо велику кількість інструментів, які дають змогу оцінити окремі виробничі та експлуатаційні властивості ПЗ [12]. Відповідно до стандарту ISO 25010 оцінювання якості ПЗ відбувається на основі атрибутів якості, зазначених у ISO 25023:2019 [1], при цьому оцінюють підхарактеристики та характеристики якості ПЗ, які, водночас, надають

комплексну його оцінку. Стандартом ISO 25010 поняття «характеристика» визначено як «набір властивостей ПЗ, за допомогою яких можна описати та оцінити його якість» [3]. Підхарактеристика якості ПЗ виражається середньозваженим арифметичним показником з урахуванням значень атрибутів, що оцінюють цю підхарактеристику, та коефіцієнтів їхньої вагомості [12].

Якість ПЗ вимірюється за допомогою метрик якості. За визначенням стандарту ISO/IEC 25022:2016 метрика якості ПЗ є "... моделлю вимірювання атрибута, пов'язаного з характеристикою якості ПЗ. Це комбінація конкретного методу вимірювання (способу отримання значень) атрибута сутності й шкали вимірювання" [2]. Згідно з даними стандарту ISO 24765:2010, метрику можна визначити як міру ступеня володіння властивістю, яка має числове значення. Загалом, метрика ПЗ – це «міра, яка дозволяє одержати чисельне значення деякої властивості ПЗ або його специфікацій» [7]. Отримане чисельне значення дозволяє зрозуміти в якій мірі система, її компоненти або процес відповідають встановленим нормам.

Застосування кількісних метрик якості ПЗ сприяє вирішенню наступних практичних завдань: “прогнозування кількості помилок у ПЗ від початку проектування; прогнозування рівня складності ПЗ та його супроводу на підставі аналізу результатів проектування; прогнозування рівня складності процесів тестування та кількості невиявлених помилок на підставі аналізу коду програми; прогнозування остаточного розміру коду програми на підставі аналізу оцінок складності проектування архітектури ПЗ; визначення впливу окремих характеристик програмного коду на якість готового ПЗ; контроль етапів реалізації програмного проекту; аналіз явних і прихованих дефектів у готовому ПЗ; виявлення кращих методів і технологій розроблення ПЗ на підставі їх порівняння” [9].

Виділяють також ряд метрик, які дають прогнозоване значення на етапі проектування : 1) очікувана LOC- оцінка (експертна); 2) метрика Холстеда – обчислюється на підставі аналізу кількості рядків і синтаксичних елементів вихідного коду програми; 3) метрика Маккейба – цикломатична складність; 4) метрика Джилба – відносна логічна складність програми; 5) прогнозована кількість операторів програми; 6) прогнозована оцінка складності інтерфейсів компонент ПЗ; 7) прогнозована загальна тривалість процесу розробки ПЗ – метрика процесу розробки ПЗ; 8) тривалість виконання робіт етапу проектування ПЗ – метрика процесу розроблення ПЗ; 9) очікувана вартість процесу розробки ПЗ; 10) прогнозована вартість перевірки якості – метрика процесу розробки ПЗ; 11) прогнозована продуктивність процесу розробки ПЗ; 12) прогнозовані витрати на реалізацію коду – метрика процесу розробки ПЗ; 13) прогнозований функціональний розмір FP – вимірює суть можливостей майбутньої програми; 14) прогнозована оцінка трудовитрат і тривалості реалізації програмного проекту – за моделлю Боема [12]. Більш детально метрики будуть розглянуті у Розділі 2.

За видом (станом) об'єкта вимірювання (робоча версія програми) метрики умовно поділяються на зовнішні, внутрішні й експлуатаційні [13]. Внутрішні метрики призначені для оцінювання внутрішньої якості ПЗ, тобто множини властивостей, які визначають його здатність задовольняти встановленим або реальним потребам замовників при його використанні в певних умовах безпосередніми користувачами. Зовнішні метрики призначені для оцінювання зовнішньої якості ПЗ, тобто відповідності встановленими (заявленими) і передбачуваними вимогами під час його використання у певних умовах. Метрики експлуатаційної якості дають змогу виміряти рівень якості ПЗ, що встановлене в замовника й використовується у певному середовищі експлуатації за безпосереднім призначенням, задовольняє потреби користувачів

стосовно ефективного, продуктивного та безпечного вирішення завдань, покладених на нього. Ці метрики допомагають оцінити не властивості самого ПЗ, а видимі результати його експлуатації, тобто експлуатаційну якість ПЗ. Найчастіше для оцінювання ПЗ використовують метрики зв'язності, зчеплення, очікуваної вартості розроблення, очікуваної загальної тривалості процесу розробки ПЗ [13].

Одним із способів перевірки якості ПЗ є тестування. “Тестування програмного забезпечення — техніка контролю якості, що перевіряє відповідність між реальною і очікуваною поведінкою програми завдяки кінцевому набору тестів, які обираються певним чином” [23].

Тестування ПЗ є засобом перевірки характеристик готової програми в статистиці (перегляди, інспекції, налагодження вихідного коду) і в динаміці (прогін на наборі тестових даних) з метою аналізу різних шляхів виконання програми і порівняння отриманих результатів із заздалегідь заданими.

Існує дві форми тестування коду – модульна й інтеграційна, які проводять з урахуванням вимог стандартів: ISO/IEC/IEEE 29119-1:2013, Part 1: Concepts and definitions; ISO/IEC/IEEE 29119-3:2013, Part 3: Test documentation; ISO/IEC/IEEE 29119-2:2013, Part 2: Test processes; ISO/IEC/IEEE 29119-4:2015, Part 4: Test techniques [5].

Спочатку використовують стандарти з перевірки і тестування модулів. Потім проводиться інтеграційне тестування модулів системи і їх інтерфейсів у динаміці виконання. Під час різних видів перевірок збираються дані про помилки, дефекти, відмови тощо і оформляється відповідна документація (таблиці типів помилок, частоти і часу виявлення відмов і ін.). Зібрані дані використовуються при оцінюванні характеристик якості готового ПЗ.

Статичний аналіз коду – це метод виявлення потенційних помилок, аномалій чи інших проблем у програмному коді без його активного виконання - без запуску програми. Використання статичного

аналізу коду не обмежується лише виявленням помилок у програмі. Деякі статичні аналізатори дають змогу перевіряти, чи відповідає вихідний код прийнятому в компанії стандарту оформлення коду [15].

Для статичного аналізу коду, згідно [18] використовують засоби:

1. Linter (лінтери): це інструменти, які перевіряють програмний код на відповідність визначеним стандартам і правилам оформлення коду. Лінтери вимагають дотримання стандартів, що сприяє єдиному стилю написання коду командою та полегшує читання коду. Приклади включають ESLint для JavaScript і Pylint для Python.

2. Статичні аналізатори коду: інструменти проводять більш глибокий аналіз програмного коду для виявлення потенційних помилок, проблем безпеки та інших аномалій. Наприклад, SonarQube, Checkmarx.

3. Аналізатори вразливостей: ці інструменти спеціалізуються на виявленні вразливостей у коді, таких як Cross-Site Scripting (XSS), SQL Injection тощо. Аналізатори вразливостей дозволяють ідентифікувати проблеми безпеки та вживати заходів для їх усунення. Наприклад, OWASP SonarQube, Veracode.

4. Аналізатори покриття коду: - визначають, яка частина коду була викликана під час тестування, що допомагає визначити, наскільки ефективно тестування покриває код. Наприклад, JaCoCo для Java.

5. Аналізатори коду з погляду безпеки: ці інструменти фокусуються на виявленні потенційних проблем безпеки у програмному коді, таких як використання небезпечних функцій чи недостатнім є контроль доступу. Наприклад, Fortify, Veracode [26].

6. Аналізатори залежностей: визначають, які бібліотеки та компоненти використовуються в проєкті, допомагаючи уникнути проблем з сумісністю та безпекою. Наприклад, Snyk, Dependency-Check.

Використання цих засобів дозволяє розробникам забезпечити високу якість коду, виявляти помилки на ранніх етапах розробки та підвищувати безпеку програмного забезпечення. Загалом, використання

засобів статичного аналізу коду є важливою складовою сучасних методів розробки програмного забезпечення, сприяючи покращенню його надійності, безпеки та ефективності.

Динамічний аналіз коду відіграє важливу роль у забезпеченні високої якості ПЗ та виявленні потенційних проблем під час його виконання. Для проведення ефективного динамічного аналізу використовують різноманітні засоби та інструменти, які дозволяють виявляти помилки, проводити профілювання виконання програми та здійснювати тестування продуктивності. Динамічний аналіз коду – це спосіб аналізу програми безпосередньо під час її виконання. Аналіз виконується за допомогою набору даних, які подаються на вхід досліджуваній програмі. Тому ефективність аналізу безпосередньо залежить від якості та кількості вхідних даних для тестування. Саме від них залежить повнота покриття коду, яка буде отримана за результатами тестування. Динамічне тестування найбільш важливе в тих галузях, де головним критерієм є надійність програми, час відгуку або споживані ресурси. У таких областях будь-яка допущена помилка може виявитися критичною [34].

Такими засобами є:

1. Дебагери: надають розробникам можливість вставляти точки зупинки в код, вивчати значення змінних та покроково виконувати програми для виявлення та усунення помилок.
2. Інструменти для профілювання: дозволяють аналізувати роботу програми в режимі реального часу. Вони надають інформацію про час виконання окремих функцій, використання ресурсів та інші аспекти продуктивності.
3. Юніт-тести та фреймворки для автоматизованого тестування: це спеціальні фреймворки для написання та виконання юніт-тестів дозволяють автоматизувати тестування окремих частин програми. Це сприяє виявленню помилок та забезпеченню коректності функціоналу.

4. Інструменти для аналізу безпеки: виявляють потенційні уразливості в програмному коді, допомагаючи підвищити рівень безпеки програмного забезпечення.

5. Інструменти для тестування навантаження: генерують імітовані навантаження та вимірюють реакцію системи на них [25].

Застосування динамічного тестування у поєднанні зі статичним аналізом дозволяє розробникам створювати надійне, ефективне та безпечне програмне забезпечення.

При проведенні тестових сценаріїв важливим є також аналіз покриття коду. Мета проведення аналізу покриття коду полягає в виявленні частин програмного коду, які не виконуються під час проведення тестових сценаріїв. Тестові сценарії, що розроблені на основі вимог, можуть не охоплювати всі частини структури коду. Тому, для покращення рівня покриття проводиться аналіз того, наскільки тести охоплюють різні аспекти коду. У разі необхідності виконуються додаткові перевірки для визначення причин недостатнього покриття та визначення заходів для його усунення [6]. Зазвичай такий аналіз виконується з дотриманням наступних умов:

- Підтвердження того, що повнота покриття тестами структури коду відповідає встановленому типу покриття і визначеному мінімально допустимому рівню покриття.
- Аналіз повноти покриття тестами структури коду може проводитися за допомогою оригінального тексту, за умови, що програмне забезпечення не відноситься до класу А. Для рівня А необхідно перевірити об'єктний код, який генерується компілятором, на предмет того, чи відтворює він вихідний текст. Якщо об'єктний код не відтворює вихідний текст, необхідно перевірити правильність генерації послідовності команд. Прикладом об'єктного коду, що не відтворюється вихідним текстом, але генерується компілятором, може бути перевірка виходу за визначені межі масиву.

- Аналіз повинен підтверджувати правильність передачі даних та управління між компонентами коду [34].

Аналіз того, наскільки повно тестові приклади охоплюють код, може виявити частини вихідного коду, які залишаються не виконаними під час проведення тестування. У таких випадках можуть бути вжиті додаткові заходи для вирішення цієї ситуації у процесі перевірки ПЗ. Невиконані ділянки коду можуть виникати з наступних причин:

- Недоліків у формуванні тестових прикладів, що базуються на вимогах: Ситуацію можна вирішити доповненням тестових прикладів або зміною тестових процедур для охоплення пропущеної частини коду. Це може також вимагати перегляду методу (методів), використовуваного для аналізу повноти тестів на основі вимог;
- Неадекватності в вимогах до програмного забезпечення: Ситуацію можна вирішити модифікацією вимог до програмного забезпечення, розробкою та виконанням додаткових тестових прикладів та тестових процедур;
- «Мертвого» коду: Такий код слід вилючити, і провести аналіз для оцінки ефекту видалення та необхідності подальшого перегляду;
- Дезактивованого коду: Для такого коду, який не передбачається до виконання в кожній конфігурації, комбінація аналізу та тестів повинна продемонструвати можливості засобів, якими ненавмисне виконання такого коду запобігається, ізолюється або усувається. Для дезактивованого коду, який виконується тільки при певних конфігураціях, слід визначити нормальну робочу конфігурацію для його виконання і розробити додаткові тестові приклади та тестові процедури для досягнення повноти покриття тестами структури коду;
- Надмірності умов: Логіку роботи таких умов слід переглянути, щоб упевнитися в її правильності. Наприклад, умова типу `if (A && B || !B)` може бути принципово недосяжною, оскільки неможливо перевірити, що частина умови `A && B` буде дорівнювати `False` в разі,

коли $A = \text{True}$ і $B = \text{False}$, через те що друга частина умови ($!B$) буде дорівнювати True і загальний результат логічного виразу буде True [30];

- Захисного коду: Цю частину коду використовують для уникнення виняткових ситуацій, які можуть виникнути в процесі роботи програми. Наприклад, це може бути гілка `default` в операторі вибору `switch`, при цьому вхідна умова оператора `switch` може приймати певні значення, які він описує, і як результат, гілка `default` ніколи не буде виконана.

У результаті дослідження було виявлено, що існують різноманітні підходи та інструменти забезпечення якості програмного забезпечення. Однією з груп оцінок якості ПЗ є метричний аналіз, який базується на зовнішніх, внутрішніх й експлуатаційних метриках. Для визначення ймовірності того, що розроблений програмний продукт буде високої якості застосовують формулу повної ймовірності Баєса. Статичний та динамічний аналіз коду, тестування продуктивності та інші види тестів допомагають виявляти та усувати різноманітні проблеми, забезпечуючи надійність та ефективність програмних продуктів. При цьому важливо враховувати повноту покриття коду тестами, виявляти та виправляти невиконані частини коду, а також вдосконалювати тестові сценарії для забезпечення високої якості програмного забезпечення.

РОЗДІЛ 2. ЗАСТОСУВАННЯ МЕТРИЧНОГО АНАЛІЗУ У ВИЗНАЧЕННІ ЯКОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1. Застосування метрики Холстеда

При застосуванні метрики Холстеда частково компенсуються недоліки, пов'язані з можливістю запису однієї і тієї ж функціональності різною кількістю рядків і операторів [18]. Використовуючи метрику Холстеда, визначають:

- 1) n_1 – число операторів даної реалізації (унікальних операторів), включаючи символи-роздільники, імена процедур і знаки операцій;
- 2) n_2 – число операндів даної реалізації (унікальних операндів);
- 3) N_1 – загальне число всіх операторів;
- 4) N_2 – загальне число всіх операндів.

На основі наведених вище характеристик знаходять:

- а) довідник (словник) програми

$$n = n_1 + n_2 \quad (2.1);$$

- б) довжину програми

$$N = N_1 + N_2 \quad (2.2);$$

- в) обсяг програми

$$V = (N_1 + N_2) * \log_2(n_1 + n_2) \quad (2.3).$$

Далі Холстедом введено n – теоретичний словник програми, тобто словниковий запас, необхідний для написання програми з урахуванням того, що необхідна функція вже реалізована в даній мові і, отже, програма зводиться до виклику цієї функції.

Використовуючи n , Холстед вводить оцінку:

- теоретичний обсяг програми, що максимально компактно реалізує даний алгоритм (V'):

$$V' = n * \log 2n \quad (2.4),$$

де n – теоретичний словник програми.

– складність програми D ;

– рівень програми (рівень якості програмування) L ;

– теоретична довжина програми (для стилістично коректних програм відхилення N від N' не перевищує 10%):

$$N' = n_1 * \log_2(n_1) + n_2 * \log_2(n_2) \quad (2.5);$$

– рівень якості програмування

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

(для ідеальної програми $L=1$);

– рівень якості програмування, заснований лише на параметрах реальної програми без урахування теоретичних параметрів

$$L' = (2 * n_2) / (n_1 * N_2) \quad (2.7)$$

– складність розуміння програми

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

– трудомісткість кодування програми

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

– інформаційний зміст програми - дана характеристика дозволяє визначити розумові витрати на створення програми

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

- оцінка необхідних інтелектуальних зусиль при розробці програми, що характеризує число необхідних елементарних рішень при написанні програми[^]

$$E = N' * \log_2(n / L) \quad (2.11).$$

Метрика Холстеда «інтелектуальності» - для виміру “інтелектуального” змісту алгоритму, інваріантне по відношенню до мов реалізації: $I = L * V$.

Для програмних кодів, що наведені у додатках А (лінійна програма), Б (програма із вказівниками), В (програма з модулями) було визначено **словники** цих програм (додатки Г, Д, Е) за формулою (2.1):

$$n_1 = 43 + 9 = 52$$

$$n_2 = 46 + 9 = 55$$

$$n_3 = 28 + 23 = 51$$

Найбільший словник має програма з вказівниками.

Довжина програм визначалася за формулою (2.2):

$$N1 = 181 + 41 = 222; N2 = 196 + 41 = 237; N3 = 209 + 89 = 298$$

Отже, найбільшу довжину має програма з модулями ($N3 = 298$).

Теоретична довжина програм визначалася за формулою (2.5):

$$N'1 = 43 * \log_2(43) + 9 * \log_2(9) = 233,06 + 28,53 = 261,59$$

$$N'2 = 46 * \log_2(46) + 9 * \log_2(9) = 253,92 + 28,53 = 282,45$$

$$N'3 = 28 * \log_2(28) + 23 * \log_2(23) = 134,4 + 103,96 = 238,96$$

Найбішу теоретичну довжину має програма із вказівниками.

Обсяг програм визначався за формулою (2.3)

$$V1 = 222 * \log_2(52) = 1265,4$$

$$V2 = 237 * \log_2(55) = 1370,097$$

$$V3 = 298 * \log_2(51) = 1690,256$$

Згідно розрахунку, найбільший обсяг має програма з модулями: ($V3 = 1690,256$).

Теоретичний обсяг програм обчислювався за формулою (2.4):

$$V'1 = 28 * \log_2(28) = 134,4$$

$$V'2 = 35 * \log_2(35) = 179,515$$

$$V'3 = 54 * \log_2(54) = 310,77$$

Найбільший теоретичний обсяг має програма з модулями ($V'3 = 310,77$).

Рівень якості програмування (2.6):

$$L1 = V'1 / V1 = 134,4 / 1265,4 = 0,1$$

$$L2 = V'2 / V2 = 179,515 / 1370,097 = 0,13$$

$$L3 = V'3 / V3 = 310,77 / 1690,256 = 0,18$$

Найвищим є рівень якості програмування у програма з модулями.

Рівень якості програмування, заснований лише на параметрах реальної програми без урахування теоретичних параметрів (2.7):

$$L'1 = 18 / 1763 = 0,01; L'2 = 18 / 1886 = 0,009; L'3 = 56 / 2492 = 0,02$$

Найнижчий рівень якості, згідно розрахунків, має програма із вказівниками: $L'2 = 0,009$, а найвищий- програма з модулями: $L'3 = 0,02$.

Складність програми визначається за формулою (2.9):

$$D1 = 10; D2 = 7,69; D3 = 5,55$$

Згідно розрахунків, найбільш складною виявилась лінійна програма.

Інформаційний зміст програми визначається за формулою (2.10):

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

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

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

Найбільшим інформаційним змістом характеризується програма з модулями: $I3 = 304,55$, найнижчий — $I2 = 126,54$.

2.2. Застосування метрики Маккейба

Однією з груп оцінок складності програм є метрики складності потоку керування програм. Як правило, за допомогою цих оцінок оперують або щільністю керуючих переходів усередині програм, або взаємозв'язками цих переходів. І в тому чи в іншому випадку стало традиційним уявлення програм у вигляді керуючого орієнтованого графа $G(V, E)$, де V – вершини, які відповідають операторам, а E – дуги, що відповідають переходам. У дузі (U, V) – вершина V є вихідною, а U – кінцевою [7].

Для представлення програми використовується потоковий граф, особливостями якого є:

- Граф будується відображенням керуючої структури програми.
- У ході відображення закривають дужки умовних операторів і операторів циклів (end if; end loop) розглядаються як окремі (фіктивні) оператори.
- Вузли (вершини) потокового графа відповідають лінійним ділянкам програми, включають один або кілька операторів програми.
- Дуги потокового графа відображають потік управління у програмі (передачі управління між операторами). Дуга – це орієнтоване ребро.

- Розрізняють операторні і предикатні вузли. З операторного вузла виходить одна дуга, а з предикатного – дві дуги.
- Предикатні вузли відповідають простим умовам у програмі.

Складна умова програми відображається в кілька предикатних вузлів. , при цьому можуть бути використані одна чи кілька булевих операцій (OR, AND). Уперше графічне представлення програм було запропоновано Мак-Кейбом. Основною метрикою складності він пропонує вважати цикломатичну складність графа програми, або, ще називають, **циклوماتичне число Маккейба**, що характеризує трудомісткість тестування програми [12].

Цикломатичне складність – метрика програмного забезпечення, яка забезпечує кількісну оцінку логічної складності програми. У способі тестування базового шляху цикломатична складність визначає:

- кількість незалежних шляхів у базовому безлічі програми.
- верхню оцінку кількості тестів, яке гарантує одноразове виконання всіх операторів.

Цикломатичну складність можна обчислити декількома способами:

- Цикломатична складність визначається за формулою:

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

де m – кількість дуг, n – кількість вузлів *потокowego* графа.

- Цикломатичне складність формується як

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

де p – кількість предикатних вузлів в потоковому графі G .

Для обчислення цикломатичного числа Мак-Кейба $Z(G)$ застосовується формула $Z(G) = m - n + 2p$ (2.14),

де m – число дуг *орієнтованого* графа G ;

n - число вершин;

p – число компонентів зв'язності графа.

Цикломатичне число Мак-кейба – визначає кількість незалежних контурів у повнозв'язному графі і, як наслідок, кількість різних шляхів, що ведуть з початкової вершини до кінцевої.

Число компонентів зв'язності графа можна розглядати як кількість дуг, які необхідно додати для перетворення графа у сильнозв'язний. Сильнозв'язний називається граф, будь-які дві вершини якого взаємнодосяжні. Для графів коректних програм, тобто графів, які не мають недосяжних від точок входу ділянок і "висячих" входу і виходу, сильнозв'язний граф, як правило, отримують шляхом замикання однієї вершини, що позначає кінець програми, на вершину, що позначає точку входу в цю програму.

Незалежним називається будь-який шлях, що вводить новий оператор обробки або нову умову. У термінах потокового графа незалежний шлях повинен містити дугу, що не входить у раніше визначені шляхи. Шлях починається у початковому вузлі, а закінчується в кінцевому вузлі графа. Незалежний шлях формуються в порядку від найкоротшого до найдовшого. По суті $Z(G)$ визначає число лінійно незалежних контурів в сильнозв'язному графі. Інакше кажучи, цикломатичне число Мак-Кейба показує необхідне число проходів для покриття всіх контурів сильнозв'язного графа або кількість тестових прогонів програми, необхідних для вичерпного тестування за критерієм "працює кожна гілка".

Для орієнтованого графа (додаток Ж) цикломатична складність $Z(G)$ була визначена за формулою (2.12): $Z(G) = m - n + 2 = 20 - 14 + 2 = 8$,

де m – число дуг орієнтованого графа, $m = 20$

n – число вершин, $n = 14$;

p – число компонентів зв'язаного графа, $p = 1$.

Застосуємо ще один варіант для знаходження складності графа для перевірки: P – кількість предикатних вузлів в потоковому графі G , $P = 7$, тоді:

$$Z(G) = P + 1 = 7 + 1 = 8$$

$$\text{Цикломатичне число: } Z(G) = m - n + 2p = 20 - 14 + 2 = 8$$

При оцінці складності програми з використанням цикломатичного числа Мак-Кейба діє правило: якщо цикломатичне число більше за $Z > 10$: програма має зайву складність і її слід розбити на складові частини з меншим значенням цикломатичного числа. У даному випадку, програма не має зайвої складності.

2.3. Застосування метрик визначення складності програмного продукту (метрика Чепіна, Джілбі Т.)

2.3.1. Призначення метрики Чепіна

Суть методу полягає в оцінці інформаційної міцності окремо взятого програмного модуля за допомогою аналізу характеру використання змінних зі списку вводу-виводу [13].

Всі змінні вводу-виводу розбиваються на 4 функціональні групи:

1. Р – вводяться змінні для розрахунків і для забезпечення виведення,
2. М – модифікуються, або створювані всередині програми змінні,
3. С – змінні, що беруть участь в управлінні роботою програмного модуля (керуючі змінні),
4. Т – які не використовуються в програмі («паразитні») змінні.

Оскільки кожна змінна може виконувати одночасно кілька функцій, необхідно враховувати її в кожній відповідній функціональній групі. Метрика Чепіна обчислюється за формулою: $Q = a_1 * P + a_2 * M + a_3 * C + a_4 * T$,

де a_1, a_2, a_3, a_4 – вагові коефіцієнти для врахування рівня впливу на складність програми кожної функціональної групи.

Найбільшу вагу має функціональна група С, так як вона впливає на потік управління програми, ваговий коефіцієнт $a_3 = 3$. Вагові коефіцієнти інших груп розподіляються наступним чином: $a_1 = 1, a_2 = 2, a_4 = 0.5$. Ваговий коефіцієнт a_4 групи Т не дорівнює нулю, оскільки

«паразитні» змінні не збільшують складність потоку даних програми, але іноді ускладнюють її розуміння.

З урахуванням вагових коефіцієнтів:

$$Q = P + 2M + 3C + 0.5T \quad (2.15)$$

Для програмних кодів, що наведені у додатках А (лінійна програма), Б (програма із вказівниками), В (програма з модулями) було визначено змінні Р, М, С, Т (табл.2.1) та виконано розраховунки метрики Чепіна за формулою (2.15):

$$Q1 = 4 + 6 + 3 + 0.5 = 13.5; \quad Q2 = 4 + 6 + 3 + 0 = 13; \quad Q3 = 4 + 6 + 3 + 0 = 13$$

Таблиця 2.1

Значення змінних для розрахунку метрика Чепіна

	Лінійна програма	Програма з вказівниками	Програма з модулями
Р	4	4	4
М	3	3	3
С	1	1	1
Т	0	0	0

2.3.2. Розрахунок метрики Джілбі Т.

Метрика Джілбі Т. відноситься до кількісних метрик і показує складність програмного забезпечення на основі насиченості програми умовними операторами або операторами циклу [14]. Дана метрика визначає складність написання і розуміння програми, а при додаванні такого показника, як максимальний рівень вкладеності умовних і циклічних операторів, ефективність даної метрики значно зростає. Логічна складність програми визначається як насиченість програми виразами IF_THEN_ELSE. При цьому вводяться дві характеристики:

1) CL – абсолютна складність програми, що характеризується кількістю операторів умови та циклів;

2) cl – відносна складність програми, що характеризується насиченістю програми операторами умови, тобто cl визначається як відношення CL до загальної кількості операторів:

$$cl = CL / N \quad (2.16),$$

де CL – абсолютна складність програми,

N – кількість всіх операторів.

Для програмних кодів, що наведені у додатках А (лінійна програма), Б (програма із вказівниками), В (програма з модулями) було визначено кількість операторів умови та циклів: $CL1 = 7$; $CL2 = 7$; $CL3 = 7$

Відносна складність програм за формулою (2.16):

$$cl1 = 7 / 34 = 0.20; \quad cl2 = 7 / 38 = 0,18; \quad cl3 = 7 / 52 = 0,13$$

Згідно розрахунків, найбільшу відносну складність має лінійна програма: $cl1 = 0.20$.

РОЗДІЛ 3. ЗАСОБИ АВТОМАТИЗОВАНОГО ОЦІНЮВАННЯ ЯКОСТІ ПЗ

3.1. Передумови автоматизованого тестування

Автоматизоване тестування та автоматизовані тести є важливою частиною розробки програмного забезпечення. Автоматизовані тести - це скрипти чи програми, які виконуються автоматично для перевірки функціональності або властивостей програми. Вони використовуються для забезпечення високої якості продукту, ефективності та швидкості тестування.

Основні переваги автоматизованих тестів включають:

1. **Ефективність:** Автоматизовані тести можуть бути виконані швидше і ефективніше, ніж ручні тести, що дозволяє зекономити час та ресурси.
2. **Повторюваність:** Тести можуть бути легко повторені без втрати точності, що особливо важливо при регулярних випусках програми.
3. **Покриття:** Автоматизовані тести дозволяють охопити велику кількість тестових сценаріїв та даних, що допомагає виявляти більше дефектів.
4. **Надійність:** Тести виконуються без помилок та відповідно до зазначених критеріїв, що забезпечує надійність результатів.
5. **Підтримка нічного тестування:** Тести можуть бути виконані автоматично під час нічних годин, що забезпечує неперервний процес тестування.
6. **Виявлення дефектів:** Автоматизовані тести допомагають швидше виявляти та локалізувати дефекти в коді [11].

Незважаючи на ці переваги, важливо також враховувати, що автоматизація тестів не підходить для всіх видів тестування, і для деяких випадків ручне тестування може бути більш ефективним.

Важливо враховувати, що хоча автоматизовані тести забезпечують багато переваг, їх ефективність також залежить від правильного планування і використання. Вибір тестових сценаріїв для автоматизації, належне управління тестовими даними і встановлення стабільності тестового середовища — це важливі аспекти автоматизації. Крім того, слід враховувати, що автоматизовані тести не можуть замінити творчого підходу ручного тестування, особливо в тих випадках, коли потрібна людська інтуїція, чутливість до користувацького досвіду або тестування нових функцій.

Здійснення автоматизованих тестів також вимагає відповідного навчання команди, розробників та тестувальників, щоб вони володіли необхідними навичками і інструментами. Важливо проводити регулярне оновлення автоматизованих тестів для відповідності змінам у програмному забезпеченні.

У підсумку, впровадження автоматизованих тестів може значно полегшити процес тестування, зменшити час, витрачений на виявлення та виправлення дефектів, і покращити якість програмного продукту. Однак успіх такої ініціативи залежить від ретельного планування і правильного використання автоматизації відповідно до конкретних вимог проекту.

Налаштування і підтримка інфраструктури для автоматизованих тестів також варто уваги, оскільки це забезпечить стабільність і надійність процесу тестування. Розробка ретельних звітів та моніторинг результатів автоматизованих тестів допоможе ефективно виявляти та вирішувати проблеми [6].

Також, варто зазначити, що в процесі автоматизації тестів слід пам'ятати про потребу постійного удосконалення і оптимізації тестових скриптів. Це дозволить уникати зайвих витрат часу на виконання тестів та підтримку автоматизованих сценаріїв.

Загалом, правильно впроваджена автоматизація тестів стає важливою частиною розробки програмного забезпечення, сприяючи прискоренню процесу виходу нового функціоналу та забезпечуючи високу якість продукту.

Продовження автоматизації тестів передбачає інтеграцію з системами для неперервної інтеграції та постійної поставки (CI/CD). Це дозволяє автоматизовано проводити тести при кожному зміні коду, що сприяє вчасному виявленню помилок та швидкому випуску оновлень.

Підтримка розширеного тестування, такого як навчання машин, може значно підвищити ефективність автоматизованих тестів. Використання аналізу даних та інтелектуальних алгоритмів дозволяє створювати більш складні та розширені тести для виявлення складних дефектів.

Окрім того, важливо пам'ятати про тестування безпеки та враховувати потенційні загрози для програмного продукту. Впровадження автоматизованих засобів тестування безпеки дозволить виявляти та усувати уразливості ще на етапі розробки.

Усі ці аспекти разом забезпечують ефективну та надійну систему автоматизованого тестування, що є важливим етапом в розробці високоякісного програмного забезпечення. Додатково, автоматизовані тести дозволяють створювати швидкі, повторювані та об'єктивні перевірки функціональності програмного продукту. Це допомагає зменшити людський вплив на тестування та забезпечити однакові умови для кожного запуску тестів [19].

Важливим аспектом є також можливість масштабування тестів для різних конфігурацій та платформ. Автоматизація дозволяє швидко виконувати тести на різних пристроях, операційних системах та браузерах, що є критичним для забезпечення широкого охоплення аудиторії користувачів.

Завершенням процесу автоматизованого тестування є генерація детальних звітів та логів, які допомагають команді розробників швидко локалізувати та виправляти виявлені проблеми. Це сприяє підвищенню якості програмного продукту та зменшенню часу, необхідного для виправлення помилок.

Отже, автоматизоване тестування є ключовим елементом сучасного виробництва програмного забезпечення, сприяючи швидкому впровадженню та підтримці високоякісних та надійних програм.

У додаток до переваг, зазначених вище, важливою перевагою автоматизованих тестів є їхній внесок у забезпечення безпеки програмного продукту. Автоматизоване тестування може допомагати виявляти потенційні уразливості та помилки безпеки, забезпечуючи раннє їх виявлення та усунення.

Крім того, автоматизація тестування є ефективним рішенням для регресійного тестування, коли необхідно перевіряти, чи впливають нові зміни або додатки на вже існуючий функціонал. Це зменшує ризик введення нових помилок під час розробки та підтримки програмного продукту.

Загалом, автоматизовані тести є необхідним інструментом для сучасних розробників, який сприяє швидкому виконанню тестів, забезпечує стабільність та надійність програмного продукту та спрощує процес його впровадження та підтримки.

Позитивні аспекти автоматизованих тестів розширюються на покращення ефективності розробки та зменшення тривалості циклів випуску програмного продукту. Застосування автоматизованих тестів дозволяє вчасно виявляти та виправляти помилки, знижуючи час, необхідний для виправлення дефектів [21].

Крім того, автоматизація тестування полегшує процес сумісності програмного забезпечення з різними платформами та пристроями. Здатність швидко виконувати тести на різних конфігураціях сприяє

впевненості в тому, що програмний продукт працює стабільно в різних середовищах.

Усі ці аспекти спільно визначають автоматизовані тести як важливий елемент життєвого циклу розробки програмного продукту, допомагаючи покращити його якість, ефективність та безпеку.

Наслідком цього є можливість швидко реагувати на зміни в коді та внесення нового функціоналу, адже автоматизовані тести можуть бути швидко адаптовані до змін. Це робить їх ефективним інструментом для розробників у сучасному середовищі, де швидкість та гнучкість є ключовими факторами успіху.

Крім того, автоматизовані тести можуть знижувати ризик людської помилки під час тестування, оскільки вони виконують однакові завдання точно та послідовно. Це полегшує виявлення та виправлення дефектів на ранніх етапах розробки, що, в свою чергу, допомагає зменшити витрати на виправлення проблем у подальшому.

Таким чином, автоматизовані тести визначаються як ефективний інструмент для поліпшення процесу розробки програмного продукту, забезпечуючи швидкість, надійність та якість усього життєвого циклу продукту.

Крім цього, використання автоматизованих тестів полегшує масштабування тестових сценаріїв та їх виконання на різних конфігураціях та платформах. Це особливо важливо в умовах швидко змінюючогося середовища розробки та випуску програмного забезпечення.

Також варто відзначити, що автоматизовані тести дозволяють проводити тестування в нічний час або без прямої участі користувача, що сприяє ефективному використанню робочого часу та прискорює цикл розробки [18].

Загалом, використання автоматизованих тестів є стратегічно важливим для забезпечення високої якості програмного продукту та збільшення продуктивності розробницького процесу.

Однак, важливо враховувати, що хоча автоматизовані тести забезпечують численні переваги, вони не є універсальним рішенням. Наприклад, деякі аспекти тестування, такі як тестування інтерфейсу користувача та аспекти, пов'язані зі специфічними вимогами до дизайну, можуть вимагати ручного тестування.

Додатково, слід враховувати витрати на розробку та підтримку автоматизованих тестів, які можуть бути значними, особливо в початковій фазі впровадження автоматизації. Однак з плином часу та зростанням обсягів розробки ці витрати можуть бути виправдані завдяки збільшенню ефективності та якості тестування.

У кінці дня, успішне використання автоматизованих тестів потребує уважного планування, правильного вибору інструментів та регулярного оновлення тестових сценаріїв.

Найбільш ефективно використання автоматизованих тестів вимагає стеження за змінами в програмному забезпеченні та адаптації тестових сценаріїв до нових вимог. Розробники та тестувальники повинні тісно співпрацювати для підтримання актуальності автоматизованих тестів і вчасного виявлення та виправлення проблем.

Додатково, автоматизовані тести можуть бути використані для неперервної інтеграції та неперервної доставки, що сприяє швидкішому впровадженню нових функцій та зменшенню ризиків.

У великих проектах, де обсяги тестування великі, автоматизація є важливим елементом стратегії тестування. Однак важливо пам'ятати, що автоматизовані тести можуть бути лише однією частиною загальної стратегії тестування, а їх ефективність залежить від правильного планування та управління [12].

Для максимізації користі від автоматизації тестів важливо визначити критерії вибору тестових сценаріїв для автоматизації. Переважно це тестові випадки, які виконуються часто, мають стабільний функціонал і вимагають повторюваності.

При розробці та підтримці автоматизованих тестів важливо слідкувати за їхньою стабільністю, адже зміни в коді програми можуть впливати на їх ефективність. Крім того, розуміння принципів автоматизованого тестування та вміння правильно вибирати інструменти можуть значно покращити результати тестового процесу.

Далі, для оптимального використання автоматизованих тестів, слід враховувати потребу у їхній регулярній перевірці та апдейті в разі змін у програмному продукті. Це включає в себе перегляд та оновлення тестових сценаріїв, виправлення помилок у скриптах автоматизації та адаптацію до змін у вимогах до програмного забезпечення.

Важливо пам'ятати, що автоматизація тестування не замінює ручне тестування, але доповнює його. Ручне тестування залишається необхідним для виявлення аспектів, які важко або неможливо автоматизувати, таких як користувацький досвід та емоції [6].

У випадку ефективного впровадження автоматизації тестування можна очікувати зменшення часу на тестування, збільшення точності виявлення помилок та підвищення загальної якості програмного продукту.

Далі, важливим етапом в процесі автоматизованого тестування є моніторинг та аналіз результатів тестів. При виявленні проблем чи помилок важливо провести їх аналіз, визначити причини і внести необхідні корективи в тестові сценарії чи в сам програмний продукт.

Для підтримки ефективної автоматизації важливо мати систему контролю версій для збереження та відстеження змін в тестових сценаріях та коді автоматизації. Це дозволяє зберігати стабільну версію тестової бази та відновлювати попередній стан в разі необхідності.

Узагальнюючи, автоматизоване тестування вимагає системного та дбайливого підходу. Постійне вдосконалення тестових сценаріїв, регулярний моніторинг результатів і ефективна комунікація між командами грають ключову роль у забезпеченні високої якості програмного продукту.

Далі, для забезпечення стабільності та надійності автоматизованих тестів, важливо враховувати зміни у вихідних даних та відслідковувати їх вплив на тестові сценарії. Регулярна перевірка та оновлення тестових наборів дозволяє враховувати зміни в функціоналі програми та уникнути застарілих тестів.

Однією з важливих практик є також використання параметризації у тестових сценаріях, що дозволяє запускати один і той же тестовий сценарій з різними вхідними даними. Це забезпечує більшу гнучкість та розширює покриття тестів.

Необхідно також розглядати можливість інтеграції автоматизованих тестів у процес неперервної інтеграції (CI) та неперервної доставки (CD). Це дозволяє автоматично запускати тести при змінах у коді та швидко виявляти проблеми [26].

Загалом, ефективне впровадження автоматизованих тестів вимагає постійного вдосконалення, адаптації до змін, та співпраці всіх членів розробницької команди.

Далі важливо звертати увагу на підтримку та покращення тестової інфраструктури. Це включає в себе забезпечення ефективного моніторингу та звітності про виконання тестів, а також зберігання результатів для аналізу.

Паралельне виконання тестів на різних конфігураціях та пристроях може покращити ефективність та розширити покриття тестів. Також слід враховувати можливість використання хмарних сервісів для автоматизації тестування на різних платформах.

Збереження чіткої документації та опису структури та призначення автоматизованих тестів є важливим етапом. Це полегшує роботу новим членам команди, дозволяє швидше реагувати на зміни та забезпечує легшу розуміння та підтримку тестового коду.

Загалом, для успішного використання автоматизованих тестів важливо удосконалювати їхню структуру, регулярно адаптувати до змін у проекті та забезпечувати їхню неперервну інтеграцію в розробницький процес.

Далі, з огляду на зростання розміру та складності проекту, важливо вдосконалювати стратегії вибору тестових сценаріїв. Це включає в себе виявлення критичних частин системи та покриття їх тестами. Зосередження на найважливіших компонентах дозволяє оптимізувати використання ресурсів та забезпечити більш швидку зворотню зв'язку.

Підтримка автоматичного аналізу відхилень та виявлення помилок також є ключовою. Використання систем моніторингу та інструментів для трасування та аналізу виключень може значно спростити процес виявлення та виправлення проблем [26].

Важливим елементом є інтеграція тестової автоматизації в процес неперервної інтеграції та неперервної доставки (CI/CD). Забезпечення автоматичної перевірки коду при кожному зміні та автоматичної поставки стабільних версій може сприяти швидкому виявленню та вирішенню проблем.

Усі зазначені аспекти сприяють покращенню ефективності, стабільності та надійності автоматизованого тестування в розробці програмного забезпечення.

Далі слід звертати увагу на розширення покриття тестами для різних конфігурацій, пристроїв та середовищ виконання. Тестування на різних платформах та браузерях допомагає гарантувати, що програма працює стабільно та коректно в різних умовах.

Важливо також вдосконалювати систему збору та аналізу результатів тестування. Це включає в себе автоматизацію генерації звітів, виявлення та фіксацію дефектів, а також відстеження їх статусу протягом усього циклу розробки.

Необхідно постійно адаптувати стратегії тестування до змін у вимогах до програмного продукту та внесених змін. Розширення тестових сценаріїв для нового функціоналу та перевірка відповідності їх актуальним бізнес-вимогам є ключовим етапом у забезпеченні якості.

Загалом, неперервне вдосконалення та адаптація стратегій автоматизованого тестування є важливими компонентами успішного процесу розробки програмного забезпечення.

Далі важливо розглядати можливості використання інструментів для тестування безпеки програмного забезпечення. Врахування аспектів безпеки на ранніх етапах розробки та впровадження ефективних заходів є критичним для запобігання можливим атакам та забезпечення захисту користувачів [19].

Також слід розвивати методики та практики тестування на витривалість, щоб переконатися в стабільності та ефективності програмного забезпечення під час тривалого функціонування в реальних умовах. Це може включати тестування великого обсягу даних, періодів навантаження та інших сценаріїв, що дозволяють оцінювати продуктивність та надійність системи.

Налаштування процесу неперервної інтеграції та неперервної доставки (CI/CD) також є важливою складовою ефективного тестування. Це дозволяє автоматизувати та прискорити процеси тестування та впровадження нового функціоналу, зменшуючи час циклу розробки та забезпечуючи швидке виявлення та виправлення помилок.

Узагальнюючи, ефективне впровадження автоматизованих тестів, фокус на безпеці та витривалості, адаптація до змін у вимогах та

використання сучасних методів CI/CD є ключовими аспектами успішного тестування програмного забезпечення.

Далі слід активно вдосконалювати засоби для автоматизації тестування, такі як інструменти для юніт-тестування, інтеграційні тести та тести на прийняття. Розвиток повністю автоматизованих тестових наборів сприяє швидкому виявленню помилок і полегшує процес впровадження змін у код.

Значна увага також повинна приділятися тестуванню безпеки. Перевірка на наявність потенційних уразливостей, аудит безпеки та регулярне вдосконалення заходів захисту є необхідними для запобігання можливим кібератакам.

Паралельно з розширенням автоматизованих тестів слід розглядати можливості впровадження тестів на рівні взаємодії з користувачем. Це дозволяє виявляти проблеми, пов'язані з користувацьким інтерфейсом та взаємодією з системою в реальних умовах [34].

Важливо також пам'ятати про постійне навчання команди тестування та впровадження сучасних методологій, таких як DevOps, для покращення співпраці між розробкою та тестуванням, що сприяє швидкому виправленню помилок та постійному вдосконаленню якості програмного забезпечення.

3.2. Види автоматизованих тестів

Автоматизовані тести можуть включати в себе різні типи, спрямовані на тестування різних аспектів програмного забезпечення. Основні види автоматизованих тестів включають:

1. Юніт-тести:

- **Опис:** Тести, що перевіряють правильність роботи окремих компонентів або модулів програми.

- Мета: Виявлення дефектів на ранніх етапах розробки та забезпечення коректної роботи окремих частин програми.

2. Інтеграційні тести:

- Опис: Тести, що перевіряють взаємодію між компонентами або модулями програми.
- Мета: Виявлення проблем, пов'язаних з інтеграцією, та переконання, що взаємодія між компонентами працює правильно.

3. Функціональні тести:

- Опис: Тести, що перевіряють функціональність програми відповідно до визначених вимог.
- Мета: Забезпечення відповідності програмного забезпечення функціональним вимогам замовника.

4. Прийняття та регресійні тести:

- Опис: Тести, що використовуються для перевірки, чи відповідає програмне забезпечення заданим критеріям та чи не виникають нові дефекти після внесення змін.
- Мета: Забезпечення якості випуску нових версій програми та виявлення можливих проблем після внесення змін.

5. Тести безпеки:

- Опис: Тести, спрямовані на виявлення уразливостей та визначення рівня безпеки програми.
- Мета: Захист від потенційних загроз та забезпечення надійності системи.

6. Тести взаємодії з користувачем:

- Опис: Тести, що перевіряють взаємодію програми з користувачем через графічний інтерфейс або інші засоби введення.

- Мета: Переконавання в тому, що програма коректно реагує на введення користувача та забезпечення зручності використання.

Розуміння і використання різних видів автоматизованих тестів дозволяє забезпечити високу якість програмного забезпечення та швидке виявлення можливих проблем [15].

Важливою частиною процесу тестування є використання різних видів автоматизованих тестів, які охоплюють різноманітні аспекти функціональності та продуктивності програмного забезпечення. Зокрема, визначення коректності реакції програми на введені дані та різні сценарії використання забезпечується за допомогою функціональних тестів. Виявлення можливих вразливостей та недоліків допомагають тести безпеки. Для забезпечення оптимальної продуктивності та надійності використовуються навантажувальні та тести відновлення. Комплексне тестування також включає тести сумісності, які перевіряють взаємодію програми з різними середовищами, а також тести локалізації для перевірки коректності відображення в різних мовах та регіонах. Важливим етапом є також тести відновлення та стійкості, що перевіряють здатність програми відновлювати роботу після негативних сценаріїв. Комплексне використання цих видів тестів допомагає забезпечити якість та надійність програмного забезпечення у різних аспектах його функціонування.

Також, у рамках автоматизованого тестування, використовуються регресійні тести, спрямовані на перевірку того, чи не виникають нові дефекти після внесення змін у програмний код. Тести взаємодії перевіряють, як система співпрацює з іншими компонентами чи сервісами. Також, тестування продуктивності дозволяє визначити, як швидко та ефективно програма виконує свої функції при різних умовах завантаження.

Паралельно з цим, важливо використовувати тести на користувацький досвід (UI/UX), які дозволяють переконатися, що програмний продукт відповідає вимогам щодо зручності та естетичності інтерфейсу для кінцевого користувача.

У рамках автоматизованих тестів важливим етапом є тестування безпеки, що дозволяє виявити потенційні вразливості та забезпечити захист системи від можливих атак. Тестування сумісності визначає, як добре програма працює на різних платформах та пристроях, забезпечуючи широкий охоплення аудиторії [25]

Варто також зазначити, що автоматизовані тести сприяють підтримці неперервної інтеграції та постійної доставки, що важливо для швидкого впровадження нових функцій та виправлення помилок.

Використання різних видів автоматизованих тестів узгоджене з кращими практиками розробки програмного забезпечення і сприяє створенню стійких, функціональних та безпечних продуктів.

Далі, автоматизовані тести можуть включати тести на відмовостійкість, спрямовані на перевірку того, як система поводить себе під час виникнення помилок або невірностей, та визначення її здатності до відновлення. Тестування продуктивності оцінює роботу програми при великих обсягах даних чи високому навантаженні, забезпечуючи ефективність та швидкодію.

Важливим етапом в автоматизованих тестах є також тестування користувацького інтерфейсу, щоб забезпечити зручність взаємодії та задоволення користувачів. Тестування відмінностей слідкує за виявленням змін та впливом виправлень на існуючий функціонал.

Узагальнюючи, використання різних видів автоматизованих тестів допомагає забезпечити якість, надійність та ефективність програмного забезпечення, прискорюючи процес розробки та впровадження нових функцій.

3.3. Основні інструменти для автоматизованого тестування

Автоматизоване тестування використовує різноманітні інструменти для ефективності та автоматизації процесу. Декілька основних інструментів включають:

1. Selenium: Це потужний інструмент для автоматизованого тестування веб-додатків. Він дозволяє створювати та виконувати тестові сценарії для веб-інтерфейсів.
2. Appium: Використовується для автоматизації тестування мобільних додатків на платформах Android та iOS.
3. JUnit та TestNG: Фреймворки для тестування, які дозволяють структурувати та виконувати тестові сценарії для додатків на мовах Java та інших.
4. Jenkins: Інструмент для автоматизації процесу CI/CD, який дозволяє виконувати тести під час різних етапів розробки.
5. Postman: Використовується для тестування API. Надає можливості взаємодії та перевірки правильності роботи API.
6. Cucumber: Фреймворк для автоматизованого тестування на основі використання природної мови (BDD). Дозволяє створювати тестові сценарії, які зрозумілі не тільки розробникам, але й бізнес-аналітикам.
7. LoadRunner: Використовується для виконання тестів на великих навантаженнях, для перевірки продуктивності та масштабованості систем.
8. TestComplete: Інтегрована платформа для автоматизації тестування на різних рівнях, включаючи тестування веб-додатків, мобільних додатків і десктопних додатків.

Використання цих інструментів дозволяє автоматизувати та полегшити процес тестування, забезпечуючи швидкий та надійний розробку програмного забезпечення [32].

Тестування програмного забезпечення (ПЗ) є ключовим етапом у розробці програм. Процес тестування поділяється на мануальне та автоматизоване. Автоматизоване тестування використовує різні підходи та інструменти для часткового заміщення людської участі під час рутинних процедур. В цьому огляді представлені різні програмні інструменти та технології з їхніми короткими описами та сферами використання.

Технології автоматизованого тестування включають:

1. Приватні рішення
2. Тестування під керуванням даними (DDT)
3. Тестування під керуванням ключовими словами (KDT)
4. Використання фреймворків
5. Запис та відтворення (Record & Playback)
6. Поведінкове тестування (BBT - Behaviour Driven Testing)

Зазначимо, що один інструмент може використовувати кілька технологій. Наприклад, Selenium IDE використовує технологію запису та відтворення, але всі його команди є тестуванням під керуванням ключовими словами. Фреймворки значно полегшують навчання фахівців з тестування програмного забезпечення, а використання технології запису та відтворення прискорює створення перших тестів навіть для тих, хто не має досвіду в автоматизованому тестуванні.

Інструменти Selenium включають також Selenium WebDriver і Selenium Server. Selenium WebDriver – це набір бібліотек для різних мов програмування, який дозволяє управляти браузером з програми, написаної на цій мові. Це надійний фреймворк автоматизації, який може працювати з будь-яким браузером. Selenium Server може приймати команди від віддаленого комп'ютера, де виконується сценарій

автоматизації, і виконувати їх у браузері. Кілька серверів Selenium можуть створювати розподілену мережу, яка називається Selenium Grid і забезпечує масштабування тестового середовища [28].

Selenium підтримує три типи команд:

- ****дії (actions)** - команди, які зазвичай керують станом додатка, виконуючи процедури, такі як "клацання за посиланням" або "вибір цієї операції". Якщо дію неможливо виконати або вона виконується з помилкою, тест переривається.
- ****считувачі (accessors)** - аналізують стан додатка і зберігають результати у змінні (наприклад, команда "storeTitle"). Вони також використовуються для автоматичного створення перевірок.
- **перевірки (assertions)** - подібні до считувачів, але, на відміну від них, перевіряють відповідність поточного стану додатка очікуваному.

Модульне тестування націлене на виявлення можливих дефектів і верифікацію функціонування програмних модулів, об'єктів, класів, які можна тестувати ізольовано. Для цього можна використовувати сімейство бібліотек XUnit, такі як TestNG і JUnit. Інтеграційне тестування розглядає взаємодію між програмними компонентами, а системне тестування перевіряє взаємодію між системами або апаратним забезпеченням. Прийомочне тестування служить для верифікації відповідності вимогам, бізнес-процесам і побажанням користувачів.

Всі технології автоматизованого тестування програмного забезпечення мають спільну особливість: вони спрямовані на технічні аспекти поведінки додатків. Однак вони також мають загальний недолік: їх важко використовувати для перевірки високорівневих сценаріїв користувача. Цей недолік старається вирішити тестування за принципами поведінки, де акцент робиться не на окремих технічних деталях, а на загальній працездатності додатка при вирішенні типових завдань користувачів [21].

Такий підхід не лише спрощує виконання ряду перевірок, але й полегшує взаємодію між розробниками, тестувальниками, бізнес-аналітиками та замовниками ПЗ. Його основою є формула "given – when – then" [19].

Given ("дано") описує початкову ситуацію, в якій знаходиться користувач у контексті його взаємодії з додатком. When ("коли") описує набір дій користувача в даній ситуації. Then ("тоді") описує очікувану поведінку додатка. Прикладом використання цієї технології є Serenity BDD [28]. Код для тесту має наступний вигляд (див. рис. 3.1):

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

Рис. 3.1. Код теста для Serenity BDD

Один з кроків Serenity BDD:

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

Рис. 3.2. Приклад для Serenity BDD.

Також створено звіт (див. рис. 3.3):

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

Рис. 3.3 Звіт проходження тесту для Serenity BDD

Тестування продуктивності (performance testing) – це аналіз "швидкісних показників" додатка при різних рівнях навантаження (за характером та обсягом).

Навантажувальне тестування (load testing) – дослідження здатності додатка зберігати встановлені показники якості при допустимому навантаженні та деякому перевищенні цих меж («запас прочности»).

Тестування стресу (stress testing) – аналіз поведінки додатка при неочікуваних змінах навантаження.

Об'ємне тестування (volume testing) – аналіз продуктивності системи при обробці різних обсягів даних.

Тестування масштабованості (scalability testing) – аналіз здатності додатка підвищувати продуктивність відповідно до збільшення кількості доступних ресурсів [11].

Конкурентне тестування (concurrency testing) – вивчення поведінки додатка в ситуаціях, коли йому доводиться обробляти велику кількість одночасно надходячих запитів, що викликає конкуренцію між цими запитами за ресурси (базу даних, пам'ять, канал передачі даних, дискову систему тощо).

Тестування надійності (reliability testing) - аналіз здатності додатка виконувати свої функції в заданих умовах протягом визначеного часу або кількості операцій.

Тестування відновлення (recoverability testing) – перевірка здатності додатка відновлювати свої функції та встановлений рівень продуктивності, а також відновлювати свої дані в разі критичної ситуації, що призводить до тимчасового чи часткового втрати працездатності додатка.

Тестування відмовостійкості (failover testing) – емуляція або реальне створення критичної ситуації з метою перевірки здатності додатка використовувати вбудовані механізми, що забезпечують

запобігання порушення працездатності, продуктивності та пошкодження даних [31].

Елементи тест-плану JMeter [19]:

- Групи потоків (Thread groups);
- Логічні контролери (Logic controller);
- Типові контролери (Sample generating controller);
- Слухачі (Listeners);
- Таймери (Timers);
- Перевірки (Assertions);
- Конфігураційні елементи (Configuration Elements).

****Групи потоків** – це вихідні точки будь-якого тест-плану. Всі контролери та зразки повинні бути в групі потоків. Інші елементи, такі як слухачі, можуть розташовуватися під тест-планом, в якому вони застосовуються до всіх потоків групи. Елемент групи потоків керує кількістю потоків, які JMeter буде використовувати для виконання тесту. Загалом можна встановити:

- Кількість потоків (користувачів) (Number of Threads);
- Час наростання (Ramp-Up Period);

-
- Кількість повторів для виконання тесту.

Інші тестові потоки використовуються для моделювання конкурентних з'єднань з сервером додатка.

Ramp-Up Period "повідомляє" JMeter, скільки часу буде потрібно для наростання кількості потоків до повного обсягу. Наприклад: якщо використовується 10 потоків, а Ramp-Up Period - 100 секунд, то JMeter витратить 100 секунд, щоб отримати всі 10 потоків. При цьому кожен потік буде починатися через 10 секунд (100/10) після того, як запустився попередній потік. Якщо є 30 потоків, і час наростання становить 120 секунд, то кожен наступний потік буде затримуватися на 4 секунди.

Loop count – кількість циклів тесту, forever – тест триватиме нескінченно [24].

Час наростання повинен бути достатньо довгим, щоб уникнути занадто великого навантаження в початковий момент тесту, і достатньо коротким, щоб останні потоки починали запускатися до закінчення перших.

Thread group також надає планувальник Scheduler. Слід клікнути на відповідну кнопку, щоб отримати доступ до додаткових полів, в яких користувач може встановити: - тривалість (Duration) тесту; - затримку запуску (Startup Delay); - час початку і закінчення тесту. Крім того, користувач може вибрати тривалість і затримку, щоб контролювати кожен потік окремо. Тут можна запускати тест в певний час, наприклад, вночі [31].

Процедура може бути розглянута як коротка програма, що функціонує в масштабах мініатюрної варіантної реалізації, і водночас, є складовою частиною основної програми або іншої процедури. Структура синтаксису процедури повністю відтворює синтаксис основної програми. Важливою особливістю є те, що процедури можуть містити свої власні внутрішні процедури та функції, розширюючи тим самим можливість вбудованого програмного коду.

Таблиця 3.1

Класи еквівалентності

Вхідні умови	Правильні класи еквівалентності	Неправильні класи еквівалентності
N	$2 \leq n \leq 50$	$n < 2$ $n > 50$ Символ
M	$2 \leq m \leq 50$	$m < 2$ $m > 50$ Символ

Елемент матриці	дійсне число	Символ
-----------------	--------------	--------

В контексті розробки програмних систем процедури розглядаються згідно з методологією проектування програм, оскільки вони виступають як ключові компоненти програмної архітектури. Ця методологія передбачає створення докладних специфікацій для кожної програмної одиниці, включаючи процедури.

Таблиця 3.2

Тести з правильних класів еквівалентності

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

Специфікація визначається як точний, однозначний і недвозначний опис, що створюється постановником завдання для подальшої імплементації програмістом. Цей опис включає концептуальне визначення програмної одиниці, опис потоків даних, виокремлення деталей алгоритму та, можливо, інші розділи, що залежать від обраного методу проектування.

Таблиця 3.3

Тести з неправильних класів еквівалентності

№ 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!

Важливим етапом є розробка тестів, які використовуються для перевірки правильності роботи підпрограм за допомогою методів, що базуються на специфікаціях, а також методів, що базуються на самому програмному коді.

Розробка програми передбачає знаходження максимального значення середнього арифметичного усіх елементів тих рядків вказаної матриці, які впорядковані за зростанням або спаданням. Для втілення цієї програми згідно визначеним вимогам було обрано середовище програмування Microsoft Visual Studio 2009 та мову програмування C++. Програма структурована у вигляді модулів та головної програми. У **додатку 3** наведено фрагмент коду є реалізацією цієї програми.

Результати тестування методом «Білого ящика» представлені у табл. 3.4

Результат тестування методом «Чорного ящику» представлені в табл. 3.5.

Тест на введення правильних даних на основі написаного вище коду наддано у **додатку I**.

Цей тест підготовлює введення користувача у вигляді рядка і перенаправляє `std::cin` для симуляції введення користувача. Потім він викликає головну функцію і перевіряє вивід, щоб переконатися, що програма працює правильно. Тест перевіряє, чи виводиться очікуване повідомлення з максимальним значенням.

У ході дослідження був розглянутий конкретний програмний код, призначений для пошуку максимального значення серед упорядкованих рядків матриці.

Таблиця 3.4

Результати іспитів методом «Білого ящика»

№ тесту	№ класу еквівалентності	Вхідні умови	Результат
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
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			

Таблиця 3.5

Результат тестування методом «Чорного ящика»

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

Код було розглянуто з точки зору певних метрик тестування, таких як:

- Модульне тестування: Було враховано введення правильних даних для перевірки коректності роботи програми у ситуації, коли у матриці є упорядковані рядки.
- Тестування взаємодії з користувачем: Програма була випробувана на можливість введення нових матриць у разі натискання клавіші 'Y', і вона коректно виконує повторні обчислення.

У ході тестування було виявлено, що програма працює правильно та відповідає вимогам. Однак, для більш глибокого аналізу якості коду та ефективності програми, можливо, буде доцільно розширити тестовий набір та врахувати інші аспекти тестування, такі як ефективність в разі великих обсягів даних та обробка винятків.

ВИСНОВКИ

У результаті проведеного дослідження теоретичних аспектів якості програмного забезпечення було встановлено, що якість ПЗ визначається як:

- набір властивостей продукту, що характеризують його здатність задовольнити встановлені або передбачувані потреби замовника;
- комплексна характеристика, що охоплює різноманітні аспекти, такі як функціональність, надійність, продуктивність та зручність у використанні.

При встановленні вимог до якості ПЗ та засобів його оцінювання необхідно керуватися діючими стандартами та враховувати умови його функціонування у конкретній сфері господарювання.

Для визначення комплексу критеріїв, що належним чином відображають властивості та характеристики програмного забезпечення з різних сторін, формується модель якості. Згідно стандарту ISO/IEC 25010, показниками якості ПЗ є: функціональність, надійність, зручність використання (практичність), продуктивність або ефективність, зручність супроводу, мобільність.

Інструментами оцінювання якості програмного забезпечення є метрики якості, статичне та динамічне тестування, автоматизоване тестування, визначення покриття програмного коду тестами.

Метрика якості ПЗ – це міра, яка дозволяє одержати чисельне значення деякої властивості ПЗ або його специфікацій. Отримане чисельне значення дозволяє зрозуміти в якій мірі система, її компоненти або процес відповідають встановленим нормам. Зокрема, метрика Мак-Кейба визначає цикломатичне число, що показує необхідне число проходів для покриття всіх контурів сильнозв'язного графа або кількість тестових прогонів програми, необхідних для вичерпного її тестування. Метрика Джілбі Т. відноситься до кількісних метрик і показує складність програмного забезпечення на основі насиченості програми

умовними операторами або операторами циклу. За допомогою метрики Чепіна аналізується складність потоку керування даними. Суть методу полягає в оцінці інформаційної міцності окремо взятого програмного модуля за допомогою аналізу характеру використання змінних з переліку введеної та оброблюваної інформації. Застосування метрики Холстеда дозволяє визначити складність програмного продукту в цілому.

Інструментами оцінювання якості ПЗ є також тестування, статичний та динамічний аналіз коду, визначення покриття програмного коду тестами. Статичний аналіз дозволяє виявляти потенційні проблеми на етапі розробки, тоді як динамічний аналіз дозволяє спостерігати за реальним виконанням програми та виявляти дефекти.

Аналіз покриття коду тестами дозволяє визначити, наскільки ефективно тестові приклади покривають весь обсяг коду. Забезпечення високого покриття тестами дозволяє виявляти більше дефектів та забезпечує більшу впевненість у стабільності програми.

Засоби для автоматизованого оцінювання якості ПЗ, (автоматизовані тести) дозволяють ефективно виявляти та виправляти помилки в програмному забезпеченні.

У цілому, комплексне 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.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

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.
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.
3. ДСТУ ISO/IEC 25010:2016 Інженерія систем і програмних засобів. Вимоги до якості систем і програмних засобів та її оцінювання (SQuaRE). Моделі якості системи та програмних засобів (ISO/IEC 25010:2011, IDT).
4. ДСТУ ISO/IEC/IEEE 12207:2018 (ISO/IEC/IEEE 12207:2017, IDT) Інженерія систем і програмних засобів. Процеси життєвого циклу програмних засобів
5. ISO/IEC/IEEE 29119 Software and systems engineering -- Software testing. URL:
https://en.wikipedia.org/wiki/ISO/IEC_29119#ISO/IEC/IEEE_29119-1:2013, Part 1: Concepts and definitions
6. Волочій Б.Ю. Технологія моделювання алгоритмів поведінки інформаційних систем. Вид-во НУ “Львівська політехніка”. Львів, 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.
https://nv.nltu.edu.ua/Archive/2018/28_6/33.pdf
9. Lavrishcheva, K. M. Prohramna inzheneriia: pidruchnyk. Kyiv: Akademperiodyka. 320 p. 2018. [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. NAU "KhAI". Radioelektronni i kompiuterni systemy, 2013, 5, c. 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, 2(2), 25–31.
14. Zalewski, Janusz, Kornecki, Andrew J., & Pfister, Henry L. Numerical Assessment of

Software Development Tools in RealTime Safety Critical Systems Using Bayesian Belief Networks. Retrieved from: <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. 588p.
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. Вид-во НУ “Львівська політехніка”. Львів, 2022.
21. Клієнт-серверна архітектура, URL: https://uk.wikipedia.org/wiki/%D0%9A%D0%BB%D1%96%D1%94%D0%BD%D1%82-%D1%81%D0%B5%D1%80%D0%B2%D0%B5%D1%80%D0%BD%D0%B0_%D0%B0%D1%80%D1%85%D1%96%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0
22. Огляд протоколу HTTP, URL: <https://developer.mozilla.org/ru/docs/Web/HTTP/Overview>
23. Тестування програмного забезпечення , URL: https://uk.wikipedia.org/wiki/%D0%A2%D0%B5%D1%81%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%B7%D0%B0%D0%B1%D0%B5%D0%B7%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%BD%D1%8F9.
24. A.B. Badiru, C.F. Rusnock, V.V. Valencia: Project Management for Research: A Guide for Graduate Students, CRC Press, 2019.
25. Complete Guide to Test Automation: Технічні, практичні, та інструменти для Building and Mainting Effective Software Projects , URL: <http://tisten.ir/wp-content/uploads/2019/01/Complete-Guide-to-TestAutomation-Techniques-Practices-and-Patterns-for-Building-andMaintaining-Effective-Software-Projects-Apress-2018-Arnon-Axelrod.pdf>
26. Driver requirements Maven , URL: https://www.selenium.dev/documentation/en/webdriver/driver_requirements/
27. Getting Started with Selenium for Automated Website Testing , URL: <https://wiki.saucelabs.com/display/DOCS/Getting+Started+with+Selenium+for+Automated+Website+Testing>

28. Installing Selenium libraries. URL:
https://www.selenium.dev/documentation/en/selenium_installation/installing_selenium_libraries/
29. IntelliJ IDEA , URL: <https://www.jetbrains.com/ru-ru/idea/>
30. Oracle Java , URL: <https://www.oracle.com/java/>
31. Selenium Webdriver , URL: https://drive.google.com/file/d/0B7TBmsv_w76nQ1FfQVdyWWRXYk0/view
32. Selenium with Java , URL:
<https://www.browserstack.com/guide/selenium-with-java-for-automated-test>
33. Strategies for Testing Web Applications від Client Side , URL:
<https://cs.fit.edu/media/TechnicalReports/cs-2003-11.pdf>
34. Top 10 Automation Testing Tools in 2021 , URL:
<https://www.netsolutions.com/insights/top-10-automation-testing-tools/>