Хмельницький національний університет
Факультет інформаційних технологій
Кафедра інженерії програмного забезпечення

ДИПЛОМНА РОБОТА

Методи оцінювання об'єктно-орієнтованих програмних систем на етапі проектування
Назва теми

Рівень вищої освіти Другий (магістерський)

Галузь знань 12 «Інформаційні технології»

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

Шифр ДРІПЗ. 170103.01.02.ПЗ

Виконав студент 2 курсу, група ІПЗм-21-1 _____ Е. С. Дзюрбан
Підпис Ініціали, прізвище

Керівник канд. техн. наук, доцент _____ О. М. Яшина
Науковий ступінь, звання Підпис Ініціали, прізвище

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

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

2 грудня 2022 р.

Хмельницький 2022

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет Інформаційних технологій
Кафедра Інженерії програмного забезпечення
Рівень вищої освіти Другий (магістерський)
Галузь знань 12 «Інформаційні технології»
Спеціальність 121 «Інженерія програмного забезпечення»
Освітня програма Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри ІПЗ
Л. П. Бедратюк
01.09.2022 р.

ЗАВДАННЯ
НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ)

Дзюрбану Едуарду Станіславовичу
Прізвище, ім'я, по батькові студента

1. Тема проєкту (роботи) Методи оцінювання об'єктно-орієнтованих програмних систем на етапі проєктування

Керівник проєкту (роботи) Яшина Оксана Миколаївна, канд. техн. наук, доцент
Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 01.07.2022 р. № 83
2. Строк подання студентом проєкту (роботи) на кафедру 01.12.2022 р.
3. Вихідні дані до проєкту (роботи) Матеріали переддипломної практики
4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)
1 Дослідження предметної області та постановка задачі
2 Концепції, моделі та методи вирішення задачі
3 Технології вирішення задачі
4 Реалізація та тестування програмного засобу
5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень)
Презентаційні матеріали (слайди)

6. Консультанти розділів дипломного проєкту (роботи)

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

7. Дата видачі завдання « 01 » вересня 2022 р.

КАЛЕНДАРНИЙ ПЛАН

Назва етапів (розділів) дипломного проєкту (роботи)	Строк виконання етапів проєкту	Примітка
1 Вивчення предметної області; формулювання мети та задач дослідження; визначення об'єкта та предмета дослідження; формування логістичної структури дипломної роботи	01.09-10.09.2021	
2 Робота над розділом 1 дипломної роботи – вивчення літературних та Інтернет-джерел; аналіз відомих моделей, методів та засобів за темою роботи; визначення методологічних підходів до вирішення задачі; висновки до розділу та постановка задач дослідження	11.09-25.09.2021	
3 Робота над розділом 2 дипломної роботи – розробка моделей, методів та алгоритмів вирішення задачі; висновки до розділу	26.09-10.10.2021	
4 Робота над науковими статтями	11.10-30.10.2021	
5 Робота над розділом 3 дипломної роботи – розробка інформаційної технології вирішення задачі (аналіз вимог до програмного засобу та його проектування, аналіз та вибір засобів реалізації програмного засобу тощо); висновки до розділу	11.10-26.10.2021	
6 Робота над розділом 4 дипломної роботи – програмна реалізація спроектованих рішень, результати експериментів та їх аналіз; дослідження ефективності запропонованих рішень; висновки до розділу	27.10-17.11.2021	
7 Попередній захист дипломної роботи	Листопад (згідно графіка)	
8 Узгодження постановки задачі, отриманих результатів та висновків; написання вступу, загальних висновків, оформлення джерел посилання та додатків; оформлення пояснювальної записки та графічних матеріалів згідно вимог чинних стандартів	18.11-30.11.2021	
9 Перевірка роботи на наявність плагіату; нормоконтроль; брошурування пояснювальної записки; підготовка супровідних документів	01.12-04.12.2021	
10 Підготовка до захисту дипломної роботи	з 01.12.2021 р	

Студент _____ Дзюрбан Е. С.
Підпис Ініціали, прізвище

Керівник проєкту (роботи) _____ Жишко О. М.
Підпис Ініціали, прізвище

РЕФЕРАТ

Тема дипломної роботи: Методи оцінювання об'єктно-орієнтованих програмних систем на етапі проектування.

Автор роботи: Дзюрбан Едуард Станіславович.

Керівник проекту: Яшина Оксана Миколаївна.

Пояснювальна записка: 85 с., 12 рис., 18 табл., 1 дод., 39 джерел.

ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОЕКТУВАННЯ, ОЦІНКА, МЕТРИКИ, СКЛАДНІСТЬ ПРОГРАМНОЇ СИСТЕМИ, ЯКІСТЬ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ.

Метою роботи є удосконалення методу та метрики оцінювання об'єктно-орієнтованих програмних систем на етапі проектування.

У дипломній роботі детально проаналізовано наявні методології оцінки об'єктно-орієнтованих програмних систем, досліджено існуючі методи та метрики для оцінювання. Виділено основні недоліки наявних рішень та запропоновано нову метрику для оцінки складності проектованих програмних систем. Також розглянуто методології для валідації об'єктно-орієнтовних метрик та вибрана найкраща для перевірки запропонованої метрики оцінки складності.

У ході виконання дипломної роботи було удосконалено метрику оцінки складності проектованої програмної системи під назвою СЧМА (складність через методи та атрибути). Запропонована метрика була провалідована за допомогою властивостей Вейюкера та верифікована за допомогою апробації її результатів на реальних прикладах з подальшою кореляцією отриманих значень та значень результатів уже відомих та перевірених метрик.

В результаті удосконалена метрика пройшла всі перевірки, продемонструвала хороші показники кореляції з уже перевіреними рішеннями, що доводить її дієвість та придатність до використання на реальних проектах.

02 12 2022
Дата

Дзюрбан
Підпис

ABSTRACT

Master's thesis: « Methods of evaluating object-oriented software systems at the design stage»

Author: Eduard Stanislavovych Dziurban.

Head of work: Oksana Mykolaivna Yashyna.

Master's thesis consists of: 85 p., 12 pic, 18 tab., 1 add., 39 srs.

OBJECT-ORIENTED DESIGN, EVALUATION, METRICS, SOFTWARE SYSTEM COMPLEXITY, SOFTWARE QUALITY.

The purpose of the work is to improve the method and metrics for evaluating object-oriented software systems at the design stage.

The thesis analyzed in detail the existing methodologies for evaluating object-oriented software systems, investigated the existing methods and metrics for evaluation. The main shortcomings of the existing solutions are highlighted and a new metric for assessing the complexity of the designed software systems is proposed. Methodologies for the validation of object-oriented metrics are also considered and the best one is selected for testing the proposed complexity assessment metric.

In the course of the thesis, a new metric for assessing the complexity of the designed software system was developed called CTMA (Complexity through to methods and attributes). The proposed metric was validated using Weyuker's properties and verified by testing its results on real examples with further correlation of the obtained values and the values of the results of already known and verified metrics.

As a result, the developed metric passed all tests, demonstrated good correlation indicators with already tested solutions, which proves its effectiveness and suitability for use on real projects.

02 12 2022
Дата

Дзюрбан
Підпис

ЗМІСТ

ВСТУП

Прогрес сучасного розвитку комп'ютерної техніки та програмного забезпечення надав можливість у сучасних реаліях створювати найскладніші програмно-технічні комплекси, що дозволяють автоматизувати процеси у різних галузях життєдіяльності людини. Завдяки бурхливому розвитку якісно змінився і сам процес програмування. Нові засоби написання програм, такі як об'єктно-орієнтоване програмування, різні типи систем управління базами даних, мова універсального моделювання і т. д. дозволяють за короткий термін відносно невеликим колективам програмістів виконувати розробку програмного забезпечення для вирішення найскладніших завдань.

Дослідження показали, що все існуюче в даний час розробки програмне забезпечення умовно можна розбити на три наступні категорії.

Інформаційні системи та інші додатки, розроблені для використання компанією у власних цілях. Ця категорія є основою індустрії інформаційних систем (інформаційних технологій) або IS/IT.

Програмне забезпечення, яке розробляється та продається як комерційний продукт. Компанії-розробники такого ПЗ зазвичай називаються незалежними постачальниками програмного забезпечення.

Програмне забезпечення комп'ютерів, що вбудовується в інші прилади, машини чи складні системи. Програмне забезпечення такого типу називають додатками для вбудованих систем або вбудованими програмами.

Разом із цим компанії-розробники програмного забезпечення часто стикаються зі збоями у програмних системах або виявляють прогалини в архітектурі своїх додатків уже на етапі розробки, або ще гірше – на етапі експлуатації. Через що вартість обслуговування такого програмного забезпечення зростає, і компанія може понести великі збитки.

Попит на ефективне програмне забезпечення зростає з кожним днем і прийом об’єктно-орієнтованого проектування програмних систем здатен

задовольняти цей попит, оскільки це чи не найпотужніший механізм розробки ефективних програмних системи. Це може не тільки допомогти зменшити вартість, але також допомагає в розробці високоякісного програмного забезпечення системи. Розробникам програмного забезпечення потрібні відповідні показники розробки ефективної програмної системи. Ця практика спрямована на дослідження методів оцінювання об'єктно-орієнтованої системи програмного забезпечення за допомогою аналізу впливу на програмне забезпечення на основі відстеження вимог до змін функціональних вимог з використанням функціональних вимог.

Не зважаючи на те, що є багато переваг об’єктно-орієнтованого підходу, та те що цей підхід є найбільш розповсюдженим зараз, та й буде у майбутньому, він буде справді визнаним, доведеним і практичним лише тоді, коли аспекти управління процесом розробки програмного забезпечення з використанням цієї методології будуть ретельно розглянуті. Тут показники програмного забезпечення відіграють важливу роль, дозволяючи краще планувати, оцінювати покращення, зменшувати непередбачуваність, раннє виявлення потенційних проблем і оцінювати продуктивність. У цій роботі пропонується набір показників, які найкраще підходять для оцінки використання основних концепцій об’єктно-орієнтованої парадигми, таких як успадкування, інкапсуляція, поліморфізм, а також великий акцент на повторному використанні коду, які однозначно відповідальні за збільшення якості програмного забезпечення та продуктивності розробки. Ці показники мають на меті допомогти встановити порівняння практик та рекомендацій щодо дизайну, щоб з часом перетворити найкращі в певний єдиний стандарт. Також буде наведено деякі бажані властивості для цього набору показників та також передбачено наступні напрямки досліджень.

Відсутність чітких критеріїв оцінювання часто стає причиною критики об’єктно-орієнтованого підходу до розробки програмного забезпечення. Всі концепції і прийоми об’єктно-орієнтованого підходу до програмування мають на меті полегшити розробку та підтримку програмного забезпечення, пришвидшити процеси вродження нового функціоналу та розширюваність. Отож при виконанні

практики буде досліджено методи оцінки об’єктно-орієнтованих програмних систем ще на етапі проектування.

Одна з проблем аналізу систем – оцінювання їх складності. Складність системи є якісною характеристикою. Зменшення складності програмних систем дає змогу знизити трудомісткість проектування, розробки, тестування та супроводження, забезпечує простоту і надійність виробленої програмної системи. У даний час питанням управління якістю програмних систем приділяється підвищена увага, пов’язано це з ростом застосувань програмних систем у різних сферах діяльності людини. Управляння якістю включає в себе планування якості, забезпечення якості і контроль якості. Таким чином, однією зі складових розробки ПС із запланованим рівнем якості є контроль, який базується на застосуванні метрик якості для вимірювання основних показників в процесі розробки ПС.

Об’єкт дослідження – процес оцінювання об’єктно-орієнтованих програмних систем на етапі проектування.

Предмет дослідження – методи оцінювання об’єктно-орієнтованих програмних систем на етапі проектування.

Мета дослідження – удосконалення методу та метрики оцінювання об’єктно-орієнтованих програмних систем на етапі проектування.

Відповідно до мети, предмету та об’єкту дослідження можна визначити такі завдання:

– здійснити аналіз предметної області;

– дослідити існуючі методи вирішення цих проблем;

– проаналізувати використовувані метрики в наявних методах із визначенням їх переваг та недоліків;

– на основі аналізу удосконалити метод та метрику оцінки складності програмної системи на етапі проектування;

– провести тестування та апробацію отриманої метрики, дослідити її ефективність на реальних прикладах;

– провести аналіз отриманих результатів та сформулювати рекомендації для подальшого удосконалення розробленого рішення.

Наукова новизна:

1.Удосконалено метод оцінювання складності об’єктно-орієнтованих програмних систем на етапі проектування.

2. Удосконалено метрику оцінювання складності об’єктно-орієнтованих програмних систем на етапі проектування із встановленням взаємозв’язків між результатами розробленого методу та факторами якості програмної системи.

Практичне значення даного дослідження полягає у простішому методі оцінки складності об’єктно-орієнтованих програмних систем на етапі проектування, результати роботи якого добре корелюються із уже відомими, проте більш складними методами. Також, відповідно до отриманих результатів, за запропонованими таблицями відповідностей можна визначити фактори якості проектованої програмної системи.

За темою кваліфікаційної роботи опублікована наукова стаття у фаховому виданні «Метод оцінки об’єктно-орієнтованих програмних систем на основі аналізу зміни вимог до програмної системи».

1 ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Проектування об’єктно-орієнтованих програмних систем

Основою будь-якої програмної системи є її проект. Це скелет, на якому буде нарощуватися руховий апарат (код). А поганий скелет не дозволить гармонійно рости основній масі програми та буде перешкоджати як під час створення, так і під час експлуатації програмної системи. Оскільки аналіз вимог найчастіше є неповним, ми повинні вміти легко створювати проекти програмного забезпечення зрозумілими, відкритими до змін, перевіреними та, бажано, стабільними (з можливістю додавання модифікацій) [1]. Об'єктно-орієнтована парадигма включає набір механізмів такі як наслідування, інкапсуляція, поліморфізм і обмін повідомленнями, які, як вважають її творці, дозволяють проєктувати системи так, де всі ці аспекти застосовуються. Однак архітектор повинен вміти використовувати ці механізми «зручним» способом.

Задовго до того, як мови ООП набули широкого розповсюдження, можна було створювати програмне забезпечення з допомогою попередників об’єктно-орієнтованих мов програмування, які в собі мали частини концепцій сучасного ООП [2]. І навпаки, просто використовуючи об’єктно-орієнтований підхід, який підтримує ці механізми, ми автоматично надаємо перевагу збільшенню продуктивності системи та програмістів, проте це покладає велику відповідальність на плечі архітектора системи, оскільки потрібно проєктувати систему без помилок. Проектування, будучи «творчою» діяльністю, де часто доступні кілька альтернатив, вибрати один найкращий буває досить складно. Саме тому було б добре мати набір певних евристичних правил для того, щоб вибрати найкращий для того чи іншого випадку. Для цього використовуються метрики проєктування [3].

У сучасному сценарії розробки програмного забезпечення об’єктно-орієнтований підхід є провідним. Цей підхід підвищує продуктивність

програмного забезпечення, дає багаторазове використання коду та гнучкість програмних систем. Об'єктно-орієнтовані системи набувають популярності як ефективні програмні системи день у день, тому що об'єктно-орієнтовані методи зменшують розмір системи та кількості логічних конструкцій у ній. Об'єктно-орієнтований програмне забезпечення зазвичай містить велику кількість атрибутів, які містять в собі повний опис програмних компонентів та елементів. Між ними будується взаємодія через обмін повідомлень, в результаті чого компоненти мають слабку зв’язність та залежність одне від одного. Різні оціночні концепції, як складність, зручність використання, багаторазове використання, можливість тестування, зрозумілість тощо використовуються для підвищення якості програмного забезпечення системи, яка також дуже пов’язані з об’єктно-орієнтованими підходами [4]. Ці всі концепції активно використовуються для покращення об’єктно-орієнтованих систем.

Показники програмного забезпечення стали важливими в деяких дисциплінах програмної інженерії, оскільки вони використовуються для вимірювання якості програмного забезпечення та оцінки вартості та затрат на розробку програмного забезпечення. Зазвичай показники використовуються для визначення якості програмного забезпечення на ранній стадії циклу розробки програмного забезпечення (SDLC) для моніторингу впливу змін на вартість і покращення системи програмного забезпечення [5].

Оскільки об'єктно-орієнтовані метрики вимагають розуміння об'єктно-орієнтованих концепцій і немає єдиної метрики, яка показує всі особливості об'єктно-орієнтованої програмної системи, тому у цій дипломній роботі будуть розглянуті різні об’єктно-орієнтовані метрики.

Необхідно встановити деякі основні стандарти і керівні принципи, яких повинен дотримуватися розробник програми щоб досягнути очікуваних вигод і переваг об'єктно-орієнтованої технології.

Методи проектування передбачають набір прийомів для аналізу, декомпозиції та модульної системи архітектури програмного забезпечення. Існує широке застосування об'єктно-орієнтованого підходу в сучасному сценарії

розробки програмного забезпечення, оскільки це сприяє кращій архітектурі продукту, а програмна являє собою систему як сукупність взаємодіючих об'єктів. Компоненти різного об'єктно-орієнтованого програмного забезпечення наведено в таблиці 1.1.

Таблиця 1.1 – компоненти об’єктно-орієнтованої системи

Об’єкти	Будують	Класи
Об’єкти	Мають	Атрибути
Об’єкти	Наслідують	Атрибути
Об’єкти	Мають	Методи
Об’єкти	Наслідують	Методи
Об’єкти	Надсилають	Повідомлення
Об’єкти	Отримують	Повідомлення
Повідомлення	Це	Дані
Повідомлення	Це	Відношення

Підтримка програмного забезпечення є (і, безперечно, залишатиметься) основним джерелом трати ресурсів протягом усього життєвого циклу програмного забезпечення. Придатність програмної системи до підтримки можна оцінити за допомогою кількох характеристик якості, таких як можливість аналізу, змінність, стабільність і придатність до тестування. Усе це описано у стандарті ISO9126 [6]. Об’єктно-орієнтований підхід до проектування полегшує програмування програмної системи, а також її удосконалення та покращення. Проте це правило працює лише тоді, коли проектована система правильно та без критичних помилок.

Розробнику хотілося би вірити і надіятися що на системному рівні є механізми для забезпечення основних принципів ООП, а саме інкапсуляції, успадкування, поліморфізму або обміну повідомлень між класами. Усе це має відбуватися між об’єктами, які тісно взаємопов'язані з описаними вище якісними характеристиками. Знаходячи ці закономірності шляхом ретельної

експериментальної перевірки, ми не лише хочемо показати хороші сторони одного прийому над іншим, а і хочемо у порівняльному варіанті привести аргументи чому саме цей прийом буде кращим і актуальнішим в даному випадку ніж будь який інший [7]. Це особливо корисно для малодосвідчених архітекторів програмних систем, які ще не мають вироблених практикою звичок і кола розумінь коли і як краще застосовувати той чи інший прийом при проектуванні.

Також для оцінки теоретичного часу необхідного для реалізації програмної системи потрібно спочатку визначити її розмір. Для цього потрібно оцінити відсоток коду який потрібно буде розробляти з нуля, а також відсоток який або буде використаний в якості бібліотек, або як частини бібліотек, але адаптовані під наші потреби. Тому що вибір і адаптація компонентів під наші потреби може вимагати затрат великої кількості зусиль. Тому повна модель оцінки об’єктно-орієнтованих проектів є однією з складових мети даних досліджень [8].

Об'єктно-орієнтоване проектування та розробка є відомим способом у сучасному середовищі розробки програмного забезпечення. Це підвищує продуктивність програмного забезпечення, можливість повторного використання та гнучкість програмних систем. Об’єктно-орієнтовані системи стають популярними як ефективні програмні системи, оскільки вони зменшують розмір системи та кількість логічних конструкцій.

Об’єктно-орієнтоване програмне забезпечення зазвичай має величезну кількість атрибутів, і ці атрибути забезпечують більш вичерпний опис внутрішньої природи та структури програмного забезпечення. Ці програмні системи складаються з взаємодіючих об’єктів, які залишаються у своєму власному локальному стані та виконують свою власну інформацію.

Якість є одним з найважливіших аспектів, які приваблюють клієнтів. Це стосується не лише. Багато членів команди, таких як інженери, тестувальники та дизайнери, контролюють якість продукту. У процесі розробки висока якість повинна бути ціллю номер один. Показники якості коду – найкращі інструменти для перевірки готовності продукту до ринку. Якщо він не має помилок, не повільний і не дає збоїв, настав час його випускати.

Такі поняття, як складність, зручність використання, багаторазове використання, можливість тестування, зрозумілість тощо, використовуються для покращення якості системи програмного забезпечення, які також мають відношення до об’єктно-орієнтованих функцій і можуть бути використані для підвищення ефективності об’єктно-орієнтованих систем.

Аспекти якості програмного забезпечення:

Надійність. Аналізуючи кількість дефектів, ви побачите, наскільки добре буде працювати програмне забезпечення і як довго система працюватиме безперебійно без збоїв. Якщо ви хочете мати надійну кодову базу, низька кількість дефектів особливо важлива.

Придатності до підтримки. Чи важко підтримувати програмне забезпечення? Давайте з’ясуємо це за допомогою показників структури, розміру, складності та узгодженості, які аналізують кодову базу. Також важливо перевірити придатності до тестування і зрозумілість.

Придатність до тестування. Наскільки добре продукт підтримує тестування? Це залежить від того, наскільки добре ви контролюєте, автоматизуєте, ізолюєте та спостерігаєте за процесом тестування.

Портативність. Він показує, чи можна використовувати програмне забезпечення в різних середовищах. Немає інструментів, які можуть продемонструвати портативність проекту, але є деякі засоби, щоб це зробити. Оптимальним варіантом є тестування коду на різних платформах, не чекаючи закінчення розробки.

Багаторазове використання. Щоб перевірити, чи можна знову використовувати такі активи, як код, використовуйте показники, які вимірюють кількість взаємозалежностей. Можливість повторного використання залежить від наявності модульності або слабкого зв’язку.

Отже, як керувати якістю програмного забезпечення? Це залежить не тільки від ефективності метрик, а й від розробників. Тому при розробці програмних систем слід приділити увагу таким аспектам:

Стандарт кодування. Одним із найкращих способів забезпечити високоякісний продукт є використання стандарту кодування. Це гарантує, що кожен член команди виконує роботу правильно, і забезпечує послідовність і читабельність коду. Наявність стандарту полегшує використання проекту та покращує якість програмного забезпечення.

Аналіз коду. Досвідчені фахівці знають, що проблеми легше попередити, ніж боротися з ними після звільнення. Якість має бути пріоритетом номер один протягом усього процесу розробки. Чим раніше ви виявите помилки, тим швидше, простіше і дешевше їх виправити.

Використання новітніх технологій. Краще покладатися не тільки на розробників, але й використовувати перераховані вище метрики. Ручна перевірка коду все ще корисна, але не настільки ефективна. Нехай показники якості розробки програмного забезпечення будуть автоматизовані.

Рефакторинг. Якщо мова йде про вдосконалення вже існуючого і застарілого продукту, використовуйте рефакторинг. Це допомагає очистити кодову базу та значно полегшити використання. Найкраще робити це поступово.

1.2 Методи оцінювання об’єктно-орієнтованих програмних систем

Оцінки архітектури можна виконати в одному або кількох етапах процесу розробки програмного забезпечення. Їх можна використовувати для порівняння та визначення сильних та слабких сторін в різних архітектурних альтернатив на ранніх стадіях проектування. Вони також можуть бути використані для оцінки існуючих систем з перспективою майбутнього обслуговування або вдосконалення системи, а також для виявлення архітектурних проблем та нюансів.

Розвиток програмного забезпечення є динамічним і постійно зазнає серйозних змін. Сьогодні для розробки програмного забезпечення доступна величезна кількість інструментів і методологій, а розробка програмного

забезпечення відноситься до всіх видів діяльності, пов’язаних із створенням рішення інформаційної системи. Діяльність із розробки систем складається з системного аналізу, моделювання, проектування, впровадження, тестування та обслуговування, а також стан метрик програмного забезпечення в розробці програмного забезпечення протягом останнього десятиліття є обнадійливим, і багато дослідників займаються метриками програмного забезпечення. Застосовуються показники програмного забезпечення, і з критикою отримано хороші результати. Використання програмних метрик довело ефективність процесів і продукту. У розробці програмного забезпечення нещодавно дослідники метрик програмного забезпечення запровадили нові метрики та підтвердили метрики програмного забезпечення за допомогою емпіричних і теоретичних методів, а метрики програмного забезпечення використовувалися під час прийняття рішень, а також у різних процесах, і все більше дослідників беруть участь у емпіричних дослідженнях. Видатні дослідники скеровують фахівців з програмного забезпечення для оцінки ефективності програмного продукту за допомогою програмних показників. Існує багато типів метрик програмного забезпечення, починаючи від традиційних метрик і закінчуючи новітньою галуззю комп’ютерних наук, тобто пов’язаних з Інтернетом метрик у розробці програмного забезпечення.

Щоб запропонувати об’єктно-орієнтовану метрику в розробці програмного забезпечення, глибоке розуміння попередньої об’єктно-орієнтованої метрики має важливе значення для вимірювання програмного забезпечення. Краще розуміння існуючих метрик призведе до чіткого уявлення та розробки концепцій для вирішення проблем неоднозначності в об’єктно-орієнтованих метриках.

Методи оцінки архітектури програмного забезпечення можна розділити на три основні категорії, тобто на основі досвіду, на основі моделювання, на основі математичного моделювання.

Оцінки на основі досвіду базуються на попередньому досвід і знаннях предметної області розробників або архітекторів. Люди, які зіткнулися з вимогами

та доменом який раніше може базуватися на попередньому досвід показує, чи буде архітектура програмного забезпечення достатньо хорошою [9].

Оцінки на основі моделювання покладаються на високий рівень впровадження деяких або всіх компонентів у архітектурі програмного забезпечення. Симуляцію можна використовувати для оцінки вимог до якості, таких як продуктивність і правильність архітектури. Симуляція також може бути у поєднанні з прототипуванням, таким чином прототипи архітектури може виконуватися в передбаченому контексті завершеної системи.

Математичне моделювання використовує математичні докази і методи оцінки експлуатаційних вимог до якості, таких як продуктивність і надійність компонентів в архітектурі [10]. Математичне моделювання можна поєднувати з моделювання для більш точної оцінки ефективності компонентів в системі.

Оцінка архітектури на основі сценарію намагається оцінити певний атрибут якості шляхом створення профілю сценарію, який вимагає дуже конкретного опису вимог до якості. Сценарії з профілю потім використовуються для проходження архітектури програмного забезпечення та як наслідок, має бути задокументовано всі сценарії. Існує кілька методів оцінки на основі сценаріїв, наприклад, метод аналізу архітектури програмного забезпечення (SAAM) [11], метод компромісного аналізу архітектури (ATAM) [12] та аналіз модифікованості рівня архітектури (ALMA) [13].

Метод LSP був розроблений в сімдесятих роках. А досить широкий опис методу можна знайти в статтях [14,15]. Документи, які досліджують цей метод, включають [16] і [17]. Метод LSP вперше був використаний для оцінки програмного забезпечення та вибору систем баз даних. Інші останні програми включають оцінку віконних середовищ, веб-браузерів, пошукових систем і різних веб-сайтів. Метод LSP включає і суттєво розширює та узагальнює модель оцінки програмного забезпечення, викладена в стандарті ISO 9126 [6]. У цьому дослідженні ми представимо підхід LSP до оцінювання програмного забезпечення через його впровадження в ISEE та його застосування в оцінці пошукових систем.

Для кожної конкретної проблеми оцінки виконано декомпозицію глобальної якості ПЗ використовуючи вибрані групи атрибутів, які зазвичай є підмножиною із зображених на рисунку 1.1. Процес декомпозиції використовується для розбиття складних критеріїв на простіші компоненти. Наприкінці цього процесу ми генеруємо елементарні атрибути якості, які не можуть бути розкладені далі. В контексті методу LSP ці атрибути називаються змінними продуктивності.

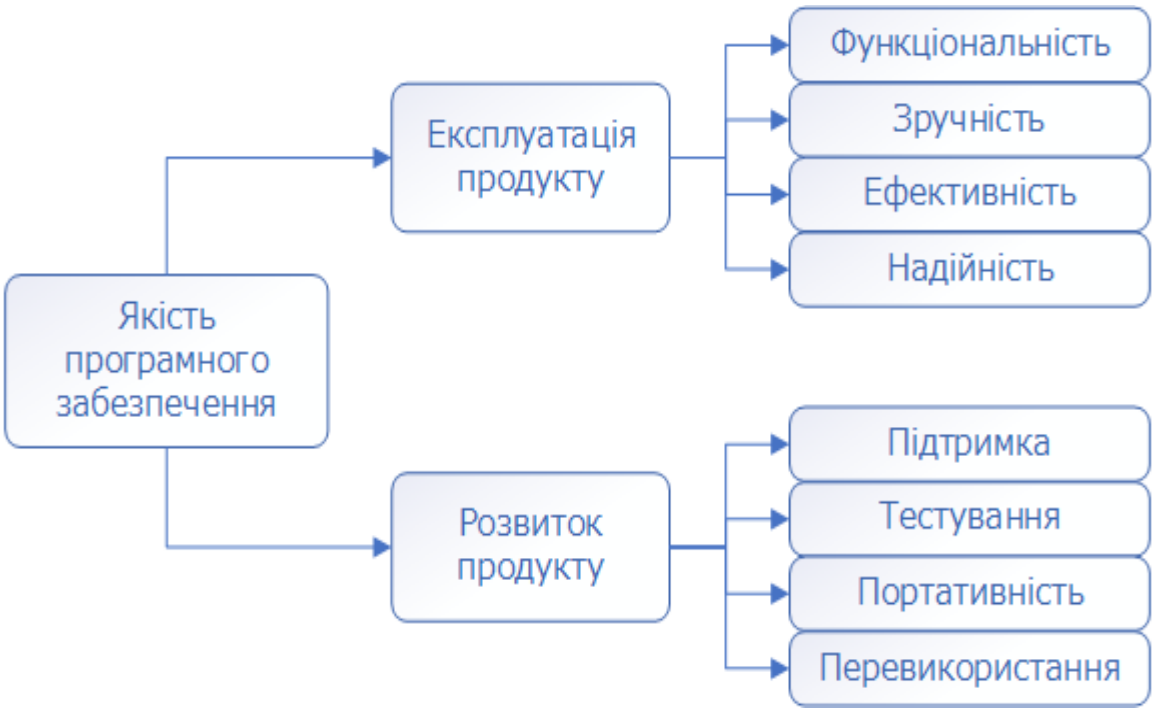


Рисунок 1.1 – Декомпозиція глобальної якості програмного забезпечення

Наприклад, критерій ефективність зазвичай складається з продуктивності та ресурсу споживання. Продуктивність можна виразити за допомогою різних індикатори, одним з яких завжди буде час відгуку. Час відгуку - елементарний атрибут, який не може далі розкладається. Таким чином, час відгуку є одним із змінних ефективності, які можна виміряти та оцінити.

Для кожної змінної продуктивності ми визначаємо елементарний критерій, який використовується для обчислення відповідного рівень відповідності вимогам користувача. Цей рівень відповідності називається перевагою та її значення варіюється між 0 (немає відповідності) і 1 або 100% (повна відповідність).

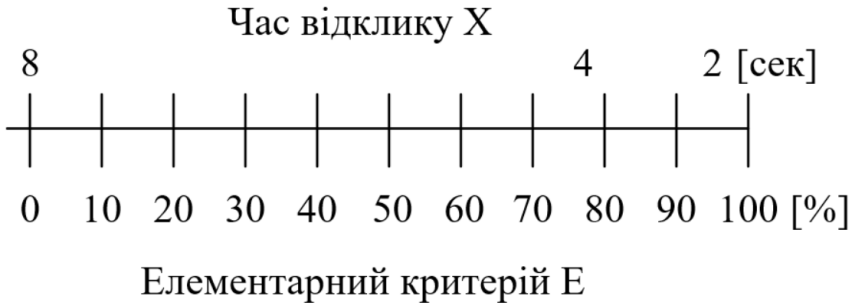


Рисунок 1.2 – Приклад елементарного критерію

Типовий елементарний критерій оцінки часу відповіді показано на рисунку 1.2. У цьому прикладі (що типово для програм електронної комерції) усі відповіді, час яких менше або дорівнює 2 секундам, цілком задовольняють користувача. Час відгуку 4 секунди прийнятний у 80% користувачів (задовольняє 80% вимог), а також час відповіді більше або дорівнює 8 секундам неприйнятний.

Для всіх інших значень часу відгуку ми використовуємо лінійну інтерполяція: наприклад час відгуку 3 секунди задовольняє вимоги на 90% і час відгуку 5 секунд задовольняє вимоги на 60%. Елементарні критерії породжують набір з n елементарних переваги (для складних систем n може бути великим, до кілька сотень). З елементарних критеріїв можемо обчислити глобальні переваги за допомогою конкретних. Структура агрегації переваг показана на рисунку 2. Структура агрегації переваг включає в себе набір логічних операторів. Ці оператори можуть виражати різноманітну логіку та зв'язок між окремими вимогами, такими як одночасність, замінність та інші. Кінцевим результатом процесу агрегації є глобальні переваги, які кількісно визначають глобальне задоволення всіх вимог користувача.

Метод аналізу архітектури програмного забезпечення (SAAM) [11] – це метод оцінки архітектури програмного забезпечення на основі сценаріїв, призначених для оцінки однієї архітектури або створення декількох архітектур, порівнювані за допомогою метрик, таких як зчеплення між компонентами архітектури. SAAM був спочатку зосереджено на порівнянні модифікованості

архітектури різного програмного забезпечення в домені організації. З тих пір перетворився на структурований метод для оцінки програмного забезпечення на основі сценаріїв оцінки архітектури.

Метод складається з п'яти кроків. Починається з документації архітектури таким чином, щоб усі частини оцінки можна було зрозуміти. Потім розробляються сценарії які описують передбачуване використання системи. Сценарії повинні представляти всі зацікавлені сторони, які використовуватимуть систему. Потім оцінюються сценарії та набір сценаріїв, які представляє вибраний аспект, який ми хочемо оцінити. Тоді взаємодіючі сценарії визначаються як міра модульність архітектури.

Метод компромісного аналізу архітектури (ATAM) [12] – це сценарний метод оцінки архітектури програмного забезпечення. Метою методу є оцінка дизайну на рівні архітектури яка враховує численні атрибути якості та отримує уявлення про те, чи відповідатиме реалізація архітектури її вимогам. ATAM будується на SAAM і розширює його для обробки компромісів між кількома атрибутами якості. Архітектурна оцінка виконується в шість етапів. Перший – збирати критерії, які реалізують вимоги до системи (як функціональні, так і якісні вимоги). Другий крок – зібрати інформацію щодо обмежень і середовище системи. Ця інформація використовується для підтвердження того, що сценарії актуальні для системи. Третій крок використовується щоб описати архітектуру, використовуючи представлення, які мають відношення до атрибутів якості, які були визначені на першому кроці. Крок четвертий – це аналіз архітектури щодо атрибутів якості. Атрибути якості оцінюються по одному. Крок п'ятий – це визначити чутливі точки в архітектурі, тобто виявити ті точки, на які впливають коливання атрибутів якості. Шостий і останній крок полягає в ідентифікації та оцінці точки компромісу, тобто точки варіації, які є спільними для двох або більше якісних атрибутів. ATAM було використано та перевірено в кількох дослідженнях [18, 19].

Аналіз модифікованості на рівні архітектури (ALMA) [11, 12] є а сценарний метод оцінки архітектури програмного забезпечення з такими характеристиками

як: фокус на модифікованості, розподіл на кілька цілей аналізу, важливість чітких припущень та надати повторювані методи для виконання наступних кроків. Метою ALMA є надання структурованого підходу для оцінки трьох аспектів легкості підтримки програмних архітектур, тобто прогнозоване технічне обслуговування, оцінка ризиків та архітектура порівняння програмного забезпечення. ALMA - це метод оцінки, який слідує SAAM у своїй організації.

Інженерія продуктивності програмного забезпечення (SPE) [20, 21] є загальним метод вбудовування продуктивності в програмну систему. Концепція полягає в тому, що продуктивність має бути врахована протягом усього процесу розробки, а не тільки оцінюватися або оптимізуватися, коли система вже розроблена. SPE покладається на дві різні моделі програмної системи, тобто модель виконання програмного забезпечення та модель виконання системи. Модель виконання програмного забезпечення моделює програмні компоненти, їх взаємодію та потік виконання. Крім того, ключові вимоги до ресурсів для кожного компонента також можуть бути включені, наприклад, час виконання, вимоги до пам’яті та кількість операції введення/виведення.

Модель виконання програмного забезпечення передбачає продуктивність без конкуренції за апаратні ресурси одиниці програмної системи. Приклади апаратних ресурсів, які можна моделювати, це процесори, пристрої введення/виведення та пам'ять. Далі – очікування часу і конкуренція за ресурси також моделюються. Модель виконання програмного забезпечення генерує вхідні параметри для моделі виконання системи. Модель виконання системи може бути розв’язана за допомогою математичних методів або моделювання. Метод може бути використаний для оцінки різних показників, наприклад, час відгуку, пропускна здатність, використання ресурсів, та виявлення вузьких місць. Методи – перш за все націлені на оцінку продуктивності. Проте автори стверджують, що їх метод можна використовувати для оцінки інших атрибутів якості також [22].

SAM [23] – це формальна систематична методологія для побудови специфікації та аналіз архітектури програмного забезпечення. SAM в основному

призначений для аналізу правильності та продуктивності системи. SAM має дві головні цілі.

Перша мета – це вміння точно визначити архітектуру програмного забезпечення та її властивості, а також потім виконати їх формальний аналіз за допомогою формальних методів. Крім того, SAM також підтримує архітектуру специфікації з використанням часових мереж Петрі та часової логіки для виконуваного програмного забезпечення.

Друга мета полягає в тому, щоб полегшити масштабовану архітектуру програмного забезпечення специфікації та аналізу з використанням ієрархічної архітектури розкладання.

Метод архітектурних стилів на основі атрибутів (ABAS) [24] базується на понятті архітектурних стилів [25, 26], і його розширюваності по асоціативних основах проектування з архітектурним стилем. Метод можна використовувати для оцінки різних атрибутів якості, наприклад, продуктивності або легкості підтримки, і таким чином не націлений на певний набір атрибутів якості. Основою аналізу для архітектурного стиль може бути якісною або кількісною, а також є аналіз на основі моделей для конкретних атрибутів якості. Таким чином, ABAS дозволяє аналізувати різні аспекти якості архітектури програмного забезпечення. Спосіб загальний і кілька атрибутів якості можна аналізувати одночасно, тому що моделі якості надаються для відповідної якості атрибутів. Однією з сильних сторін ABAS є те, що цей метод також можна використовувати для архітектурного проектування. Крім того, ABAS був використаний як частина оцінок за допомогою ATAM [12].

Автори статті описують у своїй праці [27] метод під назвою Lindvall, який демонструє приклад редизайну або ж перевпровадження більш-менш розробленої програмної системи вдома. Основною метою була оцінка легкості підтримки нової система порівняно з попередньою версією. У статті описано процес на основі емпіричного досвіду оцінки архітектури програмного забезпечення.

У документі визначено та використовується певна кількість архітектурних показників, які використовуються для оцінки та порівняння архітектури. Основні

етапи процесу: вибрати перспективу для оцінки, визначити/вибрати показники, збирати показники та оцінювати/порівняти архітектури. У цьому дослідженні задачі оцінки були оцінити легкість підтримки, а показниками були структура, розмір і зв'язність системи. Оцінки проводилися на запізнілому етапі розробки програмної системи, тобто коли модулі системи вже впроваджені. Архітектура програмного забезпечення була перероблена з використанням вихідного коду та застосуванням до нього різного роду метрик.

Незважаючи на багатообіцяючу кількість первинних досліджень, тобто 38, виявилося, що лише 10 методів оцінок архітектури програмного забезпечення було можливо ідентифікувати та виявити, що їх результати стосуються одного або кількох показників продуктивності, легкості підтримки, тестування або будь яких інших атрибутів якості програмної системи.

Тільки один (ATAM) із решти методів оцінки забезпечує підтримку компромісного аналізу між атрибутами якості. Немає конкретних методів, які оцінюють придатність системи до тестів та міграції явно. Ці атрибути якості можна розглянути за допомогою будь-якого з трьох методів оцінки, які є більш загальними за своєю природою, тобто можуть охоплювати більше довільно вибраних атрибути якості, ATAM [12], SAAM [11], або метод Lindvall [27].

Багато з методів використовувалися кілька разів авторами при дослідженні. Багаторазове використання методу свідчить про велику валідність методу. Однак було лише два методи, які використовувалися у своїх дослідженнях іншими авторами. Це свідчить про зрілість методу та його більшу вагу. Цими двома методами є SAAM і ATAM.

1.3 Постановка задачі

Методи, які оцінюють декілька атрибутів якості та компромісні в аналізі особливо цікаві. Було виявлено, що багато методів оцінюють лише однин атрибут якості, і дуже небагато методів можуть оцінювати кілька атрибутів якості одночасно в одному і тому ж алгоритмі.

Дослідження, проведені в ході аналізу предметної області, наводять до висновків про те що найкращими методами для оцінки об’єктно-орієнтованих програмних систем це ті, які орієнтуються на оцінку кількох атрибутів якості.

Об’єктом дослідження є методи оцінки об’єктно-орієнтованих програмних систем етапі проектування.

Метою дослідження є аналіз та удосконалення існуючих методів оцінки об’єктно-орієнтованих програмних систем на етапі проектування шляхом розробки нової метрики оцінки складності проектованої програмної системи.

Для досягнення поставленої мети дослідження потрібно:

– дослідити проблему оцінки об’єктно-орієнтованих програмних систем на етапі проектування

– дослідити існуючі методи вирішення цих проблем;

– проаналізувати використовувані метрики в наявних методах;

– визначити переваги та недоліки уже існуючих метрик;

– на основі попереднього аналізу розробити власну метрику оцінки складності програмної системи на етапі проектування;

– провести тестування та апробацію отриманої метрики, дослідити її ефективність та надійність.

1.4 Висновки

Отже, у першому розділі було проведено огляд та аналіз методів та метрик оцінки для атрибутів якості архітектури програмного забезпечення. Було зосереджено увагу на метриках оцінки одного або кількох атрибутів якості, таких як ефективність, складність, придатність до тестування та підтримки.

Архітектура програмної системи була визначена як важливий аспект у розробці програмного забезпечення, оскільки архітектура впливає на якість самої програмної системи.

Якість архітектури програмного забезпечення має п’ять атрибутів. Це ефективність, складність, зрозумілість, придатність до багаторазового

використання та придатність до тестування та підтримки. Вони охоплюють ключові концепції об'єктно-орієнтованого проектування: методи, класи та наслідування. Для кожного показника можуть бути прийняті порогові значення залежно від застосовних атрибутів якості та цілей застосування.

Хороша архітектура програмного забезпечення підвищує ймовірність того, що система буде виконувати свою задачу якісно, а також буде відповідати усім вимогам. Отже, методи та метрики оцінки якості архітектури програмного забезпечення є важливими.

2 МЕТРИКИ ОЦІНЮВАННЯ ОБ’ЄКТНО-ОРІЄНТОВАНИХ ПРОГРАМНИХ СИСТЕМ

2.1 Класифікація метрик оцінки об’єктно-орієнтованих програмних систем

Розробники програмного забезпечення намагаються оцінити програмне забезпечення, щоб отримати кількісне розуміння його властивостей і якості з моменту його створення. Стандарт IEEE визначає розробку програмного забезпечення як застосування «систематичного, дисциплінованого, кількісно вимірюваного підходу до розробки, експлуатації та підтримки програмного забезпечення».

З появою об'єктно-орієнтованого підходу були введені спеціальні заходи для оцінки якості програмних систем. Обґрунтування метрик полягає в тому, що хороший дизайн повинен зберігати низьку складність, і цього можна досягти шляхом мінімізації зв’язків і підвищення зв’язності.

Першою спробою в цьому напрямку був набір метрик Чідамбера та Кемерера, і вони стали найпопулярнішим набором метрик, а процес визначення нових показників все ще є жвавою областю досліджень. Проте теоретичних причин, які підтримують прийняття конкретних показників, недостатньо. Інженери-програмісти повинні мати емпіричні докази того, що ці показники дійсно пов’язані з якістю програмного забезпечення.

Загалом, поняття якість програмного забезпечення є досить абстрактним та недосяжним поняттям, програмне забезпечення є нематеріальною сутністю, яку неможливо фізично виміряти традиційними способами. Загалом якість програмного забезпечення має багато різних значень, вона пов’язана з практиками, які призводять до програмних продуктів, які є точними, ефективними, доставленими вчасно та в рамках закладеного бюджету.

Існує два основні типи метрик оцінки програмних систем. Це метрики продукту та метрики процесу (зображені на рисунку 2.1)

Метрики процесу відомі як метрики керування та використовуються для вимірювання властивостей процесу, який використовується для отримання програмного забезпечення. Показники процесу включають показники вартості, показники затрат, показники просування та показники повторного використання продукту. Показники процесу допомагають передбачити розмір кінцевої системи та визначити, чи виконується проект згідно з графіком.

Метрики продукту також відомі як метрики якості та використовуються для вимірювання властивостей програмного забезпечення. Метрики продукту включають метрики ненадійності продукту, метрики функціональності, метрики продуктивності, метрики зручності використання, метрики вартості, метрики розміру, метрики складності та метрики стилю. метрики продуктів допомагають покращувати якість різних системних компонентів і порівнювати існуючі системи за певними показниками та атрибутами.

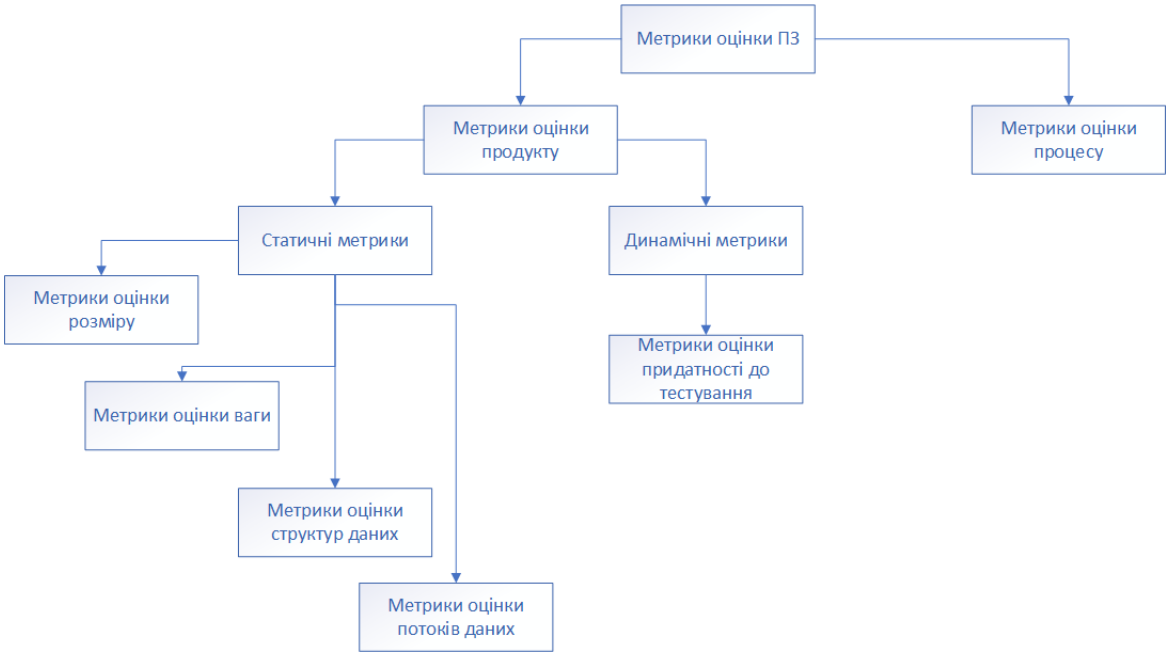


Рисунок 2.1 – Класифікація метрик оцінки програмних систем

Об’єктно-орієнтовані метрики відіграють ключову роль у розробці безвідмовного програмного продукту. Об’єктно-орієнтовані показники в основному розроблені для об’єктно-орієнтованих систем, які базуються на

принципах локалізації, абстракції, інкапсуляції, приховування інформації та успадкування. Береться схема компанії, що займається програмним забезпеченням, його розробкою та підтримкою, і створюються різні проекти, які демонструють різний рівень успадкування.

Швидкий розвиток технологій програмного забезпечення та зростання вимог ринку призводять до створення дедалі складнішого програмного забезпечення. Через обмеження вартості розробки та часу виходу на ринок забезпечення якості програмного забезпечення є критично важливим і складним завданням. Як наслідок, багато компаній зіткнулися з проблемами якості програмного забезпечення. Хоча якість програмного забезпечення по-різному сприймається та визначається багатьма фахівцями в індустрії програмного забезпечення, загальноприйнятим є те, що однією з важливих характеристик високоякісного програмного забезпечення є якомога менше дефектів.

Отже, важливо мати можливість оцінити та передбачити якість програмного забезпечення на ранній стадії розробки, що дозволяє, серед іншого, розподіляти дефіцитні ресурси під час фази тестування, зосереджуючи увагу розробника на частинах програмного забезпечення, схильних до помилок.

У відповідь на цю потребу було створено багато моделей прогнозування, які часто спрямовані на визначення зв’язку між внутрішніми атрибутами якості програмного забезпечення та вмістом їх дефектів (наприклад, схильність до дефектів). На ранній стадії циклу розробки, коли метрики на рівні коду недоступні, складність дизайну є важливим показником дефектів у архітектурі програмного забезпечення.

Показники програмного забезпечення мають важливе значення для розробки програмного забезпечення та для вимірювання складності та якості програмного забезпечення, оцінки вартості та проектних зусиль, і це лише деякі з них. Традиційні показники, такі як функціональна точка, наука про програмне забезпечення та цикломатична складність, добре використовуються в процедурній парадигмі. Однак вони не завжди застосовні до аспектів об’єктно-орієнтованої парадигми.

Важливо розрізняти принципи проектування об’єктно-орієнтованого підходу та принципи проектування функціонально-орієнтованого підходу, щоб прояснити багато аспектів об’єктно-орієнтованого підходу та забезпечити кращу якість та адміністративне управління.

Використання метрик при оцінювання об’єктно-орієнтованих програмних систем дає низку переваг, а саме:

– Порівняльне дослідження різних методологій проектування об’єктно-орієнтованих програмних систем.

– Для аналізу, порівняння та критичного вивчення різних мов програмування щодо їх характеристик.

– Для порівняння оцінки і продуктивності людей, залучених до розробки програмного забезпечення.

– При підготовці специфікацій якості програмного забезпечення.

– При перевірці відповідності програмних систем вимогам і специфікаціям.

– Роблячи висновки щодо зусиль, які необхідно докласти до проектування та розробки систем програмного забезпечення.

– Отримати уявлення про складність коду.

– При прийнятті рішень щодо подальшого поділу комплексного модуля робити чи ні.

– У наданні вказівок менеджеру ресурсів щодо їх належного використання.

– Порівнювати та робити компроміси між розробкою програмного забезпечення та вартістю обслуговування.

– У наданні зворотного зв'язку керівникам програмного забезпечення щодо прогресу та якості на різних етапах життєвого циклу розробки об’єктно-орієнтованого програмного забезпечення.

– У розподілі тестових ресурсів для тестування коду.

Проте, не обійшлося і без обмежень. Основні обмеження при використанні метрик перелічимо нижче:

– Застосування метрик програмного забезпечення не завжди просте, а в деяких випадках це складно та дорого.

– Перевірка та обґрунтування показників програмного забезпечення базується на історичних/емпіричних даних, достовірність яких важко перевірити.

– Вони корисні для керування програмними продуктами, але не для оцінки продуктивності технічного персоналу.

– Визначення та визначення показників програмного забезпечення зазвичай ґрунтується на припущеннях, які не є стандартизованими та можуть залежати від доступних інструментів і робочого середовища.

– Більшість прогнозних моделей покладаються на оцінки певних змінних, які часто точно невідомі.

Об’єктно-орієнтоване проектування та розробка є популярними поняттями в сучасному середовищі розробки програмного забезпечення. Їх часто називають «срібною кулею» для вирішення програмних проблем. Незважаючи на те, що в дійсності не існує жодної «срібної кулі», об’єктно-орієнтована розробка довела свою цінність для систем, які необхідно підтримувати та модифікувати. Розробка об’єктно-орієнтованого програмного забезпечення вимагає підходу, відмінного від більш традиційних методів функціональної декомпозиції та розробки потоку даних. Це включає метрики програмного забезпечення, які використовуються для оцінки об’єктно-орієнтованого програмного забезпечення.

У той час як метрики для функціональної декомпозиції та підходу до проектування вимірюють структуру проекту та даних незалежно, об’єктно-орієнтовані метрики повинні мати можливість зосереджуватися на поєднанні функції та даних як інтегрованого об’єкта [28]. Оцінка корисності метрики як кількісної міри якості програмного забезпечення ґрунтувалася на вимірюванні атрибута якості програмного забезпечення. Однак вибрані показники корисні для багатьох моделей. Таким чином, критерії об’єктно-орієнтованої метрики слід використовувати для оцінки таких атрибутів:

– Ефективність

– Складність

– Зрозумілість

– Можливість повторного використання

– Тестування/ремонтопридатність

Незалежно від того, чи є метрика «традиційною» чи «новою», вона має бути ефективною для вимірювання одного чи кількох цих атрибути.

В об’єктно-орієнтованій системі традиційні метрики зазвичай застосовуються до методів, які містять операції класу. Метод – це компонент об’єкта, який оперує даними у відповідь на повідомлення та визначається як частина опису класу. Методи відображають, як проблема розбита на сегменти та можливості, які інші класи очікують від даного. Дві найпопулярніші традиційні метрики це метрика цикломатичної складності та метрика розміру (кількості строк коду).

Більшість існуючих метрик визначено на рівні окремих програмних компонентів (класів, методів). Типове рішення полягає в знаходженні середнього значення результатів метрики для всіх програмних компонентів. Такий підхід має небажаний ефект згладжування, потенційно розмиваючи погані результати в загальній прийнятній якості.

Цикломатична складність (Cyclomatic Complexity) за МакКейбом використовується для оцінки складності алгоритму в методі. Метод із низькою цикломатичною складністю, як правило, кращий, хоча це може означати, що рішення відкладено через передачу повідомлень, а не те, що метод не є складним. Цикломатичну складність не можна використовувати для вимірювання складності класу через успадкування, але цикломатичну складність окремих методів можна комбінувати з іншими показниками для оцінки складності класу. Хоча ця метрика спеціально застосовна до оцінки такого атрибута якості як складність, вона також пов’язана з усіма іншими атрибутами [29].

Метрика для оцінки розміру методу використовується для оцінки простоти розуміння коду розробниками та тими, хто продукт буде підтримувати. Розмір можна виміряти різними способами. Вони включають підрахунок усіх фізичних рядків коду, кількість операторів і кількість порожніх рядків. Порогові значення для оцінки показників розміру відрізняються залежно від використовуваної мови програмування та складності методу. Однак, оскільки розмір впливає на легкість

розуміння, підпрограми великого розміру завжди створюватимуть вищий ризик щодо таких атрибутів, як зрозумілість, можливість повторного використання та придатність до обслуговування [29].

Як згадувалося раніше, для об'єктно-орієнтованих систем є багато різних показників. Клас – це шаблон, з якого можна створювати об’єкти. Цей набір об’єктів має спільну структуру та спільну поведінку, що проявляється набором методів. Вибрані три метрики для класів, описані нижче, вимірюють складність класу, використовуючи його методи, повідомлення та зв’язність.

Метрика Weighted Methods per Class (WMC) – це показник кількості методів, реалізованих у класі, або сума складностей методів (складність методу вимірюється цикломатичною складністю). Друге вимірювання важко реалізувати, оскільки не всі методи доступні в ієрархії класів через успадкування. Кількість методів і складність задіяних методів є показником того, скільки часу та зусиль потрібно для розробки та підтримки класу. Чим більша кількість методів у класі, тим більший потенційний вплив на дочірні класи, оскільки дочірні класи успадковують усі методи, визначені в батьківському. Класи з великою кількістю методів, швидше за все, будуть більш специфічними для кожної частини програмної системи, що обмежує можливість повторного використання. Цей показник вимірює зрозумілість, ремонтопридатність і придатність до повторного використання [28, 30].

Метрика Response for a Class (RFC) – це набір усіх методів, які можуть бути викликані у відповідь на повідомлення до об’єкта класу або якимось методом у класі. Це включає всі методи, доступні в ієрархії класів. Ця метрика розглядає комбінацію складності класу через кількість методів і рівень зв’язку з іншими класами. Чим більша кількість методів може бути викликана з класу через повідомлення, тим більша складність класу. Якщо велика кількість методів може бути викликана у відповідь на повідомлення, тестування та налагодження класу стає складним, оскільки це вимагає більшого рівня розуміння з боку тестувальника. Найгірше значення для можливих відповідей допоможе правильно

розподілити час тестування. Цей показник оцінює зрозумілість, придатність до підтримки та тестування [30].

Метрика Number of Children (NoC) – це кількість безпосередніх підкласів, підпорядкованих класу в ієрархії. Це індикатор потенційного впливу, який клас може мати на архітектуру і систему. Чим більша кількість дочірніх елементів, тим більша ймовірність неправильного абстрагування батьківського елемента та може бути випадком неправильного використання підкласу. Але чим більша кількість дочірніх класів, тим більша можливість повторного використання, оскільки успадкування є формою повторного використання. Якщо в класі велика кількість дочірніх класів, може знадобитися більше часу на тестування цього класу. Таким чином, NOC в першу чергу оцінює ефективність, можливість повторного використання та придатність до тестування [28, 30].

Метрику Depth of Inheritance Tree можна визначити як максимальну довжину шляху від класу до кореневого класу в дереві успадкування. Чим глибше клас буде в ієрархії, тим більша можливість успадкувати більшу кількість методів, що його створюють складніше передбачити його поведінку.

2.2 Валідація метрик оцінки програмних систем

Метрики вважається фундаментальною частиною будь-якої інженерної дисципліни, і дисципліни програмної інженерії не є винятком. У цьому контексті метрики програмного забезпечення стосуються вимірювань, які можна застосувати для перевірки показників процесів, проектів і програмних продуктів. Оцінка якості програмного забезпечення за допомогою вимірювань дозволяє кількісно визначити успішність чи неуспішність певного атрибута, виявивши потребу у вдосконаленні. Управління якістю програмного забезпечення може дозволити досягти низької кількості дефектів і надійних стандартів ремонтопридатності, надійності та портативності.

Під час оцінки програмного забезпечення кожен показник повинен бути перевірений на валідність. Існує два типи перевірки метрик програмного

забезпечення: «теоретична» та «емпірична» перевірка. Ці два типи перевірки метрик програмного забезпечення наведені на рисунку 2.1.

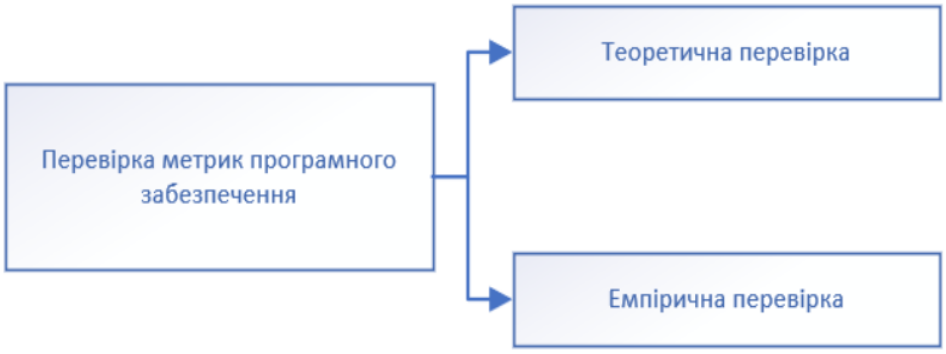


Рисунок 2.1 – Типи перевірки метрик програмного забезпечення

Існує два основні методи перевірки, які наведені в моделях метрик. Теоретична перевірка підтверджує, що вимірювання не порушує жодних необхідних властивостей елементів метрики. Емпірична перевірка підтверджує, що виміряні значення атрибутів узгоджуються зі значеннями, передбаченими моделями, що включають атрибут. Теоретичні методи перевірки дозволяють проводити дійсні вимірювання щодо певних визначених критеріїв, а емпіричні методи є підтверджуючими доказами працездатності чи недійсності [36].

Ці два типи методів перевірки реалізуються під час внутрішньої та зовнішньої перевірки.

Внутрішня перевірка – це теоретична методологія, яка гарантує, що метрика є правильною числовою характеристикою властивості, яку вона, як стверджує, вимірює. Демонстрація того, що метрика вимірює те, що вона має намір вимірювати, є формою теоретичної перевірки.

Методологія зовнішньої перевірки передбачає емпіричне дослідження показників програмного забезпечення. Крім того, внутрішні та зовнішні перевірки зазвичай називають «теоретичними та емпіричними перевірками».

Теоретична перевірка вимагає властивостей для перевірки програмних показників. Теоретична перевірка ґрунтується на аналізі властивостей атрибута, що підлягає вимірюванню. Теоретична перевірка надає інформацію про математичні та статистичні операції, які можна виконати з мірою, що є важливим під час роботи з мірою. Існують два основні підходи до теоретичної перевірки, і це репрезентативна теорія вимірювання та підходи, що ґрунтуються на властивостях (Рис. 2.2). Основна мета теоретичної перевірки полягає в тому, щоб оцінити, чи метрика дійсно вимірює те, що вона має на меті вимірювати, а теоретична перевірка метрик встановлює їх конструктивну валідність, тобто, це доводить, що вони є дійсними мірами для конструкцій, які використовуються як змінні в дослідженні.

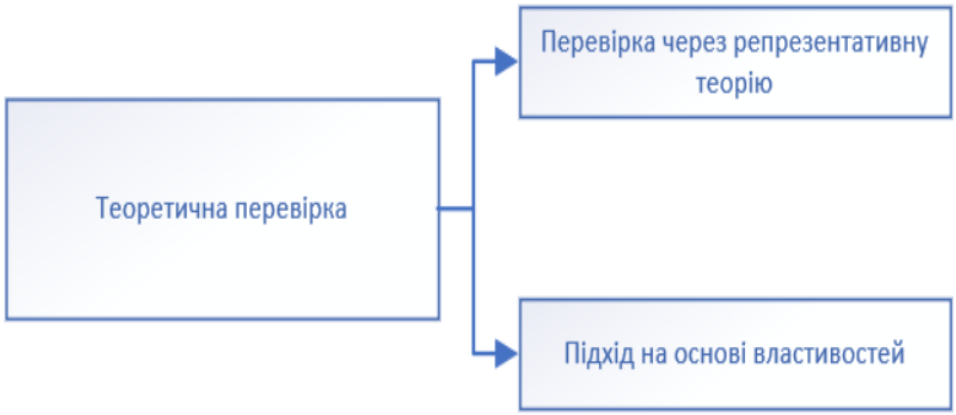


Рисунок 2.2 – Підходи до теоретичної перевірки метрик

Ще не існує стандартного загальноприйнятого способу теоретичної перевірки метрики програмного забезпечення. Підхід, заснований на репрезентативній теорії, пояснюється Кітченхемом Б., Пфлігером С.Л., і Фентоном Н. [37]. Підхід на основі властивостей, званий аксіоматичним підходом, запропонований Вейюкером [31] та і Бріан Л. та ін. [38].

Репрезентативна теорія перевірки базується на гомоморфізмі між емпіричним світом і числовим світом, а також на умові представлення, яка засвідчує, що властивості атрибутів у реальному світі повинні підтримуватися мірами в числовому світі. Це передбачає визначення емпіричного та числового світу та побудову відображення між ними. Цей тип перевірки називається внутрішньою перевіркою заходів для оцінки та теоретичної перевірки.

Теоретичні підходи, засновані на властивостях, використовуються з низкою аксіом для теоретичної перевірки, і ця категорія теоретичної перевірки також називається «аксіоматичною, аналітичною та алгебраїчною перевіркою». Найбільш відомі з них зображені на рисунку 2.3.

Набір аксіом визначено для програмного атрибута. Вейюкер [31] запропонував дев’ять властивостей для оцінки синтаксичних метрик програмного забезпечення. Ці властивості використовуються для застосовності об’єктно-орієнтованих метрик, і лише властивості необхідні для підтвердження дійсності метрики. Бріан, Л. та ін. [38] описують властивості метрик для розміру, складності, довжини, зв’язку і зв’язності. Вони стверджують, що умова представлення є очевидною передумовою для перевірки.

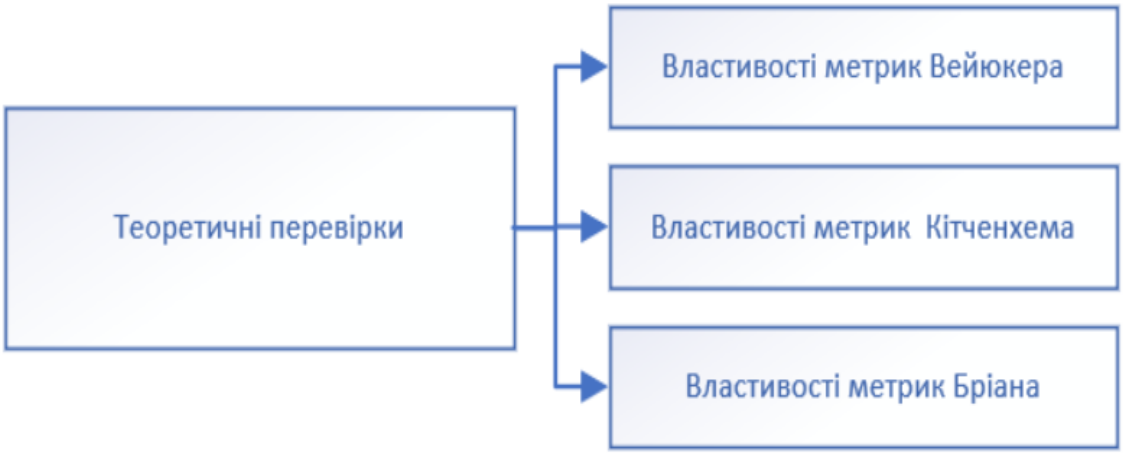


Рисунок 2.3 – Найпопулярніші теоретичні перевірки

Процедура емпіричної перевірки суттєво відрізняється залежно від мети заходів, тобто оцінювання чи прогнозування, типу та кількості зібраних даних. Один показник можна використовувати для прогнозування значення іншого показника. Запропоновані на даний момент метрики перевіряються за допомогою теоретичних і емпіричних підходів. Три типи емпіричних перевірок – це опитування, тематичні дослідження та експерименти зображені на рисунку 2.4.

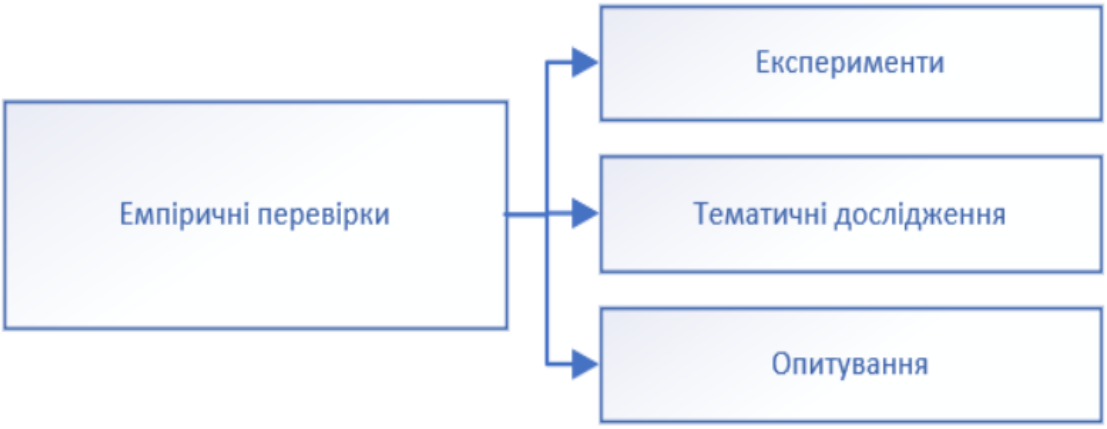


Рисунок 2.4 – Типи емпіричних перевірок

Відповідно до властивостей метрик, не всі метрики задовольняють критерії перевірки. А метрики, які не задовольняють критерії, можна сміливо виключати.

Проте є теоретичні та емпіричні докази перевіреного метриками програмного забезпечення. Підтвердженням перевірки є те, що сотні метрик були визначені та перевірені в минулому.

Як зазначив Розенберг [39], коли метрики використовуються для оцінки проектів, немає вказівок для інтерпретації їх результатів. Часто кваліфікація результату базується на здоровому глузді та досвіді. Визначення прийнятного значення залежить від вимог підприємства та досвіду розробника. Наприклад, деякі компанії вимагають, щоб глибина успадкування не перевищувала заданий поріг, тоді як інші зосереджуються на загальній архітектурі або на використанні стандартів іменування.

2.3 Методологія Вейюкера та Кітченхема

Властивості Вейюкера [31] відіграють важливу роль в оцінці показників складності програмного забезпечення. Ці властивості використовуються для оцінки надійності вимірювання та, у свою чергу, призводять до визначення хороших понять складності програмного забезпечення. За допомогою цих властивостей можна визначити найбільш прийнятну міру серед різних доступних мір складності. Більшість нових запропонованих методів [3] [12] використовують ці властивості з метою оцінки. Проте кілька авторів [32] [33] і [34] також прокоментували властивості Вейюкера.

Відповідно до Фентона, властивості Вейюкера не ґрунтуються на єдиному узгодженому погляді на складність [34]. Цузе [32] коментує, що ці властивості не узгоджуються з принципами масштабування. Чернявський і Сміт [33] критикують, що властивості Вейюкера слід використовувати обережно, оскільки властивості можуть давати лише необхідну, але не достатню умову для хорошої метрики складності. Розглядаючи вищезазначені пункти, можна зробити висновок, що властивості Вейюкера можуть бути недостатніми для визначення якості вимірювання складності програмного забезпечення. З іншого боку, аналізи цих властивостей все ще потрібні. Після цього може бути рекомендовано додати або видалити ці властивості. Потім ці властивості можна застосувати для визначення ефективності програмного забезпечення будь-якої складності.

Властивості Вейюкера також використовуються для оцінки метрики складності для об'єктно-орієнтованого програмування. Тому ці властивості також відіграють важливу роль у виборі найбільш підходящої метрики для об’єктно-орієнтованого програмування.

Основні дев'ять властивостей, запропонованих Вейюкером [31], розглянемо нижче. Використовуються такі позначення:

– P, Q і R позначають класи;
– $P + Q$ позначає комбінацію класів P і Q;
– μ позначає вибрану метрику;

– $\mu(P)$ позначає значення метрики для класу P;

– $P \equiv Q$ (P еквівалентно Q) означає, що два класи, P і Q, забезпечують однакову функціональність.

Визначення комбінації двох класів тут прийнято таким же, як запропоновано в [35], тобто, комбінація двох класів продукує інший клас, властивості якого (методи та змінні екземпляра) є об'єднанням властивостей класів компонентів. Крім того, «комбінація» за Вейюкером означає поняття про «конкатенацію» класів.

Властивість 1. Не грубість: задано клас P і метрику μ, інший клас Q завжди можна знайти так, що $\mu(P) \neq \mu(Q)$.

Ця властивість стверджує, що метрика не повинна оцінювати всі програми як однаково складні. Зроблено висновок, що будь-яка метрика складності, яка класифікує всі програми як однаково складні, не може вважатися хорошою метрикою складності. Тому що, якщо будь-яка метрика класифікує всі програми як однаково складні, немає значення метрики складності, яка не може розрізнити всі можливі програми.

Іншими словами, жодна метрика складності не класифікує всі програми як однаково складні, тому всі метрики складності повинні задовольняти цю властивість.

Властивість 2. Деталізація: існує кінцева кількість випадків, що мають однакове значення метрики. Цій властивості відповідатиме будь-який показник, виміряний на рівні класу.

Ця властивість стверджує, що може існувати лише кінцева кількість програм із заданим значенням складності. Більша частина міри складності повинна задовольняти цій властивості; оскільки існує лише кінцева кількість програм однакової складності. Сама Вейюкер каже, що міра недостатньо чутлива, якщо вона ділить усі програми лише на кілька класів складності. Властивість 2 є спробою формалізувати цю інтуїцію. Для будь-якої міри складності можна припустити деяке найбільше можливе число, яке можна прийняти як верхню

межу. Проблема невдачі деяких метрик складності є те, що вони оцінюють занадто багато програм як однаково складні.

Властивість 3. Не унікальність (поняття еквівалентності): можуть існувати різні класи P і Q, такі що $\mu(P) = \mu(Q)$.

Ця властивість стверджує, що навіть якщо існують різні програми, складність цих програм може бути однаковою, тобто існує кілька програм однакової складності.

У кожному показнику складності є кілька програм однакової складності. Хоча для безперервних вимірювань метрик можна вважати рівними лише в межах обмежень похибки вимірювань, ця властивість є розумною вимогою. Це також узгоджується з вимірюваннями інших атрибутів в інших доменах.

Властивість 4. Деталі реалізації важливі: для проектів двох класів, P і Q, які забезпечують однакову функціональність, це не означає, що метричні значення для P і Q будуть однаковими.

Ця властивість стверджує, що навіть якщо дві програми обчислюють ту саму функцію, складність програми визначається деталями реалізації. Незважаючи на те, що ця властивість задовольняється всіма показниками складності, але коли складність однакова, вихідне значення має бути різним для двох різних реалізацій.

Властивість 5. Монотонність: для всіх класів P і Q має виконуватися наступне твердження: $\mu(P) \leq \mu(P+Q)$ і $\mu(Q) \leq (P+Q)$, де $P+Q$ означає комбінацію P і Q.

Ця властивість стверджує, що вимірювання для комбінації двох програмних сутностей ніколи не може бути меншим, ніж вимірювання, виконане для будь-якої з компонентів програмної сутності. Це також важлива властивість для оцінки будь-якої метрики складності, тому її можна використовувати для оцінки.

Ця властивість відображає думку про те, що розмір програми є ключовим фактором її складності [34]. Він не включає низьку зрозумілість як ключовий фактор складності. Це пояснюється тим, що широко поширена думка, що в деяких

випадках ми можемо легше зрозуміти програму, коли ми бачимо її більше. Таким чином, тип розміру метрик складності повинен задовольняти цю властивість.

Властивість 6. Не еквівалентність взаємодії: $\exists P$, $\exists Q$, $\exists R$ така, що $\mu\,(P) = \mu\,(Q)$ не означає, що $\mu(P + R) = \mu\,(Q + R)$.

Ця властивість стверджує, що взаємодія між P і R може відрізнятися від взаємодії між Q і R, що призводить до різних значень складності для R і Q, де складність P і Q однакова.

Оскільки і цикломатичне число, і метрика зважених методів у класі мають притаманну складність, яка є статичною, незалежно від їхнього контексту, вони не здатні відобразити цю можливу різницю у взаємодії, і, отже, жодна з них не задовольняє цій властивості.

Властивість 7. Перестановка елементів в об’єкті вимірювання може змінити метричне значення.

Ця властивість стверджує, що перестановка елементів у вимірюваному елементі може змінити значення складності. Мета полягає в тому, щоб забезпечити зміну значень складності через перестановку операторів програми. Метрики цикломатичної складності та зважених методів у класі не задовольняють цій властивості, оскільки складність програми повністю не залежить від розміщення, а отже, потенційної взаємодії між операторами програм, які використовують ці показники. Навпаки, оскільки розташування операторів може впливати на їхню взаємодію і, отже, на складність програми, якщо оцінювати її за допомогою показника потоку даних і показника когнітивної складності, можна очікувати, що ця властивість буде зберігатися.

Властивість 8. Коли ім’я вимірюваного об’єкта змінюється, метрика має залишатися незмінною.

Як і в усіх метриках складності, зміна назви програми не впливає на значення складності, тому це безглузда властивість для оцінки будь-якої метрики.

Властивість 9. Взаємодія збільшує складність. $\exists P$ та $\exists Q$ такі, що: $\mu\,(P) + \mu\,(Q) < \mu\,(P + Q)$.

Ця властивість допускає можливість того, що в міру того, як програма розростається зі своїх складових програмних тіл, вводиться додаткова складність, тобто комбінована програма може бути складнішою, ніж її складові частини.

Метрики цикломатичної складності та зважених методів в класі не задовольняють цій властивості. І цикломатичне число, і число зважених методів у класі розглядають тіла програм як такі, що мають невід'ємну складність, яка є статичною, незалежно від їхнього контексту, отже, властивість 9 не виконується для жодної з цих метрик.

Деякі дослідники розкритикували список властивостей Вейюкера; однак це широко відомий формальний підхід і служить важливим заходом для оцінки показників метрики.

Однак у наведеному вище списку властивості 2 і 8 тривіально задовольнятимуть будь-які метрики, визначені для класу.

Друга властивість Вейюкера «деталізація» вимагає лише наявності кінцевої кількості випадків, які мають однакове значення метрики. Цей показник відповідатиме будь-якому показнику, виміряному на рівні класу.

Властивість 8 також задовольнятиметься всіма показниками, вимірянними на рівні класу, оскільки на них не впливатимуть імена класів або методи та змінні екземпляра класу.

Властивість 7 вимагає, щоб перестановка операторів програми могла змінити значення метрики. Ця метрика має значення в традиційному проектуванні програм, де порядок блоків if-then-else може змінити логіку програми і, отже, метрику. В об'єктно-орієнтованому проектуванні клас є абстракцією проблеми реального світу, і впорядкування операторів у класі не матиме жодного ефекту в остаточному виконанні. Отже, було припущено, що властивість 7 не підходить для метрики об'єктно-орієнтованого проектування.

Аналітична оцінка необхідна для математичної перевірки правильності метрики. Наприклад, властивості 1, 2 і 3, а саме не грубість, деталізація і не унікальність, є загальними властивостями, яким має задовольняти будь-який показник. Оцінюючи метрику за будь-якою властивістю, можна проаналізувати

природу метрики. Наприклад, властивість 9 зазвичай не задовольнятиме жодна метрика, для якої високі значення є показником поганої архітектури, виміряного на рівні класу. У випадку, якщо це так, це означатиме, що це випадок поганої композиції, і класи, якщо їх об’єднати, потребують реструктуризації. Провівши аналітичну оцінку метрики, можна продовжити її перевірку на основі даних.

Видатні дослідники Кітченхемом Б., Пфлігером С.Л., і Фентоном Н. запропонували властивості для перевірки метрик програмного забезпечення та пояснили, як перевірити метрики на їх валідність [37]. Вони запропонували основне припущення для метрик щодо задоволення двох умов: метрика не повинна порушувати необхідні властивості, і кожна модель, що використовується в процесі вимірювання, має бути дійсною (рисунок 5).

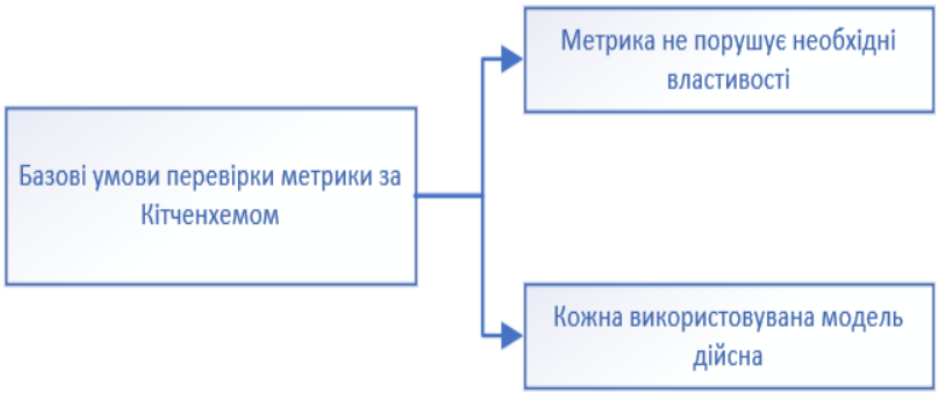


Рисунок 2.5 – Базові умови перевірки метрики за Кітченхемом

Формально, Кітченхемом визначена дійсна метрика, оскільки вона не може підтвердити теорію, а може лише її фальсифікувати. Отже, дійсна метрика – це та, який не може бути визначена недійсною.

Щоб визначити достовірність вимірювань, необхідно підтвердити різні типи достовірності вимірювань, запропоновані Кітченхемом. і вони проілюстровані в таблиці 2.1.

Типами достовірності вимірювань програмного забезпечення є валідність атрибутів, одиниць, інструментів і протоколів.

Достовірність атрибута – атрибут фактично демонструється сутністю. Достовірність атрибутів необхідно розглядати як для безпосередньо вимірюваних атрибутів, так і для опосередковано вимірюваних атрибутів, які є похідними від інших атрибутів.

Достовірність одиниці – одиниця вимірювання, що використовується, є відповідним засобом вимірювання атрибута.

Достовірність інструменту означає, що будь-яка модель, що лежить в основі вимірювального інструменту, має бути дійсною, а вимірювальний інструмент має бути належним чином відкалібрований.

Достовірність протоколу – прийнятий допустимий протокол вимірювання.

Таблиця 2.1 – Достовірності для підтвердження

Достовірність	Вимір
Достовірність 1	Атрибут
Достовірність 2	Одиниця
Достовірність 3	Інструмент
Достовірність 4	Протокол

Кітченхем визначив властивості для вимірювання, які називаються «Властивості Кітченхема», і заявив, що метрики програмного забезпечення повинні демонструвати ці властивості.

Властивості Кітченхема для метрики зображені на рисунку 2.6.

Рисунок 2.6 – Прямі властивості Кітченхема

Властивість 1: щоб атрибут був вимірюваним, він повинен дозволяти відрізняти різні сутності один від одного.

Властивість 2: дійсна метрика повинна підкорятися умовам початкових тверджень, тобто вона повинна зберігати інтуїтивні уявлення про атрибут і спосіб, у який він розрізняє різні сутності.

Властивість 3: кожна одиниця атрибута, що вносить свій внесок у дійсну метрику, є еквівалентною.

Властивість 4: різні сутності можуть мати однакове значення атрибута в межах похибки вимірювання.

2.4 Висновки

У другому розділі проаналізовано наявні методології для перевірки метрик оцінки програмного забезпечення. Досліджено роботи у даній області, виділено ключові аспекти основних методологій. Визначено методології основані на властивостях такі, як найбільш придатні для перевірки розроблюваної метрики.

З основних методологій за властивостями виділено «Властивості Вейкюкера» та «Властивості Кітченхема». Проаналізовано їх принципи оцінки та основні положення.

Найчастіше для перевірки метрик програмного забезпечення використовують перевірку Вейюкера, а тому вона буде обрана для перевірки розроблюваної метрики оцінки складності програмного забезпечення.

Також у розділі розглянуто припущення, на яких буде будуватися розроблювана метрика. Це дозволить встановити певні рамки розробки, а також полегшити перевірку цієї метрики.

3 УДОСКОНАЛЕННЯ МЕТРИКИ ОЦІНКИ СКЛАДНОСТІ ПРОГРАМНОЇ СИСТЕМИ

3.1 Вимоги оцінювання метрики складності програмної системи

Одна з проблем аналізу систем – оцінювання їх складності. Складність системи є якісною характеристикою. Зменшення складності програмної системи дає змогу знизити трудомісткість проектування, розробки, тестування та супроводження, забезпечує простоту і надійність виробленої програмної системи. У даний час питанням управління якістю програмних систем приділяється підвищена увага, пов’язано це з ростом застосувань програмних систем у різних сферах діяльності людини. Управляння якістю включає в себе планування якості, забезпечення якості і контроль якості. Таким чином, однією зі складових розробки програмної системи із запланованим рівнем якості є контроль, який базується на застосуванні метрик якості для вимірювання основних показників в процесі розробки програмної системи.

Складні програмні системи мають ряд специфічних особливостей, які визначають особливості аналізу їхньої роботи.

Вкрай важливою особливістю складних програмних систем є наявність корпоративного користувача як однієї зі складових. Під поняттям «корпоративний користувач» розумітимемо організацію, що працює з програмною системою за вказаною технологією.

Користувач налаштовує систему, використовує для своєї мети результати роботи системи, вводить у систему дані, необхідні йому самому та іншим користувачам системи. Кількість користувачів, що одночасно працюють із сучасними програмними системами, вимірюється досить великими числами. Наприклад, кількість одночасно активних робочих місць у системі автоматизації великого підприємства обмежується десятками тисяч, у сучасному великому банку одночасно працюють тисячі користувачів.

Крім великої кількості одночасно активних робочих місць, важливою особливістю є наявність складних технологічних процесів, що автоматизуються системою, при цьому різні етапи цих процесів виконуються різними користувачами.

Різні ПС характеризуються різною жорсткістю технологій роботи користувача у системі.

Деякі основні припущення, використані для удосконалення нової метрики, були взяті у Чідамбера та Кемерера з [4] щодо розподілу методів та змінних екземплярів у обговорених метричних властивостях.

Припущення 1:

Нехай X_i буде рівне кількість методів у заданому класі i;

Y_i рівне кількості методів, викликаних із заданого методу i;

Z_i рівне кількості змінних екземплярів, які використовує метод i

X_i, Y_i, Z_i є дискретними випадковими величинами, кожна з яких характеризується деякими загальними функціями розподілу. Крім того, усі X є незалежними та однаково розподіленими. Те саме стосується всіх Y і Z. Це означає, що кількість методів і змінних відповідає статистичному розподілу, який не є очевидним для спостерігача системи. Крім того, цей спостерігач не може передбачити змінні та методи одного класу на основі знання змінних та методів іншого класу в системі.

Припущення 2: загалом, два класи можуть мати кінцеву кількість «ідентичних» методів у тому сенсі, що поєднання двох класів в один клас призведе до того, що версія ідентичних методів одного класу стане зайвою. Наприклад, клас «foo_one» має метод «draw», який відповідає за малювання піктограми на екрані; інший клас «foo_two» також має метод «draw». Тепер дизайнер вирішує мати єдиний клас «foo» і об’єднує два класи. Замість двох різних методів «малювання» дизайнер може вирішити мати лише один метод для виконання «малювання».

Припущення 3: дерево наслідування є «повним», тобто є корінь, проміжні вузли та листя. Це припущення просто стверджує, що програма не складається лише з окремих класів; існує деяке використання підкласів.

Відповідно до визначених у попередньому розділі припущень було розроблено нову метрику оцінки складності об’єктно-орієнтованої програмної системи. Розроблена метрика використовується для оцінки складності класів програмної системи ще на етапі проектування. В неї закладені певні ідеї, які будуть описані в наступному підрозділі. Ця метрика дозволить оцінювати складність об’єктно-орієнтованих програмних систем за дуже простим алгоритмом, проте її результати не будуть поступатися більш складним метрикам оцінки програмних систем.

Метрика, буде основана на кількості атрибутів та методів у класі, та матиме назву «Складність через методи та атрибути» (СЧМА). Для неї розроблено формулу а також таблиці відповідностей між показниками та відносними значеннями компонентів формули.

Формула метрики:

$$\text{СЧМА} = \text{M}' + \text{A}'$$

M’ – відображає значення, яке основане на діапазоні кількості наявних методів у класі (приватних, публічних та захищених). Відповідності кількості методів до відносного значення наведені у таблиці 3.1.

Таблиця 3.1 – Відносне значення M’

Кількість методів у класі	**Відносне значення** M’
0 чи 1-5	1
6-10	2
11-15	3
16-20	4
21+	5

A’ – відображає значення, яке основане на діапазоні кількості наявних атрибутів у класі (приватних, публічних та захищених). Відповідності кількості атрибутів до відносного значення наведені у таблиці 3.2.

Таблиця 3.2 – Відносне значення A’

Кількість атрибутів у класі	**Відносне значення** A’
0 чи 1-5	1
6-10	2
11-15	3
16-20	4
21+	5

Алгоритм застосування метрики такий: спочатку підраховується будь яким програмним засобом кількість методів та атрибутів для кожного класу. Після цього, відповідно до отриманих результатів та таблиць відповідностей, які наведені вище, для кожного класу визначається показник СЧМА. Далі отриманий показник можна співвіднести з критеріями для аналізу результатів, які будуть

наведені в пункті 3.3, та отримати відповідні для отриманих значень результати та оцінки програмної системи.

Принципи метрики СЧМА такі:

– Чим більше методів та атрибутів містить клас, ти більше значення СЧМА, що в свою чергу збільшує об’єм та складність класів. Це тягне за собою збільшення часу та витрат на розробку.

– У випадку наслідування, дочірні класи мають доступ до атрибутів та методів батьківських, і якщо батьківський клас має більше значення СЧМА, це призводить до покращення показника СЧМА у дочірньому класі через повторне використання методів та атрибутів.

– Дуже високе значення СЧМА (~10) ще на етапі проектування сповіщає про потенційно великі затрати на реалізацію спроектованого рішення.

– Кількість методів та атрибутів в класі є першим індикатором того, наскільки багато зусиль доведеться прикласти для реалізації та підтримки проектованої програмної системи.

3.2 Критерії аналізу результатів метрики

Показники програмного забезпечення стають частиною структури розробки програмного забезпечення, важливою для розуміння того, чи відповідає якість програмного забезпечення, яке ми створюємо, нашим очікуванням . Як наслідок, було запропоновано багато різних показників і безліч інструментів для їх обчислення та проведення оцінки якості. Беручи до уваги різні зацікавлені сторони, які беруть участь у проектах програмного забезпечення (наприклад, розробники, менеджери, користувачі), якість потрібно оцінювати на різних рівнях деталізації. Однак практичне застосування метрик програмного забезпечення ускладнюється через необхідність комбінувати різні метрики, як це рекомендовано методами проектування моделі якості, такими як фактор-критерій-метрика або ціль-питання-метрика. Існує потреба отримати розуміння якості всієї

системи на основі метричних значень, отриманих для елементів системи низького рівня, таких як класи та методи.

Для простішої інтерпретації результатів застосування метрики було розроблено декілька типів результатів та таблиці відповідностей з критеріями для кожного з них.

В залежності від отриманого значення нашу систему можна визначити як: складну, помірно складну та відносно просту.

Дані таблиці відповідностей дозволять швидко і легко розпізнавати програмні системи, інтерпретувати отримані результати розробленої метрики, що дозволить ефективніше її використовувати. Це беззаперечний плюс в порівнянні з наявними рішеннями, оскільки лише деякі з них дозволяють проводити такі паралелі та мають детальні таблиці для розшифрування результатів.

Також проведено спробу провести взаємозв’язок між нашою класифікацією та факторами якості програмного забезпечення (було обрано такі фактори, як зрозумілість, тестування та рівень ризику). Дані взаємозв’язки відображені у таблицях відповідностей.

Це дає змогу бачити взаємозв’язок між складністю програмної системи, та тим як її збільшення впливає на якість самого продукту. Це дуже важливий момент, адже це дозволяє раніше виявляти потенційну проблемність проектованої програмної системи, а отже і уникати цих помилок. Це вкрай важливо в сучасних реаліях розробки програмних продуктів. Також це суттєво зменшує витрати на розробку та час, який витратять на це розробники.

Запропонована модель взаємозв’язків дозволяє уявити картину про те, наскільки тісно зв’язані показники складності програмної системи з її якістю, та як це впливає на весь процес розробки програмного продукту.

Першим випадком буде результат «Відносно проста». Застосовний він для систем які отримали значення СЧМА від 2 до 4. Вплив на якість програмної системи відображено у таблиці 3.3.

Таблиця 3.3 – Атрибути якості для результату «Відносно проста»

Зрозумілість	Тестування	Рівень ризику
Програмна система зрозуміла. Переважна кількість її кодової бази підпадає під визначення «чистий код». Потрібно менше затрат та сил для імплементації методів та атрибутів у даній системі, що позитивно впливає на час її розробки.	Тестування такої програмної системи легке, тому що будуть тестуватися прості та добре структуровані методи.	Низький

Другим випадком буде результат «Помірно складна». Застосовний він для систем які отримали значення СЧМА від 5 до 8. Вплив на якість програмної системи відображено у таблиці 3.4.

Таблиця 3.4 – Атрибути якості для результату «Помірно складна»

Зрозумілість	Тестування	Рівень ризику
Програмна система складна. Переважна кількість її кодової бази має не зовсім чистий код. Потрібно менше затрат та сил для імплементації методів та атрибутів у даній системі, що позитивно впливає на час її розробки.	Тестування такої програмної системи менш стабільне, тому що буде тестуватися більше складних методів. Це тягне за собою більші витрати на розробку та тестування системи.	Помірний

Останнім випадком буде результат «Складна». Застосовний він для систем які отримали значення СЧМА 9 чи 10. Вплив на якість програмної системи відображено у таблиці 3.5.

Таблиця 3.5 – Атрибути якості для результату «Складна»

Зрозумілість	**Тестування**	**Рівень ризику**
Програмна система має складну поведінку. Переважна кількість її кодової бази важка для розуміння та має погану структуру. Класи в такій системі дуже складно і довго реалізуються, що негативно впливає не весь процес розробки та якість програмної системи в цілому.	Тестування такої програмної системи не стабільне, тому що будуть тестуватися складні методів. Також в такій системі можлива наявність методів які взагалі неможливо протестувати.	Високий або ж дуже високий

3.3 Висновки

У даному розділі викладено визначення нової розробленої метрики СЧМА. Описано формулу, компоненти формули та спосіб їх визначення. Надано алгоритм застосування метрики. Також висвітлено ідеї, закладені в цю метрику. Ідеї демонструють, що дійшовши компромісу в деяких моментах, метод можна зробити простим, при тому не нехтувати основами оцінки об’єктно-орієнтованих програмних систем.

У цьому розділі також надано критерії для інтерпретації результатів розробленої метрики. Подано це у таблицях відповідностей, що дозволяє чітко розділити результати метрики на певні категорії. Відповідно до потрапляння у категорію, програмна система буде мати певні характеристики та атрибути якості. Додатково, це дозволить користувачам з легкістю визначати та розуміти отримані результати без додаткових порівнянь чи дій.

4 ВАЛІДАЦІЯ ТА ВЕРИФІКАЦІЯ РОЗРОБЛЕНОЇ МЕТРИКИ

4.1 Валідація метрики СЧМА за властивостями Вейюкера

У цьому розділі проведена аналітична оцінка метрики СЧМА за властивостями Вейюкера.

Властивість 1 (не грубість) і властивість 3 (не унікальність) задовольняються для розробленої метрики, оскільки передбачається, що існує статистичний розподіл методів і атрибутів між класами.

Властивість 4 (деталі реалізації важливі) виконується, оскільки вибір методів і атрибутів залежить від реалізації проекту. Коли два класи поєднуються, кількість методів і атрибутів ніколи не може перевищувати кількість окремих класів у цій системі.

Розглянемо властивість 5 (монотонність). Розглянемо три класи P, Q і R. Нехай значення метрики для класу P і класу Q однакові. Також нехай клас R має спільні методи та атрибути з класом P, але не з класом Q. Таким чином, комбінація класу P і класу R матиме менше значення метрики, ніж комбінація класу Q і класу R. Отже, властивість під номером 5 виконується.

Властивість 6 (не еквівалентність) також виконується. Нехай значення A' і M' для класу P будуть a і m, для класу Q – a' і m', а для класу $P + Q$ - a'' і m''. Через те, що у них будуть спільні методи, виконуватимуться закономірності:

$$a'' \leq a + a' \text{ та } m'' \leq m + m'$$

Властивість 2 (деталізація) тривіально задовольняє будь-яка метрика, визначена для класу. За таким же принципом наша метрика задовольняє властивість 8, а саме, коли ім’я вимірюваного об’єкта змінюється, метрика повинна залишатися незмінною.

Єдина властивість, яку не задовольняє наша метрика це дев’ята.

Причина цього, полягає в тому, що внаслідок розбиття класу відбувається загальне збільшення значення A' і M' для всіх створених підкласів. Іншими словами, складність зростає.

Підсумки валідації метрики наведені в таблиці 4.1. Як бачимо, наша метрика задовольняє 7 з 9 властивостей за Вейюкером, що є досить хорошим результатом та свідчить про те що розроблена метрика придатна для використання на реальних прикладах.

Таблиця 4.1 – Результати валідації розробленої метрики СЧМА за властивостями Вейюкера.

Властивість	СЧМА
Властивість 1	Задовольняє
Властивість 2	Задовольняє
Властивість 3	Задовольняє
Властивість 4	Задовольняє
Властивість 5	Задовольняє
Властивість 6	Задовольняє
Властивість 7	Неможливо визначити
Властивість 8	Задовольняє
Властивість 9	Не задовольняє

4.2 Верифікація метрики СЧМА

Верифікація метрики потрібна для доведення її достовірності, відповідності вимогам та підтвердження правильності роботи. Верифікація надасть можливість з впевненістю сказати, що метрика готова до застосування на реальних прикладах, а також те що вона демонструє очікувані результати роботи, є достовірною, тобто такою, яка своїми результатами демонструє реальний стан речей. Верифікація метрики буде проводитися в два етапи. Перший етап це застосування її на реальних проектах. Це дозволить протестувати метрику, продемонструвати її роботу на реальному проекті. Другий етап буде порівняльний. Ми порівняємо результати роботи розробленої метрики з результатами уже існуючих. Також

проведемо кореляцію цього порівняння. Це дозволить перевірити, чи метрика справді дає достеменні результати.

4.2.1 Застосування СЧМА на програмних системах з відкритим кодом

Щоб ефективно верифікувати запропоновану метрику складності об’єктно-орієнтованих програмних систем та її зв’язок із факторами якості, було обрано провести її тестування на програмному забезпеченні з відкритим вихідним кодом. Для експериментів вибрано програмне забезпечення середнього розміру.

Коли проект має відкритий код, це означає, що будь-хто може вільно використовувати, вивчати, змінювати та поширювати цей проект для будь-яких цілей. Ці дозволи надаються через ліцензію про відкритий код.

Відкритий вихідний код є потужним, оскільки він зменшує кількість перешкод для впровадження та співпраці, дозволяючи людям швидко поширювати та вдосконалювати проекти. Наприклад, компанія, яка використовує програмне забезпечення з відкритим вихідним кодом, має можливість найняти когось, щоб удосконалити програмне забезпечення на замовлення, замість того, щоб покладатися виключно на рішення у постачальника програмних продуктів із закритим кодом.

Було обрано наступні три системи програмного забезпечення з відкритим кодом написані мовою Java:

- Фреймворк Spark (https://github.com/perwendel/spark).
- Фреймворк Javalin (https://github.com/javalin/javalin).
- Фреймворк Blade (https://github.com/lets-blade/blade).

Вибір обґрунтований тим, що Java це об’єктно-орієнтована строго типізована мова програмування, а отже вона чудово підходить під наші критерії. Обрані програмні системи з відкритим кодом є досить відомими, розробляються ентузіастами. Це найбільш наочний та реальний зразок для застосування нашої розробленої метрики.

У таблиці 4.2 показано основну інформацію про ці три системи. Варто зауважити, що кількість рядків вказує на всі рядки у файлах Java, включаючи коментарі та пробіли. Розміри файлів округлені до цілих чисел.

Таблиця 4.2 – Інформація про тестовані системи

	Spark	Javalin	Blade
Стрічок коду	5978	25421	34883
Класів	35	117	289
Методів	396	1754	2442
Атрибутів	194	1001	1812
Розмір (в Кб)	243	806	1135

Тепер визначимо показники для кожної програмної системи та спочатку відобразимо їх на графіках для більшої наочності. Це дозволить краще розуміти розподіл між значеннями параметрів розробленої метрики та результатом її застосування до обраних програмних систем. Графіки значень параметрів для обраних систем буде показано на рисунках 4.1, 4.2, 4.3.

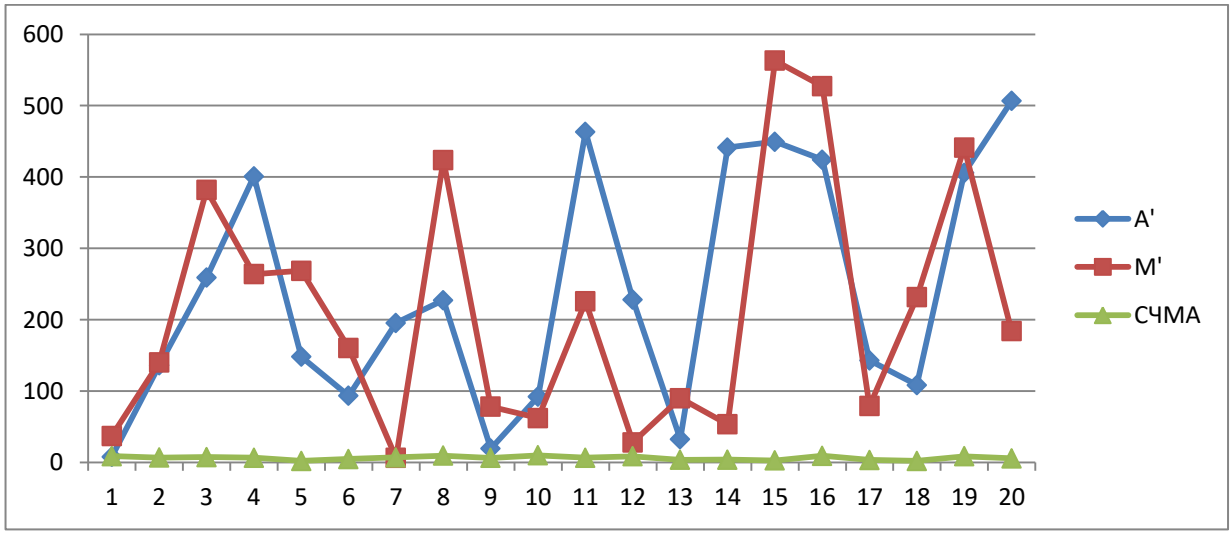


Рисунок 4.1 – Значення параметрів СЧМА для системи Spark

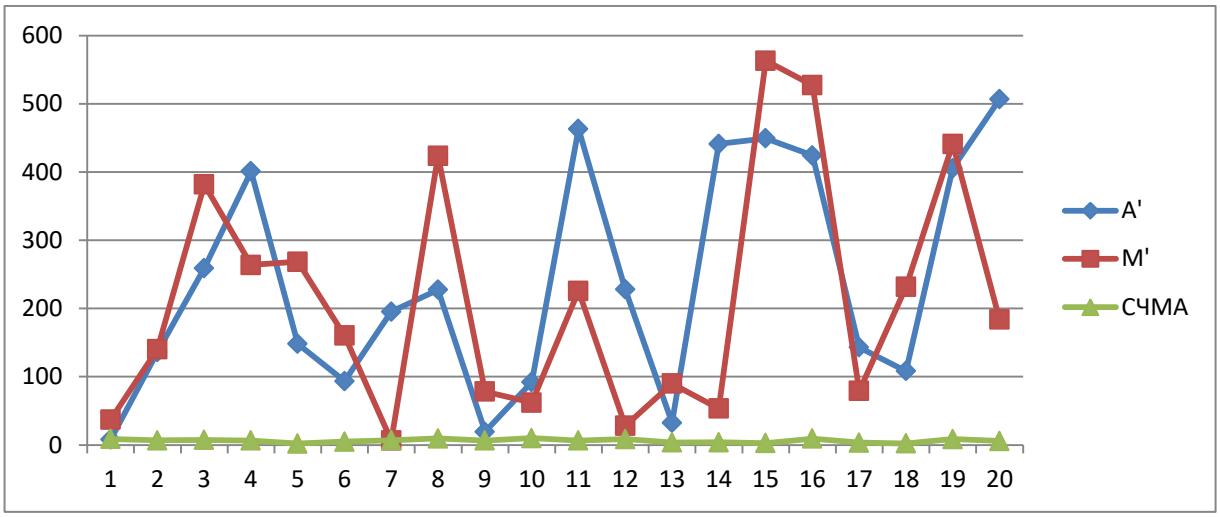


Рисунок 4.2 – Значення параметрів СЧМА для системи Javalin

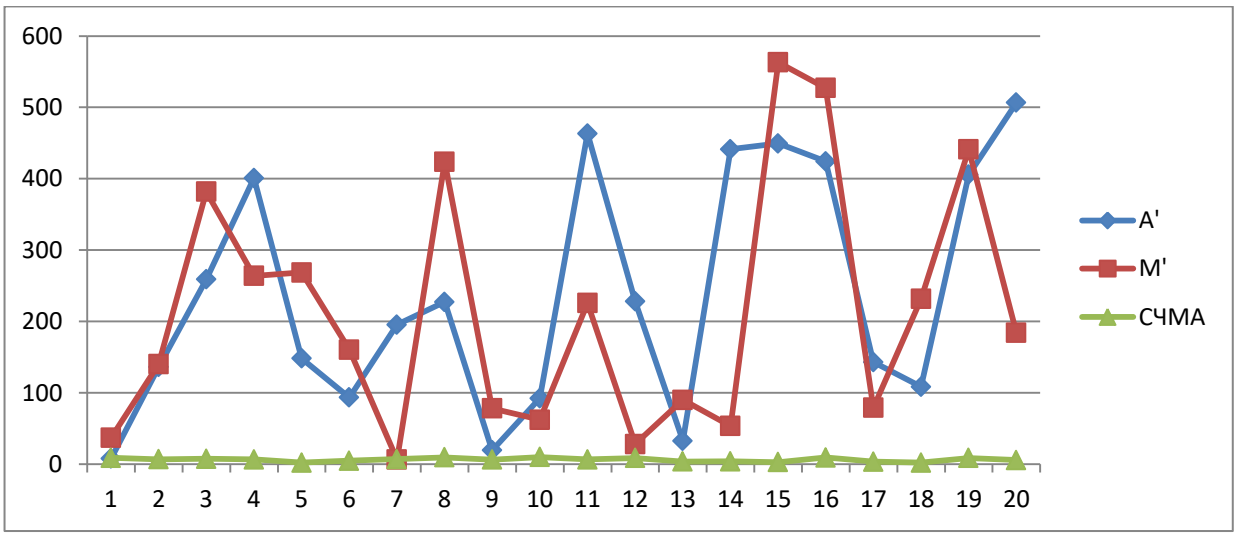


Рисунок 4.3 – Значення параметрів СЧМА для системи Blade

Тепер визначимо мінімум, максимум та середнє значення параметрів, необхідних для обчислення значення СЧМА, а також сам показник СЧМА для кожного з обраних проектів відкритого програмного забезпечення. Ця інформація знадобиться для статистичного аналізу та кореляції отриманих результатів, та в свою чергу для верифікації розробленої метрики. Ці показники демонструють ступінь складності обраних програмних систем, і з допомогою яких ми зможемо зробити висновки стосовно їх якості. Статистика цих параметрів буде відображена у таблицях 4.3, 4.4, 4.5. Результат кореляції між отриманими показниками буде надано у таблиці 4.6.

Таблиця 4.3 – Статистика параметрів СЧМА для системи Spark

	Середнє	**Мінімум**	**Максимум**
M’	11,88	4	35
A’	7	0	21
СЧМА	4,65	2	9

Таблиця 4.4 – Статистика параметрів СЧМА для системи Javalin

	Середнє	**Мінімум**	**Максимум**
M’	9,15	0	221
A’	19,04	1	315
СЧМА	3,7	2	10

Таблиця 4.5 – Статистика параметрів СЧМА для системи Blade

	Середнє	**Мінімум**	**Максимум**
M’	10,12	0	540
A’	8,77	1	567
СЧМА	3,1	2	10

Таблиця 4.6 – Коефіцієнти кореляції атрибутів СЧМА для трьох проектів

	M’	A’
Spark	0,884	0,873
Javalin	0,743	0,712
Blade	0,643	0,620

З рисунків 4.1, 4.2 та 4.3 видно, що в системі Spark 64% класів, значення СЧМА яких лежить між 2, 3 або 4, у порівнянні з системами Javalin і Blade, 72% яких знайдено в Javalin і 88% знайдено в Blade, у яких значення СЧМА лежить між 2, 3 або 4.

Характер розподілу значень метрики СЧМА у всіх трьох системах дещо відрізняється. У системах Blade і Javalin приблизно 81% класів містять СЧМА зі значенням 2 (у випадку Blade) і 45% класів містять СЧМА зі значенням 2 (у випадку Javalin).

Таким чином, може бути можливість покращення цих систем шляхом об'єднання класів у деякі інші класи в тому самому пакеті, без впливу на абстракцію та інкапсуляцію класів (що значно підвищить зрозумілість коду).

Можуть виникнути питання щодо вимог до класів, які мають СЧМА рівне 2, як у системах Blade, так і в Javalin, що в свою чергу в перспективі вимагає перепроектування програмної системи для підвищення її якості.

З таблиць 4.3, 4.4 та 4.5 видно, що середнє значення СЧМА (приблизно 4,143), яке перевищує середнє значення СЧМА (приблизно 3,7) для системи Javalin, а також середнє значення СЧМА (приблизно 3,1) для систем Blade. Попри те, кількість класів, заподіяних у Javalin та Blade більша, ніж у Spak. Це може свідчити про те, що природа розподілу методів і атрибутів між класами на ранній стадії проектування в системах Javalin та Blade є неправильною.

З таблиці 4.6 видно, що запропонована метрика складності СЧМА має дуже хорошу кореляцію з A' і M' у системі Spark. Це вказує на те, що розробник доклав більше зусиль і часу для розробки методів і атрибутів класів на ранніх етапах розробки програм, і пов'язаний із цим ризик низький завдяки правильному розподілу методів і атрибутів.

У той же час, показник СЧМА також має хорошу кореляцію з тим самим у системах Javalin і Blade, але все ж не такий хороший, як зі Spark. Це можливе через безладний розподіл методів і атрибутів між класами у цих двох системах.

Загальний результат з огляду на отримані показники такий, що всі три системи в основному складаються з класів, значення СЧМА яких рівне 2, 3 чи 4.

Це означає, що протестовані системи за своє природою є відносно простими. Єдиним зауваженням є те, що для систем Javalin та Blade є бажаним перегляд вимог до деяких класів, значення СЧМА яких більше 4.

4.2.2 Порівняльний аналіз СЧМА з уже існуючими метриками

У цьому розділі представлено аналіз СЧМА із наявними метриками складності, такими як CC, RFC і WMC (огляд яких проводився у першому розділі). Ці метрики доволі відомі та перевірені часом, а тому порівняльний аналіз розробленої метрики з ними дасть наочну картину того, наскільки розроблений нами метод достовірний і правдивий.

Порівняння буде проводитися шляхом знаходження коефіцієнтів кореляції. Ці коефіцієнти будуть обчислені для СЧМА, CC, RFC і WMC для обраних систем програмного забезпечення з відкритим кодом. Увагу буде зосереджено на тому, як запропонована метрика корелюється з уже існуючими метриками складності. Підсумкова статистика та коефіцієнти кореляції існуючих, а також запропонованих показників складності наведені у таблицях 4.7-4.12.

Коефіцієнти кореляції використовуються для вимірювання того, наскільки сильний зв’язок між двома змінними. Існує кілька типів коефіцієнта кореляції, але найпопулярнішим є коефіцієнт Пірсона.

Коефіцієнт Пірсона – це тип коефіцієнта кореляції, який представляє зв’язок між двома змінними, які вимірюються на одному інтервалі або шкалі співвідношення. Коефіцієнт Пірсона – це міра міцності зв’язку між двома безперервними змінними.

Коефіцієнт кореляції рангу Спірмена – непараметрична міра статистичної залежності між двома змінними; названий на честь Чарльза Спірмена. Він оцінює наскільки добре можна описати відношення між двома змінними за допомогою монотонної функції. Якщо немає повторних значень даних, то коефіцієнт Спірмена дорівнює 1 або −1, це відбувається коли кожна змінна є монотонною функцією від іншої змінної.

Таблиця 4.7 – Загальна статистика метрик складності для Spark

	Мінімальне	Максимальне	Середнє	Стандартне відхилення	Дисперсія
WMC	3	31	11,96	7,65	58,65
CC	5	87	35,68	24,65	544,12
RFC	4	140	17,65	47,21	845,63
СЧМА	2	9	4,54	2,044	4,024

Таблиця 4.8 – Загальна статистика метрик складності для Javalin

	Мінімальне	Максимальне	Середнє	Стандартне відхилення	Дисперсія
WMC	2	44	12,95	9,25	87,63
CC	4	112	35,68	144,65	544,12
RFC	2	543	244,65	125,124	124,23
СЧМА	2	10	5,12	3,21	5,242

Таблиця 4.9 – Загальна статистика метрик складності для Blade

	Мінімальне	Максимальне	Середнє	Стандартне відхилення	Дисперсія
WMC	5	124	12,42	5,25	87,65
CC	6	537	112,48	24,65	643,12
RFC	2	644	155,54	544,21	1242,63
СЧМА	2	10	5,78	3,38	6,12

Отже, в таблицях 4.7-4.9 наведено результати застосування метрик CC, WMC, RFC та СЧМА до обраних програмних систем з відкритим вихідним кодом. Це загальна статистика по показникам метрик, яка відображає ступінь складності обраних програмних систем. З їх допомогою можна теоретично порівняти показники та провести певний аналіз між цими метриками. Проте краще це

зробити практично за допомогою коефіцієнтів кореляції. Результати кореляцій наведені у таблицях нижче.

Таблиця 4.10 – Кореляція між результатами метрик CC, WMC, RFC та СЧМА до системи Spark (П – Пірсон, С – Спірмен)

	WMC		CC		RFC		СЧМА	
	П	С	П	С	П	С	П	С
WMC	1	1	0.911	0.887	0.907	0.872	0.885	0.875
CC	0.911	0.887	1	1	0.984	0.975	0.925	0.915
RFC	0.907	0.872	0.984	0.975	1	1	0.881	0.842
СЧМА	0.885	0.875	0.925	0.915	0.881	0.842	1	1

Таблиця 4.11 – Кореляція між результатами метрик CC, WMC, RFC та СЧМА до системи Javalin (П – Пірсон, С – Спірмен)

	WMC		CC		RFC		СЧМА	
	П	С	П	С	П	С	П	С
WMC	1	1	0.911	0.887	0.907	0.872	0.885	0.875
CC	0.911	0.887	1	1	0.984	0.975	0.925	0.915
RFC	0.907	0.872	0.984	0.975	1	1	0.881	0.842
СЧМА	0.885	0.875	0.925	0.915	0.881	0.842	1	1

Таблиця 4.12 – Кореляція між результатами метрик CC, WMC, RFC та СЧМА до системи Blade (П – Пірсон, С – Спірмен)

	WMC		CC		RFC		СЧМА	
	П	С	П	С	П	С	П	С
WMC	1	1	0.911	0.887	0.907	0.872	0.885	0.875
CC	0.911	0.887	1	1	0.984	0.975	0.925	0.915
RFC	0.907	0.872	0.984	0.975	1	1	0.881	0.842
СЧМА	0.885	0.875	0.925	0.915	0.881	0.842	1	1

З таблиці 4.7, таблиці 4.8 і таблиці 4.9 видно, що такі статистичні дані, як дисперсія, стандартне відхилення, середнє та максимальне, є найвищими для RFC у всіх трьох системах (Spark, Javalin і Blade). CC має друге за величиною статистичне значення для того самого показника. СЧМА має найнижчі статистичні значення для того ж показника. За значеннями RFC, CC і WMC трохи важко передбачити, чи стане система складнішою, помірно складною чи менш складною. Але зі статистичними значеннями СЧМА можна легко передбачити складність класів у системах на ранніх стадіях розробки програми, адже для його показників розроблено таблиці відповідностей та класифікація результатів.

У всіх системах середні значення для СЧМА є нижчими, ніж у CC, RFC і WMC. Певні цікаві спостереження також зроблено з таблиць 4.10, таблиць 4.11 і таблиць 4.12. З таблиці 4.10 видно, що СЧМА дуже добре корелює (як по Пірсону, так і по Спірмену) з RFC, CC і WMC. Особливо СЧМА добре корелюється з CC (P: 0,925 S: 0,915). Це через те, що в системі Spark значення для всіх показників дають правильний прогноз щодо складності класів. З таблиці 4.12 видно, що кореляція Пірсона хороша з CC, RFC і WMC, а коефіцієнт кореляції Спірмена дуже добре корелює СЧМА з CC, RFC і WMC.

З таблиці 4.12 видно, що в системі Blade кореляція Пірсона по СЧМА не дуже добре корелюється з показниками CC, RFC і WMC. Однією з можливих причин може бути те, що в системі Blade може бути брак вимог до класів.

4.3 Висновки

У даному розділі проведена апробація розробленої метрики, а саме її валідація та верифікація.

Валідація метрики проводилася за властивостями Вейюкера. В результаті ми отримали те, що пропонована метрика задовольняє більшості необхідних властивостей Вейюкера, а отже є дійсною і придатною до використання на реальних прикладах. Це важливий показник, адже він доводить що розроблена

метрика відповідає більшості стандартів, яких необхідно дотримуватися при розробці метрик об’єктно-орієнтованих програмних систем.

Верифікація розробленої метрики необхідна для доведення її достеменності та правдивості її результатів, ця перевірка проводилася у два етапи.

Першим етапом було її застосування на реальних програмних системах із відкритим сирцевим кодом. Це дозволило продемонструвати процес застосування метрики на реальному прикладі, а також отримати оцінку реального проекту, та інтерпретувати її відповідно до розроблених критеріїв.

Другим етапом був порівняльний аналіз розробленої метрики з уже відомими та протестованими. Порівняльний аналіз відбувався на результатах застосування обраних метрик до кількох реальних програмних систем із відкритим сирцевим кодом. Порівняння проводилося із застосуванням коефіцієнтів кореляції Пірсона та коефіцієнтом кореляції Спірмена. Отримані результати показали, що значення пропонованої метрики дуже добре корелюються зі значеннями уже відомих та перевірених метрик, а отже пропонована метрика достовірна.

ВИСНОВКИ

У результаті виконання дипломної роботи здійснено аналіз наявних методів оцінки об’єктно-орієнтованих програмних систем на етапі проектування, визначені основні проблеми цієї галузі. На основі проведеного аналізу прийнято рішення розробки власного методу та метрики оцінки складності об’єктно-орієнтованої програмної системи на етапі проектування.

Швидкий розвиток технологій програмного забезпечення та зростання вимог ринку призводять до створення дедалі складнішого програмного забезпечення. Через обмеження вартості розробки та часу виходу на ринок забезпечення якості програмного забезпечення є критично важливим і складним завданням. Як наслідок, багато компаній зіткнулися з проблемами якості програмного забезпечення. Хоча якість програмного забезпечення по-різному сприймається та визначається багатьма фахівцями в індустрії програмного забезпечення, загальноприйнятим є те, що однією з важливих характеристик високоякісного програмного забезпечення є якомога менше дефектів.

У першому розділі проведено аналіз наявних методологій вирішення проблем у цій галузі. Проведено дослідження основних методів оцінки програмних систем, а також метрик для отримання показників цих оцінок. В результаті аналізу було підсумовано основні проблеми та поставлено відповідні задачі, які потрібно вирішити.

У сучасному сценарії розробки програмного забезпечення об’єктно-орієнтований підхід є провідним. Цей підхід підвищує продуктивність програмного забезпечення, дає багаторазове використання коду та гнучкість програмних систем. Об'єктно-орієнтовані системи набувають популярності як ефективні програмні системи день у день, тому що об'єктно-орієнтовані методи зменшують розмір системи та кількості логічних конструкцій у ній. Об'єктно-орієнтований програмне забезпечення зазвичай містить велику кількість атрибутів, які містять в собі повний опис програмних компонентів та елементів.

У другому розділі розглянуто методології для валідації метрик, проведено їх порівняльний аналіз та вибрано найкращу, яка буде застосована для валідації розроблюваної метрики. Це дозволить її перевірити на дійсність, тобто підтвердити факт що розроблена метрика відповідає стандартам. Також у цьому розділі наведено припущення, які допоможуть у розробці нової метрики.

Показники програмного забезпечення мають важливе значення для розробки програмного забезпечення та для вимірювання складності та якості програмного забезпечення, оцінки вартості та проектних зусиль, і це лише деякі з них. Традиційні показники, такі як функціональна точка, наука про програмне забезпечення та цикломатична складність, добре використовуються в процедурній парадигмі. Однак вони не завжди застосовні до аспектів об’єктно-орієнтованої парадигми.

Третій розділ присвячено опису розробленої метрики оцінки програмних систем на етапі проектування. У ньому детально розписано її складові, компоненти. Викладено ідеї, які закладені у пропоновану метрику. Також представлено критерії для аналізу результатів розробленої метрики, за допомогою яких можна легко інтерпретувати отриманий показник метрики, та швидко провести паралелі між ним та деякими атрибутами якості програмних систем.

Щоб ефективно верифікувати запропоновану метрику складності об’єктно-орієнтованих програмних систем та її зв’язок із факторами якості, було обрано провести її тестування на програмному забезпеченні з відкритим вихідним кодом. Для експериментів вибрано програмне забезпечення середнього розміру.

Коли проект має відкритий код, це означає, що будь-хто може вільно використовувати, вивчати, змінювати та поширювати цей проект для будь-яких цілей. Ці дозволи надаються через ліцензію про відкритий код.

Після вдосконалення методу та метрики була проведена їх практична апробація. Цьому присвячено четвертий розділ цієї роботи. Практична апробація складалася з валідації та верифікації розробленої метрики. Валідація відбувалася за раніше обраною методологією по властивостях Вейюкера. Результат валідації показав, що розроблена метрика дійсна та відповідає більшості критеріїв.

Верифікація розробленої метрики проводилася у два етапи: застосування її на реальних програмних системах з відкритим кодом, а потім порівняльний аналіз отриманих результатів з уже відомими та перевіреними метриками. Цей аналіз показав, що кореляційні коефіцієнти між результатами розробленої метрики та уже відомими досить хороший. Це означає що розроблена метрика достовірна, і її можна використовувати для оцінки реальних проектів, адже вона дає правдиві результати та є доволі ефективною.

Отже, в результаті виконання дипломної роботи було удосконалено метод та метрику та доведену її придатність до використання на реальних проектах. Результати цього дослідження мають наукову новизну та практичну цінність.

ПЕРЕЛІК ПОСИЛАНЬ

1. A Validation of Object Oriented Design Metrics as Quality Indicators / V. R. Basili, L. C. Briand, and W.L. Melo, // IEEE Transactions on Software Engineering, Vol. 22, No. 10, 1996. – pp. 751-761.

2. OORASS: seamless support for the creation and maintenance of object oriented systems / T. Reenskaug, E. Andersen A. Berre, A. Hurlen, A. Landmark, O. Lehne, E. Nordhagen, E. Ness-Ulseth, G. Oftedal, A. Skaar, and P. Stenslet // Journal of Object Oriented Programming, Vol. 5, No. 6, 1992. – pp. 7-41.

3. Managerial Use of Metrics for Object-Oriented Software: An Exploratory Analysis / S. R. Chidamber, D. P. Darcy, and C. F. Kemerer // IEEE Transactions on Software Engineering, Vol. 24, No. 8, 1998. – pp. 629-637.

4. A Metrics Suite for Object Oriented Design / S. R. Chidamber and C. F. Kemerer // IEEE Transactions on Software Engineering, Vol. 20, No. 6, 1994. – pp. 476–493.

5. Software change impacts–an evolving perspective / Bohner.S.A. // Proceedings of the International Conference on Software maintenance, 2002. – pp. 263 – 272.

6. Empirical of CK Metrics for Object-Oriented Design Complexity: Implication for Software Defects / R. Subramanya and M. S. Krishnan // IEEE Transaction on Software Engineering, Vol. 29, 2003. – pp. 297-310.

7. Towards a Metrics Suite for Object-Oriented Design / Chidamber S. R., Kemerer C. F. // OOPSLA'91 Conference Proceedings, SIGPLANNotices, Vol. 26, 1991. – pp. 197-211.

8. Workshop Report - Processes and Metrics for Object-Oriented Sotware Development / Bilow S.C, Lea D. // OOPSLA'93 Addendum to the Proceedings, OOPSMessenger, Vol. 5, 1994. – pp. 95-98.

9. Design & Use of Software Architectures: Adopting and evolving a product-line approach / Bosch, J. – Pearson Education, 2000. – 354 p.

10. Reliability prediction for component-based software architectures / Reusner, R., Schmidt, H.W., and Poernomo, I. H. // Journal of Systems and Software, 2003. – pp. 76-252.

11. Pattern-Oriented Software Architecture: A System of Patterns / Buschmann, F., Meunier, R., Rohnert, H., Sommerland P. and Stal, M. – Wiley. 1996. – 490 p.

12. The Architecture Tradeoff Analysis Method / Kazman, R., Klein, M., Barbacci, M., Longstaff, T., Lipson, H., and Carriere, S. J. // Proc. 4th IEEE International Conference on Engineering of Complex Computer Systems, 1998. – pp. 68-78.

13. Architecture-Level Modifiability Analysis / Bengtsson, PO. // Blekinge Institute of Technology, Dissertation Series No 2002-2, 2002.

14. Metrics to Assess the Likelihood of Project Success Based on Architecture Reviews / Avritzer, A. and Weyuker E. J. // Empirical Software Engineering, 1999. – pp. 199-215.

15. A framework for classifying and comparing software architecture evaluation methods / Babar, M. A., Zhu, L., and Jeffery, R. // Proc. Australian Software Engineering Conference, 2004. – pp. 309-318.

16. Enabling iterative software architecture derivation using early non-functional property evaluation / Barber, K. S., Graser, T., and Holt, J. // Proc. 17th IEEE International Conference on Automated Software Engineering, 2002. – pp. 23-27.

17. Software Architecture in Practice / Bass, L., Clements, P., and Kazman, R. – Addison-Wesley, 2003. – 464 p.

18. Encapsulating Legacy Software for Reuse in Client/Server Sstem / H. Sneed // In proceedings of WCRE-96 – Monterey, IEEE press, 1996.

19. Object-Oriented Software Engineering / I. Jacobson – Addison-Wesley, 1992 – 552 p.

20. Requirement traceability and impact analysi methodology t evaluate software requirements changes / M.Z.Kurian and S Manjunath // National Conference on Trend in Advanced Computing, a DMCE, Airoli – Navi Mumbai, 2007. – pp. 28-29.

21. Change Prediction in Object-Oriented Software Systems: Probabilistic Approach / Ali R. Sharafat and Ladan Tahvildari // Journal of Software, Vol. 3, No. 5, 2008. – pp.10-38.

22. Object-Oriented Modeling and Design / Rumbaugh J., Blaha M., Premerlani W., Eddy F. and Lorensen W. // Prentice Hall, Englewood Cliffs, New Jersey, 1991

23. An Introduction to Trellis/Owl / Schaffert C., et al. // Meyrowitz N. (ed.) Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications – OOPSLA '86, Portland, Oregon, 1986. – pp. 9-16.

24. Designin Object Oriented Software / Wirfs-Brock R., Wilkerson B. and Wiener L. // Prentice Hall, Englewood Cliffs, New Jersey, 1990.

25. Program Development by Stepwise Refinement / Wirth N. // Communications of the ACM, 14(4), 1971 – pp. 221-227.

26. Metrics and Models in Software Quality Engineering, Second Edition / Kan S.H. // Addison Wesley, 2002. – 560 p.

27. Another Metric Suite for Object Oriented Programming / W. Li The // Journal of Systems and Software, Vol. 44, No. 2, 1998. – pp. 155-162.

28. A Metrics Suite for Object-Oriented Design / Chidamber, Shyam and Kemerer, Chris // IEEE Transactions on Software Engineering, 1994 – pp. 476-492.

29. Software Metrics for Object-oriented Designs / Hudli, R., Hoskins, C., Hudli, A., // IEEE, 1994.

30. Object-Oriented Software Metrics / Lorenz, Mark and Kidd, Jeff // Prentice Hall Publishing, 1994.

31. Evaluating software complexity measure / E.J. Weyuker, // IEEE Transaction on Software Engineering, 14(9): 1357-1365, September 1988.

32. Properties of Software Measures / H. Zuse // Software Quality Journal, vol. 1, 1992 – pp. 225- 260.

33. On Weyuker's Axioms for Software Complexity Measures / J.C.Cherniavsky and C.H.Smith, // IEEE Transactions on Software Engineering, vol. 17, 1991 – pp. 636-638.

34. Software Metrics. A rigorous approach / N.E. Fenton // New York: Chapman and Hall, 1991.

35. Evaluating the Impact of OO Design on Software Quality / B. Abreu and W. Melo // presented at Third International Software Metrics Symposium, Berlin, 1996.

36. Design and Development of a Procedure to Test the Effectiveness of Object-Oriented Design / Srinivasan, K.P., and Devi, T. // International Journal of Engineering Research and Industrial Applications, Vol.2, No.6, 2009 – pp. 15-25.

37. Towards a Framework for Software Measurement Validation / Kitchenham, B., Pfleeger, S.L., and Fenton, N. // IEEE Transactions on Software Engineering, Vol. 21, No.12, 1995 – pp. 929-943.

38. Property-Based Software Engineering Measurement / Briand, L.C., Morasca, S., Basili, V.R. // IEEE Transactions on Software Engineering, 1996 – pp. 68-85.

39. Applying and interpreting object oriented metrics / Rosenberg L. H. // Software Technology Conference, Utah, April 1998.

ДОДАТОК А

(обов’язковий)

КОПІЇ НАУКОВИХ ПУБЛІКАЦІЙ

СТАТТЮ ПОДАНО ДО ДРУКУ
УДК 004.9

ДЗЮРБАН Е. С.
Хмельницький національний університет
e-mail: eduard.dziurban@gmail.com
ЯШИНА О. М.
Хмельницький національний університет
e-mail:oksana.yashyna@ukr.net

МЕТОД ОЦІНКИ ОБ'ЄКТНО-ОРІЄНТОВАНИХ ПРОГРАМНИХ СИСТЕМ НА ОСНОВІ АНАЛІЗУ ЗМІНИ ВИМОГ ДО ПРОГРАМНОЇ СИСТЕМИ

Добре відомий факт, що технічне обслуговування програмного забезпечення відіграє важливу роль і набуває важливого значення в життєвому циклі програмного забезпечення. Оскільки об'єктно-орієнтоване програмування давно вже стало стандартом, дуже важливо розуміти проблеми підтримки об'єктно-орієнтованих програмних систем, та спосіб виявлення їх потенційних місць виникнення. Ця стаття спрямована на оцінку об'єктно-орієнтованих систем за допомогою аналізу зміни вимог до програмної системи. Основні проблеми порушені в статті це покращення алгоритму аналізу впливу зміни не функціональних вимог до програмної системи на функціональні та їх наслідування.

Попит на ефективне програмне забезпечення зростає з кожним днем, і впровадження об'єктно-орієнтованого проектування програмних систем здатне задовольнити цей попит, оскільки це, мабуть, найпотужніший механізм розробки ефективних програмних систем. Це може допомогти не тільки зменшити витрати, але й розробити високоякісне системне програмне забезпечення. Розробникам програмного забезпечення потрібні відповідні показники для розробки ефективної системи програмного забезпечення. Ця стаття спрямована на дослідження методів оцінки об'єктно-орієнтованої програмної системи за допомогою аналізу впливу змін функціональних вимог до програмного забезпечення за допомогою не функціональних вимог.

Незважаючи на те, що об'єктно-орієнтований підхід має багато переваг, а також він є найпоширенішим зараз та буде таким у майбутньому, його практичність буде доведена лише тоді, коли аспекти управління процесом розробки програмного забезпечення за допомогою цієї методології буде ретельно розглянуто. Саме тут показники програмного забезпечення відіграють важливу роль, забезпечуючи краще планування, зменшення ризиків, раннє виявлення потенційних проблем, оцінку якості та ефективності. У цій статті пропонується набір показників, які найкраще підходять для оцінки використання основних концепцій об'єктно-орієнтованої парадигми, таких як наслідування, інкапсуляція, поліморфізм та повторне використання коду, які однозначно відповідають за підвищення якості програмного забезпечення та продуктивності розробки.

Ключові слова: об'єктно-орієнтована архітектура, аналіз, оцінка, програмна система, зміна алгоритму аналізу впливу, наслідування функціональності.

DZIURBAN E. S.
Khmelnytsky national university, Ukraine
YASHYNA O. M.
Khmelnytsky national university, Ukraine

METHOD OF EVALUATIONG OF OBJECT-ORIENTED SOFTWARE SYSTEMS BASED ON THE ANALYSIS OF CHANGES IN THE SOFTWARE SYSTEM REQUIREMENTS

It is a well-known fact that software maintenance plays an important role and becomes important in the software life cycle. Since object-oriented programming has long become the standard, it is very important to understand the problems of maintaining object-oriented software systems, and how to avoid them by identifying potential gaps in the software system as early as the design analysis. This article is aimed at evaluating object-oriented systems using the analysis of changes in the requirements for the software system. The main problems raised in the article are the change of the algorithm for analyzing the impact of changing non-functional requirements on functional ones and their inheritance.

The demand for efficient software is increasing day by day, and the adoption of object-oriented design of software systems is able to satisfy this demand, as it is perhaps the most powerful mechanism for developing efficient software systems. This can not only help in reducing the cost but also helps in developing high quality system software. Software developers need appropriate metrics to develop an effective software system. This practice is aimed at researching methods for evaluating an object-oriented software system using software impact analysis based on tracking requirements to changes in functional requirements using non- functional requirements.

Although there are many advantages to the object-oriented approach, and the fact that this approach is the most widespread now and will be in the future, it will be truly recognized, proven and practical only when the management aspects of the software development process using of this methodology will be carefully considered. This is where software metrics play an important role, enabling better planning, evaluating improvements, reducing unpredictability, early detection of potential problems, and evaluating performance. This paper proposes a set of metrics best suited to evaluate the use of core concepts of the object-oriented paradigm, such as inheritance, encapsulation, polymorphism, and a strong emphasis on code reuse, which are uniquely responsible for increasing software quality and development productivity.

Keywords: object-oriented architecture, analysis, evaluation, software system, change of impact analysis algorithm, imitation of functionality.

Постановка проблеми

Існує декілька стандартів відстеження змін до вимог програмної системи. Це, зокрема ISO15504 та CMMI. Також за останні десятиліття було розроблено декілька методик для забезпечення вимог до власне відстеження цих змін. Більшість традиційних методів схожі на два найбільш відомих, це Trace-based Impact Analysis Methodology (TIAM), та Work Product Model (WoRM). Згадані методології дають передбачуване значення для знаходження класів зі схожими змінами. Методологія TIAM призначена більше для планування, ніж для прямого впровадження змін. TIAM потенційно може бути використана для оцінки ризиків щодо нестабільності вимог. У випадку проектних змін існують певні наслідки застосування об'єктно-орієнтованого підходу. На практиці було виявлено, що архітектори-початківці мають проблеми зі створенням класів та кругообігом між декларативними та процедурними аспектами рішення. Відповідно, має місце впровадження таких патернів або методів відстеження змін до вимог як «вимога-компонент», які можуть бути застосовані як до традиційних, так і до найбільш сучасних процесів розробки програмного забезпечення. Такий підхід дозволив досягти відповідності структури вихідного коду патернам та методам, які забезпечують легкість відслідковування змін у системи відповідно до нових вимог. У життєвому циклі програмне забезпечення зазнає змін на всіх етапах. Програмний продукт є успішним, якщо зміни в програмному забезпеченні ідентифікуються або здійснюється їх управління протягом усіх фаз життєвого циклу програмного забезпечення.

Для отримання програмного продукту повинна бути чітко встановлена межа, і вона має ставати більш точнішою задля того щоб програмний продукт пройшов всі етапи життєвого циклу. Обслуговування програмного забезпечення споживає приблизно сорок відсотків витрат на програмне забезпечення, оскільки є нетривіальною фазою в життєвому циклі розробки програмного забезпечення. А оскільки це нетривіальна фаза, то відстеження зв'язку між кодом та елементами в програмному забезпеченні може полегшити виконання багатьох завдань. Розуміння програми, супровід, вимоги трасування, аналіз впливу та повторне використання існуючого програмного забезпечення, це все важливі елементи про які не слід забувати.

Аналіз останніх джерел

У житті програмного продукту є кілька етапів. Модель водоспаду, як описано у [1], має п’ять основних фаз. Це аналіз вимог і специфікації, кодування та тестування модулів, інтеграційне тестування, системне тестування та обслуговування. Це дослідження стосується лише аспекту заключної фази, обслуговування. Етап обслуговування є найдовшою фазою життєвого циклу. Обслуговувати програмне забезпечення з часом стає все складніше, оскільки система прогресує і розвивається. У дослідженні [2] показали алгоритми для обчислити транзитивного замикання кожного з потенційно зачеплених класів і методів. За допомогою них можна значно покращити інформацію, яку надають алгоритми розпізнавання шаблони проектування, ефекти змін типу даних, а

також ефекти додавання та видалення класів. У роботі [3] представлено інтегроване середовище для обслуговування програм C++, яке описує три нові графи залежностей, що характерні для об'єктно-орієнтованих програмних систем: повідомлення, клас і оголошення залежності у моделі під назвою C++ DG. Описані залежності, зокрема щодо ефекту пульсації та регресійного тестування. Застосування виявлених залежностей і розподіл програми призводить до рекурсивного аналізу ефекту хвиль, спричинених модифікацією коду. У міру розташування цих ефектів їх місця можна «позначити» для тестування або повторного виконання на етапі тестування.

У дослідженні [4] пояснили чотири алгоритми, які вимірюють вплив запропонованих змін на об’єктно-орієнтовані системи. Ефект пульсації розраховується шляхом застосування таких алгоритмів як:

1. Обчислення впливу змін всередині класу.
2. Розрахунок впливу змін серед клієнтів.
3. Розрахунок впливу серед підкласів.
4. Вимірювання загального ефекту, керуючи алгоритмами 1, 2 та 3.

Автор також представив деталі того, як різні типи змін впливають на систему. Зміни широко класифікуються, а потім уточнюються до більш детальної інформації, наприклад додавання учасника або зміна атрибута. Відповідно до роботи [5] візуальний аналіз впливу покращив розпізнавання додаткових залежностей. У дослідженні [6] автор вказав, що практика розробки програмного забезпечення розвивається відповідно до вимог розподілених програми на різнорідних платформах; зміна програмного забезпечення все більше впливає на проміжне програмне забезпеченням і компоненти. Відносини залежності взаємодії тепер вказують на більше відповідні наслідки зміни програмного забезпечення та обов’язково приводять до обов’язкового їх аналізу. Сучасні моделі аналізу впливу змін програмного забезпечення не враховують ці тенденції належним чином.

У статті [7] автор пояснив підхід, який застосовується до написання програмного забезпечення на об’єктно-орієнтованій мові для відстеження об’єктно-орієнтованого коду та реалізації функціональних вимог. Тут розглядається проблема встановлення зв’язків та їх аналізу між вільною текстовою документацією, пов’язаною з циклом розробки та обслуговування програмної системи, а також її кодом.

Моделі відстеження змін до вимог

Відстеження вимоги означає здатність описувати та стежити за життям вимоги як у прямому, так і у зворотному напрямку. Це здатність відстежувати вимоги до компонентів проекту або їх впровадження. Зворотне відстеження – це можливість відстежити вимогу до її джерела, тобто до особи, установи або ж закону чи аргументу тощо. Відслідковування вимог стосується також зв’язків між ними, або ж їх зв’язку з атрибутами програмної системи [8].

Аналіз впливу на основі відстеження змін до вимог

Аналіз впливу на основі відстеження змін вимог до програмної системи – це нефункціональне та неформальне відстеження. Функціональне відстежування пов’язане з добре встановленим відображенням між типами моделі об’єктів і типами відображення. Це дозволяє аналізувати моделі проектування, моделі процесів, організаційні моделі. Нефункціональне відстеження пов’язане з відстеженням нефункціональних аспектів розробки програмного забезпечення. Зазвичай, вони пов’язані з аспектами якості та є результатом зв’язку з нематеріальними концепціями. Нефункціональне відстеження класифікується за чотирма категоріями, такими як: причина, контекст, рішення та технічний аспект.

У цій статті оглянута методологія аналізу впливу на основі відстеження змін вимог до програмної системи, яка призначена для розширення можливостей відстеження потенційних проблем а також проведення об’єктно-орієнтованого аналізу та аналізу деяких аспектів архітектури. Запропоновано розглянути кілька етапів, а саме:

Фаза перша включає в себе такі пункти

A. Перевірка нових вимог від будь-якої із зацікавлених сторін.
B. Класифікація вимог, функціональних чи нефункціональних.

C. Матриця відстеження може допомогти відстежити вимогу.
D. Огляд вимог.
E. Оцінка вимог.
F. Вимоги до документації.
G. Приймальні випробування.

Після проходження першої фази, відповідно до визначених показників буде розділено аналіз по них у другій фазі. Це такі показники, як:

A. Стабільність: нестабільні вимоги.
B. Повнота: неповні вимоги.
C. Чіткість: незрозумілі вимоги.
D. Дійсність: Недійсні вимоги.
E. Здійсненність: нездійсненні вимоги.
F. Прецедент: безпрецедентні вимоги.

Стабільність

Цей показник вказує на вразливість системи до змін. Було помічено, що підтримка програмного забезпечення погіршується, оскільки до нього вносяться зміни, що збільшує складність програмного забезпечення. В такому випадку стабільність системи розраховується за формулою:

$$S(\#NORS + \#NOCNR + \#NOCUR + \#NOCDR) / (\#NORS)$$

Де S – стійкість, NORS – кількість вхідних вимог систему, NOCNR – кількість сукупної кількості вимог, NOCUR – сукупна кількість запитів, оновлених у системі, NOCDR – сукупна кількість запитів, видалених із системи.

Повнота

Цей показник визначає повноту вимоги та обчислюється за формулою:

$$CMP = NARS - NIR$$

CMP – ступінь завершеності системи, NARS – кількість фактичних/початкових вимог до системи, NIR – кількість незавершених вимог у системі.

Ясність

Цей показник відображає чіткість вимог до системи

$$CL = NARS - NIR - UCLR$$

CL – ступінь чіткості системи, NARS – кількість фактичних/початкових вимог до системи, NIR – кількість незавершених вимог у системі, UCLR – кількість незрозумілих вимог.

Здійсненність

Цей показник відображає ступінь здійсненності системи, тобто фактичне число її реалізованості.

$$FR = IFR - UCLR$$

FR – показник ступеня всіх техніко-економічних вимог системи, IFR – кількість нездійсненних вимог у системі, UCLR – кількість незрозумілих вимог.

Прецедент

Це показник, який відображає наскільки система завершена на даний момент.

$$PR = CMP + CL + FR$$

PR – показних попередніх вимог до системи, CMP – завершеність системи, CL – чіткість системи, FR – здійсненність системи.

Результати

У цьому розділі буде проведено ідентифікацію, візуалізація та аналіз відстеження змін до вимог на об’єктно-орієнтованій програмній системі. Тут як обов’язкову вимогу було прийнято практичне дослідження системи бронювання авіаквитків. Виходячи з рівня вимоги, необхідно розділити вимоги на певні рівні, після чого робити подальший аналіз. В результаті було виділено такі рівні:

1. Потреби замовника.
2. Функціонал.
3. Варіанти використання.
4. Додаткові вимоги.
5. Варіанти тестування.

Вимоги на верхньому рівні (потреби замовника) збираються за допомогою різних методів виявлення вимог, такі як опитування, дзвінки, зустрічі тощо.

Бізнес-аналітик виводить другий рівень (функціонал) із запитів зацікавлених сторін, корегуючи вимоги переводить їх із проблемної області в область вирішення. Функціонал повинен мати всі атрибути адекватних вимог до програмної системи.

Третій рівень містить варіанти використання програмної системи, які виділяються на першому рівні аналізу системи.

Додаткові вимоги охоплюють переважно нефункціональні вимоги. Вони також можуть фіксувати деякі загальні функціональні вимоги, не пов’язані з жодними конкретними випадками використання.

Варіанти тестування створені для перевірки вимог третього рівня.

Як приклад, можна привести алгоритми для системи бронювання авіаквитків.

Крок 1: Запуск алгоритму.

Крок 2: Введення URL-адреси.

Крок 3: Введення даних про рейси для пошуку рейсів.

Крок 4. Вибір рейсу.

Крок 5. Системне відображення зворотних рейсів.

Крок 6: Системне відображення деталей рейсів

Крок 7. Підтвердження рейсу.

Крок 8: Реєстрація нового користувача.

Крок 9: Вхід.

Крок 10: Надання інформацію про пасажира.

Крок 11: Відображення вільних місць.

Крок 12: Вибір місця.

Крок 13: Введення платіжної інформації.

Крок 14: Введення номеру підтвердження.

Крок 15: Звершення алгоритму.

На рисунку 1 показано діаграму варіанту використання для користувача.

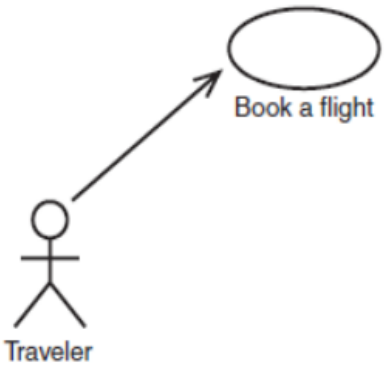


Рисунок 1 - Користувач та варіант використання

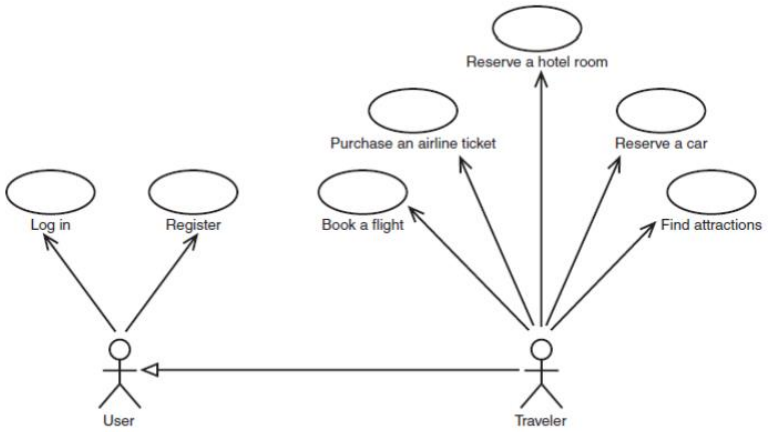


Рисунок 2 - Варіант використання для нового на зареєстрованого користувача

Основною метою структурування моделей є усунення будь-якої надмірності, що полегшить розуміння та підтримку визначених варіантів використання. Для цього визначається структуру відстеження змін до вимог в програмній системі, що зображено на відповідній діаграмі (рисунок 3).

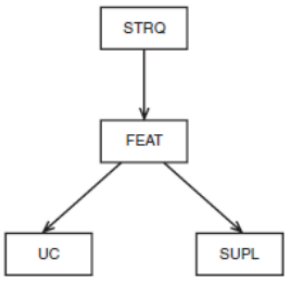


Рисунок 3 - Структура відстеження змін до вимог для прикладу «забронювати рейс»

Потреби замовника (STRQ) будуть відстежуватися до функціоналу (FEAT), визначених у документі вимог та додаткових вимог. Між STRQ і FEAT може існувати зв'язок «багато-до-багатьох», але зазвичай це один запит зацікавленої сторони до багатьох функцій. Кожен схвалений запит має стосуватися принаймні однієї функції або додаткової вимоги.

Вимоги до функціоналу (FEAT) будуть відстежуватися або у конкретному випадку використання, або в додатковій вимозі. Кожен схвалений функціонал має відстежувати принаймні один варіант використання або додаткову вимогу. Між функціями, варіантами використання та додатковими вимогами можуть існувати зв'язки «багато-до-багатьох».

Вимоги до варіантів використання (UC), визначені в специфікаціях варіантів використання, будуть відстежуватися до функціоналу.

Додаткові вимоги (SUPL) будуть відстежуватися до функціоналу.

Висновки

Отже, у даній статті було здійснено об'єктно-орієнтований аналіз для прикладу «забронювати авіаквиток» і отримано діаграми варіантів використання. Відповідно до визначених варіантів використання було отримано алгоритм та залежності між основними фазами внесення змін до програмної системи. Запропонований аналіз сильно залежить від добре визначених вимог до програмного забезпечення та не функціональних вимог, які можна відстежити. Саме завдяки цьому на основі аналізу змін у вимогах можна визначити вплив на діаграму класів до атрибутів даного випадку, а також визначити які тестові варіанти слід змінити, щоб врахувати більше потенційних

помилок. Це дозволить не нести збитки у майбутньому, що покращить загальну надійність системи та збільшить шанс на успішну її реалізацію.

Література

1. Carlo Ghezzi, Mehdi Jazayeri, Dino Mandrioli, Fundamentals of Software Engineering, Prentice Hall Publishing, 2011.

2. Chandra Shrivastava, D. L. Carver, "Using Low-Level Software Architecture for Software Maintenance of Object-Oriented Systems", Proceedings of the 1995 Software Engineering Forum, Boca Raton, FL, November, pp. 31-40, 2005.

3. Chen. X., Tsai. W., Hunag H., Poonawala M., Rayadurgam. S., Wang. Y., Omega-an Integrated Environment for C++ Program Maintenance, Proceedings of the International conference on software Maintenance, pp.114-123, 1996.

4. Li L., Offutt A. J., Algorithmic Analysis of the Impact of Changes to Object-oriented Software, Proceedings of the International Conference on Software Maintenance, pp. 171-184, 1996.

5. Hutchins, M., Gallagher, K., Improving Visual Impact Analysis, Proceedings of the International Conference on Software Maintenance, pp.294-301, 2016.

6. Bohner S. A., Software change impacts–an evolving perspective, Proceedings of the International Conference on Software maintenance, pp 263 – 272, 2020.

7. Peter Zielczynski, IBM, Requirements Manangement Using IBM Rational Requisite Pro, 2013.

8. Sarah Maadawy and Akram Salah, Measuring Change Complexity from Requirements: A proposed methodology, IMACST Volume 3, 2012.

ДОДАТОК В

(обов’язковий)

ПРЕЗЕНТАЦІЙНІ МАТЕРІАЛИ

Методи оцінювання об’єктно-орієнтованих програмних систем на етапі проектування

Виконав:
Студент II курсу групи ІПЗм-21-1
Дзюрбан Едуард
Керівник:
Кандидат технічних наук, доцент
Яшина Оксана Миколаївна

Актуальність теми

- Одна з проблем аналізу систем — оцінювання їх складності. Складність системи є якісною характеристикою. Зменшення складності ПС дає змогу знизити трудомісткість проектування, розробки, тестування та супроводження, забезпечує простоту і надійність виробленої ПС. У даний час питанням управління якістю програмних систем приділяється підвищена увага, пов’язано це з ростом застосувань програмних систем у різних сферах діяльності людини. Управління якістю включає в себе планування якості, забезпечення якості і контроль якості. Таким чином, однією зі складових розробки ПС із запланованим рівнем якості є контроль, який базується на застосуванні метрик якості для вимірювання основних показників в процесі розробки ПС.

Об’єкт, предмет і мета дослідження

- Об’єкт дослідження – процес оцінювання об’єктно-орієнтованих програмних систем.
- Предмет дослідження – методи та метрики оцінювання об’єктно-орієнтованих програмних систем на етапі проектування.
- Мета дослідження – удосконалення методу та метрики оцінювання об’єктно-орієнтованих програмних систем на етапі проектування.

Завдання дослідження

- здійснити аналіз предметної області;
- дослідити існуючі методи вирішення цих проблем;
- проаналізувати використовувані метрики в наявних методах із визначенням їх переваг та недоліків;
- на основі аналізу удосконалити метод та метрику оцінки складності програмної системи на етапі проектування;
- провести тестування та апробацію отриманої метрики, дослідити її ефективність на реальних прикладах;
- провести аналіз отриманих результатів та сформулювати рекомендації для подальшого удосконалення розробленого рішення.

Аналіз наявних проблем та шляхи їх вирішення

Виділені проблеми в галузі оцінювання складності об'єктно-орієнтованих програмних систем:

- складність та комплексність існуючих підходів та метрик.
- проблеми при інтерпретації отриманих результатів у чітко детерміновані границі показників.

Відповідно до виявлених проблем, з'являється потреба у розробці простішого підходу до оцінки складності об'єктно-орієнтованих програмних систем, результати якого можна було б легко інтерпретувати та чітко визначити границі отриманих показників.

Розроблене рішення

- Удосконалено метрику СЧМА оцінки складності об'єктно-орієнтованої програмної системи, шляхом зміни кількості атрибутів та методів у класі.
- Розроблена метрика використовується для оцінки складності класів програмної системи ще на етапі проектування.

Визначення метрики СЧМА

- Формула: $\text{СЧМА} = M' + A'$
- M' – відображає значення, яке основане на діапазоні кількості наявних методів у класі (приватних, публічних та захищених).
- A' – відображає значення, яке основане на діапазоні кількості наявних атрибутів у класі (приватних, публічних та захищених).
- Співвідношення діапазонів до значень наведено у таблицях:

Кількість методів у класі	Відносне значення M'
0 чи 1-5	1
6-10	2
11-15	3
16-20	4
21+	5

Кількість атрибутів у класі	Відносне значення A'
0 чи 1-5	1
6-10	2
11-15	3
16-20	4
21+	5

Метрика СЧМА

- Чим більше методів та атрибутів містить клас, ти більше значення СЧМА, що в свою чергу збільшує об'єм та складність класів. Це тягне за собою збільшення часу та витрат на розробку.
- У випадку наслідування, дочірні класи мають доступ до атрибутів та методів батьківських, і якщо батьківський клас має більше значення СЧМА, це призводить до покращення показника СЧМА у дочірньому класі через повторне використання методів та атрибутів.
- Дуже високе значення СЧМА (~10) ще на етапі проектування сповіщає про потенційно великі затрати на реалізацію спроектованого рішення.
- Кількість методів та атрибутів в класі є першим індикатором того, наскільки багато зусиль доведеться прикласти для реалізації та підтримки проектованої програмної системи.

Класифікація та критерії

- Для простішої інтерпретації результатів застосування метрики було розроблено декілька типів результатів та таблиці відповідностей з критеріями для кожного з них.
- В залежності від отриманого значення нашу систему можна визначити як: складну, помірно складну та відносно просту.
- Також проведено спробу провести взаємозв'язок між нашою класифікацією та факторами якості програмного забезпечення (було обрано такі фактори, як зрозумілість, тестування та рівень ризику)
- Дані взаємозв'язки відображені у таблицях відповідностей

Вплив на якість програмної системи для результату «Відносно проста»

- Показник СЧМА від 2 до 4

Зрозумілість	Тестування	Рівень ризику
• Зрозуміла • Чистий код • Потрібно менше затрат та сил для імплементації методів та атрибутів	• Легке (менше затрат) • Тестування простих та добре структурованих методів	Низький

Вплив на якість програмної системи для результату «Помірно складна»

- Показник СЧМА від 5 до 8

Зрозумілість	Тестування	Рівень ризику
• Складна • Код менш чистий • Потрібно менше затрат та сил для імплементації методів та атрибутів	• Менш стабільне • Тестування більш складних методів • Більше затрат на розробку	Помірний

Вплив на якість програмної системи для результату «Складна»

- Показник СЧМА 9 чи 10

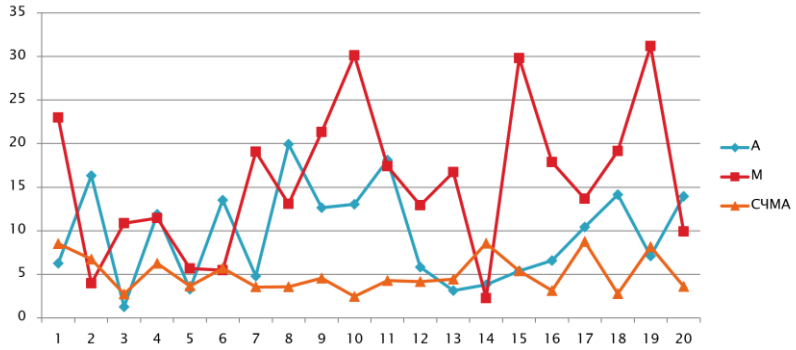
Зрозумілість	Тестування	Рівень ризику
• Складна поведінка системи	• Не стабільне та більш дорожче • Наявність методів які неможливо протестувати	Високі/дуже високі

Апробація результатів

- Для апробації результатів проведемо оцінку JavaScript бібліотеки під назвою undici за допомогою нашої метрики та декількох досліджених раніше, та проведем окореляцію

	undici
Строк коду	5984
Класів	32
Методів	351
Атрибутів	276

Апробація результатів



Значення параметрів метрики для бібліотеки undici

Апробація результатів

	Середнє	Мінімум	Максимум
M’	11,88	2	32
A’	7	0	20
СЧМА	4,65	2	9

Статистика значень при використанні метрики СЧМА до бібліотеки undici

Апробація результатів

	Мінімальне	Максимальне	Середнє	Стандартне відхилення	Дисперсія
WMC	2	32	11,96	7,55	57,04
CC	4	93	30,21	22,65	513,38
CMOOD	5	130	42,22	30,03	902,37
СЧМА	2	9	4,65	2,02	4,06

Статистичні дані при використанні метрик WMC, CC, CMOOD та СЧМА до бібліотеки undici

Апробація результатів

	WMC		CC		CMOOD		СЧМА	
	П	С	П	С	П	С	П	С
WMC	1	1	0.911	0.887	0.907	0.872	0.885	0.875
CC	0.911	0.887	1	1	0.984	0.975	0.925	0.915
CMOOD	0.907	0.872	0.984	0.975	1	1	0.881	0.842
СЧМА	0.885	0.875	0.925	0.915	0.881	0.842	1	1

Кореляція між результатами метрик WMC, CC, CMOOD та СЧМА до бібліотеки undici (П – Пірсон, С – Спірмен)

- Як бачимо з наведених таблиць, наша метрика СЧМА дуже добре корелюється з такими уже відомими метриками як WMC, CC та CMOOD, не допускаючи розриву більше ніж 0.2
- Найкраща кореляція отриманих результатів розробленої метрики СЧМА з метрикою CC (П: 0.925 С: 0.915)

Наукова новизна та практичне значення

- Удосконалено метод оцінювання складності об'єктно-орієнтованих програмних систем на етапі проектування.
- Удосконалено метрику оцінювання складності об'єктно-орієнтованих програмних систем на етапі проектування із встановленням взаємозв'язків між результатами розробленого методу та факторами якості програмної системи.

Практичне значення

- Практичне значення даного дослідження полягає у спрощенні методу та метрики оцінки складності об'єктно-орієнтованих програмних систем на етапі проектування, результати роботи якого добре корелюються із уже відомими, проте більш складними методами.
- Також, відповідно до отриманих результатів, за запропонованими таблицями відповідностей можна визначити фактори якості проектованої програмної системи.

Публікації за темою

- За темою кваліфікаційної роботи опублікована наукова стаття у фаховому виданні «Метод оцінки об'єктно-орієнтованих програмних систем на основі аналізу зміни вимог до програмної системи».

Висновки

У результаті виконання дипломної роботи здійснено аналіз наявних методів оцінки об'єктно-орієнтованих програмних систем на етапі проектування, визначені основні проблеми цієї галузі.

Здійснено удосконалення методу та метрики оцінки складності об'єктно-орієнтованої програмної системи на етапі проектування.

Проведено практичну апробацію удосконалених методу та метрики оцінки складності об'єктно-орієнтованої програмної системи на етапі проектування.

Результат апробації показав, що метод демонструє хороші показники кореляції з уже існуючими методами оцінки ПС. Проте, він набагато простіший, а також дає змогу провести взаємозв'язки між його результатами та факторами якості ПС.

ДЯКУЮ ЗА УВАГУ!

Завідувачу кафедри інженерії програмного
забезпечення проф. Бедратюку Л. П.
здобувача вищої освіти
Дзюрбана Е. С.
факультет ІТ, 2 курс, група ІПЗм-21-1

ЗАЯВА

З правилами чинного Положення «Про дотримання академічної доброчесності в Хмельницькому національному університеті» від 26.09.2020 (зі змінами від 26.11.2020), згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування заходів дисциплінарної та академічної відповідальності, ознайомлений. Про використання програмно-технічних засобів для перевірки кваліфікаційних робіт здобувачів вищої освіти на плагіат оповіщений та надаю свою згоду на обробку та збереження університетом моєї роботи в інституційному репозитарії університету.

Також надаю університету право на передачу моєї роботи для обробки та збереження в базах даних програмно-технічних засобів (Unicheck та Anti-Plagiarism) та використання роботи для виявлення плагіату в інших роботах, які перевіряються програмно-технічними засобами та користувачами, що мають доступ до цих програмно-технічних засобів, виключно в обмежених цілях для виявлення плагіату в текстах робіт.

Робота для перевірки університетом надається в друкованому та електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

4.12.2022 р.
дата

Дзюрбан
підпис

Tue Dec 06 11:44:23 EET 2022, Хіврич Володимир Русланович, Хмельницький національний університет, ХНУ

Anti-Plagiarism v-15.257

Максимальне співпадіння з одним документом 18.0%

Словники перевірки: en_US, ru_RU, ua_UA. **Помилок в документах: 8%**

ID: 109014 Назва: КРМ на тему: "Методи оцінювання об'єктно-орієнтованих програмних систем на етапі проектування Додано в БД: 2022-12-06 Автора: Дзюрбан Е.С. Керівники: Яшина О.М. Консультанти: Опоненти:	Документ		Сумарний збіг по Базі Даних	
	Символи	Лексеми	Символи	Лексеми
	106458	885	22559 (21%)	178 (20%)

Джерело плагіату

ID	Опис	Наявність плагіату в документі	
		Символи	Лексеми
107863	Назва: Звіт з переддипломної практики Додано в БД: 2022-10-05 Автора: Е. С. Дзюрбан Керівники: Яшина О. С. Консультанти: Опоненти:	19088 (18.0%)	145 (16.0%)

UNICHECK

Ім'я користувача:
Кафедра ІПЗ

ID перевірки:
1013202841

Дата перевірки:
06.12.2022 09:00:00 EET

Тип перевірки:
Doc vs Internet + Library

Дата звіту:
06.12.2022 09:04:58 EET

ID користувача:
100005589

Назва документа: **Записка Дзюрбан. Е. С.без дод**

Кількість сторінок: **75** Кількість слів: **15232** Кількість символів: **119058** Розмір файлу: **544.49 KB** ID файлу: **1012965638**

5.5%
Схожість

Найбільша схожість: **3.01%** з джерелом з Бібліотеки (ID файлу: **1009383373**)

2.55% Джерела з Інтернету	193	Сторінка 77
4.07% Джерела з Бібліотеки	165	Сторінка 78

0% Цитат

Вилучення цитат вимкнене

Вилучення списку бібліографічних посилань вимкнене

0%
Вилучень

Немає вилучених джерел

Модифікації

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

Замінені символи	3

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА ДИПЛОМНУ РОБОТУ

освітнього ступеня «магістр»

Магістр Дзюрбан Едуард Станіславович

Тема Методи оцінювання об'єктно-орієнтованих програмних систем на етапі проектування

Спеціальність 121 « Інженерія програмного забезпечення»

Обсяг дипломної роботи:

Кількість сторінок дипломної роботи 84.

1. Короткий зміст роботи та прийнятих рішень У дипломній роботі проведено аналіз основних методів та метрик для оцінювання об'єктно-орієнтованих програмних систем на етапі проектування. Розглянуто популярні методології для перевірки достовірності метрик. Розроблено власну метрику оцінки складності об'єктно-орієнтованих програмних систем. Проведено валідацію цієї материки, а також апробацію результатів її роботи та кореляцію з іншими, уже відомими, метриками.

2. Висновок про відповідність роботи дипломному завданню Дипломна робота освітнього ступеня «магістр» у повній мірі відповідає поставленому завданню як у теоретичній, так і в практичній її частині

2. Характеристика виконання кожного розділу роботи, ступінь використання останніх досягнень науки і техніки і передових методів роботи У вступі обгрунтовується актуальність теми роботи, формулюються цілі і завдання дослідження, описується наукова новизна та практична значимість отриманих результатів. У першому розділі охарактеризовано структуру предметної області та існуючі методи та метрики для вирішення проблеми оцінки програмних систем, виконана розгорнута постановка задачі. У другому розділі досліджено методології для валідації метрик, проведено їх вибірку, а також наведено припущення для розробки власної метрики оцінювання складності об'єктно-орієнтованих програмних систем на етапі проектування. У третьому розділі проведено детальний опис ідей нової метрики, її взаємозв'язок з атрибутами якості програмних систем. У четвертому розділі проведено валідацію розробленої метрики теоретичним шляхом за властивостями Вейюкера, а також проведено практичну апробацію результатів її роботи. Додатково проведена кореляція між уже відомими метриками та розробленою.

4. Позитивні сторони роботи Дипломна робота містить низку інноваційних рішень, зокрема, було розроблено власну метрику оцінки складності об'єктно-орієнтованих програмних систем. Також запропоновано залежність показників даної метрики з атрибутами якості програмних систем. Проведено теоретичну та практичну перевірку метрики та доведено її придатність до використання.

5. Негативні сторони роботи У роботі розроблена метрика використовує доволі прості критерії для оцінки, а також не враховує такі аспекти як складність алгоритмів реалізованих в методах, зв'язність між класами та їх розмір. Запропоновані взаємозв'язки лише з декількома атрибутами якості програмних систем, хоча затребуваним на реальній практиці є набагато ширший список.

6. Оцінка графічного оформлення та пояснювальної записки роботи Графічне оформлення виконане відповідно до теми дипломної роботи з дотриманням вимог стандартів. Пояснювальна записка відповідає вимогам стандартів до її оформлення.

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

8.Інші зауваження

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

РЕЦЕНЗЕНТ (прізвище, ім'я, по батькові, посада, місце роботи)
Н.чепурнук Андрій Олександрович, кандидат технічних наук, доцент кафедри комп'ютерної інженерії та інформаційних систем (КПС) ХНУ

«05» грудня 2022 р.

(підпис)

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

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

Назва: «Методи оцінювання об'єктно-орієнтованих програмних систем на етапі проектування»
Автор: Дзюрбан Едуард Станіславович
Спеціальність: 121 – Інженерія програмного забезпечення
Освітня програма: Освітньо-професійна програма «Інженерія програмного забезпечення»
Науковий керівник: Яшина Оксана Миколаївна, кандидат технічних наук, доцент
Після аналізу звіту подібності зроблено такий висновок:

№	Висновок	Позначка про відповідність
1	Запозичення, виявлені в роботі, є законними і не є плагіатом. Робота приймається до захисту.	**відповідає**
2	Виявлені запозичення не є плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи. Робота приймається до захисту, але має бути відкоригована. Відкоригований варіант має бути поданий на кафедру за 2 дні до захисту, разом із заявою щодо самостійності виконання письмової роботи та ідентичності друкованої та електронної версії роботи	
3	Виявлені запозичення не є плагіатом, але частково розміщені в розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. В зв'язку з цим мета роботи та поставлені завдання не були досягнені. Робота може бути допущена до захисту (наступного року) після того як буде відкоригована та допрацьована і успішно пройде повторну перевірку на академічний плагіат.	
4	Робота містить навмисні текстові спотворення, передбачувані спроби укриття запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	
5	Інше:	

Підтвердження:

Запозичення, виявлені в роботі, є законними і не є плагіатом, оскільки:

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

2) в якості запозичень системою було зафіксовано деякі послідовності вихідного коду і посилання на бібліотеки, які є стандартними мовними конструкціями програмування та не можуть розглядатися як об'єкт авторських прав і, відповідно, їх порушення;

3) усі запозичення є фрагментарними або мають належним чином оформленні посилання;

4) виявлені модифікації тексту не впливають на відсоток схожості.

Сумарний обсяг всіх запозичень, визначений системою виявлення збігів ідентичності/ схожості, складає 55% і адресується до 338 джерел, що, з урахуванням наведених обгрунтувань, відповідає характеру теми і свідчить на користь дипломної роботи.

Керівник _____ О. М. Яшина

Гарант ОП _____ О. М. Яшина

Завідувач кафедри _____ Л. П. Бедратюк