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

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач випускової кафедри
Аліна САВЧЕНКО
"___"_____2023 р.

КВАЛІФІКАЦІЙНА РОБОТА

(ДИПЛОМНИЙ ПРОЕКТ, ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСКНИКА ОСВІТНЬОГО СТУПЕНЯ "БАКАЛАВРА"
ЗА СПЕЦІАЛЬНІСТЮ 122 «КОМП'ЮТЕРНІ НАУКИ»

Тема: «Система забезпечення якості архітектури програм»

Виконавець: Захарченко Олександр Іванович

Керівник: к.т.н, доцент Харченко Олександр Григорович

Нормоконтролер: _____ Олександр ШЕВЧЕНКО
(підпис)

КИЇВ 2023

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет комп'ютерних наук та технологій

Кафедра комп'ютерних інформаційних технологій

Освітній ступінь: "Бакалавр"

Галузь знань, спеціальність, освітньо-професійна програма:

12 "Інформаційні технології, 122 "Комп'ютерні науки", "Інформаційні управляючі системи та технології" .

ЗАТВЕРДЖУЮ

Завідувач кафедри

Аліна САВЧЕНКО

"____" _____ 2023 р.

ЗАВДАННЯ

на виконання дипломного проекту студента

Захарченка Олександра Івановича

(П.І.Б. випускника)

1. **Тема роботи:** «Система забезпечення якості архітектури програм» затверджена наказом ректора № 623/ст. від 01.05.2023р.
2. **Термін виконання роботи:** з 15 травня 2023 року по 25 червня 2023 року.
3. **Вихідні дані до роботи:** Програма по визначенню нормалізованих ваг характеристик програмної архітектури та програмної системи, попарне порівняння характеристик.
4. **Зміст пояснювальної записки:** 1. Архітектура програмного продукту. Визначення та представлення архітектури. 2. Якість програмної архітектури. 3. Система забезпечення якості архітектури.
5. **Перелік обов'язкового графічного (ілюстративного) матеріалу:** діаграми варіантів використання, результати виконання обчислень значень ваг якості архітектури, слайди презентації MS PowerPoint.

6. **Календарний план-графік**

№ пор.	Завдання	Термін виконання	Підпис керівника
1.	Огляд літератури	15.05.23 - 18.05.23	
2.	Особливості архітектури програмного продукту та її представлення.	18.05.23 - 21.05.23	
3.	Оформлення 1-2 розділів дипломного проекту	21.05.23 - 25.05.23	
4.	Проектування системи та написання програми. Оформлення 3 розділу проекту	25.05.23 - 28.05.23	
5.	Оформлення пояснювальної записки. Висновки. Представлення роботи керівнику	28.05.23 - 29.05.23	
6.	Загальне редагування та друк пояснювальної записки	29.05.23 - 02.06.23	
7.	Написання тексту відповіді. Оформлення графічного метеріалу до презентації	02.06.23 - 17.06.23	
8.	Підготовка до захисту та попередній захист дипломного проекту	17.06.23 - 19.06.23	

7. **Дата видачі завдання** «15»_____травня_____2023р.

Керівник дипломного проекту _____ Олександр ХАРЧЕНКО

(підпис керівника)

Завдання прийняв до виконання _____ Олександр ЗАХАРЧЕНКО

(підпис випускника)

РЕФЕРАТ

Пояснювальна записка до дипломного проекту на тему: «Система забезпечення якості архітектури програм» містить: 76 сторінок, 18 рисунків, 20 інформаційних джерела, 1 додаток.

Мета роботи: впровадження ефективної системи, що гарантує дотримання галузевих стандартів, оцінює якість архітектури програмного забезпечення та відповідність бажаним критеріям.

Об'єкт дослідження: методи забезпечення якості архітектури програм при розробці програмного забезпечення.

Предмет дослідження: застосування методу парних порівнянь "Будинку якості", з метою покращення якості проектування програмних архітектур.

Методи дослідження: сучасні методики аналізу, моделювання систем візуалізаційними засобами.

Результати дипломного проекту рекомендується використовувати як готову модифікацію до програмного комплексу «Архітектор» та в практичній діяльності по проектуванню програмних архітектур.

Для розробки програмного забезпечення по визначенню нормалізованих ваг якості архітектури використано мову програмування C++, документацію Visual Studio.

ПРОГРАМНА АРХІТЕКТУРА, ПРОГРАМНИЙ ПРОДУКТ, ЯКІСТЬ, ПРОГРАМНА СИСТЕМА, СТАНДАРТ ISO/IEC 42010, СТАНДАРТ ISO\IEC 25010, ВИМОГИ ЯКОСТІ, МЕТОДИ ПРОЕКТУВАННЯ.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

ПА – програмна архітектура

ПС – програмна система

ПП – програмний продукт

ПІ – програмна інженерія

ADD - (Architectural Decision Diagrams - діаграми архітектурних рішень) - це інструмент для документування та візуалізації архітектурних рішень, що приймаються під час процесу проектування програмного забезпечення.

XML - (eXtensible Markup Language - розширювана мова розмітки) - це мова розмітки, яка визначає набір правил для кодування документів у форматі, придатному як для читання людиною, так і для машинного зчитування.

Alloy - формальна специфікаційна мова та інструмент для аналізу абстрактних моделей програмних систем.

TLA+ - (Temporal Logic of Actions) - формальний метод та мова специфікації для опису поведінки систем.

БД – база даних

ВСТУП

У сфері розробки програмного забезпечення, що стрімко розвивається, забезпечення якості архітектури програм набуває першочергового значення. Добре спроектована та надійна архітектура слугує основою для програмного продукту, впливаючи на його функціональність, продуктивність, ремонтопридатність та загальний успіх. Для задоволення цієї потреби необхідна розробка системи забезпечення якості програмної архітектури. Впровадження системи забезпечення якості ПА є важливими для забезпечення високої якості програмних архітектур. Дотримуючись галузевих стандартів, визначаючи та оцінюючи вимоги до якості та використовуючи ефективні методології оцінки, команди розробників програмного забезпечення можуть створювати архітектури, які є надійними, масштабованими, підтримуваними та узгодженими з очікуваннями зацікавлених сторін.

Ця робота зосереджена на розробці комплексної системи забезпечення якості програмної архітектури. Основна мета - запропонувати та впровадити ефективну систему, яка гарантує дотримання галузевих стандартів, оцінює якість архітектури програмного забезпечення та забезпечує впевненість у тому, що архітектурні проекти відповідають бажаним критеріям. Впровадивши таку систему, команди розробників програмного забезпечення зможуть підвищити надійність, ефективність та зручність обслуговування своїх продуктів.

Система забезпечення якості програмної архітектури відіграє вирішальну роль у забезпеченні якості та ефективності програмних архітектур. Вона охоплює процеси, методології та інструменти для підтримки розробки та високоякісних архітектур. Система повинна включати оцінку якості архітектури за заздалегідь визначеними критеріями, перевірку відповідності стандартам і правилам, управління архітектурними ризиками, оптимізацію та вдосконалення архітектури, документування та передачу архітектурних рішень, а також сприяння постійному вдосконаленню архітектурних практик.

Дана робота складається з трьох розділів.

Перший розділ присвячено фундаментальним аспектам архітектури програмного продукту. У ньому досліджується визначення архітектури та різні методи представлення архітектурних проектів. Обговорюється важливість дотримання стандартизованих підходів, таких як стандарт ISO/IEC 42010, який забезпечує основу для фіксації, документування та передачі архітектурних проектів.

У другому розділі розглядається поняття якості архітектури програмного забезпечення. Підкреслюється важливість визначення чітких вимог до якості програмних продуктів і наголошується на ролі зручності та простоти використання у досягненні задоволеності клієнтів. Стандарт ISO/IEC 25010 слугує орієнтиром для встановлення вимог до якості та оцінки відповідності архітектури програмного забезпечення цим вимогам.

Основна увага в цій роботі зосереджена у третьому розділі, де представлена запропонована система забезпечення якості архітектури. Метою є дослідження використання шаблонів для проектування архітектури та застосування альтернативних архітектур для прийняття оптимальних рішень. Також представлений програмний комплекс під назвою "Архітектор", який оптимізує вибір архітектури на основі попередньо визначених критеріїв якості. Крім того, розроблена програма, призначена для визначення вимог до якості під час архітектурного проектування, впроваджена у модифікацію програмного комплексу "Архітектор".

Для досягнення цілей, окреслених вище, використовується системний і ретельний підхід. Методологія дослідження охоплює як теоретичний аналіз, так і практичну реалізацію. Теоретичний аналіз передбачає поглиблений огляд відповідної літератури, галузевих стандартів та найкращих практик у сфері архітектури та забезпечення якості програмного забезпечення. Це створює всебічну теоретичну основу для розуміння ключових концепцій, принципів та методологій, пов'язаних із забезпеченням якості програмної архітектури. Аспект практичної реалізації цього дослідження передбачає розробку та перевірку запропонованої системи забезпечення якості програмної архітектури. Це включає розробку та впровадження програмних інструментів, алгоритмів та систем оцінювання, необхідних для оцінки та

забезпечення якості програмних архітектур. Система розроблена таким чином, щоб бути масштабованою, адаптованою та легко інтегрованою в існуючі процеси розробки програмного забезпечення.

РОЗДІЛ 1
АРХІТЕКТУРА ПРОГРАМНОГО ПРОДУКТУ. ВИЗНАЧЕННЯ ТА ПРЕДСТАВЛЕННЯ АРХІТЕКТУРИ

1.1. Визначення архітектури.

Архітектура програмного продукту - це фундаментальна концепція програмної інженерії, яка закладає основу для проектування та розробки системи. В рамках цього розділу розглядається визначення архітектури та її значення у розробці програмного забезпечення.

У контексті ПІ архітектура відноситься до загальної структури та організації програмної системи. Вона охоплює високорівневі проектні рішення та принципи, які формують компоненти системи, їх взаємозв'язки та взаємодію [1]. Архітектура забезпечує концептуальну основу, яка спрямовує процес розробки, гарантуючи, що система відповідає функціональним і нефункціональним вимогам.

Архітектура визначає ключові структурні елементи системи, такі як модулі, компоненти та інтерфейси, і визначає, як вони взаємодіють для досягнення цілей системи. Вона створює план системи, що дозволяє інженерам-програмістам зрозуміти і донести до зацікавлених сторін структуру, поведінку і функціональність системи.

Архітектура також визначає архітектурний стиль системи, який являє собою набір принципів і шаблонів, що керують організацією та дизайном системи [2]. До поширених архітектурних стилів належать багаторівнева архітектура, клієнт-серверна та архітектура, керована подіями. Кожен стиль має свої особливості та компроміси, що дозволяє архітектору програмного забезпечення вибрати найбільш притаманний для конкретних системних вимог.

Основними цілями архітектури програмного забезпечення є забезпечення функціональності, продуктивності, ремонтопридатності, масштабованості та безпеки системи. Заздалегідь визначивши архітектурний дизайн, потенційні ризики та проблеми можна виявити та пом'якшити на ранній стадії процесу розробки, що призведе до створення більш ефективної та результативної програмної системи. Тестування масштабованості допомагає визначити реакцію системи на динамічні зміни та її здатність ефективно розподіляти і використовувати ресурси для задоволення різних потреб [17].

Архітектуру можна розглядати як концептуальну модель, що забезпечує абстрактне представлення програмної системи. Вона фокусується на основних елементах системи, їх взаємозв'язках і загальній структурі, а не на конкретних деталях реалізації. Архітектурна модель допомагає зацікавленим сторонам, включаючи розробників, менеджерів проектів та клієнтів, візуалізувати та зрозуміти дизайн системи.

Одним з ключових аспектів архітектури є абстрагування, яке передбачає спрощення складних елементів системи до більш керованих і цілісних компонентів. Завдяки абстрагуванню архітектура визначає ключові функціональні та нефункціональні аспекти системи та декомпозує їх на менші модулі або підсистеми. Така декомпозиція дозволяє краще зрозуміти, організувати та керувати програмною системою.

Архітектура підтримує принцип поділу завдань, який передбачає поділ функціональності системи на окремі та незалежні частини. Кожен компонент або модуль в архітектурі повинен мати чітко визначену відповідальність і вирішувати конкретну проблему. Такий поділ сприяє модульності, багаторазовому використанню та простоті обслуговування, а також дозволяє ефективно співпрацювати між різними командами розробників, які працюють над конкретними частинами системи.

Архітектура передбачає прийняття чітких проектних рішень, які керують процесом розробки. Ці рішення стосуються таких важливих аспектів, як структура системи, протоколи зв'язку, управління даними, оптимізація продуктивності та обробка помилок.

Документуючи ці проектні рішення, архітектура слугує орієнтиром для розробників і допомагає підтримувати послідовність та узгодженість протягом усього життєвого циклу розробки програмного забезпечення.

Архітектура програмного забезпечення - це не одноразова діяльність; вона розвивається з часом, щоб пристосуватися до мінливих вимог, технологій та бізнес-потреб. У міру зростання системи і додавання нових функцій архітектура повинна мати можливість адаптуватися і масштабуватися, щоб забезпечити довгострокову життєздатність системи. Архітектурний дизайн повинен бути гнучким і розширюваним, що дозволяє легко інтегрувати нові компоненти або модулі та підтримувати майбутні вдосконалення та оновлення.

Розуміючи та визначаючи архітектуру програмної системи, організації можуть краще планувати та керувати процесом розробки, підвищити якість системи та її ремонтопридатність, а також зменшити ризик дорогих доопрацювань або збоїв у роботі системи.

Отже, визначення архітектури в інженерії програмного забезпечення стосується загальної структури, принципів проектування та організації програмної системи. Воно забезпечує концептуальну основу для розуміння і передачі інформації про компоненти системи, їх взаємозв'язки і взаємодію. Архітектура відіграє вирішальну роль у забезпеченні відповідності системи функціональним і нефункціональним вимогам, а також впливає на такі ключові характеристики, як продуктивність, ремонтопридатність і масштабованість.

1.2. Види представлення архітектури. Стандарт ISO/IEC 42010.

Види представлення архітектури

У контексті архітектури програмного забезпечення, представлення архітектури відіграє вирішальну роль у документуванні дизайну програмної системи. Стандарт ISO/IEC 42010 під назвою "Системи та ПІ. Опис архітектури" надає вказівки щодо різних типів представлення архітектури. Розглянемо деякі з найпоширеніших типів:

1. Текстові представлення:

Текстові представлення відіграють важливу роль в описі архітектури програмного забезпечення. Вони використовують природну мову, наприклад, англійську, для формулювання архітектурних елементів, зв'язків і властивостей. Текстові представлення надають наративний опис архітектури, специфікацій, вимог та проектної документації.

Однією з поширених технік текстового представлення є використання архітектурних патернів і стилів. Ці патерни фіксують повторювані архітектурні рішення і описують їх за допомогою текстових шаблонів або шаблонів, доповнених діаграмами. Прикладами архітектурних патернів є клієнт-сервер, багаторівнева архітектура, публікація-передплата.

Іншим підходом до текстового представлення є використання архітектурних представлень. Погляди розбивають архітектуру системи на окремі перспективи, кожна з яких фокусується на конкретних проблемах або зацікавлених сторонах. Кожне представлення надає текстовий опис архітектурних елементів, що мають відношення до цієї перспективи, що дозволяє зацікавленим сторонам зрозуміти систему з їхньої конкретної точки зору.

Текстове представлення може також включати використання формальних мов або мов моделювання, специфічних для архітектурного опису. Ці мови забезпечують структурований і стандартизований спосіб вираження компонентів, зв'язків і властивостей архітектури. Прикладами є мови опису архітектури (Architecture Description Languages, ADL), такі як мова аналізу та проектування архітектури (AADL, Architecture Analysis and Design Language) та мова опису архітектури на основі XML (xADL, XML-based Architecture Description Language).

Текстові представлення часто використовуються в поєднанні з іншими формами представлення, такими як діаграми, щоб забезпечити більш повне розуміння архітектури. Вони пропонують гнучкість у відображенні детальних пояснень, обґрунтувань та обмежень, які не завжди легко передати лише за допомогою графічних зображень.

Переваги текстових представлень включають в себе їх здатність фіксувати багату і детальну інформацію, підтримувати простежуваність між вимогами і проектними рішеннями, а також полегшувати текстовий аналіз і пошук. Однак вони можуть бути суб'єктивними і вимагають чіткого і стислого написання для забезпечення ефективної комунікації.

2. Схематичні представлення:

Діаграмні представлення широко використовуються в архітектурі програмного забезпечення для візуальної ілюстрації структури, поведінки та взаємодії компонентів системи. Ці представлення використовують графічні нотації для зображення архітектурних елементів, взаємозв'язків та обмежень, забезпечуючи візуальний засіб передачі складних проектних концепцій.

Одним із найпоширеніших діаграмних представлень в архітектурі програмного забезпечення є архітектурний план, також відомий як архітектурна діаграма. Ця діаграма надає високорівневий огляд системи, демонструючи її основні компоненти, їх взаємодію та загальну структуру. Вона часто включає клітинки, що представляють компоненти, лінії, що відображають зв'язки, і мітки, що вказують на їхні ролі та обов'язки [4].

Іншим популярним видом схематичного представлення є компонентна діаграма. Ця діаграма фокусується на ілюстрації взаємозв'язків і залежностей між компонентами системи. Вона візуально представляє, як компоненти взаємодіють і співпрацюють для виконання функціональних можливостей системи [4].

Діаграма розгортання - це ще одне корисне схематичне зображення, яке відображає фізичне розташування компонентів системи, таких як сервери, апаратні пристрої та мережі. Вона показує, як програмні компоненти розподілені між різними апаратними вузлами і як вони взаємодіють один з одним [3].

Діаграми послідовності та діаграми зв'язків використовуються для опису динамічної поведінки системи. Вони демонструють послідовність взаємодій між компонентами та повідомлення, якими вони обмінюються під час виконання [4].

UML (Unified Modeling Language - уніфікована мова моделювання) є широко прийнятим стандартом для діаграмного представлення архітектури програмного

забезпечення. Він надає повний набір нотацій і діаграм для відображення різних аспектів архітектури, включаючи діаграми класів, діаграми діяльності та діаграми станів.

Діаграмні представлення мають ряд переваг, включаючи легкість розуміння, стислу візуалізацію складних структур та ефективну комунікацію між зацікавленими сторонами. Вони забезпечують візуальну мову, яка полегшує обговорення, аналіз і прийняття рішень під час процесу розробки програмного забезпечення.

3. Табличні представлення:

Табличні представлення - це альтернативний підхід до представлення архітектури програмного забезпечення, що фокусується на організації архітектурної інформації у структурованому та табличному форматі. Ці представлення використовують таблиці для представлення даних, властивостей і взаємозв'язків архітектурних елементів, надаючи стисле і систематизоване уявлення про структуру системи.

Одним з поширених типів табличного представлення є таблиця архітектурних рішень. Ця таблиця фіксує архітектурні рішення, прийняті в процесі проектування, включаючи їх обґрунтування, залежності та вплив на систему. Це допомагає в документуванні та управлінні архітектурними рішеннями, забезпечуючи прозорість і простежуваність [4].

Іншим табличним представленням є таблиця інтерфейсів компонентів. Ця таблиця визначає інтерфейси, доступні для кожного компонента, включаючи методи, параметри та типи даних. Вона надає повне уявлення про інтерфейсний контракт компонента, полегшуючи інтеграцію та співпрацю між компонентами [4].

Словник даних - це ще одне табличне представлення, яке зазвичай використовується в архітектурі програмного забезпечення. Він каталогізує сутності даних, атрибути та зв'язки, що використовуються в системі. Він слугує довідником для розуміння моделі даних і підтримки проектних рішень на основі даних.

Табличне представлення має низку переваг з точки зору простоти, читабельності та здатності збирати та організовувати детальну інформацію у структурованому форматі. Вони полегшують швидкий доступ до конкретних точок даних,

уможливлюють легке порівняння та аналіз, а також підтримують документування та відстеження архітектурних артефактів.

4. Математичні представлення:

Математичні представлення в архітектурі програмного забезпечення створюють формальний і строгий підхід до опису, аналізу структури, поведінки та властивостей системи. Ці представлення використовують математичні концепції, моделі та нотації для вираження архітектурних елементів, зв'язків, обмежень та їхньої взаємодії.

Одним з найпоширеніших математичних представлень в архітектурі програмного забезпечення є використання формальних мов специфікацій. Ці мови, такі як Alloy або TLA+, дозволяють архітекторам визначати точні та однозначні специфікації архітектури системи, включаючи її компоненти, інтерфейси та взаємодії. Формальні мови специфікацій уможливлюють ретельний аналіз, верифікацію та валідацію архітектурних властивостей, таких як узгодженість, коректність або продуктивність [5].

Ще однією технікою математичного представлення є використання теорії графів. Графи забезпечують математичну абстракцію для представлення архітектурних елементів як вузлів, а їхніх зв'язків - як ребер. Графові представлення, такі як граф залежностей, граф взаємодії компонентів або граф потоків управління, допомагають у візуалізації та аналізі структури, залежностей і поведінки системи. Теорія графів надає потужні алгоритми та методи для аналізу таких властивостей, як зв'язність, досяжність або складність в архітектурі програмного забезпечення.

Математичні представлення в архітектурі програмного забезпечення пропонують переваги з точки зору формалізму, точності та аналітичних можливостей. Вони уможливлюють суворе обґрунтування, аналіз і перевірку архітектурних властивостей, допомагаючи архітекторам забезпечити коректність, надійність і продуктивність системи.

5. Візуальні представлення:

Візуальні представлення в архітектурі програмного забезпечення надають графічні засоби для представлення структури, поведінки та взаємозв'язків у системі. Ці представлення використовують діаграми та візуальні нотації для передачі складних архітектурних концепцій у більш інтуїтивний та зрозумілий спосіб.

Однією з широко використовуваних технік візуального представлення є уніфікована мова моделювання (Unified Modeling Language, UML). UML пропонує набір діаграм для представлення різних аспектів архітектури програмного забезпечення. Діаграми UML, такі як діаграми класів, діаграми компонентів, діаграми послідовностей та діаграми розгортання, забезпечують візуальне представлення структури системи, взаємодій та конфігурацій розгортання. Діаграми UML широко визнані та зрозумілі, що робить їх ефективними інструментами комунікації між архітекторами, розробниками та зацікавленими сторонами.

На додаток до UML, існують інші методи візуального представлення, специфічні для архітектури програмного забезпечення. Наприклад, діаграми архітектурних рішень (ADD) використовуються для документування і передачі архітектурних рішень та їх обґрунтування. ADD використовують графічні елементи, такі як поля, стрілки та вузли рішень, для представлення процесу прийняття рішень та впливу рішень на архітектуру [1].

Інструменти і методи візуалізації, такі як 3D моделювання, інтерактивні візуалізації або віртуальна реальність, також використовуються для візуального представлення архітектури програмного забезпечення [6]. Ці вдосконалені візуальні представлення забезпечують ефект занурення, дозволяючи архітекторам і зацікавленим сторонам досліджувати і взаємодіяти з архітектурою в більш захоплюючий та інтерактивний спосіб.

Візуальні представлення в архітектурі програмного забезпечення покращують розуміння, полегшують комунікацію та підтримують аналіз і прийняття рішень. Вони допомагають архітекторам і зацікавленим сторонам зрозуміти складні архітектурні концепції, виявити проблеми проектування і перевірити архітектурні рішення.

Підводячи підсумки, вибір способу представлення архітектури залежить від таких факторів, як необхідний рівень деталізації, мета комунікації або документування, а також наявні інструменти або методи. Зазвичай використовують комбінацію різних типів представлення, щоб забезпечити всебічне і цілісне уявлення про архітектуру програмного забезпечення.

Стандарт ISO/IEC 42010

Стандарт ISO/IEC 42010, також відомий як "Systems and software engineering — Architecture description" (Системна та програмна інженерія - опис архітектури), містить настанови щодо опису та документування архітектури програмно-містких систем. Він визначає основні поняття, принципи та процеси для опису архітектури і сприяє ефективній комунікації та взаєморозумінню між зацікавленими сторонами, які беруть участь у розробці та підтримці системи. Стандарт також дає інформацію про загальні принципи опису архітектури ПС, а тому його слід використовувати в процесі проектування ПС на відповідному етапі життєвого циклу.

ISO/IEC 42010 визначає архітектуру як фундаментальні концепції або властивості системи в її оточенні, втілені в її елементах, взаємозв'язках, а також у принципах її проектування та еволюції [7]. Він підкреслює важливість відображення як структури і поведінки системи, так і взаємозв'язків і взаємодії з її оточенням.

Стандарт забезпечує основу для опису архітектури, яка включає різні точки зору і погляди. Точки зору представляють перспективи, з яких архітектура аналізується і документується, наприклад, функціональна, інформаційна або точка зору розгортання. Погляди, з іншого боку, є конкретними представленнями архітектури з певної точки зору. ISO/IEC 42010 заохочує використання декількох точок зору та поглядів, щоб охопити різні проблеми та полегшити цілісне розуміння архітектури.

Стандарт також визначає архітектурні фреймворки як структури багаторазового використання, які керують розробкою та документуванням архітектур. Ці фреймворки надають набір конвенцій, настанов і рекомендованих практик для опису архітектури, сприяючи узгодженості, інтероперабельності та повторному використанню в різних системах [7].

ISO/IEC 42010 підкреслює важливість зацікавлених сторін в описі архітектури та заохочує їх до активного залучення протягом усього процесу. Він визнає, що архітектура є соціальною конструкцією, на яку впливають різні перспективи, цілі та занепокоєння зацікавлених сторін. Ефективна комунікація та співпраця між зацікавленими сторонами мають вирішальне значення для створення спільного

розуміння архітектури та забезпечення її відповідності запланованим цілям та вимогам [7].

Дотримуючись керівних принципів ISO/IEC 42010, архітектори програмного забезпечення та організації можуть підвищити якість, ясність і узгодженість своїх описів архітектури. Стандарт сприяє поширенню передових практик документування архітектури, полегшуючи комунікацію, прийняття рішень та розуміння системи серед зацікавлених сторін.

В загальному опис архітектури може бути представлений у вигляді схеми, яка містить згадані вище сутності та показана на рис.1.1.

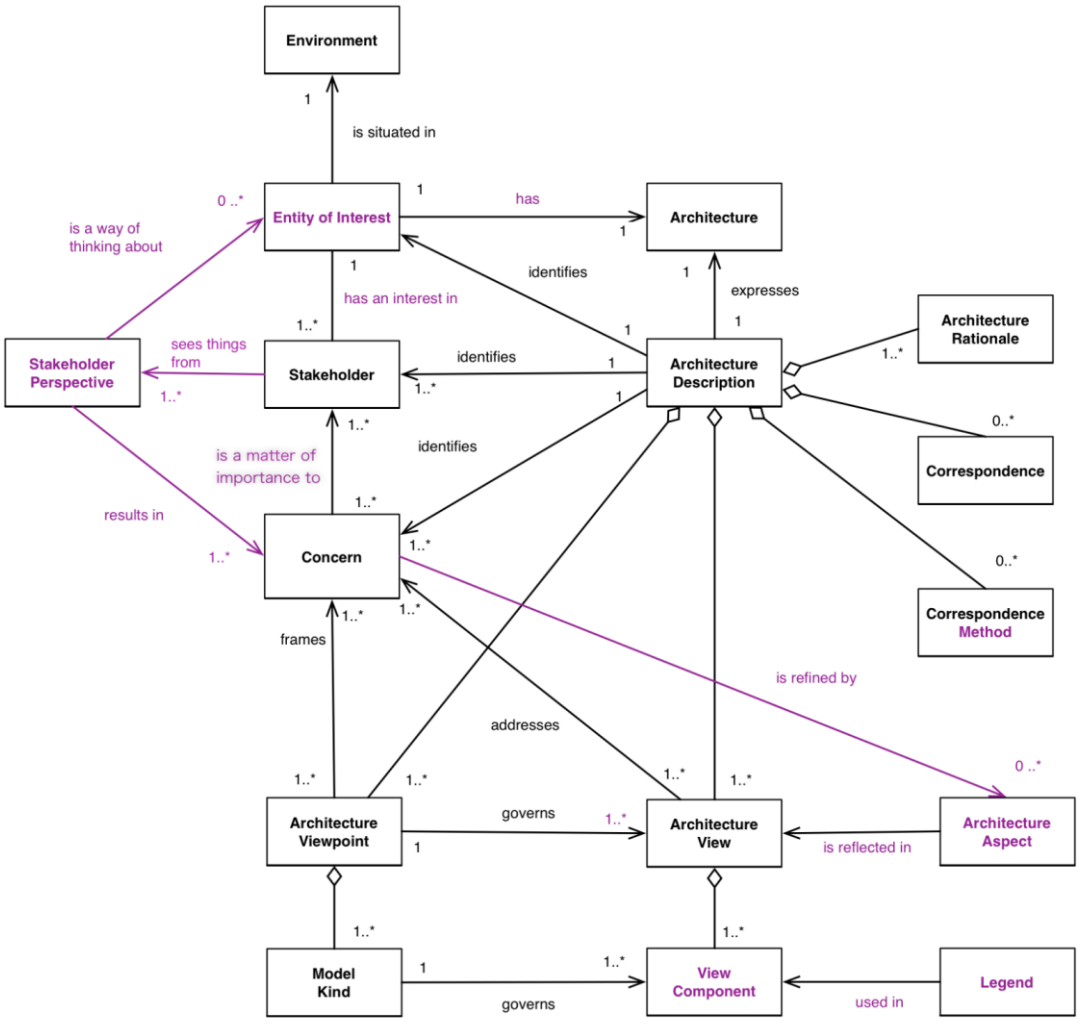


Рис. 1.1. Концептуальна модель опису архітектури програмної системи згідно стандарту ISO/IEC/IEEE 42010 2-го видання

Ця схема відображає зміни, яких завдала нова концепція у 2-му виданні стандарту ISO/IEC/IEEE 42010. У новому виданні стандарту ISO/IEC/IEEE 42010 використано діаграму класів на основі UML. Елементи, виділені фіолетовим кольором, є новими або зміненими порівняно з попереднім виданням.

Для порівняння, схема 1-го видання показана на рис.1.2.

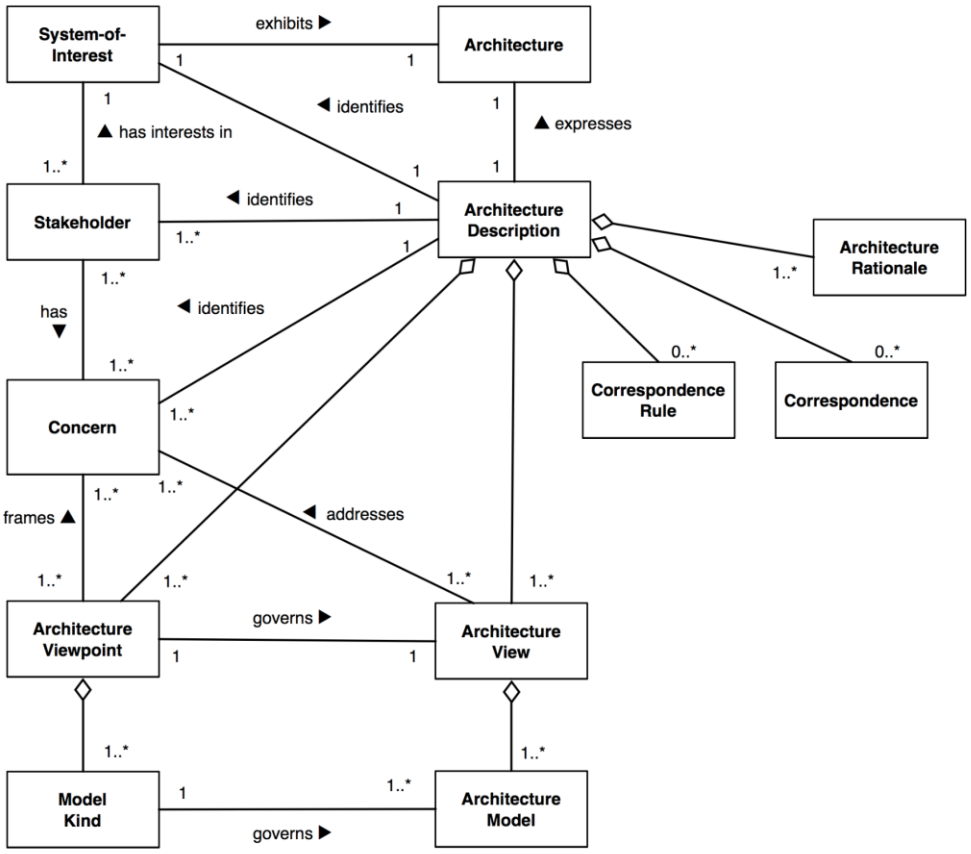


Рис.1.2. Концептуальна модель опису архітектури програмної системи згідно стандарту ISO/IEC/IEEE 42010 1-го видання

Architecture Description (Опис архітектури):

Опис архітектури - це робочий продукт, який використовується для вираження архітектури деякої системи, що представляє інтерес. Стандарт визначає вимоги до AD (Architecture Description). AD описує одну з можливих архітектур системи, що представляє інтерес. AD може мати форму документа, набору моделей, репозиторію моделей або будь-яку іншу форму.

Stakeholder (Зацікавлена сторона):

Стейкхолдери - це окремі особи, групи або організації, що мають інтереси щодо системи, яка їх цікавить. Приклади зацікавлених сторін: клієнт, власник, користувач, споживач, постачальник, розробник, супроводжувач, аудитор, генеральний директор, орган сертифікації, архітектор.

Concern (Стурбованість):

Стурбованість - це будь-який інтерес до системи. Термін походить від фрази "розділення інтересів", яку вперше запропонував Едсгар Дейкстра (Edsgar Dijkstra). Приклади проблем: призначення (системи), функціональність, структура, поведінка, вартість, підтримка, безпека, інтероперабельність.

Architecture Viewpoint (Точка зору на архітектуру):

Точка зору на архітектуру - це набір угод для побудови, інтерпретації, використання та аналізу одного типу архітектури. Точка зору включає типи моделей, мови та нотації точок зору, методи моделювання та аналітичні методи для формування певного набору проблем. Приклади точок зору: операційна, системна, технічна, логічна, розгортання, процес, інформація.

Architecture View (Подання архітектури):

Погляд на архітектуру в аналізі ризиків виражає архітектуру системи, що представляє інтерес, з точки зору однієї або декількох зацікавлених сторін для вирішення конкретних проблем, використовуючи умовності, встановлені їх точкою зору. Погляд на архітектуру складається з однієї або декількох моделей архітектури.

Architecture Model (Модель архітектури):

Погляд складається з моделей архітектури. Кожна модель побудована відповідно до умовностей, встановлених її Типом Моделі, який зазвичай визначається як частина її керівної точки зору. Моделі надають засоби для обміну деталями між поданнями та для використання декількох нотацій у межах подання.

Model Kind (Тип моделі):

Тип моделі визначає домовленості для одного типу моделі архітектури.

В цілому, друге видання узагальнює тему опису архітектури від *Системи, що представляє інтерес*, до *Сутності, що представляє інтерес*. У попередньому виданні

Архітектурне уявлення складається з однієї або декількох моделей архітектури. У новому виданні модель архітектури замінено на *компонент подання* (View Component). У цьому виданні представлено *Перспективи зацікавлених сторін* (Stakeholder Perspectives) як засіб для групування *Проблем* і, відповідно, для організації *Точок зору*, що обрамляють ці *Проблеми*. Ця редакція також вводить *Аспекти Архітектури: характеристики Об'єкта*, що представляє інтерес, які відображаються в *Поглядах Архітектури*. Серед іншого, Аспекти можуть бути використані для покращення простежуваності між *Проблемами* та *Поглядами*.

1.3. Багатошарове представлення архітектури.

Багаторівневе представлення архітектури - це підхід до зображення різних шарів або рівнів абстракції в архітектурі програмної системи. Воно забезпечує ієрархічне представлення компонентів системи та їхніх взаємозв'язків, що дозволяє краще розуміти складність системи та керувати нею.

У багатошаровому представленні програмна система поділяється на окремі шари, кожен з яких представляє певний аспект або функціональність системи. Ці шари організовані в ієрархічному порядку, де вищі шари залежать від нижчих у своїй роботі. Така ієрархічна структура допомагає модулювати систему і сприяє розподілу завдань.

Багаторівневе представлення дозволяє розділити обов'язки та інкапсуляцію функціональності в межах кожного рівня. Кожен рівень, як правило, представляє певний домен або функціональну область системи, таку як презентація, бізнес-логіка, доступ до даних та інфраструктура. Такий поділ полегшує обслуговування, масштабування та повторне використання компонентів.

Взаємозв'язки між рівнями визначаються інтерфейсами або протоколами, які вказують, як рівні взаємодіють один з одним. Ці інтерфейси визначають послуги, що надаються кожним рівнем, і залежності між рівнями. Чітко визначаючи інтерфейси, багаторівневе представлення полегшує вільний зв'язок між компонентами, дозволяючи легше модифікувати і замінювати окремі шари, не впливаючи на всю систему.

Багаторівневе представлення архітектури широко використовується в різних архітектурних стилях, таких як багаторівнева архітектура, клієнт-серверна архітектура та сервіс-орієнтована архітектура. Воно допомагає досягти модульності, розподілу завдань і масштабованості програмних систем.

Кілька авторитетних джерел надають інформацію та рекомендації на цю тему.

Згідно з Б. Клементс та Казман [1], багаторівневе представлення пропонує ієрархічний погляд на програмну систему, розділяючи її на окремі шари, які представляють певні функціональні можливості або домени. Такий підхід дозволяє краще управляти складністю і сприяє розділенню проблем.

Шоу та Гарлан підкреслюють переваги багатошарового представлення у досягненні модульності та масштабованості [2]. Організовуючи систему в шари і визначаючи чіткі інтерфейси, компоненти в межах кожного шару можуть розроблятися і підтримуватися незалежно, що полегшує еволюцію системи.

Тейлор, Медвідовік та Дасхофі обговорюють значення багаторівневого представлення в архітектурі програмного забезпечення та його застосовність у різних архітектурних стилях [5]. Вони підкреслюють важливість вільного зв'язку між шарами через чітко визначені інтерфейси, що дозволяє змінювати або замінювати окремі шари, не впливаючи на всю систему.

Багатошарове представлення узгоджується з принципами модуляризації, розподілу завдань і масштабованості, які є важливими для надійних і підтримуваних програмних архітектур.

Логічний поділ на шари

При розробці програмного забезпечення загальним підходом є логічний поділ системи на шари, незалежно від її типу чи інтерфейсу. Ці шари слугують логічними групами програмних компонентів, які виконують різні завдання і полегшують повторне використання компонентів. Кожен шар складається з певних типів компонентів, згрупованих у підшари, кожен з яких має свої власні визначені обов'язки.

Визначивши загальні типи компонентів, які зустрічаються в більшості рішень, ви можете створити діаграму, яка представляє структуру програми або сервісу. Ця діаграма слугує ескізом дизайну, окреслюючи організацію системи. Використання

багаторівневого підходу має низку переваг, серед яких покращена підтримка коду, оптимізована продуктивність у різних сценаріях розгортання, а також чітке розмежування між застосуванням технологій і проектними рішеннями.

Поділ додатків на шари полегшує підтримку коду, підвищує оптимізацію продуктивності та забезпечує чіткий фокус на застосуванні технологій і виборі дизайну.

На рис 1.3. показано спрощене високорівневе представлення цих шарів та їх взаємовідносин з користувачами, іншими додатками, що викликають сервіси, реалізовані в бізнес-шарі додатки, джерелами даних, такими як реляційні бази даних або Веб-сервіси, що забезпечують доступ до даних, і зовнішніми або віддаленими сервісами, використовуваними додатком.

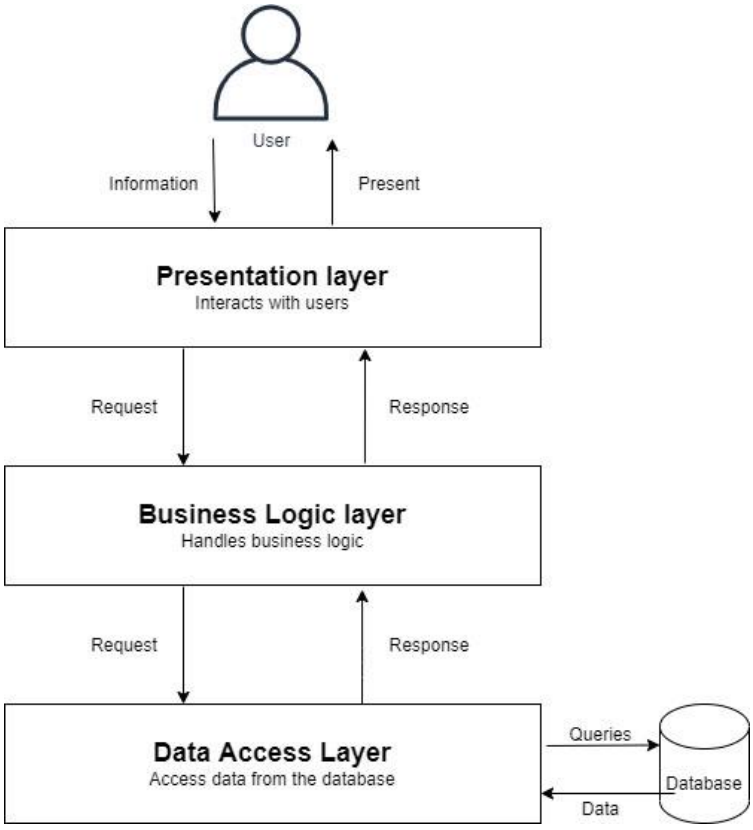


Рис. 1.3. Логічне представлення архітектури багатошарової системи

Розміщення програмних шарів може бути різним: вони можуть бути на одному рівні або розділені фізично. У випадках, коли вони розташовані на різних рівнях або мають фізичні кордони між собою, дуже важливо, щоб дизайн забезпечував належну інтеграцію та комунікацію.

На рис. 1.3 представлений приклад програми, що складається з декількох фундаментальних шарів. Звичайний тришаровий дизайн, як показано на рис. 1.3, складається з наступних шарів:

1. Presentation layer (Рівень представлення): Цей рівень фокусується на функціональності, орієнтованій на користувача, і полегшує взаємодію користувача з системою. Зазвичай він включає компоненти, що відповідають за загальний зв'язок з основною бізнес-логікою, яка знаходиться на бізнес-рівні.
2. Buisiness Logic layer (Бізнес-рівень): Слугує основою системи, цей рівень реалізує її основну функціональність та інкапсулює пов'язану з нею бізнес-логіку. Він складається з компонентів, деякі з яких надають сервісні інтерфейси, які можуть бути використані іншими учасниками взаємодії системи.
3. Data Access layer (Рівень доступу до даних): Відповідає за доступ як до даних, що зберігаються в системі, так і до даних, що надаються зовнішніми мережевими системами, і забезпечує пошук даних за допомогою сервісів. Він пропонує універсальні інтерфейси, які можуть бути використані компонентами бізнес-рівня.

Етапи проектування багатошарової структури

Проектування багатошарової структури включає кілька етапів для забезпечення добре організованої та ефективної архітектури. Нижче наведено ключові етапи проектування багатошарової структури:

1. Аналіз вимог: На цьому етапі архітектор аналізує функціональні та нефункціональні вимоги до системи. Це включає розуміння потреб користувачів, бізнес-цілей, очікуваної продуктивності, вимог до безпеки та інших важливих факторів.
2. Декомпозиція системи: Архітектор декомпозує систему на логічні компоненти та визначає різні необхідні шари. Це передбачає розбиття функціональності системи на окремі модулі, які можна згрупувати на основі пов'язаних з ними завдань та обов'язків.
3. Ідентифікація рівнів: Архітектор визначає конкретні рівні, необхідні для системи, виходячи з характеру програми. Зазвичай це такі рівні, як рівень

представлення/вигляду, рівень бізнес-логіки, рівень доступу до даних і, можливо, додаткові рівні, залежно від складності та вимог системи.

4. Відповідальність шарів: На кожен рівень покладено певні обов'язки та завдання. Архітектор визначає функції та операції, які повинен виконувати кожен рівень, забезпечуючи чіткий розподіл обов'язків і сприяючи модульному дизайну.
5. Взаємодія рівнів: Архітектор встановлює шаблони комунікації та взаємодії між рівнями. Це включає визначення інтерфейсів, протоколів і механізмів обміну даними для полегшення безперешкодної взаємодії та потоку даних між рівнями.
6. Залежності між рівнями: Архітектор визначає залежності між рівнями. Це передбачає розуміння того, як шари залежать один від одного, і забезпечення належного управління цими залежностями, щоб мінімізувати зв'язок і сприяти вільному з'єднанню між шарами.
7. Принципи та патерни проектування: Архітектор застосовує принципи та закономірності проектування, щоб забезпечити якість та ефективність багатошарової структури. Сюди входять такі принципи, як розподіл завдань, інкапсуляція та модульність, а також архітектурні патерни, які найкраще відповідають вимогам системи.
8. Ітеративне вдосконалення: Процес проектування передбачає ітеративне доопрацювання, коли архітектор переглядає та вдосконалює багатошарову структуру на основі зворотного зв'язку, тестування та валідації. Такий ітеративний підхід допомагає виявити та усунути будь-які проблеми чи недоліки в дизайні.

Дотримуючись цих етапів, архітектори можуть створити добре структуровану і модульну багатошарову архітектуру, яка відповідає вимогам системи, сприяє повторному використанню коду, покращує ремонтопридатність, а також забезпечує масштабованість і гнучкість при майбутніх вдосконаленнях або модифікаціях.

1.4. Методи проектування архітектури.

Методи проектування архітектури - це систематичні підходи та методи, що використовуються для проектування архітектури програмної системи. Ці методи допомагають архітекторам приймати обґрунтовані рішення, фіксувати вибір дизайну та забезпечувати досягнення бажаних якостей і характеристик системи. Найпоширеніші методи проектування архітектури представляють собою:

- Архітектуру, керовану моделлю (Model-Driven Architecture, MDA): MDA - це підхід, який наголошує на використанні моделей для проектування програмних систем. Він передбачає створення незалежних від платформи моделей (PIM) і перетворення їх у специфічні для платформи моделі (PSM) для генерації коду. MDA сприяє розділенню проблем і полегшує архітектурне проектування та аналіз.
- Доменно-орієнтоване проектування (Domain-Driven Design, DDD): DDD - це метод архітектурного проектування, який фокусується на розумінні предметної області програмної системи та узгодженні проекту з концепціями предметної області. Він наголошує на моделюванні проблемної області та створенні повсюдної мови, яка пов'язує бізнес-вимоги з дизайном програмного забезпечення.
- Сервіс-орієнтована архітектура (SOA): SOA - це архітектурний підхід до проектування, який передбачає проектування системи як набору слабко пов'язаних між собою сервісів. Ці сервіси мають чітко визначені інтерфейси і можуть бути об'єднані для забезпечення більшої функціональності. SOA сприяє багаторазовому використанню, гнучкості та інтероперабельності.
- Компонентно-орієнтована розробка (CBD): CBD - це метод архітектурного проектування, який наголошує на використанні багаторазових програмних компонентів. Він передбачає декомпозицію системи на менші, самодостатні компоненти, які можна зібрати і налаштувати для створення бажаної функціональності. CBD сприяє модульному дизайну, повторному використанню коду та легкості обслуговування.

- Архітектура, керована подіями (Event-Driven Architecture, EDA): EDA - це підхід до архітектурного проектування, який фокусується на потоці подій та повідомлень в системі. Він передбачає проектування системи, яка реагує на події та запускає дії, засновані на керованих подіями взаємодіях. EDA сприяє масштабованості, реагуванню та вільному зв'язку.
- Гнучка архітектура: Гнучка архітектура відноситься до практики включення принципів і рішень архітектурного проектування в гнучкий процес розробки. Вона передбачає ітеративне та інкрементальне проектування, що забезпечує гнучкість та адаптивність у відповідь на мінливі вимоги. Гнучка архітектура сприяє співпраці, зворотному зв'язку та постійному вдосконаленню.
- Патерни дизайну: Патерни дизайну - це багаторазові рішення архітектурних проблем, що часто зустрічаються. Вони надають перевірені підходи до вирішення проектних завдань і допомагають архітекторам у прийнятті проектних рішень. Прикладами патернів проектування є патерн Model-View-Controller (MVC), патерн Observer та патерн Factory.

Ці методи проектування архітектури надають фреймворки, керовані принципи і методи, які допомагають архітекторам проектувати програмні системи, що відповідають бажаним цілям, таким як масштабованість, ремонтопридатність, продуктивність і зручність для користувача. Архітектори можуть вибирати і комбінувати ці методи на основі конкретних вимог і обмежень системи, що проектується.

Метою методів проектування архітектури є забезпечення структурованого та систематичного підходу до проектування архітектури програмного забезпечення, що відповідає бажаним цілям та якостям системи. Ці методи спрямовані на те, щоб допомогти архітекторам приймати обґрунтовані рішення, фіксувати вибір дизайну і забезпечувати успішну реалізацію архітектури системи.

Вирішення нефункціональних вимог, таких як продуктивність, масштабованість, безпека та ремонтопридатність, мають вирішальне значення для успіху програмної системи [8]. Методи проектування архітектури надають механізми і рекомендації для

включення цих вимог у процес проектування, гарантуючи, що отримана архітектура задовольняє цим аспектам.

Модульні та багаторазові архітектури є дуже бажаними, оскільки вони покращують ремонтопридатність, зменшують складність і полегшують майбутні вдосконалення [9]. Методи архітектурного проектування допомагають архітекторам розробляти архітектури з модульними компонентами і чітко визначеними інтерфейсами, сприяючи повторному використанню і гнучкості.

Також, масштабованість і продуктивність є критично важливими для систем, які повинні обробляти зростаючі навантаження або великі обсяги даних. Методи проектування архітектури допомагають архітекторам розробляти масштабовані архітектури, враховуючи такі фактори, як розподілена обробка, балансування навантаження, кешування та методи оптимізації [8].

Користувацький досвід відіграє життєво важливу роль в успіху програмних систем. Методи проектування архітектури допомагають архітекторам враховувати точку зору користувача та розробляти архітектури, які забезпечують інтуїтивно зрозумілі інтерфейси, ефективні робочі процеси та швидку взаємодію. Якість у використанні по тестуванню може дати уявлення про вузькі місця, когнітивне навантаження та проблеми навігації, які можуть перешкоджати ефективності та продуктивності користувача [18].

Оцінка користувацького досвіду в системі виходить за рамки якості у використанні і зосереджується на загальному задоволенні та емоційній реакції користувачів. Вона враховує такі фактори, як естетика, відповідність бренду та емоційний вплив дизайну системи. Методи дослідження, такі як інтерв'ю з користувачами, опитування та метрики якості у використанні, можуть бути використані для фіксації сприйняття, емоцій та уподобань користувачів [19].

Отже, методи проектування архітектури відіграють вирішальну роль у створенні ефективних архітектур програмного забезпечення. Вони забезпечують структурований і системний підхід до вирішення функціональних і нефункціональних вимог, сприяють модульності та багаторазовому використанню, полегшують системну інтеграцію, забезпечують масштабованість і продуктивність, покращують користувацький досвід,

підтримують еволюцію та адаптивність. Використовуючи методи проектування архітектури, архітектори можуть приймати обґрунтовані рішення, фіксувати вибір дизайну і створювати архітектури, які відповідають бажаним цілям і якостям системи. Ці методи надають керівні принципи, техніки та фреймворки, які дозволяють архітекторам орієнтуватися в складнощах проектування архітектури програмного забезпечення та створювати успішні програмні системи.

З постійним розвитком програмних систем і технологій методи проектування архітектури стають ще більш важливими. Вони забезпечують основу для проектування архітектур, які можуть адаптуватися і розвиватися з часом, пристосовуючись до мінливих вимог і технологічного прогресу. Беручи до уваги принципи і прийоми, що надаються методами архітектурного проектування, архітектори можуть створювати архітектури, які є гнучкими, масштабованими і здатними прийняти майбутні зміни без значних порушень.

РОЗДІЛ 2
ЯКІСТЬ ПРОГРАМНОЇ АРХІТЕКТУРИ

2.1. Вимоги якості програмного продукту та зручності використання. Стандарт ISO\IEC 25010.

Забезпечення якості програмного продукту має вирішальне значення для його успіху та задоволення потреб користувачів. Стандарт ISO/IEC 25010 забезпечує комплексну основу для визначення та оцінювання характеристик якості програмних продуктів. Він пропонує настанови щодо оцінювання різних атрибутів якості, включаючи зручність використання.

ISO/IEC 25010 - це стандарт, який визначає моделі якості для якості програмного продукту та якості системи у використанні. Цей стандарт забезпечує основу для оцінювання якості програмних продуктів і систем на основі набору характеристик і підхарактеристик. Моделі якості, визначені в ISO/IEC 25010, базуються на стандарті ISO/IEC 9126-1, який був вперше опублікований у 2001 році. Також, стандарт надає систематичний підхід до визначення, специфікації та оцінки якості ПП. Він встановлює загальноприйняті терміни та визначення, що сприяють зрозумілості та узгодженості між зацікавленими сторонами; є важливим інструментом для розробників, постачальників та замовників програмного забезпечення; сприяє зростанню довіри до продуктів, поліпшенню комунікації між зацікавленими сторонами та забезпечує відповідність програмних продуктів вимогам і очікуванням користувачів.

Метою стандарту ISO/IEC 25010 є надання набору моделей якості, які можуть бути використані для оцінювання якості програмних продуктів та систем. Ці моделі якості базуються на наборі характеристик, які вважаються важливими для якості програмного забезпечення. До характеристик, визначених у стандарті, належать Функціональна придатність (Functional Suitability), Надійність (Reliability), Зручність використання (Usability), Ефективність (Performance Efficiency), Ремонтопридатність (Maintainability), Портативність (Portability), Безпека (Security) і сумісність (Compatibility). Кожна характеристика поділяється на підхарактеристики, які надають більш детальну інформацію про характеристики. Комплексний набір моделей якості, які можна використовувати для оцінювання якості програмних продуктів і систем за низкою різних характеристик і підхарактеристик зазначений на рис. 2.1.

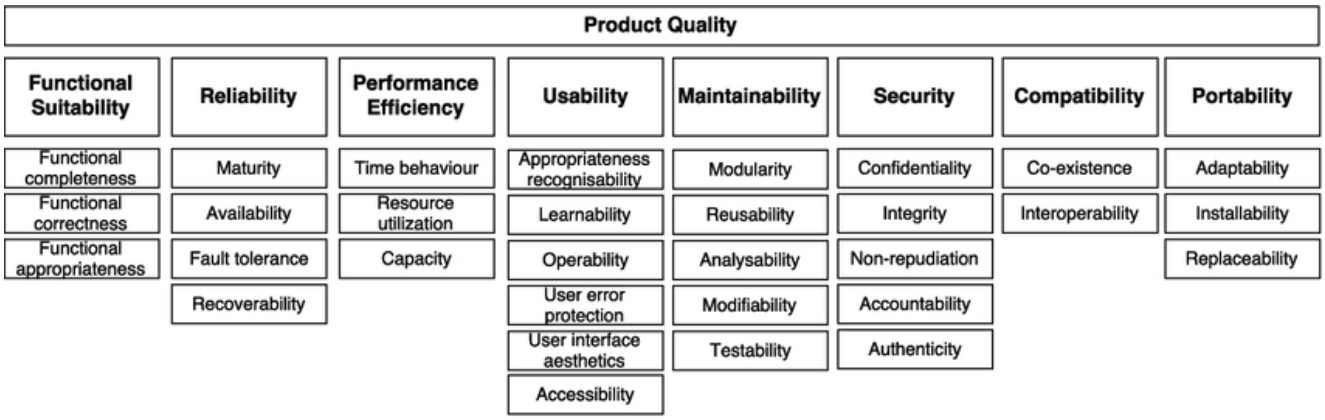


Рис. 2.1. - Модель якості програмного продукту на основі ISO/IEC 25010

Для детального дослідження функціональності моделі якості програмного продукту на основі стандарту ISO/IEC 25010, розглянемо кожну з характеристик окремо:

1. Функціональна придатність (Functional Suitability):

Функціональна придатність є однією з характеристик якості, визначених у стандарті ISO/IEC 25010 [10]. Вона демонструє ступінь, до якого програмний продукт або система задовольняє визначеним функціональним вимогам і виконує свої функції ефективно та результативно.

Функціональна придатність фокусується на функціональності та можливостях програмного забезпечення, а також на тому, наскільки добре вони відповідають потребам та очікуванням користувача. Вона оцінює, чи здатне програмне забезпечення виконувати необхідні завдання та забезпечувати бажані результати.

Для оцінки функціональної придатності програмного продукту розглядаються різні фактори, такі як:

- Повнота: Програмне забезпечення повинно надавати всі необхідні функції та можливості, як зазначено у вимогах;
- Правильність: Програмне забезпечення повинно точно виконувати передбачені функції без помилок і невідповідностей;
- Відповідність: Програмне забезпечення повинно відповідати відповідним стандартам, нормам або галузевим вимогам;
- Інтероперабельність: Програмне забезпечення повинно мати можливість безперешкодно взаємодіяти та працювати з іншими системами або компонентами;
- Безпека: Програмне забезпечення має забезпечувати конфіденційність, цілісність і доступність даних, а також захищати від несанкціонованого доступу або атак;
- Обробка помилок: Програмне забезпечення повинно коректно обробляти помилки, винятки та невірні дані, надаючи відповідний зворотній зв'язок та відновлюючись після збоїв;
- Продуктивність: Програмне забезпечення має виконувати свої функції ефективно, з прийнятним часом відгуку та можливостями обробки;
- Зручність використання: Програмне забезпечення має бути зручним для користувача, з інтуїтивно зрозумілим інтерфейсом та чіткими інструкціями, що дозволяє користувачам легко розуміти та орієнтуватися в його функціоналі.

Функціональна повнота, функціональна правильність та функціональна придатність - це три важливі підхарактеристики функціональної придатності в стандарті ISO/IEC 25010. Давайте розглянемо кожну з цих підхарактеристик більш детально:

1. Функціональна повнота: Ця підхарактеристика оцінює, чи включає програмне забезпечення всі необхідні функції та можливості, необхідні для задоволення визначених вимог та потреб користувача. Вона оцінює, чи надає програмне забезпечення повний набір функціональних можливостей без суттєвих прогалин або упущень. Функціонально повна програмна система гарантує, що всі очікувані функції доступні та належним чином реалізовані.
2. Функціональна коректність: Функціональна коректність фокусується на точності та правильності функціональної поведінки програмного забезпечення. Вона оцінює, чи виконує програмне забезпечення свої функції точно, без помилок і відхилень. Функціональна коректність гарантує, що програмне забезпечення видає очікувані результати і поводиться так, як зазначено у його функціональних вимогах. Вона передбачає перевірку програмного забезпечення на відповідність його функціональним специфікаціям і гарантує, що воно поводиться послідовно в різних сценаріях.
3. Функціональна придатність: Функціональна придатність перевіряє, чи функції та можливості програмного забезпечення є придатними та застосовними для задоволення конкретних потреб користувачів та зацікавлених сторін. Вона оцінює релевантність, сумісність та адаптивність програмного забезпечення для виконання його цільового призначення. Функціональна придатність гарантує, що програмне забезпечення забезпечує необхідну функціональність для підтримки завдань, робочих процесів і вимог користувачів.

Враховуючи ці підхарактеристики функціональної придатності, команди розробників програмного забезпечення можуть зосередитися на забезпеченні того, щоб програмне забезпечення не тільки включало всі необхідні функції, але й виконувало їх точно і відповідало конкретним потребам користувачів і зацікавлених сторін.

Оцінка функціональної придатності включає різні методи, такі як функціональне тестування, аналіз вимог та відгуки користувачів. Вона спрямована на те, щоб переконатися, що програмне забезпечення відповідає функціональним потребам користувачів і ефективно виконує свої функції.

Зосереджуючись на функціональній придатності, програмні продукти можуть надавати бажану функціональність, відповідати очікуванням користувачів і забезпечувати задовільний користувацький досвід.

2. Надійність (Reliability):

Надійність - це важлива характеристика якості програмного продукту, яка фокусується на здатності програмного забезпечення виконувати свої функції за певних умов протягом певного періоду часу. Вона вимірює здатність програмного забезпечення працювати без збоїв і надавати узгоджені результати протягом певного часу. Характеристика охоплює кілька ключових аспектів, такі як:

- Доступність: Доступність означає готовність програмного забезпечення до використання за потреби. Вона оцінює ступінь, до якого програмне забезпечення є доступним і робочим для користувачів. Надійна програмна система повинна мати високий рівень доступності, мінімізуючи час простою та забезпечуючи постійну доступність для використання [10].
- Відмовостійкість: Відмовостійкість вимірює здатність програмного забезпечення продовжувати працювати належним чином навіть за наявності збоїв або помилок. Вона включає в себе стійкість програмного забезпечення до непередбачуваних ситуацій, таких як апаратні збої або зовнішні перебої. Надійна програмна система повинна бути спроектована таким чином, щоб виявляти, ізолювати та відновлюватись після збоїв, щоб підтримувати свою функціональність.
- Відновлюваність: Відновлюваність фокусується на здатності програмного забезпечення відновлювати свою нормальну функціональність після збою або перебоїв. Вона включає такі функції, як механізми резервного копіювання та відновлення, обробку помилок і процедури відновлення. Надійна програмна система повинна бути здатна відновлюватися після збоїв і повертатися до стабільного стану з мінімальним впливом на продуктивність і цілісність даних.
- Зростання надійності: Зростання надійності - це показник того, як надійність програмного забезпечення покращується з часом завдяки виправленню помилок та вдосконаленню. Це передбачає моніторинг та аналіз виникнення збоїв, виявлення їх першопричин та впровадження коригувальних дій для підвищення

надійності програмного забезпечення. Зростання надійності спрямоване на постійне поліпшення стабільності програмного забезпечення та зменшення ймовірності збоїв.

Звертаючись до підхарактеристик надійності, розробники та тестувальники програмного забезпечення можуть прагнути до створення програмних систем, які є високодоступними, стійкими до збоїв, здатними відновлюватися після збоїв і демонструвати постійне покращення надійності з часом.

3. Ефективність (Performance Efficiency):

Ефективність - це характеристика якості програмного забезпечення, яка фокусується на здатності програмного забезпечення виконувати свої функції ефективно та результативно, оптимально використовуючи при цьому обчислювальні ресурси. Вона охоплює різні аспекти, пов'язані з продуктивністю програмного забезпечення та використанням ресурсів. Дана характеристика включає такі ключові елементи:

- Поведінка у часі (Time Behavior): Поведінка в часі стосується здатності програмного забезпечення швидко реагувати та надавати своєчасні результати [10]. Вона вимірює такі фактори, як час відгуку, швидкість обробки та пропускну здатність. Програмна система з високою ефективністю роботи повинна мінімізувати час відгуку і надавати користувачам швидкі та точні результати.
- Використання ресурсів (Resource Utilization): Використання ресурсів оцінює, наскільки ефективно програмне забезпечення використовує системні ресурси, такі як процесор, пам'ять і пропускну здатність мережі. Це передбачає оптимізацію споживання ресурсів для забезпечення ефективної роботи та уникнення непотрібних вузьких місць. Продуктивна програмна система повинна ефективно використовувати ресурси, досягаючи при цьому бажаного рівня продуктивності.
- Місткість (Capacity): Місткість вимірює здатність програмного забезпечення обробляти зростаючі робочі навантаження або великі обсяги даних без погіршення продуктивності. Це передбачає оцінку масштабованості програмного забезпечення та його здатності задовольняти зростаючі вимоги. Ефективна

програмна система повинна бути масштабованою, дозволяючи збільшувати потужність за потреби.

Ефективність фокусується на досягненні бажаного рівня продуктивності з мінімальним споживанням ресурсів. Вона передбачає оптимізацію алгоритмів, структур даних і практик кодування для максимізації обчислювальної ефективності. Ефективна програмна система повинна виконувати свої функції з мінімальними витратами ресурсів і досягати високої продуктивності [10].

Беручи до уваги підхарактеристики ефективності, розробники програмного забезпечення можуть створювати системи, які забезпечують швидкий час відгуку, ефективно використовують ресурси, справляються зі зростаючими робочими навантаженнями і досягають високої обчислювальної ефективності.

4. Зручність використання (Usability):

Зручність використання - це характеристика якості програмного забезпечення, яка фокусується на простоті використання та задоволеності користувачів при взаємодії з програмною системою. Зручність використання передбачає розробку інтерфейсів та взаємодії програмного забезпечення таким чином, щоб користувачі могли виконувати свої завдання ефективно, результативно та із задоволенням. Характеристика включає наступні підхарактеристики:

- Здатність до навчання (Learnability): Здатність до навчання стосується того, наскільки легко користувачі можуть навчитися користуватися програмною системою. Це передбачає надання чітких та інтуїтивно зрозумілих користувацьких інтерфейсів, добре організованої інформації та зручних для користувача елементів дизайну. Зручна програмна система повинна мати коротку криву навчання, що дозволяє користувачам швидко зрозуміти її функціональність і можливості.
- Функціональність (Operability): Функціональність пов'язана з легкістю експлуатації та управління програмною системою [10]. Вона включає такі характеристики, як чітка навігація, узгодженість елементів користувацького інтерфейсу та логічна організація функцій. Функціональна програмна система

повинна дозволяти користувачам безперешкодно взаємодіяти з нею та ефективно виконувати свої завдання.

- Захист від помилок користувача (User Error Protection): Захист від помилок користувача зосереджений на запобіганні або мінімізації помилок, зроблених користувачами під час взаємодії з програмним забезпеченням. Він передбачає включення механізмів запобігання помилкам, надання змістовних повідомлень про помилки та пропонування варіантів виправлення помилок. Програмна система з хорошим захистом від помилок користувача повинна допомагати користувачам уникати помилок і надавати чіткі вказівки щодо їх виправлення [10].
- Естетика інтерфейсу користувача (User Interface Aesthetics) : Естетика користувацького інтерфейсу стосується візуальної привабливості та привабливості програмної системи. Вона включає в себе розгляд таких елементів, як кольорові схеми, дизайн макета, типографіку та графіку для створення візуально приємних інтерфейсів. Програмна система з гарним естетичним інтерфейсом підвищує залученість та задоволеність користувачів.
- Доречна впізнаваємість (Appropriateness Recognizability): Ступінь, до якого програмна система є впізнаваною та знайомою для користувачів у контексті її використання.
- Доступність (Accessibility): Ступінь, до якого програмна система може бути доступною та використовуваною людьми з обмеженими можливостями.

Беручи до уваги підхарактеристики зручності використання, розробники програмного забезпечення можуть створювати інтерфейси та взаємодії, які є зручними для користувача, легкими у вивченні, ефективними в роботі та візуально привабливими. Це покращує загальний користувацький досвід і робить внесок у зручність використання програмної системи.

5. Ремонтопридатність (Maintainability):

Ремонтопридатність - це характеристика якості програмного забезпечення, яка фокусується на легкості, з якою програмна система може бути модифікована, адаптована, розширена та відремонтована. Вона оцінює здатність системи ефективно та

результативно зазнавати змін протягом свого життєвого циклу. Ремонтопридатність програмної системи має вирішальне значення для довгострокового успіху та стабільності, оскільки вона безпосередньо впливає на вартість і зусилля, необхідні для діяльності з обслуговування.

Характеристика охоплює кілька ключових аспектів, таких як:

- Аналітичність (Analyzability): Легкість, з якою проблеми або дефекти в програмній системі можуть бути виявлені за допомогою методів аналізу, таких як налагодження або аналіз першопричин. Це передбачає наявність відповідних інструментів і методів для підтримки процесу аналізу.
- Модифікованість (Modifiability): Легкість, з якою програмна система може бути модифікована або вдосконалена без внесення помилок або неочікуваних побічних ефектів. Включає такі фактори, як модульна структура системи, чіткі та зрозумілі інтерфейси, а також дотримання стандартів кодування.
- Тестування (Testability): Легкість, з якою програмна система може бути підтверджена та перевірена за допомогою тестування. Включає такі фактори, як наявність тестових даних, тестових середовищ і можливість ізолювати та маніпулювати компонентами системи з метою тестування.
- Модульність (Modularity): Модульність означає ступінь поділу програмної системи на окремі, автономні модулі або компоненти. Вона передбачає декомпозицію системи на менші, цілісні одиниці, які можна розробляти, модифікувати та тестувати незалежно [10]. Модульна архітектура сприяє підтримці, дозволяючи вносити зміни в окремі модулі, не впливаючи на всю систему. Це дозволяє розробникам розуміти і змінювати окремі компоненти без глибоких знань всієї системи.
- Багаторазовість використання (Reusability): Повторне використання - це можливість повторно використовувати програмні компоненти або модулі в різних контекстах або додатках. Вона передбачає розробку та впровадження компонентів, які є типовими, добре задокументованими та легко адаптуються для повторного використання. Використовуючи багаторазові компоненти, розробники можуть заощадити час і зусилля, не вигадуючи велосипед, а

зосередившись на інтеграції та налаштуванні існуючих компонентів. Багаторазовість підвищує зручність обслуговування за рахунок зменшення надмірності, покращення узгодженості та полегшення оновлення і вдосконалення в різних системах.

Зосереджуючи увагу на супроводжуваності під час проектування та розробки програмного забезпечення, організації можуть забезпечити гнучкість, адаптивність та легкість обслуговування своїх систем з часом. Це призводить до зменшення витрат на обслуговування, швидшого виправлення помилок та покращення загальної якості системи.

6. Безпека (Security):

Безпека - це критична характеристика якості програмних систем, яка забезпечує захист даних, ресурсів і функціональності від несанкціонованого доступу, зловмисних атак та інших загроз безпеці. Вона охоплює різні аспекти, які сприяють загальній безпеці програмної системи:

- Конфіденційність (Confidentiality): Конфіденційність стосується захисту конфіденційної інформації від несанкціонованого розголошення. Вона передбачає контроль доступу до даних і гарантування того, що тільки уповноважені особи або організації можуть мати доступ до конфіденційної інформації та переглядати її.
- Цілісність (Integrity): Цілісність означає достовірність і надійність даних, а також впевненість у тому, що вони не були підроблені або змінені несанкціонованим чином. Підтримка цілісності даних гарантує, що інформація залишається точною, послідовною та надійною протягом усього її життєвого циклу.
- Аутентифікація (Authenticity): Аутентифікація - це процес перевірки особи користувачів або організацій, які намагаються отримати доступ до програмної системи. Зазвичай вона передбачає використання імен користувачів, паролів, цифрових сертифікатів або біометричних заходів для підтвердження автентичності фізичних або юридичних осіб.
- Підзвітність (Accountability): Підзвітність означає можливість відстежити дії або події до відповідальних за них суб'єктів. Це гарантує, що фізичні або юридичні

особи можуть нести відповідальність за свої дії в програмній системі. Механізми підзвітності, такі як журнали аудиту та автентифікація користувачів, допомагають відслідковувати та пов'язувати дії з конкретними користувачами, уможливлюючи підзвітність за діяльність системи.

- Невідмова від відповідальності (Non-Repudiation): Відмова від відповідальності гарантує, що сторони, які беруть участь у комунікації або транзакції, не можуть заперечувати свою участь або автентичність своїх дій. Вона надає докази того, що дії були виконані, і не дозволяє сторонам заперечувати свою відповідальність. Механізми неспростування, такі як цифрові підписи та журнали транзакцій, допомагають встановити автентичність та цілісність транзакцій.

Ці аспекти безпеки мають вирішальне значення для захисту програмних систем від різних загроз, включаючи несанкціонований доступ, витік даних, витік інформації та зловмисні дії. Завдяки впровадженню надійних заходів і практик безпеки, програмні системи можуть створити безпечне середовище для даних і операцій, вселяючи довіру і впевненість у користувачів і зацікавлених сторін.

7. Сумісність (Compatibility):

Сумісність - це здатність програмної системи ефективно працювати з іншим програмним забезпеченням, обладнанням або системами. Вона гарантує, що система може адаптуватися до різних середовищ розгортання без погіршення продуктивності або обмежень функціональності [20].

Є дві важливі підхарактеристики сумісності, які необхідно враховувати: співіснування та інтероперабельність.

- Співіснування: Співіснування означає здатність програмної системи ефективно працювати в присутності інших програмних систем або компонентів. Це означає, що програмне забезпечення може співіснувати з іншими додатками або сервісами в одній системі, не спричиняючи конфліктів або збоїв у роботі. Сюди входить уникнення конфліктів ресурсів, проблем сумісності або втручання у функціональність інших програмних компонентів.
- Інтероперабельність: Інтероперабельність - це здатність програмної системи ефективно взаємодіяти та обмінюватися інформацією з іншими системами або

компонентами. Вона зосереджена на безперешкодній інтеграції та комунікації між різними програмними системами, платформами або технологіями. Інтероперабельність дозволяє програмному забезпеченню обмінюватися даними, послугами або функціональністю із зовнішніми системами, що дозволяє створювати спільні або розподілені середовища.

Забезпечення сумісності має вирішальне значення для полегшення безперешкодної інтеграції, уникнення конфліктів або несумісності та забезпечення узгодженої роботи користувачів на різних платформах або системах.

8. Портативність (Portability):

Портативність - ще одна важлива характеристика якості програмного забезпечення, яка фокусується на легкості, з якою програмна система може бути перенесена з одного середовища в інше. Вона передбачає адаптивність програмного забезпечення до різних апаратних засобів, операційних систем або платформ без необхідності значних модифікацій або переробки. У контексті портативності є кілька важливих підхарактеристик, які сприяють аналізу та оцінці портативності програмної системи:

- Адаптивність (Adaptability): Адаптивність означає здатність програмного забезпечення бути легко модифікованим або пристосованим до різних цільових середовищ або платформ [10]. Вона передбачає врахування таких факторів, як варіанти конфігурації, сумісність з різними апаратними чи програмними конфігураціями, а також легкість внесення змін для пристосування до різних вимог.
- Можливість встановлення (Installability): Інстальованість фокусується на простоті та ефективності встановлення та налаштування програмної системи на цільовій платформі. Вона охоплює такі аспекти, як зрозумілість інструкцій з інсталяції, наявність інсталяційних пакетів чи скриптів, а також загальну простоту процесу інсталяції.
- Замінність (Replaceability): Замінність пов'язана з легкістю, з якою програмна система може бути замінена або замінена альтернативним рішенням без значних порушень або впливу. Вона передбачає врахування таких факторів, як сумісність

форматів даних, можливість перенесення даних з однієї системи в іншу та загальні зусилля, необхідні для переходу з однієї програмної системи на іншу.

Враховуючи ці підхарактеристики, розробники та архітектори програмного забезпечення можуть оцінити та покращити портативність своїх систем, роблячи їх більш адаптивними, встановлюваними, замінними, здатними до співіснування та взаємодії на різних платформах та середовищах.

Отже, стандарт ISO/IEC 25010 надає комплексну модель якості програмного продукту, яка охоплює різні підхарактеристики та критерії для аналізу та оцінки якості програмних систем. Ця модель слугує цінною основою для визначення та вимірювання атрибутів якості програмних продуктів, дозволяючи організаціям встановлювати чіткі вимоги до якості та оцінювати продуктивність своїх систем.

Модель якості, викладена в ISO/IEC 25010, охоплює широкий спектр характеристик якості. Кожна характеристика поділяється на підхарактеристики, пропонуючи детальний та структурований підхід до оцінювання якості програмного забезпечення.

Використовуючи цю модель якості, організації можуть ефективно визначати та встановлювати пріоритети своїх вимог до якості на основі конкретного контексту та потреб користувачів. Вони можуть оцінювати програмний продукт за відповідними підхарактеристиками та критеріями, допомагаючи гарантувати, що програмне забезпечення відповідає бажаним рівням якості та продуктивності.

Прийняття моделі якості ISO/IEC 25010 в якості еталону дозволяє організаціям встановити спільну мову та розуміння якості програмного забезпечення. Це сприяє ефективній комунікації між зацікавленими сторонами, підтримує процеси прийняття рішень та спрямовує вдосконалення програмних систем протягом їхнього життєвого циклу.

2.2. Визначення вимог якості архітектури.

Визначення вимог до якості архітектури є критично важливим завданням при розробці програмного забезпечення. Воно передбачає визначення конкретних атрибутів якості та характеристик, які повинна мати архітектура, щоб забезпечити бажаний рівень якості програмного забезпечення. Процес визначення цих вимог включає кілька ключових кроків:

1. Розуміння потреб зацікавлених сторін. Важливо взаємодіяти із зацікавленими сторонами та розуміти їхні потреби, очікування та пріоритети щодо програмної системи. Це включає в себе розгляд перспектив кінцевих користувачів, клієнтів, керівництва, команди розробників та інших зацікавлених сторін.
2. Аналіз функціональних і нефункціональних вимог. Функціональні вимоги визначають, що система повинна робити, в той час як нефункціональні вимоги визначають аспекти якості та обмеження [1]. Аналіз обох типів вимог допомагає визначити атрибути якості, які є критично важливими для архітектури.
3. Визначення пріоритетності атрибутів якості. Після того, як відповідні атрибути якості визначені, їх потрібно розставити у порядку пріоритетності на основі їх важливості та впливу на систему. Це передбачає врахування таких факторів, як бізнес-цілі, потреби користувачів, галузеві стандарти та обмеження проекту.
4. Визначення сценаріїв атрибутів якості. Сценарії атрибутів якості описують конкретні ситуації або умови, в яких будуть оцінюватися атрибути якості архітектури. Ці сценарії допомагають зрозуміти бажану поведінку, продуктивність та якості, які очікуються від архітектури.
5. Застосування моделей якості. Моделі якості, такі як ISO/IEC 25010, забезпечують основу для визначення та оцінки характеристик якості програмного забезпечення. Ці моделі класифікують атрибути якості на різні підхарактеристики та надають рекомендації щодо їх вимірювання та оцінювання [11].
6. Проведення аналізу компромісів. У деяких випадках через суперечливі вимоги до якості або обмеженість ресурсів може знадобитися пошук компромісів. Аналіз

компромісів передбачає оцінку впливу різних архітектурних рішень на атрибути якості та прийняття обґрунтованих рішень на основі цілей і обмежень проекту.

7. Ітеративне вдосконалення. Визначення вимог до якості архітектури є ітеративним процесом [1]. У міру просування проекту і отримання нових знань, вимоги до якості можуть потребувати уточнення або коригування, щоб забезпечити їх відповідність потребам зацікавлених сторін, що змінюються.

Дотримуючись цих кроків, архітектори програмного забезпечення можуть встановити чіткі та вимірювані вимоги до якості архітектури. Це допомагає керувати проектними рішеннями, гарантуючи, що отримана архітектура буде добре пристосована для досягнення бажаних цілей якості програмного забезпечення.

Серед усіх вище перерахованих кроків, слід виділити «Застосування моделей якості», де основу задає стандарт якості ISO/IEC 25010. Для розробки комплексної моделі якості для архітектури програмного забезпечення, подібної до моделі якості, описаної в ISO 25010, дуже важливо встановити всеосяжний та стандартизований набір характеристик якості архітектури. Ці характеристики повинні відповідати рекомендаціям, наданим в ISO 25030 та певним настановам.

ISO/IEC 25030 - це стандарт у галузі інженерії програмного забезпечення, який фокусується на вимогах до якості та оцінюванні програмних продуктів і систем. Зокрема, ISO/IEC 25030 розглядає вимоги до якості програмних продуктів та надає настанови щодо оцінювання їхньої якості. Також, цей стандарт є частиною сімейства стандартів SQuaRE (SoftwareProductQualityRequirementsandEvaluation – вимоги та оцінювання якості програмного продукту).

Стандарт підкреслює важливість розуміння та визначення характеристик і підхарактеристик якості, які мають відношення до програмного продукту. Він забезпечує основу для ідентифікації, визначення та вимірювання цих характеристик якості, які включають такі аспекти, як функціональність, надійність, зручність використання, продуктивність, безпека, ремонтопридатність та портативність.

ISO/IEC 25030 також описує методи та прийоми оцінювання якості програмних продуктів за цими визначеними характеристиками. Він містить настанови щодо вибору

відповідних методів оцінювання, визначення критеріїв оцінювання та проведення процесу оцінювання.

Дотримуючись настанов, викладених в ISO/IEC 25030, організації розробників програмного забезпечення можуть встановити системний підхід до визначення, оцінювання та покращення якості своїх програмних продуктів. Цей стандарт допомагає узгодити процес розробки з цілями якості та гарантувати, що програмне забезпечення відповідає потребам та очікуванням зацікавлених сторін.

Системні вимоги охоплюють широкий спектр елементів, таких як програмне забезпечення, апаратне забезпечення, дані, механічні підсистеми та організація бізнес-процесів. Ці вимоги надходять з різних джерел, зокрема від кінцевих користувачів, організацій та офіційних осіб. Стандарт ISO 25030 надає вказівки щодо того, як формулювати та збирати вимоги до програмного забезпечення, як показано на рис. 2.2.

Вимоги до програмних систем можна розділити на дві категорії: вимоги, що стосуються процесу розробки програмного забезпечення, і вимоги, специфічні для самої програмної системи. Вимоги до програмної системи охоплюють функціональні вимоги, вимоги до якості (пов'язані з властивостями системи) та вимоги до управління (пов'язані з відповідними властивостями).

Функціональні вимоги стосуються конкретних потреб і функціональних можливостей предметної області, включаючи функціональні вимоги, пов'язані з користувачем. Ці вимоги повинні відповідати бажаним цілям якості і, отже, можуть розглядатися як вимоги до якості. Вимоги до якості можуть також охоплювати архітектурні та структурні аспекти системи.

<table>
<tr><td rowspan="5">Вимоги до системи</td><td rowspan="4">Вимоги до розроблюваної ПС</td><td rowspan="3">Вимоги до ПС</td><td rowspan="2">Вимоги до властивостей ПС</td><td colspan="2">Функціональні вимоги</td></tr>
<tr><td rowspan="3">Вимоги якості ПС</td><td>Вимоги якості у використанні</td></tr>
<tr><td>Вимоги зовнішньої якості</td></tr>
<tr><td>Вимоги внутрішньої якості</td></tr>
<tr><td>Вимоги до пов'язаних властивостей</td><td colspan="2">Управлінські вимоги, включаючи, наприклад, ціну, час поставки, постачальника</td></tr>
<tr><td rowspan="2">Вимоги до розробки ПС</td><td colspan="3">Вимоги до процесу розробки</td></tr>
<tr><td colspan="3">Вимоги до організації розробки</td></tr>
<tr><td>Інші вимоги</td><td colspan="4">Включають, наприклад, вимоги до апаратного забезпечення, даних, механічних підсистем, систем управління бізнес-процесами тощо</td></tr>
</table>

Рис. 2.2. Категорії вимог до системи згідно ISO 25030

(вимоги до ПС виділено сірим кольором)

Як, видно, з рис. 2.2, існує ієрархія вимог якості: на верхньому рівні розташовані вимоги до системи, які визначають частину вимог до ПС, а інша частина є специфічними вимогами.

У стандарті визначено дві моделі якості: якість програмного продукту і якість у використанні. У першій моделі включені характеристики внутрішньої і зовнішньої якості ПС. Опис моделі якості продукту можна представити у вигляді виразу (2.1):

$$Q_{prod} = \left\{ H_i^{prod}, S_{ik}^{prod}, A_{ik}^{prod}, C_{ik}^{prod}, M_{ik}^{prod} \right\}, \quad (2.1)$$

де H_i^{prod} – і-та характеристика якості програмного продукту;

S_{ik}^{prod} – k-та підхарактеристика і-ї характеристики якості;

$A_{ik}^{prod}, C_{ik}^{prod}, M_{ik}^{prod}$ – k-й атрибут, обмеження атрибуту та метрика атрибуту для і-ї підхарактеристики якості продукту.

В кінцевому результаті, необхідно визначити множину всіх характеристик якості архітектури та категоризувати їх у відповідності до артефактів стандарту ISO 25010, тобто віднести до однієї із стандартних характеристик (підхарактеристик).

Таким чином, модель якості архітектури ПС буде записана наступним чином:

$$Q_A = \{H_i^A, S_{ik}^A, A_{ik}^A, C_{ik}^A, M_{ik}^A\}, \quad (2.2)$$

де $H^A = H^{std} \cup H^{nstd}$ характеристики якості архітектури, як об'єднання інтерпретованих стосовно архітектури стандартизованих H^{std} характеристик якості програмного продукту та нестандартизованих H^{nstd} характеристик якості архітектури;

S_{ik}^A – підхарактеристики характеристик якості архітектури;

$A_{ik}^A, C_{ik}^A, M_{ik}^A$ – це атрибути, обмеження та метрики підхарактеристик якості архітектури, які визначаються предметною областю, для якої розробляється програмна система.

Згодом, використовуючи експертні методи, з повного набору характеристик якості архітектури визначаються найбільш значущі характеристики якості для конкретної предметної області. Ці відібрані характеристики слугують основою для проведення порівняльної оцінки альтернативних архітектурних рішень для передбачуваної системи. Мета полягає у визначенні та виборі архітектурного рішення, яке найкраще відповідає системним вимогам.

Таким чином, на основі аналізу ISO 25030 можна зробити наступні ключові висновки:

1. Вимоги до якості архітектури програмного забезпечення повинні враховувати його інтеграцію в зовнішню систему, а на вимоги до якості програмної архітектури впливають вимоги до якості системи в цілому;
2. Збір вимог до ПА відбувається за участю багатьох зацікавлених сторін, що призводить до виникнення суперечливих вимог, які необхідно вирішувати шляхом компромісу для вирішення суперечливих вимог до якості;
3. Вимоги до якості представлення архітектури програмного забезпечення повинні бути чітко сформульовані на основі моделей якості, викладених в ISO 25010;
4. Вимоги до якості, визначені для архітектури програмної системи, відіграють вирішальну роль у визначенні бажаного рівня якості для програмної системи в цілому;

5. Чіткого та всебічного формулювання вимог можна досягти за допомогою моделей якості, які забезпечують структуровану основу для визначення та оцінювання якості програмного забезпечення;
6. Стандарт ISO 25010 передбачає використання моделей якості, підкреслюючи їх важливість для визначення та оцінювання вимог до якості програмного забезпечення.

2.3. «Будинок якості». Метод парних порівнянь (MAI).

Підхід "Будинок якості", також відомий як метод парних порівнянь (Method of Pairwise Comparisons, MAI) - це методологія, що використовується для оцінки та визначення пріоритетів характеристик якості в архітектурі програмного забезпечення. Систематично порівнюючи пари характеристик якості, цей підхід дозволяє призначати відносні ваги або пріоритети на основі їхньої сприйнятої важливості [13,14].

Концепція "Будинку якості" забезпечує структуроване представлення взаємозв'язків між різними характеристиками якості в програмній системі [13]. Вона складається з чотирьох основних компонентів: Фундамент, Функціональність, Якість системи та Контекст. Кожен компонент представляє окремий аспект якості програмної системи.

Для полегшення процесу оцінювання в рамках підходу "Будинок якості" застосовується метод парних порівнянь (MAI). Цей метод дозволяє зацікавленим сторонам порівнювати дві характеристики якості одночасно та визначати їхню відносну значущість. Завдяки систематичному оцінюванню та порівнянню можна встановити порядок пріоритетів або рейтинг, який керує процесом прийняття рішень в архітектурному проектуванні та розробці [14].

Підхід "Будинок якості", разом з методом парних порівнянь (MAI), забезпечує структуровану і систематичну основу для оцінки і визначення пріоритетів характеристик якості в архітектурі програмного забезпечення. Цей підхід допомагає приймати обґрунтовані рішення щодо розподілу ресурсів, компромісів і вибору дизайну для ефективного досягнення бажаних цілей якості програмної системи.

Метод передбачає порівняння пар характеристик якості та визначення їх відносної важливості. Зацікавленим сторонам, які беруть участь у процесі оцінювання, представляють дві характеристики якості одночасно і просять висловити свої переваги або судження щодо того, яка характеристика є більш важливою або релевантною. Цей процес повторюється для всіх можливих пар характеристик якості, в результаті чого формується матриця попарних порівнянь.

На основі зібраних даних парних порівнянь можна застосувати математичні методи, такі як метод аналітичної ієрархії (MAI) або метод впорядкування за схожістю з ідеальним рішенням (Technique for Order Preference by Similarity to Ideal Solution, TOPSIS), щоб отримати ваги або пріоритети для кожної характеристики якості [14]. Ці ваги відображають відносну важливість характеристик в контексті конкретної програмної системи, що оцінюється.

Метод парних порівнянь "Будинку якості" (MAI) допомагає зацікавленим сторонам краще зрозуміти відносну важливість різних характеристик якості в архітектурі програмного забезпечення. Він підтримує прийняття рішень, забезпечуючи систематичний і структурований підхід до оцінки та визначення пріоритетів цих характеристик на основі переваг зацікавлених сторін. Використовуючи цей метод, вибір архітектурного дизайну може бути зроблений з більш чітким розумінням бажаних цілей якості та пов'язаних з цим компромісів.

Концептуально "Будинок якості" має вигляд (рис. 2.3):

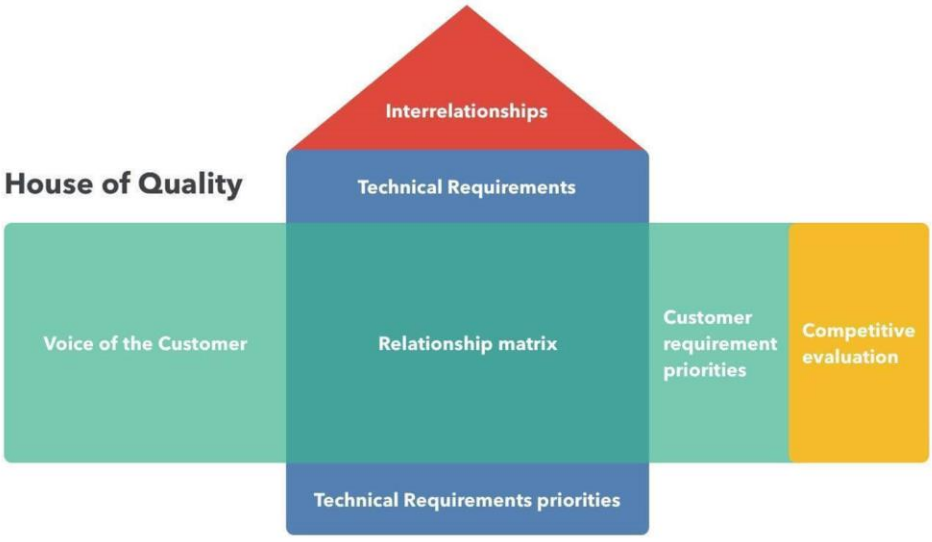


Рис. 2.3. "Будинок якості"

"Будинок якості" використовується, як візуальне представлення в методі розгортання функцій якості (Quality Function Deployment, QFD), показаний на рисунку 2.4. QFD - це системний підхід до розробки та вдосконалення продуктів або послуг шляхом забезпечення задоволення потреб та очікувань клієнтів [16]. Діаграма "Будинок якості" слугує інструментом для організації та визначення пріоритетів цих вимог замовника та їхнього зв'язку з конкретними елементами дизайну.

Цей підхід має на меті встановити комплексний набір системних вимог на основі потреб та вподобань користувача. Ліва частина будинку якості представляє вимоги користувача, тоді як верхня частина складається з системних вимог, виражених у технічних термінах. У межах будинку якості присвоюються числові значення для відображення рівня залежності між кожним елементом у верхньому рядку і елементами в лівій колонці. Ці значення в сукупності утворюють матрицю взаємозалежності або кореляційну матрицю.

Числові значення вибираються з набору {0, 3, 6, 9} і вказують на ступінь залежності. Наприклад, 0 означає незалежність між елементами, тоді як 9 вказує на те, що елемент у стовпчику повністю визначається відповідним елементом у рядку. На "даху" будинку якості зображено взаємозв'язки між елементами набору системних вимог, а для позначення ступеня взаємозалежності використовуються символи. Зокрема, символ ⊙ позначає сильно негативний зв'язок, вказуючи на те, що поліпшення одного параметра призведе до погіршення іншого. Символ ○ означає негативний зв'язок, × - позитивний зв'язок, а # - дуже позитивний зв'язок.

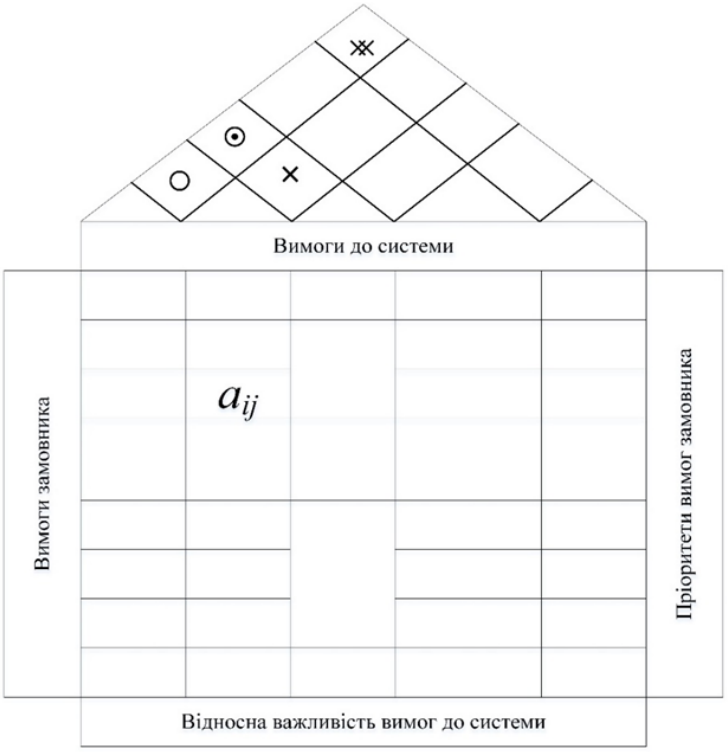


Рис. 2.4. "Будинок якості" методу QFD

Після заповнення кореляційної матриці експертами на основі пріоритетів вимог користувача (правий стовпець рис. 2.4) обчислюються відносні важливості вимог до системи як сума добутків елементів матриці та пріоритетів вимог користувача.

Таким чином, будинок якості в методі QFD забезпечує структуроване візуальне представлення, яке допомагає зафіксувати взаємозв'язки і залежності між вимогами користувача і системними вимогами [15]. Матриця взаємозалежності і символи на даху дають уявлення про ступінь і характер цих взаємозв'язків, допомагаючи в прийнятті рішень і визначенні пріоритетів під час процесу специфікації вимог.

Діаграма "Будинок якості" зазвичай складається з чотирьох основних компонентів або секцій, кожна з яких представляє окремий аспект якості. Цими компонентами є:

1. Вимоги замовника: Цей розділ представляє голос замовника, фіксуючи його потреби, очікування і бажані результати. Вимоги замовника зазвичай збирають за допомогою опитувань, інтерв'ю та інших методів дослідження ринку.
2. Технічні вимоги: Цей розділ фокусується на перетворенні вимог замовника в конкретні технічні характеристики або елементи дизайну, які можуть

задовольнити ці вимоги. Він включає в себе визначення критичних параметрів і функцій дизайну, необхідних для задоволення очікувань замовника.

3. Матриця взаємозв'язків: Матриця взаємозв'язків показує взаємозв'язок між вимогами замовника і технічними вимогами. Вона допомагає визначити ступінь важливості та впливу кожної технічної вимоги на задоволення потреб замовника. Ця матриця створюється шляхом попарних порівнянь та оцінок.
4. Технічне розгортання: Цей розділ ілюструє, як технічні вимоги розгортаються або розподіляються між різними компонентами або підсистемами продукту або послуги. Він показує, як елементи дизайну пов'язані між собою і сприяють загальному задоволенню клієнта.

Діаграма "Будинок якості" забезпечує структуровану основу для узгодження вимог замовника з технічними специфікаціями і проектними рішеннями. Вона полегшує комунікацію та співпрацю між різними зацікавленими сторонами, залученими до процесу розробки продукту, гарантуючи, що потреби замовника ефективно перетворюються на дієві елементи дизайну.

У випадку розробки програмної архітектури, справа у будинку якості записуються характеристики якості програмної системи $H_j^{ПС}$, а зверху – характеристики якості архітектури H_i^A (рис. 2.5).

	…	H_i^A	…	
		⋮		
$H_j^{ПС}$	…	a_{ij}	…	$p_j^{ПС}$
		⋮		
		w_i^A		

Рис. 2.5. "Будинок якості " для вибору характеристик якості архітектури

У клітинках таблиці "Будинок якості" експерти проставляють значення a_{ij}, які відображають ступінь впливу кожної характеристики якості архітектури на кожну характеристику якості програмного забезпечення a_{ij}.

Використовуючи алгоритм простого вибору, для кожної характеристики (підхарактеристики) якості ПС $H_j^{ПС}$ визначаються її пріоритети $p_j^{ПС}$.Згідно цього алгоритму початково визначимо ступінь переваги підхарактеристик якості програмної системи одна над одною згідно транзитивної шкали при основі 2. Слабка перевага позначатиметься коефіцієнтом 2, сильна – 4, дуже сильна – 8 та абсолютна перевага – 16 і більше.

Коли пронумерувати показники якості у використанні в порядку зростання, то тоді, до прикладу, коефіцієнт α2,1 =2 означатиме, що показник з номером 2 за своєю значимістю вдвічі переважає показник з номером 1. Через опитування експертів встановлюються всі значення коефіцієнтів переважання показників якості програмної системи один над одним. Потім цей вектор нормується до одиниці.

На останньому етапі розраховується коефіцієнт важливості (вага) для кожної характеристики якості архітектури в даній предметній області за формулою (вираз 2.3).

$$w_i^A = \sum_j a_{ij} \cdot p_j^{ПС} \tag{2.3}$$

Встановивши нижче порогове значення $w_{пор.}$ для розрахованих ваг характеристик якості архітектури, можна "відфільтрувати" менш значущі характеристики (вираз 2.4).

$$\{w_i^s\} \in \{w_i^{ПС}\} > w_{пор.}, \tag{2.4}$$

де $\{w_i^s\}$ – набір критеріїв якості архітектури, які будуть використовуватися для її порівняльної оцінки.

Цей крок на наступних етапах проектування програмної системи значно скоротить час, людські та матеріальні ресурси, оскільки не потрібно буде реалізовувати

всі характеристики якості архітектури, а лише найбільш значущі для конкретної програмної системи, що розробляється.

Встановивши цей набір критеріїв якості архітектури, можна буде отримати загальну оцінку кожної альтернативної архітектури шляхом розрахунку вагових коефіцієнтів критеріїв якості та визначити найкраще архітектурне рішення.

Підводячи підсумки, "Будинок якості" і метод розгортання функцій якості (QFD) є цінними підходами для оцінки і визначення пріоритетів характеристик якості в архітектурі програмного забезпечення.

Структура "Будинку якості" забезпечує структуроване і візуальне представлення взаємозв'язків між різними характеристиками якості. Вона допомагає організувати і зрозуміти взаємозалежності між вимогами користувачів і технічними вимогами. Систематично порівнюючи і присвоюючи вагу цим характеристикам, "Будинок якості" дозволяє приймати обґрунтовані рішення в архітектурному проектуванні.

Метод QFD, з іншого боку, пропонує систематичний процес перекладу вимог користувача в технічні вимоги. Створюючи матрицю QFD та оцінюючи взаємозв'язки, зацікавлені сторони можуть визначити пріоритетність технічних вимог на основі їхнього внеску у задоволення потреб користувачів. Цей метод полегшує визначення критичних сфер для вдосконалення, компромісів і прийняття рішень під час проектування та розробки.

Разом "Будинок якості" і метод QFD забезпечують комплексну основу для узгодження архітектури програмного забезпечення з потребами та очікуваннями користувачів. Вони дозволяють ефективно розподіляти ресурси, приймати обґрунтовані рішення та безперервно вдосконалюватись протягом усього життєвого циклу розробки. Застосовуючи ці методи, організації можуть покращити якість своїх програмних систем та підвищити задоволеність клієнтів.

РОЗДІЛ 3
СИСТЕМА ЗАБЕЗПЕЧЕННЯ ЯКОСТІ АРХІТЕКТУРИ

3.1. Проектування архітектури з використанням шаблонів. Альтернативні архітектури.

Проектування архітектури з використанням шаблонів передбачає комплексний набір сценаріїв, де *Розробник* створює основу для взаємодії цієї архітектури з другими елементами проекту.

Для відображення процесу проектування архітектури з використанням шаблонів побудуємо use-case діаграму (рис. 3.1).

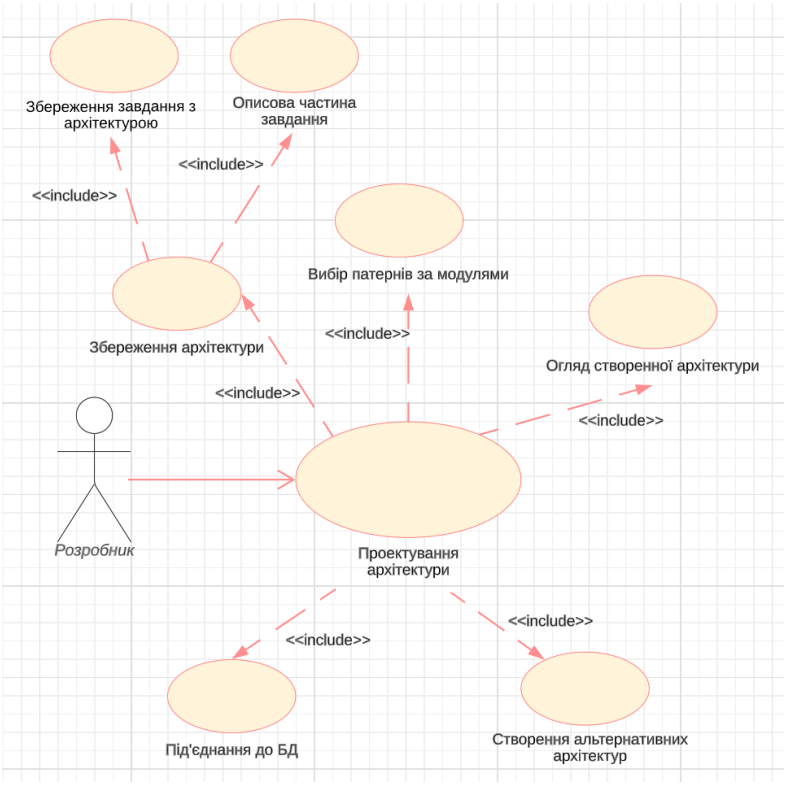


Рис. 3.1. Сценарій роботи *Розробника* по проектуванню архітектури з використанням шаблонів

Для проектування і подальшої побудови об’єктно-орієнтованої системи застосовано графічне моделювання з використанням діаграм UML. При розробці функціональних вимог системи використовувались – use case діаграми (діаграми сценаріїв).

Актор:

- Розробник – людина, що запускає процес проектування архітектури, обирає патерни для підстановки у модулі (категорії) шарів програмного додатку, вводить описову частину завдання, приймає конструктивні рішення по архітектурі програмного додатку.

Use case:

- Проектування архітектури – базовий use case, що представляє головну задачу підсистеми і викликає інші сценарії;
- Під'єднання до бази даних – вибір бази даних (репозиторію патернів) для подальшої роботи з нею;
- Створення альтернативних архітектур – створення альтернативних рішень, завдання для подальшої роботи *Архітектора*;
- Вибір патернів за модулями – вибір патернів для кожного модуля відповідного шару архітектури;
- Огляд створеної архітектури – огляд *Розробником* створеної архітектури програмного додатку перед збереженням;
- Збереження архітектури – виконання збереження сформованих рішень у вигляді завдання для подальшої роботи *Експертів*;
- Описова частина завдання;
- Збереження завдання з архітектурою.

Створення альтернативних архітектур, в свою чергу, предбачає залучення *Архітектора* до роботи над цим процесом. Програмний комплекс «Архітектор» необхідний для створення нових варіантів архітектур і їх оцінки, буде використаний в якості інструменту для досягнення цих цілей. Для відображення побудови альтернативних архітектур побудуємо use-case діаграму (рис. 3.2).

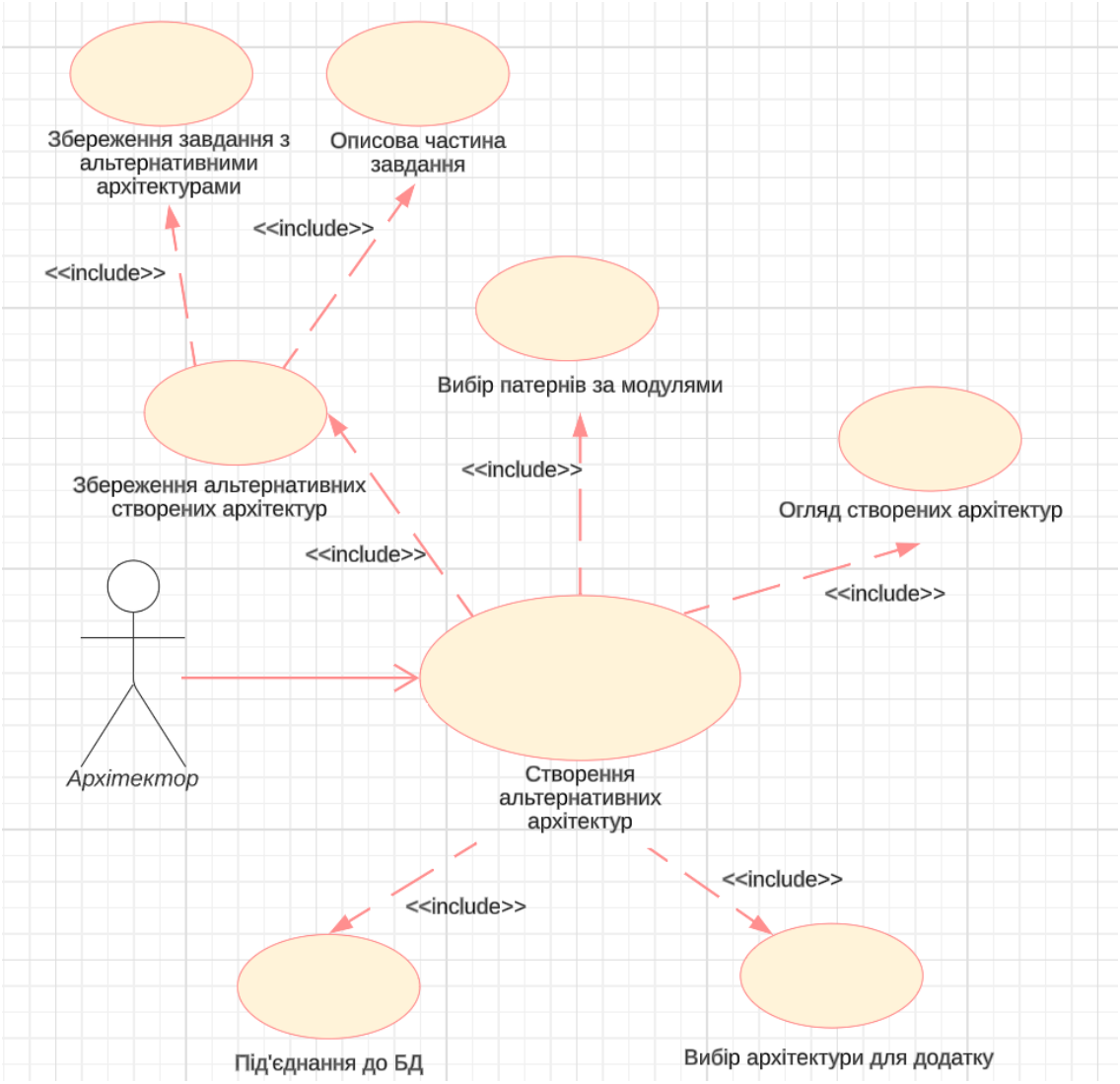


Рис. 3.2. Сценарій роботи *Архітектора* по проектуванню нових варіантів архітектур

Актор:

- *Архітектор* – людина, яка запускає процес створення альтернативних архітектур, обирає батьківську архітектуру та патерни для підстановки у модулі (категорії) шарів програмного додатку, вводить описову частину завдання, приймає конструктивні рішення по альтернативним архітектурам програмного додатку.

Use case:

- Створення альтернативних архітектур – базовий use case, що представляє головну задачу підсистеми і викликає інші сценарії;
- Під'єднання до БД – вибір бази даних (*репозиторію патернів*) для подальшої роботи з нею;
- Вибір архітектури для додатку – обрання батьківської архітектури для побудови альтернативних архітектур додатку;
- Вибір патернів за модулями – вибір патернів для кожного модуля відповідного шару архітектури та формування альтернатив;
- Огляд створених архітектур – огляд *Архітектором* створених альтернативних архітектур програмного додатку перед збереженням;
- Збереження альтернативних архітектур – виконання збереження сформованих альтернатив у вигляді завдання для подальшої роботи *Експертів*;
- Ввід описової частини завдання;
- Збереження завдання з альтернативними архітектурами додатків.

3.2. Оптимізація вибору архітектури. Програмний комплекс «Архітектор».

Програмний комплекс «Архітектор» складається з трьох основних програм:

- Програма створення альтернативних архітектур та попарної експерної оцінки їх;
- Програма перегляду оцінок;
- Програма виставлення критеріальних пріорітетів та прийняття рішення.

В системі створення альтернативних архітектур та попарної експерної оцінки існує три користувача:

- *Адміністратор* – додає у базу нові архітектури, шари, модулі та патерни та обслуговує базу даних архітектур і систему в цілому;
- *Архітектор* – комбінує патерни, для вирішення класів специфікованих задач шляхом формування наборів альтернативних архітектур;

- *Експерт* – оцінює сформовані архітектором множини альтернативних архітектур програмних додатків за певними визначеними критеріями якості.

Щоб покращити вибір програмної архітектури, необхідно його оптимізувати. Для оптимізації вибору архітектури побудуємо use-case діаграму (рис. 3.3) на основі сценарію роботи *Архітектора* по проектуванню нових варіантів архітектур (рис. 3.2).

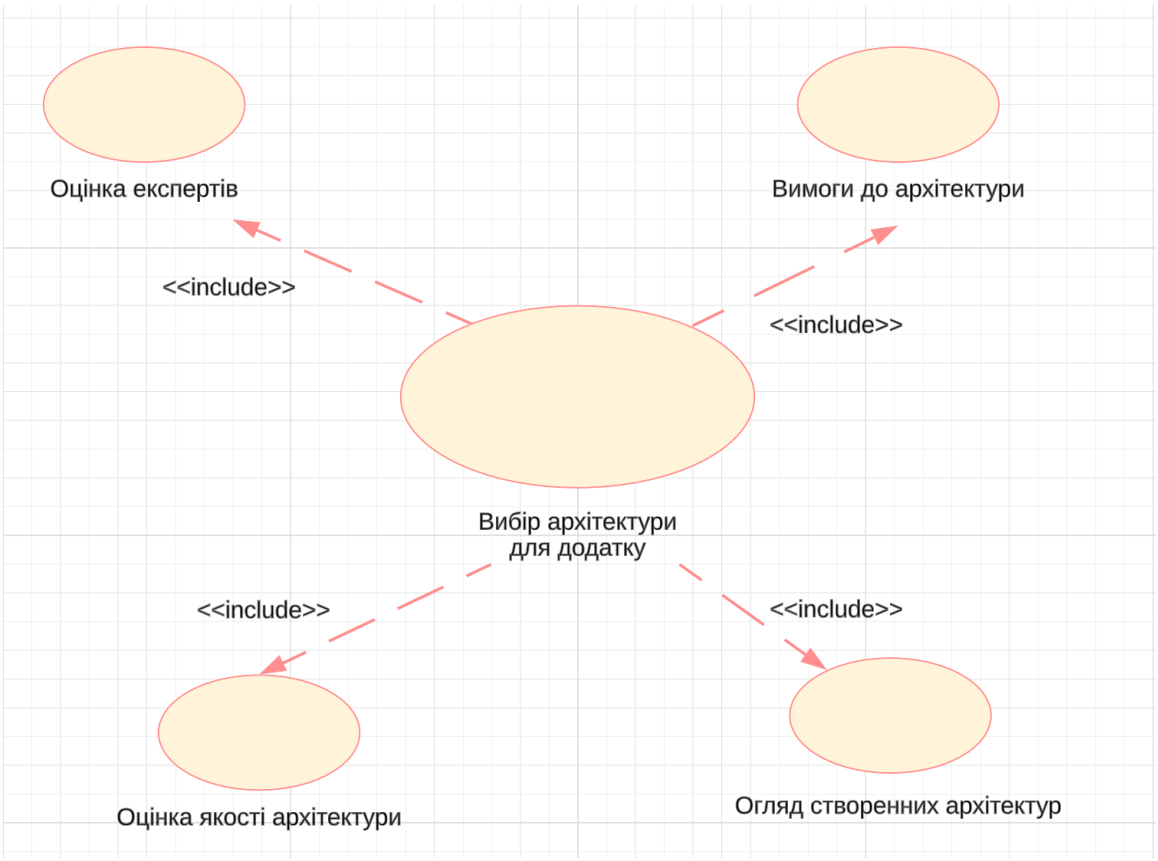


Рис. 3.3. Діаграма використання по оптимізації вибору архітектури

Use case:

- Вибір архітектури для додатку – базовий use case, що описує головну задачу підсистеми і викликає інші сценарії;
- Вимоги до архітектури – перегляд переліку вимог, необхідних для створення та оцінки архітектури;
- Оцінка експертів – підключення до програми перегляду експертних оцінок;
- Оцінка якості архітектури – оцінка якості архітектури згідно відповідного програмного забезпечення;

- Огляд створених архітектур – перегляд усіх створених варіантів архітектур.

Функціональні можливості *Програми перегляду експертних оцінок* представлені на діаграмі варіантів використання на рис. 3.4.

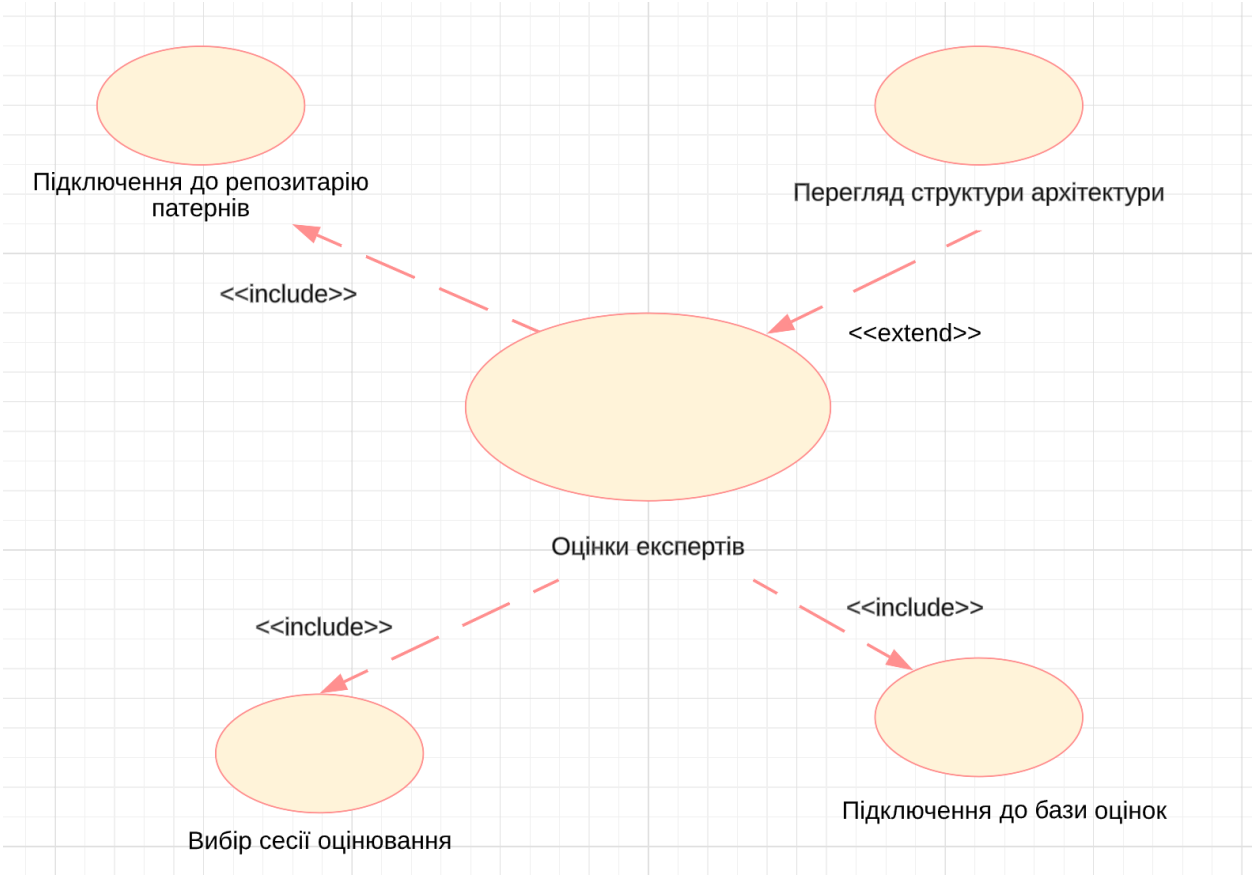


Рис. 3.4. Діаграма використання *Програми перегляду експертних оцінок*

Use case:

- Перегляд оцінок – базовий use case, що описує головну задачу програми і викликає інші сценарії;
- Підключення до репозитарію патернів – вибір бази даних (*репозиторію патернів архітектур*) для подальшої роботи з нею;
- Підключення до бази оцінок – вибір бази даних оцінок для подальшої роботи з нею;
- Перегляд структури архітектури – перегляд структури альтернативної архітектури;

- Вибір сесії оцінювання – вибір сесії оцінювання для візуалізації матриці попарних порівнянь.

3.3. Програма визначення вимог якості архітектури.

При розробці програмної архітектури, у розробників виникає необхідність оцінки вимог якості до неї. Для того щоб модифікувати програмний комплекс «Архітектор», розробимо додаткове програмне забезпечення. Мета полягає у створені програми для обчислення коефіцієнтів важливості для кожної із заданих характеристик якості архітектури та програмної системи, а також "відсіюванні" менш значущих характеристик. Це необхідно для скорочення часу, людських та матеріальних ресурсів, оскільки не доведеться затрачати сили на реалізацію всіх характеристик якості архітектури, а тільки на найбільш значимі в контексті використовуваної програмної системи.

Функціональні можливості *Програми визначення вимог якості архітектури*. Діаграма варіантів використання роботи *Програми визначення вимог якості архітектури* показана на рис. 3.5.

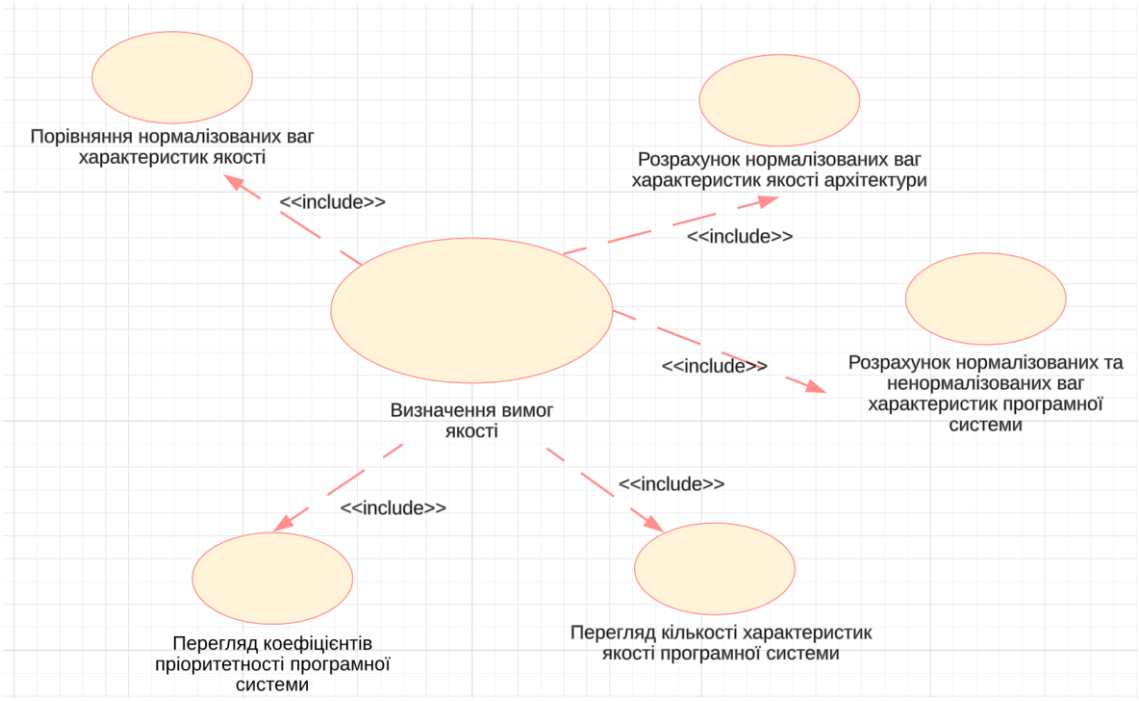


Рис. 3.5. Діаграма використання Програми визначення вимог якості архітектури

Use case:

- Визначення вимог якості – базовий use case, що описує головну задачу програми і викликає інші сценарії;
- Перегляд кількості характеристик якості програмної системи – перегляд кількості введених характеристик якості;
- Перегляд коефіцієнтів пріоритетності програмної системи – перегляд коефіцієнтів введених по кожній характеристиці якості;
- Розрахунок нормалізованих ваг характеристик якості архітектури – розрахунок нормалізованих ваг заданих характеристик;
- Розрахунок нормалізованих та ненормалізованих ваг характеристик програмної системи – розрахунок нормалізованих та ненормалізованих ваг заданих характеристик;
- Порівняння нормалізованих ваг характеристик якості – порівняння нормалізованих ваг, згідно введеного порогового значення.

Як описано в розділі 2, для розрахунку ваги характеристик якості треба визначити ступінь переваги підхарактеристик якості одна над одною згідно транзитивної шкали, де слабка перевага позначатиметься коефіцієнтом 2, сильна – 4, дуже сильна – 8 та абсолютна перевага – 16 і більше.

На початку, користувач може вибрати кількість характеристик якості архітектури та програмної системи, які треба оцінити. Далі, експертним методом користувач ПЗ вказує значення переваги характеристик. Наступним кроком, обчислюються коефіцієнти важливості для кожної характеристики за формулою (вираз 2.3).

ПЗ формує перелік характеристик та відповідні до них значення з коефіцієнтів важливості, де нормує значення до одиниці. Програма виводить на екран дані по вазі характеристик якості архітектури та програмної системи.

Наступною опцією буде вибір порогового значення для розрахованих ваг характеристик якості архітектури та ПС, щоб відсікти зайві характеристики (вираз 2.4). Після цього, ПЗ вирахує за формулою необхідні дані та виведе їх на екран.

Отже, для перевірки цього ПЗ, вираховуємо значення для вибору характеристик якості архітектури.

Для оцінки характеристик якості архітектури оберемо:

1. Модульність (Modularity)
2. Переносимість (Portability)
3. Розширюваність (Extensibility)

Та наступні вагові коефіцієнти програмної системи ($p_j^{ПС}$) для кожної характеристики програмної архітектури:

1. Модульність (Modularity): $p_1^{\text{ПС}} = 4$
2. Переносимість (Portability): $p_2^{\text{ПС}} = 3$
3. Розширюваність (Extensibility): $p_3^{\text{ПС}} = 2$

Розрахуємо ваги для кожної характеристики:

Для Модульності (Modularity):

$$w_1^{\text{A}} = a_{1,1} * p_1^{\text{ПС}} + a_{1,2} * p_2^{\text{ПС}} + a_{1,3} * p_3^{\text{ПС}} = a_{1,1} * 4 + a_{1,2} * 3 + a_{1,3} * 2$$

Для Переносимості (Portability):

$$w_2^{\text{A}} = a_{2,1} * p_1^{\text{ПС}} + a_{2,2} * p_2^{\text{ПС}} + a_{2,3} * p_3^{\text{ПС}} = a_{2,1} * 4 + a_{2,2} * 3 + a_{2,3} * 2$$

Для Розширюваності (Extensibility):

$$w_3^{\text{A}} = a_{3,1} * p_1^{\text{ПС}} + a_{3,2} * p_2^{\text{ПС}} + a_{3,3} * p_3^{\text{ПС}} = a_{3,1} * 4 + a_{3,2} * 3 + a_{3,3} * 2$$

Після отримання ваг для кожної характеристики, ми повинні пронормувати їх до одиниці:

$$w_{1norm}^{\text{A}} = w_1^{\text{A}} / \sum(\text{i})\ w_i^{\text{A}};$$
$$w_{2norm}^{\text{A}} = w_2^{\text{A}} / \sum(\text{i})\ w_i^{\text{A}};$$
$$w_{3norm}^{\text{A}} = w_3^{\text{A}} / \sum(\text{i})\ w_i^{\text{A}}.$$

За такими значеннями:

$$w_1^{\text{A}} = 3;$$
$$w_2^{\text{A}} = 4;$$
$$w_3^{\text{A}} = 2.$$

Сума ваг обчислюється наступним чином:

$$\sum(\text{i})\ w_i^{\text{A}} = w_1^{\text{A}} + w_2^{\text{A}} + w_3^{\text{A}} = 3 + 4 + 2 = 9.$$

Отже нормалізовані ваги для характеристик якості програмної архітектури обчислюються, як:

$$w^{A}_{1norm} = w^{A}_{1} / \sum(i)\, w^{A}_{i} = 3/9 = 1/3 \approx 0.333;$$

$$w^{A}_{2norm} = w^{A}_{2} / \sum(i)\, w^{A}_{i} = 4/9 \approx 0.444;$$

$$w^{A}_{3norm} = w^{A}_{3} / \sum(i)\, w^{A}_{i} = 2/9 \approx 0.222.$$

Отримані значення вказують на відносну важливість кожної характеристики в межах всього набору характеристик якості програмної архітектури. Порівняємо значення в програмі по визначенню вимог якості архітектури (рис. 3.6).

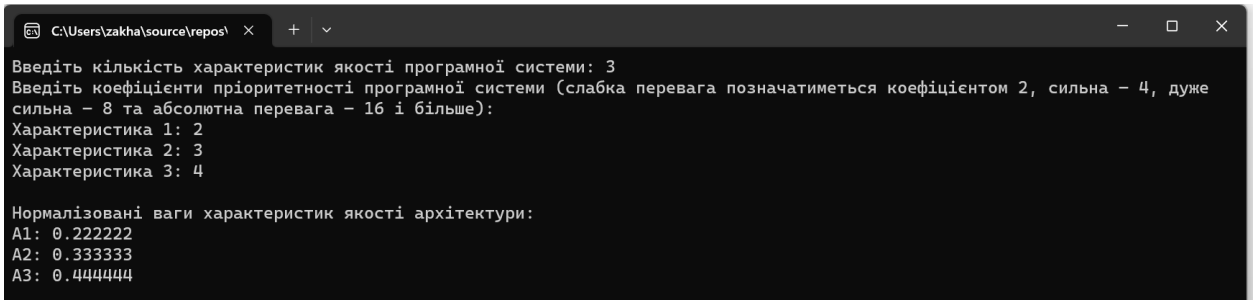


Рис. 3.6.1. Результат виконання обчислень програми по визначенню вимог якості архітектури

Як показано на рис. 3.6.1, програма визначає нормалізовані ваги характеристик якості архітектури за заданою пріоритетністю програмної системи. Наступним кроком, введемо значення характеристик якості програмної системи, де програма розраховує нормалізовані та ненормалізовані ваги характеристик якості ПС (рис. 3.6.2).

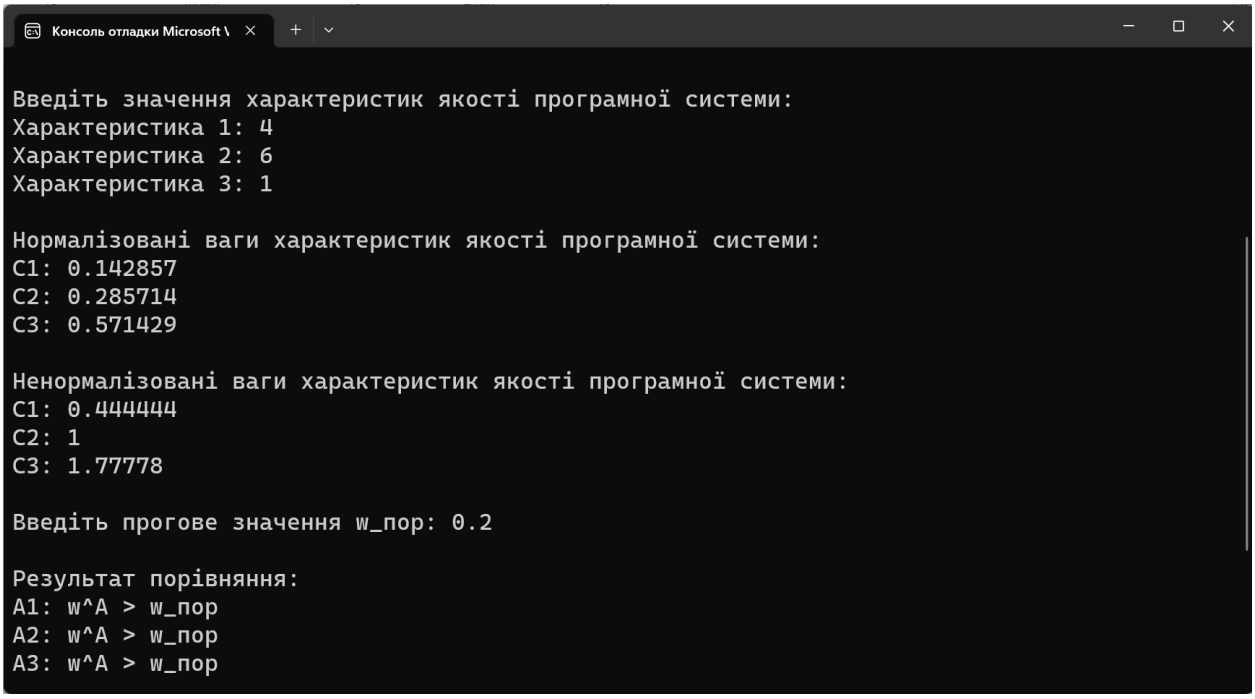
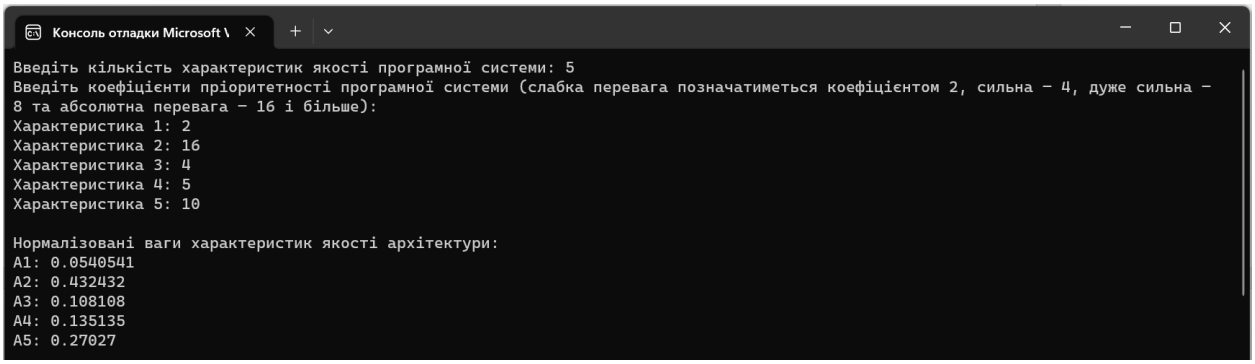


Рис. 3.6.2. Результат виконання обчислень значень ваг якості ПС та попарного порівняння

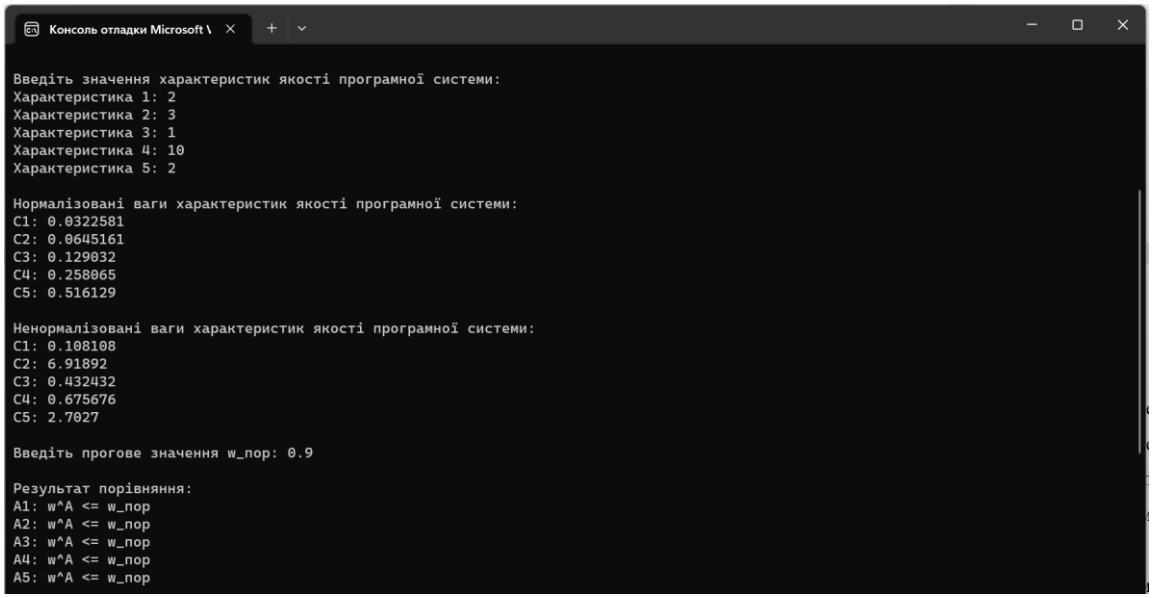
Також на рис. 3.6.2 в програму можна ввести порогове значення для попарного порівняння характеристик програмної системи та характеристик якості архітектури.

Щоб детально дослідити функціонал програми, візьмемо для прикладу п’ять характеристик якості архітектури та ПС. Отримаємо результати обчислень зображені на рис. 3.7.

Результат виконання обчислень рис. 3.7:



а.



b.

Рис. 3.7. a) Результат виконання обчислень значень нормалізованих ваг якості архітектури за п’ятьма характеристиками; b) Результат виконання обчислень значень ваг якості ПС та попарного порівняння за п’ятьма характеристиками.

Таким чином, за результатом порівняння на рис. 3.7 жодна з характеристик не задовольняє вимогам якості, через високе порогове значення. Якщо його зменшити, наприклад до 0.15, як показано на рис. 3.8, то ситуація зміниться.

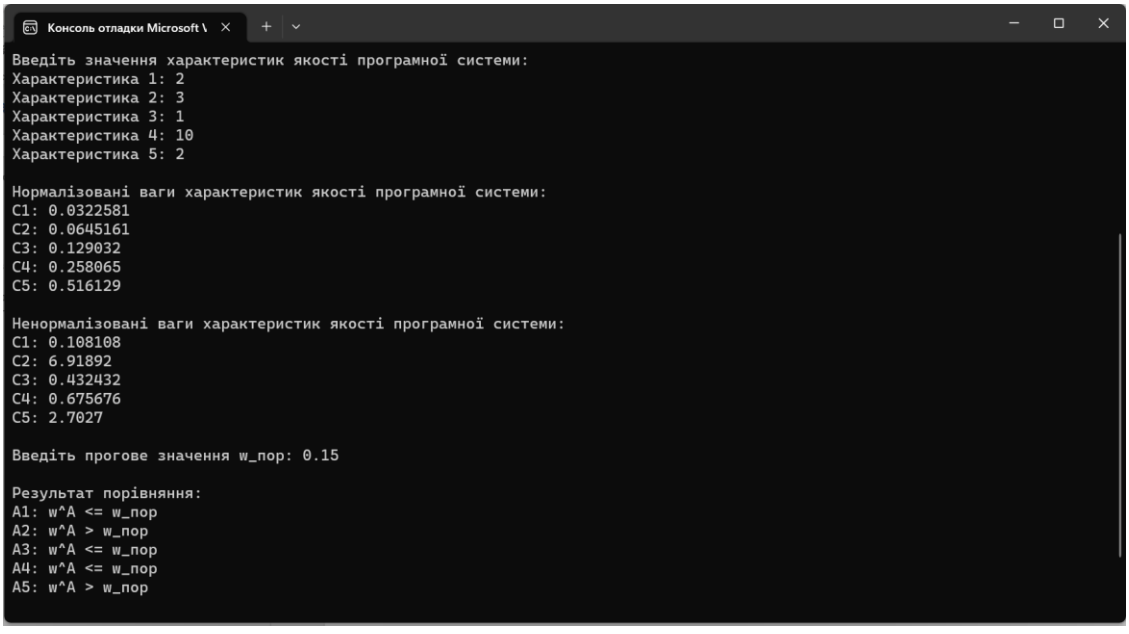


Рис. 3.8. Результат виконання обчислень значень ваг якості ПС та архітектури з пороговим значенням 0.15

За цими обчисленнями (рис. 3.8) слідує, що друга і п’ята характеристика задовольняють вимогам якості до архітектури, через те, що значення їх ваг перевищують порогове значення.

Отже, ця програма є прикладом розрахунку нормалізованих та ненормалізованих ваг характеристик якості архітектури та програмної системи. Вона дозволяє користувачу ввести вагові коефіцієнти для кожної характеристики і обчислити нормалізовані ваги за формулою.

Нормалізовані ваги характеристик якості архітектури визначаються на основі введених вагових коефіцієнтів програмної системи. Результати розрахунку виводяться на екран.

Крім того, програма має функцію введення прогового значення w_пор для порівняння нормалізованих ваг характеристик якості архітектури.

В цілому, програма може бути використана як основа для подальшого розроблення більш складних програм, пов'язаних з оцінюванням якості архітектур та програмних систем.

3.4. Модифікований програмний комплекс «Архітектор».

Для модифікації програмного комплексу «Архітектор» додаємо програму по визначенню вимог якості архітектури. Таким чином програмний комплекс складатиметься з чотирьох основних програм:

- Програма створення альтернативних архітектур та попарної експерної оцінки їх;
- Програма перегляду оцінок;
- Програма виставлення критеріальних пріорітетів та прийняття рішення;
- Програма визначення вимог якості архітектури.

Додавання програми визначення вимог якості архітектури розширює функціональність комплексу та дозволяє враховувати важливі аспекти якості в процесі проектування.

Основна ідея полягає в тому, що вимоги до якості архітектури програмного забезпечення визначаються і враховуються ще на етапі проектування. Це дозволяє

забезпечити високу продуктивність, ефективність роботи програми, надійність та безпеку. Наприклад, такі вимоги можуть включати швидкодію, масштабованість, стійкість до помилок, безпеку даних, зручність використання та багато інших аспектів.

Додавання програми визначення вимог якості архітектури до програмного комплексу "Архітектор" має наступні переваги:

- Забезпечення узгодженості. Програма дозволяє визначити вимоги до якості архітектури заздалегідь і враховувати їх під час проектування. Це допомагає уникнути суперечностей та неузгодженостей у майбутньому.
- Підвищення якості продукту. Врахування вимог до якості на етапі проектування допомагає створити продуктивне та ефективне програмне забезпечення. Це може включати оптимізацію продуктивності, використання кращих практик програмування, уникнення проблем з пам'яттю та інші аспекти, які впливають на якість програмного забезпечення.
- Підвищення надійності та безпеки. Визначення вимог до якості архітектури дозволяє враховувати аспекти надійності та безпеки на ранніх етапах проектування. Це допомагає уникнути потенційних проблем та забезпечити високу рівень стабільності та захищеності програми.
- Зменшення витрат часу та ресурсів. Визначення вимог до якості архітектури заздалегідь дозволяє уникнути необхідності внесення значних змін пізніше в процесі розробки. Це допомагає зекономити час та зусилля, а також запобігає витратам на виправлення помилок та непередбачених проблем.

Для відображення модифікації, внесемо зміни до діаграми проектування архітектури з використанням шаблонів. Побудуємо нову use-case діаграму з використанням програми визначення вимог якості архітектури (рис. 3.9.1).

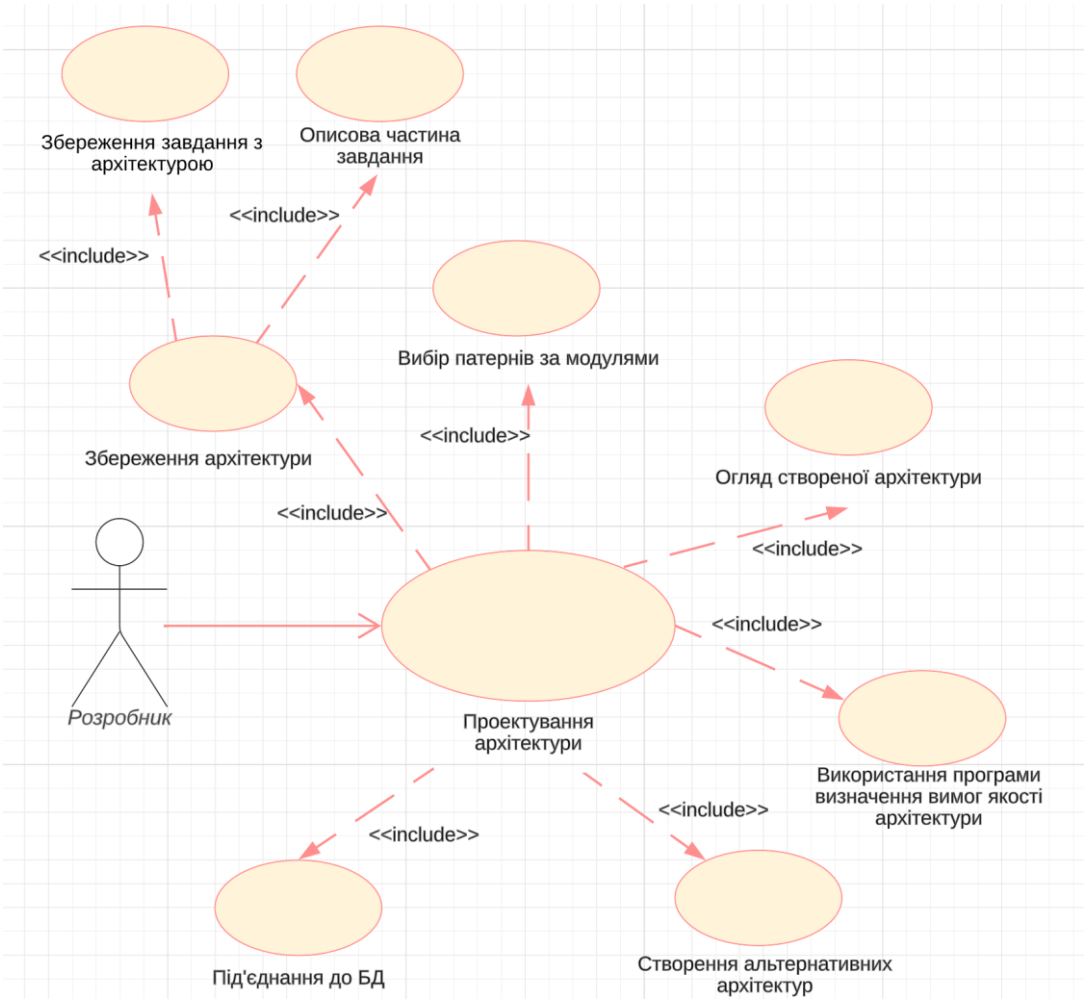


Рис. 3.9.1. Сценарій роботи *Розробника* по проектуванню архітектури з використанням програми визначення вимог якості архітектури

Актор:

- Розробник – людина, що запускає процес проектування архітектури, обирає патерни для підстановки у модулі (категорії) шарів програмного додатку, вводить описову частину завдання, приймає конструктивні рішення по архітектурі програмного додатку.

Use case:

- Проектування архітектури – базовий use case, що представляє головну задачу підсистеми і викликає інші сценарії;
- Під'єднання до бази даних – вибір бази даних (репозиторію патернів) для подальшої роботи з нею;

- Створення альтернативних архітектур – створення альтернативних рішень, завдання для подальшої роботи *Архітектора*;
- Вибір патернів за модулями – вибір патернів для кожного модуля відповідного шару архітектури;
- Огляд створеної архітектури – огляд *Розробником* створеної архітектури програмного додатку перед збереженням;
- Використання програми визначення вимог якості архітектури – застосування програми по оцінці якості вимог до створюваної архітектури;
- Збереження архітектури – виконання збереження сформованих рішень у вигляді завдання для подальшої роботи *Експертів*;
- Описова частина завдання;
- Збереження завдання з архітектурою.

Загалом, модифікація програмного комплексу "Архітектор" з додаванням програми визначення вимог якості архітектури має велику вагомість, оскільки вона допомагає забезпечити створення високоякісного та надійного програмного забезпечення. Врахування вимог до якості на етапі проектування дозволяє побудувати міцну основу для подальшої розробки та забезпечити якість продукту відповідно до очікувань користувачів і бізнес-вимог. Вона допомагає знизити ризик помилок і проблем у майбутньому, забезпечуючи ефективну та безпроблемну роботу програмного забезпечення.

ВИСНОВКИ

У цьому дипломному проекті створено модифікацію програмного комплексу «Архітектор».

За результатами можна зробити наступні висновки:

1. На початку створена функціональна модель роботи програмного забезпечення для визначення вимог якості архітектури. Для цього побудована відповідна UML діаграма, де зображені основні можливості програми та різні сценарії її використання.

2. Для вирішення проблеми визначення оптимальних вимог якості до програмної архітектури, було розроблено програмне забезпечення, яке за ваговими коефіцієнтами якості програмної системи рахує нормалізовані ваги для архітектури та виводить їх для користувача на екран. Розроблений діалог з користувачем, для подальшої зручної роботи, засобами середовища розробки Visual Studio. Зроблений аналіз отриманих результатів та математичні розрахунки, для перевірки ПЗ.

3. На останньому етапі дипломної роботи відбувається порівняння розрахованих нормалізованих ваг якості архітектури з пороговим значенням для уникнення непотрібних характеристик в програмній системі. Таким чином, це забезпечує оптимізацію розробки програмної архітектури та дозволяє комплексно оцінити необхідні вимоги та побудувати потрібну програмну систему.

Отже, розробка та впровадження системи забезпечення якості архітектури програм може зробити значний внесок у загальний успіх програмних продуктів. Забезпечуючи дотримання найкращих архітектурних практик, фіксуючи та оцінюючи вимоги до якості та використовуючи ефективні методології оцінювання, ця система підвищує надійність, зручність використання та загальну якість архітектури програмного забезпечення.

Надалі можна провести певні модифікації запропонованого програмного забезпечення, щоб покращити якість отриманих результатів та збільшити масштаб в рамках розробки більш складних архітектур.

СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Bass L. Software Architecture / Bass L., Clements P., Kazman R. ; Pub. by Addison-Wesley, 2012. – 510 p.
2. Shaw M. Software Architecture: Perspectives on an Emerging Discipline / Shaw M., Garlan D. ; Pub. by Prentice Hall, 1996. – 264 p.
3. Fowler M. UML Distilled: A Brief Guide to the Standard Object Modeling Language (3rd ed.) / M. Fowler ; Pub. by Addison-Wesley, 2003. – 208 p.
4. Bass L. Software Architecture in Practice (3rd ed.) / Bass L., Clements P., Kazman R. ; Pub. by Addison-Wesley, 2012. – 640 p.
5. Medvidovic N. Software Architecture: Foundations, Theory, and Practice (2nd ed.) / Medvidovic N., Taylor R. N., Dashofy, E. M. ; Pub. by Wiley, 2020. – 720 p.
6. DeLine R. Evaluating the Benefits of 3D Task Visualizations / DeLine R., Czerwinski M., Robertson G. ; In Proceedings of the Conference on Computer Supported Cooperative Work ; Pub. by ACM, 2002. – 299 p.
7. ISO/IEC/IEEE. Systems and Software Engineering - Architecture Description ; ISO/IEC/IEEE 42010:2011. 2011, pp. 5-13.
8. Rozanski N. Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives (2nd ed.) / Rozanski N., Woods E.; Pub. by Addison-Wesley Professional, 2011. – 736 p.
9. Kruchten, P. The Rational Unified Process: An Introduction (3rd ed.) / P. Kruchten ; Pub by Addison-Wesley Professional, 2003. – 352 p.
10. ISO/IEC. Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models ; ISO/IEC 25010:2011. 2011.
11. ISO/IEC/IEEE. Systems and Software Engineering - Systems and Software Quality Requirements and Evaluation (SQuaRE) - System and Software Quality Models ; ISO/IEC/IEEE 25010:2018. 2018.

12. ISO/IEC/IEEE. Systems and software engineering - Systems and software quality requirements and evaluation (SQuaRE) - Quality requirements ; ISO/IEC/IEEE 25030:2017. 2017.

13. França A. C. Quality attributes prioritization using the quality house model / França A. C., Carvalho M. ; In Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR); Pub. by IEEE, 2012, pp. 399-404.

14. França A. C. Software architecture evaluation using quality attributes prioritization techniques / França A. C., Carvalho M. ; Pub by Journal of Systems and Software, 2013, 86(6), 1628-1643 pp.

15. Akao, Y. Quality Function Deployment: Integrating Customer Requirements into product design / Translate by G.H Mazur – Cambridge ; Pub by M.A Productivity Press, 1990. – 369 p.

16. M. Mousavi Shafaee. Quality House: A Framework for Software Quality Evaluation / M. Mousavi Shafaee, M. Oroujzadeh, A. Keshavarzi; Pub. in the International Journal of Computer Theory and Engineering, Vol. 2, No. 3, 2010.

17. Smith M. Scalable Performance Signalling and Congestion Avoidance / M. Smith ; Pub by Springer, 2015. – 178 p.

18. Rubin J. Handbook of Usability Testing: How to Plan, Design, and Conduct Effective Tests / J. Rubin ; Pub. by John Wiley and Sons, 1994. – 384 p.

19. Hassenzahl M. User Experience: Towards an Experiential Perspective on Product Quality / M. Hassenzahl ; Proceedings of the 20th International Conference on Association Francophone d'Interaction Homme-Machine, 2008, 11-15, p. 4.

20. Leveson N. G. System Safety Engineering and Risk Assessment: A Practical Approach. / N. G. Leveson ; Pub. by CRC Press, 2004. – 444 p.

ДОДАТКИ

Додаток А

Код програми по визначенню вимог якості архітектури у середовищі розробки Visual Studio

```cpp
#include <iostream>
#include <vector>
#include <cmath>
#include <locale>
#include <Windows.h>

using namespace std;

// Функція для розрахунку нормалізованих ваг
vector<double> calculateNormalizedWeights(const vector<double>& systemWeights) {
    int numCharacteristics = systemWeights.size();
    double sum = 0.0;
    vector<double> normalizedWeights(numCharacteristics);

    // Розрахунок суми вагових коефіцієнтів програмної системи
    for (double weight : systemWeights) {
        sum += weight;
    }

    // Розрахунок нормалізованих ваг характеристик якості архітектури
    for (int i = 0; i < numCharacteristics; i++) {
        normalizedWeights[i] = systemWeights[i] / sum;
    }

    return normalizedWeights;
}

int main() {
    setlocale(LC_CTYPE, "ukr");
    SetConsoleOutputCP(1251);

    vector<double> systemWeights;
    int numCharacteristics;

    cout << "Введіть кількість характеристик якості програмної системи: ";
    cin >> numCharacteristics;

    // Введення коефіцієнтів пріоритетності програмної системи
    cout << "Введіть коефіцієнти пріоритетності програмної системи:\n";
    for (int i = 0; i < numCharacteristics; i++) {
        double weight;
        cout << "Характеристика " << (i + 1) << ": ";
        cin >> weight;
        systemWeights.push_back(weight);
    }

    // Розрахунок нормалізованих ваг характеристик якості архітектури
    vector<double> normalizedWeights = calculateNormalizedWeights(systemWeights);
```

```
49
50      // Виведення результатів
51      cout << "\nНормалізовані ваги характеристик якості архітектури:\n";
52      for (int i = 0; i < numCharacteristics; i++) {
53          cout << "A" << (i + 1) << ": " << normalizedWeights[i] << endl;
54      }
55
56      // Розрахунок характеристик якості програмної системи
57      vector<double> systemCharacteristics(numCharacteristics);
58      double prioritySum = 0.0;
59
60      cout << "\nВведіть значення характеристик якості програмної системи:\n";
61      for (int i = 0; i < numCharacteristics; i++) {
62          cout << "Характеристика " << (i + 1) << ": ";
63          cin >> systemCharacteristics[i];
64      }
65
66      // Розрахунок суми пріоритетів характеристик якості програмної системи
67      for (int i = 0; i < numCharacteristics; i++) {
68          prioritySum += pow(2, i);
69      }
70
71      // Розрахунок нормалізованих ваг характеристик якості програмної системи
72      cout << "\nНормалізовані ваги характеристик якості програмної системи:\n";
73      for (int i = 0; i < numCharacteristics; i++) {
74          double normalizedWeight = pow(2, i) / prioritySum;
75          cout << "C" << (i + 1) << ": " << normalizedWeight << endl;
76      }
77
78      cout << "\nНенормалізовані ваги характеристик якості програмної системи:\n";
79      for (int i = 0; i < numCharacteristics; i++) {
80          double unnormalizedWeight = normalizedWeights[i] * systemWeights[i];
81          cout << "C" << (i + 1) << ": " << unnormalizedWeight << endl;
82      }
83
84      // Введення прогового значення w_пор
85      double wThreshold;
86      cout << "\nВведіть прогове значення w_пор: ";
87      cin >> wThreshold;
88
89      // Порівняння нормалізованих ваг характеристик якості
90      // архітектури з вагами характеристик якості програмної системи
91      cout << "\nРезультат порівняння:\n";
92      for (int i = 0; i < numCharacteristics; i++) {
93          if (normalizedWeights[i] > wThreshold) {
94              cout << "A" << (i + 1) << ": w^A > w_пор\n";
95          }
96          else {
97              cout << "A" << (i + 1) << ": w^A <= w_пор\n";
98          }
99      }
100
101     return 0;
102 }
```