МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук

Кафедра Програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

другий (магістерський)
(рівень вищої освіти)

Дослідження методологій розробки ПЗ щодо проблем комунікації під час проєктування архітектури ПЗ

Виконав:
студент 2 курсу групи ІПЗм-21-1
Шевченко Б.М.
(прізвище, ініціали)

Спеціальність: 121 – Інженерія програмного забезпечення

Тип програми: Освітньо-наукова

Керівник доц. Афанасьєва І.В.
(посада, прізвище, ініціали)

Допускається до захисту:
Зав. Кафедри _____ З.В. Дудар

2023

Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук

Кафедра Програмної інженерії

Рівень вищої освіти другий (магістерський)

Спеціальність 121– Інженерія програмного забезпечення
(код і повна назва)

Тип програми освітньо-наукова програма

Освітня програма Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:

Зав. кафедри проф. З.В. Дудар

(підпис)

« 29 » березня 202 3 р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студента Шевченку Богдану Миколайовичу
(прізвище, ім'я, по батькові)

1. Тема роботи «Дослідження методологій розробки ПЗ щодо проблем комунікації під час проєктування архітектури ПЗ»

затверджена наказом університету від « 29 » березня 202 3 р. № 302 Ст

2. Термін подання студентом роботи до екзаменаційної комісії « 22 » травня 202 3 р.

3. Вихідні дані до роботи *методології розробки ПЗ, підходи до проєктування та документування ПЗ, якість ПЗ, інструменти для командного проєктування, підходи до управління технічним боргом, пояснювальна записка.*

4. Перелік питань, що потрібно опрацювати в роботі *мета роботи, аналіз проблемної галузі і постановка задачі, дослідження предметної області і методологій розробки ПЗ, застосування систематичного огляду літератури для отримання дороговказу предметної області, пошук відомих інструментів для покращення комунікації підчас розробки ПЗ, створення візії та прототипу інструменту для цього.*

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Аналіз предметної галузі та постановка задачі	06.02.2023	виконано
2	Аналіз поточного стану індустрії	16.04.2023	виконано
3	Створення прототипу інструменти на базі гри «Морський бій»	30.04.2023	виконано
4	Підготовка пояснювальної записки	12.05.2023	виконано
5	Підготовка презентації та доповіді	12.05.2023	виконано
6	Перевірка на академічний плагіат	13.05.2023	виконано
7	Нормоконтроль	16.05.2023	виконано
8	Рецензування	17.05.2023	виконано
9	Занесення диплома в електронний архів	18.05.2023	виконано
10	Попередній захист	19.05.2023	виконано
11	Допуск до захисту у зав. кафедри	20.05.2023	виконано
12	Захист кваліфікаційної роботи	23.05.2023	

Дата видачі завдання «15» січня 2023 р.

Студент Шевченко Богдан Миколайович
(підпис)

Керівник роботи _____ доц. Афанасьєва І.В.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ / ABSTRACT

Записка до кваліфікаційної роботи містить: 122 с., 18 рис., 3 табл., 143 джерел, 5 додатки.

АРХІТЕКТУРА, ІНСТРУМЕНТ, КОМУНІКАЦІЯ, МЕТОДОЛОГІЯ, ПРИЙНЯТТЯ РІШЕНЬ, ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ, AGILE.

Мета цього дослідження полягає в тому, щоб визначити, які інструменти з їх характеристиками існують для пом’якшення проблем комунікації в сучасній розробці програмного забезпечення, а також точні проблеми, які виникають під час проєктування архітектури програмного забезпечення з їх характеристиками та дієвістю.

В якості основного методу дослідження використовується систематичний огляд літератури (СОЛ, SLR). У результаті роботи було досліджено область колективної розробки ПЗ, проведено систематичний огляд літератури, згідно з розробленим протоколом, сформульовано наше бачення ідеального інструменту та запропоновано прототип найпростішого інструменту на базі гри «Морський бій».

ARCHITECTURE, TOOL, COMMUNICATION, METHODOLOGY, DECISION MAKING, SOFTWARE, AGILE.

The purpose of this research is to define which instruments with their characteristics exist to mitigate the communication issues in modern software development as well as the exact issues that emerge during software architecture design with their characteristics and validity.

A systematic Literature Review (SLR) is used as a major method of the research. As a result of the work, the field of collective software development was investigated, a systematic review of the literature was conducted, according to the developed protocol,

our vision of the ideal tool was formulated and a prototype of the simplest tool based on the game "Battleship" was proposed.

Умови публікації пояснювальної записки

Я, Шевченко Богдан Миколайович
(прізвище, ім’я, по батькові)

студент(ка) групи ІПЗм-21-1 здобувач вищої освіти на другому (магістерському) рівні

кафедра програмної інженерії,
(повна назва кафедри)

заявляю: моя кваліфікаційна робота на тему

Дослідження методологій розробки ПЗ щодо проблем комунікації під час проєктування архітектури ПЗ
(назва роботи)

що буде представлена до ЕК для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в електронному архіві відкритого доступу ElArKhNURE. Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлений (а) з діючим положенням «Про протидію академічному плагіату в ХНУРЕ», згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

ГАП – фреймворк Глобальної архітектурної практики розробки програмного забезпечення (GAP, Global Software Development Architectural Practice)

ГРПЗ – глобальна розробка програмного забезпечення.

ДП – дослідницьке питання.

ДП№ – дослідницьке питання, де № – порядковий номер, який унікально ідентифікує дослідницьке питання.

ІнОФ – інструмент на основі форм, умовна назва візії (бачення) ідеального інструменту.

КПР – керована поведінкою розробка (англ. Behavior-Driven Development, BDD).

КТР – керована тестами розробка (англ. Test-Driven Development, TDD).

ПЗ – програмне забезпечення.

ПЗВК – програмне забезпечення з відкритим кодом.

РПЗ – розробка програмного забезпечення.

СВД – систематичне відображувальне дослідження (Systematic Mapping Study, SMS).

СОЛ – систематичний огляд літератури (англ. Systematic Literature Review, SLR), метод дослідження літератури призначений для ознайомлення з певною предметною галуззю та визначення напрямків майбутнього дослідження [1].

ХР – Extreme Programming, укр. екстремальне програмування.

ВСТУП

Розробка програмного забезпечення існує вже більше 50 років. Ця галузь розвивається і, за прогнозами, буде розвиватися найближчим часом. Враховуючи складність додатків, розробка програмного забезпечення стає все більш і більш колективною діяльністю, що підвищує роль комунікації та управління знаннями під час розробки. Крім того, групи розробників програмного забезпечення часто розкидані по всьому світу. Через важливість комунікації на початку 21 століття з'явився новий набір методологій під назвою Agile. Agile-методології (або гнучкі) надають перевагу швидкому та ефективному спілкуванню, а не обширному й детальному плануванню та документації, тому не дивно, що Agile-методології сьогодні використовуються в переважній більшості програмних проєктів. Однак важливою частиною планування є проєктування архітектури, а важливою частиною документації є управління технічними та архітектурними боргами. Проєктування й планування формують більшість подальших виборів і документування відомих проблем є однаково важливим. У найпопулярніших гнучких підходах (Scrum, Kanban, Extreme Programming, DevOps) не вказується, як потрібно керувати цими аспектами планування та документування, і до даного дослідження не було відомо про будь-яку інформацію про те, що існують додаткові, широко використовувані та прийнятні інструменти для цієї мети.

На додачу до систематичного огляду літератури, у цьому дослідженні представлено бачення (візія) ідеального інструменту, а також апробація висновків СОЛ у невеликому сурогатному проєкті гри «Морський бій», застосовуючи загальновживані нефункціональні вимоги з індустріальних проєктів із високим ступенем співпраці. Крім того, зроблено наголос на простоті та легкості використання інструментів, а також важливості популяризації та просування як знайдених, так і майбутніх інструментів для полегшення перевірки та впровадження в галузі.

Значна увага приділяється наявності перевірки знайдених засобів на практиці зі зворотним зв'язком. Якщо інструмент не перевірено на практиці чи впроваджено в реальні проєкти, такі інструменти було порівняно з баченням

ідеального інструменту, щоб зрозуміти, чи можна цей інструмент швидко застосувати на практиці, чи необхідно продовжувати подальші дослідження, або потрібні певні зміни. На жаль, не було знайдено інструментів, які б відповідали візії та були популяризованими, ретельно перевіреними на практиці або відносно широко використовуваними в галузі. Найближче, що було знайдено, це 2 галузеві документи, які зосереджені на перепрофілюванні існуючих практик Agile для пом'якшення проблем планування та документації. Також було знайдено кілька інструментів, які, згідно висновків цього дослідження, можна відносно легко перевірити на практиці або впроваджувати до щоденного використання в промисловості. Однак частиною висновків цього дослідження є визнання, що подальші дослідження знайдених інструментів, які не задовольняють визначеним вимогам, можуть призвести до їх переоцінки в майбутньому для повторної перевірки бажаних властивостей. Крім того, було визначено дороговказ (англ. roadmap) для майбутніх досліджень як для пошуку інструменту, так і для розробки нового інструменту.

У контексті цієї цього дослідження комунікація використовується у значенні взаємодії між зацікавленими сторонами та розробниками програмного забезпечення, не міжмашинну комунікацію за допомогою комп'ютерних мереж чи інших засобів.

Даний напрямок дослідження, хоч і не є основним для кафедри Програмної інженерії, але також висвітлений у нещодавніх роботах викладачів за напрямком якості ПЗ та методологій РПЗ. Зокрема, у роботі [2] та за участі доцента І.В. Афансьєвої було розглянуто нові архітектурні віяння, засновані на філософії Agile. У даному дослідженні було детально проаналізовано керованої поведінкою розробку – методологію, яка краще зв'язує розробку з безпосередньо запитами зацікавлених сторін програмного продукту засобами тестування та легковагої фіксації вимог. Через значне покладання на тестування, дану методологію можна вважати розширенням КТР. Як видно з визначення КПР, дана методологія концентрується на побудові комунікації між командою розробки та іншими зацікавленими сторонами, а також технічній складовій щодо організації процесу

тестування. Поточне дослідження зосереджується на побудові ефективної системи комунікації між розробниками, прийнятті рішень та управлінні технічним боргом.

Викладачі кафедри також приділили суттєву увагу аналізу якості продукту засобами різноманітних метрик [3] включаючи різноманітні характеристики коду, як-от кількість рядків, кількість контролюючих блоків коду, різні аспекти складності програми. Однак дані метрики не можна використати безпосередньо для впливу на спілкування підчас РПЗ.

Документація є непрямим засобом спілкування у процесі РПЗ [4]. Проте, документація є частиною документообігу, поняття, яке не є унікальним для РПЗ. Саме тема покращення документообігу була розібрана у публікації [5]. За допомогою побудови графової моделі авторам дослідження вдалося зменшити документообіг відкритого акціонерного товариства Харківобленерго на 11%. Це є значним результатом, особливо беручи до уваги розміри підприємства, однак у контексті РПЗ документація не має такого вирішального значення. Тому запропонований підхід хоч і може бути застосований до проєктів ПЗ, не може вважатися рішенням для покращення комунікації у процесі ПЗ загалом.

Відсутність готових рішень, що стосуються саме збільшення ефективності документації ПЗ, зменшення технічного боргу та покращення процесу прийняття рішень, як на кафедрі Програмної інженерії, так і за її межами, підкреслює необхідність у розробці власного інструменту. На основі гри «Морський бій» було проведено повний цикл проєктування ПЗ: від визначення цільової аудиторії та платформи й до документування найдрібніших проблем. У прагненні збереження простоти та легковагості рішення було використано лише ті інструменти, якими зазвичай користуються майже всі розробники, та мінімізовано додаткові інструменти.

1 АНАЛІЗ ПРОБЛЕМНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Аналіз проблемної області дослідження

Інженерія програмного забезпечення (ІПЗ) – це застосування інженерних принципів до комп'ютерного програмного забезпечення, як правило, для вирішення проблем реального світу. Комп'ютерні науки (КН) – це застосування наукового методу до комп'ютерного програмного забезпечення [6]. Зацікавлені сторони або стейкхолдери (англ. stakeholders) – це «люди або речі (наприклад, інші системи), які мають вимоги чи очікування щодо системи» [7], тобто це охоплює кілька груп людей, таких як розробники, кінцеві користувачі, менеджери, тощо.

Згідно з [8], кінцевою метою розробки пропрієтарного (комерційного) програмного забезпечення є отримання прибутку чи іншої вигоди. Вони також підкреслили необхідність високої продуктивності комерційного програмного забезпечення. Гонитва за певними вигодою є вірною для open-source (з відкритим вихідним кодом) розробки програмного забезпечення, як обговорювалося в [9]: діаграми ілюструють, що окрім оплачуваних учасників відкритого коду, студенти складають значну частину відкритого коду внески. Очевидно, що студенти використовують розробку з відкритим вихідним кодом як чудовий спосіб відточити свої навички в реальних проєктах. І ці навички згодом будуть використані для роботи в галузі.

Як зазначили [8], ефективність дуже важлива, і на неї впливає безліч факторів. Важливою частиною будь-якого процесу створення продукту є початкове планування. Це особливо вірно, оскільки розробка програмного забезпечення – це командна робота. Комунікація разом із документацією потрібна, якщо немає лише одного розробника з ідеальною пам'яттю [10].

Щоб організувати роботу або, іншими словами, бізнес-процес (БП), використовуються різні методології розробки програмного забезпечення, як показано в [11]. Автори демонструють, що багато програмних проєктів (ПП) розробляються невеликими або середніми командами, і команди дотримуються процесів гнучкої розробки програмного забезпечення.

Гнучка (Agile) розробка програмного забезпечення – це набір методологій з метою прискорення процесу розробки шляхом зменшення обсягу документації та початкового планування та покращення гнучкості шляхом уможливлення змін вимог у процесі [12]. Однак, як показує опитування, команди часто використовують гібридні підходи для покращення планування процесів [11].

Додатковим аспектом Agile-розробки програмного забезпечення є глобальне розподілення співробітників: початковий Agile Manifesto [12] передбачався для використання для локальної розробки програмного забезпечення. Під час пандемії COVID-19 промисловість гостро відчула потребу в ефективній віддаленій співпраці [13].

Архітектура програмного забезпечення (АПЗ) є важливою частиною планування, і окремі рішення не можна змінити пізніше [4]. Тому важливо виконати гарне планування, зокрема, створити провести добре архітектурне проєктування. Однак методології Agile не вказують, як слід вирішувати вищезгадану проблему [14].

Існує багато різних визначень архітектури програмного забезпечення. У поточному дослідженні приймається визначення, надане в стандарті ISO/IEC/IEEE 42010:2011 (джерело №35 в [15])[1]: «фундаментальні поняття або властивості системи в її середовищі, втілені в її елементах, зв’язках, а також у принципах її конструкції та еволюції». Більше інформації про це визначення в цьому стандарті можна знайти в Таблиці 3 у роботі [16]. SWEBOK, на сторінці 53 у розділі «Структура та архітектура програмного забезпечення» [17] визначає архітектуру як «набір структур, необхідних для міркування про систему, яка включає елементи програмного забезпечення, зв’язки між ними та властивості обох». Коротша та простіша альтернатива надана у [18]: «Архітектура – це план-схема (англ. blueprint) системи». Зауважується, що визначення, які використовуються в сучасних роботах, таких як SWEBOK [17] або [19],

1 Доступу до цього платного стандарту був відсутній, тому використовується транзитивне дослідження з цитуваннями.

посилаються на визначення, які датуються 1990-ми та початком 2000-х років, наприклад [7] або [20].

Якість програмного продукту – ще один важливий термін, який необхідно визначити. Опис якості програмного продукту знайдено в SWEBOK. Існує цілий підрозділ «Основи якості програмного забезпечення, моделі та характеристики якості, якість програмного продукту», який описує основні концепції на сторінці 10-4 або 177 у [17]. Саме це повне визначення і буде використане у поточному дослідженні. Для стислості також можна навести визначення зі сторінки 6 «2.1 Визначення якості» у [21]: «Якість програмного забезпечення – це ступінь, до якого визначений галуззю набір бажаних функцій включено в продукт для підвищення його продуктивності протягом усього терміну служби».

Комунікація в проєктуванні архітектури програмного забезпечення означає офіційне або напівофіційне ділове спілкування, структура та аспекти якого спеціально узгоджені з проєктом програмного забезпечення [4]. Це означає, що існують особливості комунікації, які допомагають ефективніше передавати ідеї, пов’язані з програмним забезпеченням. Комунікація має бути дещо схожою за структурою на проєктну документацію, як-от плани тестування та тестові випадки (англ. test cases), документ SRS [22, 10]. Наявність інструкцій щодо такого типу спілкування допоможе не забути оговорити конкретні аспекти проєктування архітектури програмного забезпечення [15, 22]. Суміжною та важливою сферою є прийняття рішень. Автори [23] провели аналіз уже існуючих досліджень щодо прийняття рішень у програмній інженерії, яка є однією з важливих складових процесу розробки програмного забезпечення.

Один з СОЛ на тему ерозії архітектури програмного забезпечення в програмному забезпеченні з відкритим кодом [19]. Особливістю досліджень ПЗВК є те, що дані можна брати безпосередньо з відкритих джерел, якщо тільки ПЗВК не розробляє комерційна компанія. Стаття охоплює ерозію архітектури, використовуючи як кількісні статистичні, так і якісні методи. Тема публікації є ширшою, ніж проблемна комунікація, і обговорює проблеми процесу розробки

програмного забезпечення в цілому. Автори також дуже ретельно розбирають у свою методологію та термінологію, вони визначають дуже чіткий процес СОЛ.

У цій же роботі [19] піднімається проблема «необізнаності розробника», яка безпосередньо пов’язана з проблемами комунікації. Автори підкреслюють важливість цього питання, що можна побачити як у статтях, які вони вивчали в процесі СОЛ, так і в їхніх висновках зі статті.

Наскільки відомо, комунікація в контексті проєктування архітектури програмного забезпечення має чимало публікацій. Мотивація для проведення даного дослідження виникла після роботи над кількома галузевими проєктами та спостереженням за накопиченням дрібних проблем, які збільшували ризик поломки всієї системи. припускається, що основною причиною проблеми є початкове архітектурне планування та непорозуміння між розробниками. Ця область була досліджена за допомогою інженерних [24, 25] та наукових підходів [22, 26, 19, 13]. Зокрема, перераховані роботи досліджували проблему дрейфу архітектури, тобто розбіжності спроєктованої та реалізованої архітектури та проблеми, як наслідок дрейфу [19]. Дрейф архітектури може виникнути як під час обслуговування програмного забезпечення, так і безпосередньо під час початкової розробки програмного забезпечення [23, 22].

Хоча в науковій літературі є багато публікацій про Agile-процеси та розпізнавання архітектурного дрейфу [19], є лише кілька таких публікацій про зв’язок практик, які допомагають уникнути архітектурного дрейфу, і Agile процесів. У книзі [14] припускається, що майбутнє архітектури програмного забезпечення – це її інтеграція з методологіями Agile.

Кілька наукових досліджень розглядають роль комунікації в розробці програмного забезпечення. Наприклад, деякі з них називають проблеми з комунікацією, які впливають на якість програмного забезпечення, як «соціальний борг»[2)] [13, 27, 26, 28]. У всіх цих документах в якості основного методу збору даних використовуються опитування та інтерв’ю розробників програмного забезпечення й інженерів із комерційних компаній, що займаються розробкою

[2] Визначення можна знайти у 2.12 Дослідження додаткових понять і взято з [27].

програмного забезпечення. Не було знайдено жодного огляду літератури, який би зосереджувався на пошуку нових інструментів для пом’якшення проблем комунікації під час розробки програмного забезпечення, тому є потреба виконати таке дослідження [29].

Згідно з класичними визначеннями, архітектура програмного забезпечення – це статична діаграма, яка окреслює систему: у визначеннях стверджується, що архітектура програмного забезпечення – це план-схема [18], тобто знімок або «набір структур» [17]. Було знайдено лише одне визначення, яке нечітко згадує мінливість у часі – ISO/IEC/IEEE 42010:2011 (джерело №35 в [15]) згадує, що «еволюція» є частиною архітектури. Архітектура програмного забезпечення – це динамічне поняття, яке змінюється з часом під впливом багатьох факторів, включаючи спілкування між розробниками та іншими зацікавленими сторонами [26, 30].

Архітектура в Agile не відіграє важливої ролі через орієнтований на людей характер цих методологій [14]. Однак, існують практики на основі Agile, які вирішують цю проблему:

- галузева стаття [24], стверджує, що вирішення проблем має бути спільним і націленим на першопричину, а не «маскування» проблем розробниками, які працюють окремо. Рішення, які пропонує авторка, уже описані в Agile методологіях [31], особливо екстремальне програмування (англ. extreme programming, XP), і включають перевірку коду, стандарти кодування та командну роботу під час виправлення помилок. Ці рекомендації можуть підвищити якість як продукту, так і процесу, але вони не стосуються комунікаційного аспекту, а зосереджуються на більш технічному аспекті розробки програмного забезпечення;
- основна ідея іншої статті [25], ще одного індустріального погляду на цю проблему, полягає в тому, що команді не потрібні спеціально призначені люди, які тільки проєктують архітектуру. Натомість до прийняття архітектурних рішень має бути залучена вся команда або, принаймні, зацікавлені сторони доречної частини проєкту. Ця проста ідея чітко

вказує на головну проблему необізнаності про прийняття рішень, яку також розглядають у [26] та [19]. Однак отримати зворотний зв’язок від кожного члена команди щодо кожного рішення неможливо, якщо немає чітко визначеної комунікаційної структури [15]. Стаття не містить жодних статистичних або вимірюваних даних;

- ще одне дослідження [13] зосереджено на тому факті, що команди розподілені та переважно працюють віддалено, у тому числі через пандемію COVID-19. Автори статті провели опитування двох великих страхових компаній, обидві компанії з офісами Німеччини. Запропоновані зглажування (англ. mitigations) були дуже схожі на статтю інженера [25]. Автори [13] визнають, що їхні методи дослідження були обмеженими, оскільки вони не опитали багато людей і працювали лише з людьми з двох компаній.

Пропозиції як [24], так і [25] були перевірені на практиці. Часто під час роботи над проєктом важливістю командної роботи нехтують [23]. Це може призвести до проблем у розробці програмного забезпечення, які важко відстежити, так званого «розмивання архітектури» [23].

На відміну від [13], [15, 22] досліджують глобальну розробку програмного забезпечення (ГРПЗ) без урахування методології Agile. Інші дослідники намагаються розглянути дрейф архітектури [19] окремо від гнучких методологій і ГРПЗ:

- в одному з досліджень [26] було проведено опитування в одній компанії, але між різними відділами. Автори зосереджуються на запахах спільноти, які є антишаблонами (антипаттернами) в комунікаційному процесі розробки програмного забезпечення. Крім того, автори розширюють свою попередню роботу над інструментом DAHLIA (англ. Debt-Aimed arcHitecture-Level Incommunicability Analysis, аналіз некомунікабельності на рівні боргів) і докладають зусиль для кількісної оцінки результатів опитувань за допомогою формул DAHLIA. Концепції, які вони використовують для розробки формул для DAHLIA, запозичені

з економіки. Крім того, вони розглядають різні антишаблони в архітектурному дизайні та виділяють різні типи пом'якшення, які, на їхню думку, є дещо схожими на інші вищезгадані документи;

– дослідження [22] проводить масштабне опитування розробників із семи різних компаній і детальний аналіз. В результаті їхньої роботи було представлено Global Software Development Architectural Practice (GAP) Framework, яка базується на Concern Framework із попередньої роботи тих же авторів [15]. Структура, яку вони представили, прагне структурувати комунікацію в компаніях, використовуючи ті самі принципи, що й UML-діаграма діяльності або класу [32]. Як розширення структури, кожна область процесу програмного забезпечення (у документі називається «темою», від англ. theme) GAP Framework містить перелік суміжних викликів і проблем. Загалом, автори намагаються кількісно визначити проблему порушення зв'язку, представивши свої фреймворки, але під іншим кутом відносно з [26]. Результатом роботи є велика діаграма з багатьма взаємозв'язками.

Дослідження [28] доповнює попередню публікацію аналізом запахів громади (community smell) з архітектурними проблемами, про які повідомлялося під час інтерв'ю. Загалом, [27] проводять широке дослідження соціального боргу[з] [33, 26, 28].

Наявність багатьох відокремлених інструментів означає, що дана предметна область достатньо зріла, але не означає, що ця область досліджень добре досліджена. Не всі інструменти широко використовувалися на практиці, не кажучи вже про те, щоб стати стандартом де-факто в галузі, як це сталося з методологією Agile. Це означає, що ця область досліджена, але не остаточно.

Гнучкі практики та методології використовуються як стандарт де-факто для процесів розробки програмного забезпечення [31].

з Визначення можна знайти у 2.12 Дослідження додаткових понять і взято з [27].

Негнучкі методології (англ. non-Agile) [11] зазвичай поділяють процес розробки на кілька етапів:

- отримати вимоги від зацікавлених сторін;
- зпланувати завдання для команди на заздалегідь визначений період часу;
- працювати над поставленими завданнями та впровадження функціоналу;
- розгортати реалізовані функції.

Гнучкі методології, такі як Scrum і Kanban, чітко не визначають етап проєктування програмного забезпечення, зокрема архітектури, навіть якщо це важлива частина між плануванням і реалізацією. Введення цього додаткового етапу не порушує роботи методологій Agile і може розглядатися як розширення.

Архітектурне проєктування часто розглядається як технічне завдання [24] без колаборативного (спільного) характеру процесу проєктування архітектури. Це призводить до появи багатьох статей і дослідницьких робіт, які обговорюють, як вирішити певні проблеми лише з технічної точки зору, з дуже невеликим обговоренням того, як побудувати процес співпраці та командної роботи.

Проведено багато досліджень щодо того, як побудувати процес спільної розробки, наприклад, використовуючи такі методи екстремального програмування, як перегляд коду, стиль коду та конвеєри CI/CD [31, 24]. Ці найкращі галузеві практики не охоплюють процес проєктування архітектури.

1.2 Постановка задачі

Основна мета дослідження полягає в тому, щоб знайти, чи існує існуючий інструмент для усунення комунікаційних недоліків під час проєктування архітектури програмного забезпечення. Ще краще, якщо такий користується хоча б помірною популярністю, що має підтверджуватися статистикою. Популярність гіпотетичного інструменту означає:

- перевірку, тому що вважається, що галузь не буде використовувати проблемний інструмент;
- відсутність необхідності досліджувати додаткові інструменти, оскільки вважається, що було б легше популяризувати вже існуючий інструмент.

Для переконання, що такий інструмент не буде пропущений і для отримання актуального розуміння сучасного стану у цій галузі, слід проаналізувати цільову літературу [29]. Якщо інструмент не надто популярний, слід ще ретельніше вивчати літературу, щоб не пропустити його. Відповідно до [29] існує 2 основні підходи до огляду літератури: систематичне відображувальне дослідження (СВД, англ. Systematic Mapping Study, SMS) і систематичний огляд літератури (СОЛ, англ. Systematic Literature Review, SLR). Обидва наукові методи пов’язані з пошуком джерел, видобуванням, аналізом інформації та підсумуванням отриманих результатів. Однак СОЛ є більш глибоким, тому було вирішено рухатися вперед із СОЛ як основним науковим методом поточного дослідження.

Як видно з опису СОЛ, це є міцним підходом, чітко направленим на пошук дуже конкретної інформації в усій множині джерел. Тому доцільно буде провести детальне дослідження вже існуючих популярних методологій РПЗ задля знаходження та можливого перевикористання вже існуючих, але не дуже популярних практик. Задля більш детального аналізу існуючих інструментів, необхідно сформувати вимоги до цього інструменту, тобто подати авторське бачення (візію) ідеального інструменту для покращення комунікації між розробниками.

Для пошуку інструменту методом СОЛ необхідно сформувати протокол СОЛ [29], який описує, яким чином буде відбуватися пошук літератури для огляду, які критерії для включення й виключення літератури з огляду та яким чином перевірятиметься якість дослідження. Протокол СОЛ нерідко включає класифікаційні питання, що дозволяє зібрати інформацію про оглянуту літературу в більш структурованому вигляді. Враховуючи важливість наявності впровадження інструменту, слід включити відповідні питання до класифікаційного протоколу, зокрема, наявність підтвердження про впровадження безпосередньо в публікації або посилання чи згадки даної публікації в інших джерелах.

Якщо такий інструмент буде знайдено, то дослідження вважається завершеним та успішним, мета є виконаною. Якщо ні, то необхідно зрозуміти,

чому існуючі дослідження не запропонували рішення чи чому ці рішення не використовуються наразі. Після аналізу можливих помилок в існуючих досліджень та можливих недоліків в існуючих інструментах, з’являється необхідність у розробці власного інструменту, такого, що задовольняє візії (табл.1.1).

Таблиця 1.1 – Дослідницькі питання.

ДП1	Які найпоширеніші типи дрейфу архітектури через неправильне спілкування та які найпоширеніші наслідки таких помилок?
ДП2	Які існують запропоновані інструменти для пом'якшення дрейфу архітектури та проблем зв'язку?
ДП3	Які були причини не прийняти існуючі інструменти? Чи є наявні інструменти занадто розпливчастими чи складними?
ДП4	Які дії можна зробити та які заходи можна вжити, щоб пом’якшити дрейф архітектури та проблеми з комунікацією на практиці?

Таким чином, у таблиці 1.1 було сформовано дослідницькі питання цієї роботи. Задля зручності посилання на дослідницькі питання в подальшому, їм присвоєні акроніми у форматі ДП№ – дослідницьке питання та його номер.

1.3 Аналіз категорій читачів

Перед виконанням аналізу методів дослідження, варто зрозуміти, хто буде зацікавлений у результатах поточного дослідження. Є кілька цільових груп, які можуть бути зацікавлені в цьому дослідженні:

- спільнота дослідників, що працюють у сфері комунікації та прийняття рішень у розробці програмного забезпечення. Дослідження може бути цікавим не лише для дослідників комунікації та архітектури програмного забезпечення, але й для дослідників у суміжних областях, таких як технічний борг, прийняття рішень або Agile-методології. Крім того, дослідники є основною цільовою групою, оскільки поточне дослідження

не є остаточним у тому сенсі, що не було знайдено єдиного популярного інструменту;

- керівники команд, архітектори, менеджери проєктів, менеджери продуктів, власники продуктів та інший персонал, пов’язаний з управлінням або архітектурою, у групах розробки програмного забезпечення. Для стислості ми називатимемо цю категорію «зацікавлені сторони управління». Вважається, що потенційною цільовою аудиторією є керівники, оскільки це дослідження має на меті підвищити продуктивність процесу розробки програмного забезпечення та зменшити кількість і витрати, пов’язані з прийняттям неправильних рішень;
- консалтингові компанії Agile також можуть бути зацікавлені в цьому дослідженні, оскільки воно може допомогти їм знайти певні ідеї щодо покращення процесів Agile у проєктах їхніх клієнтів.

2 ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ

2.1 Аналіз розвитку галузі

Історія комп'ютерних наук сягає корінням у 19 століття, коли Чарльз Беббідж працював над своєю «аналітичною машиною», яка зазвичай вважається першим комп'ютером [34]. Дехто може заперечити, що початок інформатики пов'язаний із представленням архітектури фон Неймана у «Вступі до "Першого чорнового звіту про EDVAC"» [35] у 1945 році. Останнє знайдене визначення комп'ютерних наук – це «вивчення комп'ютерів і алгоритмічних процесів, включаючи їх принципи, дизайн апаратного та програмного забезпечення, їх застосування та їхній вплив на суспільство» від [36]. Визначення показує, що комп'ютерні науки охоплюють величезну область науки та техніки, а також включає вивчення соціального впливу комп'ютерів на суспільство.

Дослідження в поточній роботі зосереджено лише на комунікації під час розробки програмного забезпечення. Розробка програмного забезпечення – це «застосування систематичного, дисциплінованого, кількісно виміряного підходу до розробки, експлуатації та підтримки програмного забезпечення» згідно з [37]. Конференція НАТО з програмної інженерії 1968 року вважається початком програмної інженерії як дисципліни [6], де було проведено порівняння між цивільною та програмною архітектурою. Однак термін архітектура програмного забезпечення почали використовувати лише в 1990 році, як показано в [20] і [7].

З 1960-х років комп'ютери переживають експоненціальний ріст, згідно із законом Мура [38]. Відповідно до статистики та закону Мура кількість резисторів, як і продуктивність, подвоюються кожні 18 місяців. Це означає, що комп'ютери стають дедалі складнішими. Але це не означає, що користувачі можуть відчути таке різке зростання продуктивності. Закон Вірта описує роздуття програмного забезпечення, ситуацію, коли обсяг програмного забезпечення комп'ютера збільшується з часом, що нівелює деякі переваги продуктивності апаратного забезпечення [39]. Це явище також відоме як роздуття програмного забезпечення.

Вірт [39] спостерігав різні причини роздуття програмного забезпечення, і багато з них певною мірою пов'язані з архітектурою програмного забезпечення,

як-от монолітне програмне забезпечення, брак часу на розробку та навмисне ускладнення. Ще одна важлива причина, за словами Вірта, – це плутанина споживачів щодо їхніх потреб: функції, які «приємно мати», не є основними. Праця Вірта була опублікована в 1995 році; отже, можна стверджувати, що рекомендації щодо використання об’єктно-орієнтованого програмування (ООП) і його принципів, модулів і вибіркового імпорту з них, безпечних мов і середовищ все ще допомагають сьогодні. Дослідження [40] підтвердило наявність тих самих проблем у 2019 році.

2.2 Перспективи програмної інженерії

Програмне забезпечення – це набір інструкцій, які дозволяють апаратному забезпеченню комп’ютера виконувати завдання [41]. Існує 2 типи програмного забезпечення [41]:

- системне програмне забезпечення для керування обладнанням комп'ютера;
- прикладне програмне забезпечення, яке дозволяє користувачам виконувати різні завдання.

Приклади прикладного програмного забезпечення включають електронні таблиці, текстові процесори, редактори відео та аудіо [41]. Це робить прикладне програмне забезпечення втіленням функції комп’ютера – діяти як інструмент та допомагати людям виконувати певні задачі.

Крім того, зростає потреба в програмному забезпеченні. Зараз у світі існує приблизно 9 трильйонів пристроїв [42]. Важко виміряти, скільки програмного забезпечення розроблено, але важливим непрямим показником розробки програмного забезпечення є кількість розробників та її історична тенденція. За оцінками, у світі налічується 31,1 мільйона розробників програмного забезпечення. Інші джерела стверджують, що у світі налічується 26,8 мільйонів розробників програмного забезпечення [43, 42].

Не вдалося знайти світову тенденцію популяції розробників програмного забезпечення. Отже, будуть використовуватися статистичні дані Сполучених

Штатів Америки (США), оскільки США є найбільшою та загалом здоровою та стабільною економікою [44] у світі за номінальним ВВП [45]. Для простоти ми розглядатимемо програмних інженерів разом із розробниками програмного забезпечення, тому, якщо не зазначено інакше, «розробники програмного забезпечення» означатимуть як розробників програмного забезпечення, так і програмних інженерів. У США зараз працює майже 800 000 і відкрито понад 600 000 вакансій для розробників програмного забезпечення [46, 47]. Згідно з офіційною статистикою [43, 48], в Україні майже 290 000 розробників програмного забезпечення, а експорт галузі становив 5 мільярдів доларів США у 2020 році. Крім того, під час повномасштабного російського вторгнення в Україну у 2022 [49] IT-галузь продовжувала розвиватися, незважаючи на повномасштабну війну: експорт за перше півріччя 2022 року зріс на 23% порівняно з першим півріччям 2021 року [50], а обсяг експорту прогнозувався на рівні 8,5 мільярдів доларів США до кінця 2022 року [51].

В останнє десятиліття з’явилася нова концепція – «Індустрія 4.0», яка описує інтеграцію автоматизованого виробництва (так звана «Індустрія 3.0») із передовими обчислювальними системами й системами моніторингу та підключення за допомогою Інтернету з набагато меншим залученням людей, тобто роблячи виробництво розумним [52]. Концепція порівняно нова, але німецький уряд намагався використати її як керівну ідею для подальшого промислового розвитку [52]. Програмне забезпечення є невід’ємною частиною Industry 4.0, що підвищує значущість розробки програмного забезпечення в майбутньому. Багато сучасних команд розробників програмного забезпечення не обмежуються однією локацією, а зазвичай розподілені по всьому світу, запроваджуючи Глобальну розробку програмного забезпечення [11]. Пандемія COVID-19 привнесла цікаву динаміку на ринок розробки програмного забезпечення: порушення добре налагоджених економічних процесів уповільнило зростання ринку, про що свідчить скорочення пропозиції вакансій у США на 50% [46, 47]. Але в той же час зріс попит на програмне забезпечення. Підтверджено, що Індустрія 4.0 може допомогти задовольнити зростаючий попит на медичні

товари та послуги [53]. У той же час «Індустрія 4.0» забезпечує більшу гнучкість під час карантину [54], і було припущено, що COVID-19 створив можливість пришвидшити перехід до «Індустрії 4.0» [54].

Крім Індустрії 4.0, COVID-19 перешкоджав звичайній діяльності, спричинивши карантини в багатьох країнах [54]. Це породило потребу надавати послуги віддалено в різних частинах світу:

- припускають, що багато організацій охорони здоров'я в США значно розширили використання телемедицини для дистанційного консультування за допомогою технологій аудіо- та відеоконференцзв'язку [55];
- були випадки дистанційного моніторингу пацієнтів, щоб уникнути скупчення людей і потенційного впливу COVID-19 [56];
- стверджують, що завдяки карантинним заходам цифрова релігія матиме значний вплив на суспільство [57];
- наводять, що телемедицина в Європі зменшує тиск з боку лікарень і має потенціал стати звичним явищем у майбутньому [58];
- представили перелік проблем індійської освітньої системи із закликом до інших країн щодо підготовки до дистанційної освіти заздалегідь [59].

Вищенаведені факти підтверджують зростання попиту на нове програмне забезпечення. Згідно з даними Бюро статистики праці [60] США, навіть без COVID-19 кількість розробників програмного забезпечення зросте на 5–35% у наступне десятиліття для різних професій. Крім того, прогнозується зростання IT-індустрії в Україні на рівні не менше 16% на рік у майбутньому [48].

2.3 Аналіз якості програмного забезпечення

Якість програмного забезпечення має кілька визначень, як було показано раніше у ВСТУПі. У SWEBOK, у розділі «Основи якості програмного забезпечення, моделі та характеристики якості», «Якість програмного продукту» на сторінці 10-4 (або 177 у друкованому вигляді) також зазначається, що якість програмного забезпечення визначається вимогами клієнта до якості [17]. Автори

вважають, що вимоги до якості мають вирішальне значення для оптимального використання програмного забезпечення зацікавленими сторонами.

Вимоги до якості є нефункціональними вимогами до програмного забезпечення. Згідно з розділом SWEBOK [17] “Вимоги до програмного забезпечення”, “Основні вимоги до програмного забезпечення”, стор. 1-1 або 32, в загальному випадку, вимога – це властивість об’єкта, якій має задовольняти інструмент для вирішення проблеми реального світу. Важливою частиною будь-якої вимоги є можливість перевірки. Автори розрізняють 2 об’єкти вимог [17] “Вимоги до програмного забезпечення”, “Вимоги до продукту та процесу”, с. 1-2 або 33:

- продуктова вимога – «потреба або обмеження щодо програмного забезпечення, яке буде розроблено», тобто властивість кінцевого програмного продукту;
- процесова вимога – «обмеження щодо розробки програмного забезпечення».

Може бути 2 типи вимог [17], як зазначено у розділі “Вимоги до програмного забезпечення”, “Функціональні та нефункціональні вимоги” (стор. 1-3 або 34):

- функціональні – це вимоги, які описують функціональність програмного продукту;
- нефункціональні вимоги накладають “обмеження на рішення. Нефункціональні вимоги іноді називаються обмеженнями або вимогами до якості.”

Як розширення нефункціональних вимог, у розділі “Якість програмного забезпечення”, “Моделі та характеристики якості”, с. 10-3 або 176 [17] рекомендується модель якості програмного забезпечення з ISO/IEC 25010:2011. Ця модель ISO визначає 8 характеристик якості [61]:

- функціональна придатність;
- ефективність продуктивності;
- сумісність;

- зручність використання;
- надійність;
- безпека;
- ремонтопридатність;
- переносність.

Підводячи підсумок, якість програмного забезпечення не стосується того, що програмне забезпечення робить, а описує, наскільки добре програмне забезпечення це робить. Таким чином, усі зацікавлені сторони зацікавлені у вищій якості програмного забезпечення, як зазначено в розділі “Основи якості програмного забезпечення”, “Моделі та характеристики якості, якість програмного продукту” [17]. Крім того, якість програмного забезпечення є багатогранним поняттям, оскільки численні властивості впливають на якість програмного забезпечення, а моделі якості програмного забезпечення допомагають краще відстежувати різні аспекти якості програмного забезпечення.

На кафедрі програмної інженерії проводяться наукові дослідження у цьому напрямку. Зокрема, у роботі [3] розбираються усі можливі метрики, що можуть характеризувати програмне забезпечення. На відміну від моделі якості ISO, дані метрики дозволяються вимірювати якість програмного забезпечення та проєктування у визначених кількісних величинах.

2.4 Аналіз підходів до розповсюдження ПЗ

Усе програмне забезпечення можна об’єднати у 2 великі групи [62]:

- пропрієтарне або програмний продукт з закритим вихідним кодом розробляється оплачуваною командою спеціалістів з метою заробляння грошей і без розголошення деталей реалізації програмного продукту (наприклад, детальні технічні характеристики, вихідний код програмного забезпечення);
- програмне забезпечення з відкритим вихідним кодом, що має вихідний код загальнодоступним. Команда розробників може оплачуватися (як у випадку з пропрієтарним програмним забезпеченням) або складатися з

ентузіастів або поєднувати обидва підходи [9]. Програмне забезпечення з відкритим кодом можна монетизувати іншими способами.

Існують численні ліцензії, які можна застосовувати як до програмного забезпечення з закритим, так і до відкритого коду [63], однак обговорення юридичних деталей цих ліцензій не відноситься до теми цього дослідження.

Важко зрозуміти, чи якість будь-якого з типів програмного забезпечення перевищує якість іншого [62]. Однак методології для пропрієтарного та відкритого програмного забезпечення відрізняються [64]. Гнучка розробка програмного забезпечення та розробка програмного забезпечення з відкритим вихідним кодом часто поєднуються, особливо в гібридному програмному забезпеченні з не повністю відкритим вихідним кодом [9, 64].

2.5 Аналіз методологій розробки ПЗ

Важливими частинами життєвого циклу кожного програмного продукту є розгортання та доставляння (англ. delivery). Розгортання програмного забезпечення – це “складний процес пост-виробництва, який полягає в тому, щоб зробити програмне забезпечення доступним для використання, а потім підтримувати його в робочому стані” [65]. Наприклад, для публічного веб-сайту розгортанням буде завантаження вихідного коду HTML з усіма ресурсами на віддалений сервер, доступний через Інтернет, і моніторинг цього веб-сайту, щоб переконатися, що він є загальнодоступним у будь-який час з усіх місць і на всіх клієнтських пристроях. Для програми iPhone процес розгортання складатиметься з:

- для розробників: завантажити програму в App Store і переконатися, що програму все ще можна встановити;
- для користувачів: завантаження та встановлення програми на iPhone.

Доставлення програмного забезпечення є синонімом розгортання програмного забезпечення. У розгортанні програмного забезпечення наголос робиться на технічній частині забезпечення доступності програмного

забезпечення, тоді як доставлення означає факт надання програмного забезпечення для використання [65].

«Методологія розробки системи відноситься до структури, яка використовується для структурування, планування та контролю процесу розробки інформаційної системи». [66] Іноді методології розробки програмного забезпечення називають підходами [11]. Період часу програмного забезпечення, «життя» програмного забезпечення, називають життєвим циклом розробки програмного забезпечення (англ. software development life cycle), тому методології іноді просто називають SDLC, [11]. Більшість методологій розробки програмного забезпечення класифікуються як гнучкі та негнучкі або на основі плану [11]. Глобальна розробка програмного забезпечення представляє власні виклики та проблеми, які потрібно вирішити.

2.6 Аналіз негнучких методологій розробки ПЗ

Три найпопулярніші негнучкі методології, згідно з [11]:

- водоспадна або каскадна;
- ітеративна та інкрементна розробка [67];
- V-Model.

Водоспадна модель є найстарішою негнучкою традиційною методологією [68]. У контексті поточного дослідження сильною стороною моделі водоспаду є фаза, яка охоплює архітектурне планування – Проєктування системи (англ. System Design). Але в цій моделі важко внести зміни, оскільки зворотнє відкочування (англ. backtracking) між фазами не передбачено. Також, дана модель не описує спілкування. Опитування 2016 року показало, що 60% компаній використовують у своїх проєктах певні практики на основі плану або каскаду [11];

Ще одна традиційна модель розробки програмного забезпечення – V-Model [11]. Вона була незалежно розроблена у Німеччині як V-Modell [69] і в

США як «Vee» [70], але у поточному дослідженні зосереджуються більше на німецькому варіанті цієї моделі. За даними опитування 263 компаній у 2016 році, V-модель використовувалася в 32% випадків [11]. V-Model є розширенням водоспадної моделі розробки програмного забезпечення. У V-моделі діяльність після кодування «спрямована» вгору, а не вниз [71]. Ця модель має на меті покращити спілкування між зацікавленими сторонами та якість продукту завдяки перевірці результатів на кожному кроці. Однак V-модель також успадкувала свої недоліки від моделі водоспаду [69, 71].

Ітеративні методології належать до спеціальної власної групи, що включає RUP, Spiral, Agile [72, 73]. Традиційно, гнучкі методологій розглядаються окремо, але більш ранні ітеративні моделі є перехідними від традиційних. Основна ідея ітеративних методологій у побудові більш ефективної моделі комунікації. Однак, від цього може потерпати увага до початкового планування та документації [73]. Згідно з опитуванням 2016 року, 81% з 263 компаній використовували ітераційні підходи, відмінні від Agile.

2.7 Інтерпретування Agile-маніфесту

Agile-маніфест (Agile Manifesto) – це набір ідей, що лежать в основі Agile (або гнучких) методологій. Як зазначено в Історії: Agile-маніфесту [12], Маніфест містить певні конкретні ідеї, але є глибший задум, який керує багатьма членами альянсу. Манфест був складений у 2001 році, коли Agile-альянс зібрався в горах Васатч, штат Юта, щоб узагальнити ідеї Agile-методологій. Agile-альянс – це група незалежних мислителів щодо розробки програмного забезпечення, що є іноді й конкурентами один одному. Члени альянсу є представниками Extreme Programming, SCRUM, DSDM та інших методологій, які є прихильниками необхідності в альтернативі важких методологій процесу розробки програмного забезпечення, що керуються документацією, як зазначено в Історії: Agile-маніфесту [12].

Сам Маніфест є дуже коротким документом. З метою збільшення доступності його перекладено багатьма мовами [12]. Він складається з 4 тез і 12 принципів, які розвивають тези. Чотири тези включають:

- люди та співпраця важливіші за процеси та інструменти;
- працюючий продукт важливіший за вичерпну документацію;
- працюючий продукт важливіший за вичерпну документацію;
- готовність до змін важливіша за дотримання плану.

Дванадцять Принципів, що стоять за Agile-маніфестом [12] описують, як організований процес і як пріоритезуються його складові, як керувати гнучким процесом розробки програмного забезпечення, наскільки важливо залучати клієнтів, змінювати та адаптувати вимоги, співпрацювати, часто доставляти працююче ПЗ, мотивувати зацікавлених сторін та спілкуванися з ними [12]. Зокрема, принципи включають: «Постійна увага до технічної досконалості та гарного дизайну підвищує гнучкість» і «Найкращі вимоги, архітектурні та технічні рішення виникають у командах, що здатні самоорганізовуватись». Отже, в Agile-маніфесті зазначено важливість хорошого, але гнучкого дизайну та архітектури.

2.8 Аналіз гнучких методологій розробки ПЗ

Під час опису своєї історії, Agile-маніфест посилається на кілька методологій гнучких (або Agile) методологій [12]. Це означає, що маніфест було створено після використання багатьох гнучких методологій. Згідно з [11], до 2016 року серед 263 компаній 4 найпопулярніші гнучкі підходи та фреймворки за використанням:

- Scrum (86%);
- Kanban (64%);
- DevOps (62%);
- XP (49%).

Кілька досліджень у 2009–2010 роках констатують, що п’ять найпопулярніших гнучких підходів – це Scrum (10,9%), гнучке моделювання

(Agile modelling, 6%), керована функціями розробка (feature-driven development, 3,8%), керована тестуванням розробка (test-driven development, 3,4%), екстремальне програмування (2,9%) [72]. Дослідження [11] є більш свіжим, через що зібрана в ньому статистика є більш актуальною. Тому буде проаналізовано 4 найпопулярніші підходи зі списку.

2.8.1 Аналіз Scrum

Відповідно до [72] та [11], Scrum (або скрам) є найпопулярнішою методологією. У той же час на нього посилається Agile Manifesto у 2000 році під час опису своєї історії, що робить його однією з найстаріших гнучких методологій: Scrum вперше використано на практиці в 1995 році [74].

Офіційна діаграма робочого процесу Scrum (рисунок 2.1) показує, що центральною частиною діаграми є цикл, який концептуально дуже схожий на узагальнену діаграму ітераційного процесу розробки [74]. На відміну від традиційних методологій, Scrum не описує етапи створення частини програмного забезпечення (дизайн, впровадження, тестування). Натомість Scrum описує, як зацікавлені сторони повинні взаємодіяти один з одним і як має доставлятися розроблене програмне забезпечення.

Основними принципами Scrum є прозорість, інспекція та адаптація [74], що дуже схоже на концепції, введені в Agile-маніфесті [12].

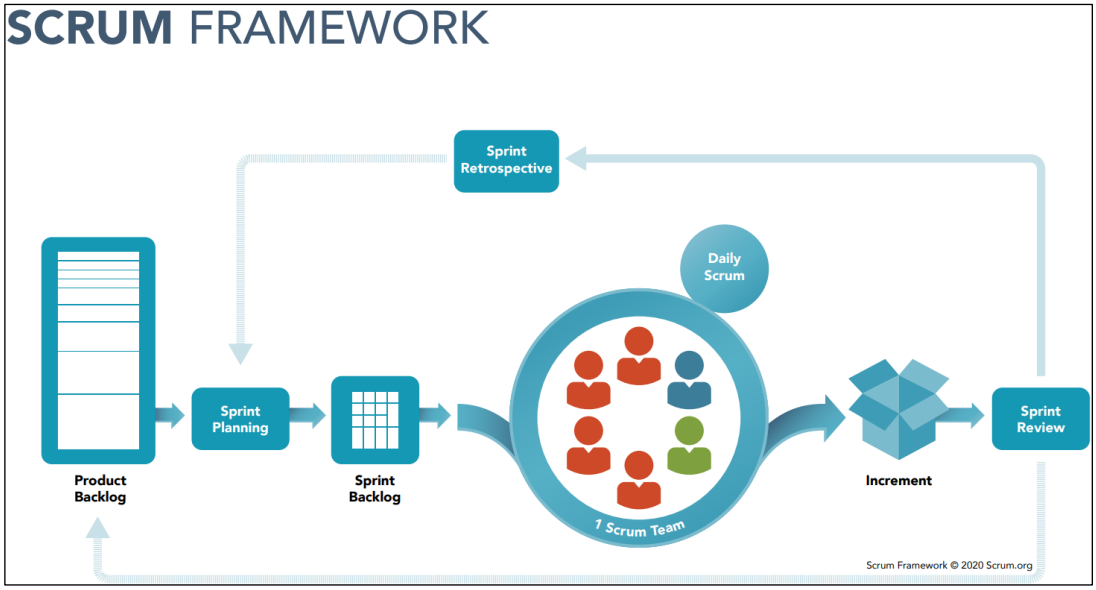

Рисунок 2.1 – Методологія розробки ПЗ Scrum [74].

Люди, залучені до розробки програмного забезпечення на основі Scrum (відповідальники або англ. accountabilities), включають [74]:

- Scrum Master – особа, відповідальна за розвиток ефективності процесу (синя особа на циклі діаграми);
- Product Owner – особа, відповідальна за програмне забезпечення якість продукту (зелена особа на циклі діаграми);
- розробники – команда технічних спеціалістів, які впроваджують програмний продукт (червоні силуети на циклі діаграми).

Відповідальників часто називають командою Scrum.

Scrum також має 3 динамічні об’єкти для встановлення та підтримки процесу – артефакти [74]. Усі 3 з них можна знайти на:

- беклог продукту – список функцій, які складають готовий програмний продукт - Мета продукту. Цей список можна змінювати;
- беклог спринту – список функцій, які мають бути реалізовані під час заданої ітерації, спринту, щоб досягти цілі спринту;
- інкременти – результат, результат кожного заданого спринту, який є частиною програмного забезпечення на шляху до досягнення цілі продукту. Це описано Визначенням готовності (англ. Definition of Done).

Події на основі часу, відомі як Події Scrum (англ. Scrum Events) [74], проілюстровано на і включають:

- спринт (англ. Sprint) – короткий кількатижневий цикл, протягом якого виконується робота [74]. Sprint містить інші події Scrum. Між спринтами немає перерв, як тільки завершується попередній спринт, починається наступний;
- планування спринту - зустріч на самому початку спринту, присвячена плануванню шляхом визначення цілі спринту та беклогу спринту для майбутнього спринту;
- щоденний Scrum – щоденна зустріч для розробників для обговорення того, що було зроблено на шляху до мети спринту. Назва зустрічі та

самої методології походять від регбі, де сутичкою (англ. scrum) є взаємодія гравців «для просування м’яча вперед» [74];

- перевірка спринту (англ. Sprint Review) – зустріч команди Scrum і «основних зацікавлених сторін» [74] наприкінці кожного спринту для перевірки та обговорення інкременту та результатів спринту. Під час зустрічі може бути оновлено беклог продукту, а також ціль продукту;
- ретроспектива спринту – зустріч команди Scrum для обговорення результатів спринту та змін до процесу розробки програмного забезпечення для наступного спринту для підвищення ефективності, якщо це можливо.

Звичайно, Scrum не обмежується цими поняттями, і багато інших концепцій можна безкоштовно знайти в Словнику Scrum [74] або Покажчику Scrum [74, 75]. Але жоден із ресурсів не обговорює процес розробки програмного забезпечення так, як це робив водоспад із дуже чіткими фазами розробки програмного забезпечення. Натомість звертається увага на те, що процес всередині спринту має бути встановлений розробниками. Завдання формуються шляхом декомпозиції беклогу продукту, щоб завданнями було легше вправляти [75].

2.8.2 Аналіз Kanban

Методологію Scrum можна розширити другою за популярністю гнучкою методологією [11] – Канбан (Kanban). У той час як Scrum більше зосереджується на регулярності інкрементів програмного продукту, Kanban зосереджується на плині завдань, тобто описує, як завдання повинні ефективно переходити від беклогу до доставлення, як показано у покажчику Scrum з Kanban [74]. Отже, ці дві методології можуть доповнювати одна одну без особливих протиріч. Є навіть посібник про те, як поєднати ці 2 методології – Посібник Kanban для команд Scrum [74].

Канбан (Kanban) має довгу історію. Його вперше було використано в 1950-х роках у виробничій системі JIT (англ. Just-In-Time, точно вчасно, в процесі) на заводах Toyota [76] для оптимізації процесу складання шляхом маркування

компонентів [76]. Його використовували як «ощадливу» техніку, яка спрямована на «усунення відходів» [76]. Вважається, що Канбан як методологія розробки програмного забезпечення була винайдена в 2010 році Девідом Дж. Андерсоном, коли до нього звернулися представники Microsoft, щоб краще візуалізувати свою діяльність з розробки [76].

Існує 3 основні принципи Kanban [76]:

- "Почніть із того, що маєте зараз – це ваш поточний процес";
- "Погодьтеся дотримуватися еволюційного підходу до змін і вдосконалення";
- "Поважайте поточні ролі та обов'язки команди/організації".

Основоположні принципи формують ментальність, необхідну для впровадження Kanban на практиці. Сам процес визначається 5 основними принципами, згідно з Девідом Дж. Андерсоном [76]:

- "Візуалізуйте роботу та подальший робочий процес";
- "Обмежити роботу в процесі (англ. WIP, work-in-progress) за допомогою віртуальної системи Kanban";
- "Керувати плином." Плин у Kanban регулюється за допомогою закону Літтла: чим менше речей опрацьовується одночасно, тим швидше їх можна завершити, як подано в покажчику Scrum з Kanban [74];
- "Зробіть політики керування явними";
- "Використовуйте моделі та науковий метод і вдосконалюйте спільно".

Методологію Kanban не слід плутати з дошкою Kanban – невід’ємним інструментом для організації робочого процесу Kanban [77]. Дошку Kanban можна використовувати в Scrum та інших гнучких методологіях без запозичення додаткових практик із Kanban [78].

Важливим параметром ефективності процесу розробки Kanban є Обмеження (ліміт) роботи в процесі (WIP Limit): відповідно до закону Літтла, поданому в покажчику Scrum з Kanban [74], кількість завдань, що виконуються, має бути обмежена уникнути розпорошення ресурсів. На рисунку 2.2 ліміт роботи в процесі встановлено на 2, тому можна розпочати ще одне завдання (перемістити з

колонки «Беклог» у колонку «В процесі»). Є також 2 метрики, характерні для Канбану, як можна побачити в на рисунку 2.2 [79, 78]:

– час циклу (англ. Cycle Time) – середній час виконання завдання, тобто час виконання завдання;}
– час виконання або проходження (англ. Lead Time) – середній час, протягом якого клієнт отримує запитану функцію, тобто час між створенням завдання в беклозі та його виконанням.

Зрозуміло, чим нижчі метрики, тим краще.

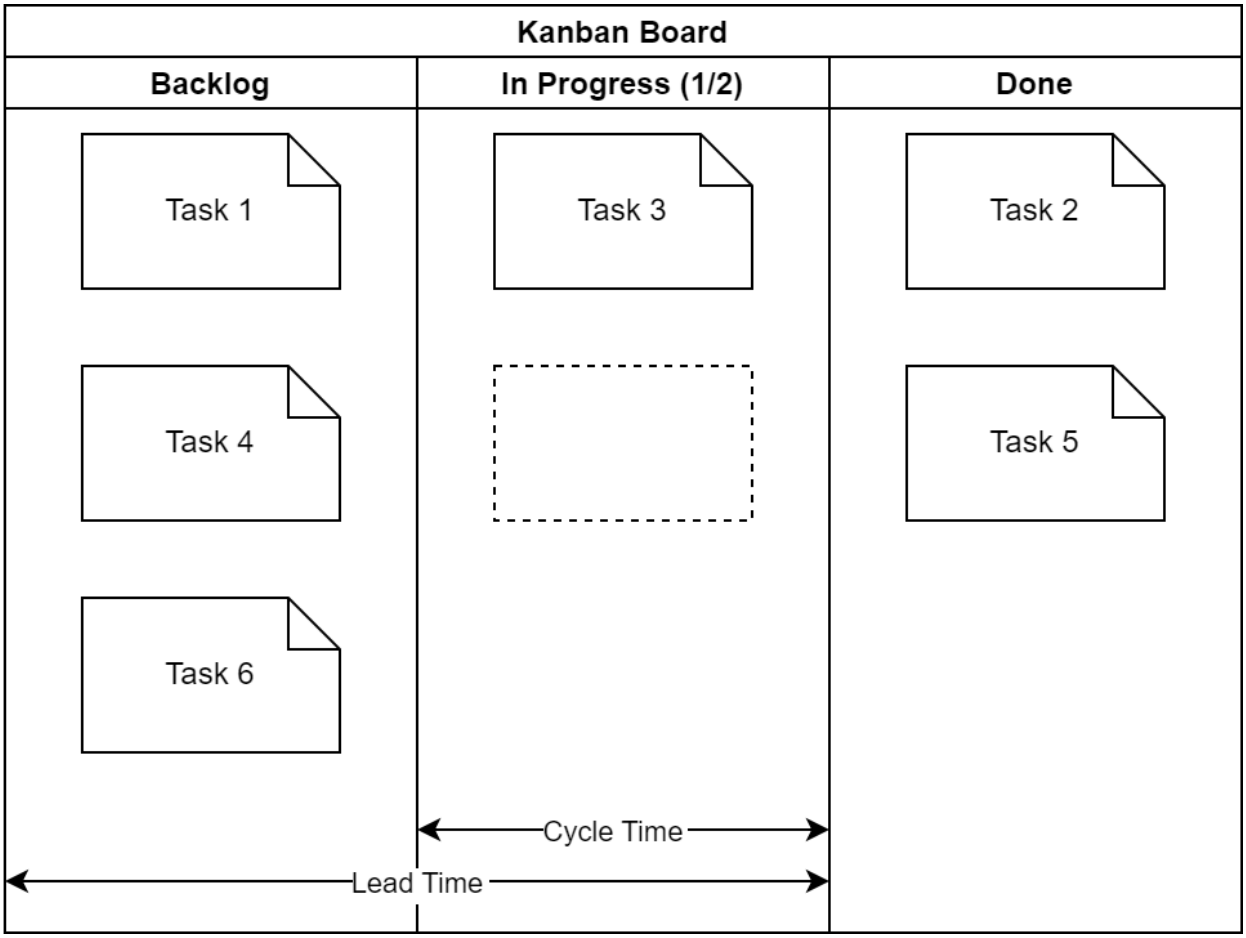

Рисунок 2.2 – Процес розробки ПЗ Kanban [79].

І Scrum, і Kanban використовують велосіті (англ. velocity) – відносну метрику, яка показує кількість завдань, завершених під час ітерації [80]. Це допомагає створювати схожі за розміром завдання та передбачити кінець проєкту.

По суті, Scrum з Kanban робить спринт у Scrum «м’якшим», запроваджуючи плин, а також визначення робочого процесу [81]. І Scrum, і Kanban, і Scrum з

Kanban зосереджені на комунікації між зацікавленими сторонами, але, незважаючи на їхню популярність [11, 72], жоден із них не роз'яснює, як робити ані проєктування архітектури програмного забезпечення, ані особливості комунікації під час проєктування архітектури. Як було зазначено в [76]: «Гнучкі методи фокусуються на коді, а не на проєктуванні».

2.8.3 Аналіз DevOps

Третій найпоширеніший гнучкий підхід– це DevOps (англ. Development & Operations) [11]. Концепція, як і назва DevOps, з'явилися в 2007 – 2009 роках [82]. Назва DevOps складається з 2 частин – developers і operations, тобто розробники та операції, що підкреслює інтеграцію між цими двома родами діяльності під час розробки ПЗ [83, 82]. Це відносно нова концепція, яка вибухнула у популярності в дослідницьких публікаціях у 2015 року [83]. Немає загальноприйнятого визначення DevOps згідно з [83]. Отже, ми будемо використовувати те саме визначення, яке використовували вони:

DevOps – це спільне міждисциплінарне зусилля в рамках організації для автоматизації безперервного доставлення нових версій програмного забезпечення, гарантуючи їх правильність і надійність.

Команда розробників працює над створенням нових функцій, тоді як оператори зосереджені на підтримці та налаштуванні програмного продукту у виробництві [83]. Різний характер діяльності цих двох команд призводив до проблем із комунікацією та затримок у доставлянні, що перешкоджало використанню гнучких методологій [83]. Крім того, інші методології Agile не охоплюють розгортання [83]. Навіть Extreme Programming не має практик, які зосереджені на розгортанні [83].

Через відсутність єдиного визначення DevOps, [83] визначає концептуальну мапу. Як видно з рисунку 2.3 і визначення, DevOps об'єднує кілька областей для підвищення ефективності. DevOps сприяє в [83]:

- підкресленні та обстоєнні співпраці та спілкування;
- обміні знаннями та інструментами.

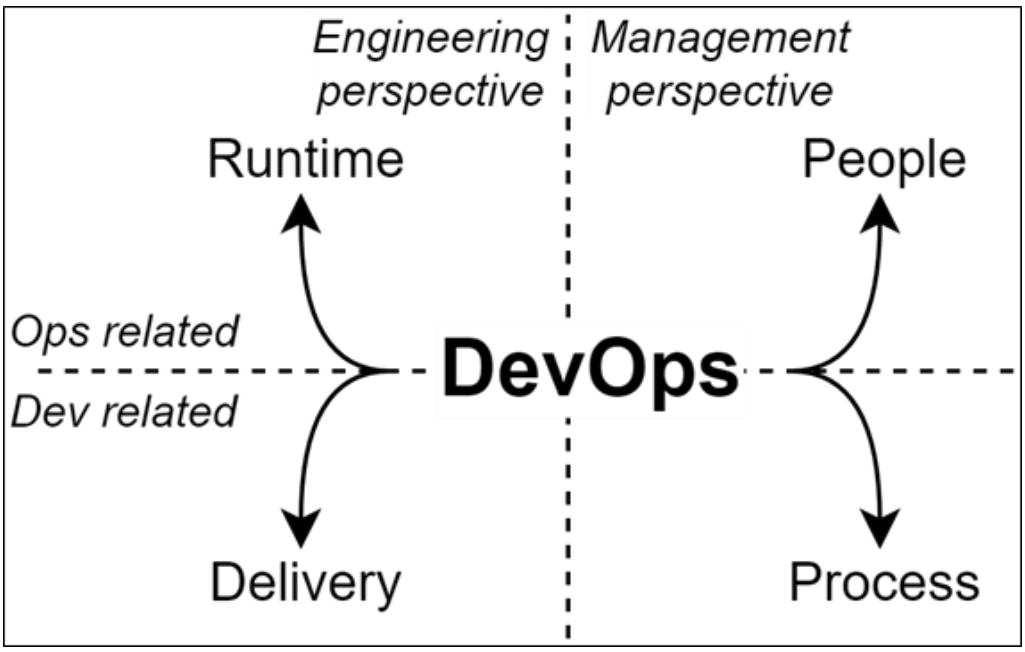

Рисунок 2.3 – Загальна концептуальна мапа DevOps [85].

Ці концепції реалізовано за допомогою набору практик [83, 84]:

– безперервна розробка – використання поступового та відстежуваного підходу до створення нового вихідного коду, використання систем контролю версій;
– безперервне тестування – регулярне тестування щойно створеного та зміненого коду;
– безперервна інтеграція – автоматичне відстеження готовності коду до виробництва. Його можна реалізувати за допомогою системи автоматичного збирання (компілювання), запуску тестів і аналізу коду за допомогою різних інструментів;
– безперервне доставлення та безперервне розгортання - автоматичне розгортання та доставка розробленої функціональності до середовищ тестування та виробництва;
– інфраструктура як код – визначення та надання інфраструктури програмного забезпечення (наприклад, баз даних, сховищ файлів, контейнерів програм) за допомогою файлів конфігурації.

Іноді під DevOps люди розуміють CI/CD (англ. Continuous Integration/Continuous Delivery або Deployment) – безперервну

інтеграцію/безперервне доставлення (або розгортання). Однак усі практики, окрім безперервної розробки, мають власні артефакти та конфігурацію, над якими потрібно співпрацювати, зберігати, створювати резервні копії та версії. GitOps пропонує використовувати Git (найпопулярнішу систему контролю версій [85]) як єдине джерело правди, таким чином максимізуючи практику інфраструктуру як код [86].

Таким чином, DevOps і GitOps зосереджуються на автоматизації та прискоренні шляху від написання коду до надання нових функцій кінцевим користувачам. Однак DevOps не приділяє уваги архітектурному плануванню. ArchOps вирішує цю проблему, використовуючи артефакти архітектури як відправну точку для автоматизації та практики DevOps [87]. Але ArchOps не зосереджується на процесі розробки архітектури та комунікації, яка в ньому задіяна.

DevOps є чудовим розширенням для інших гнучких методологій [83]: у той час як Scrum і Kanban зосереджуються на тому, як завдання мають бути створені та завершені, DevOps допомагає зробити процес завершення завдань більш гладким або навіть безперебійним.

2.8.4 Аналіз Екстремального програмування

Четвертим за популярністю архітектурним підходом є екстремальне програмування [11], яке часто називають XP (англ. eXtreme Programming, екстремальне програмування) [88]. Екстремальне програмування було винайдено наприкінці 1990-х років Кентом Беком. Назва походить від основної ідеї екстремального програмування – використання найкращих практик до екстремального [88]. Оригінальне визначення:

XP – це легка методологія для малих і середніх команд, які розробляють програмне забезпечення в умовах невизначених або швидко мінливих вимог.

Концептуальна модель екстремального програмування складається з цінностей, принципів і практик (рисунок 2.4) [88].

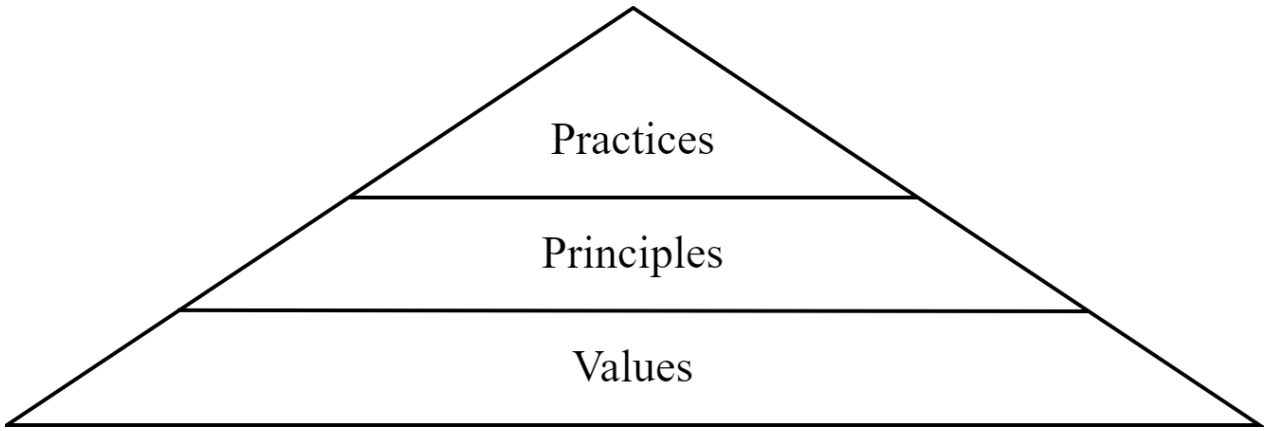

Рисунок 2.4 – Концептуальна модель екстремального програмування [88].

Основною концепцією є Цінність (англ. Values) [88]:

- комунікація – сприяє обміну знаннями;
- простота – перше рішення має бути найпростішим, дотримуючись підходу "Вам це не знадобиться" (англ. YAGNI, You aren't gonna need it) і уникання обширного проєктування наперед;
- зворотній зв’язок від системи (звіти про автоматичне тестування, сповіщення про помилки, сповіщення про збірку тощо), від клієнта (часті зустрічі, приймальне тестування, планування тощо), від команди (нові оцінки після зміни вимог);
- сміливість – не боятися змін у вимогах, системі (видалення непотрібного коду, а не коментування), у зворотному зв'язку (не приховувати неприємних істин);
- повага – порядне ставлення до себе, своїх колег, проєкту та всіх зацікавлених сторін.

Існує більше ніж 10 Принципів, які розширюють Цінності XP [88], але все ще залишаються досить абстрактними та загальними. Фактичні дії описані в Практиках Екстремального програмування [88].

Хоча XP не має чітко визначеної часової структури, як це робить Scrum. XP упорядковує дії за проміжком часу, як показано на рисунку 2.5. Ці практики можуть прямо відповідати певним етапам циклів планування чи зворотного зв’язку. Існує багато практик XP [88]. Для стислості ми згадаємо лише основні

практики XP. Основними практиками XP (з відповідними етапами циклу планування чи зворотного зв'язку в дужках) є Зібратися разом, Повна команда, Інформативний робочий простір, Енергійна робота, Парне програмування, Історії користувачів, Щотижневий цикл (План ітерації), Квартальний цикл (План випуску), Ненатягнення (англ. Slack), Десятихвилинна збірка, Безперервна інтеграція (Приймальне тестування), Тестування до реалізації (юніт-тест, англ. unit-test), Інкрементальний дизайн.

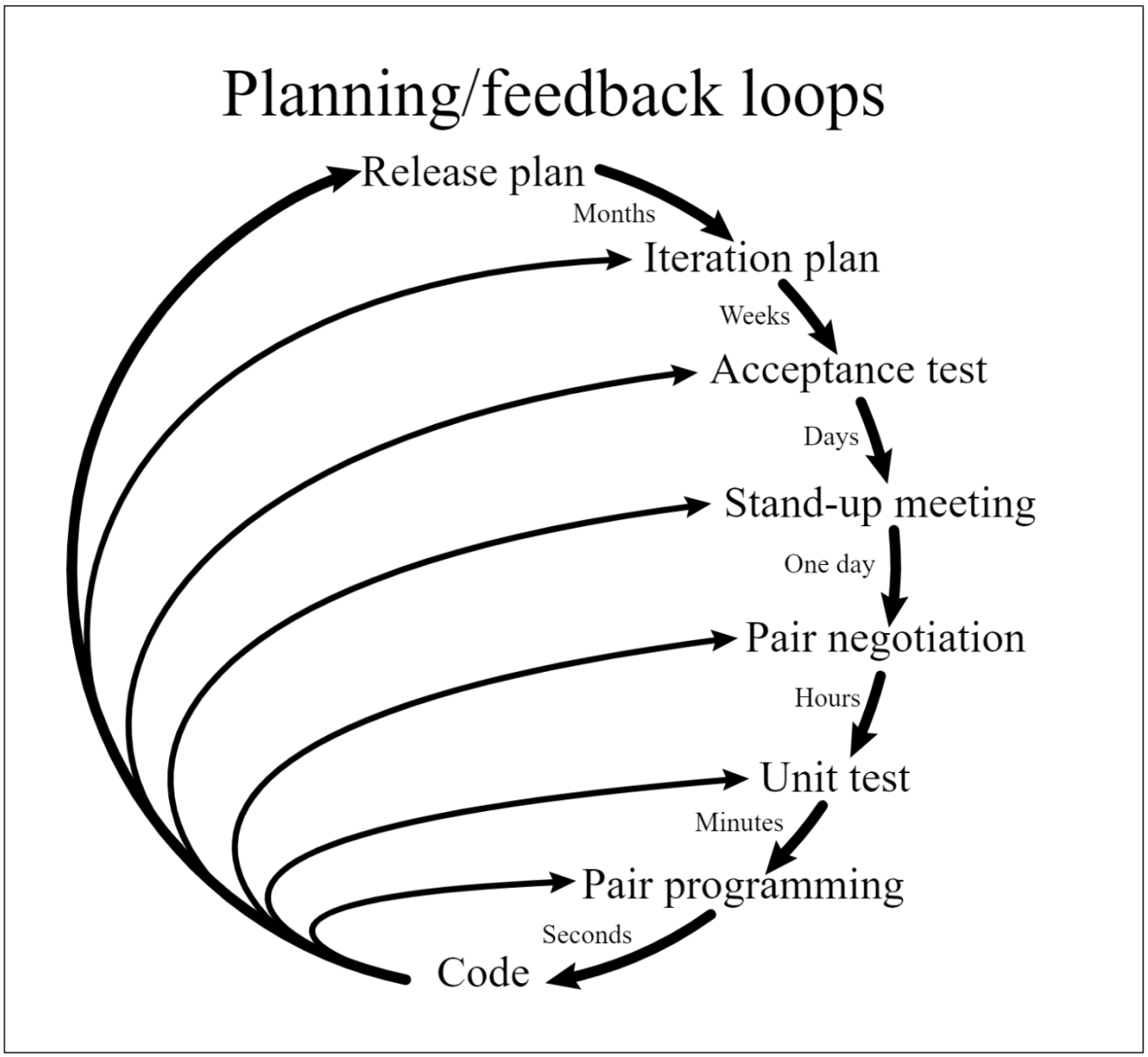

Рисунок 2.5 – Процес розробки програмного забезпечення Extreme Programming [88].

Основні практики ефективно регулюють безперервну розробку, тестування та інтеграцію. Сильною стороною основних практик є орієнтація на комунікацію:

4 (Зібратися разом, Повна команда, Інформативний робочий простір, Історії користувачів) з 13 практик зосереджені саме на спілкуванні між членами команди, а також іншими зацікавленими сторонами [88], що відрізняє XP як від Scrum, так і від Kanban. Інші практики можна умовно згрупувати таким чином [88]:

- енергійна робота, Парне програмування, Ненатягнення (Slack) (додавання малопріоритетних завдань для використання повної потужності) - ефективна розробка коду, співпраця та управління часом;
- десятихвилинна збірка, Безперервна інтеграція, Тестування до реалізації – автоматизація коду та підвищення надійності;
- тижневий цикл, Квартальний цикл, Інкрементальний дизайн – структура часу.

Незважаючи на численні практики, які регулюють спілкування, Екстремальне програмування не охоплює проєктування і архітектуру програмного забезпечення. Крім того, основні практики не охоплюють безперервне розгортання та доставку, але DevOps чудово доповнює XP у цьому плані [83].

2.9 Підсумовування аналізу гнучких методологій

Часто використовуються гібридні методології [11]. Усі 4 вищезазначені гнучкі методології можна комбінувати одна з одною:

- Scrum можна поєднати з Kanban, що забезпечує потік задач з Kanban для Scrum-спринтів, як наведено у Посібнику Kanban для команд Scrum [74];
- DevOps ефективно управляє артефактами проєкту та сприяє поширенню зворотного зв’язку, що робить його гарним доповненням до будь-якої методології Agile [83], як-от Scrum чи Kanban;
- екстремальне програмування може бути доповнено безпереревним розгортанням/доставленням з DevOps. Якщо DevOps зосереджується на управлінні результатами кодування, то XP пропонує ефективний спосіб створення надійного коду;

- XP також може розширити Scrum, запровадивши певні практики, оскільки Scrum не встановлює спосіб виконання роботи всередині спринтів [89];
- екстремальне програмування також може бути розширено Канбаном, створюючи методологію Extremeban [90]. Враховуючи ефективність завзятої команди розробників і тісну співпрацю відповідно до значень XP, автори підвищили ефективність за допомогою еволюційного потокового процесу Kanban [90].

Гнучкі методології добре з'єднуються, що підтверджується тим фактом, що існує єдиний Agile-маніфест [12].

З іншого боку, жодна з чотирьох найпопулярніших гнучких методологій або їх комбінацій не зосереджена на проєктуванні архітектури. Традиційні методології, такі як Waterfall або V-Model, використовують дизайн та архітектуру програмного забезпечення як одну з точок фокусу, але їм бракує ефективного спілкування та зворотного зв'язку. Цей недолік особливо очевидний у глобально розподілених командах, де члени команди можуть мати різні менталітети, отже, по-різному сприймають речі [11]. Можливо, іноді відсутні відносно суворі планування та документація разом з ГРПЗ можуть означати відродження традиційних методологій розробки програмного забезпечення [11]. Крім того, понад 80% досліджень між 2013 і 2015 роками називають комунікацію найбільшим фактором, що впливає на гнучку архітектуру [91].

2.10 Аналіз сучасних процесів розробки архітектури ПЗ

Архітектура програмного забезпечення була визначена у ВСТУПі. У дослідженні [92] дуже детально було досліджено процес проєктування архітектури. Один із їхніх висновків полягає в тому, що існує лише одне визначення архітектури програмного забезпечення, яке враховує час – ISO/IEC/IEEE 42010:2011 [10][4)], що співпадає з висновком цієї роботи. Але це не

[4] Не було доступу до цього платного стандарту, тому було використано транзитивне дослідження з цитуваннями.

означає відсутність досліджень, які досліджували б архітектурні зміни чи еволюцію з часом. Систематичний огляд літератури у 2012 році [93] свідчить про те, що з 1992 року зацікавленість і кількість публікацій про еволюцію архітектури програмного забезпечення постійно зростає [93]. Переглянуті дослідження були значною мірою зосереджені на формулюванні концепції та розвитку архітектури програмного забезпечення.

У 2010 автори наголошують на важливості відстеження змін програмного забезпечення та відповідному оновленні архітектури [94]. Це допомагає уникнути дегенерації архітектури [94]. Вони також представляють Схему характеристики зміни архітектури програмного забезпечення (англ. Software Architecture Change Characterization Scheme, SACCS) для ефективного відстеження змін програмного забезпечення в архітектурі. У 2016 році було пояснено високий інтерес до зміни та еволюції архітектури програмного забезпечення інтернет-природою сучасних програм [95]. Як наслідок, перша згадка про еволюцію архітектури програмного забезпечення в стандарті ISO/IEC/IEEE 42010:2011 не виглядає недоречною, оскільки з 1990-х років дослідження в цій галузі визначили важливість архітектури програмного забезпечення.

Тим не менш, існує багато підходів до архітектури з технічної точки зору [96, 14]. Це включає в себе кілька способів опису архітектури, де UML (англ. Unified Modelling Language, Уніфікована мова моделювання) є найпопулярнішою [32], проте існує ще кілька Мов опису архітектури (англ. Architecture Design Language, ADL) як альтернативи. Існує також кілька підходів, шаблонів і найкращих практик, які можна використовувати під час проєктування архітектури. Одними з найвідоміших шаблонів є архітектура клієнт-сервер та MVC (Model-View-Controller). Наразі фокус змістився на Доменну архітектуру програмного забезпечення (англ. Domain Specific Software Architecture, DSSA) та атрибути якості. Досить відомими варіантами DSSA є:

– мікросервісна архітектура – односерверна програма (монолітна) розділена на кілька менших серверів відповідно до їхнього домену та функцій, які вони надають. Це спрощує технічне обслуговування та

сприяє швидкій розробці, автоматичному розгортанню та масштабуванню під навантаженням;

– безсерверна або serverless – крок вперед для мікросервісної архітектури, де монолітний сервер розділений на конкретні функції, які виконуються в окремих контейнерах. Обслуговування серверної інфраструктури здійснюється провайдерами Інтернет-хостингу.

Ще двома архітектурними напрямками є КПР та КТР, дослідження яких також було проведено й на кафедрі Програмної інженерії [2].

Таким чином, з технічної точки зору, індустрія накопичила багато підходів і методів для вирішення реальних проблем при роботі з архітектурою програмного забезпечення. Це не означає, що її розвиток зупиниться, але технічна частина архітектурного проєктування вже добре висвітлена.

Крім того, починає з'являтися нова концепція – гнучка архітектура [97]. Було виконано систематичний огляд літератури та складено досить великі списки проблем, факторів успіху, практик і рекомендацій щодо гнучкої архітектури. На додачу, існує значна кількість книг, які висвітлюють сферу сучасної архітектури [98, 99, 14]. Ще обговорювалося Сховище керування знаннями про архітектуру [100] – місце для зберігання важливої архітектурної інформації та опису рішень, однак визначення концепції не є дуже чітким.

2.11 Аналіз комунікації у програмній інженерії

Не було знайдено єдиного визначення комунікації в програмній інженерії. Однак ця тема була широко висвітлена в інших дослідженнях. Наприклад, було проведено широке дослідження ефективної комунікації та обміну знаннями в розробці програмного забезпечення [101]. У дослідженні було обрано низку аспектів спілкування для розгляду, таких як пояснення, розуміння, пригадування («розпізнавання або пригадування знань із пам'яті для отримання або відновлення раніше вивченої інформації» [101]) і Співробітницьке міжособистісне спілкування. Співробітницьке міжособистісне спілкування включає [101]:

– активне обговорення: запитання, інформування та мотивація інших;

- творчий конфлікт: сварка та міркування у формі дискусії;
- керування розмовою: координація та підтвердження переданої інформації.

Отже, аналізуються різні аспекти комунікації в розробці програмного забезпечення та формулюється як проактивний структурований діалог [101]. Вказується на важливість розумового процесу пошуку та інтерпретації своїх знань. Однак основним фокусом цього дослідження є використання текстових або графічних засобів для підвищення продуктивності спілкування. Вони показують, що в більшості сценаріїв графічне представлення інформації, пов'язаної з програмним забезпеченням, є кращим, але краще описати міркування чи іншу описову інформацію в текстовій формі.

Відповідно до [102], софт скіли (англ. soft skills, м’які навички) – це набір навичок, які дозволяють продуктивно співпрацювати й працювати в команді. З цього дослідження можна зрозуміти, що ефективне спілкування значною мірою залежить від якості софт скілів. Проводиться СВД, щоб отримати набір категорій, а саме: Навички спілкування, Робота в команді, Аналітичні навички, Навички організації та планування, Навички міжособистісного спілкування, Лідерство, Навички вирішення проблем, Автономність, Прийняття рішень, Ініціатива, Керування конфліктами, Управління змінами, Зобов'язання та Відповідальність, Керування стресом, Орієнтація на клієнта, Гнучкість, Етика, Орієнтація на результат, Керування часом, Інновації, Навички презентації, Креативність, Критичне мислення, Навички ведення переговорів, Навички слухання, Мотивація, Бажання вчитися, Здатність то швидкого навчання, Управління командою, Методичність. Згідно з [102], Комунікативні навички є найбільш потрібними серед софт скілів і згадуються в 92% всіх знайдених досліджень. Інші навички, які зустрічалися в більш ніж 50% досліджень: Робота в команді (62%), Аналітичні навички (55%), Навички організації/планування (55%) і Навички міжособистісного спілкування (52%). Автори [102] визначили навички спілкування як:

Спілкуватися в усній і письмовій формі простим, лаконічним, недвозначним і зрозумілим способом. Набір навичок, який дозволяє людині передавати інформацію так, щоб її сприйняли та зрозуміли. Здатність передавати інформацію так, щоб вона була добре сприйнята та зрозуміла. Здатність ефективно спілкуватися з іншими.

Обидва дослідження нещодавні: дослідження [101] проводилося в 2020, а [102] – у 2019. Це означає більшу ймовірність того, що їхні висновки все ще будуть актуальними. Поєднуючи висновки обох документів, міжособистісне спілкування в розробці програмного забезпечення визначається як процес обміну інформацією та знаннями між зацікавленими сторонами та розробниками програмного забезпечення з метою розробки, виробництва та підтримки програмного забезпечення. Процес спілкування може передбачати використання текстових, графічних засобів і засобів дистанційної роботи. Крім того, технічні спеціалісти мають враховувати нетехнічний досвід інших зацікавлених сторін [102].

На кафедрі Програмної інженерії було проведено дослідження задля покращення обігу документації, тобто форми непрямого спілкування. Експеримент був проведений на підприємстві Харківобленерго [5]. У результаті подання системи спілкування у вигляді графу та покращення певних ланцюжків спілкування, було зменшено кількість документації на 11%. Безумовно, Харківобленерго не є компанією РПЗ, але таку ж саму графову модель можна застосувати задля покращення документації під час РПЗ.

2.12 Дослідження додаткових понять

Було виявлено, що загальновживаний термін для позначення недоліків у будь-якому програмному забезпеченні називається борг. Ймовірно, технічний борг є найбільш вживаним терміном. Поняття технічного боргу порівнюється з фінансовим (з посиланням на джерело №1 Cunningham публікації [103]):

Відправити код уперше [в рамках проєкту] – це все одно, що зайти в борг. Невеликий борг пришвидшує розвиток, якщо він швидко повертається шляхом

перезапису. Об'єкти роблять вартість цієї операції прийнятною. Небезпека виникає, коли борг не повертається. Кожна хвилина, витрачена на не зовсім правильний код, зараховується як відсотки за цим боргом.

У дослідженні [103] детально описано природу технічного боргу та підкреслено, що він має бути контрольованим. У роботі [104] мова йде про архітектурний технічний борг – архітектурний аспект технічного боргу, те, що впливає на архітектуру. Однак, на відміну від суто технічного боргу, архітектурна заборгованість може бути спричинена неправильним спілкуванням між зацікавленими сторонами: менеджери та розробники, ймовірно, говорять різними мовами [104].

Соціальний борг – ще одна концепція, знайдена під час пілотного огляду літератури [13, 26, 28]. Робота [27] є спеціальним дослідженням для визначення соціального боргу як концепції. Автори визначають це як щось, що впливає на процес розробки, але не випливає з будь-якого технічного рішення. Аналогія взята з соціальних наук, «соціальний борг суспільства являє собою набір напружених соціальних відносин, які виникають як наслідок обставин боржник-кредитор» [24].

Соціальні та технічні аспекти будь-якого рішення пов’язані між собою [24]. Зокрема, було представлено поняття соціально-технічну конгруентністі ([105] з посиланням на джерело №12 Cataldo et al.): зміни в соціальній або технічній частині соціально-технічної конгруентності викликають зміни в іншій, «ефект гумової стрічки». Згідно з їхнім визначенням, «узгодження між вимогами до координації, витягнутими з технічних залежностей між завданнями, та фактичною діяльністю з координації, яку виконують інженери». Соціально-технічна конгруентність базується на законі Конвея. Закон Конвея стверджує, що організація розробників програмного забезпечення формує структуру та архітектуру розробленого програмного забезпечення.

Отже, технічний, соціальний та архітектурний борг пов'язані один з одним. Було вирішено зобразити зв’язок між цими трьома поняттями за допомогою діаграми Венна [106], як можна побачити на рисунку 2.6.

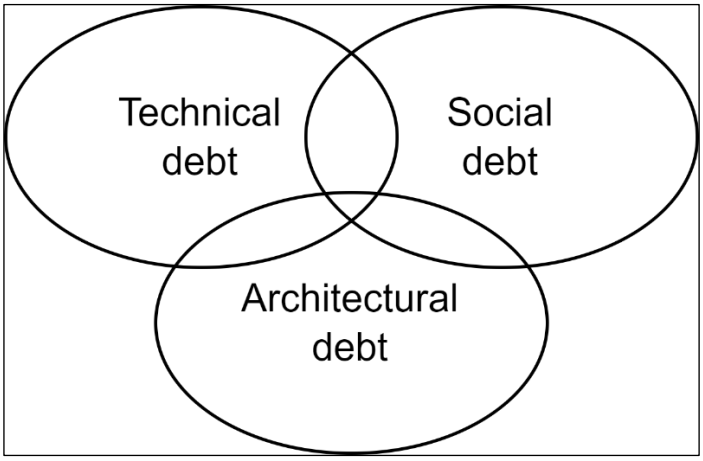

Рисунок 2.6 –Зв'язок між технічною, соціальною та архітектурною заборгованістю як Діаграма Венна.

Зображення досить схематичне, але видно, що певні проблеми (борг) можна класифікувати як один вид боргу або одночасно два чи три види боргу.

3 ОПИС ДОСЛІДНИЦЬКОГО ПІДХОДУ

Основна мета дослідження полягає в тому, щоб знайти, чи існує існуючий інструмент для усунення комунікаційних недоліків під час проєктування архітектури програмного забезпечення. Ще краще, якщо такий користується хоча б помірною популярністю, що має підтверджуватися статистикою. Популярність гіпотетичного інструменту означає:

- перевірку, тому що вважається, що галузь не буде використовувати проблемний інструмент;
- відсутність необхідності досліджувати додаткові інструменти, оскільки вважається, що було б легше популяризувати вже існуючий інструмент.

Для переконання, що такий інструмент не буде пропущений і для отримання актуального розуміння сучасного стану у цій галузі, слід проаналізувати цільову літературу [29]. Якщо інструмент не надто популярний, слід ще ретельніше вивчати літературу, щоб не пропустити його. Відповідно до [29] існує 2 основні підходи до огляду літератури: систематичне відображувальне дослідження (СВД, англ. Systematic Mapping Study, SMS) і систематичний огляд літератури (СОЛ, англ. Systematic Literature Review, SLR). Обидва наукові методи пов’язані з пошуком джерел, видобуванням, аналізом інформації та підсумуванням отриманих результатів. Однак SLR є більш глибоким, тому було вирішено рухатися вперед із SLR як основним науковим методом поточного дослідження.

3.1 Опис наукового підходу

Систематичний огляд літератури визначається [1] як «огляд літератури з визначеними дослідницькими питаннями, процесом пошуку, вилучення та представлення даних». Таким чином, СОЛ передбачає глибоку роботу з дослідженням у досить структурований спосіб для вилучення певних ідей, як підтверджено [29]. За їхніми визначеннями, дане дослідження має схожість із систематичним відображувальним дослідженням, оскільки мета цього дослідження полягає в тому, щоб зрозуміти сучасний стан, знайти «можливості

для дослідження» та переглянути посилання на статті строгим способом. Водночас для СВД характерні дуже широка предметна область, тоді як поточне дослідження відповідає на дуже конкретні запитання, що більш характерно для СОЛ. Крім того, проводиться порівняння між статтями, а також проводиться робота над створенням теорії шляхом «викрадення» (англ. “abduction”) – «узагальнюючи різні теорії, знайдені в літературі та пов’язані з різними експериментальними результатами».

Підсумовуючи все вищесказане, вважається, що провели систематичний огляд літератури. Дехто може заперечити, що дане дослідження певною мірою схоже на СВД, з чим можна погодитися. Тим не менш, СОЛ буде науковим методом цього дослідження з усіма припущеннями. Вважаємо виконаний СОЛ метод якісним (англ. qualitative) дослідницьким методом, оскільки SLR може бути якісним, кількісним або обома [29]. Але перш ніж перейти до процесу та протоколу SLR, необхідно описати, який інструмент шукаємо як відповідь на ДП2.

3.2 Візія ідеального розв’язання (ДП2)

У дослідницькому питанні №2 (ДП2) повідмляється про необхідність пошуку та аналізу існуючих інструментів. Але перш ніж знати, що потрібно знайти, необхідно це описати. Бачення (візія) інструменту дозволить у більш структурований, надійний і відтворюваний спосіб порівнювати та класифікувати існуючі інструменти.

Було розглянуто кілька методологій, які як належать, так і не належать до Agile, у підрозділі 2.6 Аналіз негнучких методологій розробки ПЗ, де було надано широкий аналіз на основі багатьох джерел. Згідно зі зробленими висновками, традиційні методології розглядають архітектурний дизайн як основний аспект, але вони не встановлюють ефективний і відгучний (англ. responsive) діалог між зацікавленими сторонами протягом усього проєкту, і вони не мають достатньо хорошого зворотного зв'язку. Гнучкі методології, з іншого боку, дуже сильні у сфері встановлення ефективної комунікації. Однак гнучкі підходи, зокрема

екстремальне програмування та DevOps, не охоплюють частини детального планування [88, 83]. Вважається, що це принаймні часткова причина, чому методології на основі плану використовуються в поєднанні з підходами Agile.

У результаті аналізу гнучких методологій вважаємо, що було б природним розширити екстремальне програмування за допомогою певних методів економічного чи гнучкого планування [88]. Практики XP вже охоплюють такі сфери, як спілкування, співпраця, ефективна розробка коду та автоматизація. Вважається, що проєктування архітектури вже підпадає під кілька категорії, як були виявилені раніше (див. 2.10 Аналіз сучасних процесів розробки архітектури ПЗ), такі як:

- колективна робота, яка потребує спілкування між зацікавленими сторонами;
- відсутність потреби в обширному дизайні, їхній дизайн і архітектура мають бути ефективними, частково через їх інкрементальний характер.

Крім того, автоматизація на всіх етапах розробки програмного продукту є однією з ключових ідей XP, DevOps і CI/CD. Якщо команди, які дотримуються інших методологій, бажають прийняти ці потенційні практики планування, вони можуть просто поєднати XP із методологією, яку вони використовують, як це було показано раніше в цій пояснювальній записці.

Маніфест Agile [12] не відкидає планування, натомість подається намір зробити його розумним, ненав’язливим і неблокуючим. Наразі існує багато інструментів для полегшення технічного процесу проєктування архітектури, але без поточного СОЛ важко сказати, чи є достатньо інструментів для спілкування та співпраці під час проєктування. Крім того, автори даного дослідження не мали досвіду використання незагальних інструментів направленних на встановлення процесу колективного проєктування архітектури під час роботи в індустрії.

Вважається, що оптимальний підхід до покращення архітектурного планування та комунікації полягає в більш індуктивному підході, знизу вгору: спочатку зосередитися на дрібних проблемах із прийняттям рішень і некоректною комунікацією, докладаючи зусиль для визначення більш загальних вказівок для

кожного конкретного проєкту. Це, на додачу, більше відповідає методології Agile, оскільки такий підхід можна вважати ітераційним. Як розширення цього підходу, до технічного боргу можна підійти по-новому або принаймні дещо змінено. Прикладом інструменту, про який ми говоримо, може бути підхід до сховища знань про архітектуру, представлений [100], але вважається, що визначення репозиторію керування знаннями про архітектуру було не дуже чітким.

Сутність проблеми полягає в тому, що сувора документація є невід'ємною частиною традиційних методологій розробки програмного забезпечення, таких як Waterfall або V-Model, і, зокрема, «важкість» документації та відсутність частого спілкування стали причиною винаходу гнучких підходів. Оскільки гнучкі підходи уникають надмірної документації, новий підхід має відповідати цій парадигмі та не вимагати багато часу для читання та оновлення. Ось чому дане дослідження передбачає щось на зразок веб-форми для заповнення. Форма служитиме двом цілям:

- визначенню, чи було прийнято рішення;
- вказанню конкретних деталей прийняття рішення, його причин та наслідків.

Відтепер цей гіпотетичний ідеальний інструмент називатиметься інструмент на основі форм (ІнОФ). Цей інструмент ні в якому разі не є панацеєю від усіх проблем з документацією, технічної заборгованості та ненавмисних рішень [104], але він показує хороший приклад простого та ефективного інструменту. Процес заповнення форми можна пришвидшити за допомогою розкривних списків, прапорців, перемикачів та інших способів обмежити вибір і уникнути відкритих питань [107]. Це дозволяє автоматично класифікувати дані, уникаючи відкритих питань. Використання закритих запитань замість відкритих робить вхідні дані більш оптимізованими та легшими для маніпулювання й аналізу [108]. Звичайно, певна описова інформація втрачається в закритих питаннях, і розробникам може знадобитися докласти трохи зусиль, щоб класифікувати рішення, які вони роблять. Але це має перевагу: розробники починають бути більш обережними та свідомішими до рішень, які вони

приймають. У результаті зростає усвідомленість прийняття рішень, що є важливою частиною ефективної комунікації [19].

Наприклад, підхід реалізації знизу до верху, про який було згадано раніше, буде створювати інструмент на основі форм для збору даних з кожного завдання (англ. task). Ці дані потрібно агрегувати, щоб побачити загальні тенденції прийняття деяких глобальних рішень. Вважається, що це може бути хорошим способом змінити парадигму мислення та дозволити документувати рішення, пов'язані з боргами [104]. Більше того, наявність такого інструменту на основі форм – це спосіб побачити навмисні (свідомі) і ненавмисні (несвідомі) рішення щодо проєктування архітектури, що викликають борги [104]. Обдумані рішення створюють менше ризиків і менші оцінки часу для виконання завдання, що може допомогти в плануванні. Крім того, це може допомогти зменшити кількість соціальних боргів [27] шляхом створення «спільної точки зору» для людей, які беруть участь у дискусії, і зробивши процес спілкування більш структурованим [22]. Можна припустити, що такий гіпотетичний інструмент може бути швидко прийнятий компаніями, судячи зі швидкості впровадження гнучких методологій [11, 72].

Вважається, що зручно зберігати всі зібрані дані в одному місці для аналізу, так званому єдиному джерелі правди (англ. single source of truth). Тому першою ітерацією ІнОФ можуть бути лише Google Forms і Google Spreadsheet. Перевагою цих інструментів є їхня безкоштовна бізнес-модель, простота та популярність, що додатково спрощує усунення неполадок у разі потреби [109]. Крім того, Google Forms збирає дані в єдину електронну таблицю Google Spreadsheet, де вихідними даними можна легко маніпулювати та ретельно аналізувати їх, як підтвердили [109]. Якщо класичних інструментів аналітики та візуальних електронних таблиць недостатньо, аналіз можна розширити за допомогою SQL-подібної мови запитів [110, 111].

Згідно з [100, 112], групове програмне забезпечення (англ. groupware) – це назва типу програмного забезпечення, яке полегшує взаємодію між членами команди і не обмежується лише аудіо- та відеоконференціями. Цей тип

програмного забезпечення зазвичай інтегрований із суміжними програмами, такими як веб-редактори тексту, керування документацією та вікі-ресурси. Одними з найвідоміших сучасних прикладів такого ПЗ є Slack і Microsoft Teams. Вважаємо природним, що ІнОФ, як передбачається, інтегрується з таким груповим програмним забезпеченням. Розраховується, що групове програмне забезпечення в контексті розробки програмного забезпечення дозволить додатково розширити такий інструмент за допомогою корисних посилань, посилань на вихідний код і швидкого доступу до додаткових ресурсів проєкту.

Короткий опис ІнОФ буде таким:

- веб-форма, яка збирає інформацію в основному за допомогою закритих питань і спадних списків, прапорців і перемикачів. Поля введення тексту дозволять додавати описові деталі або збирати відгуки. Загальне знайомство користувача з цим підходом зробить інструмент легким для прийняття. Гарним прикладом інструменту опитування через Google Forms;
- інструмент, який збирає дані з форми в сховище даних, що дозволяє статистичний аналіз і візуалізацію. Гарним прикладом інструменту збору даних є Google Spreadsheets;
- інструмент, який можна інтегрувати з сучасним груповим програмним забезпеченням, таким як Microsoft Teams або Slack [112], який може розширити функціональність ІнОФ шляхом інтеграції з артефактами, специфічними для розробки програмного забезпечення;
- гарне доповнення до екстремального програмування, яке дозволить XP охопити також архітектурне планування.

Інструмент, заснований на формі, збирав би інформацію знизу вгору – отримуючи невеликі фрагменти інформації для кожного виконаного завдання, які, накопичуючись, можуть показати загальну картину. Крім того, наявність простого чітко визначеного інструменту зробить спілкування більш структурованим.

Візія така, щоб ІнОФ був таким же простим і ефективним, як засоби відстеження часу: є кілька спадних меню для швидкої класифікації рішення або

аспекту архітектури, а вхідні дані дозволяють прудко посилатися на інші ресурси і людей, швидко вибирати дату та час. Не вважається проблемою для розробників витратити менше 15 хвилин на опис кожного важливого рішення. Використання спадних меню має ще одну перевагу: називання проблем – це половина шляху до їх вирішення [104]. В ідеалі хотілося б залишити лише одну можливу причину спричинення дрейфу архітектури – відсутність кваліфікації – та усунути інші причини. Вважається, що використання надійної моделі, якою є гіпотетичний ІнОФ, для використання під час обговорення, може різко зменшити кількість непорозумінь і соціального боргу. Відсутність кваліфікації дуже важко виправити, можна лише пом’якшити шляхом перерозподілу обов’язків і призначення нових осіб для прийняття рішень [104], що цілком логічно: неможливо відрізнити правильне від неправильного в незнайомих сферах.

Під час пілотного огляду виявили кілька інструментів, які стосувалися комунікаційних проблем або прийняття рішень, зокрема:

- у дослідженні [26] представляє DAHLIA (англ. Debt-Aimed arcHitecture-Level Incommunicability Analysis), який допомагає вимірювати некомунікабельність архітектури. Це дозволяє обчислити обізнаність про певні рішення, враховуючи відсотки вхідних даних, пов’язані з обізнаністю про рішення, щоб розрахувати соціальний борг і некомунікабельність;
- автори [15] представляють Фреймворк інтересів (Concern Framework) разом із контрольним списком, який дозволяє ідентифікувати та класифікувати окремі виклики в «проєктуванні архітектури програмного забезпечення в глобальній розробці програмного забезпечення». Деякі з цих проблем мають практики, які допомагають їх виправити. Виклики (англ. challenges) та практики (англ. practices) називаються занепокоєннями або інтересами, згрупованими за темами (англ. theme) та відображеними в концептуальній моделі в рамках Фреймворку інтересів. Їхнім методом дослідження був СОЛ;

– у своїй наступній статті [22] представили фреймворк Глобальної архітектурної практики розробки програмного забезпечення (ГАП, англ. GAP, Global Software Development Architectural Practice). Структура ГАП значною мірою базується на Фреймворці інтересів, але вона була перевірена опитуваннями з галузевими архітекторами. Структура ГАП також містить «інтереси» (англ. concerns – проблеми та практики), згруповані в теми та відображені в концептуальній моделі. Концептуальна модель дуже схожа на модель з Фреймворку інтересів. Незважаючи на схожу структуру, структура ГАП є більш ретельною та обширною.

Всі три інструменти виглядають цікаво. Однак у них є ряд недоліків:

а) жоден з них не було перевірено на практиці, тобто в реальній компанії. Звичайно, дані були зібрані за допомогою SLR та опитувань безпосередньо від спеціалістів індустрії, і, згідно з інтерпретацією їхніх досліджень, фреймворки були розроблені для впровадження галузі. Але незрозуміло, як галузь прийме ці фреймворки, автори не згадують жодних відгуків чи результатів застосування фреймворків;

б) усі три фреймворки мають значні апріорні вимоги (англ. prerequisites):

1) DAHLIA повністю покладається на обчислення статистичних показників, але незрозуміло, як можна зібрати необхідні статистичні дані. Можливо, ІнОФ можна використати для збору даних. Крім того, автори DAHLIA не надають багато пояснень щодо того, як розглядати та аналізувати числовий соціальний борг і числове значення «некомунікативності» (англ. incommunicability);

2) Фреймворк інтересів і ГАП дозволяють детально аналізувати архітектурні проблеми в ГРПЗ. Однак важко сказати, як цей інструмент сприйме індустрія, оскільки інформації про це немає. Крім того, вважається, що ці структури з їх контрольними списками вимог (англ. checklists) можуть бути надто складними для практичного повсякденного застосування. Але надбання цих фреймворків,

безумовно, можуть бути корисними під час розробки інструменту на основі форм.

Було вирішено провести систематичний огляд літератури через брак інструментів для прийняття рішень і структурованої комунікації. Три інструменти, які було спочатку знайдено під час пілотного огляду, можуть не відповідати посталеній меті, але вони можуть стати хорошим джерелом натхнення або розширенням для ІнОФ чи подібного інструменту.

3.3 Протокол систематичного огляду літератури

Систематичний огляд літератури – це науковий метод збору інформації про область дослідження [1]. Метою СОЛ є відповідь на одне чи декілька дослідницьких питань. СОЛ визначається [29, 113]:

- процесом – послідовність кроків, які визначають дії, які необхідно виконати під час дослідження (рисунок 3.1);
- протоколом – набір правил, які визначають спосіб збирання та класифікації літератури.

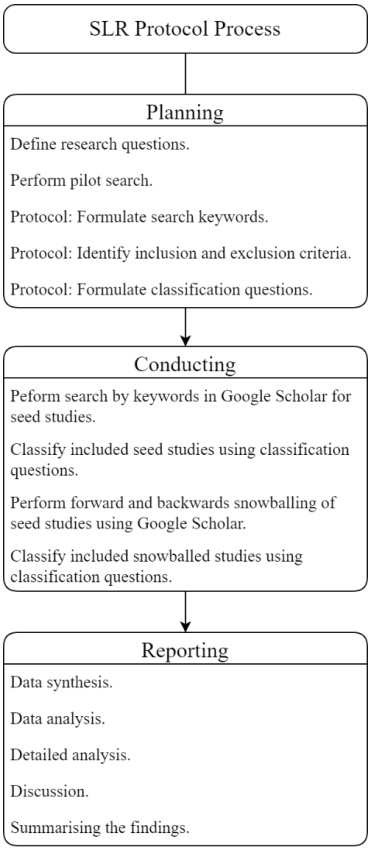

Рисунок 3.1 – Процес СОЛ (SLR process)

У цьому дослідженні використовуються 2 основні методи пошуку літератури під час СОЛ:

- пошук за пошуковим запитом (англ. search query) – з використанням уточненого набору ключових слів і однієї чи кількох пошукових систем наукової публікації [29];
- сніжний ком (англ. snowballing) – використання списку посилань публікацій для пошуку потенційних досліджень (назад направлений сніжний ком або англ. backwards snowballing) і використання засобів відстеження цитувань або пошукових систем для пошуку додаткових досліджень, які цитують дане дослідження (вперед направлений сніжний ком або англ. forwards snowballing) [114].

Було використано BibTeX, оскільки було використано LaTeX, зокрема OverLeaf, для роботи з літературою. Формат BibTeX дозволяє зберігати всі посилання, подібно до бази даних. Крім того, BibTeX можна легко конвертувати у формат Microsoft Excel, що дозволяє подальше перетворення в Google Spreadsheets [109].

Через те, що BibTeX підтримує експорт до Google Spreadsheets і Google Spreadsheets підтримують SQL-подібні запити [110, 111], бібліотеку джерел було експортувано у файл XLS за допомогою (доступно за посиланням). Потім було імпортувано файл XLS в Google Spreadsheets, і посилання можна знайти у списку джерел - [115], де міститься посилання на таблицю.

Через те, що процес класифікації має велику кількість питань та кожна з публікацій має низку атрибутів, що її характеризують, неможливо навести табицю цілком. Тому було вирішено навести скріншоти таблиці, при цьому розбивши таблицю на 3 різних частини. Скріншоти таблиці можна знайти на рисунках Д.1, Д.2 та 3.2.

У таблиці додатково було створено кілька аркушів та додано низку макросів Google Apps для автоматизації підрахунку статистики з можливістю подальшої візуалізації засобами Vega-lite [116]. Зокрема, таблиця містить аркуші:

- Footnotes – виноски;

– QuerySimple – для підрахунку статистики бінарних відповідей (так або ні) та несписковых значень (наприклад, тип рішення, можливість проведення кількісної оцінки, наявність перевірки на практиці);
– QueryMultiple – для підрахування статистики спискових значень (наприклад, валідність, джерело інформації, типи дріфту архітектури);
– SnowballingStats – статистика зібраних даних за допомогою методу сніжного кому.

Як видно з рисунків Д.1, Д.2 та 3.2, вони належать до однієї таблиці, але розбиті на кілька рисунків задля зручності.

Рисунок Д.1 демонструє загальні відомості (метадані) про кожну публікацію, рисунок Д.2 містить відповіді на більшість питань класифікаційного протоколу СОЛ, а на рисунці 3.2 показано інформацію щодо популяризації інструментів.

ID	Sol. Ref.?	Feedback?	Popularized?	Comment
dreesen2021se	No	Our: Vague	No	Social debt gen
razavian2019e	No	Our: Diff.	No	Decision makin
sievi2019softw	No	Our: Vague	No	Extensive, Distr
tamburri2019s	No	Our: Diff.	No	Decision incom
baabad2020so	Yes	Often & Rare P	No	Drift, Erosion
matsudaira201	No	Our: Vague	Yes	Applying XP Pra
orosz2020softv	No	Our: Vague	Yes	Different indus
behutiye2017a	No	Our: OK	No	Shows how em
hummel2015rc	No	Our: OK	No	-
tamburri2019e	No	Our: Diff.	No	Tweaking socia
capilla201610	No	Our: Vague	No	Shows history a
sarwar2020tov	No	Our: Diff.	No	-
soliman2021ar	No	Our: OK	No	Names the issu
wohlin2021tov	No	Our: Diff.	No	Evidence-based
tekinerdogan2	No	Our: Vague	No	Global Softwar
sievi2019challe	Yes	Our: Diff.	No	Global Softwar
ali2012framew	No	Our: OK	No	Groupware for

Рисунок 3.2 – Google Spreadsheet з даними класифікації зворотного зв'язку.

Таким чином, одна таблиця розбита на 3 рисунки.

3.3.1 Ключові слова для пошуку літератури

Пошук здійснюється англійською мовою, бо це є мова міжнародного спілкування та академічних досліджень, тому в поточному пункті не буде наводитися український переклад англійських слів. Більшість пошукових термінів було взято з вищезгаданої публікації [19], але адаптовано та доповнено ключовими словами за темою поточного дослідження, згідно з рекомендаціями [29] на слайді №31. Крім того, було використано рекомендації, подані [113]. Отже, пошукові терміни включатимуть принаймні один пошуковий термін із кожної групи нижче (операція AND):

- 'Architecture issues' OR 'Architecture problems' OR 'Architecture deterioration' or 'Architecture erosion' OR 'Architecture degradation' OR 'Architecture smell' OR 'Architecture anomaly' OR 'Architecture drift' OR 'Architectural decay' OR 'Architectural inconsistency' OR 'Architecture violation' OR 'Architecture change' OR 'Architecture inefficiencies' OR 'Architecture debt' OR 'Technical sustainability problem' OR 'Lack of technical expertise' OR 'Technical problems' OR 'Technical issues' OR 'Architecture ambiguity' OR 'Insufficient architecture' OR 'Architecture instability' OR 'Technical challenges' OR 'Architectural challenges' OR 'Architectural technical debt';
- 'Communication issues' OR 'Communication problems' OR 'Miscommunication' OR 'Social debt' OR 'Communication factor' OR 'Social factor' OR 'Human factor' OR 'Community smells' OR 'Osmotic communication' OR 'Communication impediment' OR 'Lack of communication' OR 'Social challenges' OR 'Communication challenges' OR 'Communication breakdown' OR 'Communication barrier'.

Важливо зауважити, що «коммунікація» (communication) має використовуватися у дослідженні в сенсі спілкування між людьми, а не в контексті комунікації компонентів програмного забезпечення.

Також були включені похідні форми слів. Це передбачає:

- прикметники та дієслова (наприклад, architecture та architectural, degradation та degraded та degrade). Порівняно з [19], ключові слова, які закінчуються на -al, інтерпретуються як одне ключове слово для стислості;
- множини (наприклад, issue та issues, problem та problems);
- порядок слів (наприклад, degradation of architecture та architecture degradation);
- розділення пошукових термінів іншими словами, не втручаючись у значення (наприклад, problems in relation to communication, Breakdowns in communication).

3.3.2 Критерії включення й виключення

Критерії для розуміння того, чи стосується публікація дослідницьких питань поточної роботи, мають бути строгими [29].

Отже, були обрані критерії включення та виключення:

- публікація має бути видана протягом останніх десяти років (2012 і пізніше). Це допомагає переконатися, що інформація в публікації є актуальною;
- мова публікації має бути англійською через суб’єктивний досвід авторів поточного дослідження. Крім того, англійська є визнаною міжнародною мовою;
- та сама публікація, знайдена за допомогою різних пошукових запитів, має бути включена лише один раз;
- публікації з основними частинами, які не згадують ключові слова, мають бути виключені.

У певній включеній статті достатньо інформації, щоб її можна було класифікувати відповідно до протоколу СОЛ: стаття описує СОЛ чи Оглядання (англ. Survey)) і описує розв’язання для знайденої проблеми.

3.3.3 Критерії якості

Метою даного дослідження не був пошук лише рецензованих досліджень, як рекомендовано у [29, 103]. Тому не відкидалися ненаукові роботи, але вважалися за більш неформальні. Їхня простота полегшувала роботу з ними. З іншого боку, вказівки щодо відбіру літератури для СОЛ [29] складніше дотримуватися під час пошуку публікацій із індустрії, оскільки:

- існують бази даних ненаукових публікацій, пов'язаних із розробкою програмного забезпечення, тому потрібно використовувати загальну пошукову систему, як-от Google [117], що знижує продуктивність;
- зазвичай у ненаукових публікаціях відсутні списки літератури та засоби відстеження цитувань, що робить їх майже непридатними для використання в техніках «сніжного кома» [114].

Під час пілотного огляду було помічено, що цю сферу досліджують представники як з індустрії, які мають більше практичного досвіду, так і з академічної сфери з більш науковим підходом (рисунок 3.3).

Зі знайдених публікацій було зроблено висновок, що вони зосереджені на різних аспектах питання:

- фахівці індустрії, як правило, використовують індуктивний підхід [25, 24]: вони зосереджуються на конкретних діях, щоб вирішити конкретну проблему, після чого намагаються екстраполювати рішення на подібні проблеми. Крім того, пояснення процесу їхнього мислення може бути відсутнім, і вони можуть пропустити перевірку на валідність, що є дуже важливим для наукового методу [118]. Це робить їх аналіз менш структурованим;
- науковий підхід є більш структурованим і дедуктивним [118]: метод і процес мислення чітко пояснюються, результати дослідження підсумовуються та узагальнюються за допомогою різних методик, а достовірність висновків перевіряється, як показано у [13, 22, 103, 4, 33, 10, 113, 104, 114, 30], що робить їх теоретичними.

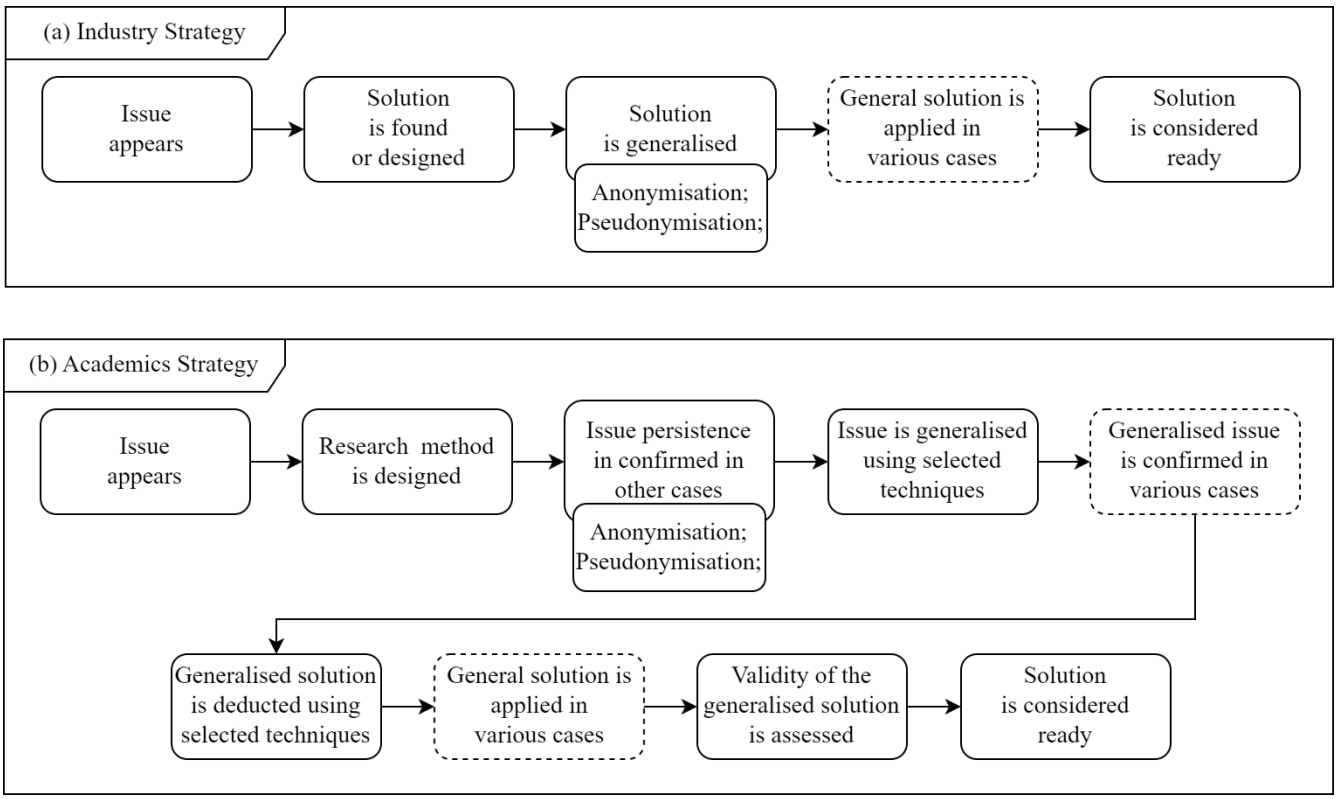

Рисунок 3.3 – Різниця у підходах до пошуку розв’язання між галузевими та академічними роботами.

Таким чином, є 2 підходи до пошуку рішення.

3.3.4 Класифікаційний протокол

Для виконання класифікації на основі вмісту потрібна надійна, але гнучка схема, як продемонстровано у [29]. Додаткове натхнення було знайдено у дослідженні [113]. Отже, було визначено набір правил для класифікації включених документів. Протокол класифікації:

а) які типи дрейфу архітектури згадуються в публікаціях?[5] (Типи дрейфу або англ. Drift Types):

1) змінено пріоритети архітектури (Pr. Ch., англ. Architecturing Priorities Change) – технічна проблема, коли підходи або пріоритети для проєктування архітектури були змінені, і новий напрямок може бути не оптимальним;

[5] Відсутність досвіду не враховувалася як тип дрейфу, оскільки це не проблема комунікації.

2) неточна документація (Inacc. Docs, англ. Inaccurate Documentation) може статися, коли документація не містить необхідної інформації через те, що вона закоротка, завелика або не містить певних важливих архітектурних аспектів;
3) мутація прийняття рішень (Mut. Dec., англ. Mutation of Decision Making), яка може статися, коли змінюється команда або відповідальні люди (англ. owners, «власники» певної частини проєкту) і розуміння поточної архітектури було спотворено, або коли рішення було неправильно зрозуміле іншими розробниками;
4) обмежена участь членів команди (Lim. Inv., англ. Limited Involvement of the Team Members) може статися, коли не налагоджено зв’язок між відповідальними людьми та іншими членами команди, які беруть участь у цій конкретній сфері;
5) нерозподілені обов'язки (Unass. Resp., англ. Unassigned Responsibilities) – ситуація, коли жодна особа не має відповідальності та права останнього слова в прийнятті рішень;
6) незрозумілий підхід до вирішення конфлікту (Confl. Res., англ. Unclear Conflict Resolution Approach) – це ситуація, коли незрозуміло, як слід вирішувати конфлікти;

б) як була отримана інформація? (Джерело або англ. Source):
1) інтерв'ю, опитування чи прикладне дослідження (Оглядання або англ. Survey)[6];
2) систематичний огляд літератури (СОЛ або англ. SLR);

в) чи є проблеми із зовнішньою, внутрішньою та структурною достовірністю знайдених проблем? Помітка: Val. Iss. (англ. validity issues). Можливі значення: External (від англ. – зовнішня), Internal (від англ. – внутрішня), Structural (від англ. – структурна) або None (від англ. – нічого, нема);

---

6 Інтерв'ю, опитування та тематичні дослідження об’єднано в одну групу, оскільки всі вони є первинними дослідженнями та використовують подібні методи дослідження [143].

г) чи піддаються кількісній оцінці проблеми? (Iss. Quantifiable?, англ. issues quantifiable);

д) який тип запропонованих рішень? (Sol. Type & Sol. Brief, англ. solution type & solution brief):

1) немає конкретного рішення, але є кілька порад (Advice, від англ. порада);
2) загальний набір практик або рекомендацій (Pr. Set, англ. practice set);
3) структура, яка дозволяє оптимізувати аналіз проблем за допомогою чітко визначеного набору перетворень і правил виведення (англ. Framework – помітка);

е) чи виявлена проблема чи запропоноване рішення навіяно Agile? Чи запропоноване рішення інтегровано з Agile? Помітка: Agile?; можливі значення Insp. (англ. inspired, надихено) або Integr. (англ. integrated, інтегровано);

ж) чи було запропоноване рішення перевірено на практиці? (англ. Practice?);

з) чи піддається кількісному виміру запропоноване рішення? (Sol. Quantifiable? , англ. solution quantifiable);

и) чи була забезпечена внутрішня, зовнішня та структурна валідність запропонованого рішення? Помітка: Sol. Val. (англ. solution validity); можливі значення: External (від англ. – зовнішня), Internal (від англ. – внутрішня), Structural (від англ. – структурна) або None (від англ. – нічого, нема;

к) чи є посилання на запропоноване рішення в інших публікаціях? (Sol. Ref.?, англ. solution referenced);

л) які відгуки про запропоноване рішення в інших публікаціях? Чи вважається запропоноване рішення занадто складним або нечітким? Які наші відгуки про рішення, що використовується на практиці? Помітка: Feedback?; питання описове, але може мати й кілька фіксованих варіантів: OK або Vague (від англ. розпливчастий) або Diff (англ. difficult,

складний). За відсутності відгуків може мати префікс «Our:» (від англ. наш);

м) чи були спроби популяризувати рішення? Це означає презентацію рішення на зустрічах з управління проєктами, розробки чи архітектури, створення публікацій у блогах, партнерство з компаніями для практичного використання інструменту [119] (Popularized?, від англ. популяризований). Це питання є ключовим для перевірки, чи була спроба інтегрувати запропоновані рішення в процеси в галузі.

Такий список класифікаційних питань дозволить звести більшість цікавої інформації до закритих питань з обмеженою кількістю варіантів, що в подальшому дозволить проводити підрахунок статистики простіше.

4 АНАЛІЗ РЕЗУЛЬТАТІВ ОГЛЯДУ ЛІТЕРАТУРИ

Після виконання пошуку та вибору публікацій для аналізу було отримано 17 джерл, а саме:

а) 7 статей отримано за допомогою методу пошукових запитів (search query, SQ);

б) 10 джерел було отримано методом «сніжного кома» (snowballing, SB):}

1) 7 джерел за допомогою сніжного кома, направленого назад (backwards snowballing, bSB), як показано в таблиці 4.2;

2) 3 публікацій через сніжний ком, направлений вперед (forward snowballing, fSB), як показано в таблиці 4.1.

4.1 Статистика методу сніжного кому

Статистику для методу сніжного кому було згруповано в таблиці для легшого аналізу (таблиці 4.1 – 4.2). Статистику було розраховано за допомогою тієї самої таблиці Google [115], аркуш SnowballingStats.

Таблиці 4.1 і 4.2 містять загальну кількість і кількість для кожної категорії для кожної публікації та для певної категорії для всіх публікацій (наприклад, Accepted від англ. прийнято, No Access від англ. нема доступу, Rejected by Missing Keywords від англ. відхилений через відсутні ключові слова) і враховується для кожної публікації для всіх категорій (наприклад, dreesen2021second [13]). Крім того, стовпець Total (від англ. усього) та рядок “Total:” у цих таблицях відображають загальну кількість публікацій для кожного автора та для кожної категорії відповідно. Підрахунок не включає дублікатів.

Таблиця 4.1 – Статистика СОЛ для методу сніжного кому, направленого назад.

	Backwards: Accepted	Backwards: Accepted and is Searched through Query	Backwards: Rejected for Other Reasons	Backwards: No Access	Backwards: Rejected by Year	Backwards: Rejected by Missing Keywords	Backwards: Total
dreesen2021second	3	1	1	1	34	21	61
razavian2019empirical	1	0	1	8	41	28	79
sievi2019software	3	0	2	3	31	12	51
Total:	7	1	4	12	106	61	191

Якщо таблиця 4.1 містить інформацію лише щодо назад направленого методу сніжного кому, то таблиця 4.2 об’єднує статистику щодо вперед направленого методу сніжного кому та загальну статистику для цього методу.

Таблиця 4.2 – Статистика СОЛ для методу сніжного кому, направленого вперед, та загальна статистика.

5	Forwards: Accepted	Forwards: No Access	Forwards: Not English	Forwards: Rejected by Missing Keywords	Forwards: Total	Total	Accepted from backwards, %	Accepted from forwards, %	Total accepted, %
dreesen2021second					0	61	4,92%		6,56%
razavian2019empirical	3	9	4	17	33	112	1,27%	9,09%	3,57%
sievi2019software					0	51	5,88%		5,88%
Total:	3	9	4	17	33	224	4,19%	9,09%	4,91%

Додатково, таблиця 4.2 містить загальну кількість і відсоток документів, знайдених за допомогою кожного методу для кожної публікації. Під час підрахунку прийнятих публікацій не були виключені дублікати, тобто були включені 2 категорії – Accepted (від англ. прийнято) й Accepted and is Searched through Query (від англ. прийнятно та знайдено через запит), оскільки хотілося бачити ефективність різних методів окремо від методу пошукового запиту СОЛ.

Для 3 публікацій було застосовано метод назад направленого сніжного кому, а для однієї статті – метод вперед направленого сніжного кому. Хоча не було отримано доступ до 12 із 191 (6,28%) публікацій протягом bSB, недоступними виявилися аж 9 із 33 (27,27%) публікацій під час застосування fSB. Не було знайдено способу збільшити кількість видань, до яких був доступ. Таблиця 4.2 показує, що було прийнято 4,91% усіх розглянутих джерел. 4,19% було прийнято під час від bSB та 9,09% було прийнято від fSB. Статистика показує, що вперед направлений сніжний ком є більш ніж удвічі продуктивніший, ніж назад направлений, навіть із більшою кількістю недоступних джерел.

4.2 Аналіз агрегованої статистики

Перш ніж приступити до аналізу змісту публікацій СОЛ, розглянемо демографічні [29] та інші метадані. Для цього було створено зручну електронну таблицю Google [115], як зазначено в 3.3 Протокол систематичного огляду літератури.

На рисунку 4.1 видно, що у літературі виконаного СОЛ немає публікацій за 2013, 2014, 2018 та 2022 роки. Найбільше публікацій було зроблено у 2019 році – 6 публікацій. Також 2015 та 2017 роки мають по одній публікації. Розподіл за типом публікації простіший: є лише одна публікація-тези конференції [13] і 16 журнальних статей, як показує рисунок 4.2.

Рисунок 4.3 демонструє, скільки розглянутих публікацій згадували певні типи дрейфу архітектури. Згідно з малюнком, найпоширенішим типом дрейфу є Мутація прийняття рішень, який зустрічається в 15 із 17 публікацій.

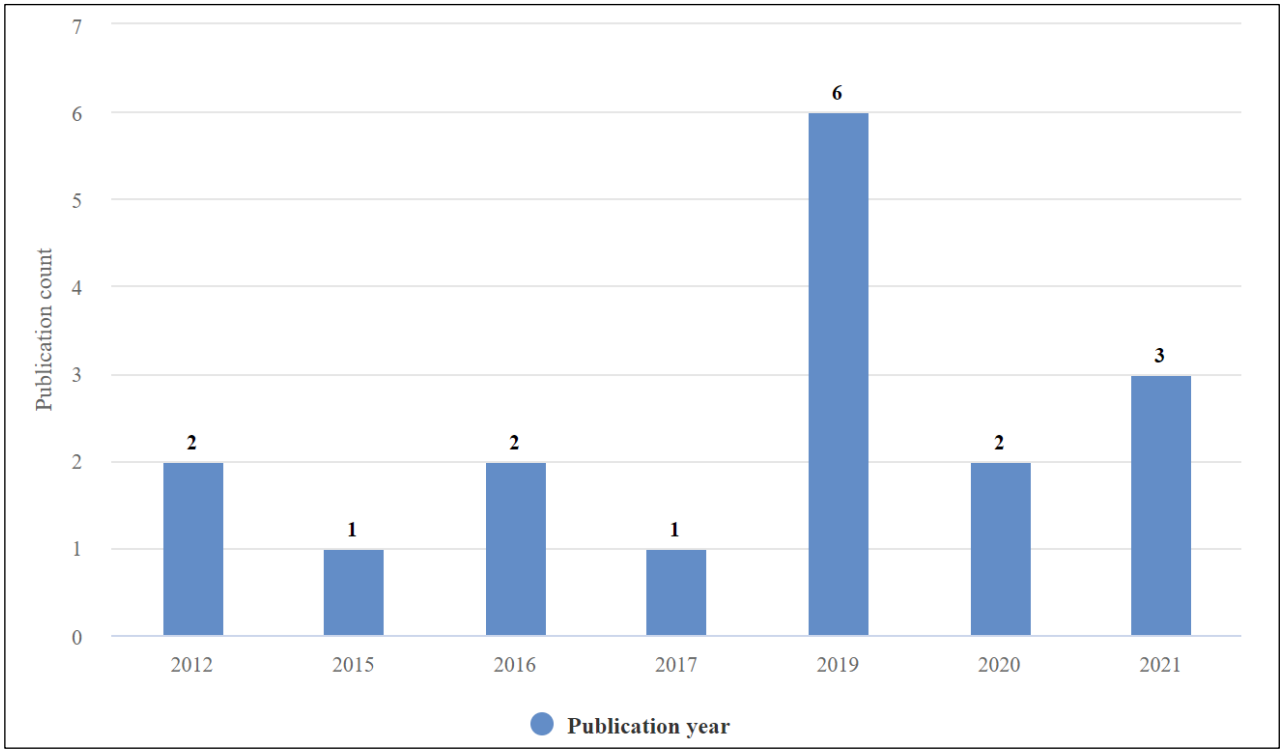

Рисунок 4.1 – Кількість публікацій СОЛ за роками.

Додатково слід зазначити, рисунки 4.1 – 4.2 були зроблені за допомогою бібліотеки Highcharts.

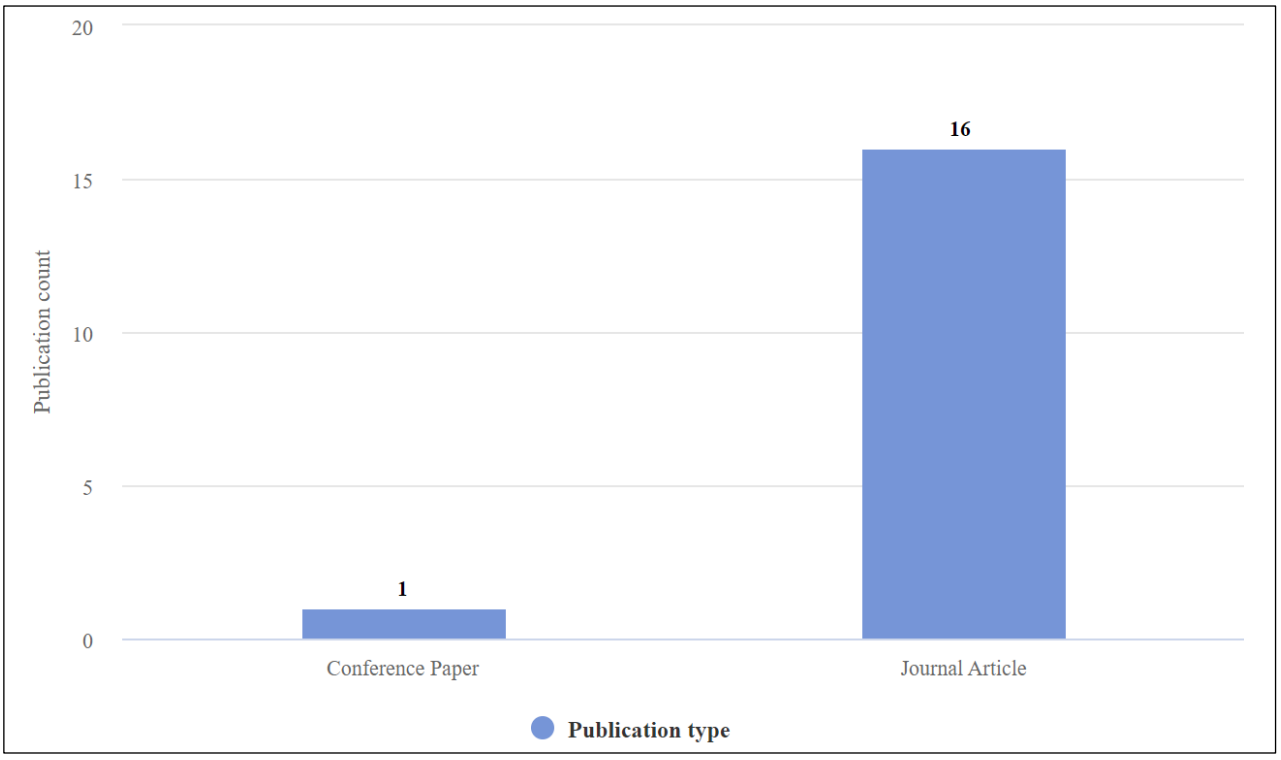

Рисунок 4.2 – Кількість публікацій СОЛ за типом.

Друга за частотою є архітектурна деградація спричинена Обмеженим участю членів команди. Цей тип архітектурної ерозії було виявлено в 12 із 17 усіх джерел СОЛ. 10 із 17 публікацій стверджують, що Неточна документація може бути причиною погіршення архітектури. Інші види архітектурного дрейфу зустрічаються менш ніж у половині розглянутих публікацій.

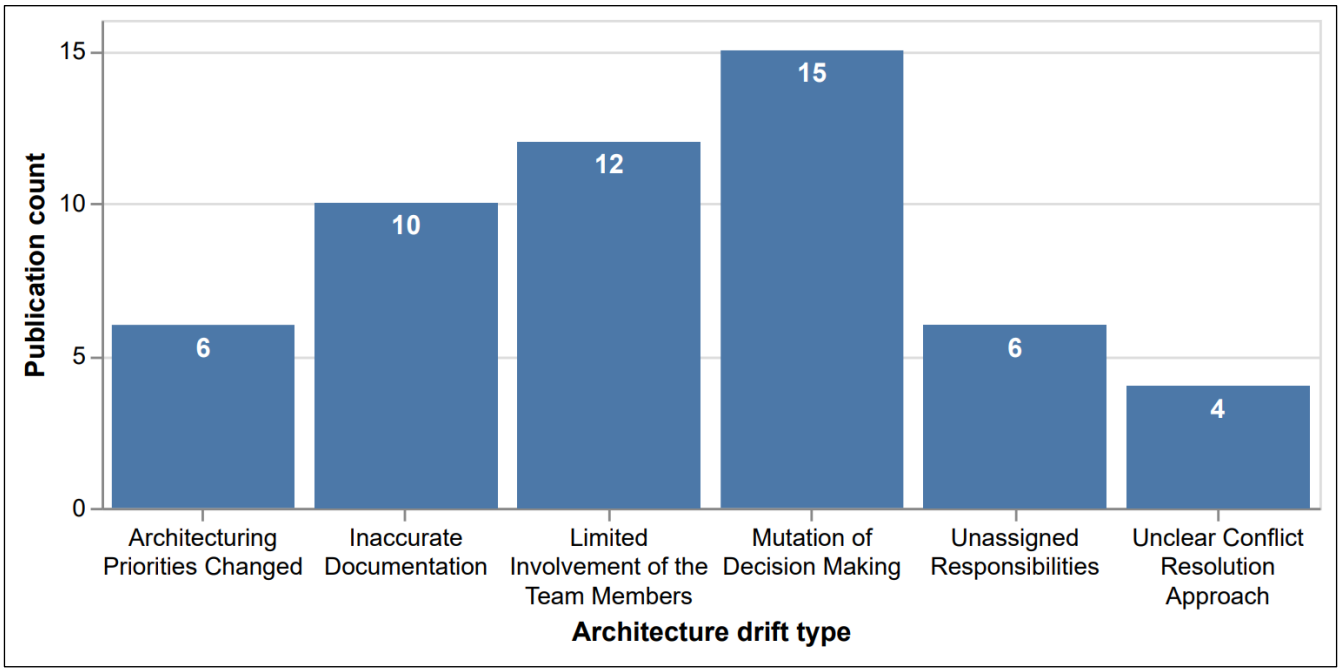

Рисунок 4.3 – Кількість публікацій СОЛ за типом дрифту архітектури.

Вважаємо, що інструмент, представлений у 3.2 Візія ідеального розв’язання (ДП2), може допомогти з розв'язанням вказаних найчастіших проблем. Надання простого та швидкого способу фіксування інформації дозволить зменшити отримати більш чітку та коротку документацію. Коротша документація дозволить приймати правильні рішення та всі члени команди зможуть краще залучитися до процесу проєктування архітектури.

З рисунку 3.3 видно, що 13 з 17 публікацій використовують СОЛ як основний чи один з основних методів дослідження, що робить дане дослідження третинним. Також, рисунок 3.3 показує, що найбільш частими загрозами для валідності зафіксованих проблем є зовнішня (англ. External, у 11 з 17 публікацій) та конструкційна (англ. Constsruct, 5 з 17). Така статистика може означати брак перевірки зібраної інформації на практиці. Додатково, рисунок 3.3 демонструє,

що лише 2 з 17 джерел СОЛ намагаються провести кількісну оцінку архітектурних проблем. Вважаємо, що це гарним показником, бо кількісна оцінка може займати багато часу та бути непрактичною для щоденного використання.

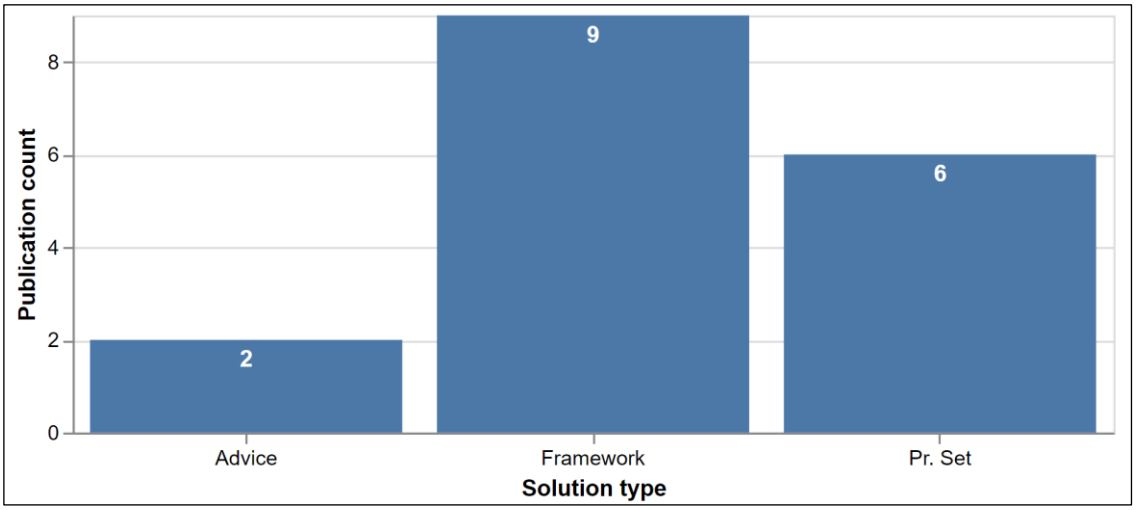

Рисунок 4.4 – Кількість публікацій СОЛ за типом представленого рішення.

Рисунок 4.4 показує, що 9 досліджень пропонують фрейфмворки, 6 пропонують набори практик, а 2 просто дають поради щодо вирішення виявлених архітектурних проблем. Це хороші результати, оскільки використання розширених рішень, таких як фреймворки, економить час. 10 із 17 публікацій представляють рішення, які не надихаються і не інтегруються підходами та філософією Agile (рисунок 4.5), 6 досліджень лише надихаються філософією Agile і лише [103] пропонують фреймворк, який водночас натхненний та інтегрований із підходами Agile.

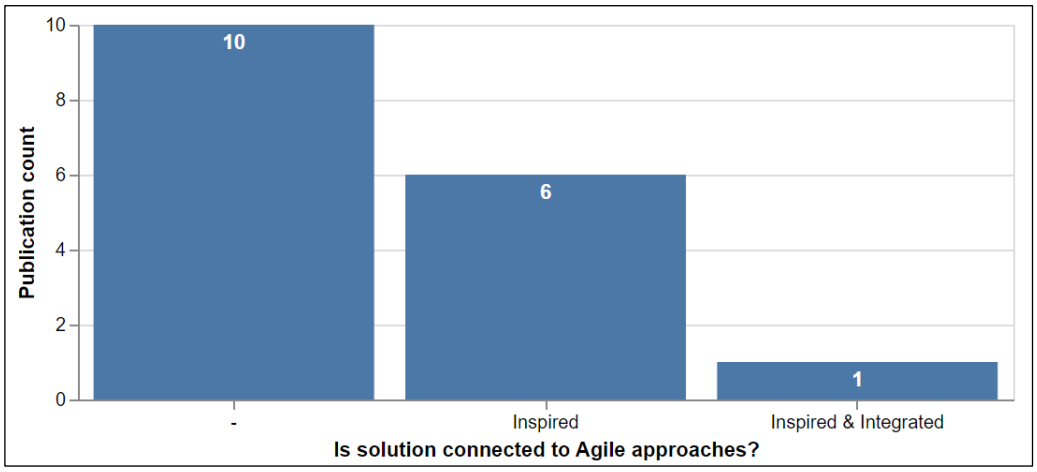

Рисунок 4.5 – Зв'язок між представленими рішеннями у публікаціях СОЛ та Agile.

Рисунок 4.5 показує, що лише 3 із 17 публікацій містять кількісну оцінку як частину представленого рішення. Також, бракує зворотнього зв'язку щодо вже розроблених рішень (рисунок 4.6). Було виявлено, що лише пропозиції від [19] містять зворотній зв'язок, тому було додано власні відгуки як відповіді на ці запитання. Загалом прийнятними були визнано [103, 4, 104, 30]. Вважаємо, що рішення з цих 4 робіт можна випробувати на практиці, у той час як інші рішення або є складними чи важкими для регулярного використання (6 із 17), або рішення є нечітким (6 з 17).

Важливим етапом будь-якого рішення є перевірка на практиці. Згідно з таблицями 4.3 та 4.4, було знайдено, що:

– 14 із 17 досліджень не показують, що їхні рішення були перевірені на практиці;
– найбільша загроза валідності є зовнішня (англ. External, 12 із 17 публікацій), загроза валідності конструкційна (англ. Construct) вказана в 5 публікацій, а 2 публікації не визначають загрози дійсності для своїх рішень;
– на рішення з більшості досліджень (15 із 17) не посилаються з інших публікацій;
– лише 2 дослідження показують, що їх рішення було популяризовано.

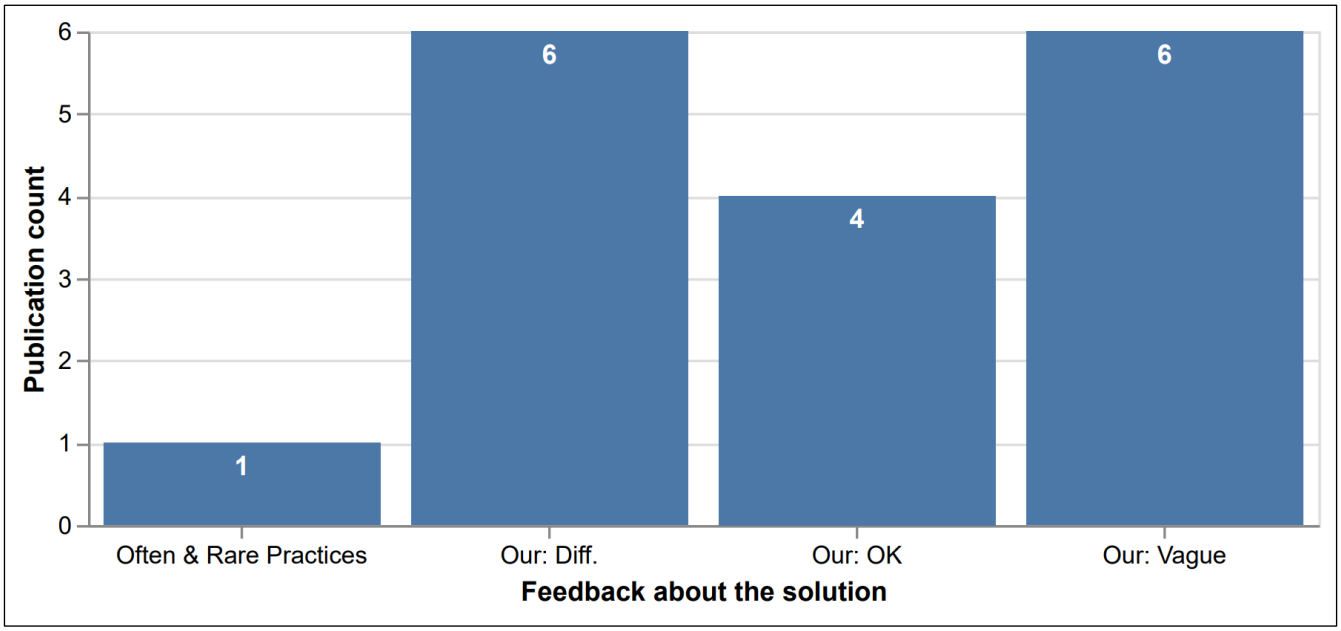

Рисунок 4.6 – Джерела СОЛ за відгуками про рішення..

Об’єднавши отримані дані, робиться висновок про серйозну відсутність перехресних посилань між публікаціями про однакові рішення (через відсутність зворотнього зв’язку, згідно рисунку 4.6, і посилань із зовнішніх джерел, згідно рисунку 4.4) і загальна відсутність обширної перевірки на практиці та популяризації запропонованих інструментів. Отже, спостерігається тенденція занедбання інструментів і рішень після ретельного збору інформації їхньої розробки рішення.

5 РЕАЛІЗАЦІЯ ПРОТОТИПУ ІНСТРУМЕНТУ

5.1 Визначення завдання та обсягу

Було створено невелику браузерну гру «Морський бій» на базі HTML. Вихідний код доступний на GitHub: https://github.com/shevchenkobn/battleship-html. Метою цього невеликого проєкту є проведення емпіричного експерименту з метою створення підтвердження концептуального рішення для:

- апробування парадигми розробки програмного забезпечення, заснованого на доказах (англ. evidence-based) [120, 121], використовуючи стислу, але продуману документацію;
- надання розумне обґрунтування прийнятих рішень [114, 103];
- запобігання ненавмисному технічному боргу [103, 104];
- явного зафіксування свідомого технічного боргу [103, 104].

На жаль, над грою працював лише 1 розробник, тому пряме спілкування між розробниками не було охоплено. Тим не менш, було докладено зусиль для виконання вище згаданої мети простим способом. Було застосовано ідеї, представлені у 3.2 Візія ідеального розв’язання (ДП2).

Завдання полягало в тому, щоб створити гру за правилами гри «Морський бій» на Вікіпедії (https://en.wikipedia.org/wiki/Battleship_(game)), але з можливими змінами правил в розумний (англ. reasonable) спосіб. Представлене визначення завдання посилається на завдання компанії РПЗ. Визначення завдання не вимагає:

- використання певної мови програмування та платформи (Windows, iOS, Android тощо);
- реалізації мережевої частини, отже, гра проводиться на одному пристрої по черзі між двома людьми, які фізично знаходяться в одній кімнаті;
- створення графічного інтерфейсу користувача [122], тому дозволено використання командного рядка.

Додаткові функції для впровадження включають наявність:

- можливість гри проти комп’ютера;
- табло з історією ігор;

– продумане автоматичне тестування через тест-кейси.

Завдання оцінюється за:

– коректно працюючу програму;
– розбірливий код, який відповідає найкращим практикам, і добре структурований проєкт. Це навіть цінніше, ніж реалізовані функції;
– розумну архітектуру;
– докладну інструкцію щодо запуску проєкту як для використання, так і для розробки;
– продуману й достатню документацію;
– використання сучасних і відповідних технологій і фреймворків. Це стосується використання допоміжних бібліотек та їх мінорних версій;
– ретельність тестування;
– гарний графічний інтерфейс користувача (GUI або UI) і хороший досвід користувача (UX) [122];
– цікаві додаткові функції.

Як видно, визначення завдання має необмежувальні, дозвільні, але заохочувальні вимоги. Крім того, у завданні високо оцінюються найкращі галузеві практики. Тому було вирішено використати це завдання як майданчик для експериментів.

5.2 Обґрунтування інструменту (ДП4)

У процесі реалізації було використано багато понять, знайдених під час дослідження СОЛ. Реалізовае завдання можна знайти у публічному доступі на GitHub. Детальний аналіз проєкту можна знайти в README.md, який зазвичай автоматично відображається на головній сторінці GitHub.

Після пояснення міркувань щодо обраної платформи та стека технологій буде пояснено цікаві частини документації. Зокрема, README.md, оскільки коментарі до коду, як правило, мають дуже локальний обсяг, здебільшого пояснюючи обґрунтування певних незначних технічних рішень.

Обрання платформи. Роботу над цим завданням було розпочато з аналізу варіантів використання гри. Важливо розуміти, хто і як використовуватиме гру, щоб зрозуміти, на яку платформу чи операційну систему орієнтуватися, оскільки це визначає мову, яка буде використовуватися [123, 124]. Вибір операційної системи та мови програмування має вирішальне значення, як підтверджено визначенням завдання, оскільки мова програмування визначає фреймворки, кроки та інструменти, які будуть використовуватися [124, 83].

За визначенням, гра повинна бути багатокористувацькою на одному пристрої. У сучасному світі смартфони є надзвичайно популярними [125], тому дозволити смартфонам запускати гру дасть додатку величезний потенціал охоплення користувацької бази. Смартфони мають різні операційні системи, як-от Android, iOS, Blackberry та Windows Phone [126]. Це означає, що для збільшення кількості потенційних користувачів потрібно охопити якомога більше мобільних операційних систем. Крім того, існує безліч інших немобільних платформ, таких як настільні комп’ютери з Linux, Windows і MacOS, ігрові консолі та навіть суперкомп’ютери та хмарні платформи [123]. Звичайно, можна було б розробити окремі додатки для кожної платформи, але це може бути не оптимальним рішенням, тому кросплатформна структура є кращою. Існує кілька різних кросплатформних фреймворків [124], серед яких найбільший досвід роботи з JavaScript, TypeScript і Flutter. Однак є додаток, який присутній майже на всіх платформах і операційних системах – веб-браузер [123, 124]. Крім того, сучасні браузери мають широкі можливості для розробки програм за допомогою JavaScript або WebAssembly [124, 127]. Отже, було вирішено використовувати підтримуваний браузером JavaScript як основну мову.

Наступним кроком буде визначення способу розгортання нашої гри на пристроях кінцевих користувачів. За визначенням, веб-браузери зазвичай запускають програми в системі клієнт-сервер [124]. Однак також можна мати повністю клієнтську програму без серверної частини [127, 124]. Очевидно, що останній підхід є кращим, оскільки він не вимагає запуску будь-якого типу сервера з виданням статичних файлів або CDN, тому немає потреби у підключені

до Інтернету на пристроях кінцевого користувача. Незважаючи на це, існують певні обмеження для програм, які працюють лише у браузері, зокрема, неможливість завантажувати ресурси (таблиці стилів, файли JavaScript, зображення тощо) з локальної файлової системи [128, 129]. Це обмеження було введено через велику кількість можливих уразливостей [129]. Цю проблему можна успішно обійти за допомогою URL-адрес даних [130]. Крім того, використання URL-адрес даних для завантаження зовнішніх ресурсів спрощує розгортання, оскільки всі ресурси вбудовано у файл HTML, тобто для розповсюдження потрібен лише один файл HTML. Деякі люди можуть стверджувати, що небезпечно передавати цілу програму чи гру одним файлом, але оскільки немає вимог щодо приховування чи інших способів захисту вихідного коду; і наразі немає жодних комерційних інтересів у додатку, вважається, що можна розповсюджувати додаток із легким доступом до вихідного коду. Отже, було вирішено використовувати єдиний HTML-файл із усіма ресурсами, вбудованими за допомогою URL-адрес даних для розгортання та доставки. Легкий доступ до вихідного коду може бути ускладнений через обфускацію [131].

Використані технології. Наступним кроком є вибір технологічного стеку, який буде використовуватися поверх браузерного JavaScript для розробки. Використання фреймворків і допоміжних бібліотек задовольняє численні нефункціональні вимоги, включаючи якість програмного забезпечення. Крім того, вибір технологічного стеку може вплинути на те, як оптимально документувати код і зберігати загальну документацію, а також керувати технічним боргом. Вибір фреймворку є важливим рішенням, оскільки він може додати додаткові обмеження [132, 133]. Логічно, що вибір популярного фреймворку може покращити потенційну зручність обслуговування в майбутньому, оскільки легше знайти розробника для популярної технології. Вивчивши статистику, було встановлено, що практично за всіма показниками React.js (або скорочено React) - найпопулярніший і перспективний фреймворк [132, 134, 133, 135]. Єдиним показником, за яким React займає друге місце замість першого, є задоволення згідно з опитуванням [135]. За збігом обставин мається значний досвід роботи з

цим фреймворком, тому його й було обрано як основний фреймворк. Також наявний досвід з Phaser [136], ігровим рушієм на базі HTML5, але його не було обрано як основний фреймворк, оскільки він набагато менш популярний, згідно із зірками GitHub, і проведена оцінка показала, що не знадобляться розширені функції Phaser, тому React буде найкращим інструментом.

Наскільки відомо, React не накладає жодних обмежень на спосіб документування проєкту. Отже, було вирішено використовувати формат GitHub Markdown, зокрема файл README.md у репозиторії та кореневому каталозі проєкту. Формат Markdown підтримує заголовки та підзаголовки (до 6 рівнів, подібно до HTML). GitHub використовує ці заголовки як фрагменти URL-адрес, що дозволяє створювати постійні Інтернет-посилання та навігацію файлом між різними заголовками, включно з використанням історії навігації браузера. Крім того, GitHub автоматично генерує зміст для файлу на основі заголовків. Markdown має інші можливості, такі як вкладені списки, цитати, виноски та багато інших, які чудово доповнюють можливості заголовків. Це дозволяє:

- мати великий файл Markdown і мати можливість швидко переміщатися по ньому;
- легко сприймати інформацію завдяки її структурованому відображенню [137];
- легко оновлювати файл завдяки простому синтаксису розмітки.

Отже, основним джерелом некодової документації буде файл README.md. Для коментарів до коду буде застосовано коментарі JavaScript і стиль коментарів JSDoc. У файлі README.md описано всі основні аспекти програми, включаючи нормалізацію моделей даних.

TypeScript – це строго типізована мова, яка транспільується в JavaScript. Використання системи типів TypeScript дозволяє писати більш надійний, але стислий код і навіть зменшити кількість юніт тестів [138]. Наявний значний досвід роботи з цією мовою, і, згідно зі знайденою інформацією, вона має найдосконалішу систему типів і це найкраща мова для транспіляції в JavaScript. Отже, буде використано TypeScript у проєкті. Чудовим прикладом системи типів

TypeScript є локалізація гри: enum як ключ для запису змушує компілятор перевіряти, чи перекладено кожне повідомлення на всі мови.

Використання TypeScript обмежує бібліотеки, які можна використовувати, оскільки бібліотеки вимагають визначення типів. Тим не менш, не було виявлено, що деякі з найкращих допоміжних бібліотек- пакетів NPM є доступними для використання. Основні бібліотеки показано в розділі Технології у файлі README.md.

Файл README. Файл README.md містить більше інформації щодо наведених нижче тем, було виділено лише частини, цікавий темі нинішнього дослідження.

Більшість тестування проводилося вручну через часові обмеження та систему типів TypeScript [138]. Однак автотести проєкту охоплюють алгоритмічні частини, які з більшою ймовірністю містять неочевидні помилки в логіці. Було додано юніт-тести як для логіки фреймворку, так і для компонентів React. Крім того, тести разом із лінтерами стилю коду запускаються автоматично перед створенням коміту Git за допомогою бібліотеки Husky.

Було змінено файли конфігурації проєкту WebPack React за замовчуванням, щоб створити єдиний файл HTML із усіма додатковими ресурсами, вбудованими в нього як URL-адреси даних [130]. Щоб дані зміни можна було відслідковувати та зробити можливим перехід до нових основних версій бібліотек у майбутньому, усі зміни внесено з коментарями з префіксом `inlineBuild:`. Ці коментарі пояснюють, як можна відновити початкову поведінку системи збірки проєкту.

Було додано паролі на додачу до імен гравців. Ці паролі використовуються для зміни конфігурації перед грою та для перегляду власного ігрового поля під час гри. Таке доповнення захищає гравців від підглядання супротивника. Паролі є необов’язковими, а пароль за замовчуванням – це порожній рядок, що означає, що кожен гравець може вибирати, чи хоче він мати пароль.

Було приділено значну кількість часу тому, щоб переконатися, що гра оптимізована для екранів різних розмірів і різної щільності пікселів. Більшу частину адаптації інтерфейсу зробила бібліотека Material UI, але також було

змінено розмір шрифту та ігрового поля в пікселях. Значною частиною впровадження адаптивності інтерфейсу була оптимізація процедур розміщення кораблів під час налаштування гри. Зокрема, виникла дуже рідкісна помилка нескінченного циклу візуалізації у функціональних компонентах React.

Щоб поекспериментувати з розширеними функціями архітектури, проєкт було надпроєктувано (англ. overengineering). Проте процес надпроєктуваннння був контрольованим, а не хаотичним. Було застосовано певні проєктні рішення, які зазвичай присутні у великих проєктах. Це було зроблено, щоб побачити та продемонструвати, як подібні рішення можна використовувати у великих реальних проєктах. Завдяки контрольованому надпроєктуванню проєкт також стає більш підготовленим до можливого ряду майбутніх функцій і потенційних покращень, таких як особливі та непрямокутні ігрові дошки, особливі кораблі та оновлення правил.

У файлі README.md є заголовок під назвою Контроль технічного боргу, який пояснює, що технічними боргами керують за допомогою файлу README.md і коментарів до коду. Надається перевага використанню коментарів коду для збору технічної заборгованості, оскільки це дозволяє негайно встановити зв’язок між проблемою та місцем коду. Однак, якщо важко знайти точне кодове місце для певної проблеми, вона поміщається в розділ TODO файлу README.md, де також можна надати деталі щодо потенційного рішення або список рішень, які не спрацювали. Розділ TODO також визначає правила використання коментарів коду та журналів. Зокрема, було визначено:

- FIXME префікс для місць, для яких існує краще рішення;
- TODO префікс для місць, які можуть мати або не мати рішень;
- dbg для тимчасових логів для налагодження. Це потрібно для випадків, коли журнали були випадково відправлені в мережевий репозиторій і інші розробники впевнені, що їх можна видаляти.

Програма не має власного версіювання, на відміну від більшості випущених продуктів має. Додати версіювання легко, оскільки це просто рядкове значення,

показане, як версія. Додатково, у розділі Контроль технічного боргу в README.md викладено, як зробити можливу подальша інтеграція з DevOps:

Крім того, за потреби процес збірки можна розширити додатковими аргументами командного рядка, щоб дозволити автоматичне та безпечне збільшення версії (мажорної, мінорної або версії патчу). Пізніше до Husky можна додати посткоммітний хук Git, щоб автоматично додати тег Git до певної версії програми.

У файлі README.md також зберігається список можливих покращень. Відомо, що дошки Kanban [77] надають набагато більше можливостей. Але, враховуючи преференцію простим і повторюваним рішенням на додаток до лише 1 активного розробника в проєкті, вважається, що для поточного розміру проєкту більш ніж достатньо мати простий список покращень. Найбільша перевага беклогу у формі списку полягає в тому, що файл README.md є єдиним джерелом істини: беклог розташовано безпосередньо біля коду, який його реалізує. «В процесі» (англ. In Progress) можна позначити за допомогою форматування Markdown, наприклад, курсивного шрифту. Видалення елемента зі списку беклогу означає, що його було відхилено або виконано. За потреби можна застосувати багатосписковий підхід, де має бути 3 списки: для невиконаних, для відхилених і для завершених функцій.

Опис прототипу рішення. Запропонований вище підхід до документування не є закінченим чи повноцінним інструментом. Тим не менш, він служить своїй меті і представляє цікаву підхід до цього питання. Він має ряд переваг, наприклад:

- стислість і змістовність;
- відсутність потреби в додатковому програмному забезпеченні третіх сторін;
- високий рівень інтеграції з кодовою базою;
- імерсивність, тобто програмістам не потрібно перемикати контекст написання коду, щоб зробити певні важливі нотатки;
- помірна структурованість і можливість пошуку. Файл README.md добре читається як текстовий файл, але його також можна переглянути у

відрендереному, візуалізованому поданні Markdown у багатьох програмах (наприклад, безкоштовний Visual Studio Code). Редактори коду показують коментарі коду різними кольорами або стилями шрифту. Редактори коду також дозволяють здійснювати пошук у проєкті, що полегшує пошук усіх коментарів TODO, FIXME і dbg.

Підводячи підсумок, видно низку переваг у прототипі рішення, описаному вище. Він дещо відрізняється від поданого бачення та ІнОФ, але вважається, що важливіше мати просте та ефективне рішення, ніж слідувати певному набору правил і вказівок. ІнОФ більше підходить для великих проєктів із кількома інструментами розробки та великим рівнем співпраці. Було вирішено, що цей конкретний проєкт має інструмент, який відрізняється від нашого початкового бачення. Крім того, це показує, що не існує жодного рішення, і кожен проєкт потребує власного рішення для документування та фіксації прийнятих рішень і технічної заборгованості. Вважається, що набагато важливіше мати правильну ментальну парадигму, щоб мати можливість швидко обрати інструмент для конкретного проєкту. Тим не менш, вважаємо, що ідеї та структура файлу README.md готові для копіювання в інших проєктах.

6 ОЦІНКА РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

Дане дослідження складається з 2 частин:

- систематичний огляд літератури виконананий з використанням Google Scholar. СОЛ може містити серйозні ризики через те, що він був виконаний 1 особою;
- інструмент і підхід для документування та фіксації технічної заборгованості та рішень, створений швидким, дешевим і високоінтегрованим способом. Даний підхід був перевірений на сурогатному проєкті, перевірка на реальних промислових проєктах відсутня.

Під час СОЛ було визначено, що існує понад 40 інструментів, але кожен інструмент не був детально досліджений через обмеження часу. На жаль, не було знайдено достатньо інформації щодо перевірки інструментів на практиці. Відповідно до представленого бачення як ідеального інструменту, так і процесу розробки інструменту, наступним кроком у розробці нового інструменту є перевірка його можливостей у промисловому проєкті. Загалом робиться висновок, що дослідження було успішним.

Метою дослідження було охопити значну сферу комунікації під час проєктування архітектури. У результаті було виявлено кілька областей, таких як соціальний борг [13, 26], прийняття рішень [19, 23, 104, 114], управління архітектурними знаннями [104, 10], розробка програмного забезпечення на основі доказів (англ. evidence-based software development) [120, 121], управління технічним боргом [103, 104], документація та пряме спілкування [4], програмне забезпечення з відкритим кодом [19, 33, 16], ГРПЗ [22, 15, 113], «запахи спільноти» (англ. community smells) [33], архітектурний дизайн [33, 19, 16, 104], Agile [113]. Тому вважається, що результати СОЛ можуть бути використані для виділення напрямків майбутніх досліджень, можливо, шляхом виконання додаткових СОЛ з охопленням меншої області.

Інструменти, які були знайдені, стосуються різних аспектів спілкування в розробці програмного забезпечення. Хоча більшість інструментів не готові для

повсякденного використання в галузі через різні причини (рисунки 3.3 – 3.4), вважається, що представники галузі можуть використати дане дослідження, щоб знайти інструмент для вирішення їхніх проблем. Крім того, були зібрані дані про перевірку інструментів на практиці або їх загальне впровадження. Ця інформація може бути використана як в академічній сфері, так і в індустрії, оскільки вона показує, що «... результати [конкретного дослідження] можна застосувати на практиці» [29], слайд 59.

6.1 Відповіді на дослідницькі запитання

Задля відповіді на ДП1 «Які найпоширеніші типи дрейфу архітектури через неправильне спілкування та які найпоширеніші наслідки таких помилок?», було визначено 3 найпоширеніші типи дрейфу архітектури з їхніми наслідками (згідно з 17 публікаціями СОЛ):

- мутація прийняття рішень – 88,2% (15/17); нові рішення не враховують попереднього бачення архітектури або некоректно реалізовані розробниками;
- обмежена участь членів команди – 70,6% (12/17); рішення приймаються ізольовано від інших частин проєкту або інших членів команди; рішення можуть не враховувати певні фактори, відомі лише деяким розробникам проєкту;
- неточна документація – 58,8% (10/17); документація або надає занадто багато інформації та розмиває важливі поняття, або не містить необхідної інформації.

Загальним наслідком усіх типів дрейфу архітектури є збільшення у проєкту технічного, архітектурного чи соціального боргу, яке:

- зменшує зручність підтримки й обслуговування та інші аспекти якості програмного забезпечення;
- перешкоджає введенню нових функцій;
- перешкоджає введенню нових розробників у проєкт.

Повний список типів можна знайти в першому питанні класифікаційного протоколу СОЛ, де визначаються 6 типів дрейфу архітектури (див. 3.3.4 Класифікаційний протокол). Ці 6 типів були визначені під час пілотного огляду літератури перед створенням протоколу СОЛ. Категорії були створені шляхом уніфікації типів дрейфу, знайдених у оглянутих документах підчас пілотного огляду. Пізніше цей протокол використовувався для класифікації 17 досліджень. Крім того, було зібрано інформацію про наявність кількісної оцінки проблем і загроз валідності для знайдених типів дрейфу.

Для відповіді на ДП2 «Які існують запропоновані інструменти для пом'якшення дрейфу архітектури та проблем зв'язку?» не було знайдено жодного інструменту загального призначення, який був би помірно популярним і широко використовуваним у галузі. Але було знайдено понад 40 різних інструментів, які стосуються різних аспектів дрейфу архітектури та питань комунікації. Крім того, було представлено бачення ідеального інструменту для цієї мети, засноване на знайдених типах дрейфу архітектури. На додачу, була реалізована гра «Морський бій», щоб спробувати використати вже існуючі інструменти загального призначення (README.md, коментарі до коду, IDE, Git), щоб вирішити проблеми, подані вище.

Інформацію про існуючі інструменти було зібрано з досліджень під час СОЛ згідно протоколу СОЛ. Зокрема, було зібрано інформацію про комплексність (тип) рішення, зв’язок із Agile, перевірку в галузі, кількісну оцінку та загрози валідності. Додатково подано статистику за знайденими публікаціями.

Було знайдено відповідь на ДП3 «Які були причини не прийняти існуючі інструменти? Чи є наявні інструменти занадто розпливчастими чи складними?». Перш за все, у знайдених дослідження було виявлено брак уваги до процесу просування інструменту. Припускається, що це може бути пов’язано з особливою стратегією розв’язання проблем в науковому середовищі, де всебічність дослідницького методу має першорядне значення. Отже, підчас окреслення методу було визначено алгоритм популяризації інструментів. Детальний опис можна знайти в розділі 3.3.3 Критерії якості.

Відсутність розповсюдження інформації щодо розробленого інструменту та зосередження на методах дослідження призвело до слабких перехресних посилань між різними дослідженнями, однак не можна стверджувати, що було знайдено всі документи щодо існуючих інструментів. Таким чином, замість зворотного зв'язку від працівників індустрії чи інших дослідників було надано авторські відгуки щодо рішень у більшості досліджень (рисунок 4.6). Було надано відгуки до 16 із 17 документів. Окрім [19], де рішення вже було застосовано на практиці у розробці програмного забезпечення з відкритим вихідним кодом із значною кількістю зворотнього зв'язку та було просто взято для аналізу, було зроблено висновок, що лише 4 із 16 документів містять рішення, які загалом готові для впровадження в індустрії.

Було проаналізовано ДП4 «Які дії можна зробити та які заходи можна вжити, щоб пом'якшити дрейф архітектури та проблеми з комунікацією на практиці?». Тому у даному досліджені акцент був на пошуці простого і ненав'язливого інструменту. Однак, не було знайдено готового до використання інструменту, який би відповідав авторській візії (див. 3.2 Візія ідеального розв'язання (ДП2)) . Оскільки не було знайдено жодних загальноприйнятих інструментів, було вирішено використати наш власний підхід для фіксації технічного боргу та прийнятих рішень. Як проєкт для експерименту було використано тестове завдання з розробки гри «Морський бій».

Відповідно до поставленого завдання було розроблено гру «Морський бій». Архітектуру було дещо надпроєктовано, щоб імітувати великий проєкт. Для документації було використано README.md у корені проєкту та коментарі до коду. Поданий підхід базується на:

- простій мові розмітки Markdown і її структурованості;
- можливостях сучасних веб-сайтів і IDE відображати відрендерене Markdown і візуально виділяти коментарі коду;
- можливостях сучасних IDE для пошуку інформації в проєкті.

Було зроблено висновок, що такий підхід є можливим та ефективним. Тому вважаємо, що його можна використовувати його в інших проєктах, але

наголошуємо на важливості уважного підходу до вибору та впровадження інструментів.

6.2 Подальші дослідження

Точкою фокусу дослідження був пошук перевірених у галузі інструментів для покращення комунікації та архітектури в розробці програмного забезпечення. На жаль, не було знайдено перевірених на практиці та популярних інструментів, але було проведено систематичний огляд літератури великої області процесу розробки програмного забезпечення. Отже, це дослідження стало дороговказом для подальших досліджень, що робить майбутню роботу одним із найважливіших результатів поточного дослідження.

Під час поточного дослідження було розглянуто кілька досліджень, які охоплювали кілька країн [15, 22, 11]. Незважаючи на це, не було знайдено згадок про Україну. Дослідження від [11] охоплювало країни з різних куточків земної кулі, але не Україну. Досліджень з України також не було знайдено. Це дивна ситуація, враховуючи масштаб розвитку IT в Україні [48, 50].

Загалом було виділено 2 великі напрямки майбутньої роботи:

- продовження пошуку існуючого інструменту;
- розробка власного інструменту.

Наприклад, Таблиця 1 з роботи [10] показує як список інструментів для аналізу, так і список функцій, необхідних для потенційного інструменту, тому інформацію можна використовувати для обох майбутніх напрямків роботи.

Першим кроком майбутніх досліджень є перевірка понад 40 уже знайдених інструментів перед тим, як перейти до додаткового пошуку та огляду літератури. У той же час під час даного дослідження було помічено, що академічне та індустрійне середовища мають різні стратегії щодо розробки рішень. Академічна стратегія – це суворість і точність методології, тому їх дослідження, як правило, є дедуктивними: збираються та аналізуються докази з різних джерел, після чого робиться висновок. Галузева стратегія є індуктивною: рішення базуються на одиничних доказах певної проблеми, тому проблема вирішується, а потім рішення

узагальнюється. Вважаємо, що академічний підхід вимагає більше часу та енергії для методології, тому менше уваги залишається на дизайні самого рішення. З іншого боку, галузеві рішення можуть бути не такими узагальненими та перевіреними, як академічні.

У поточному дослідженні основним фокусом були академічні дослідження, але також було включено 2 галузеві статті для СОЛ і використано кілька галузевих статей як посилання. Вважається, що відносна простота галузевого підходу приваблива, і він також здається доречним для поданого бачення інструменту. Таким чином, припускається, що в майбутньому буде застосовано СОЛ для публікацій у блогах та інших галузевих статей на цю тему, щоб виявити більше потенційно цікавих рішень. Визнається, що галузеві папери можуть не мати певних структурних особливостей, які перешкоджатимуть використанню протоколу СОЛ, але не очікуємо значного зниження якості.

Подальший пошук академічних джерел можна покращити за рахунок:

- звуження сфери дослідження до прийняття рішень. Наприклад, ряд документів про прийняття рішень і технічний борг можна знайти в SLR від авторів [23];
- застосування вдосконалених методів для вилучення даних за допомогою програмного забезпечення NVivo [103] і коефіцієнту надійності Cohen's Kappa [16]; синтез даних шляхом застосування тематичного синтезу [103];
- використання Матриці фокусу дослідження прийняття рішень та Циклу дослідження прийняття рішень для більш детального дослідження процесу прийняття рішень [23];
- додавання ключових слів «Поведінка» (наприклад, «Фактор поведінки»), щоб краще розуміти людський фактор в процесі розробки програмного забезпечення та взаємодії між людьми;
- дрозодавання ключових слів, пов’язаних із Agile, для більш детального аналізу проєктів, які використовують найпопулярніші методології. Дослідження [103] можна використовувати для натхнення.

Крім того, було виявлено, що найефективнішим методом пошуку нових джерел для дослідження є метод вперед направленого сніжного кома – він має 9% прийнятих джерел від усіх розглянутих. На додачу, метод вперед направленого сніжного кома дає змогу знайти новіші публікації, ніж дослідження, для якого застосовується метод сніжного кому.

Крім тематичних наукових робіт, огляд може бути розширений на:

- стандарти ISO, такі як ISO 42010:2011, ISO 25010:2011, щоб знайти певні ідеї для опрацювання. Варто зазначити, що доступ до стандартів не є безкоштовним;
- аналіз існуючих книг про гнучку або тонку (англ. lean) архітектуру програмного забезпечення, наприклад [139, 96]. Наприклад, корпорація Meta визначила перехід до більш ощадливих процесів у майбутньому як пріоритет, навіть якщо це означатиме скорочення компанії [140]. Варто зазначити, що попередній пошук джерел, що стосуються тонкої архітектури дає менше результатів, ніж пошук джерел щодо Agile архітектури;
- аналіз Роздуття програмного забезпечення [39] для можливих доказів щодо проблем під час прийняття рішень.

Дослідження [141] пояснює використання швидких оглядів (англ. Rapid Reviews) як альтернативи систематичним оглядам літератури. Їх можна використовувати для збору даних менш формальним і суворим способом. Rapid Reviews також можна використовувати в розробці програмного забезпечення на основі доказів, щоб усунути розрив між доказами щодо певних практичних питань і наявними дослідженнями.

Щодо розробки авторського інструменту, вважається, що ідеї, подані в 3.2 Візія ідеального розв’язання (ДП2) і 5.2 Обґрунтування інструменту (ДП4), слід розвинути для створення повноцінного інструменту. Основними цінностями такого інструменту мають бути простота, співпраця та аналіз даних із хорошим користувацьким досвідом. Крім того, слід узяти до уваги проблеми, згадані в GAP і Concern frameworks від [15, 22]. Важливою частиною створення інструменту є

перевірка інструменту в галузевих проєктах і просування на різних галузевих форумах так само, як Agile Alliance просував свій Маніфест [142]. Прикладом такого підходу з академічного середовища є Даміан Тамбуррі, який розробляв інструменти для управління соціальними та архітектурними боргами [26, 27], запахів спільноти та архітектури [28, 33] на основі власних досліджень у цій галузі. Однак було зроблено висновок, що він недостатньо розповсюджував свої відкриття, як це запропоновано в [119, 142].

ВИСНОВКИ

Було проведено систематичний огляд літератури та виявлено, що в багатьох сферах дослідження є незавершеним, але оскільки було охоплено велику предметну область, це не є проблемою для висновків поточного дослідження. У той же час було виявлено, що існуючі інструменти не були перевірені в промислових проєктах. Мета дослідження полягала в пошуку інструменту для вирішення проблем спілкування та прийняття рішень. Прийняття рішень і усвідомлення процесу прийняття рішень можуть покращити архітектуру. Зроблено висновок, що у даній галузі існує значний обсяг матеріалу, тому є можливість, що деякі публікації можуть бути випадково не включені. Тому поточне дослідження можна вважати дороговказом для подальших досліджень.

Проте, будь-яке рішення інструмент має бути простим у використанні, інакше його ніколи не приймуть архітектори. Крім того, він повинен бути описовим, а не директивним. Отже, головною метою такого інструмента є надання порад архітекторам, а не прийняття рішень замість них.

У розглянутій літературі було знайдено значну кількість різних інструментів. Знайдені фреймворки, як правило, складні, тоді як набори практик, як правило, нечіткі. З іншого боку, ІнОФ або авторський апробований інструмент (гра «Морський бій») віддають перевагу простоті та вузькому фокусу для точності. Тому було представлено неприємну гіпотезу: проблему, яку треба вирішити, майже неможливо вирішити простим і зрозумілим способом. Однак, додаткові дослідження потрібні для перевірки даної гіпотези.

Додатково було порівняно підходи до розробки рішень в академічному та індустрійному середовищі. Крім того, виконаний СОЛ навмисно включив неакадемічні статті. Рішення з обох сторін мають свої переваги й недоліки, переважно через більше дедуктивний підхід в академічному середовищі та індуктивний в індустрії.

Визнається, не було застосовано деяких вказівок щодо СОЛ, як-от використання надійних критеріїв якості, використання лише рецензованих досліджень або використання більш надійної та чіткої схеми кодування

(прикладом може бути СОЛ проведене). В основному було зроблено поступки в методології СОЛ, щоб включити ширший спектр документів і краще представити дороговказ області.

Можна зробити висновок, що пошук існуючих інструментів є виснажливим і тривалим процесом, який, можливо, може бути однією з причин створення такої кількості інструментів: легше розробити власне рішення а не знайти підходящу вже існуючу. Крім того, інструменти часто мають дуже мало спільного.

Були показані покоління інструментів для управління знаннями про архітектуру, багато з яких призначені для прийняття рішень. Схоже, що поширення використання архітектурних знань за межами поточних досліджень все ще знаходиться в етапі розвитку. Одне з класифікаційних запитань протоколі СОЛ даного дослідження стосувалося кількісно визначених проблем. Також цікавим поняттям, знайденим підчас дослідження, є групове програмне забезпечення та електронний робочий простір. Сьогодні ця тема набагато цікавіша в контексті дистанційної зайнятості та Індустрії 4.0 у світі після COVID-19.

У прототипі інструменту, який було представлено як гру «Морський бій», було дано відповідь на ДП4, реалізовано максимально спрощену схему нотування важливих даних та документування, без покладання на зовнішні інструменти, окрім тих, якими вже користуються розробники. Ця спроба вважається успішною, але наголошується, що цей прототип не є універсальним.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

[1] B. Kitchenham, O. P. Brereton, D. Budgen, M. Turner, J. Bailey та S. Linkman, «Systematic literature reviews in software engineering–a systematic literature review,» *Information and software technology,* т. 51, p. 7–15, 2009.

[2] O. Bezsmertnyi, N. Golian, V. Golian та I. Afanasieva, «Behavior Driven Development Approach in the Modern Quality Control Process,» в *2020 IEEE International Conference on Problems of Infocommunications. Science and Technology (PIC S&T)*, 2020.

[3] I. Gruzdo, I. Kyrychenko, G. Tereshchenko та N. Shanidze, «Metrics Applicable for Evaluating Software at the Design Stage.,» в *COLINS*, 2021.

[4] M. Hummel, C. Rosenkranz та R. Holten, «The role of social agile practices for direct and indirect communication in information systems development teams,» *Communications of the Association for Information Systems,* т. 36, p. 15, 2015.

[5] V. Kyriy, I. Sheiko та R. Petrova, «Optimization of management information support as a basis for organizational transformations at an enterprise,» *Periodicals of Engineering and Natural Sciences,* т. 7, p. 679–689, 2019.

[6] P. Naur та B. Randell, «Software engineering: Report of a conference sponsored by the nato science committee, garmisch, germany, 7th-11th october 1968,» 1969.

[7] Z. Baida, «Stakeholders and their concerns in software architecture,» *Technology Journal of Vrije Universiteit, Amsterdam,* p. 68–82, 2001.

[8] I. Fatema та K. Sakib, «Factors influencing productivity of agile software development teamwork: A qualitative system dynamics approach,» в *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, 2017.

[9] S. O. Alexander Hars, «Working for free? Motivations for participating in open-source projects,» *International journal of electronic commerce,* т. 6, p. 25–39, 2002.

[10] R. Capilla, A. Jansen, A. Tang, P. Avgeriou та M. A. Babar, «10 years of software architecture knowledge management: Practice and future,» *Journal of*

Systems and Software, т. 116, p. 191–205, 2016.

[11] M. Marinho, J. Noll, I. Richardson та S. Beecham, «Plan-driven approaches are alive and kicking in agile global software development,» в *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2019.

[12] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland та D. Thomas, «Manifesto for Agile Software Development,» [Онлайновий]. Available: https://agilemanifesto.org/. [Дата звернення: 25 August 2022].

[13] T. Dreesen, P. Hennel, C. Rosenkranz та T. Kude, «“The Second Vice is Lying, the First is Running into Debt.” Antecedents and Mitigating Practices of Social Debt: An Exploratory Study in Distributed Software Development Teams,» в *Proceedings of the 54th Hawaii International Conference on System Sciences*, 2021.

[14] V. Gruhn та R. Striemer, The Essence of Software Engineering, Springer Nature, 2018.

[15] O. Sievi-Korte, S. Beecham та I. Richardson, «Challenges and recommended practices for software architecting in global software development,» *Information and software technology,* т. 106, p. 234–253, 2019.

[16] T. Bi, W. Ding, P. Liang та A. Tang, «Architecture information communication in two OSS projects: The why, who, when, and what,» *Journal of Systems and Software,* т. 181, p. 111035, 2021.

[17] P. Bourque та R. E. Fairley, «SWEBOK v3. 0: Guide to the software engineering body of knowledge,» *IEEE Computer Society*, p. 1–335, 2014.

[18] M. Jaiswal, «Software Architecture and Software Design,» *International Research Journal of Engineering and Technology (IRJET) e-ISSN,* p. 2395–0056, 2019.

[19] A. Baabad, H. B. Zulzalil, S. B. Baharom та others, «Software architecture degradation in open source software: A systematic literature review,» *IEEE Access*, т. 8, p. 173681–173709, 2020.

[20] C. Gacek, A. Abd-Allah, B. Clark та B. Boehm, «On the definition of software system architecture,» в *Proceedings of the First International Workshop on Architectures for Software Systems*, 1995.

[21] R. Fitzpatrick, «Software quality: definitions and strategic issues,» *Staffordshire University, School of Computing Report*, 1996.

[22] O. Sievi-Korte, I. Richardson та S. Beecham, «Software architecture design in global software development: An empirical study,» *Journal of Systems and Software*, т. 158, p. 110400, 2019.

[23] M. Razavian, B. Paech та A. Tang, «Empirical research for software architecture decision making: An analysis,» *Journal of Systems and Software,* т. 149, p. 360–381, 2019.

[24] K. Matsudaira, «Bad software architecture is a people problem,» *Communications of the ACM,* т. 59, p. 42–43, 2016.

[25] G. Orosz, «Software Architecture is Overrated, Clear and Simple Design is Underrated,» *The Pragmatic Engineer*, 2019.

[26] D. A. Tamburri, «Software architecture social debt: Managing the incommunicability factor,» *IEEE Transactions on Computational Social Systems*, т. 6, p. 20–37, 2019.

[27] D. A. Tamburri, P. Kruchten, P. Lago та H. van Vliet, «What is social debt in software engineering?,» в *2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, 2013.

[28] D. Tamburri, R. Kazman та W.-J. Van den Heuvel, «Splicing community and software architecture smells in agile teams: an industrial study,» 2019.

[29] A. Ferrari, *Systematic literature reviews and systematic mapping studies.*

[30] B. Tekinerdogan, S. Cetin, M. A. Babar, P. Lago та J. Mäkiö, «Architecting in

global software engineering,» *ACM SIGSOFT Software Engineering Notes*, т. 37, p. 1–7, 2012.

[31] A. Stellman та J. Greene, Learning agile: Understanding scrum, XP, lean, and kanban, " O'Reilly Media, Inc.", 2014.

[32] B. Dobing та J. Parsons, «How UML is used,» *Communications of the ACM*, т. 49, p. 109–113, 2006.

[33] D. A. Tamburri, F. Palomba та R. Kazman, «Exploring community smells in open-source: An automated approach,» *IEEE Transactions on software Engineering*, т. 47, p. 630–652, 2019.

[34] M. R. Swaine та P. A. Freiberger, «Analytical Engine,» в *Encyclopedia Britannica*, 2022.

[35] J. von Neumann, *Introduction to "the first draft report on the edvac". Archive. org*, 1945.

[36] A. Tucker, A model curriculum for k–12 computer science: Final report of the acm k–12 task force curriculum committee, ACM, 2003.

[37] I. E. E. E. Standard, «610.12.-1990,» *IEEE Standard Glossary of Software Engineering Terminology*, 1990.

[38] P. Gelsinger, «Moore's Law–The Genius Lives On,» *IEEE Solid-State Circuits Society Newsletter*, т. 11, p. 18–20, 2006.

[39] N. Wirth, «A plea for lean software,» *Computer*, т. 28, p. 64–68, 1995.

[40] A. Quach та A. Prakash, «Bloat factors and binary specialization,» в *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation*, 2019.

[41] The Editors of Encyclopedia Britannica, «software,» в *Encyclopedia Britannica*, 2022.

[42] *How many developers are there in the world in 2021?: Blog future processing*, 2022.

[43] A. Vita та Relay, *How many software developers are there in the world?*, 2021.

[44] FocusEconomics, *The world's largest economies (1994-2026)*, 2021.

[45] *GDP (current US$) - united states.*

[46] Zippia, *Software engineer trends.*

[47] Zippia, *Software developer trends.*

[48] Ukraine IT Association, *Ukraine it report 2021*.

[49] D. Leon, D. A. Jdanov, C. J. Gerry, P. Grigoriev, M. McKee, O. Penina, F. Meslé, J. L. Twigg, J. Vallin, D. H. Vagero та others, «The Russian invasion of Ukraine and its public health consequences,» *The Lancet Regional Health-Europe*, p. 1–2, 2022.

[50] V. Agency, *Асоціація "it Ukraine" - the IT export industry continues to support Ukrainian economy*, 2022.

[51] \. В. \. Б. Васюта, \. В. \. В. Васюта та \. А. \. М. Путря, «IT-бізнес в умовах війни в Україні,» 2022.

[52] H. Lasi, P. Fettke, H.-G. Kemper, T. Feld та M. Hoffmann, «Industry 4.0,» *Business & information systems engineering*, т. 6, p. 239–242, 2014.

[53] M. Javaid, A. Haleem, R. Vaishya, S. Bahl, R. Suman та A. Vaish, «Industry 4.0 technologies and their applications in fighting COVID-19 pandemic,» *Diabetes & Metabolic Syndrome: Clinical Research & Reviews*, т. 14, p. 419–422, 2020.

[54] G. Czifra, Z. Molnár та others, «Covid-19 and Industry 4.0,» *Research papers faculty of materials science and technology slovak university of technology*, т. 28, p. 36–45, 2020.

[55] J. Wosik, M. Fudim, B. Cameron, Z. F. Gellad, A. Cho, D. Phinney, S. Curtis, M. Roman, E. G. Poon, J. Ferranti та others, «Telehealth transformation: COVID-19 and the rise of virtual care,» *Journal of the American Medical Informatics Association*, т. 27, p. 957–962, 2020.

[56] J. Best, «Wearable technology: covid-19 and the rise of remote clinical monitoring,» *bmj*, т. 372, 2021.

[57] H. O. Brien, «What does the rise of digital religion during Covid-19 tell us about

religion’s capacity to adapt?,» *Irish Journal of Sociology,* т. 28, p. 242–246, 2020.

[58] E. Richardson, D. Aissat, G. A. Williams, N. Fahy та others, «Keeping what works: remote consultations during the COVID-19 pandemic,» *Eurohealth,* т. 26, p. 73–76, 2020.

[59] S. Joshi, «Rising importance of remote learning in India in the wake of COVID-19: issues, challenges and way forward,» *World Journal of Science, Technology and Sustainable Development,* 2021.

[60] S. Hylton, L. Ice та E. Krutsch, «What the long-term impacts of the COVID-19 pandemic could mean for the future of IT jobs,» *Computer,* т. 4, p. 667–6, 2022.

[61] *ISO 25000 Portal.*

[62] A. Boulanger, «Open-source versus proprietary software: Is one more reliable and secure than the other?,» *IBM Systems Journal,* т. 44, p. 239–248, 2005.

[63] L. Troan, «Open Source from a Proprietary Perspective,» *URL: https://web. archive. org/web/20140122163130/http://www. redhat. com/f/summitfiles/presentation/May31/Open% 20Source% 20Dynamics/Troan_{O}{p}{e}nSourceProprietyPersp. pdf,* 2005.

[64] T. J. Gandomani, H. Zulzalil, A. A. A. Ghani та A. B. M. D. Sultan, «A systematic literature review on relationship between agile methods and open source software development methodology,» *arXiv preprint arXiv:1302.2748,* 2013.

[65] J.-P. Arcangeli, R. Boujbel та S. Leriche, «Automatic deployment of distributed software systems: Definitions and state of the art,» *Journal of Systems and Software,* т. 103, p. 198–218, 2015.

[66] *SELECTING A DEVELOPMENT APPROACH,* 2008.

[67] C. Larman та V. R. Basili, «Iterative and incremental developments. a brief history,» *Computer,* т. 36, p. 47–56, 2003.

[68] W. W. Royce, «Managing the development of large software systems: concepts and techniques,» в *Proceedings of the 9th international conference on Software*

Engineering, 1987.

[69] I. Gräßler, A new V-Model for interdisciplinary product engineering, Universitätsbibliothek Ilmenau, 2017.

[70] K. Forsberg та H. Mooz, «The relationship of system engineering to the project cycle,» в *INCOSE international symposium*, 1991.

[71] C. Johansson та C. Bucanac, «The v-model,» *IDE, University Of Karlskrona, Ronneby,* 1999.

[72] A. B. M. Moniruzzaman та D. S. A. Hossain, «Comparative Study on Agile software development methodologies,» *arXiv preprint arXiv:1307.3356*, 2013.

[73] V. Farcic, «Software development models: Iterative and incremental development,» *Technology Conversations.[online]: https://technologyconversations. com/2014/01/21/software-development-modelsiterative-and-incremental-development/(Retrieved 26-10-2020),* 2014.

[74] Scrum.org, *What is Scrum?.*

[75] Scrum.org, *The 2020 scrum GUIDETM.*

[76] N. Kirovska та S. Koceski, «Usage of Kanban methodology at software development teams,» *Journal of applied economics and business*, т. 3, p. 25–34, 2015.

[77] J. M. Gross та K. R. McInnis, Kanban made simple: demystifying and applying Toyota's legendary manufacturing process, AMACOM books, 2003.

[78] *What is a Kanban Board?*, 2022.

[79] D. Misevičiūtė, *Kanban cycle time: The Ultimate Guide (2022),* 2022.

[80] *What is velocity in Agile?*, 2022.

[81] *Kanban Guide for scrum teams.*

[82] *The origins of devops: What's in a name?.*

[83] L. Leite, C. Rocha, F. Kon, D. Milojicic та P. Meirelles, «A survey of DevOps concepts and challenges,» *ACM Computing Surveys (CSUR)*, т. 52, p. 1–35, 2019.

[84] *What is devops? - practices and benefits explained.*

[85] Openhub.net, Black Duck Software, Inc..

[86] T. A. Limoncelli, «Gitops: A path to more self-service it: Iac+ pr= gitops,» *Queue*, т. 16, p. 13–26, 2018.

[87] C. Castellanos, D. Correal та J.-D. Rodriguez, «Executing architectural models for big data analytics,» в *European Conference on Software Architecture*, 2018.

[88] J. T. Bell, «Extreme programming,» 2001.

[89] N. R. Darwish та others, «Enhancements In Scum Framework Using Extreme Programming Practices,» *International Journal of Intelligent Computing and Information Sciences (IJICIS), Ain Shams University*, т. 14, p. 53–67, 2014.

[90] B. Han та J. Xie, «Practical experience: Adopt agile methodology combined with kanban for virtual reality development,» 2012.

[91] C. Yang, P. Liang та P. Avgeriou, «A systematic mapping study on the combination of software architecture and agile development,» *Journal of Systems and Software*, т. 111, p. 157–184, 2016.

[92] P. Kruchten, H. Obbink та J. Stafford, «The past, present, and future for software architecture,» *IEEE software*, т. 23, p. 22–30, 2006.

[93] H. P. Breivold, I. Crnkovic та M. Larsson, «A systematic review of software architecture evolution research,» *Information and Software Technology*, т. 54, p. 16–40, 2012.

[94] B. J. Williams та J. C. Carver, «Characterizing software architecture changes: A systematic review,» *Information and Software Technology*, т. 52, p. 31–51, 2010.

[95] E. Woods, «Software architecture in a changing world,» *IEEE Software*, т. 33, p. 94–97, 2016.

[96] W. Hasselbring, «Software architecture: Past, present, future,» в *The Essence of Software Engineering*, Springer, Cham, 2018, p. 169–184.

[97] Z. Dragičević та S. Bošnjak, «Agile architecture in the digital era: Trends and practices,» *Strategic Management*, т. 24, p. 12–33, 2019.

[98] M. A. Babar, A. W. Brown and I. Mistrik, Agile Software Architecture: Aligning

agile processes and software architectures, Newnes, 2013.

[99] J. O. Coplien and G. Bjornvig, Lean Architecture, Chichester, England: John Wiley & Sons, 2010.

[100] M. Ali Babar, «A framework for groupware-supported software architecture evaluation process in global software development,» *Journal of Software: Evolution and Process*, т. 24, p. 207–229, 2012.

[101] R. Jolak, M. Savary-Leblanc, M. Dalibor, A. Wortmann, R. Hebig, J. Vincur, I. Polasek, X. Le Pallec, S. Gérard та M. R. V. Chaudron, «Software engineering whispers: The effect of textual vs. graphical software design descriptions on software design communication,» *Empirical Software Engineering*, т. 25, p. 4427–4471, 2020.

[102] G. Matturro, F. Raschetti та C. Fontán, «A Systematic Mapping Study on Soft Skills in Software Engineering.,» *J. Univers. Comput. Sci.*, т. 25, p. 16–41, 2019.

[103] W. N. Behutiye, P. Rodríguez, M. Oivo та A. Tosun, «Analyzing the concept of technical debt in the context of agile software development: A systematic literature review,» *Information and Software Technology*, т. 82, p. 139–158, 2017.

[104] M. Soliman, P. Avgeriou та Y. Li, «Architectural design decisions that incur technical debt—An industrial case study,» *Information and Software Technology*, т. 139, p. 106669, 2021.

[105] S. Betz, S. Fricker, A. Moss, W. Afzal, M. Svahnberg, C. Wohlin, J. Börstler, T. Gorschek та others, «An evolutionary perspective on socio-technical congruence: The rubber band effect,» в *2013 3rd International Workshop on Replication in Empirical Software Engineering Research*, 2013.

[106] J. Venn, «I. On the diagrammatic and mechanical representation of propositions and reasonings,» *The London, Edinburgh, and Dublin philosophical magazine and journal of science*, т. 10, p. 1–18, 1880.

[107] M. Seckler, S. Heinz, J. A. Bargas-Avila, K. Opwis та A. N. Tuch, «Designing usable web forms: empirical evaluation of web form improvement guidelines,» в

Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, 2014.

[108] U. Reja, K. L. Manfreda, V. Hlebec та V. Vehovar, «Open-ended vs. close-ended questions in web questionnaires,» *Developments in applied statistics*, т. 19, p. 159–177, 2003.

[109] N. Vasantha Raju та N. S. Harinarayana, «Online survey tools: A case study of Google Forms,» в *National Conference on Scientific, Computational & Information Research Trends in Engineering, GSSS-IETW, Mysore*, 2016.

[110] Google, *Query function*, Google.

[111] Google, *Query language reference (version 0.7) | charts | google developers*, Google.

[112] C. Flathmann, B. Schelble, B. Tubre, N. McNeese та P. Rodeghero, «Invoking Principles of Groupware to Develop and Evaluate Present and Future Human-Agent Teams,» в *Proceedings of the 8th International Conference on Human-Agent Interaction*, 2020.

[113] A. Sarwar, Y. Hafeez, S. Hussain та S. Yang, «Towards taxonomical-based situational model to improve the quality of agile distributed teams,» *IEEE Access*, т. 8, p. 6812–6826, 2020.

[114] C. Wohlin, M. Kalinowski, K. R. Felizardo та E. Mendes, «Successful combination of database search and snowballing for identification of primary studies in systematic literature studies,» *Information and Software Technology*, т. 147, p. 106908, 2022.

[115] B. Shevchenko, *SLR-References*, Google.

[116] A. Satyanarayan, D. Moritz, K. Wongsuphasawat та J. Heer, «Vega-lite: A grammar of interactive graphics,» *IEEE transactions on visualization and computer graphics*, т. 23, p. 341–350, 2016.

[117] S. Brin та L. Page, «The anatomy of a large-scale hypertextual web search engine,» *Computer networks and ISDN systems*, т. 30, p. 107–117, 1998.

[118] H. G. Gauch Jr та H. G. Gauch, Scientific method in practice, Cambridge University Press, 2003.

[119] R. Gaina, *How to market your software or SAAS product in 11 easy steps*, 2022.

[120] C. Wohlin, E. Papatheocharous, J. Carlson, K. Petersen, E. Alégroth, J. Axelsson, D. Badampudi, M. Borg, A. Cicchetti, F. Ciccozzi та others, «Towards evidence-based decision-making for identification and usage of assets in composite software: A research roadmap,» *Journal of Software: Evolution and Process*, т. 33, p. e2345, 2021.

[121] T. Dyba, B. A. Kitchenham та M. Jorgensen, «Evidence-based software engineering for practitioners,» *IEEE software*, т. 22, p. 58–65, 2005.

[122] D. P. Kristiadi, Y. Udjaja, B. Supangat, R. Y. Prameswara, H. L. H. S. Warnars, Y. Heryadi та W. Kusakunniran, «The effect of UI, UX and GX on video games,» в *2017 IEEE International Conference on Cybernetics and Computational Intelligence (CyberneticsCom)*, 2017.

[123] H. Malallah, S. R. M. Zeebaree, R. R. Zebari, M. A. M. Sadeeq, Z. S. Ageed, I. M. Ibrahim, H. M. Yasin та K. J. Merceedi, «A comprehensive study of kernel (issues and concepts) in different operating systems,» *Asian Journal of Research in Computer Science*, т. 8, p. 16–31, 2021.

[124] C. Rieger та T. A. Majchrzak, «Towards the definitive evaluation framework for cross-platform app development approaches,» *Journal of Systems and Software*, т. 153, p. 175–199, 2019.

[125] T. Li, T. Xia, H. Wang, Z. Tu, S. Tarkoma, Z. Han та P. Hui, «Smartphone app usage analysis: datasets, methods, and applications,» *IEEE Communications Surveys & Tutorials*, 2022.

[126] M. Pečujlija та D. Petrović, «Smartphone OS and User Emotion and Ethics,» *Tehnički vjesnik,* т. 27, p. 853–859, 2020.

[127] B. Spies та M. Mock, «An Evaluation of WebAssembly in Non-Web Environments,» в *2021 XLVII Latin American Computing Conference (CLEI)*,

2021.

[128] M. Kerwin, «The "file" URI Scheme,» RFC Editor, 2017.

[129] *Security in depth: Local Web Pages,* 2008.

[130] *Data urls - http: MDN.*

[131] M. Moog, M. Demmel, M. Backes та A. Fass, «Statically detecting javascript obfuscation and minification techniques in the wild,» в *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2021.

[132] *2022 stats on top JS frameworks: React, Vue, svelte & angular,* 2022.

[133] A. Ivanovs, *The most popular front-end frameworks in 2023,* 2023.

[134] EvanLi, *Ranking/javascript.md at b11c66e340b63b604ae8fc46d7304c6dde07c8ed · Evanli/github-ranking,* 2023.

[135] *Stack overflow developer survey 2022.*

[136] Photonstorm, *Photonstorm/phaser: Phaser is a fun, free and Fast 2D game framework for making HTML5 games for desktop and mobile web browsers, supporting canvas and webgl rendering..*

[137] A. H. C. vanderHeijden, *Perception for selection, selection for action, and action for perception,* т. 3, PSYCHOLOGY PRESS 27 CHURCH RD, HOVE BN3 2FA, EAST SUSSEX, ENGLAND, 1996, p. 357–361.

[138] P. Cowan, *Write fewer tests by creating better typescript types,* 2022.

[139] J. O. Coplien та G. Bjørnvig, Lean architecture: for agile software development, John Wiley & Sons, 2011.

[140] *Update on Meta's year of efficiency,* 2023.

[141] B. Cartaxo, G. Pinto та S. Soares, «The role of rapid reviews in supporting decision-making in software engineering practice,» в *Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering 2018,* 2018.

[142] R. L. S. M. M. Lynn, S. M. Manager та R. Lynn, *The history of Agile,* 2019.

[143] D. L. Driscoll, «Introduction to primary research: Observations, surveys, and interviews,» *Writing spaces: Readings on writing*, т. 2, p. 153–174, 2011.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ ЗА НАУКОВИМИ НАПРЯМАМИ КЕРІВНИКА ТА НАУКОВЦІВ КАФЕДРИ ПРОГРАМНОЇ ІНЖЕНЕРІЇ

[2] O. Bezsmertnyi, N. Golian, V. Golian та I. Afanasieva, «Behavior Driven Development Approach in the Modern Quality Control Process,» в *2020 IEEE International Conference on Problems of Infocommunications. Science and Technology (PIC S&T)*, 2020.

[3] I. Gruzdo, I. Kyrychenko, G. Tereshchenko та N. Shanidze, «Metrics Applicable for Evaluating Software at the Design Stage.,» в *COLINS*, 2021.

[5] V. Kyriy, I. Sheiko та R. Petrova, «Optimization of management information support as a basis for organizational transformations at an enterprise,» *Periodicals of Engineering and Natural Sciences*, т. 7, p. 679–689, 2019.