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

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач випускової кафедри
_________ Аліна САВЧЕНКО
«__»_______________ 2023 р.

КВАЛІФІКАЦІЙНА РОБОТА

(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСКНИКА ОСВІТНЬОГО СТУПЕНЯ МАГІСТР
ЗА ОСВІТНЬО-ПРОФЕСІЙНОЮ ПРОГРАМОЮ
«ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ ПРОЕКТУВАННЯ»

Тема: «Технологія координації команд при гнучкій розробці великих програмних проектів»

Виконавець: Марк ВРУБЛЕВСЬКИЙ

Керівник: к.т.н., доц. Олександр ХАРЧЕНКО

Нормоконтролер: к.т.н., доц. Олександр Шевченко

КИЇВ 2023

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет Комп’ютерних наук та технологій

Кафедра Комп’ютерних інформаційних технологій

Спеціальність 122 «Комп’ютерні науки»

Галузь знань, спеціальність, освітньо-професійна програма: 12 “Інформаційні технології”, 122 “Комп’ютерні науки”, “Інформаційні управляючі системи та технології”

ЗАТВЕРДЖУЮ:

Завідувач кафедри КІТ

_______ Аліна САВЧЕНКО

«_____»__________ 2023 р.

ЗАВДАННЯ

на виконання кваліфікаційної роботи

Врублевського Марка Дмитровича

(ПІБ випускника)

1. Тема роботи: «Технологія координації команд при гнучкій розробці великих програмних проектів» затверджена наказом ректора № 623/ст. від 01.05.2023р.

2. Термін виконання роботи: з 15 травня 2023 року по 19 червня 2023 року.

3. Вихідні дані до роботи: Дослідження методів координації у великих командах.

4. Зміст пояснювальної записки: 1. Основні положення гнучких технологій проектування програмного забезпечення.(Agile) 2. Методи координації при гнучкій розробці великомасштабних програмних проектів. 3. Система підтримки процесів координації.

5. Перелік обов’язкового ілюстративного матеріалу: 1.Актуальність, 2.Схеми використання Agile технологій, 3.SafE, 4. Таблиці порівняння.

6. Календарний план-графік

№ з/п	Завдання	Термін виконання	Підпис керівника
1.	Ознайомлення та аналіз основних Agile технологій. Написання 1 розділу, представлення керівнику	22.05.2023- 01.06.2023	
2.	Аналіз методів координації команд. Написання 2 розділу, представлення керівнику	02.06.2023- 10.06.2023	
3.	Дослідження системи координації. Написання 3 розділу, представлення керівнику	11.06.2023- 15.06.2023	
4.	Загальне редагування та друк пояснювальної записки	15.06.2023- 16.06.2023	
5.	Проходження нормоконтролю, переплiт пояснювальної записки.	17.06.2023- 18.06.2023	
6.	Розробка тексту доповіді. Оформлення графічного матеріалу для презентації	18.06.2023- 19.06.2023	

7. Дата видачі завдання 22.05.2023р.

Керівник кваліфікаційної роботи ________ Олександр ХАРЧЕНКО

(підпис керівника)

Завдання прийняв до виконання ________ Марк ВРУБЛЕВСЬКИЙ

(підпис випускника)

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи на тему: «Технологія координації команд при гнучкій розробці великих програмних проектів» містить: 99 сторінок, 11 рисунків, 24 інформаційних джерел, 12 таблиць.

Об'єкт дослідження – Методології огранізацій координаційного процесу в командах розробки.

Предмет дослідження – Пропозиції по покращенню координації в умовах гнучкої розробки великих проектів.

Мета кваліфікаційної роботи – огляд використання гнучких технологій та їх різновид, шляхи до покращення координації команд та якості програмних продуктів.

Методи дослідження – аналіз літературних джерел таких як: наукові статті, дослідження, реферати. Порівняння досвіду різних компаній.

Результати кваліфікаційної роботи рекомендується використовувати як допоміжний матеріал в наукових дослідженнях та при налагодженні координації в практичних випадках.

ГНУЧКА РОЗРОБКА, КООРДИНАЦІЯ МІЖ КОМАНДАМИ, ГНУЧКІ МЕТОДОЛОГІЇ, ПРОГРАМНА ПЛАТФОРМА.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ....................................6
ВСТУП...7
РОЗДІЛ 1. ОСНОВНІ ПОЛОЖЕННЯ ГНУЧКИХ ТЕХНОЛОГІЙ ПРОЕКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ.(AGILE)..10
1.1. Основні принципиAgile..11
1.2. Agile першого покоління...13
1.3. Agile другого покоління...20
1.4. Висновки до розділу 1...28
РОЗДІЛ 2. МЕТОДИ КООРДИНАЦІЇ ПРИ ГНУЧКІЙ РОЗРОБЦІ ВЕЛИКОМАСШТАБНИХ ПРОГРАМНИХ ПРОЕКТІВ..29
2.1. Види координації..29
2.2. Програмні платформи підтримки методів координації....................................39
2.3. Порівняльне оцінювання , та вибір програмної платформи...........................48
2.4. Висновки до розділу 2...55
РОЗДІЛ 3. СИСТЕМА ПІДТРИМКИ ПРОЦЕСІВ КООРДИНАЦІЇ...............................56
3.1. Емпіричні дослідження застосування методів координації при розробці конкретних проектів...57
3.2. Концепція ефективності координації...66
3.3. Вибір платформи координації...76
3.4. Висновки до розділу 3...79
ВИСНОВКИ..80
СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ ВИКОРИСТАНИХ ДЖЕРЕЛ.................81
ДОДАТКИ..83

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

Agile - клас методологій розробки програмного забезпечення, що базується на ітеративній розробці, в якій вимоги та розв'язки еволюціонують через співпрацю між багатофункціональними командами здатними до самоорганізації.

SAFe - набір загальновизначених принципів, підходів, шаблонів робочого процесу який базується на методологіях гнучкої розробки (Agile).

Scrum - підхід управління проєктами для гнучкої розробки програмного забезпечення.

ЦК – центральна команда

МКР – малі команди розробки

БКС – багатоканальна система

ВСТУП

У сучасному світі розробка програмного забезпечення відіграє важливу роль в бізнес-процесах та повсякденному житті людей. При розробці великих програмних проектів, таких як веб-додатки або мобільні додатки, командна робота стає надзвичайно важливою. Технології гнучкої розробки дозволяють зменшити терміни виконання проектів та підвищити якість продукту.

Однак, координація команди при гнучкій розробці великих програмних проектів може бути складною задачею. Команди повинні взаємодіяти між собою та замовником, дотримуватися графіку та забезпечувати високу якість продукту.

Сама координація є головною проблематикою при розробці програмного забезпечення. Згідно з Краутом і Стрітером, проблема координації діяльності при розробці великих програмних систем є головною причиною кризи програмного забезпечення, хоча це не єдиний фактор. Розробка програмного забезпечення пов'язана з керуванням багатьма залежностями та високим рівнем невизначеності щодо продуктів і технологій. Раніше дослідження були зосереджені на координації в традиційних проектах програмного забезпечення, глобальній розробці програмного забезпечення та, останнім часом, на гнучкій розробці.

У середині 2000-х років, дослідження в галузі програмної інженерії були спрямовані на глобальну програмну інженерію, де ключовою проблемою була координація між розподіленими командами. Збіг залежностей та координа-ційних дій є критичним як у добре відомих випадках, так і в контекстах з високим рівнем невизначеності [1]. Проте залишається відкритим питання про те, які практики є найбільш ефективними. У своїй статті "Глобальна інженерія програмного забезпечення: майбутнє соціально-технічної координації" (Herbsleb, 2007, стор. 9), Гербслеб зазначив, що хоча « наразі ми маємо низку індивідуальних рішень, таких як інструменти, практики та методи, … ми дуже мало розуміємо про компроміси між ними та умови їх застосування ».

В останні роки у дослідженнях програмної інженерії було звернуто увагу на гнучкі методи розробки програмного забезпечення [2],де розробка організована як командна робота. Pries-Heje [3] вказали на успіх гнучкого методу Scrum завдяки його гнучким та ефективним структурам координації, таким як спільний список робочих завдань у резерві продукту та спринт-беклог, щоденні зустрічі в команді та використання наочного табло для відображення стану роботи. Strode та ін.[4] запропонували модель координації для гнучких команд, які працюють разом, з акцентом на синхронізацію всередині гнучкої команди, близькість, яка дозволяє спілкуватися віч-на-віч, та діяльність, націлену на зовнішніх зацікавлених сторін, що вони назвали охопленням кордонів.

В сучасних великих IT-проектах з десятками команд розробників все більше застосовуються гнучкі методи розробки. Проте емпіричні дослідження свідчать про проблеми з координацією, такі як порушення залежностей між командами [5], недостатньою обізнаністю та невідповідністю порад щодо методів координації потребам проекту з плином часу [2]. Такі залежності можуть підривати автономію, яка є важливою для гнучких розробних команд[6].

Існуюча теорія недостатня для пояснення координації в контексті великомасштабної гнучкої розробки, оскільки вона має характеристики, відмінні від традиційних організацій та розподіленої розробки, щодо спрямованості на усне спілкування, роботи в командах та частої зміни механізмів координації з часом [2]. Систематичний огляд літератури про великомасштабну гнучку розробку повідомляє про проблеми координації, включаючи синхронізацію команд, боротьбу з комунікаційним перевантаженням та зменшення зовнішнього відволікання.

Масштабні гнучкі проекти розробки є критично важливими для організацій, оскільки вони призводять до значних витрат та ризиків. Ефективна координація є ключовим фактором успіху проекту та своєчасного його завершення. Наукова спільнота має надати розуміння та рекомендації щодо координації в цьому конкретному контексті. Стратегії координації описані в методологіях розробки, і покращення нашого розуміння ефективності цих підходів та того, в яких контекстах вони є ефективними, є важливим завданням.

Зараз багато організацій переходять до більш адаптованих великомасштабних методів другого покоління розробки програмного забезпечення, що замінюють практики управління проектами на практики розробки. Це призводить до іншого підходу до координації, замінюючи попередні рішення, практики та інструменти. Важливо зрозуміти, як нове покоління методів впливає на успіх проекту. Особлива увага повинна бути приділена координації, яка є важливим фактором успіху проекту. У цій дипломній роботі я розглянув міжкомандну координацію, яка є одним з основних аспектів великомасштабної гнучкої розробки програмного забезпечення, та її вплив на загальний успіх проекту.

РОЗДІЛ 1
ОСНОВНІ ПОЛОЖЕННЯ ГНУЧКИХ ТЕХНОЛОГІЙ ПРОЕКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ.(AGILE)

Гнучка методологія також називається гнучкою розробкою програмного забезпечення та становить категорію методів розробки програмного забезпечення, керованих ітераційним процесом. Його процвітання залежить від хорошої комунікації в багатогранних командах, які характеризуються здатністю до самоврядування. Примітно, що основною метою створення Agile було підвищення потенціалу серед розробників у сфері програмування.

Зокрема, більшість гнучких методологій спрямовані на зниження потенційних ризиків, одночасно оптимізуючи ефективність за допомогою короткострокових циклів розробки, визначених як ітерації тривалістю від одного до двох тижнів. За цією методологією розробки лежить процес, у якому кожна ітерація діє як незалежна одиниця проекту, що складається з кількох компонентів, таких як концептуалізація, аналіз потреб користувачів і вимог, необхідних для розробки ключових функціональних можливостей, зосереджуючись на модулях кодування перед завершенням процедур тестування та документування. Процедурно кажучи; в той час як окремі ітерації можуть не надавати щоразу абсолютно нові версії продуктів; його швидкий підхід гарантує, що всі продукти швидко реагують як на макро-, так і на мікрорівнях під час доставки. Після завершення кожного ітераційного циклу під час проектів розробки програмного забезпечення, що працюють за гнучкими методологіями, такими як фреймворки Agile та Scrum, команди зазвичай ретельно перевіряють свої вихідні списки пріоритетів, щоб відобразити отримані знання з цієї ітерації

Особливо структуровані навколо філософії, яка наголошує на фізичній взаємодії над цифровими інструментами, коли це можливо. Ці міжфункціональні групи працюють злагоджено в одному централізованому робочому просторі, який часто називають КПЗ. Це може включати всіх відповідних учасників, таких як власники проекту або зацікавлені сторони, які формують функціональні вимоги разом із спеціальним персоналом для тестування; дизайнери користувацького досвіду, зосереджені на покращенні інтерфейсів; усіма вони легко керуються в рамках досвідчених структур керівництва, які відіграють важливу роль у збереженні високого імпульсу. Вимірювання успіху в гнучких підходах зосереджується на постачанні корисного кінцевого продукту.

Зосереджуючись на безперервному спілкуванні, Agile-фреймворки оптимізують обсяг необхідної текстової документації порівняно з традиційними підходами до розробки програмного забезпечення. Проте дехто може обґрунтовано критикувати ці підходи як такі, що їм бракує необхідного рівня структури чи правозастосування в їхній практиці.

1.1. Основні принципи Agile

Розвиток технологій та методологій розробки програмного забезпечення призвів до необхідності систематизації ідей Agile. Тому з 11 по 13 лютого 2001 року, на гірськолижному курорті в горах Юти був написаний Маніфест гнучкої розробки. Маніфест підписали представники наступних методологій Extreme programming, Scrum, DSDM, Adaptive software development, Crystal Clear, Feature driven development, Pragmatic Programming. Agile Manifesto містить 4 основні ідеї та 12 принципів. Примітно, що Agile Manifesto не містить практичних порад.

Основні ідеї:

- Люди та співпраця важливіші за процеси та інструменти;
- Працюючий продукт важливіший за вичерпну документацію;
- Позитивна співпраця із замовником важливіша за обговорення; умов контракту;

- Готовність до змін важливіша за дотримання плану.

Принципи, які роз'яснює Agile Manifesto:

- Задоволення клієнта за рахунок ранньої та безперебійної поставки коштовного програмного забезпечення;
- Вітання змін вимог навіть наприкінці розробки (це може підвищити конкурентоспроможність отриманого продукту);
- Часта поставка робочого програмного забезпечення (кожен місяць або тиждень або ще частіше);
- Тісне, щоденне спілкування замовника з розробниками впродовж всього проєкту;
- Проєктом займаються мотивовані особистості, які забезпечені потрібними умовами роботи, підтримкою і довірою;
- Рекомендований метод передачі інформації — особиста розмова (віч-на-віч);
- Робоче програмне забезпечення — найкращий вимірювач прогресу;
- Спонсори, розробники та користувачі повинні мати можливість підтримувати постійний темп на невизначений термін;
- Постійна увага поліпшенню технічної досконалості та зручному дизайну;
- Простота — мистецтво не робити зайвої роботи;
- Найкращі технічні вимоги, дизайн та архітектура виходять у самоорганізованої команди;
- Постійна адаптація до мінливих обставин.

Маніфест та Принципи гнучкої розробки містять високорівневі ідеї щодо того, як потрібно вибудовувати процес розробки програмного забезпечення, щоб успішно завершувати проєкти й створювати команди, в яких приємно та цікаво працювати. Документи визначають, що потрібно для цього зробити, але не говорять, як це зробити. По-іншому й не могло бути, оскільки Маніфест та Принципи народилися внаслідок консенсусу представників різних (хоча й

споріднених) напрямів, які могли знайти спільну основу лише на рівні базових цінностей та принципів.

1.1. Agile першого покоління(гібридні технології)

Гібридна гнучка методологія - це модель проектування програмного забезпечення, яка поєднує плановий підхід з гнучкими методами. Метою гібридного підходу є поєднання найкращих аспектів гнучких та традиційних методів у проекті програмного забезпечення. Багато організацій успішно використовують гібридну гнучку методологію для управління великомасштабними проектами, спрощення підготовки належної документації та покращення техніки бізнес-аналізу. Поєднання планового підходу з гнучкою методологією також збільшує продуктивність команди завдяки співпраці з зацікавленими сторонами, щоб забезпечити правильний хід розробки. Крім того, Шпундак зазначає, що гібридні моделі використовуються в проектах розробки програмного забезпечення через потребу в різних методологіях з унікальними характеристиками та перевагами та недоліками в межах одного проекту. Однією з активно використовуваних гібридних гнучких моделей командою розробників програмного забезпечення є поєднання scrum з моделлю waterfall. Існує кілька назв, що використовуються для позначення поєднання обох моделей, таких як scrum і waterfall, scrumfall, water-scrum-fall, а також інші гібридні гнучкі моделі, такі як гібридна V-модель та гнучка етапно-воротна модель. Дане дослідження є продовженням попереднього дослідження, в якому встановлено, що команда розробників програмного забезпечення поєднує scrum і waterfall у проектах програмного забезпечення. Це поєднання призвело до створення гібридної гнучкої моделі.

Scrum

Scrum є однією з гнучких методологій, яку широко використовують практики. Згідно з дослідженням, 66% команд на рівні проекту використовують Scrum, за яким слідує гібридна гнучка модель scrumban з рівнем використання 9%

. Scrum став однією з переважаючих моделей розробки завдяки його здатності до частої комунікації з власником продукту під час постійних ітерацій розвиваючогося програмного забезпечення . Ітерації, інкрементальний розвиток, самоорганізовані команди та адаптабельність до змінних потреб - це всі характеристики Scrum, які відповідають гнучким підходам . Фраза "ітеративний підхід" описує розбиття тривалості проекту на ітерації або спринти, де весь проект розбивається на кілька менших ініціатив . Кожен спринт має однаковий формат. Scrum став однією з переважаючих моделей розробки завдяки його здатності до частої комунікації з власником продукту під час постійних ітерацій розвиваючогося програмного забезпечення.

На рисунку 1.1 зображено рамки Scrum, які включають беклог продукту, планування спринту, беклог спринту та спринт-ретроспективи, які виконуються ітеративно. Великий проект Scrum розбивається на керовані спринти, під час кожного з яких виконуються аналіз, програмування та тестування . Як показано на рисунку 1, Scrum починається з беклогу продукту. Беклоги - це список завдань, які призначаються і повинні бути виконані командою Scrum протягом певного проміжку часу, а беклоги продукту будуть переорганізовані і пріоритизовані згідно з визначеними критеріями, такими як пріоритет у беклогу спринту. Беклог спринту складається зі списку завдань, які мають бути виконані командою Scrum протягом спринту. Кожен член команди повинен оновлювати свій щоденний прогрес на зустрічі, відомій як щоденний Scrum. Щоденний Scrum допомагає членам команди Scrum переконатися, що вони досягають мети спринту та забезпечують виконання проекту. Крім того, зустрічі Scrum можуть допомогти членам команди сформулювати чітку ідею про свої призначені завдання, і кожен учасник Scrum повинен продемонструвати свій прогрес під час зустрічі Scrum . Крос-функціональні розроблювальні команди співпрацюють для виконання визначених завдань та забезпечення успішного завершення спринту, і, як зазначає Sachdeva , важливо, щоб команда підтримувала якість та максимізувала продуктивність на довгостроковій основі, а також координувала і допомагала один одному при виконанні роботи з використанням різних наборів навичок. Крім того,

в Scrum всі члени команди постійно співпрацюють між собою, щоб досягти спільної мети . Потім, на кінці кожного спринту проводиться огляд спринту для оцінки проекту відносно цілей спринту, визначених під час зустрічі планування спринту. Нарешті, проводиться ретроспектива спринту - регулярна зустріч, яка відбувається в кінці спринту, з метою оцінки того, що пройшло добре, і того, що можна покращити в наступному циклі спринту. Ретроспектива - це важлива складова методології Scrum для створення, постачання та управління складними проектами. Її мета полягає в ідентифікації досягнень і помилок попереднього спринту та перетворенні отриманих досвіду на пропозиції щодо покращення.

Scrum передбачає постійну комунікацію з власником продукту протягом неперервних ітерацій розвиваючогося програмного забезпечення , що є необхідним для забезпечення відповідності продукту цілям та очікуванням власника продукту. Крім того, часта комунікація з зацікавленими сторонами допомагає командам виявляти виникаючі проблеми та знаходити рішення для вчасного усунення проблем у процесі постійного вдосконалення. Також, визначення того, що пройшло добре, і що пішло не так, допомагає команді виявляти потребу в покращенні. Scrum сприяє збільшенню продуктивності команди і може застосовуватись у будь-якому проекті будь-якого розміру. Scrum був розроблений для підвищення швидкості розробки, узгодження індивідуальних та організаційних цілей, визначення культури, спрямованої на досягнення результативності, підтримки створення вартості для акціонерів шляхом ефективної комунікації результативності на всіх рівнях, поліпшення індивідуального розвитку та якості життя. Крім того, Scrum допомагає командам завершувати продукти вчасно та послідовно , забезпечує раціональне витрачання грошей та часу, а також залучення замовника в постійний зворотний зв'язок . Крім того, залучення замовника може бути цінним активом для допомоги керівникам проектів та лідерам у розробці відповідних стратегій для реалізації їх проектів. Таким чином, Scrum підходить, коли вимоги проекту продовжують змінюватись, потрібен частий зворотний зв'язок, а команді проекту потрібен певний рівень гнучкості у проектуванні способів постачання результатів та дослідженні нового

досвіду з новим проектом, який команда раніше не виконувала. Щодо роботи з продуктами на довгостроковій основі, Scrum допомагає усунути помилки та економить час. Scrum базується на перевагах екстремального програмування, щоб зробити його більш систематичним для визначення напрямку розвитку.

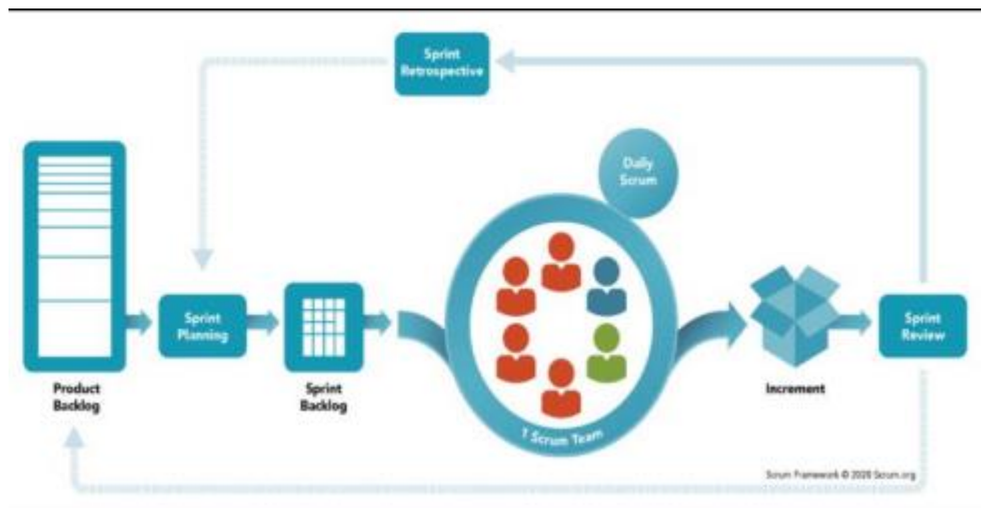

Рис. 1.1. Фреймворк Scrum

Waterfall

Модель «водоспад» є одним з найстаріших життєвих циклів розробки програмного забезпечення (ЖЦРЗ). Уїнстон В. Ройс заснував модель «водоспад» у 1970 році, а її основою є п'ять фаз: визначення вимог, проектування, реалізація, перевірка та супровід. Модель «водоспад» відома своїм лінійним підходом до розвитку фаз та іноді називається класичним життєвим циклом. Пресман та Максим зазначають, що модель «водоспад» сприяє систематичному та послідовному підходу до фаз розробки програмного забезпечення, який починається з уточнення вимог замовника та продовжується через планування, моделювання, конструкцію та впровадження.

На рисунку 1.2 зображені фази моделі "водоспад". Комунікація - перша фаза в моделі "водоспад". Це етап, на якому збираються вимоги користувачів, а комунікаційний процес включає у себе зацікавлених сторін у процес збору вимог. Після завершення фази комунікації настає фаза планування. Фаза планування встановлює часову лінію та віхи проекту, а також стратегію оцінки вартості,

розкладу та контролю проекту. Після завершення фази планування починається фаза моделювання. Фаза моделювання - це етап, на якому аналізуються вимоги, а команда програмістів проектує систему. Аналіз та проектування базуються на вимогах, зібраних на етапі комунікації, та деталях, отриманих на етапі планування. Потім відбувається фаза конструкції. Фаза конструкції включає написання коду та тестування. Нарешті, є впровадження проекту, яке включає виконання проекту, підтримку системи та зворотний зв'язок. Як видно з потоку, немає можливості повернутися до завершеної фази. Отже, жодних коригувань не можна вносити, оскільки фазу неможливо повторити. Ця модель корисна в структурованому розробці систем, де заборонено змінювати програмне забезпечення після написання коду . Модель "водоспад" робить програмне забезпечення не придатним для повторного використання, а систему важко оновлювати, оскільки весь процес потребує модифікацій при будь-якому зміні, що є витратним і часоємним . Оскільки техніка "водоспад" надає результати в кінці проекту програмного забезпечення, замовник або розробник можуть перебувати у невизначеності щодо того, чи відповідає поточний стан проекту задумці. Коли ні клієнти, ні розробники не знають, коли буде завершено або отримано проект, існує значний ризик, пов'язаний з процесом "водоспаду", який зазвичай потребує надмірного часу для управління непередбаченими ситуаціями . Однак модель "водоспад" легше керувати [6]. Крім того, перед початком проекту чітко визначаються виконавчі завдання та віхи [6], а фази моделі "водоспад" та її діяльності описані. Крім того, модель "водоспад" працює добре у великих і слабих командах [6].

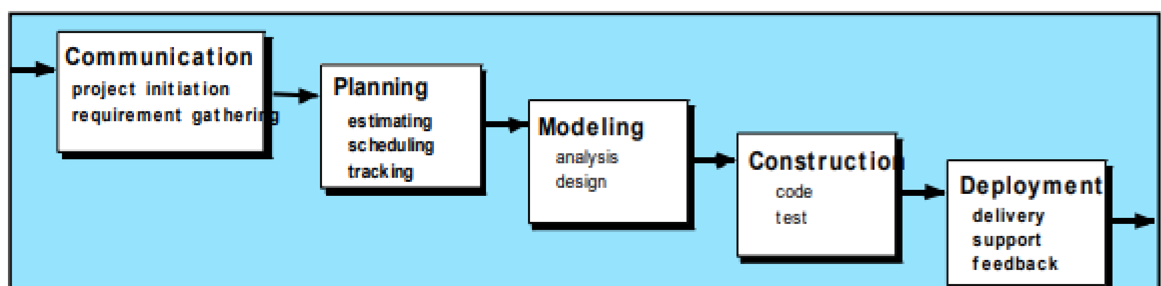

Рис 1.2. Модель Waterfall

Scrumfall

Комбінація scrum з моделлю "водоспад" призводить до гібридної гнучкої моделі, відомої як "scrumfall" [6], "water-scrum-fall" [7], [8] і "scrum and waterfall" [5]. Rahim та ін. [6] пропонують модель "scrumfall", зображену на рисунку 3, яка включає чотири фази життєвого циклу програмного забезпечення: передпочаткова, високорівневий дизайн, розробка та післяпродажна фаза. До передпочаткових дій входять комунікація, збір вимог, створення беклогу, планування проекту та оцінка вартості. У свою чергу, післяпродажна фаза спрямована на інтеграційне тестування та завершення продукту для загального розповсюдження. Модель "scrumfall", зображена на рисунку 1.3, була створена переважно для подолання недоліків моделей scrum і "водоспад". Ці недоліки в основному пов'язані з тим, що розробка в моделі scrum виконується ітеративно, з кожною ітерацією тривалістю від одного до чотирьох тижнів і з можливістю внесення змін [14], [16], [17]. Однак у практичному світі неможливо врахувати постійні зміни в вимогах на пізніших етапах спринта. Крім того, Rahim та ін. [6] заявили, що тривалість ітерації для кожного спринта занадто мала для підтримки великих і складних вимог. Таким чином, для подолання недоліків Scrum була створена модель "scrumfall", яка надає гнучкість в тривалості кожного спринта для підтримки великих і складних вимог. Rahim та ін. [6] також повідомляють, що "scrumfall" успішно працює для великих, критичних систем, географічно розподілених великих команд, де команда складається з досвідчених та недосвідчених спеціалістів. Крім того, "scrumfall" демонструє ефективність в часових, вартісних та економічних аспектах.

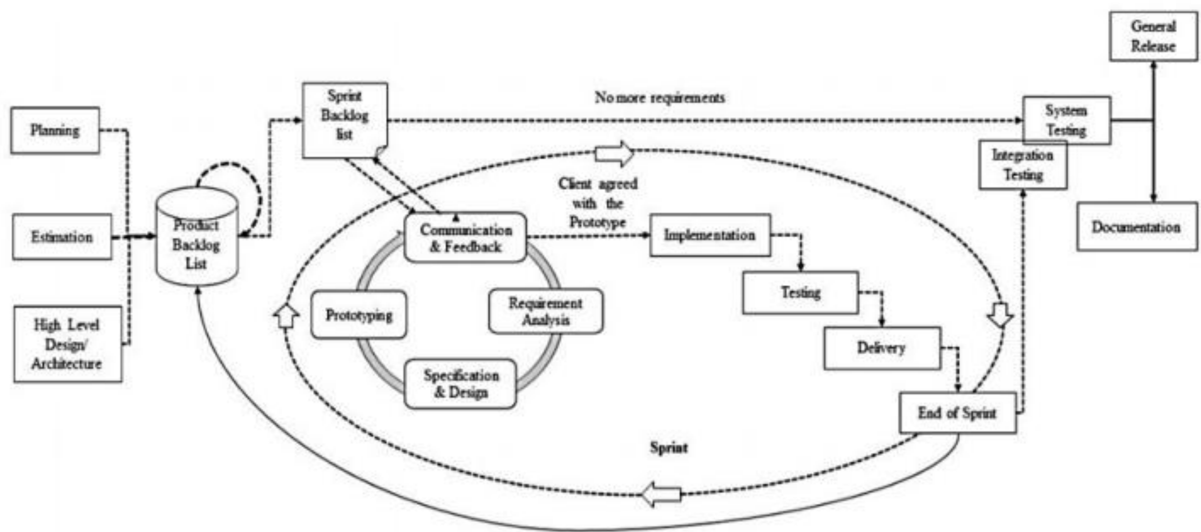

Рис 1.3. Scrumfall

Імані та ін. Провели кількісне та якісне дослідження, щоб довести, що гібридні підходи працюють краще, ніж традиційні планово-орієнтовані або гнучкі методи. Дослідження кількісно довело, що гібридний підхід може масштабуватися для проектів з високим рівнем невизначеності вимог, а також покращувати показники успішності проекту, зокрема відносно витрат. У порівнянні з цим, якісне дослідження використовувало кейс-стадію як інструмент збору даних в двох різних бізнес-організаціях, які використовували гібридні гнучкі методи, що підтверджує результати, що підтримують дві гіпотези, сформульовані в кількісному дослідженні. Кейс-стадія виявила, що гібридний гнучкий підхід можна використовувати в малих проектах з високою невизначеністю вимог. Крім того, проект успішно завершився вчасно з виміряним зменшенням витрат за рахунок використання гнучкого ітеративного розробки та фази тестування, порівняно з планово-орієнтованим підходом. Згідно зі звітом Висоцького та Орловського [12], scrumfall складається з трьох фаз. Початкова фаза, фаза розробки та кінцева фаза. Початкова фаза включає аналіз вимог та планування, що адаптується за моделлю «водоспаду». Scrum буде використовуватися протягом фази розробки, яка включає дизайн, розробку та впровадження [12]. Фази інтеграції та тестування включатимуться в кінцеву фазу.

На рисунку 1.4 показана діаграма підходу «водоспад-скрам-водоспад» і залучені в нього діяльності.

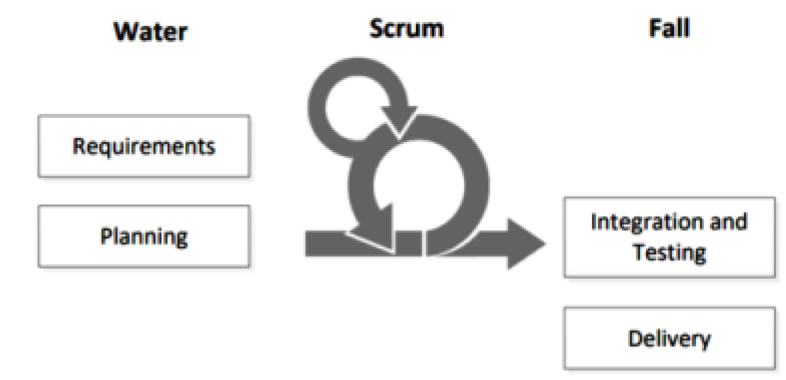

Рис. 1.4 Процес Water scrum fall

Дослідження демонструють, що немає однієї універсальної моделі, яка підходить для всіх проектів. Замість цього, необхідно адаптувати і комбінувати як гнучкі, так і негнучкі підходи для досягнення різноманітних цілей. До того ж, для планування, збору вимог, проектування, розгортання та підтримки потрібний більш класичний підхід, наприклад, модель "водоспад". При цьому розробка вимагає більшої гнучкості в ітераціях та сприяє залученню зацікавлених сторін до процесу розробки, наприклад, за допомогою скраму. З іншого боку, щодо тестування респонденти мають різні вподобання: деякі використовують модель "водоспад", а деякі - скрам. Загалом можна зробити висновок, що вибір моделі не залежить від розміру проекту або особистих уподобань команди розробників програмного забезпечення. Він повинен відповідати цілям і потребам проекту. Наявні дані свідчать про те, що респонденти погодилися, що розмір проекту, незалежно від його масштабу - великого, середнього або малого - не впливає на вибір моделі процесу для програмного проекту. Акцент робиться на тому, як обрана модель допомагатиме команді розробників програмного забезпечення досягти поставлених цілей проекту. Таким чином, можна зробити висновок, що гібридний гнучкий підхід є одним з найкращих рішень для команди розробки

програмного забезпечення для досягнення проектних цілей, прискорення процесу розробки та підвищення продуктивності команди.

1.2. Agile другого покоління

Останнім часом спостерігаєсья поширення другого покоління великомасштабних гнучких методів розробки, які замінюють більшість структурних порад з управління проектами на уроки з розробки цифрових продуктів. Серед таких методів можна виділити SAFe, Scrum-atscale, Disciplined Agile Delivery, LeSS і модель Spotify[7]. Вони зосереджуються на продукті, а не на процесі, і використовують неформальне спілкування, еволюційну модель доставки та органічну організацію для заохочення співпраці та кооперативних соціальних дій. Управління більше орієнтоване на співпрацю, а методи передбачають організацію великих проектів на основі команд, планування випуску та архітектури через дорожні карти та рекомендації, співпрацю з клієнтами та обмін знаннями між командами і типові практики для міжкомандної координації, такі як scrum of scrum зустрічі, і для обміну знаннями, такі як спільноти практики.

Як приклад можна навести багатокейсове дослідження впровадження SAFe у глобальній телекомунікаційній компанії Comptel (Paasivaara, 2017), яке описує практики на рівні команди, програми та портфоліо. Дослідження описує організацію роботи, де на рівні портфоліо використовують планування з епопеями, які надаються програмам, що називаються "поїздами гнучкого випуску". Процес розробки був проведений покроково. Кожне підвищення починалося з двохденного сеансу планування, за яким слідувала розробка протягом 10 тижнів. На цьому рівні з'явилися нові посади, такі як менеджер із продукції, системний архітектор та інженер з випуску. Інженер підготовки до випуску підготував і керував плануванням збільшення продукту, зустрічами Scrum of Scrum і піклувався про елементи вдосконалення та показники (Paasivaara, 2017, с. 4). Команди використовували гнучкі методи, такі як Scrum

або Kanban, і працювали двотижневими ітераціями. У одному кейсі було 14 команд, а в іншому - 12 команд. Також було дві команди платформ, які обслуговували обидва кейси. Команди були міжфункціональними з 5-10 учасниками. Регулярно проводилися зустрічі спільноти між власниками продуктів.

Розглянемо один з найпопулярніших методів другого покоління – Spotify.

Метод Spotify

Зі своїми вражаючими 286 мільйонами користувачів, Spotify є однією з найбільших та найуспішніших музичних платформ для прослуховування аудіо контенту. Spotify зобов'язаний своєму успіху унікальною стратегією управління, яка сприяє покращенню ефективності команд.

Коли інженерні команди Spotify працювали над поліпшенням своєї гнучкості, вони фіксували свої зусилля та ділилися своїми знаннями. Тепер їх задокументовані практики, відомі як модель Spotify, перетворили підхід до організації роботи в багатьох компаніях-розробниках програмного забезпечення.

Визначення моделі Spotify

Модель Spotify - це люди-орієнтований та автономний підхід до масштабування гнучкості, який допоміг Spotify розширити свої команди та зростати як компанії. На початкових етапах розвитку Spotify використовував принципи гнучкості, у вигляді команд Scrum, для керування своєю роботою.

Зростання компанії призвело до виявлення командою того, що стандартні практики Scrum перешкоджають їхньому прогресу. Вони вирішили відкинути правила Scrum і розробити власний підхід до гнучкої методології.

Spotify розробив свій власний набір правил, які відповідали їхньому стилю роботи та співвідносилися з їхньою корпоративною культурою. Принципи компанії стали важливішими за її практики. В результаті, Agile став важливішим за Scrum. Такі титули, як "Scrum master", були замінені на "Agile coach". Вони також використовували терміни, такі як "squad" замість "Scrum team".

Головною метою стало досягнення автономії. Вони мали на меті децентралізувати процес прийняття рішень, щоб зменшити втрату часу.

Модель Spotify походить з книги "Scaling Agile @ Spotify with Tribes, Squads, Chapters, and Guilds", опублікованої Генріком Кнібергом і Андерсом Іварссоном в 2012 році.

Генріка Кніберга в індустрії визнають "винахідником" моделі, але він наполягає на тому, що це не є правдою. Він також швидко вказує на те, що модель ніколи не була задумана для того, щоб інші компанії реплікували або застосовували її. Однак, впровадження моделі Spotify викликало цікавість у багатьох організацій. З того часу різні компанії ідеалізували та високо оцінювали модель Spotify за її простоту.

Основні компоненти моделі Spotify

Модель Spotify базується на принципі простоти. Цей легкий фреймворк сприяє автономії всередині всіх команд (Squads) та надає можливість працювати у співпраці з колегами.

У моделі Spotify існують такі основні елементи, які відрізняють її від традиційних фреймворків масштабування:

Squads

Squads (команди) є базовими одиницями гнучкого розроблення. Вони складаються з крос-функціональних команд, що налічують від 6 до 12 осіб. Кожна команда (Squad) відповідає за певні функціональності, при цьому кожна команда фокусується на розробці конкретної функції.

Команди (Squads) самоорганізуються, хоча вони отримують підтримку та керівництво від агільного коуча та власника продукту.

Кожна команда (Squad) має свою власну довготермінову місію та повинна вирішити, який фреймворк їм необхідний для досягнення успіху. Scrum, Kanban та інші агільні методології є прикладами фреймворків, якими користуються команди (Squads) для досягнення своїх цілей.

Модель Spotify впровадила таку форму автономії, щоб підвищити продуктивність та оптимізувати процес розробки.

Хоча команди (Squads) не обмежують себе жорсткими фреймворками, вони дотримуються принципів lean-розробки, таких як випуск мінімально

життєздатних продуктів (MVP) та A/B-тестування. Саме тут важлива роль відводиться агільним коучам. Вони керують командами (Squads) у процесі агільної розробки продукту та гарантують, що вони будують продукт відповідно до стратегічних цілей.

Tribes

Хоча команди (Squads) функціонують самостійно, вони не працюють у відокремленості. Функціональності різних команд (Squads) перетинаються між собою та змушують команди стати взаємозалежними. Модель Spotify не працює за принципом передачі роботи (handoffs), оскільки це уповільнює весь процес. Тому модель Spotify впровадила поняття "Tribes" (племена).

Tribes формуються, коли декілька команд (Squads) співпрацюють над однією областю функціональності. Tribes сприяють виробленню взаємодії між командами (Squads) і складаються з 40 до 150 співробітників для забезпечення взаємодії (за допомогою числа Данбара).

Кожне плем'я має лідера племені (Tribe Lead), який допомагає з координацією та співпрацею команд (Squads). Крім лідера племені, який забезпечує всім необхідні ресурси для виконання роботи, в племені немає офіційної ієрархії.

Chapters

Spotify впровадив поняття "Chapters" (розділи) для підвищення співпраці між експертами у їхній галузі. Спеціалісти, наприклад, фронтенд-розробники, зустрічаються, щоб обмінюватися інформацією та методами вирішення проблем. Вони збирають зворотний зв'язок один від одного, щоб переконатися, що вони дотримуються інженерних стандартів протягом всього процесу розробки.

Спеціалісти з усіх команд (Squads) в рамках племені (Tribe) утворюють розділ (Chapter). Подібно до сім'ї, розділи інформують інших учасників про свій прогрес під час щоденних стендапів.

Розділи також співпрацюють, щоб допомогти учасникам подолати потенційні перешкоди в їх повсякденній роботі. Відкрита корпоративна культура

та регулярне неформальне обмінювання інформацією забезпечують бізнес-гнучкість.

Guilds

Гільдії - це самоорганізовані об'єднання співробітників, які поділяють спільні інтереси. У них немає конкретних практик або набору правил, які вони дотримуються. Гільдії вітають участь людей з різних племен (Tribes), якщо вони цікавляться певною темою, якою займається гільдія.

На відміну від розділів (Chapters), які належать до конкретного племені, гільдії існують в різних племенах. В гільдії немає офіційного лідера або офіційного процесу. Замість цього, хтось добровільно стає координатором гільдії і допомагає об'єднати людей.

Не всі належать до гільдії, оскільки вона є добровільною. Наприклад, група розробників інтерфейсу - всі, хто займається розробкою інтерфейсу, приєднуються до гільдії. Навіть особи, які зацікавлені в розробці інтерфейсу, але не потребують цього у своїй повсякденній роботі, приєднуються до гільдії, щоб дізнатися більше про цю тему.

Trios

Трійки (Trios) - це пізній додаток до Spotify моделі. Трійки також відомі як TPD (Tribes, Product Owner, Design Lead) трійки, оскільки вони складаються з лідера племені (Tribe Lead), власника продукту (Product Owner) та керівника дизайну (Design Lead).

Вони працюють разом, щоб забезпечити гнучкість між трьома перспективами. Кожне плем'я має свою власну трійку, що допомагає всім зосередитися на поставленій задачі та вирішувати її відповідно до принципів гнучкості.

Alliances

Племена (Tribes) мають можливість спільно працювати над великими проектами під час масштабування компанії. Великі проекти вимагають укладення альянсів між різними трійками.

Три або більше трійки співпрацюють, щоб обговорити стратегії управління проектами та забезпечити вирівнювання між їх відповідними племенами.

Трійки також є переглянутою версією Spotify моделі. Спочатку Spotify мав складнощі з впровадженням масштабованого гнучкого фреймворку у великих проектах. Однак, після впровадження трійок та альянсів, великі проекти стали більш керованими.

Переваги моделі Spotify

Spotify потребував, щоб Сквади працювали швидко, впроваджували програмне забезпечення і робили це з мінімальними перешкодами, коли вони перетворювали свій підхід до масштабування гнучкості. Поступово вдосконалюючи свою модель, вони виявили переваги по ходу розвитку. Компанії, які намагаються впровадити модель Spotify, отримують різноманітні організаційні переваги, якщо роблять це з власної культурної перспективи.

Високий ступінь автономії та узгодженості

Модель Spotify робить сильний акцент на децентралізації прийняття рішень і делегуванні загонів, племен, відділів і гільдій.

Хенрік Кніберг і Андерс Іварссон розробили модель на основі корпоративної культури, зосередженої на автономії та співпраці. Тим самим довіряючи командам і надаючи їм можливість виконувати проекти, використовуючи будь-яку гнучку методологію, яку вони вважають за потрібну.

Структура моделі Spotify дозволяє Squads миттєво приймати рішення щодо своєї роботи. Хоча вони покладаються на гнучких тренерів, щоб узгоджувати їх зі стратегіями організації, Squads впроваджують зміни в код за власним бажанням.

Гнучка розробка продуктів Spotify піднялася на вершину світу потокового аудіо завдяки перевагам цієї децентралізованої культури прийняття рішень. Їм більше не доводилося покладатися на марнування часу перед випуском програмного забезпечення.

Баланс узгодженості та гнучкості

Щоб модель Spotify працювала, команди мають випускати оновлення часто та невеликими кроками. Це надає Squads гнучкість і послідовність, оскільки вони

не працюють над одним гігантським випуском. Великі випуски дорівнюють потенціалу великих невдач. Тож замість цього Spotify реалізував щось під назвою Release Train.

Поїзд Release Train функціонує як звичайний поїзд і «виходить зі станції» раз на тиждень. Це дає командам можливість відправляти готове програмне забезпечення в потяг для постійного вдосконалення.

Spotify використовує ще один геніальний інструмент під назвою Release Toggle, який дозволяє випускати оновлення для тестування, але видаляє оновлення, якщо програмне забезпечення не працює належним чином. Під час тестування Spotify випускає оновлення для невеликого відсотка своїх клієнтів і таким чином мінімізує свій «радіус вибуху», якщо новий код не працює.

Невдачі моделі Spotify

Незважаючи на всі перспективи моделі Spotify, виявляється, що вона не обійшлася без труднощів. У міру розвитку компанії Spotify намагався визначити загальну структуру для спілкування в міжфункціональній команді. Оскільки кожна команда вибирала власну структуру, співпраця між командами постраждала, а продуктивність сповільнилася. Звичайно, Spotify більше не використовує модель Spotify. Отже, що змусило гіганта потокового аудіо відійти від моделі?

Труднощі з великими проектами

Оскільки для функціонування моделі Spotify були потрібні послідовні менші випуски, вони виявили вузол, який виявилося важко розв’язати.Модель Spotify стала складнішою, ніж передбачали її творці, оскільки компанія та її амбіції розвивалися.

Практики, спрямовані на покращення спритності, натомість перешкоджали цьому. Проблема моделі полягає в тому, що вона працювала на основі припущень. Коли Spotify почав розвивати agile, вони припустили, що всі їхні співробітники є міжфункціональними суперзірками, які співпрацюють. Насправді члени команди мали проблеми з внутрішнім спілкуванням і не дійшли згоди щодо методології, яку використовувати для завершення свого проекту.

Зацикленість на автономії задушила міжкомандне спілкування. Комунікаційні структури команд не узгоджувалися, оскільки кожна мала свій власний метод роботи. Як наслідок, чим більший проект, тим більше в ньому команд, що призводить до неякісного спілкування.

Один розмір підходить не всім

Андерс Іварссон, співавтор технічної документації Spotify, сказав: «Мене хвилює, коли люди дивляться на те, що ми робимо, і думають, що це структура, яку вони копіюють і впроваджують. … Зараз ми намагаємося підкреслити, що у нас також є проблеми. Не все так «блищить, усе працює добре, і всі наші загони надзвичайно чудові».

Коли офіційний документ Spotify вперше з'явився у світі, організації за організаціями намагалися скористатися перевагами простої структури, яку описувала модель. На їхню шкоду, вони не зробили це правильно. Методологія, описана в офіційному документі, здавалася такою легкою для дотримання та впровадження. Якщо це спрацювало для Spotify, то це обов'язково спрацює і для них, чи не так?

Культура компанії відіграє велику роль у масштабуванні гнучкості. Організації повинні взяти до уваги свою існуючу структуру, перш ніж прийняти модель Spotify. Багато хто не замислювався про те, як зміни в їхніх традиційних системах масштабування вплинули на компанію. Для деяких це працювало до певного моменту, а потім стало надто складним. Зосередження на автономії виявилося складним для галузей, де для прийняття рішень потрібна певна форма ієрархії.

1.4. Висновки до розділу 1

У цьому розділі було розглянуто основні принципи гнучких технологій проектування програмного забезпечення (Agile) і представлено огляд Agile першого і другого покоління. Загальна мета дослідження полягала у розумінні

принципів та методологій Agile і їх впливу на процес розробки програмного забезпечення.

Аналізуючи основні принципи Agile, було виявлено, що вони базуються на ідеї гнучкості, співпраці, самоорганізації і ітераційності. Agile першого покоління, представлене методологіями, такими як Extreme Programming (XP) і Scrum, ставить акцент на технічну розвагу і робочі практики, що сприяють швидкій реакції на зміни вимог і забезпечують постійну комунікацію в команді. Agile другого покоління, до якого відносяться методології, такі як Kanban і Lean, фокусується на оптимізації потоку роботи, визначенні значущості задач і забезпеченні більшої прозорості та візуалізації процесу розробки.

РОЗДІЛ 2
МЕТОДИ КООРДИНАЦІЇ ПРИ ГНУЧКІЙ РОЗРОБЦІ ВЕЛИКОМАСШТАБНИХ ПРОГРАМНИХ ПРОЕКТІВ.

2.1. Види координації

Координація є необхідним елементом у гнучкій розробці програмного забезпечення. Вона описує організаційні заходи, які дозволяють індивідам працювати разом для досягнення спільної мети:

- У гнучких командах розробників існують різні типи залежностей, що вимагають координації. Ці залежності можуть бути пов'язані зі знаннями, процесами та ресурсами.
- Залежності від знань включають необхідність обміну інформацією та знаннями про вимоги проекту, технічні аспекти, минулі рішення та розподіл завдань. Ці знання необхідні для продовження роботи над проектом.
- Залежності процесу виникають, коли певні завдання повинні бути виконані перед тим, як інші можуть бути розпочаті. Вони включають послідовність дій та бізнес-процесів, які повинні бути дотримані для успішного виконання проекту.

Залежності ресурсів відображають необхідність доступності певних ресурсів, таких як люди, приміщення або технічні компоненти, для продовження проекту.

Управління цими залежностями здійснюється через механізми координації. Прямий контроль і стандартизація роботи, результатів, навичок і норм є основними механізмами, що допомагають управляти залежностями та забезпечувати ефективну координацію у гнучкій розробці.

Початково у дослідженнях координації в організаціях зосереджувалися на статичних механізмах, які діяли у добре передбачуваних середовищах. Динамічні аспекти координації були описані як механізми взаємоналаштування, що базувалися на зворотному зв'язку. Однак, кілька дослідників критикували такий статичний підхід та запропонували динамічне розуміння координації [9]. Наприклад, Ярзабковський та інші (2012) запропонували модель, в якій відсутність координації призводить до створення нових шаблонів координації, які стають стабільними. У зв'язку з гнучкістю робочих процесів, змінами в вимогах та технологіях, розробка програмного забезпечення є сферою, в якій координація є надзвичайно динамічною [7].

Процес координації

Strode (2012) представив модель координації в невеликих, гнучких проектах розробки програмного забезпечення на основі попередньої роботи Еспінози та ін. (2007) (див. рис. 2.1). Цю модель можна використовувати для широкомасштабної координації, зосереджуючись на координації між командами, а не всередині команд.

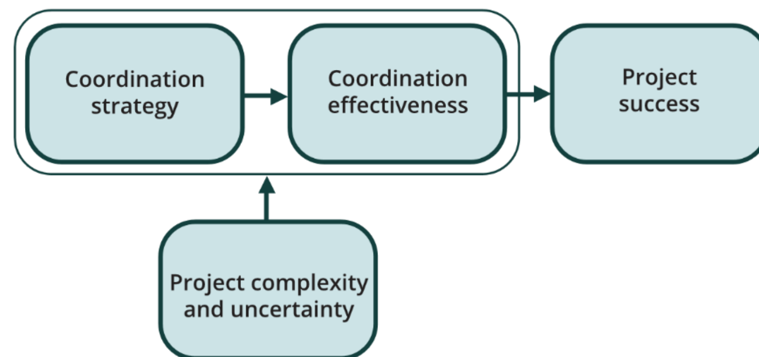

Рис. 2.1. Стратегія координації, ефективність координації та вплив складності та невизначеності проекту (модель від Strode (2012).

Ефективність координації є одним із багатьох факторів, що сприяють загальному успіху проекту. Ефективність визначається як «стан координації, досягнутий у проекті за умови виконання конкретної стратегії координації» (Strode та ін., 2012, с. 1233) і включає неявні та явні компоненти. Імпліцитний компонент базується на літературі з командної роботи та координації. Це передбачає, що учасники проекту розуміють загальну мету проекту, як завдання сприяють її реалізації, загальне уявлення про стан

проекту, завдання, над якими потрібно працювати, завдання, над якими працюють інші, та досвід управління проектом. Явний компонент полягає в тому, що люди та артефакти знаходяться в потрібному місці в потрібний час «і в стані готовності до використання з точки зору кожної особи, залученої до проекту» (Strode та ін., 2012, с. 1233).

Як ми можемо визначити, що координація в команді не працює? Якщо залежності між різними частинами проекту виявляються дуже пізно, то це може призвести до витрат на переробку. Наприклад, коли компоненти з кількох команд інтегруються, може виявитися, що нова функція в одному модулі спричинює несподівані помилки в іншому. Інші проблеми можуть виникнути, коли кілька команд одночасно працюють над однією частиною коду, що призводить до багатьох конфліктів злиття в коді. Це можна було б уникнути, якби одна команда затримала роботу над цією частиною. Крім того, можуть виникати проблеми з узгодженням, коли окремі люди чи групи працюють над завданнями низького пріоритету. Якщо координація працює добре, то це має проявлятися в постійному прогресі у виконанні робочих завдань, якщо немає інших перешкод для прогресу. Однак, якщо проект інвестує занадто багато в координацію, то механізми координації можуть сприйматися як такі, що вимагають занадто багато часу. Якщо члени команди скаржаться, що конкретні зустрічі не є корисними, це може означати занадто великі інвестиції в координацію. Також може статися, що зустрічі погано керуються і не працюють ефективно як механізми координації.

Стратегія координації включає вибір групи механізмів, що регулюють залежності в конкретній ситуації [4] .В нашому використанні терміну "стратегія координації" ми більш точно визначаємо його, ніж Berntzen та ін. (2021), який вважає автономні команди та технічну архітектуру стратегіями. Ми визначаємо механізми координації, відповідно до Ван де Вена (1976), який виділяє три широкі типи механізмів координації:

- Груповий режим – взаємне коригування на основі нової інформації через зворотній зв’язок на зустрічах, які можуть бути як запланованими, так і позаплановими.
- Особистий режим – взаємне коригування шляхом зворотного зв’язку, але між двома людьми на одному рівні організації (особистий, горизонтальний) або на

різних рівнях, наприклад розробник і керівник підпроекту (особистий, вертикальний).

- Безособовий режим – використання « кодифікованих планів дій » , таких як « попередньо встановлені плани, розклади, прогнози, формалізовані правила, політики та процедури, а також стандартизовані інформаційні та комунікаційні системи ».

Традиційний і гнучкий підходи до координації

Гнучкі методи розробки програмного забезпечення були створені з метою ефективного управління змінами та невизначеністю в невеликих командах, знижуючи акцент на традиційних механізмах координації, таких як перспективне планування, обширна документація, конкретні координаційні ролі, контракти та суворе дотримання визначеного процесу [4]. Замість цього, гнучкі методи покладаються на синхронізацію через дії та артефакти, структуру через близькість і взаємозамінність членів команди та межі між командами. Таблиця 1 наводить ключові відмінності між традиційним та гнучким підходами до координації.

Таблиця 2.1.

Відмінності підходів

Традиційний	Гнучкий
Перспективне планування	Синхронізація через дії та артефакти
Явна документація	Спілкування віч-на-віч (близькість)
Ролі	Автономні команди
Попередньо визначені процеси	Межа між командами

Pries-Heje та Pries-Heje (2011) пояснюють, що успіх гнучкого методу Scrum полягає в ефективних та гнучких структурах координації. Гнучкі методи також спрямовані на делегування влади команді та використання детальних короткострокових та загальних довгострокових планів для забезпечення адаптивності до змін . Це впливає на вибір механізмів координації та на те, хто буде відповідати за керування ними. У документі Dingsøyr et al. (2018a, стор. 82) стверджується, що « складність гнучкої

розробки вимагає переосмислення координації, зосереджуючись на таких характеристиках, як усне спілкування, робота в команді, високий рівень взаємозалежності, невизначеність завдань, багато учасників, багато взаємозв'язків між окремими особами та потреби в координації, що змінюються з часом».

Координація гнучкої розробки від малих до великих масштабів

Гнучка команда зазвичай складається з 5-9 людей, які працюють разом в одному місці. Головна мета гнучких методів - надавати швидку цінність в контексті, де вимоги можуть швидко змінюватися, а рефакторинг є недорогим. Для забезпечення ефективного зв'язку і координації в гнучких командах використовують різні канали зв'язку, включаючи зустрічі і комунікацію між членами команди. Scrum - це один з методів гнучкої розробки, який включає щоденне планування ітерації, огляд ітерації та ретроспективу. Координація в гнучкій розробці полегшує співпрацю, оскільки члени команди добре обізнані про роботу інших, загальний прогрес проекту та стан бази коду. Для координації та співпраці в гнучких командах використовують різні артефакти, такі як картки історії та фізична дошка, яка показує статус роботи в поточній ітерації. Дослідження показують, що для координації гнучких команд також використовують текстові описи бізнес-кейсів, контрактів та каркасних макетів на ранніх стадіях розробки.

За мірою збільшення розміру проектів, ймовірно, збільшується кількість залежностей, які потрібно враховувати. Недавні дослідження показали, що потреби команд у координації залежать від характеристик проекту, команди та завдання, а задоволення цих потреб може позитивно впливати на продуктивність команд [10]. У іншому дослідженні було показано, що методи координації всередині та між командами Scrum можуть покращити передбачуваність доставки великих проектів [11]. У глобально розподілених командах розробки програмного забезпечення зазвичай більші розміри команд, ніж у спільно розташованих командах, і розподілені команди проводять більше часу на нарадах (Стрей і Мо, 2020). Крім того, за результатами аналізу 71 проекту за методологією SAFe від компанії Rally, залежності були оголошені лише для близько 10% історій користувачів [6]. Ці залежності були відображені в інструментах для управління життєвим циклом продукту, якими користувалися власники продукту, scrum-

майстри та розробники. У дослідженні підкреслюється, що обсяг неідентифікованих залежностей не відомий [6]. Інше дослідження кількох великомасштабних проектів показало, що члени команди в середньому витрачали 1,1 години на день на заплановані зустрічі та 1,6 години на день на спеціальне спілкування та позапланові зустрічі. Результати дослідження програми Perform, в якій брали участь 12 команд розробників [7], показали, що було необхідно керувати кількома непередбачуваними залежностями, хоча технічна архітектура та організація роботи мали мінімізувати залежності між командами.

Дослідження багатокомандних систем, у яких багато команд працюють разом для вирішення більших завдань , показують, що внутрікомандна координація (всередині команд) необхідна для координації між командами (міжкомандна координація) (Фірт та ін. 2015). Однак для загальної продуктивності команд найважливішою є координація між командами (Marks et al. 2005).

Координація між командами

Координація між командами є важливою темою в літературі про великомасштабну гнучку розробку[2][7]. Опитування щодо координації у великих командах розробників програмного забезпечення показали, що респонденти очікують більш ефективної та ефективнішої комунікації (Бегель та ін. 2009). У систематичному огляді літератури про великомасштабну гнучку розробку були описані проблеми, що виникають в існуючих методах, таких як синхронізація між динамічними та швидкозмінними командами, перевантаження спілкуванням, зовнішні відволікання, зменшення кількості передач між командами в результаті наскрізної розробки та підтримання прозорості у великій кількості команд.

Проблеми координації великої програми розробки з 13 командами великого підприємства були досліджені в роботі Bick et al. (2018). Учасники програми були з трьох різних країн, але відстань та соціокультурні відмінності не стали причиною проблем. Проблемою був той факт, що "прогрес команд розробників був заблокований непередбаченими подіями, головним чином викликаними невизначеною залежністю від інших команд" (Bick et al., 2018) Команди часто не були свідомі діяльності інших команд, а представники команд також не приймали участі у дискусіях щодо

міжкомандних залежностей, оскільки це відбувалося в центральній команді, до складу якої належали головним чином люди з бізнес-компетентністю. Згідно з дослідженням, недостатнє усвідомлення залежності між міжгруповим та командним рівнями пов'язане з помилковим плануванням, визначенням пріоритетів, оцінкою та розподілом роботи між командою. Дослідження запропонувало дві рекомендації: (1) необхідність усвідомлення залежності, але це само по собі недостатньо для ефективної координації, та (2) необхідність узгодження планування всіх етапів, але це не є ефективним для усвідомлення залежності. Практичні рекомендації включають регулярні зустрічі між командами, використання аналогів стандартних арен на рівні команди для координації гнучких методів через спільне планування, перегляд та ретроспективні зустрічі.

Edison et al. (2021) провели огляд і визначили набір практик для різних великомасштабних методів гнучкої розробки. У таблиці 2 представлено практики, пов'язані з координацією між командами, які були згруповані відповідно до режиму координації. Далі надано інформацію про груповий, особистий та безособистісний режими координації, яку ми маємо на сьогоднішній день. Слід зазначити, що в нещодавньому тематичному дослідженні механізмів міжкомандної координації була запропонована альтернативна таксономія, в якій механізми були класифіковані за чотирма характеристиками: технічними, організаційними, фізичними та соціальними[12].

Таблиця 2.2.

Практики

Режим	Механізм
Група	Спеціальне спілкування
	Демонстраційна версія між командами (огляд)
	Огляд середини спринту
	Фізична близькість команд
	Координаційні зустрічі власника продукту, масштабне володіння продуктом
	Scrum scrum зустрічей
	Синхронний спринтерський цикл

Продовження таблиці 2.2

	Тематичні оглядові зустрічі
	Віртуальні стендап зустріч
Особистості	Спеціальне спілкування
	Ітеративна проксі-співпраця
	Ротація членів команди
Безособовий	Розпорядження центральної команди
	Платформа для співпраці
	Спільна мета для спринтів
	Регулярна повна інтеграція програмного забезпечення, обладнання та механіки
	Стратегічна дорожня карта
	Візуалізація (залежності, поставки та портфоліо ІТ-проектів)

Координація групового режиму

Рекомендація, що була надана в попередньому розділі щодо регулярних міжкомандних зустрічей для великомасштабної гібридної програми розвитку, базується на попередніх дослідженнях щодо сутичок як практики координації міжгрупових груп. Дослідження, проведені на великих програмах з більш ніж 20 командами розробників, показали, що цей підхід працює не дуже добре. Обговорювані теми не завжди були достатньо актуальними для учасників [13]. Було рекомендовано зменшити кількість таких зустрічей, щоб забезпечити більшу актуальність обговорюваних питань.

Цей вид зустрічей також розглядався в контексті SAFe, з різними результатами. У двох випадках зустрічі були зосереджені на звітах про статус, менше уваги приділялось рекомендаціям SAFe щодо усунення ризиків. В одному випадку Gustavsson зазначив, що «ніхто не розглядав зустріч як місце для вирішення проблем залежностей», тоді як у третьому випадку зустріч, заснована на залежностях між командами, використовувалася для надання допомоги іншим командам. Неузгодженість між корпоративною культурою та процедурами координації може пояснити невідповідність між намірами SAFe та реальними практиками.

Інші дослідження підтверджують використання більшої кількості зустрічей для координації великих проектів. За допомогою опитувань та тематичних досліджень було

виявлено, що гнучкі великі проекти мають декілька "комітетів спеціалістів", включаючи зустрічі scrum-майстрів у scrum scrums [14]. Дослідження програми Perform показало, що було заплановано 13 координаційних зустрічей, включаючи спільні демонстрації та scrum scrum [7]. Однак, ретроспективні зустрічі відбувалися на рівні команд, але керівництво програми прослуховувало протоколи зустрічей та приймало відповідні заходи.

Особливо цікавою зустріччю в рамках SAFe є нарада з планування збільшення продукту, що є віч-на-віч подією, спрямованою на створення спільної місії та визначення бачення. Зазвичай, горизонт планування становить від 8 до 12 тижнів, який поділяється на 4 ітерації. Густавссон описав цю зустріч як не лише спрямовану на планування та визначення залежностей, а й на "інформування та роз'яснення поточного контексту з точки зору бізнесу, продукту та архітектури". Стандартний порядок денний у SAFe передбачає найбільше місце для презентацій, проте, в трьох досліджених випадках виявлено, що все більше часу витрачається на групові сесії.

Іншим напрямком досліджень є узгодження роботи шляхом створення груп для обміну знаннями між командами, що передбачає заплановані зустрічі, відомі як "спільноти практики". Цю практику застосовують у таких організаціях, як Ericsson і Spotify , з фокусом на темах, таких як гнучкі методи, інфраструктура, бек-енд і зовнішня розробка. Деякі спільноти зосереджені переважно на навчанні чи організаційному розвитку, тоді як інші мають більш пряму увагу на координацію через стандартизацію практик, наприклад, у визначенні стандартів кодування або наданні рекомендацій щодо вибору інструментів. У компанії Ericsson відзначають, що ці спільноти грають вирішальну роль у переході до гнучких методів розробки [13].

Дослідження двох емпіричних випадків зосередилося на груповому режимі координації у великомасштабній гнучкій розробці, аналізуючи зміни у режимах координації з часом. Ці зміни включали перехід від запланованих до позапланових зустрічей та навпаки. Дослідження підкреслило необхідність чутливого підходу керівництва програми до змін у потребах у координації з часом. У своєму дослідженні Едісон та співавтори також вказали на незаплановані зустрічі як на практику, описуючи їх як спеціальні зустрічі та фізичну близькість команд, які були описані в таблиці 2.

Персональний режим

Метод персонального режиму широко використовується в гнучких методах на командному рівні, особливо в парному програмуванні. Але щодо використання індивідуального персонального режиму координації в існуючих дослідженнях великомасштабної гнучкої розробки, дана тема знаходиться на етапі обговорення. Дослідники Бік та ін. (2018) описують координацію на міжкомандному рівні як переважно традиційну, що базується на ролях та ієрархії. Однак, хоча це не повідомляється, особистий режим, ймовірно, використовується для внутрішньогрупової координації всередині команд за допомогою таких практик, як парне програмування. Також можливим є використання особистого режиму координації на вертикальному рівні в організації програми через пряме спілкування між центральними членами команди та ролями в команді, наприклад, власниками продукту. Якщо питання з команди переносяться на міжкомандний рівень, це може бути прикладом використання вертикального особистого режиму координації. У програмі Perform [2][7] горизонтальна координація була забезпечена кількома факторами, такими як спільне розташування у відкритій робочій зоні, що сприяло прямому спілкуванню (спеціальний зв'язок зазначений у таблиці 2), ротація членів команд між проектами та створення нових команд шляхом поділу існуючих команд. Крім того, існувало кілька місць для неформального спілкування, таких як перерви на обід та каву. Парне програмування було широко використовувано, переважно в групах розробників. Представники замовників були доступні для консультацій у відкритій робочій зоні. Дослідження також повідомляє про те, що члени команди зверталися за порадою до команд та організацій, які займалися підпроектами, і багато з них підкреслювали важливу роль відкритої робочої зони. Едісон та ін. (2021) також визначили проксі-співпрацю, яку ми інтерпретуємо як роль між командами, що входить до складу особистого режиму.

Безособовий режим

Бік та ін. повідомили про використання низхідного планування для безособової координації у великомасштабних програмах. Це передбачає створення тем в беклогах продукту, епосів у беклогах випусків та історій користувачів та завдань у беклогах спринтів. Підхід схожий до використаного у програмі Perform [7], де результати

складалися з епосів, які далі ділилися на історії користувачів та завдання на рівні ітерації. У таблиці 2 наведено « загальну мету для спринтів » і « стратегічну дорожню карту ».

Також було знайдено кілька описів процедур, які використовуються у програмі Perform, включаючи архітектурні рекомендації, командні процедури та міжкомандні процедури, такі як scrum of scrums. У порівнянні з гнучкою розробкою, планування виконується більш у письмовій формі, з описами аналізу потреб та описами рішень, які доступні у вікі-програмі. Це може бути розглянуто як центральні командні директиви. Принципи регулярно оновлюються на основі відгуків від ретроспектив або роботи в архітектурі та бізнес-проектах. Проте огляд після завершення проекту показав, що деякі настанови були визначені надто пізно, а деякі не дотримувалися, через що команди вважали, що вони призвели до меншої гнучкості. Огляд також було складно отримати через велику кількість вказівок у вікі. Для асинхронного спілкування між усіма учасниками програми використовувався інструмент обміну миттєвими повідомленнями.

Один з цікавих висновків, що було зроблено в рамках програми Perform, полягає в тому, що план розробки був доступний як на системі відстеження проблем, так і на фізичній дошці поруч з робочими столами команд у відкритій робочій зоні. Один з учасників проекту зазначив, що "для того, щоб отримати загальний огляд статусу [кожної] команди, потрібно було лише дві секунди, і я міг побачити майже всі дошки зі свого місця [в відкритій робочій зоні]. Це дозволяло мені знати, що сталося вчора [в кожній команді]" [2]. Дослідження використання методології SAFe також виявило, що дошка розробки на програмному рівні може бути ефективним інструментом для координації між командами. Ця дошка містить інформацію про функції, їх залежності та віхи для наступних етапів розробки продукту [9]. Дослідження також показало, що існують різні види дошок розробки, як фізичні, так і електронні, з різною частотою оновлень. Кілька інших досліджень також відзначили важливість візуалізації в процесі розробки програмного забезпечення

2.2. Програмні платформи підтримки методів координації.

Глобалізація економіки змусила організації стикатися з безліччю викликів у найрізноманітніших галузях бізнесу для досягнення гнучкості та ефективності, необхідних для збереження конкурентоспроможності та швидкої адаптації до змін на ринку [1]. У результаті цього процесу менеджери проектів вимушені були адаптувати свої процеси з метою покращення ефективності в умовах все більш гнучких середовищ, виникають альтернативи традиційним методологіям управління та розробки проектів. Гнучка розробка почала свій розвиток завдяки фахівцям, пов'язаним з галуззю інженерії програмного забезпечення, з метою мінімізації ризиків, пов'язаних з розробкою програмного забезпечення. Маніфест Гнучкої розробки програмного забезпечення був створений у 2001 році на основі чотирьох основних цінностей: (i) взаємодія та спілкування людей переважають процеси та інструменти; (ii) робоче програмне забезпечення переважає докладну документацію; (iii) співпраця зі замовником переважає переговори про контракт; та (iv) реагування на зміни переважає виконання плану. Порівняно з традиційною моделлю, гнучкі методології працюють з короткими циклами або інтеракціями, в кінці кожного етапу є готовий продукт. В результаті цього вони забезпечують швидкі зміни, які адаптуються до поточного парадигми технологічного розвитку та високої конкурентоспроможності на ринку [2].

Гнучкі практики були дуже успішними на корпоративному ринку, особливо серед невеликих команд та проектів [3]. Однак їх впровадження великими організаційними структурами часто ставиться під сумнів, з урахуванням складнощів управління незалежністю між багатьма командами, ієрархічними пірамідами, що натхнені не-гнучкими моделями, та труднощами, що виникають з культурного спадщини промислової ери. Цю точку зору підтверджують [4], [5], які встановили, що широкомасштабне застосування та інституалізація гнучких практик в компаніях, які розробляють програмне забезпечення, є складним завданням, оскільки впровадження гнучких практик великих команд та проектів передбачає, що принципи гнучкості будуть застосовуватися в усій організації. В результаті впровадження гнучких практик виникають виклики для існуючих практичних структур і породжують питання, пов'язані

з традиційними ролями, відповідальностями та очікуваннями. В [6] стверджується, що необхідно прийняти деякі рішення щодо стратегії впровадження гнучких методологій, зокрема щодо їх розширення на інші організаційні області та реструктуризації бізнес-процесів.

Успіх гнучких методологій в проектах та невеликих командах природно призвів до їх застосування в нових областях, і все більше компаній використовують гнучкі практики в проектах великого масштабу з командами, що налічують сотні професіоналів, часто розподілених географічно [7], [8]. В результаті цього процесу з'явилися фреймворки для управління гнучкими проектами великого масштабу, такі як Large-Scale Scrum (LeSS), Disciplined Agile Delivery (DAD), Scale Agile Framework (SAFe) та інші. Ці фреймворки були розроблені з урахуванням великого розмаїття гнучких практик і охоплюють багатокомандні та великі команди, в яких принципи гнучкості застосовуються в усій організації. Однак вибір найбільш підходящої гнучкої методології для організації є проблематичним завданням. Результати дослідження, проведеного [9], свідчать про те, що фахівці у галузі інженерії вказують на відсутність моделі оцінки для проведення порівняльного аналізу між різними фреймворками гнучкої роботи великого масштабу, що дозволяє керівникам приймати відповідні рішення. Ця ситуація спричиняє недосконалі і нестійкі рішення, а також призводить до зупинки деяких ініціатив щодо впровадження гнучких методологій великого масштабу. У цьому контексті це дослідження має на меті визначити фреймворки гнучкої роботи великого масштабу, які можуть впроваджувати організації, і здійснює порівняльний аналіз між ними, щоб виділити основні характеристики, які є важливими при виборі фреймворку гнучкої роботи великого масштабу.

На діаграмі(рис. 2.3) нижче, можна побачити тенденцію використання основних методологій масштабування гнучкої розробки.

Рис 2.3. Популярність фреймворків

Тож оглянемо декілька з них ближче.

DAD

DAD була розроблена Скоттом Емблером і Марком Лайнсом у 2011 році для заповнення прогалин у процесах Scrum [10]. DAD можна сприймати як гібридний підхід, що розширює Scrum за допомогою інших стратегій, походять з гнучких практик, таких як Extreme Programming (XP), Unified Process (UP), Kanban та інших. Однією з ключових цілей DAD є охоплення всього життєвого циклу доставки, від початкової концепції до поставки та підтримки. Таблиця 2.4 презентує цілі кожної фази DAD. Життєвий цикл DAD складається з трьох фаз, в яких продукт інкрементально розробляється.

Табл. 2.4

Цілі кожної фази DAD

Фаза	Цілі
Початок	Виконання початкових дій ініціалізації проекту. Визначення бачення проекту, зацікавлених сторін, початкових вимог, форм фінансування та ризиків проекту.
Побудова	Виробництво потенційно споживацького рішення на інкрементній основі. Для досягнення цієї мети проект може бути створений за допомогою набору ітерацій або шляхом використання неперервного та ефективного підходу.
Перехід	Забезпечення готовності рішення до впровадження та залучення зацікавленої сторони до цього процесу.

Ролі, запропоновані DAD, також мають гібридний характер, враховуючи розмір команд. Важливо розуміти, що ролі не пов'язані з конкретними особами. Тому в DAD кожен може виконувати одну або кілька ролей, і ці ролі можуть змінюватися з часом. В [10] пояснюється, що ролі в DAD поділяються на основні ролі (наприклад, керівник команди, власник продукту, власник архітектури, член команди та зацікавлена сторона), які існують у всіх командах незалежно від масштабу, і вторинні ролі (наприклад, спеціаліст, інтегратор, експерт у галузі, технічний експерт та незалежний тестувальник), які присутні лише у великих командах та на певний період.

Підхід DAD може масштабуватися з двох перспектив: тактичної гнучкості та стратегічної гнучкості. Тактична гнучкість враховує процес масштабування шляхом контекстного застосування (наприклад, розмір команди, географічна розподіленість, складність проекту тощо) проектних цілей та практик, які найкраще відповідають йому; стратегічна гнучкість є більш амбітною, оскільки пропагу впровадження стратегій легкості та гнучкості в усій організації [11].

LeSS

Методологія LeSS була розроблена для застосування Scrum великими проектами, в багатьох локаціях або в офшорних середовищах. LeSS передбачає

організаційні зміни, які потрібно впровадити, крім тих, які розглядаються в традиційному шаблоні Scrum, і вимагає створення команд зі змішаними функціями шляхом виключення традиційних ролей (наприклад, керівник проекту, лідер команди). У [12] зазначається, що використання гнучких методологій дозволяє створювати команди зі змішаними функціями, в яких спостерігається поєднання технічних та функціональних навичок.

LeSS використовує більшість практик і принципів інших гнучких методологій. Основною метою є завжди бути дуже об'єктивним, простим і прозорим. Крім того, фреймворк фокусується на всьому продукті, спрямованості на клієнта, постійному вдосконаленні, lean-підході та іншому. У LeSS всі спринти закінчуються одночасно, а всі команди Scrum працюють з одним продуктовим беклогом [13]. Планування спринта також складається з двох частин, як у Scrum [13]: (i) перша частина планування визначає мету та елементи продуктового беклогу, які будуть включені в спринт; і (ii) друга частина планування дозволяє командам створити свій план для розробки елементів та досягнення встановлених цілей спринту.

LeSS рекомендується для максимально 8 команд по 8 учасників у кожній команді. Для більших команд існує також LeSS Huge. Концептуально, LeSS Huge можна розглядати як застосування LeSS у середовищі з декількома вкладеними структурами LeSS. У LeSS Huge змінюються ролі, з'являється власник продукту для кожної області, на продуктовому беклогу з'являються області вимог, і виконуються паралельні LeSS спринти для кожної області вимог [14].

Nexus

Фреймворк Nexus був запропонований Кеном Швабером у 2015 році і має основну мету сприяти координації кількох команд Scrum (ідеально від 3 до 9) у розробці єдиного продукту [15]. Nexus пропонує набір правил, ролей і подій, дуже схожих на те, що відбувається у Scrum. Найбільшою відмінністю між цими двома фреймворками є акцент, який ставиться на дослідження залежностей та синхронізацію різних команд Scrum. За таким підходом Nexus намагається

збільшити єдність між командами з можливістю постачання готового приросту в кінці кожного спринту [15].

Команда інтеграції Nexus є ключовим елементом цього фреймворку. Вона відповідає за забезпечення доставки на кінець спринту. Для цього вона вирішує технічні та не-технічні проблеми, які перешкоджають командам Scrum досягти своїх цілей [15]. Команда інтеграції Nexus складається з двох основних ролей, як і у фреймворку Scrum: (i) власник продукту; і (ii) Scrum-майстер. Крім того, з'явилася третя роль - команда інтеграції Nexus. Згідно з [16], її учасники відповідають за визначення архітектури інтеграції додатків і наставництво для команд Scrum. У цьому контексті цій команді слід мати необхідні навички та ресурси для того, щоб дозволити кожній команді Scrum поступово покращувати стан інтегрованого приросту.

Nexus також ґрунтується на принципі прозорості, що дозволяє візуалізувати інтегрований стан приростів всіх артефактів.

SAFe

SAFe - це фреймворк для розробки за методологією Agile, спрямований на роботу в умовах широкомасштабних проектів. Фреймворк поділяється на три рівні: командний, програмний та портфельний [17]. Кожен рівень має свої інтеграційні діяльності та процеси. SAFe включає в себе практики Lean та Agile на всіх трьох рівнях, надаючи стандарти для розміру команд та програм, які можуть бути застосовані в масштабі [17].

В парадигмі SAFe, Agile-команда залишається схожою на типову Scrum-команду з деякими незначними відмінностями. Scrum-команда тепер називається ScrumXP, і все ще є Власником продукту та Scrum-мастером, які можуть бути спільними для 2-3 команд. Команда ScrumXP може бути спеціалізованою, необов'язково крос-функціональною. Крім того, команди ScrumXP повинні працювати в єдності та координації і мати здатність проектувати, будувати та тестувати свою роботу. На рівні програми було створено кілька ролей, зокрема: (i) менеджер продукту; (ii) системний архітектор; (iii) інженер з управління релізами; та (iv) дизайнер користувацького досвіду. Поміж цими індивідуальними ролями,

методологія SAFe пропонує додаткові команди, такі як: (i) команда власників бізнесу; (ii) команда управління релізами; (iii) команда DevOps; та (iv) системна команда.

Розробка функціональності в SAFe виконується синхронно залученням декількох команд у межах Agile Release Train (ART). Крім того, SAFe передбачає, що команди регулярно випускають, кожні чверть, Потенційно Використовуваний Приріст (PSI). ART встановлює, що взаємодії повинні бути структуровані та організовані у відведеному часі з фіксованою датою та якістю, але змінним обсягом [18]. Методологія пропонує організацію ART у чотири двотижневі інтеракції, які слідує тритижневій фазі Hardening, Innovation and Planning (HIP). Таким чином, команди присвячують себе ART, розробляючи та синхронно запускаючи PSI з однаковою чвертьрічною частотою. На рівні портфелю SAFe вводить концепції тем інвестицій та потоку цінності для забезпечення вирівнювання ART на рівні програми. Поток цінності розглядається як тривала послідовність кроків процесу визначення системи, розробки та впровадження, які використовуються для реалізації систем, що надають безперервний потік цінності бізнесу.

Scrum at Scale

Scrum at Scale - це фреймворк, який дозволяє масштабувати фреймворк Scrum до кількох команд Scrum для вирішення складних проблем і одночасної доставки цінності клієнтам. Дослідження, проведені [9], [12], вказують на те, що швидкість команд і обсяг виконаної роботи зменшується зі зростанням кількості команд Scrum, головним чином через необхідність координації залежностей між командами та дублювання роботи. У цьому контексті Scrum at Scale виникає як засіб структуризації та координації роботи декількох команд та досягнення лінійної масштабованості. Можна виділити три принципи для фреймворку Scrum at Scale: (i) легкість; (ii) простота розуміння; і (iii) складність освоєння. Scrum at Scale має два цикли: цикл Scrum Master та цикл Product Owner. Сазерленд (2019) стверджує, що поєднання цих двох циклів дозволяє створити фреймворк, який об'єднує зусилля декількох команд для досягнення однієї спільної мети.

Діяльність координується за допомогою Scrum of Scrums, який складається з Product Owner (тобто MetaScrum) та різних Scrum Master кожної команди.

У циклі Scrum Master виникають нові ролі та діяльності. Scrum of Scrums (SoS) - це команда Scrum, відповідальна за інкрементальну доставку інтегрованого продукту в кінці кожного спринту. Застосовуються ті ж принципи, що й у початковому Scrum, такі як розмір команди, наявність Scrum Master та Product Owner. Також з'являється нова подія під назвою Scaled Daily Scrum (SDS), яка спрямована на виявлення перешкод та виявлення залежностей між командами. Представник кожної команди (наприклад, Scrum Master команди) повинен брати участь у SDS. Ця модель може бути масштабована до інших рівнів координації з появою концепції Scrum of Scrum of Scrums (SoSoS). Ця багаторівнева модель вимагає керівництва, яке може бути забезпечено за допомогою Executive Action Team (EAT). Місією EAT є координація різних команд SoS та SoSoS та взаємодія з іншими частинами організації, такими як вищі менеджери та фінансові менеджери. Ця організаційна модель дозволяє прозоро виявляти перешкоди, які можуть бути легко масштабовані в організації та вирішені протягом одного дня. Крім того, цей фреймворк дозволяє організації органічно зростати на основі своїх потреб і на стійкому темпі.

Custom frameworks

Дослідження, проведене [9] з глобальними компаніями, що використовують масштабні практики Agile, показало, що 7 з 13 компаній (наприклад, Accenture, Dell, Intel) використовують індивідуальні фреймворки, розроблені самими компаніями. Ці моделі інспіровані кількома Agile-методологіями, такими як Scrum, Kanban або TDD. Багато з цих розробок не були створені з нуля, а з'явилися через невдалий досвід з попередніми моделями масштабного Agile.

Однією з причин, яка спонукає компанії розробляти свої власні фреймворки, є створення такого фреймворку, який ефективно працює на практиці, додає вартості та зменшує терміни поставки [9]. Іншою причиною є специфічність кожної компанії, що утрудняє прийняття фреймворків з дуже жорсткими правилами, що може стати перешкодою для їх прийняття організаціями [9].

Фактично, модель, яка працює для однієї організації, може не давати таких же результатів в іншому типі організаційної культури. Крім того, дослідження, проведені [23], виявили, що конкретні фактори організаційної культури корелюють з ефективним використанням Agile-методу. Ще одна проблема, що спонукає до створення індивідуальних фреймворків, - це вимоги щодо виконання процесів відповідно до національних та міжнародних регуляцій [24]. Справді, існують ринки (наприклад, сектор охорони здоров'я), які потребують більш надійних і обширних процесів (наприклад, процесу тестування та забезпечення якості).

2.3. Порівняльне оцінювання програмної платформи.

Усі гнучкі методології мають схожість у своїх процесах, оскільки вони ґрунтуються на спільних гнучких принципах і визначеннях. Цікаво відзначити, що навіть автори гнучких методологій не ставлять жорстких меж для своїх підходів і можуть застосовувати практики з інших гнучких методологій, якщо вони відповідають конкретній ситуації. Наприклад, Кент Бек у своїх майстер-класах з екстремального програмування (XP) нерідко згадує про помилки екстремізму у першому виданні своєї книги про XP. Детальний огляд гнучких методологій показує, що вони вирішують одні й ті ж проблеми, але застосовують різні моделі з реального життя.

Наприклад, Lean Development (LD) розглядає розробку програмного забезпечення як процес виготовлення артизанського виробу. Scrum описує процеси розробки програмного забезпечення за допомогою метафори керуючої інженерії. Екстремальне програмування розглядає розробку програмного забезпечення як соціальну діяльність, де розробники працюють разом. Розробка адаптивних систем (ASD) розглядає проекти розробки програмного забезпечення як процес адаптації до змін.

Таким чином, гнучкі методології використовують різні моделі, але спрямовані на вирішення схожих проблем у розробці програмного забезпечення.

В таблиці 2.5 міститься узагальнений аналіз гнучких методологій. Для ілюстрації методики оцінки було вибрано лише певні гнучкі методології. Подана не повна таксономія методологій, оскільки її метою було виявлення схожостей між різними гнучкими підходами. Це допомагає розробникам, які шукають відповіді на питання щодо вибору методології, розібратись у гнучкому методичному лабіринті. Вибір правильної методології є важливим, оскільки використання непридатної методології може призвести до невдачі проекту. Хоча методологія розробки програмного забезпечення сама по собі не є гарантією успіху або високої якості продукту, неправильний вибір методології може призвести до проблем. Організації, які займаються розробкою програмного забезпечення, не можуть собі дозволити використовувати різні методології для кожного проекту, хоча це було б ідеально. Немає практичного способу мати команду, яка оволоділа багатьма методологіями. Водночас, дотримання однієї методології та сподівання, що вона підходить для всіх проектів, також є наївним. Тому ця методика оцінювання дає організаціям з розробки програмного забезпечення інноваційний підхід до адаптації свого процесу розробки, використовуючи загальні практики різних гнучких методологій. Вона дозволяє використовувати різні методології без необхідності витрачати кошти на їх придбання. Для ретельного аналізу кожної гнучкої методології необхідно докладно розібратися в її основних принципах. Ця методика надає детальну інформацію про те, з якою метою була розроблена конкретна методологія. Це дозволяє виявити основні проблеми, які стимулювали розробку даної методології, а також визначити з якими страхами вона пов'язана. Потенційний користувач методології може вирішити, чи вона відповідає особливостям їхнього проекту, на основі цієї інформації. Виявлення проблем, які методологія призначена вирішувати, є ще одним аспектом цього оцінювання. Деякі методології спрямовані на технічні проблеми розробки (наприклад, екстремальне програмування зосереджується на тестуванні коду), інші – на проблеми управління проектами (наприклад, Scrum акцентується на ефективному комунікації всередині проекту), а деякі гнучкі методології ставлять за мету

вирішення загальних проблем гнучкої філософії (наприклад, Crystal розглядає питання розміру методології, розміру команди та критичності проекту).

Таблиця 2.5

Узагальнений аналіз

	Практики
XP	Процес планування(1), невеликі випуски(2), метафора, тестова розробка(2), пріоритетність сюжетів3, колективна власність(3), парне програмування(3), сорокагодинний робочий тиждень(3), замовлення на місці(4), рефакторинг(5), простий дизайн(5) та безперервна інтеграція(5).
LD	Усуньте відходи(1), мінімізуйте товарні запаси(1), максимізуйте потік(2), витягніть з попиту(2), відповідайте вимогам замовника(2), забороніть місцеву оптимізацію(2), надайте можливості працівникам(3), зробіть це правильно вперше(4), співпрацюйте з постачальниками(4) та створіть культуру постійного вдосконалення(5).
Scrum	Вимоги до захоплення як відставання продукту (1), тридцятиденний спринт без змін під час спринту(2), зустрічі Scrum (3), самоорганізуючих команд (3) та зустрічі планування спринту(4).

Щоб оцінити кожен підхід, необхідно також визначити, які види діяльності та практики переважають у цьому підході. Це допоможе потенційним користувачам визначити практики, які можна застосувати в їхній конкретній ситуації. Ця методика оцінювання показує, що деякі практики з різних підходів насправді дотримуються однакових гнучких принципів, і розробники можуть вирішити, які практики підходять для їхніх випадків. Тому той факт, що на рівні реалізації використовується конкретний гнучкий підхід, стає менш важливим. Іншим аспектом, який розкриває цей метод оцінки, є те, що метод передбачає для розробки проекту. Коли розробники шукають метод, вони зазвичай мають очікування щодо того, що вони хочуть від нього отримати. Тому виникають проблеми, якщо результати використання цього методу нечіткі та важкі для розуміння. Наприклад, якщо розробник очікує, що метод надасть функціональний код, але насправді мета методу полягає в тому, щоб створити набір артефактів дизайну, таких як ті, що виробляються гнучким моделюванням, це може

спричинити певні проблеми.Крім того, оцінка Методи також розкривають предметні знання авторів методу. На цьому етапі аналізу немає необхідності згадувати конкретних авторів, але варто розкрити їхню експертизу відповідно до галузі. Розкриття припущень авторів методології допомагає прояснити фактичні упередження, на яких ґрунтується методологія, часто похідні від досвіду авторів і можливих страхів.

Основні елементи методики оцінювання.

Метафора реальної життєдіяльності

Вказує на основну модель або ідею, яка спонукала до створення методології. Наприклад, спостереження за мурашником і збиранням мурашками може надихнути на застосування подібного процесу у розробці програмного забезпечення.

Методологічний фокус

Зосереджений на конкретних аспектах процесу розробки програмного забезпечення, на які спрямована методологія. Наприклад, гнучке моделювання акцентує увагу на дизайні процесу розробки та розглядає питання моделювання складних проектів з використанням гнучких підходів.

Методика області застосування

Визначає обмеження та область застосування методології. Вона вказує, які завдання методологія може керувати в рамках проекту. Важливо розуміти, що методологія не охоплює всі аспекти проекту, а пропонує рекомендації для управління. Розмір проекту програмного забезпечення є фактором, який враховується при визначенні розміру команди.

Методичний процес

Визначає як методологія відтворює реальну ситуацію або процес. Це може бути відображено в життєвому циклі чи процесі розробки, що створює модель для спілкування, фіксації проблем чи конструкцій, а також для отримання розуміння проблемної області. Важливість цього параметра полягає в тому, що він дозволяє користувачеві отримати реалістичне уявлення про послідовність дій, які виконуються в процесі розробки.

Результати методології

Визначають форму результатів, які можна очікувати від застосування методології. Наприклад, при використанні методології розробки організація може отримати готовий код або документацію. Кожна гнучка методологія може мати свої унікальні результати, тому користувач може вибрати методику, яка найкращим чином задовольняє їхні потреби і вимоги.

Прийоми та інструменти методології

Цей параметр допомагає користувачеві визначити методи та інструменти, які використовуються в методі. Інструменти можуть включати програми, які автоматизують завдання під час розробки, або прості матеріали, такі як дошки та фліпчарти. Використання відповідних інструментів є важливою частиною впровадження цього підходу, і організації зазвичай виділяють значні ресурси на придбання інструментів і навчання співробітників їх використанню. З розвитком технологій стають доступними нові інструменти, що може призвести до додаткових витрат на придбання та навчання. Більшість гнучких методів не прив'язані до конкретних інструментів, і багато гнучких методів використовують програмне забезпечення з відкритим вихідним кодом, щоб зменшити витрати на програмне забезпечення. Кожен метод має свої власні спеціальні методики, які можуть підходити або не підходити для конкретної проблеми. Наприклад, у методології Scrum технікам можуть бути парне програмування та щоденні зустрічі.

Автор методики

Цей параметр визначає предметні знання автора методу та допомагає зрозуміти початкові передумови для розробки методу. Цей параметр необхідно відрізняти від докладної біографії автора або згадки його прізвища. При аналізі практики з різних методологій подібні практики можна класифікувати та об'єднати за допомогою верхніх індексів, які відображають спільні принципи гнучкості. Таблиця 1 класифікує практики за допомогою міток 1, 2, 3, 4 і 5. Кожна мітка представляє певну практику, пов'язану з певним аспектом розробки програмного забезпечення. Після визначення схожих практик розробники можуть

вибрати конкретні практики та адаптувати їх до свого робочого середовища на основі конкретних проблем і пріоритетів проекту:

- «1» - це практика, яка займається такими питаннями планування, як збір вимог. Три підходи тут використовують різну термінологію, але принцип полягає в тому, щоб охопити мінімальні вимоги та почати кодування найпростішим способом.
- «2» - це практики, пов'язані з підвищенням якості при задоволенні мінливих вимог.
- «3» - означає практику, яка сприяє співпраці розробників-фрілансерів, ефективній комунікації, масштабуванню проблем прийняття рішень і динаміці команди.
- «4» - практика, пов'язана зі швидкою доставкою..
- «5» - представляє практики, пов’язані із забезпеченням безперервного вдосконалення атрибутів гнучкого забезпечення якості продукту до розгортання. Визначивши подібні практики, розробники можуть вибрати та адаптувати деякі практики до свого середовища на основі відповідності проекту та клієнта та пріоритетів

Цей аналіз зміщує фокус із конкретних методологій на конкретні практики, доступні для розробки програмного забезпечення. Це дає можливість вибору та адаптації практик відповідно до потреб проекту та клієнта, тим самим забезпечуючи гнучкість у виборі діяльності з розробки.

Аналіз екстримального програмування

Екстремальне програмування (XP) — це легка методологія для малих і середніх команд, які розробляють програмне забезпечення на основі нечітких або швидко мінливих вимог. У другій версії XP Бек розширив визначення XP, включивши розмір команди та програмні обмеження наступним чином:

• XP — це просто: ви просто робите те, що вам потрібно, щоб створити цінність для своїх клієнтів.

• XP адаптується до нечітких і швидко мінливих вимог: досвід показує, що XP можна успішно використовувати навіть для проектів зі стабільними вимогами.

• XP усуває обмеження розробки програмного забезпечення: вона не стосується безпосередньо управління портфелем проектів, фінансування проекту, операцій, маркетингу чи продажів.

• XP працює з командами будь-якого розміру: є емпіричні докази того, що XP масштабується для великих команд.

Процес розробки програмного забезпечення з використанням методології Extreme Programming (XP) розпочинається зі створення історій функціональності замовником. Ці історії є невеликими функціональними одиницями, які можуть бути реалізовані протягом тижня-двох шляхом кодування та тестування. Замовник визначає пріоритети для історій, враховуючи їх вартість та значимість, і програмісти складають кошторис для кожної історії.

Розробка відбувається ітераційно та поетапно. Кожні два тижні команда програмування доставляє виконані робочі історії замовнику. Після цього замовник визначає нові історії для наступного двотижневого циклу робіт. Таким чином, програмне забезпечення зростає поступово та функціональності додаються поетапно залежно від потреб замовника.

Таблиця 2.6 демонструє застосування аналізу до методології XP, що дозволяє систематично аналізувати та оцінювати прогрес розробки, враховуючи виконані роботи та пріоритети замовника.

Таблиця 2.6

Аналіз екстремального програмування

Елементи	Опис
Метафора реального життя	Соціальна діяльність, де розробники сидять разом.
Фокус	Технічні аспекти розробки програмного забезпечення.
Область застосування	Менше десяти розробників в кімнаті. Масштабується до більших команд.
Процес	Фаза 1: Написання розповідей користувачів. Фаза 2: Оцінка зусиль, визначення пріоритетності історії. Фаза 3: Кодування, тестування, тестування інтеграції. Фаза 4: Малий випуск. Фаза 5: оновлений випуск.

Продовження Табл. 2.6

Результати	Робоча система.
Прийоми та інструменти	Парне програмування, рефакторинг, тестова розробка, безперервна інтеграція, метафора системи.
Автор методики (два)	1. Розробник програмного забезпечення . Сильно віруюча у спілкуванні, роздумах та новаторстві. Шаблон для програмного забезпечення. Тест-перша розробка. 2. Розробник програмного забезпечення. Директор з досліджень та розробок. Розробив Wiki. Розроблена основа для інтегрованого тестування.

2.4. Висновки до розділу 2

У цьому розділі було досліджено методи координації, що використовуються при гнучкій розробці великомасштабних програмних проектів. Аналізувалися різні види координації, фреймворки для масштабування гнучких практик, порівнювальне оцінювання та вибір методології. Один з ключових висновків полягає в тому, що для успішної координації великомасштабних проектів необхідно використовувати комбінацію різних видів координації, таких як вертикальна, горизонтальна, групова та міжкомандна координація. Кожен вид координації має свої особливості та застосовується у відповідних ситуаціях.

Фреймворки для масштабування гнучких практик, такі як Scaled Agile Framework (SAFe), Large-Scale Scrum (LeSS) і Nexus, надають структуру та організаційні принципи для ефективного масштабування Agile на рівні всього організації. Вибір підходящого фреймворка залежить від специфіки проекту та потреб команди. Порівнювальне оцінювання різних методологій та фреймворків може допомогти визначити найбільш підходящий варіант для конкретного проекту. Урахування факторів, таких як розмір команди, складність проекту, вимоги замовника і культура організації, є важливими при прийнятті рішення щодо вибору методології. Дослідження показало, що вибір відповідних методів координації та фреймворків для масштабування гнучких практик є критичним для успіху великомасштабних програмних проектів.

РОЗДІЛ 3
СИСТЕМА ПІДТРИМКИ ПРОЦЕСІВ КООРДИНАЦІЇ

Система підтримки процесів координації в проектах гнучкої розробки включає набір інструментів та практик, що сприяють ефективній організації та координації роботи команди, яка працює за гнучкими методологіями, такими як Scrum, XP (екстремальне програмування) чи Kanban. Основна мета такої системи - забезпечити прозорість, співпрацю та згуртованість команди, а також зменшити зайву адміністративну тяготу, дозволяючи розробникам більше уваги приділяти роботі над продуктом.

Основні компоненти системи підтримки процесів координації в проектах гнучкої розробки можуть включати:

- Інструменти спільної роботи: це веб-платформи або програми, які дозволяють команді спільно працювати над проектом, обмінюватися інформацією, спільно редагувати документи, вести обговорення та відстежувати прогрес виконання завдань.
- Дошки завдань: це візуальні інструменти для організації завдань команди. Кожне завдання представлене як картка або елемент, який може бути переміщений по колонках або категоріях (наприклад, "У процесі", "Готово", "На перевірці" тощо), відображаючи його поточний статус.
- Інструменти спілкування: це можуть бути чати, електронна пошта, відеоконференції та інші засоби, що дозволяють команді спілкуватися в режимі реального часу або асинхронно

- Інструменти візуалізації та звітності: це інструменти, які дозволяють відображати графіки, діаграми, звіти та інші візуальні представлення даних про прогрес проекту, роботу команди та інші метрики, що допомагають зрозуміти поточний стан проекту.
- Інструменти контролю версій: це системи, які дозволяють команді відстежувати та керувати змінами у вихідному коді, документах або інших елементах проекту. Вони забезпечують можливість повернення до попередніх версій, об'єднання змін, вирішення конфліктів та спільну роботу над кодом.

Ці інструменти та практики допомагають забезпечити злагоджену та організовану роботу команди, зменшити залежність від індивідуальних членів команди, підвищити комунікацію та співпрацю, а також забезпечити прозорість та контроль над проектом.

3.1. Емпіричні дослідження застосування методів координації при розробці конкретних проектів.

Щоб дослідити питання — як на стратегію міжкомандної координації впливає перехід від широкомасштабних гнучких методів розробки першого покоління до другого — Dingsoyr та ін. розробили кейс із вбудованим поясненням (Runeson and Höst 2009). Систематичний огляд літератури великомасштабних гнучких методів показує, що « цілеспрямовано розроблені поздовжні дослідження прийняття та застосування великомасштабних гнучких методів рідко зустрічаються в існуючій літературі » (Едісон та ін. 2021). Спираючись на раніше створені теорії координації, в основному з науки про менеджмент, і на попередніх дослідженнях міжкомандної координації у великомасштабній гнучкій розробці. Це дослідження позиціонується як позитивістське тематичне дослідження, яке прагне пояснити наслідки змін, спираючись на попередню теорію для визначення набору нових пропозицій. Далі описується план дослідження , процедури збору та аналізу даних.

Це дуже масштабна гнучка програма розробки. Програма передбачає тимчасову організацію, яка відрізняється від постійної організації розробки програмного забезпечення тим, що багато учасників працюватимуть протягом коротшого періоду. Цей випадок було обрано як один із кількох великомасштабних проектів розробки програмного забезпечення, які слідували в рамках дослідницького проекту. Критерієм вибору кейсу було те, що це має бути екстремальний випадок для координації, оскільки він мав велику кількість команд розробників (Dingsøyr та ін. 2014). У дослідженні (Dingsøyr та ін. 2014) взяли участь щонайбільше 200 учасників, близько 130 працювали в 10 командах розробників та в організації програми.

Дослідники провели аналіз міжкомандних стратегій координації між бізнесом і проектами розвитку в програмі. Початковий план передбачав фокусуватися на тому, як програма адаптує свої стратегії координації з часом. Програма була розпланована на три етапи, і план збору даних зосереджувався на документуванні практик і сприйнятті цих практик різними групами для кожного етапу. Однак, внаслідок реорганізації програми, дослідники отримали унікальну можливість вивчити зміни в координації після реорганізації. В результаті були переглянуті процедури збору даних, як описано нижче. Особлива увага була приділена двом етапам програми, на яких одночасно працювали 10 команд розробників: одна використовувала великомасштабний гнучкий метод розробки першого покоління, а інша — великомасштабний гнучкий метод розробки другого покоління.

Учасники дослідження почали стежити за програмою з початку 2017 року, отримали доступ для проведення інтерв'ю з її учасниками, прочитали відповідні документи та спостерігали за зустрічами. Також було отримано серію брифінгів про організацію та хід виконання програми.

Це дослідження стало частиною більш масштабної роботи, на яку дослідники вже отримали схвалення від Норвезького центру дослідницьких даних. Була отримана інформована згода учасників інтерв'ю, і було підтверджено,

що дані, використані у звітах, не можуть бути пов'язані з окремими особами. Крім того, регулярно надсилаються відгуки про висновки учасникам дослідження.

Збір даних

Дослідники, включаючи Дінгсоір і його команду, докладно розробили стратегію збору даних. Хоча програма знаходилася в Осло, більшість дослідників були розташовані в Тронхеймі, віддаленому на 500 км. Тому було вирішено регулярно відвідувати місце подій, під час яких три-чотири дослідники брали участь у зборі даних та наступних обговореннях. Крім того, кандидат доктора філософії частково долучився до збору даних і надав більш глибоке розуміння контексту, вивчаючи зміни в центральному IT-відділі організації справи (Vestues 2021). Обговорення після збору даних мало вирішальне значення для формування загального розуміння програми та проблем координації серед дослідницької групи.

Збір даних проводився за допомогою індивідуальних інтерв'ю, групових інтерв'ю, спостережень та збору документів. Дослідники також зустрічалися з керівництвом програми, щоб отримати розуміння її організації. Після зустрічей і спостережень були створені полеві записи.

Дінгсоір та його колеги опитали осіб у різних посадах, щоб зрозуміти проблеми та методи координації, як показано в таблиці 3.1. Основна увага була зосереджена на практиках розробки програмного забезпечення, і більшість інформантів мали ролі, пов'язані з розробкою. Однак, було також опитано кілька осіб на інших посадах, щоб зрозуміти організацію програми. Посібники для інтерв'ю були підготовлені на основі попереднього дослідження (Dingsøyr та ін. 2018b; Dingsøyr та ін. 2018c). Ці посібники зосереджувалися на проблемах координації та практиках, а також на контрасті між роботою над релізами. Запитання були в основному відкритими і сформульованими мовою, зрозумілою для респондентів, наприклад, "Як ви залежите від інших команд? З якими проблемами стикаєтеся?" і "Як ви управляєте залежностями?" Під час останнього раунду співбесід були внесені незначні зміни, щоб зосередитися на ефектах

реорганізації роботи, яку вони називають переходом від великомасштабних гнучких методів розробки першого покоління до другого покоління.

Табл. 3.1

Ролі опитаних після інтерв'ю та фази програми

Фаза	Ролі інтерв'ю
Перший етап – 1 тур	Архітектори додатків (2), відповідальний за будівництво, розробник (2), функціональні архітектори (2), відповідальний за функціональність, scrum master (2), старший архітектор рішень,менеджер рішень,тестувальник
Кінець першого / початок другого – раунд 2	Менеджер з роботи з клієнтами, центральний ІТ, розробник, функціональний архітектор, власники продукту (2), власник програми, scrum майстри (3), тестувальники (2)
Другий етап – 3 тур	Архітектор додатків, центральний ІТ (2), розробник (2), менеджер програми, розгортання менеджера проекту, власник продукту (3), розробка менеджера проекту, архітектор рішення, автоматизація тестування, тестувальник

За два дні вони відвідали справу тричі. Їх було троє-чотири дослідники, які паралельно проводили напівструктуровані інтерв'ю, за якими слідувала сесія зворотного зв'язку з нашою інтерпретацією сказаного. Під час візитів перші інтерв'ю проводилися парою дослідників, щоб забезпечити послідовність у використанні посібника для інтерв'ю. Пізніше інтерв'ю проводив один дослідник. Інтерв'ю тривало від 24 до 120 хвилин, зазвичай близько 30 хвилин. Вони були записані та переписані для аналізу. Загалом ми опитали 39 інформантів — 13 у грудні 2017 року, 12 у січні 2019 року та 13 у листопаді 2019 року. Ми провели ще одне інтерв'ю у січні 2020 року (див. ролі учасників у таблиці 3). Як описано в розділі обмежень, дослідники не могли опитати учасників з усіх команд під час усіх візитів, але вони завжди опитували людей, які займаються розробкою чи тестуванням, розробкою вимог, архітектурою та управлінням проектами чи програмами. Загалом матеріал інтерв'ю містив 456 сторінок тексту.

Вони також запросили ключових людей з програми на семінар у жовтні 2020 року, під час якого ми встановили графік і обговорили, що спрацювало добре, а що можна покращити. Цей семінар призвів до окремої статті про ключове навчання в процесі трансформації, написаної спільно з практиками з цього випадку (Dingsøyr та ін. 2022). Крім того, команда провела групові інтерв'ю, щоб обговорити координацію та процес розробки вимог. Групове інтерв'ю щодо координації включало керівника проекту та власника продукту з NAV та керівника проекту, допоміжного керівника проекту та конструкцію, відповідальну за проект розробки з Sopra Steria. Це двогодинне інтерв'ю було записано та переписано в 42-сторінковий документ.

Під час переговорів про доступ до справи вчені уникали збору даних у періоди, близькі до публікації. Отже, перший раунд інтерв'ю проводився у відносно спокійний період і міг характеризуватися нейтральним настроєм серед суб'єктів. Другий раунд був проведений після того, як початковий шок від реорганізації врегулювався, який характеризувався сумішшю розчарування та оптимізму. Третій тур був завершений після закінчення програми. Один із дослідників писав: « Я ніколи не брав інтерв'ю у людей, які однаково настільки задоволені своїм становищем!» (Польові нотатки, інтерв'ю раунд 3).

Перший етап

Перший етап включав два випуски. Базовим випуском була цифрова система обробки заявок, яка автоматично обробляла заявки на одноразові виплати. Випуск поселенця розширив систему обробки заяв, включивши всі види батьківських виплат та інтеграцію з системами оплати праці роботодавців . Цей етап мав на меті розробку повної системи прийняття рішень, адаптованої до вимог закону щодо розрахунків.

На цьому етапі робота була організована за чотирма проектами: бізнес, розробка, тестування та управління змінами (рис. 3.1). Бізнес-проект відповідав за етап аналізу потреб, який проводився у співпраці з проектом розробки, отримавши опис рішення, перш ніж його було призначено групі розробників на етапі будівництва; після розробки слідувала фаза затвердження, організована

тестовим проектом. Ця модель була подібна до тієї, що використовувалася в програмі Perform (Dingsøyr та ін. 2018b). Тоді програма може в певний час перебувати на етапі виробництва одного випуску, перебуваючи на етапі будівництва другого випуску та проводячи аналіз потреб для третього (рис. 3.3).

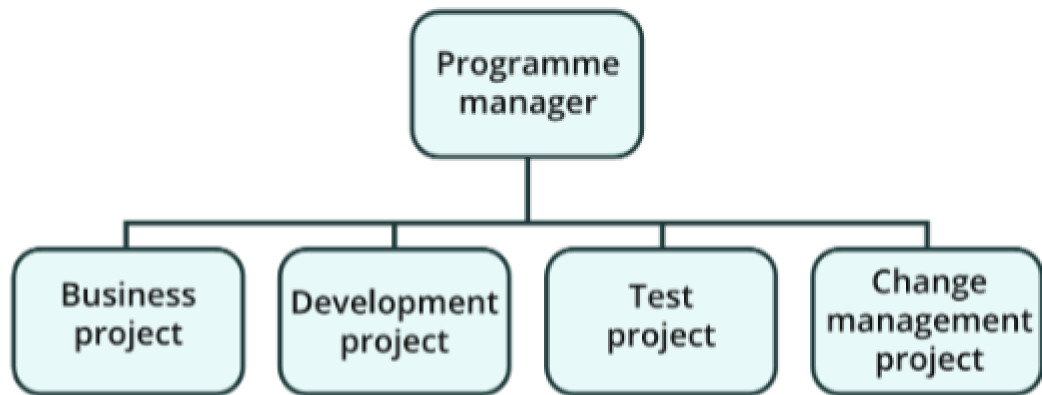

Рис. 3.2 Організація програми з чотирма основними проектами

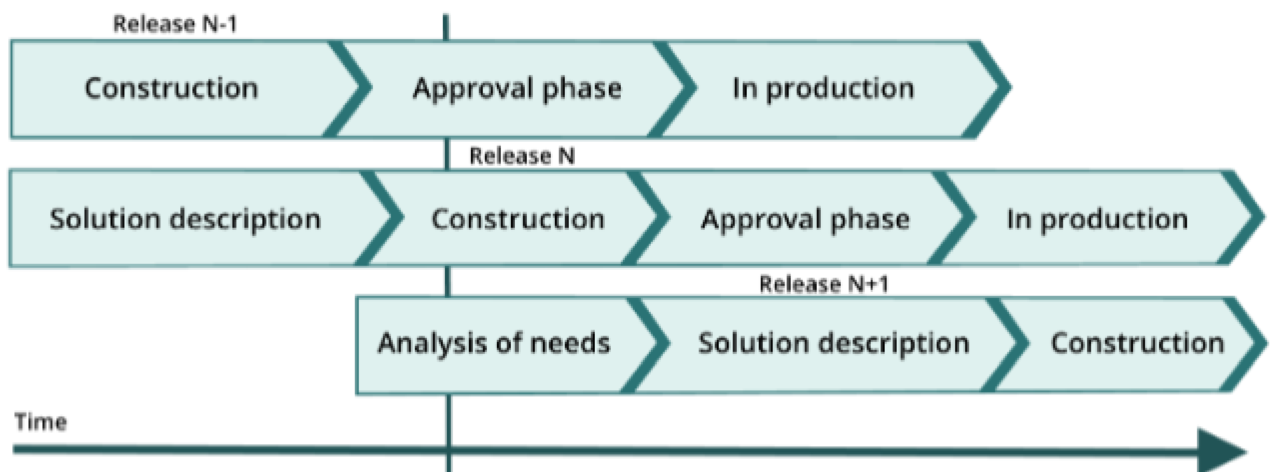

Рис. 3.3 Фази розвитку

Проект управління змінами запровадив нові рішення для основних груп користувачів, кінцевих користувачів, які шукають батьківських пільг, і кейсівців NAV. Групи розробників працювали тритижневими ітераціями з чотирма ролями, описаними в таблиці 4 . Бізнес-проект і групи розробників були розташовані в різних частинах робочої зони, а функціональні архітектори були розташовані разом з бізнес-проектом, але вони підготували описи рішень історій користувачів

для команд розробників. Вони були зроблені в програмі wiki. На рівні програми було 16 ролей. Починаючи з другої поставки, програма створила пілотний тест для вивчення великомасштабних гнучких методів розробки другого покоління в крос-функціональному автономна команда. Було сформовано комітет, щоб оцінити, чи повинна вся програма змінити модель надання.

Координація на першому етапі програми характеризувалася ланцюжком створення вартості з формальними передачами між етапами (рис. 3.2). NAV використовувала консультаційну компанію « Альфа » для допомоги у створенні описів рішень. NAV і консультанти з « Альфи » координували внутрішню роботу, щоб визначити пріоритети та узгодити вимоги по всьому ланцюжку створення вартості (CI1 на рис. 6). Потім описи рішень були передані групі консультантів із проекту розробки, які обробили їх у історії користувачів; вони мали бути схвалені NAV, перш ніж їх можна було передати групам розробників (CE2). Команди розробників мали внутрішню координацію (CI2), щоб розробити необхідний код на етапі будівництва, перш ніж передати результати назад NAV для тестування та затвердження. Якщо описи рішень стосуються зовнішніх систем, NAV або консультанти з « Альфи » ініціюють контакт із зовнішніми партнерами, щоб уточнити, як можна виконати процес (CE1). Коли історії користувачів передаються на рівень команди, команда повинна буде ініціювати новий контакт із зовнішніми партнерами, щоб координувати та бронювати необхідні ресурси для розробки зовнішньої системи (CE3).

Внутрішня координація між командами розробників у проекті розробки була добре структурована. Дослідники ідентифікували 18 механізмів координації, як показано в таблиці 6 , з яких дев’ять є механізмами групового режиму, п’ять є особистими і чотири є безособовими. Ітерація починалася з наради з планування, на якій програма збирала всі команди та представляла завдання та залежності для майбутньої ітерації. Потім команди розбивалися для індивідуального командного планування. Залежності з іншими командами здебільшого вирішувалися через скрам-майстра, який зв’язувався зі скрам-майстром команди, яка мала залежність. Після встановлення контакту залучені розробники спілкувалися безпосередньо,

використовували миттєві повідомлення чи пошту або проводили спеціальні зустрічі для усунення залежностей. Команди, які тісно співпрацюють під час ітерації, також можуть бути фізично переміщені одна до одної, щоб полегшити неформальну координацію.

Скрам-майстри проводили щоденний стендап для своєї команди. Стендапи проходили в шаховому порядку, тому можна було відвідати стендап іншої команди , якщо в команди були залежності, які потрібно було обговорити. Скрам-майстри також збиралися двічі або тричі на тиждень для зустрічі зі скрамів. У кожній команді був технічний архітектор, який відвідав форум технічної архітектури. Проект розробки проводив те, що вони називали технічним оглядом для передачі знань про нові технології, і всі розробники могли бути присутніми. Цю зустріч назвали однією з найважливіших для взаємодії між командами. Один учасник заявив:

« Технічний огляд дуже хороший для узгодження технічного розвитку між командами » (хвилини з ретроспективи, присвяченої міжкомандній координації в листопаді 2017 року).

На першому етапі проект розробки розширювався за рахунок додавання нових людей; як тільки команди стали занадто великими, їх розділили та додали нових людей. Це призвело до того, що вони назвали « перемішуванням » , і більшість розробників чергували між кількома командами, приносячи з собою знання предметної області. Проект розробки також мав деякі ролі на додаток до командної структури; вони вважалися важливими координуючими ролями. Відповідальний за будівництво часто згадувався як роль, яка брала участь у частих обговореннях з командами, щоб переконатися, що правильні люди координують роботу команд.

Наприкінці ітерації кожна команда провела ретроспективу та задокументувала результати у вікі. Вони також організували спільну демонстрацію, під час якої кожна команда показала внутрішнім і зовнішнім зацікавленим сторонам, що вона створила під час ітерації, і прагнула узгодити демонстрації команд.

Табл. 3.4

Механізми координації, класифікації, описи та режими координації для внутрішньої координації всередині команди в проекті розробки

Координаційний механізм	Класифікація	Опис	Режим
Демонстраційна зустріч	Зустріч	Участь взяли всі команди	Група, за розкладом
Форум функціональної архітектури	Зустріч	Форум для обговорення залежностей	Група, за розкладом
Планування зустрічей	Зустріч	Офіційна зустріч для початку ітерації. Участь взяли всі команди. Представлені завдання та залежності для майбутньої ітерації.	Група, за розкладом
Ретроспективи	Зустріч	Обов'язковий в кінці кожної ітерації. Усі ретроспективи зібрані в загальну вікі.	Група, за розкладом
Скрам зі скрамів	Зустріч	Потік інформації між командами. Керівник проекту був головною зацікавленою стороною. Деякі дискусії про залежності.	Група, за розкладом
Поетапна зустріч	Зустріч	Стендап в шаховому порядку , щоб інші команди могли відвідувати одна одну	Група, за розкладом
Форум технічної архітектури	Зустріч	Форум для обговорення Архітектурних залежностей	Група, за розкладом
Технічний огляд	Зустріч	Внутрішній форум консалтингової компанії для передачі знань	Група, за розкладом
Спеціальні зустрічі	Зустріч	Використовується для роз'яснення питань	Група, за розкладом
Ротація членів команди	Середовище	Часто використовувався, коли кількість команд зростала для поширення знань	Особовий, горизонтальний
Парне програмування	Зустріч	Використовується для прискорення роботи випускників у проекті	Особовий, горизонтальний
Миттєві повідомлення	Інструмент	В основному Skype і Hipchat	Особовий, горизонтальний

Продовження таблиці 3.4

Розмова один на один	Зустріч	Контакт з іншими командами зазвичай проходив через скрам-майстра.	Особовий, вертикальний
Ключові ролі	Роль	Деякі люди були критично важливі для збереження огляду та координації. Головний архітектор, відповідальний за будівництво.	Особовий, вертикальний
Архітектурні вказівки	Артефакт	Деякі загальні цифри доступні на Confluence	Безособовий
Карта залежності	Артефакт	Представлено через деякий час, щоб дати розробникам уявлення про те, хто вони буде взаємодіяти з під час an ітерація	Безособовий
Фізичне планування	Середовище	Під час роботи над взаємопов’язаними завданнями команди переміщувалися	Безособовий
Біла дошка	Середовище	У всіх команд були свої.	Безособовий

Координація була ключовою проблемою великомасштабної гнучкої розробки програмного забезпечення (Dingsøyr та ін. 2019b ; Едісон та ін. 2021). Цей розвиток характеризується високою невизначеністю щодо того, як слід вирішувати завдання, великою кількістю взаємозалежностей між завданнями та великою кількістю залучених людей — те, що van de Ven et al. (1976) описано як великий розмір одиниці.

Координація вже давно є ключовою темою глобальної інженерії програмного забезпечення. Гербслеб (2007 , стор. 9) дійшов висновку, що для проблем координації нам бракує розуміння компромісів між інструментами, практиками та методами та розуміння того, коли рішення застосовні.

3.2. Концепція ефективності координації

"Ефективність координації" є залежною змінною в моделі, зображеній на рис. 3.1. Певна стратегія координації призводить до певного рівня ефективності координації. Ефективність координації визначається як стан досягнутої координації в проекті при виконанні певної стратегії координації; іншими словами, це результат цієї стратегії координації. У пов'язаній роботі Strode et al. (2011) описують розробку цього поняття та наводять докази з тих самих трьох випадків, що представлені в цій статті, а також з одного іншого випадку розробки програмного забезпечення, що не використовує методологію Agile. Їх поняття ефективності координації коротко узагальнюється тут.[18]

Ефективність координації має явну та неявну компоненту. Явна компонента охоплює об'єкти (особи або артефакти), що беруть участь у проекті. Використовуючи підходи Теорії координації (Malone and Crowston, 1994) та таксономію загальних залежностей та механізмів координації Crowston (2003), проект координується ефективно, коли необхідний об'єкт знаходиться в правильному місці, в правильний час і готовий до використання з точки зору кожної окремої особи, що бере участь у проекті. [18]

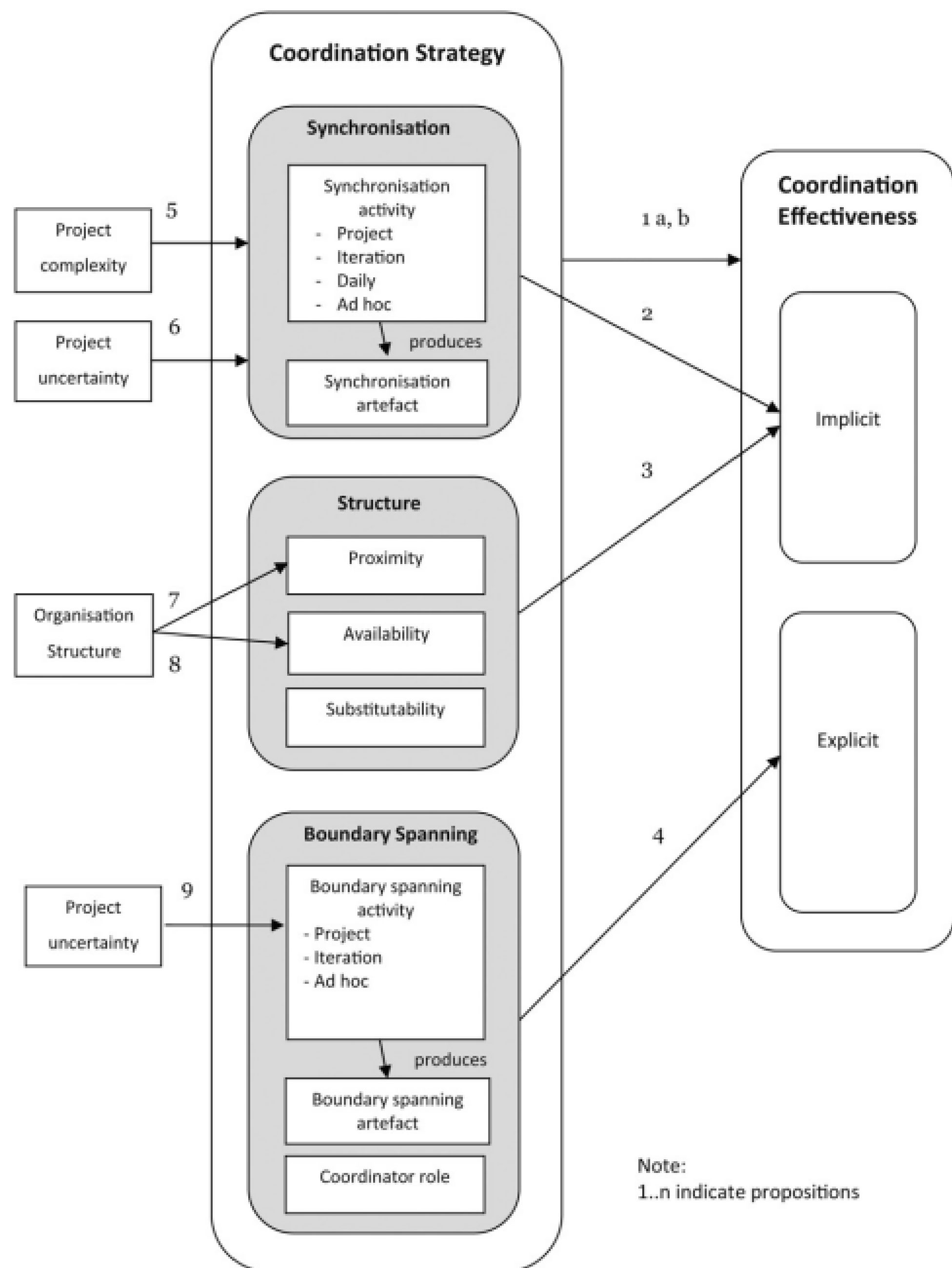

Рис. 3.5. Теорія координації в проектах розробки програмного забезпечення у гнучкому (агільному) стилі.

У контрасті до явної координації, яка акцентується на фізичних об'єктах, неявна координація стосується координації, що відбувається в робочих групах без явного мовлення або передачі повідомлень, як обговорюється в літературі про командну роботу щодо координації. Неявна координація охоплює п'ять компонентів: "Знати чому", "Знати, що відбувається і коли", "Знати, що робити і коли", "Знати, хто що робить" і "Знати, хто знає що".[18]

"Знати чому" охоплює розуміння кожною особою, що працює над проектом, загальної цілі проекту і те, як завдання сприяють досягненню цієї загальної цілі. "Знати, що відбувається і коли" стосується того, що кожна особа, що працює над проектом, має загальне уявлення про стан проекту, тобто завдання, що виконуються в даний момент, і завдання, які мають бути виконані в майбутньому. "Знати, що робити і коли" стосується того, що кожна особа, що працює над проектом, знає, над яким завданням вона повинна працювати і коли вона повинна працювати над цим завданням у порівнянні з усіма іншими завданнями, які мають бути виконані. "Знати, хто що робить" стосується того, що кожна особа в проекті знає, над якими завданнями працюють інші. Нарешті, "Знати, хто знає що" стосується визначення експертизи. Ця компонента підтримується доказами з досліджень, проведених Фараджем та Спроуллом (2000) щодо координації експертизи в проектах розробки програмного забезпечення. [18]

Означення ефективності координації є наступним:

"Ефективність координації є станом координації, в якому у всякій команді розробки програмного забезпечення, яка працює у методології Agile, є всебічне розуміння цілі проекту, пріоритетів проекту, того, що відбувається і коли, того, що кожна особа повинна робити і коли, хто що робить і як робота кожної особи вписується в роботу інших членів команди. Крім того, кожен об'єкт (річ або ресурс), необхідний для досягнення цілі проекту, знаходиться у правильному місці або розташуванні в правильний час і в готовому до використання стані з точки зору кожної осои, що бере участь у проекті." (Строуд та ін., 2011, с. 10). [18]

Рисунок 3.6 ілюструє концептуалізацію ефективності координації.

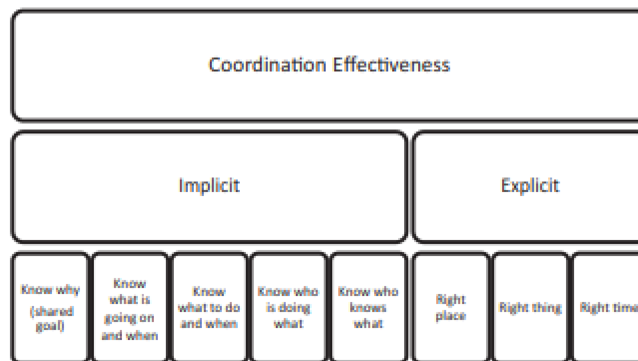

Рис. 3.6. Компоненти ефективності координації згідно з Strode et al. (2011).

Пропозиції, що пов'язують концепти координації.

Модель координації стратегії - ефективності координації, показана на рис. 1, була отримана шляхом якісного аналізу даних кейс-студій. Diane E. Strode, Sid L. Huff, Beverley Hope, Sebastian Link запропонували цю модель не як готовий продукт, а як початкову точку для розуміння ключових конструкцій та взаємозв'язків, що входять до складу координації в проектах розробки Agile. Для уточнення запропонованої моделі вони наводять обґрунтування кожного зв'язку у формі тимчасових пропозицій. Кожна пропозиція ілюструється на рис. 1 і обговорюється нижче. [18]

Основним відношенням, запропонованим в цій теорії, є те, що стратегія координації, яка використовується в Agile-проекті, призводить до ефективності координації. Однак стратегія координації відрізняється залежно від того, як замовник пов'язаний з проектом. Коли замовник або його представник є частиною команди, стратегія координації проекту, яка включає механізми синхронізації і структурної координації, призводить до високого рівня ефективності координації. Синхронізаційні дії повинні відбуватися на кожній частоті - проектній, ітераційній, щоденній та ad hoc. Синхронізаційні артефакти повинні створюватися на кожній частоті, а характер артефакту повинен бути видимим для всієї команди на перший погляд або великою мірою невидимим, але доступним (наприклад, доступним у RallyTM). Артефакт може бути фізичним або віртуальним, тимчасовим або постійним. Механізми структурної координації, а саме,

близькість, доступність та замінюваність, повинні бути на високому рівні. Коли замовник є частиною команди, це послаблює механізми координації для перетину межі. Ці механізми все ще необхідні для взаємодії з зовнішніми сторонами, наприклад, для запиту сервера від відділу IT, але ця форма координації потрібна досить рідко. [18]

Однак, коли замовник або його представник не є частиною команди, стратегія координації проекту стає складнішою. У цій ситуації стратегія координації включає механізми координації синхронізації та структури, а також кілька механізмів координації для перетину межі. Перетин межі тоді включає дії перетину межі на кожній частоті - проектній, ітераційній і ad hoc, а також виробництво артефактів перетину межі на кожній з цих частот. Крім того, може стати необхідним, щоб один або декілька членів команди взяли на себе роль координатора. Коли замовник знаходиться поза командою, механізми координації перетину межі збільшуються за частотою, оскільки саме вони є засобом постійного постачання інформації в проект. Ця інформація надходить від замовників і є необхідною для створення беклогу продукту та надання вимог та деталей функціоналу. Інформація також може надходити через неформальний (ad hoc) зворотний зв'язок від замовника щодо якості робочого програмного забезпечення або більш формально після проходження тестування користувачем. Це призводить до першої пропозиції, яка складається з двох частин.

Пропозиція 1а

Стратегія координації, що включає механізми синхронізації і структурної координації, покращує ефективність координації проекту, коли замовник включений до команди проекту. Для цього потрібні синхронізаційні дії та відповідні артефакти на всіх частотах - проектній, ітераційній, щоденній та ad hoc. [18]

Пропозиція 1б

Стратегія координації, що включає механізми синхронізації, структурної координації та координації перетину межі, покращує ефективність координації проекту, коли замовник є зовнішньою стороною проекту. Для цього потрібні

синхронізаційні дії та відповідні артефакти на всіх частотах - проектній, ітераційній, щоденній та ad hoc. Для перетину межі потрібні дії перетину межі та відповідні артефакти на всіх частотах - проектній, ітераційній та ad hoc. [18]

Пропозиція 1 розглядає ефективність координації як єдине поняття і не робить розрізнення між впливом стратегії координації на неявні та явні компоненти ефективності координації. Багато механізмів координації сприяють як неявній, так і явній координації. Однак, деякі механізми координації сприяють неявній координації, тоді як інші сприяють явній координації. Наступні пропозиції відображають цю тенденцію. [18]

Синхронізаційні заходи та пов'язані з ними артефакти збільшують неявну координацію. Наприклад, всій команді вдається дізнатися, що відбувається і коли, і вони знають, що робити і коли, беручи участь у створенні епічних історій під час першої ітерації та переглядаючи поточний стан дошки, на якій відображаються відкриті історії та стан виконання завдань. Команда може знати, хто що робить, переглядаючи аватар, прикріплений до завдання, запитуючи всю кімнату [Silver, Розробник] або беручи участь у щоденній зустрічі стендап.

Пропозиція 2

Синхронізаційні заходи на всіх рівнях - проектному, ітераційному, щоденному та за потреби, разом з пов'язаними з ними артефактами синхронізації, збільшують ефективність неявної координації. [18]

Коли члени проектної команди перебувають поруч, особливо якщо вони знаходяться в одному приміщенні з сусідніми робочими місцями, вони краще сприймають роботу інших членів команди, спостерігаючи і прослуховуючи їх діяльність. Вони також можуть детальніше ознайомитися з тим, як їх власна задача поєднується зі задачами інших людей; з іншими словами, вони знають, чому вони виконують свою задачу. Крім того, вони дізнаються, що відбувається і коли, вони знають, хто що робить, і вони усвідомлюють, хто знає що.

Коли член проектної команди постійно й без перешкод доступний, інші члени команди можуть звертатися до нього з консультаціями, коли це потрібно і

навіть в короткі терміни. Наявність таким чином підвищує усвідомленість членів команди про те, хто що знає в проекті.

Коли члени проектної команди можуть виконувати роботу одне одного через спільні навички, це замінюваність. Замінюваність підвищує усвідомленість про те, хто що знає, оскільки для виконання роботи іншої людини ви розумієте, що вони знають і не знають.

Пропозиція 3

Структурні механізми координації, а саме близькість розташування, висока доступність і висока замінюваність, підвищують ефективність неявної координації. [18]

Метою перетину меж є здобуття ресурсів для проекту. Ресурси можуть бути фізичними (наприклад, сервери) або інформаційними (інформація про вимоги або технічну галузь). Діяльності, такі як зустрічі з постачальниками та клієнтами, артефакти, такі як офіційні запити, і член команди проекту, який приймає роль координатора, сприяють перетину меж і забезпеченню наявності необхідних ресурсів у відповідному місці, в потрібний час, щоб не заважати прогресу проекту ніяким чином.

Пропозиція 4

Високі рівні механізмів координації перетину меж, а саме діяльності перетину меж на всіх частотах - проект, ітерація і за потреби, пов'язані артефакти перетину меж та роль координатора, підвищують ефективність явної координації. [18]

Складність проекту може бути впорядкована шляхом збільшення частоти ітерацій. Проект Storm використовував цю тактику для кращого справляння з некерованими складними сесіями розбиття історій. Коротші ітерації означають менше відкритих історій і, отже, менше завдань для виконання в спринті. Це спрямовує команду на менший піднабір загальних вимог і зменшує складність загального завдання, обмежуючи кількість факторів, які необхідно розглядати одночасно.

Пропозиція 5

За умов високої складності проекту, збільшення частоти ітерацій та ад-хок синхронізаційних дій забезпечує підтримку ефективності координації. Виробництво відповідних артефактів синхронізації повинно бути відповідно налаштоване. [18]

Умова невизначеності проекту переважно проявляється у невизначеності вимог, але також може включати такі речі, як невизначеність щодо інструментів, технік, інфраструктури та інших факторів. У команди Silver виникала висока невизначеність проекту через замовника, який не міг надати деталі вимог або зворотний зв'язок щодо якості робочого програмного забезпечення вчасно. Для того, щоб справитися з цим, команда використовувала тактику відкладання заблокованих історій та їх завдань, відкриття додаткових історій протягом спринту для того, щоб завершити деякі історії до кінця спринту, і переключення на інші історії (зниження пріоритету заблокованих історій та відкриття історій нижчого пріоритету).

Пропозиція 6

За умов високої невизначеності проекту, для підтримки частоти синхронізаційних дій та виробництва відповідних артефактів, зміна пріоритету історій забезпечує підтримку ефективності координації. [18]

Організації можуть обрати різні способи організації своєї проектної роботи. Деякі організації обирають структуру з одиночним проектом, при якій для кожного проекту створюється команда, яка повністю присвячується цьому проекту. У інших випадках, матрична або багатопроектна структура означає, що члени проектної команди працюють одночасно над кількома проектами або працюють над проектом, виконуючи оперативні обов'язки. Компанія "Land", як пояснено в описі випадку, використовувала матричну організаційну структуру, і це вплинуло на близькість та доступність членів команди.

Пропозиція 7.

Організаційна структура з одиночним проектом забезпечує близькість в порівнянні зі структурами багатопроектної або матричної організації. [18]

Пропозиція 8

Організаційна структура з одиночним проектом покращує доступність в порівнянні зі структурами багатопроектної або матричної організації. [18]

У випадку, коли джерело вимог (тобто замовник або кінцевий користувач) відокремлене від команди, а не знаходиться разом з нею та не є її частиною, підвищена невизначеність має трохи інший вплив, ніж описано в Пропозиції 6. Коли невизначеність висока, а замовник не є легкодоступним, можна підтримувати ефективність координації шляхом збільшення механізмів координації, що стосуються перетину меж. В проекті "Storm" це включало налаштування бета-тестерів як основного джерела деталей вимог, а також ініціювання додаткових спонтанних зустрічей з ними за потреби. Крім того, тестувальник у команді виконував роль координатора між інженерами та командою розробки проекту. Отже:

Пропозиція 9

У випадку високої невизначеності проекту, коли замовник не є частиною команди, збільшення механізмів координації, що стосуються перетину меж, допоможе підтримати ефективність координації. Виробництво відповідних артефактів, пов'язаних з перетином меж, повинно бути відповідно адаптовано. [18]

Практичні наслідки

Хоча кожен гнучкий метод спочатку розроблявся як цілісний набір практик, практики часто обирають окремі практики з методу, а не впроваджують повний метод (Conboy та Fitzgerald, 2007). Також зустрічається поєднання практик з двох або більше методів (Fitzgerald та ін., 2006). Як практична рекомендація, коли практики обирають практики, або з одного гнучкого методу, або з кількох гнучких методів, вони можуть використовувати запропоновану модель як допомогу в ідентифікації практик, які треба обрати для забезпечення координації. Модель вказує, що слід обирати практики, які забезпечують синхронізацію на всіх рівнях, зазначених у моделі: проектному, ітераційному, щоденному та за потребою. Артефакти можуть бути створені на всіх цих рівнях і сприяють координації. Крім

того, практики повинні враховувати, чи може їхня організаційна структура забезпечити близькість та доступність для всіх членів команди. Слід підтримувати можливість взаємозамінності замість пригнічення. Крім того, співпраця з зовнішніми сторонами - це проблема, яку не вирішують належним чином гнучкі методи. XP не надає вказівок з цього питання, хоча у Scrum особа, яка виконує роль Scrum Master, працює над вирішенням проблем, які уповільнюють прогрес проекту, до яких можуть входити дії зі співпраці зі зовнішніми сторонами. З точки зору координації, ця роль має важливе значення для керування взаємодіями та ведення ефективних переговорів з зовнішніми сторонами, звільняючи інших членів команди проекту для фокусу на внутрішніх питаннях проекту. [18]

Ще один висновок з практичними наслідками полягає в тому, що тривалість ітерацій надає можливість керувати ефективним рівнем координації. Іншими словами, за умов складності зменшення тривалості ітерацій збільшує частоту синхронізаційних активностей, але зберігає ефективну координацію. З цього дослідження менш зрозуміло, як проекти можуть ефективно управляти невизначеністю вимог, коли зовнішні сторони (клієнти або кінцеві користувачі) не бажають або не можуть тісно співпрацювати з командою. Ця проблема може бути до певної міри зменшена, підтримуючи формальний зв'язок з цими сторонами принаймні один раз протягом кожної ітерації. [18]

Розподілена розробка програмного забезпечення є поширеним явищем у деяких частинах світу, і зусилля з використання гнучкого розроблення програмного забезпечення в цьому контексті не такі прості, як у випадку з розташуванням на одному місці. Ця модель надає точку виходу для розуміння координації в проектах з розташуванням на одному місці і може бути розширена на випадок глобального розподіленого розроблення програмного забезпечення.

Оперативна реалізація конструкції ефективності координації надасть цінний показник ефективності координації у гнучких (і, можливо, інших) проектах. Такий показник можна використовувати для оцінки ефективності координації на різних етапах проекту, надаючи профіль координації проекту і ранній сигнал попередження, коли починаються проблеми з координацією. Це

допоможе організаціям зрозуміти, як їх проекти працюють щодо координації протягом життєвого циклу проекту. Вона також може допомогти організаціям виявити та вирішити свої проблеми з координацією вчасно і покращити ймовірність успішного завершення гнучкого проекту. [18]

3.3. Вибір платформи для координації

Світ розробки програмного забезпечення постійно змінюється та розвивається. Постійно створюються та вдосконалюються нові технології, методології та інструменти.

Гнучка розробка є методологією розробки програмного забезпечення, що характеризується короткими циклами розробки, називаними ітераціями, які часто мають однакову тривалість. Вона наголошує на співпраці з клієнтом, постійному отриманні зворотного зв'язку та швидкій доставці працездатного програмного забезпечення.

Протягом останніх років спостерігається підвищений інтерес до гнучких методів. Багато організацій впроваджують якусь форму гнучкої методології для своїх проектів з розробки програмного забезпечення. В результаті було створено кілька різних гнучких фреймворків, які вказують, як реалізувати гнучкі методи в масштабах організації.

Але як ви визначите, який з них підходить для вашого проекту? Вибір відповідного процесу або фреймворку для вашої команди може бути складним завданням.

Оскільки порівняння платформ виконується по багатьом критеріям, то для вибору найкращої пропонується використати метод аналізу ієрархії Сааті. Для цього розробимо програму для цього методу. Код програми викладений в Додатку А.

Для прикладу, порівняємо дві платформи Spotify та Scrum of Scrums та дві кастомні платформи. Сформуємо таблицю 3.3 критеріїв на основі досліджень Philipp Diebold, Anna Schmitt, Sven Theobald.

Таблиця 3.3

	Scrum of Scrums	Spotify
Опис	Важливий механізм маштабування який достатній для невеликих організацій	Метод, котрий започаткувала велика компанія Spotify
Рік розробки	1996	2011/2012
Популярність	Висока	Середня
Основа гнучкого методу	Scrum	Scrum/Kanban
Повнота охоплення…		
Портфоліо	Низька	Низька
Структура програми	Низька	Низька
Внутр.-командна координація	Середня	Висока
Рівень команди	Середня	Висока
Розмір цільового проекту	Малий-Середній	Середній-Великий
Використання автоматизації	Висока	Середня
Контроль: центральний, розподілений	Розподілений	Розподілений
Рівень гнучкості	Висока	Високий
Фокус	-команда/структура -внутрішньо-командні залежності	-команда/структура -міжкомандна взаємодія -фокус на культурі

Заповнимо матрицю порівнянь по декільком критеріям, а саме розмір цільового проекту та повнота охоплення внутр.-командна координації и. Де 1 – Scrum of Scrums, 2 – Spotify, 3 – кастомний фреймворк для малих команд, 4 – кастомний фреймворк для великих команд.

Координація на рівні команди у Spotify вища за SoS, таким чином перша показує кращий результат. Якщо порівняти Spotify з двома кастомними фреймворками, то перша має перевагу в координації на рівні команди на фреймоворком номер 4, так як він не оптимізований для роботи в невеликих проектах. По аналогічній системі заповнюємо таблицю 3.4.

Таблиця 3.4

	1	2	3	4
1	1	1/5	3	7
2	5	1	1/3	6
3	1/3	3	1	1/4
4	1/7	1/6	4	1

Аналогічно заповнюємо таблицю 3.5 для оцінювання розміру цільового проекта.

Таблиця 3.5

	1	2	3	4
1	1	3	2	1/4
2	1/3	1	3	9
3	1/2	1/3	1	1/9
4	4	1/9	9	1

Заносимо дані з таблиці в програму і отримуємо такий результат, котрий зображаємо в таблиці 3.6:

Таблиця 3.6

Критерій	1	2	3	4
Координація	0.53	0.62	0.43	0.26
Розмір проекту	0.33	0.56	0.15	0.47

Далі заповнюємо матрицю (таблиця 3.7) попарних порівнянь для критеріїв, після чого використовується метож аналізу ієрархії Сааті.

Таблиця 3.7

Критерій	Координація	Розмір проекту
Координація	1	4
Розмір проекту	1/4	1

Отримуємо результат, заповнюємо таблицю 3.8:

Таблиця 3.8

Платформа	1	2	3	4
Оцінка	0.313	0.432	0.142	0.257

Таким чином виходячи з результатів, можна підвести висновок що платформа 2 (Spotify), має переваги над іншими по зрівняним критеріям. При цьому SoS, не сильно їй програє. Також, платформа 4 відстає майже вполовину. Врешті платформа номер 3, програє усім іншим.

3.4. Висновки до розділу 3

У цьому розділі було розглянуто систему підтримки процесів координації при розробці програмного забезпечення. Досліджувалися емпіричні дослідження застосування методів координації при розробці конкретних проектів, концепція ефективності координації та практичні наслідки використання системи підтримки.

Емпіричні дослідження виявили, що застосування методів координації при розробці конкретних проектів сприяє поліпшенню комунікації та співпраці між учасниками проекту. Виявлено, що певні методи координації, такі як регулярні синхронізаційні зустрічі, дозволяють забезпечити ефективну взаємодію і координацію у команді.

Концепція ефективності координації визначає, що ефективна координація передбачає забезпечення взаємодії та синхронізації між учасниками проекту на різних рівнях: проектному, ітераційному, щоденному та за потреби. Впровадження системи підтримки процесів координації допомагає досягти цієї ефективності шляхом забезпечення структури, інструментів та комунікаційних каналів для спільної роботи команди.

Практичні наслідки використання системи підтримки процесів координації полягають у поліпшенні комунікації, зниженні конфліктів та підвищенні ефективності роботи команди. Ця система дозволяє забезпечити прозорість, взаємодію та спільне розуміння між учасниками проекту, що сприяє успішному виконанню проектів і досягненню поставлених цілей.

ВИСНОВКИ

Під час кваліфікаційної роботи було детально розглянуто основні положення гнучкої розробки, методів координації під час такої розробки, та систему підтримки координації та її покращення.

В цілому, гнучкі технології проектування програмного забезпечення (Agile) є цінним інструментом для розробки програмного забезпечення з урахуванням змінних вимог та швидкого реагування на зміни. Результати дослідження можуть бути корисними для практикуючих спеціалістів у виборі та впровадженні підходів Agile, а також для організацій, які прагнуть поліпшити координацію та успішність своїх проектів з використанням Agile.

Також було детально розглянуто модель Spotify, одноіменної компанії. Було виявлено переваги та недоліки даної методології. Також, на основі досліджень великих компаній, сформовано ряд пропозицій для поліпшення систем координації команд.

Отже, Загальною метою дипломної роботи було розкриття та аналіз гнучких технологій проектування програмного забезпечення, методів координації та систем підтримки для досягнення успішної реалізації великомасштабних програмних проектів. Результати досліджень підтверджують, що використання гнучких підходів та ефективних методів координації сприяє поліпшенню результатів розробки програмного забезпечення. Дані висновки можуть бути використані для подальшого розвитку гнучких методологій та практик в програмній індустрії.

СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ ВИКОРИСТАНИХ ДЖЕРЕЛ

1. On Coordination Mechanisms in Global Software Development [Electronic resource]. – Access mode: https://www.researchgate.net/publication/4274100_On_Coordination_Mechanisms_in_Global_Software_Development(lastaccess:15.05.23) - Title from the screen.
2. A decade of agile methodologies: Towards explaining agile software development [Electronic resource]. – Access mode: https://www.researchgate.net/publication/236211358_A_decade_of_agile_methodologies_Towards_explaining_agile_software_development(lastaccess:25.05.23) - Title from the screen.
3. Strategies for Design Science Research Evaluation [Electronic resource]. – Access mode: https://www.researchgate.net/publication/221408853_Strategies_for_Design_Science_Research_Evaluation(lastaccess:12.05.23) - Title from the screen.
4. A Taxonomy of Dependencies in Agile Softwar onomy of Dependencies in Agile Software Development [Electronic resource]. – Access mode: https://www.researchgate.net/publication/267706181_A_Taxonomy_of_Dependencies_in_Agile_Software_Development(lastaccess:16.05.23) - Title from the screen.
5. The global environmental injustice of fast fashion [Electronic resource]. – Access mode: https://www.researchgate.net/publication/329938899_The_global_environmental_injustice_of_fast_fashion(lastaccess:17.05.23) - Title from the screen.
6. Magdalena Biesialska's research [Electronic resource]. – Access mode: https://www.researchgate.net/scientific-contributions/Magdalena-Biesialska-2160935328(lastaccess:18.05.23) - Title from the screen.
7. Agile Software Development [Electronic resource]. – Access mode - https://agilemanifesto.org/ - Title from the screen.
8. Project Management Body of Knowledge [Electronic resource]. – Access mode- https://www.academia.edu/36463554/A_Guide_to_the_Project_Management_Body_of_Knowledge_PMBOK_Guide(lastaccess:15.05.23) - Title from the screen.
9. Coordination in organizations: An integrative perspective [Electronic resource]. – Access mode: https://psycnet.apa.org/record/2011-23227-010(lastaccess:12.05.23) - Title from the screen.
10. Team-external coordination in large-scale software development projects [Electronic resource] – Access mode: https://onlinelibrary.wiley.com/doi/full/10.1002/smr.2297(lastaccess:11.05.23) - Title from the screen.

11. Aligning codependent Scrum teams to enable fast business value delivery: A governance framework and set of intervention actions [Electronic resource]. – Access mode: https://rinivansolingen.nl/wp-content/uploads/2019/06/2_Aligning-codependent-Scrum-teams.pdf(lastaccess:14.05.23) - Title from the screen.

12. A Taxonomy of Inter-Team Coordination Mechanisms in Large-Scale Agile [Electronic resource]. – Access mode: https://www.researchgate.net/publication/359421393_A_Taxonomy_of_Inter-Team_Coordination_Mechanisms_in_Large-Scale_Agile(lastaccess:15.05.23) - Title from the screen.

13. Teaching students global software engineering skills using distributed Scrum [Electronic resource]. – Access mode: https://www.researchgate.net/publication/261314445_Teaching_students_global_software_engineering_skills_using_distributed_Scrum(lastaccess:16.05.23) - Title from the screen.

14. Agile Methods on Large Projects in Large Organizations [Electronic resource]. – Access mode: https://journals.sagepub.com/doi/10.1177/875697281704800301(lastaccess:17.05.23) - Title from the screen.

15. Agile manifesto [Electronic resource]. – Access mode: https://agilemanifesto.org/ - Title from the screen.

16. Coordination Challenges in Large-Scale Software Development: A Case Study of Planning Misalignment in Hybrid Settings [Electronic resource]. – Access mode: https://www.researchgate.net/publication/318668193_Coordination_Challenges_in_Large-Scale_Software_Development_A_Case_Study_of_Planning_Misalignment_in_Hybrid_Settings(lastaccess:18.05.23) - Title from the screen.

17. Coordination in co-located agile software development projects [Electronic resource]. – Access mode: https://www.researchgate.net/publication/256991816_Coordination_in_co-located_agile_software_development_projects(lastaccess:17.05.23) - Title from the screen.

18. Agile Processes in Software Engineering and Extreme Programming – Workshops [Electronic resource]. – Access mode: https://link.springer.com/book/10.1007/978-3-030-88583-0(lastaccess:15.05.23) - Title from the screen.

19. A theory of coordination in agile software development projects [Electronic resource]. – Access mode - https://www.academia.edu/4716730/A_theory_of_coordination_in_agile_software_development_projects(lastaccess:14.05.23) - Title from the screen.

20. Coordination in Large-Scale Agile Software Development [Electronic resource]. – Access mode:

https://www.researchgate.net/publication/335508001_Coordination_in_Large-Scale_Agile_Software_Development(lastaccess09.05.23) - Title from the screen.

21. Empirical Software Engineering [Electronic resource]. – Access mode: https://www.sciencedirect.com/science/article/abs/pii/S0950584908000256(lastaccess: 07.05.23) - Title from the screen.

22. A Coordination Perspective on Agile Software Development [Electronic resource]. – Access mode: https://www.researchgate.net/publication/263236581_A_Coordination_Perspective_on_Agile_Software_Development(lastaccess:04.05.23) - Title from the screen.

23. Coordination In A Fully Remote Agile Software Development: A Conceptual Study [Electronic resource]. – Access mode: https://www.researchgate.net/publication/353008985_Coordination_In_A_Fully_Remote_Agile_Software_Development_A_Conceptual_Study(lastaccess:03.05.23) - Title from the screen.

ДОДАТКИ

Додаток А

```
#include <math.h>
#include <vector>
#include <complex>
#include <iostream>
#include <fstream>

//#define BG_LIGHT
#ifndef BG_LIGHT
    #define BLACK     "\033[1;30m"
    #define RED       "\033[1;31m"
    #define GREEN     "\033[1;32m"
    #define YELLOW    "\033[1;33m"
    #define BLUE      "\033[1;34m"
    #define PURPLE    "\033[1;35m"
    #define CYAN      "\033[1;36m"
    #define GREY      "\033[1;37m"
#else
    ...
#endif

#define DEFAULT_COLOR "\033[0;m"

#define FORMAT(color_delimiter,color_text, color_comment,delimiter, comment, x,y)   \
    do { printf("%s%s%s%s%s%s%s  %s%s%s\n",\
            color_delimiter,delimiter,\
            color_text, comment,\
            color_delimiter,delimiter,\
            color_comment,\
            x,y,\
            DEFAULT_COLOR);\
        } while (0)

#define OK(x,y)     do { FORMAT(BLUE,GREEN,DEFAULT_COLOR,"|","ok",x,y);} while (0)
#define NOK(x,y)    do { FORMAT(BLUE,RED,DEFAULT_COLOR,"|","nok",x,y);} while (0)
```

```cpp
vector<vector<double> > Inverse(vector<vector<double> > a, bool &judge) {
    FUNCTION("Inverse");
    vector<vector<double> > b(a);
    if (b.size() != b[0].size()) {
        judge = false;
        return b;
    } else {
        double temp = 1;
        for (int k = 0; k < (int) b.size(); ++k) {
            if (k < ((int) b.size() - 1)) {
                int tk = k;
                double tem = fabs(b[k][k]);
                for (int i = k; i < (int) b.size(); ++i) {
                    if (fabs(b[i][k]) > tem) {
                        tk = i;
                        tem = fabs(b[i][k]);
                    }
                }
                if (tk != k) {
                    swap(b[tk], b[k]);
                }
                if (b[k][k] == 0) {
                    temp = 0;
                } else {
                    for (int i = k + 1; i < (int) b.size(); ++i) {
                        tem = b[i][k];
                        for (int j = k; j < (int) b[i].size(); ++j) {
                            b[i][j] -= (b[k][j] * tem / b[k][k]);
                        }
                    }
                }
            }
        }
        if (temp != 0) {
            for (int i = 0; i < (int) b.size(); ++i) {
                temp = temp * b[i][i];
            }
        }
        if (temp == 0) {
            judge = false;
```

```
103             return a;
104         } else {
105             vector<vector<double> > I;
106             for (int i = 0; i < (int) a.size(); ++i) {
107                 vector<double> z(a.size(), 0);
108                 z[i] = 1;
109                 I.push_back(z);
110             }
111             for (int k = 0; k < (int) a.size(); ++k) {
112                 int tk = k;
113                 double tem = fabs(a[k][k]);
114                 for (int i = k; i < (int) a.size(); ++i) {
115                     if (fabs(a[i][k]) > tem) {
116                         tk = i;
117                         tem = fabs(a[i][k]);
118                     }
119                 }
120                 if (tk != k) {
121                     swap( &: a[tk], &: a[k]);
122                     swap( &: I[tk], &: I[k]);
123                 }
124                 tem = a[k][k];
125                 for (int i = 0; i < (int) a[k].size(); ++i) {
126                     a[k][i] /= tem;
127                     I[k][i] /= tem;
128                 }
129                 for (int i = 0; i < (int) a.size(); ++i) {
130                     if (i != k) {
131                         double temp = a[i][k];
132                         for (int j = 0; j < (int) a[i].size(); ++j) {
133                             a[i][j] -= (a[k][j] * temp);
134                             I[i][j] -= (I[k][j] * temp);
135                         }
136                     }
137                 }
138             }
139             judge = true;
140             return I;
141         }
```

```
144
145 vector<vector<double> > operator -(vector<vector<double> > a,vector<vector<double> > b) {
146     OPERATOR( y: "-");
147     bool dd = true;
148     if (a.size() != b.size()) {
149         dd = false;
150     } else {
151         for (int i = 0; i < (int) a.size(); ++i) {
152             if (a[i].size() != b[i].size()) {
153                 dd = false;
154             } else {
155                 for (int j = 0; j < (int) a.size(); ++j) {
156                     a[i][j] -= b[i][j];
157                 }
158             }
159         }
160     }
161     vector<vector<double> > ss;
162     if (dd == false) {
163         return ss;
164     } else {
165         ss = a;
166         return ss;
167     }
168 }
169
170 vector<vector<double> > operator *(double a, vector<vector<double> > b) {
171     OPERATOR( y: "*");
172     for (int i = 0; i < (int) b.size(); ++i)
173         for (int j = 0; j < (int) b[i].size(); ++j) {
174             b[i][j] *= a;
175         }
176     return b;
177 }
178
179 vector<double> operator *(vector<vector<double> > a, vector<double> b) {
180     OPERATOR( y: "*");
```

```
181         vector<double> c;
182         for (int i = 0; i < (int) a.size(); ++i) {
183             double s = 0;
184             for (int j = 0; j < (int) b.size(); ++j) {
185                 s += a[i][j] * b[j];
186             }
187             c.push_back(s);
188         }
189         return c;
190     }
191 
192     vector<vector<double> > operator *(vector<vector<double> > a,
193             vector<vector<double> > b) {
194         OPERATOR( y: "*");
195         if (a[0].size() != b.size()) {
196             vector<vector<double> > ss;
197             return ss;
198         } else {
199             vector<vector<double> > ss(a.size());
200             for (int i = 0; i < (int) ss.size(); ++i) {
201                 for (int j = 0; j < (int) b[0].size(); ++j) {
202                     double temp = 0;
203                     for (int k = 0; k < (int) b.size(); ++k) {
204                         temp += (a[i][k] * b[k][j]);
205                     }
206                     ss[i].push_back(temp);
207                 }
208             }
209             return ss;
210         }
211     }
212 
213     vector<vector<double> > operator *(vector<double> a, vector<double> b) {
214         OPERATOR( y: "*");
215         vector<vector<double> > result(a.size());
216         for (int i = 0; i < (int) result.size(); ++i) {
217             for (int j = 0; j < (int) b.size(); ++j) {
218                 result[i].push_back(a[i] * b[j]);
219             }
```

```
224  vector<double> operator /(vector<double> a, double b) {
225      OPERATOR( y: "/");
226      for (int i = 0; i < (int) a.size(); ++i) {
227          a[i] /= b;
228      }
229      return a;
230  }
231  
232  int sgn(double x) {
233      FUNCTION( y: "Sgn");
234      if (x > 0) {
235          return 1;
236      } else if (x == 0) {
237          return 0;
238      } else {
239          return -1;
240      }
241  }
242  
243  vector<vector<double> > Hessenberg(vector<vector<double> > A) {
244      FUNCTION( y: "Hessenberg");
245      for (int r = 0; r < (int) A.size() - 2; ++r) {
246          vector<double> ar(A.size(), 0);
247          for (int i = 0; i < (int) A.size(); ++i) {
248              ar[i] = A[i][r];
249          }
250          double c = 0;
251          for (int i = r + 1; i < (int) A.size(); ++i) {
252              c += pow(ar[i],  _Right: 2);
253          }
254          c = sqrt(c);
255          c = (-c * sgn( x: ar[r + 1]));
256          double p = sqrt(2 * c * (c - ar[r + 1]));
257          vector<double> u(A.size(), 0);
258          for (int i = r + 1; i < (int) A.size(); ++i) {
259              if (i == r + 1) {
260                  u[i] = (ar[i] - c) / p;
261              } else {
262                  u[i] = ar[i] / p;
263              }
264          }
```

```
264             }
265             vector<vector<double> > I;
266             for (int i = 0; i < (int) A.size(); ++i) {
267                 vector<double> z(A.size(), 0);
268                 z[i] = 1;
269                 I.push_back(z);
270             }
271             vector<vector<double> > H = I - 2 * (u * u);
272             bool s;
273             A = H * A * Inverse(H, s);
274         }
275         return A;
276     }
277 
278     vector<vector<double> > QR(vector<vector<double> > A,
279             vector<vector<double> > &Q) {
280         FUNCTION("QR");
281         vector<vector<double> > I;
282         for (int i = 0; i < (int) A.size(); ++i) {
283             vector<double> z(A.size(), 0);
284             z[i] = 1;
285             I.push_back(z);
286         }
287         for (int i = 0; i < (int) A.size() - 1; ++i) {
288             double theta = atan(A[i + 1][i] / A[i][i]);
289             vector<vector<double> > P;
290             for (int r = 0; r < (int) A.size(); ++r) {
291                 vector<double> z(A.size(), 0);
292                 z[r] = 1;
293                 P.push_back(z);
294             }
295             P[i][i + 1] = sin(theta);
296             P[i][i] = cos(theta);
297             P[i + 1][i + 1] = cos(theta);
298             P[i + 1][i] = (-sin(theta));
299             I = P * I;
300             vector<double> aa(A[i]);
301             vector<double> aa1(A[i + 1]);
302             for (int j = i; j < (int) A.size(); ++j) {
303                 A[i][j] = aa[j] * cos(theta) + aa1[j] * sin(theta);
```

```
505                 }
506             }
507             bool s;
508             Q = Inverse( a: I,  &: s);
509             return A;
510         }
511
512         double Delta(vector<vector<double> > a) {
513             FUNCTION( y: "Delta");
514             double ss = 0;
515             for (int i = 0; i < (int) a.size(); ++i) {
516                 for (int j = 0; j < (int) a[i].size(); ++j) {
517                     if (i != j) {
518                         if (fabs(a[i][j]) > ss) {
519                             ss = fabs(a[i][j]);
520                         }
521                     }
522                 }
523             }
524             return ss;
525         }
526
527         double Delta1(vector<vector<double> > A) {
528             FUNCTION( y: "Delta1");
529             double d = 0;
530             for (int i = 0; i < (int) A.size() - 1; ++i) {
531                 if (fabs(A[i][i + 1]) > d) {
532                     d = fabs(A[i][i + 1]);
533                 }
534                 if (fabs(A[i + 1][i]) > d) {
535                     d = fabs(A[i + 1][i]);
536                 }
537             }
538             return d;
539         }
540
541         vector<complex<double> > Namta(vector<vector<double> > A, double delta) {
542             FUNCTION( y: "Namta");
```

```
341 vector<complex<double> > Namta(vector<vector<double> > A, double delta) {
342     FUNCTION( y: "Namta");
343     double delta1 = 0;
344     while ((Delta( a: A) >= delta) && (fabs(Delta1( A: A) - delta1) >= delta)) {
345         delta1 = Delta1( A: A);
346         A = Hessenberg( A: A);
347         vector<vector<double> > Q;
348         vector<vector<double> > R;
349         R = QR( A: A,  &: Q);
350         A = R * Q;
351     }
352     vector<complex<double> > namta;
353     if (Delta( a: A) < delta) {
354         for (int i = 0; i < (int) A.size(); ++i) {
355             complex<double> dd(A[i][i],  _Imagval: 0);
356             namta.push_back(dd);
357         }
358     } else {
359         int r = 0;
360         while (r < A.size() - 1) {
361             if (fabs(A[r + 1][r]) >= delta) {
362                 double b = -(A[r][r] + A[r + 1][r + 1]);
363                 double c = (A[r][r] * A[r + 1][r + 1]
364                         - A[r][r + 1] * A[r + 1][r]);
365                 if (pow(b,  _Right: 2) - 4 * c < 0) {
366                     complex<double> d1(-b / 2, sqrt(4 * c - pow(b,  _Right: 2)) / 2);
367                     complex<double> d2(-b / 2, -sqrt(4 * c - pow(b,  _Right: 2)) / 2);
368                     namta.push_back(d1);
369                     namta.push_back(d2);
370                 } else {
371                     complex<double> d1(-b / 2 + sqrt(pow(b,  _Right: 2) - 4 * c) / 2, 0);
372                     complex<double> d2(-b / 2 - sqrt(pow(b,  _Right: 2) - 4 * c) / 2, 0);
373                     namta.push_back(d1);
374                     namta.push_back(d2);
375                 }
376                 r += 2;
377             } else {
378                 complex<double> d(A[r][r],  _Imagval: 0);
379                 namta.push_back(d);
380                 ++r;
381             }
```

```
383             if (r == A.size() - 1) {
384                 complex<double> d(A[r][r],  _Imagval: 0);
385                 namta.push_back(d);
386             }
387         }
388         return namta;
389     }
390 
391     void Print(vector<complex<double> > a) {
392         FUNCTION( y: "Print");
393         for (int i = 0; i < (int) a.size(); ++i) {
394             cout << "\t\t" << a[i] << endl;
395         }
396     }
397 
398     void Print(vector<double> a) {
399         FUNCTION( y: "Print");
400         cout << "\t\t";
401         for (int i = 0; i < (int) a.size(); ++i) {
402             printf("%3.5f | ", a[i]);
403         }
404         cout << endl;
405     }
406 
407     double Delta(vector<double> a, vector<double> b) {
408         FUNCTION( y: "Delta");
409         double x = 0;
410         for (int i = 0; i < (int) a.size(); ++i) {
411             if (fabs(a[i] - b[i]) > x) {
412                 x = fabs(a[i] - b[i]);
413             }
414         }
415         return x;
416     }
417 
418     double Max(vector<double> a) {
419         FUNCTION( y: "Max");
420         double x = 0;
421         int n = 0;
422         for (int i = 0; i < (int) a.size(); ++i) {
423             if (fabs(a[i]) > x) {
```

```
492  int main() {
493      L1( y: "Preference Matrix definition");
494      L2( y: "Enter matrix size: ");
495      int n;
496      cin >> n;
497
498      L2( y: "create matrix");
499      double **a;
500      a = new double*[n];
501      for (int i = 0; i < n; i++) {
502          a[i] = new double[n];
503      }
504
505      L2( y: "intitialize matrix with user's values");
506      for (int i = 0; i < n; i++) {
507          for (int j = i + 1; j < n; j++) {
508              cout << "\t\tline [" << i << "] column [" << j << "] Enter value : "<< endl;
509              cin >> *(a[i] + j);
510          }
511      }
512
513      L1( y: "Define stop conditions");
514      L2( y: "enter alpha value: ");
515      double alpha;
516      cin >> alpha;
517
518      L2( y: "enter beita value: ");
519      double beita;
520      cin >> beita;
521
522      L1( y: "Tranverse matrix");
523      tranverse(a, alpha, beita, n);
524
525      L1( y: "Create a one dimension vector from preference matrix");
526      vector<vector<double> > A(n);
527      for (int i = 0; i < n; i++) {
```

```
535        L1( y: "Show preference matrix");
536        for (int i = 0; i < n; i++) {
537            cout << "\t\t";
538            for (int j = 0; j < n; j++) {
539                if (i > j) {
540                    cout << BLUE;
541                }
542                if (i == j) {
543                    cout << CYAN;
544                }
545                printf("%7.5f | ", (*(a[i] + j)));
546                cout << DEFAULT_COLOR;
547            }
548            cout << endl;
549        }
550
551        L2( y: "Enter delta value:");
552        double delta;
553        cin >> delta;
554
555        L1( y: "Compute Namta");
556        vector<complex<double> > namta = Namta( A: A, delta);
557        L2( y: "namta complex vector: ");
558        Print( a: namta);
559
560        L2( y: "search max real Namta value");
561        double maxnamta;
562        for (int i = 0; i < n; i++) {
563            // found value is real AND bigger than current
564            if ((namta[i].real() > maxnamta) && (namta[i].imag() == 0)) {
565                maxnamta = namta[i].real();
566            }
567        }
568
569        L2( y: "max real Namta value: ");
570        cout << maxnamta << endl;
571
572        L1( y: "Check consistency");
573        if (checkConsistency(maxnamta, n) == 1) {
```

```
572        L1( y: "Check consistency");
573        if (checkConsistency(maxnamta, n) == 1) {
574            OK( x: "", y: "—consistency check : Passed! ");
575        } else {
576            NOK( x: "", y: "—consistency check : Failed! ");
577        }
578
579        L1( y: "Preference Eigenvector");
580        L2( y: "compute");
581        vector<double> ve = ComputeVector( A: A, namta: maxnamta, delta);
582        L2( y: "show values");
583        Print( a: ve);
584
585        L1( y: "Normalize vector");
586        L2( y: "define vector sum");
587        double sum = 0;
588        for (int i = 0; i < n; i++) {
589            sum = sum + ve[i];
590        }
591
592        L2( y: "normalized result: ");
593        vector<double> venorl = ve / sum;
594        Print( a: venorl);
595
596        L1( y: "End");
597        return 0;
598    }
```