Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)
Кафедра комп'ютерних наук
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

магістр
(назва освітнього ступеня)

на тему: Огляд архітектурних практик для їх впровадження в Agile-розробку програмних продуктів

Виконав(ла): студент(ка) 6 курсу, групи СНм-61
спеціальності 122 Комп'ютерні науки

(шифр і назва спеціальності)

Вивюрка А.М.
(підпис) (прізвище та ініціали)

Керівник Боднарчук І.О.
(підпис) (прізвище та ініціали)

Нормоконтроль Дуда О.М.
(підпис) (прізвище та ініціали)

Завідувач кафедри Боднарчук І.О.
(підпис) (прізвище та ініціали)

Рецензент
(підпис) (прізвище та ініціали)

Тернопіль
2023

Міністерство освіти і науки України

Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)

Кафедра комп'ютерних наук
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри
____________ Боднарчук І.О.
(підпис) (прізвище та ініціали)
«___» __________ 202__ р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня Магістр
(назва освітнього ступеня)

за спеціальністю 122 Комп'ютерні науки
(шифр і назва спеціальності)

студенту Вивюрка Андрій Михайлович
(прізвище, ім'я, по батькові)

1. Тема роботи **Огляд архітектурних практик для їх впровадження в Agile-розробку програмних продуктів**

Керівник роботи **к.т.н., доц. Боднарчук І.О.**
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом ректора від «*24*» *листопада* 2023 року № *4/7-1099*

2. Термін подання студентом завершеної роботи 25 грудня 2023 р.

3. Вихідні дані до роботи Літературні джерела з тематики роботи

4. Зміст роботи (перелік питань, які потрібно розробити)

ВСТУП 1 АНАЛІЗ AGILE-МЕТОДІВ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 1.1 Роль системного архітектора в Agile-проєкті 1.2 Гнучке моделювання (Agile Modelling – AM) 1.3 Розробка на основі вастивостей (Feature Driven Design) 1.4 Метод розробки динамічних систем (Dynamic Systems Development Method – DSMD) 1.5 Екстремальне програмування (Extreme Programming – XP) 1.6 Scrum 1.7 Адаптивна розробка програмного забезпечення (ASD) 2 ОГЛЯД ПРОЦЕСУ ПРОЄКТУВАННЯ ПРОГРАМНОЇ АРХІТЕКТУРИ 2.1 Визначення програмної архітектури 2.2 Подання архітектури програмного забезпечення 2.3 Види активностей при проєктуванні програмної архітектури 2.4 Процеси при проєктування програмної архітектури 3 ОСОБЛИВОСТІ ПРОЄКТУВАННЯ ПРОГРАМНОЇ АРХІТЕКТУРИ В УМОВАХ AGILE-РОЗРОБКИ 3.1 Визначення вимог до програмної архітектури 3.2 Ідентифікація архітектурних стилів 3.3 Оцінка програмної архітектури 3.4 Визначення відповідності архітектури вимогам 3.5 Опис програмної архітектури 3.6 Інтеграція архітектури програмного забезпечення 3.7 Постійне архітектурне вдосконалення 3.8 Зв'язок між вимогами до програмного продукту та архітектурними активностями в Agile-проєктах 4 ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ ВИСНОВКИ; СПИСОК ДЖЕРЕЛ; ДОДАТКИ

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, слайдів)

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Охорона праці	Семчишин В. С., к.т.н., доц.		
Безпека в надзвичайних ситуаціях	Клепчик В.М., ст. викл.		

7. Дата видачі завдання "   "   202 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1.	Ознайомлення з завданням до кваліфікаційної роботи	21.09.23-27.09.23	*Виконано*
2.	Підбір наукових джерел по темі роботи	28.09.23-04.10.23	*Виконано*
3.	Переклад та опрацювання наукових джерел	05.10.23-11.10.23	*Виконано*
	по темі кваліфікаційної роботи		
4.	Виконання дослідження щодо теми	12.10.23-18.10.23	*Виконано*
	Кваліфікаційної роботи		
5.	Оформлення першого розділу	19.10.23-25.10.23	*Виконано*
6.	Оформлення другого розділу	26.10.23-01.11.23	*Виконано*
7.	Оформлення третього розділу	02.11.23-08.11.23	*Виконано*
8.	Виконання завдання до підрозділу «Охорона праці»	09.11.23-15.11.23	*Виконано*
9.	Виконання завдання до підрозділу «Безпека в	16.11.23-23.11.23	*Виконано*
	надзвичайних ситуаціях»		
10.	Оформлення кваліфікаційної роботи	23.11.23-29.11.23	*Виконано*
11.	Нормоконтроль	30.11.23-09.12.23	*Виконано*
12.	Перевірка на плагіат	10.12.23	*Виконано*
13.	Попередній захист кваліфікаційної роботи	14.12.23	*Виконано*
14.	Захист кваліфікаційної роботи	26.12.2023	

Студент _______________ Вивюрка А.М.
(підпис) (прізвище та ініціали)

Керівник роботи _______________ Боднарчук І.О.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

"Огляд архітектурних практик для їх впровадження в Agile-розробку програмних продуктів" // Кваліфікаційна робота освітнього рівня «Магістр» // Вивюрка Андрій Михайлович // Тернопільський національний технічний університет ім. І. Пулюя, факультет комп’ютерно-інформаційних систем і програмної інженерії, кафедра комп'ютерних наук, група СНм-61 // Тернопіль, 2023 // с. – 45, рис. – 8, табл. – 2, джерел – 28.

Ключові слова: життєвий цикл розробки програмного забезпечення, гнучка методологія, архітектурний дизайн програмного забезпечення

У цій роботі пропонується новий підхід до керівництва та підтримки практик, що стосуються архітектури та проєктування програмного забезпечення у гнучких середовищах. Архітектура та дизайн програмного забезпечення є фундаментальними основами системи, визначаючи її поведінку з урахуванням різних функціональних і нефункціональних вимог. На сьогоднішній день відсутня чітка специфікація процесів проєктування архітектури програмного забезпечення у гнучких середовищах. Наша методологія докладно описує етапи гнучкого проєктування програмного забезпечення та пропонує методи та інструменти для їх реалізації.

ANNOTATION

“Review of Architectural Practices for Implementation in Agile-Development of Software Products” // Master’s degree qualification paper // Vyviurka Andrii Mychailovych // Ternopil Ivan Puluj National Technical University, Faculty of Computer Information Systems and Software Engineering, Computer Science Department, group, group СНм-61 // Ternopil, 2023 // p. – 45, fig. – 8, tables – 2, references – 28.

Key words: software development life cycle, Agile methodology, software architectural design

This work proposes a novel methodology for guiding and supporting practices related to architecture and software design in flexible environments. Software architecture and design serve as the backbone of a system, dictating its behavior concerning various functional and non-functional requirements. Currently, there's no clear specification for the activities and processes involved in designing software architecture in flexible environments. Our methodology provides a detailed description of the stages in the flexible software design process and offers methods and tools for implementing these stages.

ЗМІСТ

ВСТУП ... 7
1 АНАЛІЗ AGILE-МЕТОДІВ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ... 10
1.1 Роль системного архітектора в Agile-проєкті ... 13
1.2 Гнучке моделювання (Agile Modelling – AM) ... 14
1.3 Розробка на основі вастивостей (Feature Driven Design) ... 15
1.4 Метод розробки динамічних систем (Dynamic Systems Development Method – DSMD) ... 16
1.5 Екстремальне програмування (Extreme Programming – XP) ... 17
1.6 Scrum ... 18
1.7 Адаптивна розробка програмного забезпечення (ASD) ... 19
2 ОГЛЯД ПРОЦЕСУ ПРОЄКТУВАННЯ ПРОГРАМНОЇ АРХІТЕКТУРИ ... 20
2.1 Визначення програмної архітектури ... 20
2.2 Подання архітектури програмного забезпечення ... 20
2.3 Види активностей при проєктуванні програмної архітектури ... 21
2.4 Процеси при проєктування програмної архітектури ... 22
3 ОСОБЛИВОСТІ ПРОЄКТУВАННЯ ПРОГРАМНОЇ АРХІТЕКТУРИ В УМОВАХ AGILE-РОЗРОБКИ ... 24
3.1 Визначення вимог до програмної архітектури ... 24
3.2 Ідентифікація архітектурних стилів ... 25
3.3 Оцінка програмної архітектури ... 26
3.4 Визначення відповідності архітектури вимогам ... 27
3.5 Опис програмної архітектури ... 28
3.6 Інтеграція архітектури програмного забезпечення ... 29
3.7 Постійне архітектурне вдосконалення ... 30
3.8 Зв'язок між вимогами до програмного продукту та архітектурними активностями в Agile-проєктах ... 31

4 ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ 34
4.1 Вплив факторів трудового середовища на здоров'я та працездатність розробника програм 34
4.2 Шкідливий вплив іонізуючого випромінювання 38
ВИСНОВКИ 42
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 43
ДОДАТКИ

ВСТУП

Актуальність задачі.

Принципи розробки програмного забезпечення, які визначені в «Agile Manifesto» [1], стали ідеалом, до якого прагне багато компаній. Agile підкреслює важливість гнучкого життєвого циклу розробки, спрямованого на співпрацю замість формальних процесів, віддаючи перевагу функціонуючому програмному продукту, постійній зміні і спілкуванню замість докладної документації. Ці принципи стали основою для підходів у розробці, що акцентують на прискоренні часу реакції на зміни, задоволенні потреб клієнта та ефективному використанні ресурсів.

Дослідження в області проєктування програмної архітектури проводять також в Тернопільському національному технічному університеті імені Івана Пулюя. Наприклад, у статтях [2], [3], [4] висвітлюються процеси забезпечення якості в проєктах за гнучкими технологіями етапах проєктування програмної архітектури.

Проєкти розробки програмного забезпечення, спрямовані на швидке, стійке постачання нового функціоналу, поєднують методи гнучкості та розробки архітектури для досягнення конкуруючих цілей швидкості в короткостроковій перспективі та стабільності в довгостроковій перспективі [5], [6]. Життєвий цикл розробки програмного забезпечення – це, по суті, серія кроків або фаз, включаючи специфікацію вимог; проєктування програмного забезпечення; побудову програмного забезпечення; верифікація та валідація і розгортання програмного забезпечення. Ці фази забезпечують модель для розробки та управління програмним забезпеченням [7].

Проєктування архітектури програмного забезпечення – це процес застосування різних методів і принципів з метою визначення модуля, процесу або системи досить детально, щоб дозволити їх фізичне кодування. Традиційний підхід до процесу розробки програмного забезпечення зосереджується на розділенні задачі та її вирішення на детальні частини перед тим, як перейти до

фази створення. Це піднімає роль архітектури програмного забезпечення, яка стає основою проєкту і є критично важливою, не залишаючи місця для задоволення мінливих вимог пізніше в циклі розробки.

Відсутня чітка специфікація дій у процесі гнучкого проєктування програмного забезпечення, а також бракує набору методів для вибору при архітектурному проєктуванні [9].

Існує очевидна потреба в підході до проєктування архітектури програмного забезпечення в гнучких середовищах. Наскільки відомо, в жодній літературі не було запропоновано добре встановленої методології розробки програмного забезпечення в контексті вирішення задачі розробки архітектури програмного забезпечення, які повністю підтримують основи Agile методів.

Мета роботи.

Метою дослідження є пропозиція фреймворку для проєктування програмної архітектури у Agile-проєктах.

Об'єкт дослідження: Процеси розробки програмного забезпечення за Agile-методологіями.

Предмет дослідження: моделі життєвого циклу проєктів з Agile–методологіями.

Наукова новизна отриманих результатів. Наукова новизна полягає у пропозиції фреймворку проєктування програмної архітектури в проєктах, організованих за гнучкими технологіями розробки Agile.

Практичне значення отриманих результатів. У цій роботі представлене дослідження на основі огляду відкритих друкованих джерел, що дозволило запропонувати методи та практики проєктування програмної архітектури в гнучких проєктах. Фреймворк дозволить вирішити деякі з проблем, з якими стикаються організації, які беруть участь у проєктуванні програмного забезпечення [8]:

1. Вимоги змінюються з часом через зміни в потребах клієнтів і користувачів, технологічний прогрес і обмеження графіку.

2. Зміни вимог систематично передбачають зміну дизайну програмного забезпечення та, у свою чергу, програмного коду.

3. Пристосування до зміни дизайну програмного забезпечення є дорогою критичною діяльністю в умовах швидкої зміни вимог.

Апробація результатів та особистий внесок здобувача. Основні положення роботи доповідались, розглядались та обговорювались на науковій конференції Тернопільського національного технічного університету імені Івана Пулюя та опубліковані у тезах студентської наукової конференції "Інформаційні моделі системи та технології – 2023", та " Актуальні задачі сучасних технологій – 2023", які проводились у ТНТУ.

1 АНАЛІЗ AGILE-МЕТОДІВ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Мета гнучких методів полягає в тому, щоб дозволити організації бути гнучкою, але варто чіткіше описати поняття гнучкості (Agile). Agile означає здатність «швидко виконувати»; «мінятися швидко і часто» [1].

Незважаючи на те, що різні гнучкі методи відрізняються за практикою та акцентом, вони дотримуються тих самих принципів, що лежать в основі гнучкого маніфесту [9]:

1. Працююче програмне забезпечення доставляється часто (тижнями, а не місяцями).
2. Працююче програмне забезпечення є основним показником прогресу.
3. Задоволення клієнтів шляхом швидкої та безперервної доставки корисного програмного забезпечення.
4. Пізні зміни вимог до програмного забезпечення приймаються.
5. Тісна щоденна співпраця між бізнесменами та розробниками програмного забезпечення.
6. Розмова віч-на-віч – найкраща форма спілкування.
7. Проєкти будуються навколо мотивованих людей, яким варто довіряти.
8. Постійна увага до технічної досконалості та гарного дизайну.

Гнучкі методи розробки були створені, щоб вирішити проблему своєчасної доставки високоякісного програмного забезпечення в умовах постійної та швидкої зміни вимог і бізнес-середовища. Гнучкі методи мають перевірену історію в індустрії програмного забезпечення та IT. Основна перевага гнучкого методу створення програмного забезпечення полягає в тому, що він забезпечує адаптивний процес, у якому команда та розробники реагують на зміни вимог і специфікацій і обробляють їх навіть на пізній стадії процесу розробки. Рисунок 1.1 ілюструє абстрактний вигляд еволюційної карти основних гнучких методів розробки.

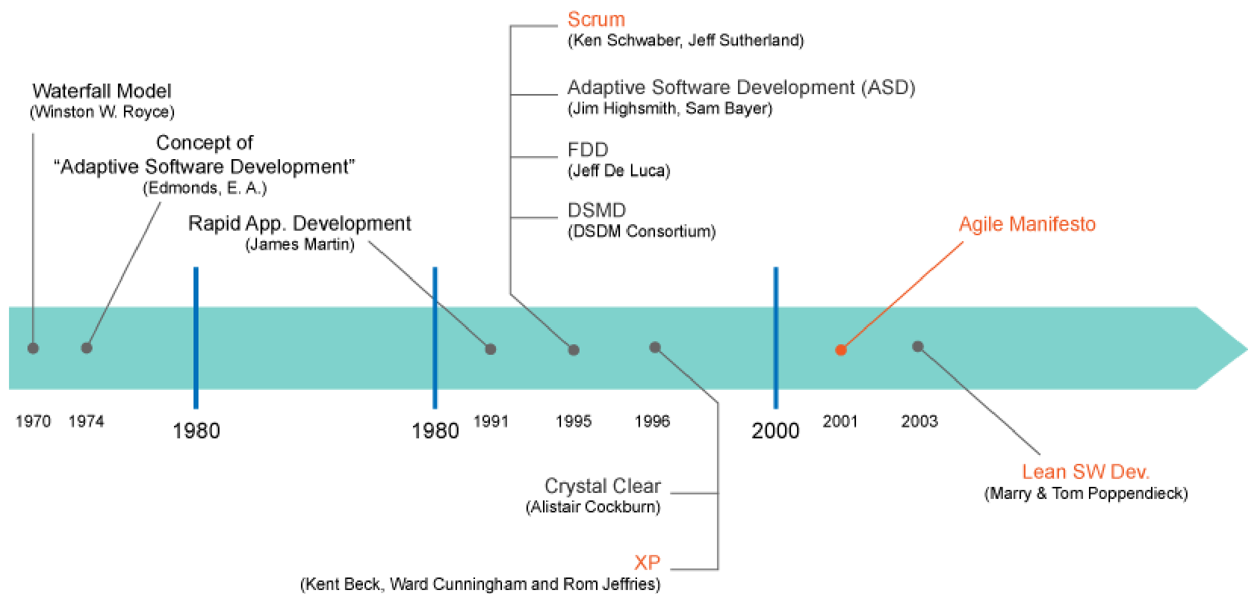

Рисунок 1.1 – Історія розвитку Agile-методів розробки ПЗ

Завдяки використанню кількох робочих ітерацій впровадження гнучких методів дозволяє створювати якісне функціональне програмне забезпечення з невеликими командами та обмеженими ресурсами.

Прихильники традиційних методів розробки критикують гнучкі методи за недостатню увагу до документації та нездатність співпрацювати в традиційному робочому процесі.

Основні обмеження Agile-розробки:

– Agile добре працює для малих та середніх команд;

– гнучкі методи розробки не масштабуються, тобто через кількість залучених ітерацій буде важко зрозуміти поточний статус проєкту;

– гнучкий підхід вимагає високомотивованих і кваліфікованих людей, які не завжди будуть доступні;

– недостатня кількість письмової документації в гнучких методах призводить до втрати інформації під час фактичного впровадження коду.

Стосовно архітектури програмного забезпечення, цікаво відзначити, що саме «полегшені» методи Agile, зокрема Scrum і XP, отримують найбільше впровадження на підприємстві. Ще цікавіше відзначити, що ці методи дають найменше вказівок (принаймні порівняно з FDD, DSDM, OpenUP та іншими)

щодо архітектури в розробці Agile. І це один із викликів. Ці методи базуються на припущенні, що архітектура виникає як природний результат швидкого ітераційного циклу, реалізації пріоритетних запитів користувачів, керованих цінністю, і безперервного процесу рефакторингу.

Для розробників, які мають досвід побудови великомасштабних систем із розширюваністю та масштабованістю, яких вимагає ринок, це здається суперечним тому, що передбачає досвід, а саме те, що для надійного виробництва та підтримки таких систем необхідні певна кількість архітектурного планування та керування.

Отже, завдання полягає в тому, щоб перенести бажані та якісні переваги Agile від окремих команд до кількох команд, щоб ці методи можна було використовувати для створення все більших програмних систем. Чи масштабуються гнучкі методи? Чи зможуть команди, які використовують ці методи, створювати програмні системи, які масштабуються до рівнів надійності та розширюваності, необхідних підприємствам? Відповідь на обидва запитання може бути «так», за умови, що ми відновимо центральну роль архітектурs системи.

Рефакторинг є прикладом. Рефакторинг є такою ж невід’ємною частиною Agile, як щоденний стендап, модульне тестування та ретроспектива. Це має бути саме так, тому що з мінімальними або відсутніми «попередніми» вимогами та дизайном команди розуміють, що вони не можуть зробити це правильно з першого разу. Натомість вони покладаються на навички рефакторингу, щоби швидко адаптувати систему до потреб клієнта.

Однак, коли з’являється краще розуміння потреб ринку, безперервний рефакторинг великомасштабних нових архітектур стає менш практичним у міру зростання розміру системи. Крім того, для покращення зручності використання, розширюваності, продуктивності та обслуговування більшість великих команд застосовують підходи на основі компонентів. Може бути корисно, якщо ці компоненти використовують загальні підходи до поведінки та реалізації.

Переваги застосування «архітектурного керівництва» до цих практик включають:

Уникнення масштабних і непотрібних переробок. Навіть незначні рефакторинги системного рівня можуть спричинити суттєві переробки для великої кількості команд, деяким із яких інакше не доведеться переробляти свій модуль. Одна справа, коли одна команда виконує рефакторинг коду на основі уроків, які вона засвоїла, а зовсім інша – вимагати від кількох команд рефакторинг коду на основі уроків, отриманих іншими командами.

Управління ризиками впливу на розгорнуті системи та користувачів. Навіть найкращі з можливих систем недосконалі, тому загроза появи регресивної помилки в розгорнутій системі завжди присутня. А вартість, ризик і вплив помилки зростають разом із масштабом. Зведення до мінімуму непотрібного рефакторингу є основною технікою зменшення ризику.

Деякі загальні, нав’язані архітектурні конструкції можуть полегшити зручність використання, розширюваність, продуктивність і обслуговування. Наприклад, щось таке просте, як нав’язування спільного дизайну презентації, може призвести до підвищення рівня задоволеності кінцевих користувачів і, зрештою, додаткового прибутку.

1.1 Роль системного архітектора в Agile-проєкті

Існують значні переваги, коли ефективно застосований уніфікований стиль архітектури, за умови, що розробка не сповільнюється і ми не капітулюємо перед фазами водоспаду проєктування минулого.

Історично єдиний архітектурний стиль був основною функцією архітектора системи. Але найпоширеніші методи Agile не визначають і навіть не підтримують таку роль. Оскільки Agile зосереджується на використанні потужності колективної команди, а не окремої людини, системний архітектор більше не диктує технічний напрям.

Хоча системні архітектори мають десятиліття технічного досвіду, цей досвід, швидше за все, був поза процесом Agile, і вони можуть не розуміти конструкції побудови коду, що піддається рефакторингу. Дійсно, вони можуть розглядати цю практику як непотрібну переробку та не підтримувати модель Agile.

Системні архітектори також можуть бути стурбовані потенційною архітектурною ентропією всіх нових повноважень та команд Agile. Вони також можуть мати тверду думку про практику розробки програмного забезпечення, яку використовують команди. Якщо ми не зможемо залучити цих ключових зацікавлених сторін до парадигми Agile-розробки, вони можуть швидко знищити всю ініціативу.

Ми хочемо будь-якою ціною уникнути битви між командами Agile та системними архітекторами, оскільки в цій боротьбі не буде переможця. Тому, безперечно, в наших інтересах залучати системних архітекторів до Agile-процесу, і команда повинна високо цінувати їхній внесок.

Однак за належної реалізації гнучкі методи можуть доповнювати та приносити користь традиційним методам розробки. Крім того, слід зазначити, що традиційні методи розробки в неітераційних режимах реалізації проєктів сприйнятливі до поломки архітектури на пізній стадії, тоді як гнучкі методології ефективно вирішують цю проблему шляхом частих поступових побудов, які стимулюють зміну вимог. Опишемо деякі поширені гнучкі методи з точки зору розробки вимог.

1.2 Гнучке моделювання (Agile Modelling – AM)

Гнучке моделювання – це новий підхід для проведення моделювання [10]. Він дає розробникам вказівки щодо того, як створювати моделі, використовуючи гнучку філософію як основу для вирішення задачі проєктування та створення допоміжної документації, але не «перебудовують» ці моделі (рис. 1.2).

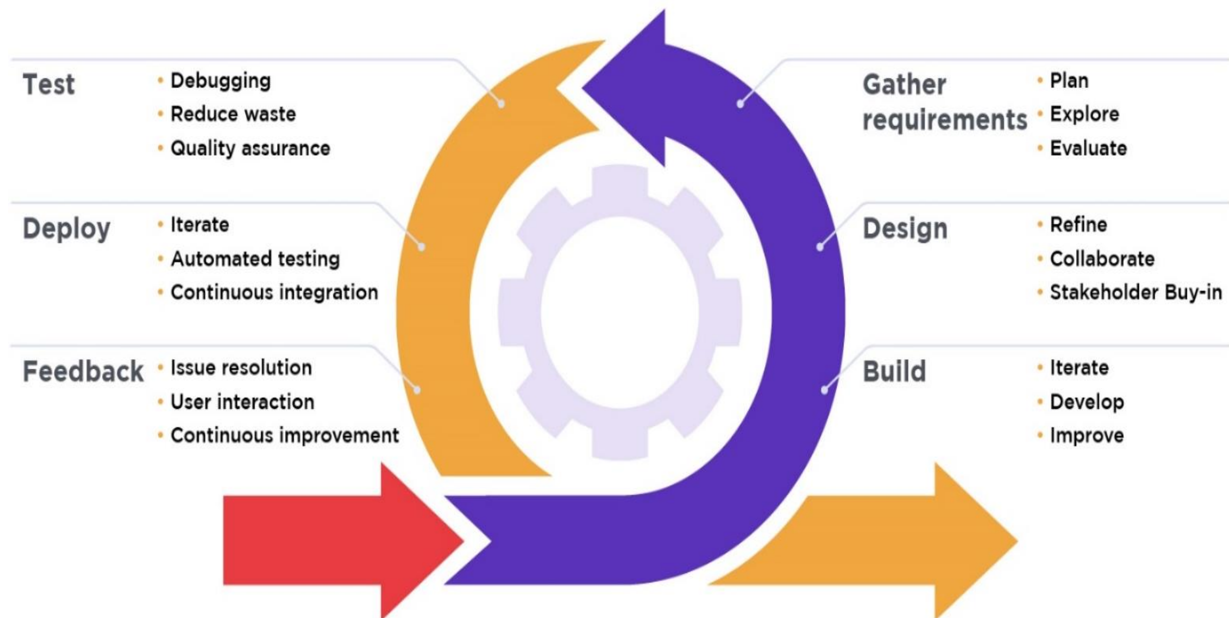

Рисунок 1.2 – Етапи Agile Modelling (AM)

Мета методу полягає в тому, щоб зберегти кількість моделей і документації.

1.3 Розробка на основі вастивостей (Feature Driven Design)

Розробка, керована властивостями, складається з мінімалістичного п'ятиетапного процесу, який зосереджується на фазах створення та проєктування [11], кожна з яких визначається критеріями входу та виходу, створення списку властивостей, а потім планування за властивостями, за яким слідує ітераційне проєктування за кроки побудови алгоритму роботи функції реалізації кожної властивості. На першому етапі експертами та розробниками домену створюється модель предметної області. Загальна модель складається з діаграм класів із власне класами, зв'язками, методами та атрибутами. Методи виражають функціональність і є основою для створення списку властивостей (рис. 1.3).

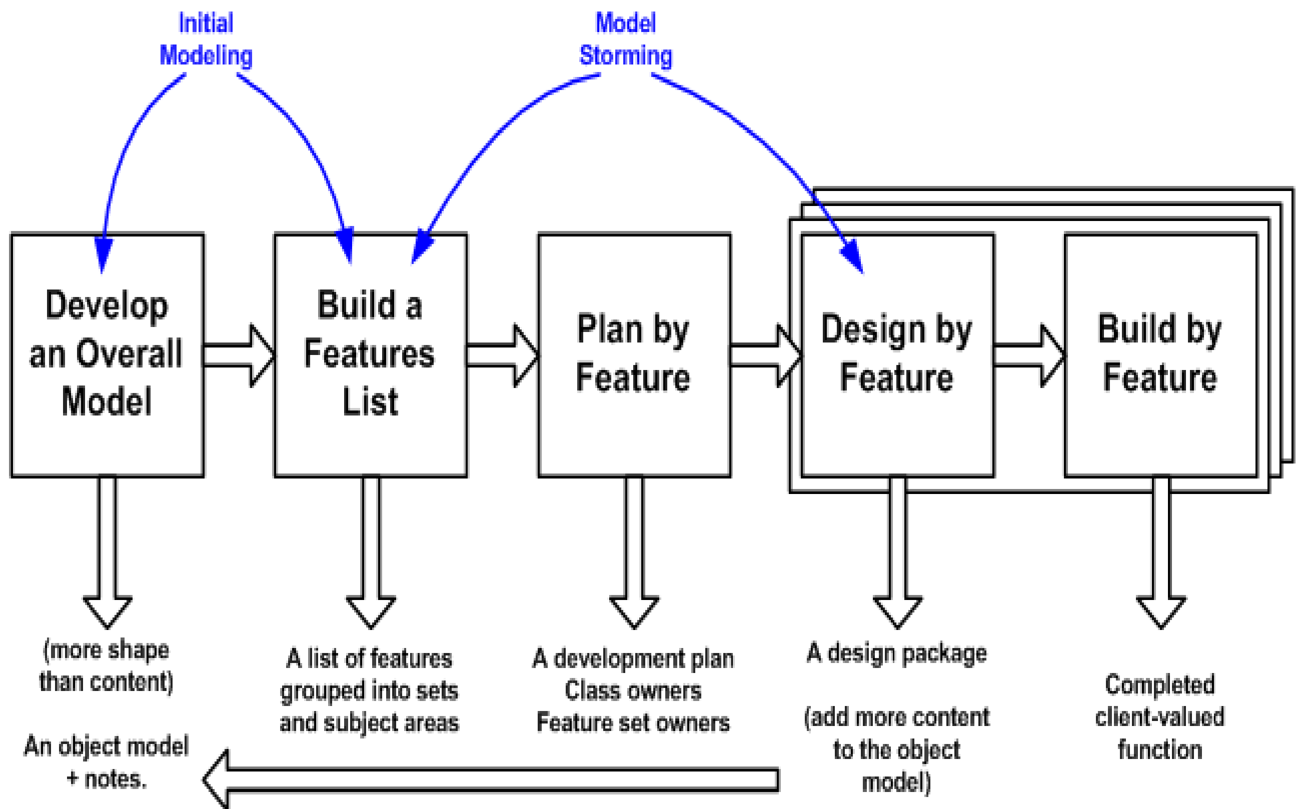

Рисунок 1.3 – Принцип FDD-розробки

Особливістю FDD є клієнтська функція. Списки властивостей визначаються командою за пріоритетністю. Список властивостей переглядається учасниками проєкту. FDD пропонує проводити щотижневі 30-хвилинні зустрічі, під час яких обговорюється статус функцій і пишеться звіт про зустріч.

1.4 Метод розробки динамічних систем (Dynamic Systems Development Method – DSMD)

Метод динамічної розробки систем був створений у Великобританії в середині 1990-х років. Це наслідок і розширення практики швидкої розробки додатків (Rapid Application Development – RAD) [12]. Перші дві фази DSDM – це техніко-економічне обґрунтування та бізнес-дослідження. Під час цих двох фаз визначаються базові вимоги (рис. 1.4).

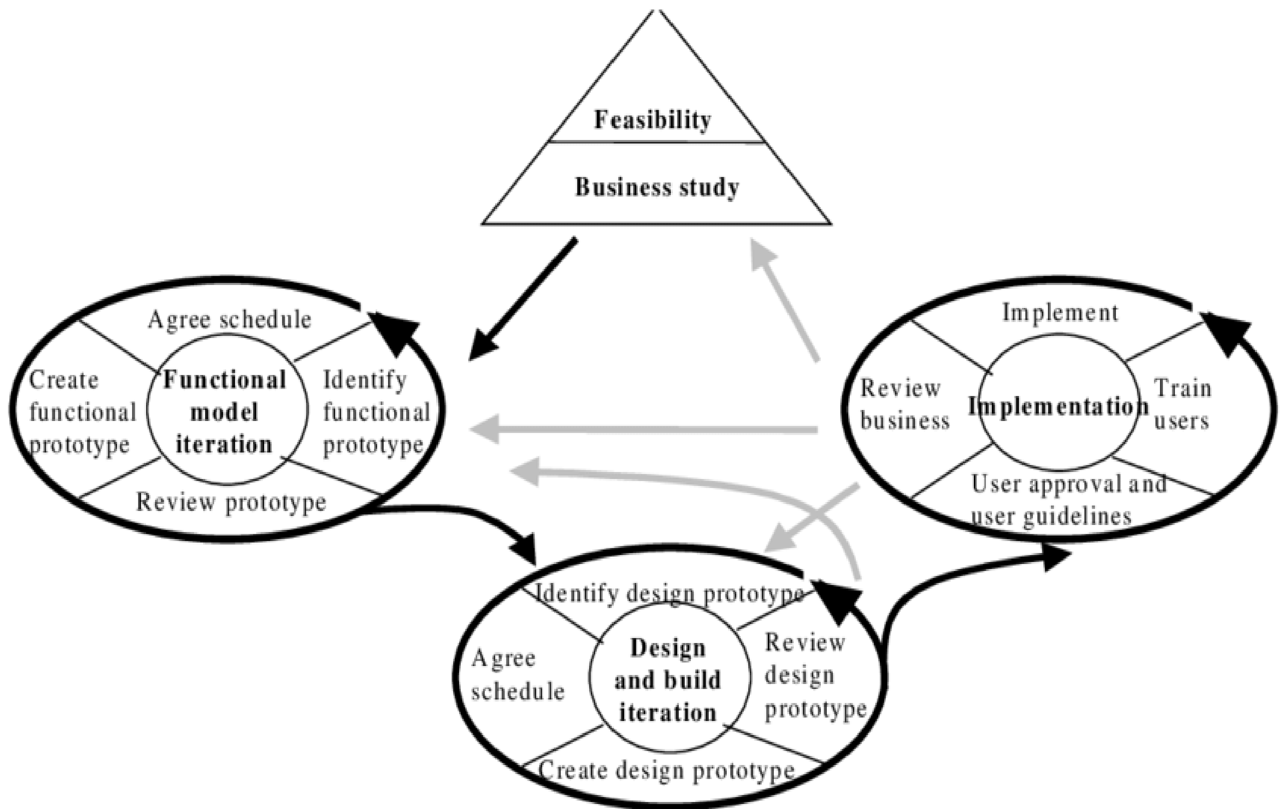

Рисунок 1.4 – Діаграма процесу DSDM-розробки

DSDM має дев’ять принципів, серед яких активне залучення користувачів, регулярна доставка нового функціоналу, командне прийняття рішень, інтегроване тестування протягом життєвого циклу проєкту та оборотні зміни в розробці.

1.5 Екстремальне програмування (Extreme Programming – XP)

Екстремальне програмування базується на цінностях простоти, спілкування, зворотного зв’язку [13]. XP має на меті забезпечити успішну розробку програмного забезпечення, незважаючи на невизначені або постійно змінювані вимоги до програмного забезпечення (рис. 1.5).

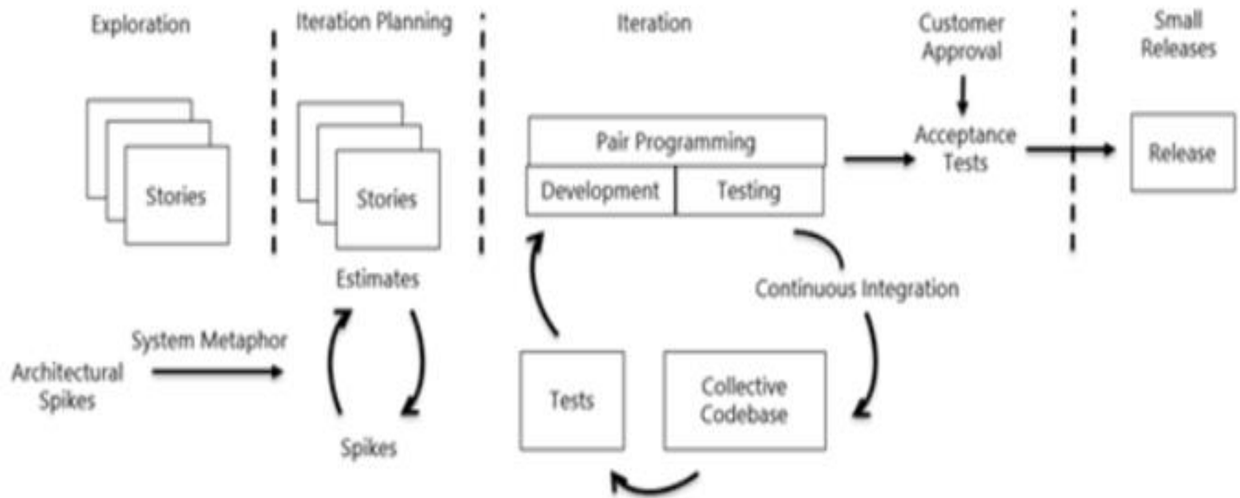

Рисунок 1.5 – Діаграма процесу екстремального програмування

XP спирається на методи, за якими окремі практики збираються та вибудовуються, щоб працювати одна з одною. Деякі з основних практик XP – це короткі ітерації з невеликими випусками та швидким зворотним зв’язком, тісна участь клієнтів, постійний зв’язок і координація, безперервний рефакторинг, безперервна інтеграція та тестування, а також парне програмування.

1.6 Scrum

Scrum – це емпіричний підхід, заснований на гнучкості, адаптивності та продуктивності [14]. Scrum дозволяє розробникам вибирати конкретні техніки, методи та практики розробки програмного забезпечення для процесу впровадження. На рисунку 1.6 детально описано робочий процес гнучкої розробки програмного забезпечення Scrum.

Scrum надає структуру управління проєктами, яка зосереджує розробку на 15 – 30-денних циклах, під час яких надається певний набір невиконаних функцій. Основною практикою Scrum є використання щоденних 15-хвилинних зустрічей команди для координації та інтеграції. Scrum використовується вже майже 15 років і успішно реалізує широкий спектр продуктів.

1.7 Адаптивна розробка програмного забезпечення (ASD)

Адаптивна розробка програмного забезпечення намагається створити новий спосіб бачення розробки програмного забезпечення в організації, просуваючи адаптивну парадигму [15]. Він пропонує рішення для розробки великих і складних систем.

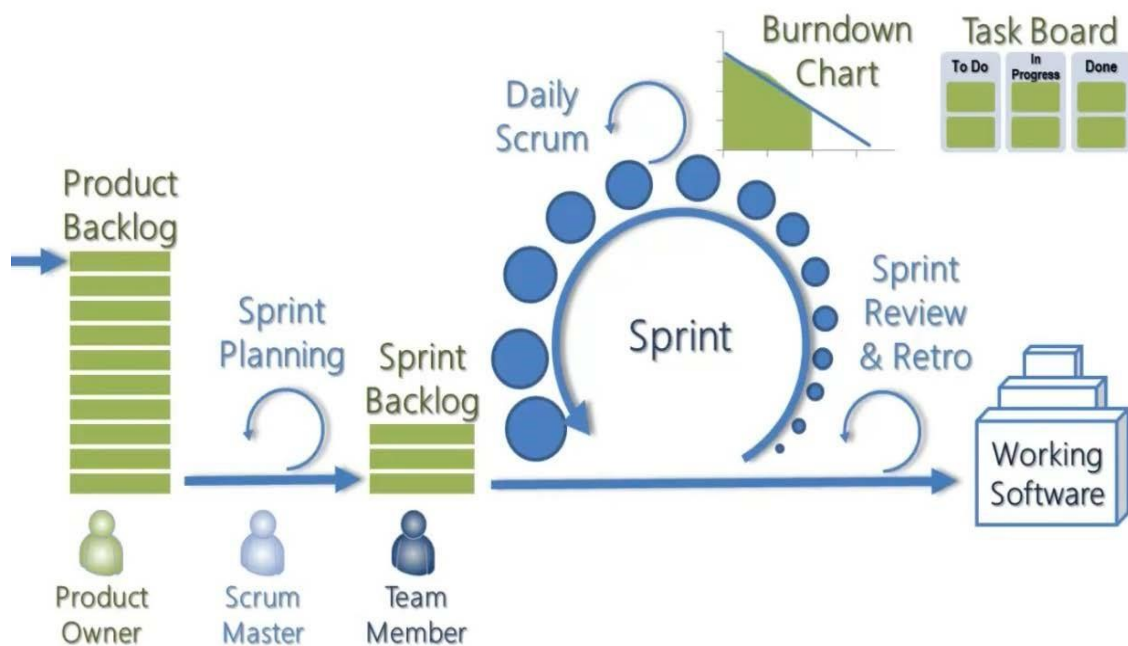

Рисунок 1.6 – Схема Scrum-розробки

Метод заохочує поступову та ітераційну розробку з постійним створенням прототипів. Одним із предків ASD є «RADical Software Development». ASD стверджує, що забезпечує структуру з достатніми вказівками, щоб запобігти хаосу проєктів, не пригнічуючи при цьому креативність та творчість учасників команди.

2 ОГЛЯД ПРОЦЕСУ ПРОЄКТУВАННЯ ПРОГРАМНОЇ АРХІТЕКТУРИ

2.1 Визначення програмної архітектури

Архітектура програмного забезпечення – це спосіб подання інформації про обчислювальні системи, наприклад, їх конфігурацію та дизайн. Під обчислювальними системами ми маємо на увазі апаратне забезпечення, програмне забезпечення та комунікаційні компоненти [16]. Набір компонентів, зібраних разом, не дає нам рішення проблеми [17]. Ми повинні нав'язати їм топологію взаємодії та зв'язку та забезпечити, щоб компоненти як інтегрувалися (фізично спілкувалися), так і взаємодіяли (логічно спілкувалися).

2.2 Подання архітектури програмного забезпечення

Процес проєктування архітектури програмного забезпечення зазвичай поділяють на чотири види: концептуальний, модульний, виконання та код. Цей поділ ґрунтується на нашому вивченні архітектур програмного забезпечення великих систем, а також на досвіді проєктування та перегляду архітектур програмного забезпечення [13]. Різні погляди вирішують різні інженерні проблеми, і поділ таких проблем допомагає архітектору приймати обґрунтовані рішення щодо компромісів у дизайні. Ідея такого поділу не є унікальною: більшість робіт з архітектури програмного забезпечення на сьогоднішній день або розпізнають різні види архітектури, або зосереджуються на одному конкретному представленні, щоб дослідити його відмінні характеристики та відрізнити його від інших [17].

Підхід 4 + 1 розділяє архітектуру на кілька видів [19]. Робота [20] зосереджена на концептуальному погляді на архітектуру ПЗ. Крім того, інші роботи зосереджені на представленні виконання програми, і, зокрема, досліджуються динамічні аспекти системи. Перегляд коду досліджувався в контексті керування конфігурацією та побудови системи.

Концептуальне подання програмної архітектури (ПА) описує архітектуру в термінах елементів предметної області. Тут архітектор проєктує функціональні особливості системи. Наприклад, одна спільна мета полягає в тому, щоб організувати архітектуру так, щоб функціональні властивості можна було додавати, видаляти або змінювати. Це важливо для еволюції, для підтримки лінійки продуктів і для повторного використання в поколіннях продукту.

Подання архітектури у вигляді модулів описує декомпозицію програмного забезпечення та його організацію на рівні. Тут важливо обмежити вплив змін у зовнішньому програмному чи апаратному забезпеченні. Іншим фактором є зосередження досвіду інженерів програмного забезпечення, щоб підвищити ефективність впровадження.

Подання роботи продукту – це представлення системи під час виконання, зіставлення модулів із зображеннями часу виконання, визначення зв’язку між ними та призначення їх фізичним ресурсам. Використання ресурсів і продуктивність є ключовими проблемами в поданні виконання. Тут приймаються такі рішення, як використання бібліотеки посилань чи спільної бібліотеки, чи використання потоків чи процесів, хоча ці рішення можуть повертатися до перегляду модуля та вимагати змін в ньому.

Подання на основі програмного коду відображає, як модулі та інтерфейси в поданні модуля відображаються у файлах з програмним кодом, а подання під час виконання відображаються у виконуваних файлах. Деякі представлення також мають конфігурацію, яка обмежує елементи, визначаючи, які ролі вони можуть відігравати в певній системі. У конфігурації архітектор може захотіти описати додаткові атрибути чи поведінку, пов’язану з елементами, або описати поведінку конфігурації системи в цілому.

2.3 Види активностей при проєктуванні програмної архітектури

Проєктування архітектури програмного забезпечення передбачає реалізацію ряду конкретних дій та процесів (що охоплюють весь життєвий цикл

архітектури) і ряду загальних дій з розробки архітектури (що підтримують певні процеси). У наступних розділах подамо короткий огляд активностей та процесів проєктування архітектури програмного забезпечення. Конкретні активності щодо архітектури програмного забезпечення є наступними:

– Архітектурний аналіз (AA) визначає проблеми, які має вирішити архітектура. Результатом цієї діяльності є набір архітектурно значущих вимог (ASR).

– Архітектурний синтез (AS) пропонує варіанти архітектурних рішень для реалізації ASR, зібраних в AA, таким чином ця діяльність переходить від задачі до її рішення.

– Архітектурна оцінка (AE) гарантує, що прийняті архітектурні проєктні рішення є правильними, а кандидати на архітектурні рішення, запропоновані в AS, оцінюються відносно ASR, зібраними в AA.

– Архітектурна реалізація (AI) реалізує архітектуру шляхом створення детального проєкту.

– Супровід та розвиток архітектури (AME) – це зміна архітектури з метою усунення недоліків, а архітектурна еволюція (розвиток) – реагування на нові вимоги на архітектурному рівні.

2.4 Процеси при проєктування програмної архітектури

Загальний процес створення ПА складається з шести конкретних процесів:

– Архітектурне відновлення (AR) використовується для отримання поточної архітектури системи з реалізації цієї системи (реверсна інженерія).

– Архітектурний опис (ADp) використовується для опису архітектури з набором архітектурних елементів (наприклад, подання архітектури). Ця діяльність може допомогти зацікавленим сторонам (наприклад, архітекторам) зрозуміти систему та покращити спілкування та співпрацю між зацікавленими сторонами.

– Архітектурне розуміння (AU) використовується для розуміння архітектурних елементів (наприклад, архітектурних рішень) та їхніх взаємозв’язків в архітектурному проєкті.

– Аналіз архітектурного впливу (AIA) використовується для визначення архітектурних елементів, на які впливає реалізація змін у вимогах. Результати аналізу включають компоненти в архітектурі, на які чиниться вплив безпосередньо, а також непрямі наслідки змін в архітектурі.

– Архітектурне повторне використання (ARu) спрямоване на повторне використання існуючих елементів архітектурного дизайну, таких як архітектурні структури, рішення та шаблони в архітектурі нової системи.

– Архітектурний рефакторинг (ARf) спрямований на покращення архітектурної структури системи без зміни її зовнішньої поведінки.

3 ОСОБЛИВОСТІ ПРОЄКТУВАННЯ ПРОГРАМНОЇ АРХІТЕКТУРИ В УМОВАХ AGILE-РОЗРОБКИ

Запропонована методологія проєктування архітектури програмного забезпечення в гнучких середовищах детально описана на рисунку 3.1.

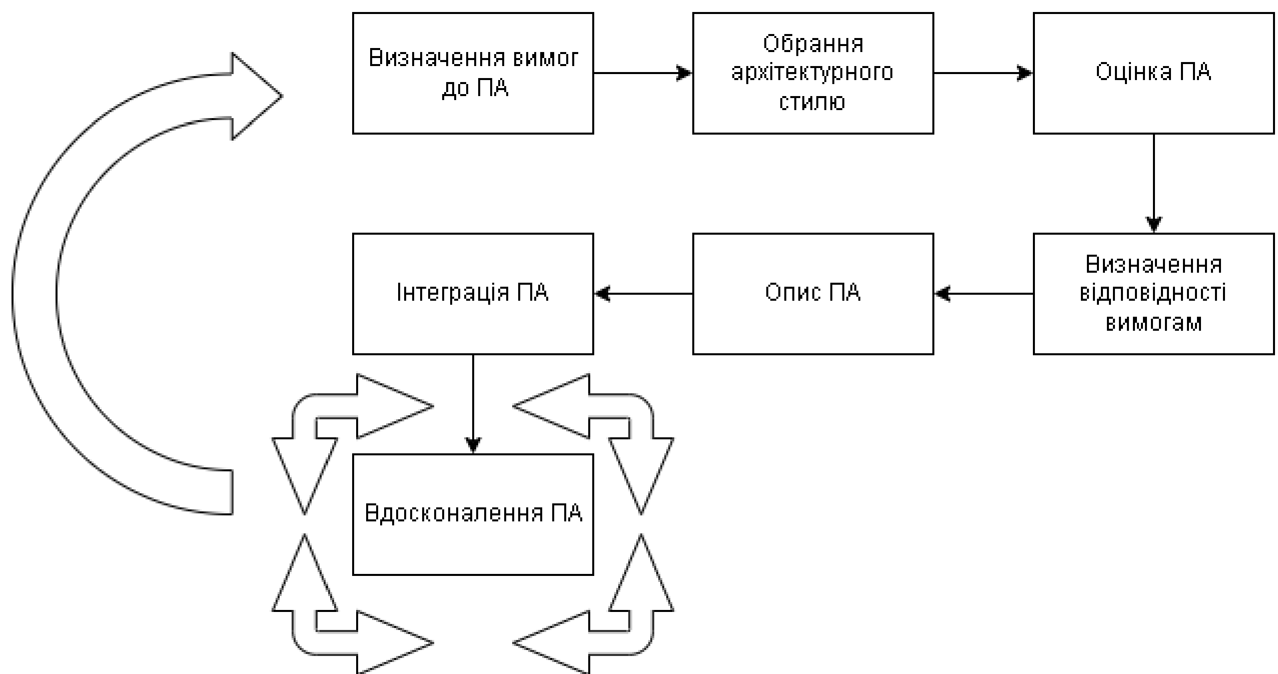

Рисунок 3.1 – Проєктування програмної архітектури в Agile-розробці

Опишемо детальніше наведені етапи та проаналізуємо особливості реалізації процесів проєктування програмної архітектури.

3.1 Визначення вимог до програмної архітектури

Встановлення переліку визначальних архітектурних вимог отримують шляхом аналізу бізнес-чинників і контексту системи, а також питань, які зацікавлені сторони продукту вважають критично важливими для успіху системи. Метою є специфікація архітектури, яка спрямовує архітекторів на створення структури системи, яка є достатньою для забезпечення успіху в очах зацікавлених сторін. Ці вимоги запобігають створенню архітектури, яка є надто складною або яка прагне до непотрібної елегантності за рахунок критичних

властивостей системи. Визначення архітектурних вимог має на меті досягнення наступних цілей:

– Опис необхідних змін в компонентах архітектури. Це може означати додавання нових компонентів, видалення застарілих, заміну чи вдосконалення компонентів або зміну способу їх організації та способу їх спільної роботи.

– Включення до опису ПА міркувань або мотивів для зміни. Необхідно пояснити, чому існуючі компоненти є неадекватними, обмежують або стримують процес розробки. Встановити, які проблеми, питання чи занепокоєння викликані поточною архітектурою.

– Окреслити доступні варіанти для майбутніх архітектур, які вирішують усі проблеми. Отримаємо відповідь на запитання, як альтернативні цільові архітектури усувають проблеми поточної архітектури.

– Пояснення того, які саме переваги, цінності, ризики, витрати, можливості, обмеження має поточний варіант ПА та майбутні варіанти системи, пов’язані з кожною альтернативою.

– Пропозиція списку альтернативні рішення для усунення прогалин і переходу від поточної до цільової архітектури. Отримаємо відповідь на питання, як ми здійснимо перехід або трансформацію від того, що маємо зараз, до того, що нам потрібно в майбутньому.

3.2 Ідентифікація архітектурних стилів

Архітектурні конструкції та стратегії координації елементів архітектури розробляються для задоволення визначальних архітектурних вимог. Альтернативні архітектурні рішення можуть бути запропоновані та проаналізовані для визначення оптимального рішення для продукту або лінійки продуктів, що розробляються. Якщо йдеться про лінійку продуктів, то здійснюється адаптація архітектури цієї лінійки до конкретних вимог продукту або повна розробка архітектури для окремого продукту. Ідентифікація стилів

архітектури програмного забезпечення спрямована на точне визначення пов’язаних елементів, форм і обґрунтувань:

– Елементи. Є три класи елементів програмного забезпечення, а саме елементи обробки, елементи даних і з’єднувальні елементи. Елементи обробки – це ті компоненти, які беруть деякі дані та застосовують до них перетворення, а також можуть генерувати оновлені або нові дані. Елементи даних – це ті, що містять інформацію, яку потрібно використовувати, трансформувати та маніпулювати нею. З’єднувальні елементи пов’язують опис архітектури разом, забезпечуючи комунікаційні зв’язки між іншими компонентами. З'єднувальні елементи самі по собі можуть бути елементами обробки даних, наприклад, виклики процедур або повідомленнями.

– Форми. Архітектурна форма складається із зважених властивостей і зв’язків. Визначення передбачає, що кожен компонент архітектури буде характеризуватися певними обмеженнями, які зазвичай встановлюються архітектором, і деяким видом зв’язку з одним або кількома іншими компонентами. Властивості визначають обмеження на елементи програмного забезпечення до ступеня, вказаного архітектором.

– Обґрунтування вибраного стилю: обґрунтування пояснює різні архітектурні рішення та їх вибір; наприклад, чому було обрано певний архітектурний стиль, елемент чи форму. Обґрунтування пов’язане з вимогами, архітектурними поданнями та зацікавленими сторонами. Ймовірно, всі вибори регулюються вимогами. Є багато різних зовнішніх компонентів, які зацікавлені в системі та очікують різних речей від однієї системи. Тому ми повинні враховувати різні зовнішні вимоги та очікування, які впливають на архітектуру та її еволюцію.

3.3 Оцінка програмної архітектури

Оцінка архітектури програмного забезпечення передбачає початковий вибір методів оцінювання, коли та які методи оцінки архітектури підходять.

Потім результати такої оцінки аналізуються, визначаються та вживаються заходи для покращення розроблюваної архітектури. Формальна оцінка архітектури програмного забезпечення має бути стандартною частиною методології архітектури програмного забезпечення в гнучких середовищах. Оцінка архітектури програмного забезпечення є економічно ефективним способом пом'якшення значних ризиків, пов'язаних із цим надзвичайно важливим артефактом.

Досягнення потрібного рівня якості програмної системи залежить набагато більше від архітектури програмного забезпечення, ніж від проблем, пов'язаних з кодом, таких як вибір мови, детальний дизайн, алгоритми, структури даних, тестування тощо. Більшість складних програмних систем повинні бути модифікованими та мати хорошу продуктивність. Вони також можуть бути безпечними, сумісними, портативними та надійними. У літературі існує кілька методів оцінки архітектури програмного забезпечення; Метод компромісного аналізу архітектури (ATAM) [21], Метод аналізу архітектури програмного забезпечення (SAAM) [22], метод на основі модифікованого МАІ (MAHP) [4], [23].

3.4 Визначення відповідності архітектури вимогам

Перед вибором архітектури розробники визначають, скільки проєктних рішень системи має бути реалізовано архітектурою системи. Ця сфера розмежовує діяльність розробників програм, дозволяючи їм зосередитися на тому, що вони роблять найкраще. Обсяг архітектури програмного забезпечення є відображенням системних вимог і компромісів, зроблених для їх задоволення. Можливі фактори визначення обсягу вимог для реалізації в архітектурі включають:

- Продуктивність.
- Сумісність із застарілим програмним забезпеченням.
- Повторне використання програмного забезпечення.

– Спосіб дистрибуції (поточний і майбутній).
– Безпека, захищеність, відмовостійкість, можливість внесення змін.
– Зміни в алгоритмах обробки або поданні даних.
– Зміни в структурі/функціоналі.

3.5 Опис програмної архітектури

Архітектура має бути описана досить детально та в легкодоступній формі для розробників та інших зацікавлених сторін. Архітектура є одним із основних механізмів, які дозволяють зацікавленим сторонам спілкуватися про властивості системи. Документація архітектури визначає, які види програмного забезпечення є корисними для зацікавлених сторін, необхідну кількість деталей і спосіб ефективного представлення інформації.

Гнучкі методи повністю узгоджуються з центральним пунктом: «Якщо інформація не потрібна, не документуйте її». Уся документація має мати на увазі цільове використання та аудиторію, а також бути створена таким чином, щоб служити обом. Одним із основоположних принципів технічної документації є «Пиши для читача». Ще одна головна ідея, про яку слід пам’ятати, полягає в тому, що документація не є монолітною діяльністю, яка стримує інший прогрес, доки вона не буде завершена. Маючи це на увазі, нижче пропонується підхід для опису архітектури програмного забезпечення з використанням гнучких принципів:

– Створити каркасний документ (схему документа) для комплексного документування архітектури програмного забезпечення на основі подання, використовуючи стандартні організаційні схеми.

– Вирішити, які архітектурні подання потрібно створити, враховуючи область архітектури програмного забезпечення щодо доступних ресурсів.

– До кожного розділу плану додати перелік зацікавлених сторін, яким інформація, що міститься в ньому, має бути корисною.

– Розставити пріоритети для завершення решти розділів. Наприклад, якщо до складу розділу входять зацікавлені сторони, для яких особиста розмова є недоцільною або неможливою, цей розділ потрібно буде заповнити. Якщо він включає лише таких зацікавлених сторін, його заповнення можна відкласти до завершення етапу розробки архітектури програмного забезпечення та проєктування.

3.6 Інтеграція архітектури програмного забезпечення

Процес інтеграції архітектури програмного забезпечення – це набір процедур, які використовуються для об’єднання компонентів архітектури програмного забезпечення в більші компоненти, підсистеми або остаточну архітектуру програмного забезпечення. Інтеграція архітектури програмного забезпечення дозволяє організації спостерігати за всіма важливими атрибутами, які матиме програмне забезпечення; функціональність, якість і продуктивність.

Це особливо важливо для програмних систем, оскільки інтеграція є першим випадком, де можна спостерігати повний результат зусиль розробки програмного забезпечення. Отже, діяльність з інтеграції є надзвичайно важливою частиною процесу розробки програмного продукту в гнучких проєктах.

Зазвичай для створення інтегрованих програмно-залежних систем використовується мова аналізу архітектури та проєктування. Ця мова призначена для специфікації, аналізу, автоматизованої інтеграції та генерації коду програмного забезпечення, що має критичне значення для продуктивності (час, безпека, відмовостійкість, безпека тощо). Це дозволяє аналізувати дизайн системи (і систему в цілому) до розробки та підтримує підхід до розробки, заснований на моделі, протягом усього життєвого циклу розробки програмного забезпечення. Під час інтеграції архітектури програмного забезпечення архітектор перевіряє, чи є повними моделі, надані розробниками компонентів, особами, які не погоджуються з системою, і експертами в предметній області, а

також його чи її власна модель складання компонентів. Якщо значення відсутні, архітектор програмного забезпечення оцінює їх або зв'язується з відповідальною особою. Результатом цього кроку (інтеграція архітектури програмного забезпечення) є загальна модель із анотацією якості.

3.7 Постійне архітектурне вдосконалення

Архітектурне вдосконалення має на меті допомогти забезпечити ступінь архітектурної стабільності, необхідний для підтримки наступних ітерацій розробки. Ця стабільність особливо важлива для успішної роботи кількох паралельних команд Scrum. Відображення архітектурних залежностей дає змогу керувати ними та узгоджувати команди між собою. Удосконалення архітектури підтримує відокремлення команди, необхідне для прийняття незалежних рішень і зменшення витрат на зв'язок і координацію.

На етапі підготовки Agile-команди визначають архітектурний стиль інфраструктури, достатній для підтримки розробки функцій у найближчому майбутньому. Архітектурне вдосконалення є одним із ключових факторів успішного масштабування Agile. Опис і підтримка (шляхом удосконалення) архітектурного дизайну програмного забезпечення забезпечує системну інфраструктуру, достатню для включення високопріоритетних функцій із беклогу продукту.

Запропонована методологія архітектури програмного забезпечення в гнучких середовищах дозволяє архітектурі та дизайну програмного забезпечення підтримувати функції без потенційного створення непередбаченої переробки через дестабілізуючий рефакторинг. Більші програмні системи (і команди) потребують більш тривалого вдосконалення архітектури. Створення та перебудова програмного забезпечення займає більше часу, ніж одна ітерація чи цикл релізу. Надання запланованої функціональності є більш передбачуваним, коли архітектура для нових функцій уже існує. Для цього потрібно дивитися наперед у процесі планування та інвестувати в архітектуру, включаючи

проєктування в поточну ітерацію, яка підтримуватиме майбутні функції та потреби клієнтів. Архітектурне доопрацювання не завершено. Процес удосконалення навмисно не завершений через невизначене майбутнє зі зміною технологічних орієнтацій та розробки вимог. Це вимагає постійного вдосконалення архітектури для підтримки команд розробників.

3.8 Зв'язок між вимогами до програмного продукту та архітектурними активностями в Agile-проєктах

Різні гнучкі методи охоплюють різні фази життєвого циклу розробки програмного забезпечення. Однак жоден із них не охоплює етап проєктування архітектури програмного забезпечення. Крім того, була відсутня раціоналізація охоплених фаз. Виникає питання: коли гнучкий метод вигідніший, щоб охоплювати більше і бути більш обширним, чи охоплювати менше і бути більш точним і конкретним. З одного боку, деякі гнучкі методи, які охоплюють занадто багато питань, тобто всі фази та ситуації, є занадто загальними або поверхневими для використання. З іншого боку, гнучкі методи, які охоплюють занадто мало (наприклад, одну фазу), можуть бути занадто обмеженими або не мати зв’язку з іншими методами.

Під час остаточного аналізу було зрозуміло, що «повнота» є елементом, пов’язаним як з вертикальним (тобто рівень деталізації), так і горизонтальним (тобто охоплення життєвого циклу) вимірами. Жоден із існуючих гнучких методів не задовольняв вимогам повноти. Практики та експерти все ще борються з частковими рішеннями проблем, які охоплюють ширшу область, ніж гнучкі методи.

Важливою особливістю гнучких методів є те, що вони не припускають, що існує послідовний процес, коли очікується, що кожна фаза життєвого циклу розробки програмного забезпечення буде завершена перед переходом до наступної, як, наприклад, у класичному каскадному процесі. Таким чином, очікується, що етапи розробки вимог або архітектури програмного забезпечення

відбуваються не один раз, а постійно розподіляються в процесі розробки. Коли є перший, як правило, неповний набір вимог, архітектор приступає до архітектурного проєкту.

У той час як етапи розробки вимог і етапи архітектурної діяльності чергуються в традиційних процесах, модель подвійного піку в гнучких проєктах підкреслює, що ці дві дії повинні виконуватися паралельно, щоб підтримувати негайний безперервний зворотний зв’язок від одного до іншого (рис. 3.2) [24].

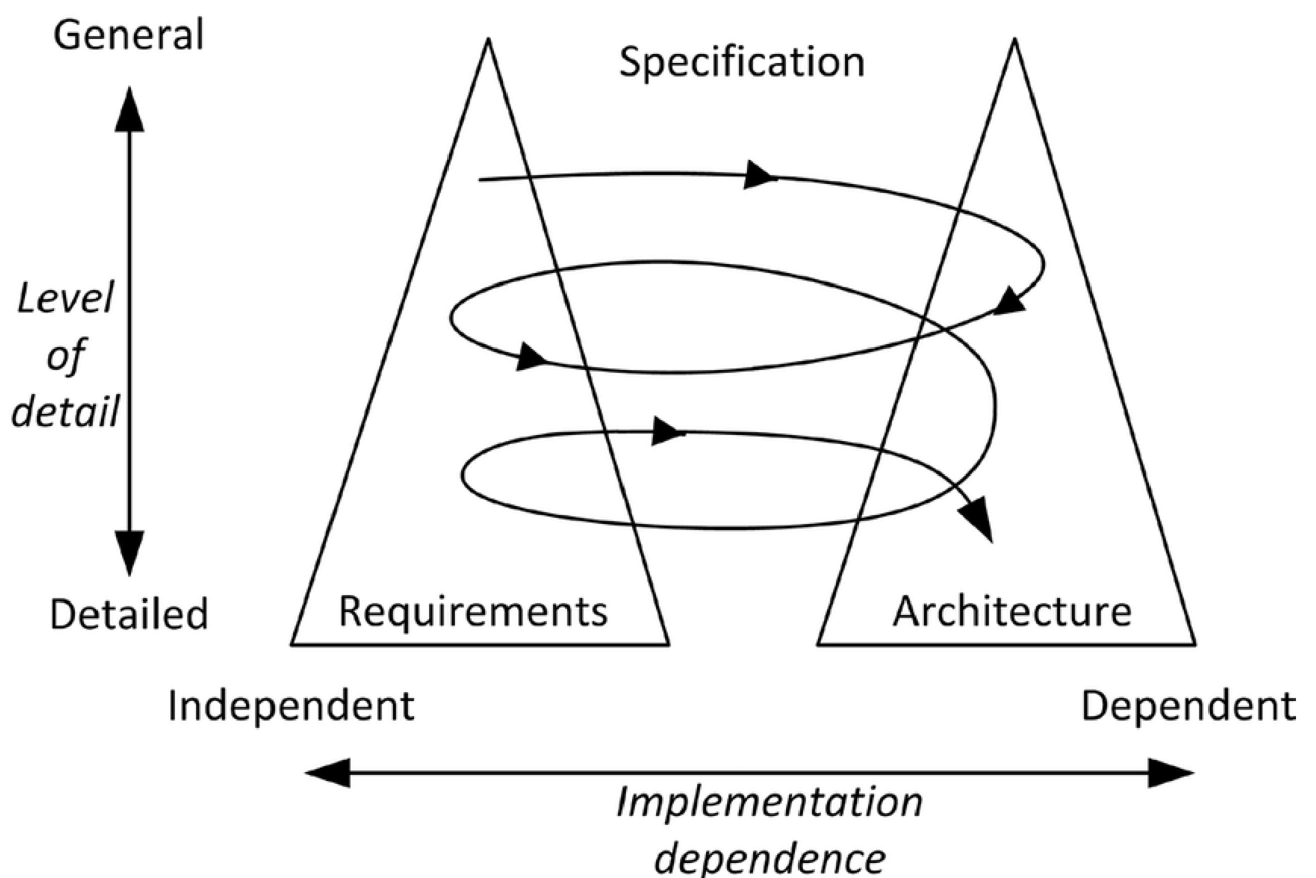

Рисунок 3.2 – Модель подвійного піку

Мета цього процесу полягає в тому, щоб аналітики вимог і архітектори програмного забезпечення краще розуміли проблеми, усвідомлюючи вимоги та їх пріоритезацію, з одного боку, і архітектуру та, зокрема, архітектурні обмеження, з іншого боку. Крім того, можливість швидкого перемикання між проблемою, яку потрібно вирішити (вимоги) та її рішенням (архітектура), може

допомогти чіткіше розрізнити обидва та уникнути змішування проблеми та рішення вже на етапі розробки вимог.

Організація команди в її найпростішому варіанті для гнучкого середовища розробки складається з однієї спільно розташованої багатофункціональної команди з навичками, повноваження та знаннями, необхідними для визначення вимог та архітектора, команди тестування системи. У міру того, як програмне забезпечення зростає в розмірах і складності, однокомандна модель може більше не відповідати вимогам розробки.

Ряд різних стратегій можна використовувати для розширення загальної організації розробки програмного забезпечення, зберігаючи гнучкий підхід до розробки. Одним із підходів є реплікація, по суті, створення кількох команд Scrum з однаковою структурою та відповідальністю, достатніми для виконання необхідного обсягу роботи. Деякі організації масштабують Scrum за допомогою гібридного підходу. Гібридний підхід передбачає реплікацію команди Scrum, але також доповнює міжфункціональні команди традиційними функціонально орієнтованими командами.

Прикладом може бути використання команди інтеграції та тестування для об’єднання та перевірки коду в кількох командах Scrum.

4 ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ

4.1 Вплив факторів трудового середовища на здоров'я та працездатність розробника програм

Безпека працівника нерозривно пов'язана з оточуючим її виробничим середовищем. Останнє характеризується породжуваними діяльністю людини об'єктами, явищами, фізичними, хімічними, біологічними та соціальними факторами, які прямо чи опосередковано впливають на самопочуття та стан здоров'я працюючих [25].

Людини може бути у безпеці тільки в такому стані виробничого середовища, коли виключена дія на неї небезпечних та шкідливих чинників.

Існує класифікація небезпечних та шкідливих факторів, яка розроблена для виробничих умов. Згідно з цією класифікацією небезпечні та шкідливі фактори за природою дії підрозділяються на 4 групи: фізичні, хімічні, біологічні та психофізіологічні [25], [26].

Будь-які фізіологічні, фізичні, хімічні чи емоційні впливи, будь то температура повітря, зміна атмосферного тиску або хвилювання, радість, сум можуть бути приводом до виходу організму зі стану динамічної рівноваги. Автоматично, на основі єдності різних механізмів регуляції здійснюється саморегуляція фізіологічних функцій, що забезпечує підтримку життєдіяльності організму на постійному рівні. При малих рівнях впливу подразника людина просто сприймає інформацію, що надходить ззовні. Вона бачить навколишній світ, чує його звуки, вдихає аромат різних запахів, сприймає дотиком і використовує у своїх цілях вплив багатьох факторів. При високих рівнях впливу виявляються небажані біологічні ефекти. Компенсація змін факторів довкілля виявляється можливою завдяки активації систем, відповідальних за адаптацію (пристосування).

Напруженість праці відображає навантаження на центральну нервову систему, психічні функції, характеризується обсягом сприйманої інформації,

щільністю сигналів, що надходять, станом аналізаторних систем, рівнем емоційної напруги і визначається ступенем напруги уваги. За цим показником працю поділяють на 4 групи (табл. 4.1)

Таблиця 4.1 – Класифікація робіт за напруженістю праці

Ступінь напруженості праці	Концентрація уваги
ненапружений	25 % часу роботи
мало напружений	50 % часу роботи
напружений	75 % часу роботи
дуже напружений	Більше 75 % часу роботи

За фізіологією, праця підрозділяється на:

– динамічну м'язову роботу, при якій м'язи різних м'язових груп поперемінно розтягуються і скорочуються (наприклад, при обертанні кривошипних рукояток);

– статичну м'язову роботу, при якій м'язи не рухаються.

При статичній роботі м'язи недостатньо поповнюються живильними речовинами, які переносяться кров'ю, і не звільняються від продуктів розпаду, що виникають при обміні речовин в організмі людини; це викликає хворобливе відчуття в м'язах і фізичну утому. Напруга при статичній роботі в 5 разів перевищує напругу, викликувану динамічною. При статичній роботі потрібно в 3 – 4 рази більше часу на відновлення енергії. Статична робота менш ефективна. При роботі в положенні стоячи ряд м'язів перебуває в постійній напрузі. При статичній роботі з навантаженням великої групи м'язів необхідно регулярно вводити перерви на відпочинок.

Основні принципи використання статичної роботи:

– статичне навантаження, що виникає при маніпулюванні органами керування, не повинні перевищувати 15 % максимального зусилля відповідної кінцівки при даній робочій позі оператора;

– при зусиллі перевищуючому 25% максимального зусилля, фізична утома спостерігається через 5 хв., а при зусиллі перевищуючому 50% максимального зусилля, м'язи витримують статичну напругу не більш 1 хв.

– робоче місце і робочі рухи повинні вибиратися таким чином, щоб обмежити статичну роботу до можливого мінімуму.

Для цього необхідно:

– обмежити до мінімуму виконання роботи в незручному положенні тіла або кінцівок;

– виключити виконання робіт у перебігу тривалого періоду часу в положенні руки розведені в сторони, підняті нагору, витягнуті вперед;

– обмежити тривалість утримання інструменту, матеріалу або перенесення вантажу;

– обмежити випадки збереження нерухомого положення тіла при виконанні робіт або дуже повільних робочих рухів руками.

Монотонна праця – це праця одноманітна, потребуюча від людини тривалого виконання однотипних простих операцій (монотонність дії) або безперервної концентрації уваги в умовах надходження малого обсягу професійно значимої інформації (монотонність обстановки).

При монотонній праці а організмі людини може розвитися комплекс фізіологічних і психологічних змін, відомий як стан монотонії. При виникненні стану монотонії – знижується продуктивність праці:

– збільшується брак продукції;

– зростає можливість прийняття невірних рішень; - одержання виробничих травм.

У результаті зменшується надійність людини, притупляється його пильність з можливими тяжкими наслідками в таких професіях як водії транспортних засобів, оператори пультів керування в енергетичній і хімічній промисловості, диспетчери аеропортів.

Серед факторів, що перешкоджають розвиткові монотонії, одне з ведучих місць займає ступінь функціональної робочої напруги, що включає.

– величину м'язових зусиль,

– темп роботи,

– ступінь її точності,

– наявність примусового темпу, ступенем складності і відповідальності, - рівень нервово-емоційної напруги.

Чим більше фізична чи тяжкість нервова напруженість праці, тим у меншому ступені монотонна, одноманітна праця приводить до розвитку стану монотонії.

До факторів, що сприяють розвиткові стану монотонії, відносяться:

– гіпокінезія, низька відповідальність

– фактори навколишнього оточення: постійний фоновий шум і вібрація, недостатнє освітлення, некомфортний мікроклімат, замкнутість робочого простору й одноманітність оформлення інтер'єру виробничих приміщень.

Стосовно монотонної діяльності люди поділяються на дві групи: монотофілів і монотофобів.

Для монотофілів характерні слабкий тип нервової діяльності, інертні нервові процеси, низькі показники по шкалі інтроверсії - екстраверсії, замкнутість, низький рівень нейротизму, низька тривожність. Монотофіли стійкі до розвитку монотонії, можуть виконувати монотонну роботу протягом тривалого часу .

Монотофоби володіють сильними процесами збудження, високою рухливістю нервових процесів, вираженої екстраверсію, високий рівень нейротизму, емоційну нестійкість, високу тривожність. Монотофоби схильні до розвитку монотонії при виконанні монотонної роботи.

4.2 Шкідливий вплив іонізуючого випромінювання

Іонізуючі випромінювання знаходять широке використання в різних галузях промисловості. Їх використовують для автоматичного контролю технологічних процесів, контролю якості виробів, зварних швів, структури металів тощо [27].

Для виробництва електроенергії на атомних електростанціях необхідне ядерне пальне, виробництво якого, починаючи від добування уранової руди і закінчуючи виготовленням та транспортуванням паливних елементів, призводить до опромінення персоналу. Незначні додаткові дози опромінення працівники отримують від таких техногенних джерел, як теплові електростанції (підвищена активність їх відходів та аерозолів), підприємств, які пов'язані з видобуванням та переробкою корисних копалин, а також різноманітних приладів та обладнання з джерелами випромінювання, що знаходять широке використання у промисловості і сільськогосподарському виробництві.

Основним документом, що встановлює радіаційно-гігієнічні регламенти для забезпечення прийнятих рівнів опромінення, є Норми радіаційної безпеки України (НРБУ-97).

НРБУ-97 регламентують опромінення людини джерелами іонізуючого випромінювання в умовах:

– нормальної експлуатації індустріальних джерел іонізуючого випромінювання;

– медичної практики;

– радіаційних аварій;

– опромінення техногенно-підсиленими джерелами природного походження.

Відповідно до цього НРБУ-97 встановлено чотири групи радіаційно-гігієнічних регламентів:

– перша – обмежує опромінення від ядерно-радіаційних об'єктів;

– друга – обмежує опромінення людей від медичних джерел;

– третя – обмежує опромінення в умовах радіаційних аварій;

– четверта – обмежує опромінення від техногенно підсилених джерел природного походження.

Враховуючи різнобічні наслідки опромінення людей іонізуючим випромінюванням, їх нормування здійснюється залежно від категорії людей, що опромінюються, а також від чутливості органів тіла людини, на які діє іонізуюче випромінювання.

Виділяють наступні категорії:

А – особи з числа персоналу, які постійно чи тимчасово працюють безпосередньо з джерелами іонізуючого випромінювання;

Б – особи з числа персоналу, які безпосередньо не зайняті роботою з джерелами іонізуючого випромінювання, але у зв’язку з розташування робочих місць в приміщеннях та на промислових майданчиках об’єктів з радіаційно-ядерними технологіями можуть отримувати додаткове опромінення; В – все населення.

Частину населення, яке за своїми статевовіковими, соціально-професійними умовами, місцем проживання та іншими ознаками може отримувати найбільші рівні опромінення від даного джерела, прийнято виділяти як критичну групу.

Для осіб категорій А і Б НРБУ-97 встановлюються ліміти річних ефективних доз зовнішнього опромінення, а також ліміти річних еквівалентних доз зовнішнього опромінення окремих органів і тканин людини. Аналогічні ліміти вводяться і для критичних груп осіб категорії В. Ліміти дози опромінення наведені в табл. 4.2.

Є також обмеження стосовно швидкості накопичення дози для жінок дітородного віку та вагітних жінок, підвищеного опромінення в непередбачуваних ситуаціях та інші.

Крім лімітів дози опромінення, встановлюють допустимі рівні (ДР): потужності дози зовнішнього опромінення, забруднення поверхонь,

надходження радіонукліду через органи дихання тощо, які визначають виходячи із наведених лімітів дози опромінення.

Таблиця 4.2 – Ліміти дози опромінення (мЗв/рік)

Назва лімітів	Категорія осіб, які зазнають опромінення		
	А	Б	В
ЛД$_E$ (ліміт ефективної дози)	20*	2	1
Ліміт еквівалентної дози зовнішнього опромінення: ***ЛД***$_{lens}$ (для кришталика ока) ***ЛД***$_{skin}$ (для шкіри) ***ЛД***$_{e[trim}$ (для кистей та стіп)	 150 500 500	 15 50 50	 15 50 -

З метою зниження рівнів опромінення населення Міністерство охорони здоров'я України запроваджує рекомендовані рівні медичного опромінення. При проведенні профілактичного обстеження населення річна ефективна доза не повинна перевищувати 1 мЗв.

Медичне опромінення – це опромінення працівників при медичних обстеженнях чи лікуванні. Опромінення повинно бути обґрунтованим і призначеним тільки лікарем для досягнення корисних діагностичних та терапевтичних ефектів, які неможливо отримати іншими методами діагностики та лікування.

Рекомендовані рівні медичного опромінення та детальні вимоги до обмеження і контролю за опроміненням пацієнтів регламентуються окремими спеціальними документами Міністерства охорони здоров'я України. При проведенні профілактичного медичного обстеження працівників річна ефективна доза не повинна перевищувати 1 мЗв.

Для радіометричного і дозиметричного контролю використовуються: дозиметри – для вимірювання зовнішніх потоків радіоактивного випромінювання; радіометри – для вимірювання рівнів забруднення навколишнього середовища; індивідуальні дозиметри – для індивідуального контролю.

Серед індивідуальних дозиметрів найбільше розповсюджені прилади, в яких використовують іонізаційні (за величиною іонізації середовища, через яке пройшло випромінювання) та фотографічні (за величиною опромінення фотографічної плівки іонізуючим випромінюванням) методи виміру.

У приладах для контролю потужності дози випромінювання широко застосовують іонізаційний та сцинтиляційний методи (за інтенсивністю світлових спалахів, що виникають внаслідок люмінесценції в деяких речовинах під час проходження через них іонізуючих випромінювань).

При роботі з джерелами іонізуючих випромінювань здійснюють контроль і оцінку параметрів радіаційного фактора відповідно до НРБУ-97. При дотриманні контрольних рівнів умови праці на даному робочому місці оцінюються як допустимі. У разі їх перевищення оцінка шкідливості та небезпечності за радіаційним фактором здійснюється органами Держсанепіднагляду.

Засоби та заходи захисту від іонізуючих випромінювань поділяють на організаційні, технічні, санітарно-гігієнічні та лікувально-профілактичні.

Як правило, ефективний захист від іонізуючого випромінювання досягається при одночасному комплексному використанні зазначених заходів та засобів. При їх виборі враховуються особливості джерел випромінювання. Так, основними заходами, направленими на захист від альфа- та бета-випромінювань, є заходи, що націлені на недопущення накопичення альфа- і бета-активних ізотопів в організмі людини та забруднення шкіри: використання спеціального одягу та взуття, протипилових респіраторів, обезпилення повітря, вологе прибирання помешкань, недопущення вживання радіоактивно забруднених харчових продуктів, води та інші. При роботі з джерелами гама- та рентгенівського випромінювання захист персоналу досягається шляхом зниження активності джерел випромінювання, обмеження часу роботи з ними, збільшення відстані до джерел, екранування джерела іонізуючого випромінювання або зони знаходження людини.

ВИСНОВКИ

У кваліфікаційній роботі магістра виконано огляд питань, пов'язаних з проєктуванням архітектури програмного забезпечення в гнучкому середовищі, і запропоновано методологію для застосування та допомоги практикам, які застосовують гнучкий дизайн програмного забезпечення в Agile-середовищах. Пропонована методологія базується на семи процесах, а саме:

1) Визначення архітектурних вимог.

2) Ідентифікація архітектурних стилів програмного забезпечення.

3) Оцінка архітектури програмного забезпечення.

4) Визначення відповідності архітектури вимогам.

5) Опис архітектури програмного забезпечення.

6) Інтеграція архітектури програмного забезпечення.

7) Вдосконалення архітектури.

Гнучкі методи розробки програмного забезпечення викликали широко обговорюються в наукових та професійних соціумах. Однак академічних досліджень на цю тему все ще мало, оскільки більшість існуючих публікацій написані практиками або консультантами. Тим не менш, багато організацій розглядають можливість майбутнього використання або вже застосовують практики, які стверджують, що успішно виконують і постачають програмне забезпечення в більш гнучкій формі.

Підсумовуючи, варто зауважити, що гнучкі методи без раціоналізації охоплюють лише певні фази життєвого циклу. Більшість із них не забезпечували справжньої підтримки архітектурного дизайну програмного забезпечення для управління проєктами. У той час як універсальні рішення мають сильну підтримку у відповідній літературі, емпіричні дані щодо їх адаптації та використання в гнучких середовищах наразі дуже обмежені.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. K. Beck, M. Beedle, A. Van Bennekum, A. Cockburn, W. Cunning- ham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries et al., “Manifesto for agile software development,” 2001.

2. Kharchenko, A., Raichev, I., Bodnarchuk, I., & Matsiuk, O. (2021, October). The Survey of Global Software Design Processes. In 2021 IEEE 8th International Conference on Problems of Infocommunications, Science and Technology (PIC S&T) (pp. 291-294). IEEE.

3. Боднарчук, І., Харченко, О., Хоміцький, Б., & Шимчук, Г. (2019). Проєктування архітектури програмних систем в проєктах з гнучкими методами управління. Матеріали XXI наукової конференції Тернопільського національного технічного університету імені Івана Пулюя, 46-48.

4. Bodnarchuk, I., Lisovyi, V., Kharchenko, O., & Galai, I. (2018, September). Adaptive Method for Assessment and Selection of Software Architecture in Flexible Techniques of Design. In 2018 IEEE 13th International Scientific and Technical Conference on Computer Sciences and Information Technologies (CSIT) (Vol. 1, pp. 292-297). IEEE.

5. Buchmann, F., Nord, R.L. and Ozakaya, I. (2012) Architectural Tactics to Support Rapid and Agile Stability. Technical Report, DTIC Document.

6. Floyd, C. (1992) Software Development as Reality Construction. In: Software Development and Reality Construction, Springer, 86-100.

7. Edeki, Charles. "Agile software development methodology." European Journal of Mathematics and Computer Science 2.1 (2015).

8. Choudhary, Bharat, and Shanu K. Rakesh. "An approach using agile method for software development." 2016 International Conference on Innovation and Challenges in Cyber Security (ICICCS-INBUSH). IEEE, 2016.

9. Babar, Muhammad Ali, Alan W. Brown, and Ivan Mistrík, eds. Agile software architecture: Aligning agile processes and software architectures. Newnes, 2013.

10. Postma, André, Pierre America, and Jan Gerben Wijnstra. "Component replacement in a long-living architecture: the 3RDBA approach." Proceedings. Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA 2004). IEEE, 2004.

11. Martini, Antonio, Lars Pareto, and Jan Bosch. "Enablers and inhibitors for speed with reuse." Proceedings of the 16th International Software Product Line Conference-Volume 1. 2012.

12. Lata, P. "Agile software development methods." International Journal of Computer Science 20 (2016).

13. Lindstrom, Lowell, and Ron Jeffries. "Extreme programming and agile software development methodologies." IS management handbook. Auerbach Publications, 2003. 531-550.

14. Permana, Putu Adi Guna. "Scrum method implementation in a software development project management." International Journal of Advanced Computer Science and Applications 6.9 (2015): 198-204.

15. Qureshi, M. Rizwan Jameel, and S. A. Hussain. "An adaptive software development process model." Advances in Engineering Software 39.8 (2008): 654-658.

16. Harchenko, A., I. Bodnarchuk, and I. Halay. "Stability of the solutions of the optimization problem of software systems architecture." A. Harchenko, I. Bodnarchuk, I. Halay/Proceeding of VIIth International Scientific and Technical Conference CSIT. 2012.

17. Bengtsson, PerOlof, et al. "Architecture-level modifiability analysis (ALMA)." Journal of Systems and Software 69.1-2 (2004): 129-147.

18. Bass, Len, Paul Clements, and Rick Kazman. "Software Architecture in Practice." (2013).

19. Kruchten, Philippe B. "The 4+ 1 view model of architecture." IEEE software 12.6 (1995): 42-50.

20. Garlan, David. "Software architecture: a roadmap." Proceedings of the Conference on the Future of Software Engineering. 2000.

21. Kazman, Rick, Mark Klein, and Paul Clements. ATAM: Method for architecture evaluation. Pittsburgh, PA: Carnegie Mellon University, Software Engineering Institute, 2000.

22. Kazman, Rick, et al. "SAAM: A method for analyzing the properties of software architectures." Proceedings of 16th International Conference on Software Engineering. IEEE, 1994.

23. Harchenko, Alexandr, Ihor Bodnarchuk, and Iryna Halay. "Decision support system of software architect." 2013 IEEE 7th International Conference on Intelligent Data Acquisition and Advanced Computing Systems (IDAACS). Vol. 1. IEEE, 2013.

24. Salfischberger, Tomas, Inge van de Weerd, and Sjaak Brinkkemper. "The functional architecture framework for organizing high volume requirements management." 2011 Fifth International Workshop on Software Product Management (IWSPM). IEEE, 2011.

25. Державні санітарні норми та правила "Гігієнічна класифікація праці за показниками шкідливості та небезпечності факторів виробничого середовища, важкості та напруженості трудового процесу" // Офіційний вісник України – 2014. – № 41.– с. 95-132.

26. Крушельницька Я. В. К 84 Фізіологія і психологія праці: Підручник. – К.: КНЕУ, 2003. – 367 с.

27. Батлук В.А., Гогіташвілі Г.Г. та ін. Охорона праці в галузі телекомунікацій. – Львів: Афіша, 2003. – 320 с.

28. Методичні рекомендації для проведення атестації робочих місць за умовами праці. Затверджено міністром праці України 1.09.1992 р, постанова № 41.

ДОДАТКИ

УДК 004.41

А. Вивюрка, Л. Мариненко, О. Нога, Б. Хоміцький, Т. Ланевич
(Тернопільський національний технічний університет імені Івана Пулюя, Україна)

ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ПРОЦЕСІВ CI/CD В ГНУЧКИХ ТЕХНОЛОГІЯХ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

A. Vyviurka, L. Marynenko, B. Khomitskyi, O. Noha, T Lanevych

RESEARCH OF THE EFFICIENCY OF CI/CD PROCESSES IN AGILE SOFTWARE DEVELOPMENT TECHNOLOGIES

Процеси безперервної інтеграції (Continuous Integration – CI) та безперервного розгортання (Continuous Deployment – CD) були популяризовані наприкінці 90-х як частина eXtreme Programming [1], що загалом належить до гнучких (Agile) технологій розробки [2]. Поточні IT-стратегії значною мірою залежать від здатності компаній впроваджувати зміни або виправлення гнучким чином та безпомилково. Потреба в автоматизації різко зростає з розвитком технологій, оскільки останні десятиліття методи водоспадної розробки програмного забезпечення були замінені гнучкими підходами до розробки програмного забезпечення.

В даній доповіді пропонується фреймворк для дослідження проєктів з розробки програмних продуктів з метою оцінювання ефективності процесів розробки з впровадженням CI/CD. Для досягнення цієї мети пропонується оцінювати процеси за такими характеристиками.

1. Можливість впровадження технологій Agile в процеси тестування.
2. Ефективність комунікації в команді та в рамках проєкту між командами.
3. Збільшення продуктивності роботи розробників.
4. Збільшення передбачуваності проєкту.

Можливість впровадження гнучких технологій тестування завдяки гнучкій інтеграції розглядалась в роботі [3]. Безперервна інтеграція вважається важливою для підтримки гнучкого тестування. Вважається, що гнучке тестування включає в себе такі практики, як визначені клієнтом приймальні (acceptance) тести, автоматизація цих тестів і їх виконання в наборі регресійних тестів щонайменше щодня, а також розробка модульних (unit) тестів для всього нового коду під час кожної ітерації (спринту) з наступним виконанням цих модульних тестів з кожною збіркою. Задача дослідження полягатиме виявленні того, чи використовуються на проєкті модульні тести з їх автоматичним виконанням.

Задача оцінки ефективності спілкування полягає в тому, щоби при дослідженні проєкту інформація про дефекти автоматично формувалась на надсилалась усім зацікавленим сторонам включно з розробником. Тобто даний пункт досліджень логічно побудований на наявності автоматичних модульних тестів та необхідності застосування процесів інтеграції. Тоді при невдалому завершенні будь-якого тесту формуватиметься відповідний звіт та надсилатиметься тому, хто створив частину програмного коду, котра спричинила збій. Тобто проєктний менеджер чи тестувальник не затрачатиме час на формування таких звітів і їх розсилку. Особливо це актуально, коли над проєктом працює декілька команд, і коли часка затрат комунікацію та узгодження процесів розробки між командами значно зростає.

Безперервна інтеграція сприяє збільшенню продуктивності розробників завдяки очевидній можливості паралельної розробки, тобто роботи декількох програмістів над одним і тим же програмним кодом. Як наслідок, зменшується час на компіляцію та тестування кожним розробником та надає йому можливість впровадити більше нових властивостей (вимог) та змін. Крім того, в роботі [4] розраховано чистий прибуток від

впровадження безперервної інтеграції шляхом вимірювання часу, зекономленого завдяки тому, що розробники не здійснювали компіляцію вручну та не проводили тестування перед кожною операцією коміту, оскільки ці процеси виконувались автоматично в рамках безперервної інтеграції. Це означає, що основна перевага постійної інтеграції полягає в економії часу для розробників.

Можливість точнішої оцінки прогресу проєкту та термінів його завершення відносно проміжних етапів та кінцевого терміну відповідно розглядалась, наприклад, в роботах [5] та [6]. Актуальна інформація про покриття продукту тестами, відсоток успішно виконаних та кількість дефектів дозволяє оперативно вживати управлінські заходи, як то зміна пріоритетів, перерозподіл завдань, залучення необхідних ресурсів, та дотримуватись графіку виконання проєкту. Зокрема, проєктний менеджер матиме змогу виявляти проблеми на ранніх етапах, запроваджувати раннє тестування нефункціональних вимог.

Таким чином, впровадивши даний фреймворк, можна досягнути позитивних результатів завдяки впровадженню процесів CI/CD у Agile-проєктах. Ми дійшли висновку, що існує не одна, а декілька переваг постійної інтеграції. Для кожного проєкту можна виявити взаємозв'язок між безперервною інтеграцією та гнучкими методами тестування автоматизованих тестів клієнта та написанням модульних тестів у поєднанні з новим початковим кодом. При цьому недослідженими залишаються наступні питання. Наприклад, чи підтримує безперервна інтеграція практику модульного тестування, чи, навпаки, вони підтримують один одного. Також важко відокремити вплив безперервної інтеграції на практику автоматизованого α- та β-тестування від контекстних факторів (таких як організаційна структура, культура та доступність клієнтів).

Питання підвищення продуктивності розглянуто в аспекті збільшення кількості завдань, виконаних розробником. Проте його продуктивність в контексті обсягу написаного програмного коду залишається незмінною. Тобто економія часу не зовсім очевидна в кожному проєкті. Звичайно, ефекти, які обговорюються в цій статті, не становлять вичерпний перелік переваг постійної інтеграції. Однак можна стверджувати, що краще розуміння потенційних відмінностей у реалізаціях CI/CD-процесів та їхніх наслідків може допомогти проєктам сформувати свою безперервну інтеграцію таким чином, щоб оптимізувати переваги, яких вони прагнуть досягнути.

Література

1. Beck, K. (2000). Extreme Programming Explained Addison-Wesley. Reading.

2. Agile alliance: manifesto for agile software development. Available at: http:// agilemanifesto.org [retrieved 10.10.2022].

3. Stolberg, S. (2009, August). Enabling agile testing through continuous integration. In 2009 agile conference (pp. 369-374). IEEE.

4. Miller, A. (2008, August). A hundred days of continuous integration. In Agile 2008 conference (pp. 289-293). IEEE.

5. Goodman, D., & Elbaz, M. (2008, August). "It's Not the Pants, it's the People in the Pants" Learnings from the Gap Agile Transformation What Worked, How We Did it, and What Still Puzzles Us. In Agile 2008 conference (pp. 112-115). IEEE.

6. Liu, H., Li, Z., Zhu, J., Tan, H., & Huang, H. (2009, July). A unified test framework for continuous integration testing of SOA solutions. In 2009 IEEE International Conference on Web Services (pp. 880-887). IEEE.

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ТЕРНОПІЛЬСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ ІМЕНІ ІВАНА ПУЛЮЯ

М А Т Е Р І А Л И

XI НАУКОВО-ТЕХНІЧНОЇ КОНФЕРЕНЦІЇ

«ІНФОРМАЦІЙНІ МОДЕЛІ, СИСТЕМИ ТА ТЕХНОЛОГІЇ»

13-14 грудня 2023 року

ТЕРНОПІЛЬ
2023

В. О. Крушельницький
ІНТЕГРАЦІЯ ТА УПРАВЛІННЯ БІЗНЕС АКАУНТОМ FACEBOOK ТА INSTAGRAM В МОБІЛЬНОМУ ДОДАТКУ НА ПРИКЛАДІ “EASY ACCOUNTING FOR SOCIAL BIZ”
V. O. Krushelnystkyi
INTEGRATION AND MANAGEMENT OF FACEBOOK AND INSTAGRAM BUSINESS ACCOUNTS IN THE MOBILE APPLICATION USING THE EXAMPLE OF "EASY ACCOUNTING FOR SOCIAL BIZ" 205

В. О. Крушельницький
ІНТЕГРАЦІЯ ТА УПРАВЛІННЯ БІЗНЕС АКАУНТОМ FACEBOOK ТА INSTAGRAM В МОБІЛЬНОМУ ДОДАТКУ НА ПРИКЛАДІ “EASY ACCOUNTING FOR SOCIAL BIZ”
V. O. Krushelnystkyi
INTEGRATION AND MANAGEMENT OF FACEBOOK AND INSTAGRAM BUSINESS ACCOUNTS IN THE MOBILE APPLICATION USING THE EXAMPLE OF "EASY ACCOUNTING FOR SOCIAL BIZ" 206

О. Кузьмич
РОЗРОБКА ANDROID-ДОДАТКУ ДЛЯ ГРАВЦІВ НАСТІЛЬНО-РОЛЬОВОЇ ГРИ D&D З ВИКОРИСТАННЯМ XAMARIN ТА SOCKET
O. Kuzmych
DEVELOPMENT OF AN ANDROID APP FOR TABLETOP ROLE-PLAYING GAME D&D USING XAMARIN AND SOCKET 207

Володимир Кухарський; Дмитро Михалик
ІНТЕГРАЦІЯ SQL SERVER REPORTING SERVICES У СУЧАСНУ ІНФРАСТРУКТУРУ АВТОМАТИЗОВАНИХ СИСТЕМ ПРИЙНЯТТЯ РІШЕНЬ.
Volodymyr Kuharsky; Dmytro Mykhalyk
INTEGRATION OF THE SQL SERVER REPORTING SERVICES INTO THE MODERN INFORMATIONAL DECISION-MAKING SYSTEMS INFRASTRUCTURE. 208

Володимир Семенюк, Андрій Вивюркка, Олександр Нога, Богдан Хоміцький, Олександр Кучма
ПОРІВНЯЛЬНЕ ОЦІНЮВАННЯ ПРОГРАМНИХ АРХІТЕКТУР
Volodymyr Semeniuk, Andrii Vyviurka, Oleksandr Noha, Bohdan Khomitskyi, Oleksandr Kuchma
COMPARATIVE EVALUATION OF SOFTWARE ARCHITECTURES 209

А.П.Ландяк, Д. М. Михалик
РОЗРОБКА АВТОМАТИЗОВАНОЇ СИСТЕМИ АНАЛІЗУ ТА ВІЗУАЛІЗАЦІЇ ДАНИХ З ВИКОРИСТАННЯМ ТЕХНОЛОГІЙ ІОТ, JAVA ТА JAVASCRIPT
A.P. Landiak, D. M. Mykhalyk
DEVELOPMENT OF AN AUTOMATED DATA ANALYSIS AND VISUALIZATION SYSTEM USING IOT, JAVA, AND JAVASCRIPT TECHNOLOGIES 211

Володимир Чичук, Леонід Мариненко, Володимир Сенківський, Богдан Хоміцький, Тимофій Ланевич
ОЦІНЮВАННЯ АЛЬТЕРНАТИВНИХ ПРОГРАМНИХ АРХІТЕКТУР МЕТОДОМ АНАЛІЗУ ІЄРАРХІЙ
Volodymyr Chychuk, Leonid Marynenko, Volodymyr Senkivskyi, Bohdan Khomitskyi, Oleksandr Noha, Tymofii Lanevych
EVALUATION OF ALTERNATIVE SOFTWARE ARCHITECTURES WITH ANALYTIC HIERARCHY PROCESS 212

Май А., Мудрик І.
ВИКОРИСТАННЯ ШТУЧНОГО ІНТЕЛЕКТУ ДЛЯ РОЗРОБКИ СИСТЕМИ ВІДЕОСПОСТЕРЕЖЕННЯ З ВИКОРИСТАННЯМ ТЕХНОЛОГІЙ ХМАРНИХ ВЕБ-СЕРВІСІВ AWS
Mai A., Mudryk I.
USING ARTIFICIAL INTELLIGENCE TO DEVELOP A VIDEO SURVEILLANCE SYSTEM USING CLOUD AWS TECHNOLOGIES 215

УДК 004.41

Володимир Семенюк, Андрій Вивюркка, Олександр Нога, Богдан Хоміцький, Олександр Кучма
Тернопільський національний технічний університет імені Івана Пулюя

ПОРІВНЯЛЬНЕ ОЦІНЮВАННЯ ПРОГРАМНИХ АРХІТЕКТУР

Volodymyr Semeniuk, Andrii Vyviurka, Oleksandr Noha, Bohdan Khomitskyi, Oleksandr Kuchma

COMPARATIVE EVALUATION OF SOFTWARE ARCHITECTURES

Якість архітектурного рішення оцінюється за сукупністю критеріїв, і його вибір завжди є компромісом, оскільки для заданого набору функціональних вимог та вимог якості не існує єдиного найкращого рішення і покращення одних показників веде до погіршення інших та навпаки. Тому при виборі архітектурного рішення для ПС необхідно використовувати методи багатокритеріального оцінювання з врахуванням конфліктів між показниками якості та пошуком компромісів.

Існує раннє і пізнє оцінювання архітектур. Раннє оцінювання базується на досвіді розробників та логічному обгрунтуванні, оскільки відсутні артефакти, які дають змогу імітувати роботу ПС. Методи, які реалізують раннє оцінювання, базуються на сценаріях. До цих методів належать наступні: SAAM і ATAM [1].На основі пріоритетів зацікавлених сторін визначаються критерії якості. Для перевірки задоволення кожного атрибута якості розробляється сценарій і проводиться оцінка рівня задоволення даного атрибуту варіантом архітектури.

Метод ATAM подібний до SAAM, але в ньому на основі аналізу сценаріїв для відібраних архітектур проводиться оцінка ризиків задоволення атрибутів якості. Оцінку ризиків проводить група експертів, яка також ранжує альтернативні варіанти за рівнем ризику і визначає так звані точки чутливості у компонентах чи зв'язках архітектури, також аналізуються компроміси між критеріями якості.

Для обгрунтованого вибору рішення в методі SAAM/ATAM вибрані альтернативні архітектури аналізуються на ефективність витрат методом CBAM [2]. Цей метод забезпечує економічний аналіз ПС, яка базується на вибраних в попередніх методах варіантах архітектури та сценаріях моделювання. Експерти призначають оцінки критеріям якості в балах від 1 до 100 і ранжують архітектури за значенням, яке ці архітектурні рішення забезпечують для атрибуту якості. Оцінка кожного варіанта архітектури обчислюється за формулою:

$$B(A_i) = \sum_{j=1,K} \left(Cont_{i,j} \cdot Q_j \right) \; i = \overline{1,n}. \quad (1)$$

Тут $Cont_{ij}$ – вага i-ї архітектури відносно j-го атрибута;

Q_j – пріоритет j-го атрибута.

Метод забезпечує оцінку витрат на реалізацію кожної альтернативи і дає можливість обчислити показник бажаності як відношення прибутку до витрат. На основі отриманих даних проводиться вибір кращого рішення.

Спільним недоліком розглянутих методів є послідовне оцінювання архітектури по одному критерію якості що викликає необхідність кожного разу розробляти новий сценарій і проводити експертне оцінювання ризиків, що робить процес вибору трудомістким і неформалізованим. Тут також неможливо отримати порівняльні оцінки для множини альтернатив.

Подальші дослідження в цій області показали ефективність методу аналізу ієрархій при рішенні цих задач. Однією з перших публікацій по застосуванню МАІ до

оцінювання архітектур ПС по множині показників якості є робота [2]. В ній приведений покроковий алгоритм вирішення задачі оцінювання множини альтернативних архітектур по сукупності показників якості та вибору найкращої архітектури ПС. Застосування методу продемонстроване на конкретному прикладі.

Для подолання недоліку МАІ, пов'язаного з неузгодженістю матриці парних порівнянь при великій кількості альтернатив (n≥7) в роботі пропонується коригувати елементи матриці. А це приводить до неповного використання та перекручування експертних даних, що зменшує правдивість отриманих результатів.. В роботі [3] для коректного використання МАІ у випадку великої кількості альтернатив (n>9) було використано модифікований метод аналізу ієрархій.

В методі МАІ використовується порівняльне оцінювання альтернатив по їх реалізації атрибутів якості. Він дає змогу визначити відносні ваги альтернатив по кожному атрибуту якості і проранжувати їх. За призначеними, зацікавленими сторонами, пріоритетами атрибутів якості обчислюється їх усереднене значення і визначаються ваги альтернатив відносно сукупності атрибутів якості.

Отримані відносні оцінки альтернатив можуть використовуватись для аналізу конфліктів між атрибутами якості і пошуку компромісного рішення.

Пропонований підхід базується на використанні методу аналізу ієрархій з оптимізаційним алгоритмом визначення ваг альтернатив, що дає змогу розширити межі застосування методу на більшу кількість порівнюваних альтернатив (n>9). Також слід дослідити чутливість рішення до зміни пріоритетів критеріїв якості і проаналізувати конфлікти і компроміси між критеріями якості.

Для розширення меж коректного застосування МАІ слід застосувати оптимізаційний метод обчислення (визначення) ваг альтернатив, який базується на моделі мінімізації неузгодженості матриці парних порівнянь [3].

Література

1. Kazman, R. ATAM: Method for Architecture Evaluation [Text] / Rick Kazman, Mark Klein, Paul Clements. – Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, August 2000. – CMU/SEI-2000-TR-004, ADA377385. – 83 p

2. Kazman, R. Quantifying the costs and benefits of architectural decision [Text]/ Kazman, R., Asundi, J., and Klein // Proceedings of the 23rd International Conference on Software Engineering (ICSE), 2001. – pp. 297–306

3. Harchenko Alexandr, Bodnarchuk Ihor, Halay Iryna. Stability of the Solutions of the Optimization Problem of Software Systems Architecture // Proceeding of VIIth International Scientific and Technical Conference CSIT 2012. pp. 47–48, Lviv, 2012