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: Методика забезпечення якості при проектуванні архітектури програмного забезпечення в Agile-проектах

Виконав(ла): студент(ка) 6 курсу, групи СНм-61
спеціальності
112 Комп'ютерні науки
(шифр і назва спеціальності)

_____ Оберванюк Н.-П. Б.
(підпис) (прізвище та ініціали)

Керівник _____ Гром'як Р.С.
(підпис) (прізвище та ініціали)

Нормоконтроль _____ Мацюк О.В.
(підпис) (прізвище та ініціали)

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

Рецензент _____ Загородна Н.В.
(підпис) (прізвище та ініціали)

Тернопіль
2020

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)
«21» вересня 2020 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня Магістр
(назва освітнього ступеня)

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

студенту Оберванюк Назарій-Петро Богданович
(прізвище, ім'я, по батькові)

1. Тема роботи **Методика забезпечення якості при проектуванні архітектури програмного забезпечення в Agile-проектах**

Керівник роботи **к.т.н., доц. Гром'як Р.С.**
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом ректора від «*06*» *листопада* 2020 року № *4/7-825*

2. Термін подання студентом завершеної роботи 23 грудня 2020 р.

3. Вихідні дані до роботи Літературні джерела з тематики роботи

4. Зміст роботи (перелік питань, які потрібно розробити)
ЗМІСТ; ВСТУП; РОЗДІЛ 1 МІСЦЕ АРХІТЕКТОРА ПРОЄКТУ З ІТЕРАЦІЙНОЮ МОДЕЛЛЮ ЖИТТЄВОГО ЦИКЛУ; 1.1 Ролі в SCRUM; 1.2 Роль архітектора програмного забезпечення; 1.3 Пов'язані роботи щодо ролі архітектора у спритному розвитку; 1.4 Методика дослідження; РОЗДІЛ 2 ОГЛЯД ТЕХНОЛОГІЇ SCRUM ТА ПРОЕКТУВАННЯ АРХІТЕКТУРИ У ГНУЧКИХ ПРОЄКТАХ; РОЗДІЛ 3 РЕЗУЛЬТАТИ ДОСЛІДЖЕННЯ; 3.1 Огляд результатів та первинних спостережень; 3.2 Сценарій "Внутрішній архітектор"; 3.3 Сценарій „Зовнішній архітектор"; 3.4 Сценарій "Внутрішні та зовнішні архітектори"; РОЗДІЛ 4 ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ; ВИСНОВКИ; ПЕРЕЛІК ПОСИЛАНЬ; ДОДАТКИ;

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, слайдів)
1. Тема, мета роботи. 2. Наукова новизна отриманих результатів. 3. Постановка задачі. 4. Фреймворк розробки програмної архітектури. 5. Оцінка програмної архітектури. 6. Поєднання Agile та архітектурних практик між собою. 7. Сценарій «Внутрішній архітектор» 8. Сценарій «Зовнішній архітектор». 9. Сценарій "Внутрішні та зовнішні архітектори". 10. Висновки

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Охорона праці	Дмитроца Л.П., доц.		
Безпека в надзвичайних ситуаціях	Стадник І.Я., проф.		

7. Дата видачі завдання 21 вересня 2020 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1.	Ознайомлення з завданням до кваліфікаційної роботи	21.09.20-27.09.20	*Виконано*
2.	Підбір наукових джерел по темі роботи	28.09.20-04.10.20	*Виконано*
3.	Переклад та опрацювання наукових джерел	05.10.20-11.10.20	*Виконано*
	по темі кваліфікаційної роботи		
4.	Виконання дослідження щодо огляду атак на	12.10.20-18.10.20	*Виконано*
	комп'ютерні системи		
5.	Оформлення першого розділу	19.10.20-25.10.20	*Виконано*
6.	Оформлення другого розділу	26.10.20-01.11.20	*Виконано*
7.	Оформлення третього розділу	02.11.20-08.11.20	*Виконано*
8.	Виконання завдання до підрозділу «Охорона праці»	09.11.20-15.11.20	*Виконано*
9.	Виконання завдання до підрозділу «Безпека в	16.11.20-22.11.20	*Виконано*
	надзвичайних ситуаціях»		
10.	Оформлення кваліфікаційної роботи	23.11.20-29.11.20	*Виконано*
11.	Нормоконтроль	30.11.20-05.12.20	*Виконано*
12.	Перевірка на плагіат	05.12.20	*Виконано*
13.	Попередній захист кваліфікаційної роботи	14.12.20	*Виконано*
14.	Захист кваліфікаційної роботи	23.12.2020	

Студент _____ Оберванюк Н.-П. Б.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Гром'як Р.С.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

"Методика забезпечення якості при проектуванні архітектури програмного забезпечення в Agile-проектах" // Оберванюк Назарій-Петро Богданович // Тернопільський національний технічний університет ім. І. Пулюя, факультет комп'ютерно-інформаційних систем і програмної інженерії, кафедра комп'ютерних наук, група СНм-61 // Тернопіль, 2020 // с. – , рис. – , табл. – , джерел – .

Ключові слова: ПРОГРАМНА ІНЖЕНЕРІЯ, ГНУЧКІ ТЕХНОЛОГІЇ, AJILE, SCRUM, АРХІТЕКТУРА, ЯКІСТЬ.

У кваліфікаційній роботі магістра виконано дослідження способів забезпечення якості програмного продукту на ранніх етапах проєктування архітектури та ролі архітектора в команді, що працює над проєктом зва методологією SCRUM.

ANNOTATION

"Method of quality assurance in software architecture design in Agile projects" // Diploma paper of Master degree level // Obervaniuk Nazarii-Petro Romanovytch // Ternopil Ivan Puluj National Technical University, Faculty of Computer Information Systems and Software Engineering, Computer Science Department // Ternopil, 2020 // p. – , Fig. – , Tables – , Refence. – .

Key words: SOFTWARE ENGINEERING, FLEXIBLE TECHNOLOGIES, AGILE, SCRUM, ARCHITECTURE, QUALITY.

A study of the possibilities of quality assurance of the software product in the early stages of architecture design and the role of the software architect in the teams working on the project organized with SCRUM methodology was performed in the master's qualification work.

ЗМІСТ

ВСТУП

Актуальність теми. Архітектура програмного забезпечення має вирішальне значення для програмного проєкту та успіху продукту [1]. Однак архітектурна діяльність та роль архітектора програмного забезпечення часто явно не розглядаються у гнучких підходах до розробки програмного забезпечення (наприклад, SCRUM) [2]. Це дозволяє спритним командам: а) враховувати особливості окремих проєктів; б) не обмежувати розподіл архітектурних обов'язків, а максимально використовувати досвід, навички та досвід окремих членів команди; в) виділяти, що кожен член команди є відповідає за всі заходи, необхідні для успішного реалізації проєкту. Попередні дослідження досліджували, як можна підходити до архітектури програмного забезпечення в гнучких проєктах розвитку, включаючи процеси та практики для проєктування та підтримки гнучких, адаптивних та розвиваються архітектур [3]. Наприклад, було визнано, що в гнучких проєктах архітектура може виникнути під час проєкту, заснованого на метафорі архітектури та "швидкі сеанси проєктування", а не повністю розроблені заздалегідь [4]. Однак наразі мало уваги приділяється ролі архітекторів програмного забезпечення (тобто акторам (процесорам) у процесі розробки програмного забезпечення, які виконують діяльність, пов’язану з архітектурою), і тому, як ця роль (та відповідні завдання та можливості) виглядає в гнучких проєктах розвитку.

У зв'язку з актуальністю теми та важливістю нею займаються ряд науковців, в тому числі і в ТНТУ. Зокрема серед робіт цієї тематики варто згадати такі: [10], [11], [12], [13], [14].

Мета роботи. Метою роботи є проаналізувати функції архітектора в команді розробників ПЗ SCRUM щодо його ролі та взаємодії з іншими ролями в SCRUM в контексті промислового середовища з метою визначення сценаріїв того, як архітектори мають працювати та взаємодіяти командою в умовах виробництва програмних продуктів в практичних командах.

Виходячи з мети дослідження, описаної вище, ми визначили наступні питання дослідження:

– Яка позиція архітектора в проєктах SCRUM?
– Як архітектор взаємодіє з іншими ролями в SCRUM?
– Яка роль архітектора в гнучких проєктах для забезпечення якості?

Об’єкт дослідження: процеси забезпечення якості програмного продукту на ранніх етапах життєвого циклу з ітераційними моделями.

Предмет дослідження: ітераційна (гнучка) модель життєвого циклу SCRUM.

Методи дослідження: Для досягнення мети кваліфікаційної роботи використовувались:

– цільова вибірка (тобто, ми досліджуємо організації, які використовують SCRUM; організації були обрані на основі фактичної практики SCRUM, а не лише на основі їх власних заяв про використання SCRUM), доповнені зручністю вибірки (відбирались випадки на основі їх доступності) [15]. Крім того, досліджувані організації повинні бути представницькими, а не в дуже конкретних сферах. Тому ми відібрали дані з відкритих джерел про шість компаній, які використовують SCRUM і мають усталену практику розробки програмного забезпечення (з міркувань конфіденційності використано лише номери у таблицях);

Наукова новизна отриманих результатів. Наукова новизна полягає у вирішенні задачі керування якістю програмного продукту на етапі проектування архітектури в проєктах з ітераційною моделлю життєвого циклу. Під час виконання роботи були отримані результати:

– систематизовано інформацію про ітераційні моделі життєвого циклу програмних продуктів;

– кваліфікаційна робота дозволяє оптимізувати управління проектом на ранніх етапах з ітеративними методиками для забезпечення якості при проєктуванні архітектури програмного продукту.

Практичне значення отриманих результатів. Основним внеском цієї роботи є емпірично обґрунтована модель ролі архітектора в проєктах SCRUM, включаючи

сценарії, що описують взаємодію архітекторів з іншими ролями в SCRUM. Ці ролі та взаємодії можуть бути відсутні в стандартному SCRUM Guide. Крім того, результати допомагають навчати менш досвідчених архітекторів в SCRUM, пояснюючи їх позицію та стосунки з іншими учасниками процесу розробки. Нарешті, зроблені висновки є вказівками для визначення та розуміння ролей, пов’язаних з архітектурою в SCRUM при створенні нових команд або проєктів.

Апробація результатів та особистий внесок здобувача. Основні положення роботи доповідались, розглядались та обговорювались на науковій конференції Тернопільського національного технічного університету. Результати кваліфікаційної роботи опубліковані у тезах студентської наукової конференції, яка проводились у ТНТУ.

РОЗДІЛ 1
МІСЦЕ АРХІТЕКТОРА ПРОЄКТУ З ІТЕРАЦІЙНОЮ МОДЕЛЛЮ ЖИТТЄВОГО ЦИКЛУ

Найбільш часто використовуваною гнучкою моделлю життєвого циклу розробки програмного продукту є SCRUM [5]. Путівник по SCRUM [1] рекомендує підбирати членів команди з перекриттям виконуваних функціональних обов'язків, але явно не враховує роль архітектора або будь-яку іншу роль, пов'язану із завданням. Крім того, свідчення промисловості свідчать про відсутність явної ролі архітектора в SCRUM [4]. Однак, як обговорювали в [2], настійно рекомендується включати роль архітектора в гнучкі проєкти: "Очікується, що архітектори програмного забезпечення будуть виступати фасилітаторами в цілих проєктах з розробки програмного забезпечення та як представники загальних атрибутів якості системи".

Незважаючи на те, що у багатьох організаціях, які слідують за SCRUM, немає спеціальної ролі архітектора, певні ролі в цих організаціях створюють архітектурні проєкти та повідомляють свої рішення командам розробників [4]. Це означає, що, прийнявши SCRUM, організація не раптом “забуває”, що існує багато різних видів діяльності (включаючи дизайн та архітектуру, пов’язані з діяльністю, забезпечення якості та тестування). У команді SCRUM повинні бути актори, які виконують ці дії. Залежно від типу продукту та зрілості / досвіду / навичок членів команди, ці члени команди можуть мати більший чи менший вплив. Однак детального та систематичного розуміння ролі архітектора в даний час бракує. У цій роботі ми розглядаємо не лише спритні архітектурні практики, але й досліджуємо роль самих архітекторів. Це призводить до наступної мети дослідження:

Ми зосереджуємося на SCRUM, оскільки це найчастіше використовувана гнучка система розробки [5]. Крім того, зосередження на SCRUM допомагає нам здійснити наше дослідження та пояснити застосовність наших результатів. Для досягнення мети нашого дослідження було проведено тематичний огляд у

промисловості [6]. Це тематичне дослідження включало оброблення результатів інтерв’ю з практиками проєктів SCRUM у шести голландських компаніях.

Спочатку обговоримо SCRUM на основі SCRUM Guide, зосереджуючись на ролях у SCRUM. Ми не описуватимемо SCRUM детальніше, оскільки з деталями можна ознайомитись у [7] та [8]. Потім ми обговоримо роль архітекторів загалом. Нарешті, буде виділено суміжні функції архітекторів у гнучкій розробці.

1.1 Ролі в SCRUM

SCRUM-Master піклується про належну реалізацію процесу SCRUM і пов'язаних з ними методів. Він/вона також допомагає усунути будь-які перешкоди, з якими може зіткнутися команда, та керувати ресурсами (програмне забезпечення, обладнання, простір, час тощо). Нарешті, SCRUM-Master захищає команду розробників від небажаних впливів під час спринтів. Команда розробників є самоорганізуючою та багатофункціональною (тобто члени команди розподіляють собі завдання та виконують дії за потреби) і виконує всі заходи з розробки програмного забезпечення, включаючи архітектурні роботи, якщо це потрібно.

Власник продукту (product owner) здійснює всі комунікації між командою і зацікавленими особами поза командою (наприклад, кінцевих користувачів або управління). У цьому сенсі власник продукту захищає команду від зовнішнього середовища. Власник продукту повинен бути експертом у галузі продукту або, принаймні, повинен мати можливість швидко стати експертом. Більшість власників продуктів дбають лише про функціональні вимоги, але не про нефункціональні вимоги та атрибути якості. Як правило, власники продуктів не беруть участі в архітектурних аспектах проєкту, оскільки вони часто є працівниками бізнес-відділу або іншого не-IT-відділу.

1.2 Роль архітектора програмного забезпечення

Як правило, перед архітекторами є три основні завдання: отримати вхідні дані із зовнішнього світу (вислухати зацікавлені сторони, дізнатись про технології тощо), приймати архітектурні рішення щодо декомпозиції систем, вибирати технологічні технології, приймати рішення про архітектурні зразки та стилі тощо та надання інформації (архітектура комунікацій, допомога зацікавленим сторонам тощо). Діяльність архітекторів та необхідні навички зосереджуються на процесах, практиках та технологіях. Грань між розробником та архітектором є тонкою. Однак, на відміну від архітекторів, розробники впроваджують, тестують і підтримують пов'язані з кодом програмні артефакти і витрачають більшу частину свого часу на кодування (щодо інших видів діяльності). Розробник та архітектор не обов'язково відокремлені: це ролі (а не ранги чи посади), тобто одна посада може зайняти більше однієї ролі.

1.3 Пов’язані роботи щодо ролі архітектора у спритному розвитку

На думку автора [9], архітектори повинні активно керувати розробниками, але не домінувати, тобто архітектори повинні приймати відхилення від оригінальних архітектурних проєктів, якщо розробники вимагають (і обґрунтовують) це. Виділяють три типи ролей, пов'язаних з архітектурою, у великих гнучких компаніях (не для SCRUM): головний архітектор (приймає архітектурні рішення високого рівня та керує іншими архітекторами та командами), архітектор управління (роль комунікації між командами та головним архітектором) та архітектор команди (відповідальний за архітектуру в команді). Ці ролі взаємодіють з різними типами команд у великомасштабних проєктах, тобто з функціональними командами (відповідальними за реалізацію функцій у специфікації (backlog) і, отже, найбільш подібними до "команд" у SCRUM), виконавчі командами (відповідальними за "особливість архітектури" та рефакторинг архітектури), архітектурних команд (групи

архітекторів для різних проєктів) та управлінських груп (команди архітекторів та інших осіб, що приймають рішення на високому рівні).

Що стосується ролей спритних команд, Скотт Емблер доводить, що "власники архітектури" у великих командах сприяють архітектурним рішенням підгрупи. У цьому сенсі власники архітектури подібні до головного архітектора або архітектора управління. Поточна робота

– досліджує SCRUM у малих, середніх та великих організаціях, а не лише у великих гнучких організаціях загалом;

– фокусується на ролі (з точки зору завдань та можливостей) архітекторів, які взаємодіють з іншими ролями в SCRUM та командах, а не з іншими ролями в організації загалом.

Деякі фреймворки для SCRUM у більшому масштабі (наприклад, SCRUM @ Scale [3], The Nexus – масштабований професійний фреймворк [4], Large-scale SCRUM Framework [5], “SCRUM of SCRUMs” [6]) включають практики високого рівня, пов’язані з архітектурою, але не обговорюють роль архітектора в процесі розробки. Винятком є SAFe (Scaled Agile Framework) [7], який розглядає “System Architect / Engineer” на рівні програми розробки (на рівні програми команди працюють над спільною місією підприємства). Існують також гнучкі фреймворки "не SCRUM", такі як DSDM (метод розробки динамічних систем) [8], які явно включають ролі, пов'язані з архітектурою (наприклад, «технічний координатор» у DSDM розробляє архітектуру системи).

Поточне дослідження фокусується на тій ролі, яку архітектори реалізують в гнучких командах, а також сценарії для опису взаємодії архітекторів з іншими учасниками проєктів SCRUM для забезпечення якості створюваного програмного продукту.

1.4 Методика дослідження

Ми вивчаємо архітекторів у SCRUM на практиці. Крім того, архітекторів не можна вивчати в залежності від їхнього контексту (наприклад, організації, проєкту), і ми мало контролюємо всі змінні (наприклад, людей, організаційні структури). Тому було застосовано тематичне дослідження. Тематичні дослідження пропонують глибше розуміння завдань архітекторів у проєктах SCRUM та контексту, в якому вони працюють. Дослідження було розроблене на основі рекомендацій, описаних в [6], а його основні методологічні ричи описані далі. Виходячи з мети дослідження, описаної вище, ми визначили наступні питання дослідження (RQ):

- RQ1: Яка позиція архітектора в проєктах SCRUM?
- RQ2: Як архітектор взаємодіє з іншими ролями в SCRUM?

Дослідження, проведене у роботі, являє собою багаторазове тематичне дослідження з шести випадків (таблиця 1). Наше дослідження є дослідницьким, оскільки ми розглядаємо незвідане явище [15]. Наша одиниця аналізу – це архітектор (зокрема, завдання та роль) у проєктах SCRUM.

Підготовка до збору даних. Дані для кожного випадку збирали за допомогою напівструктурованих нотаток з літератури. Усі опитані займали представницькі ролі в своїх організаціях, щоб звітувати про архітектурні практики, проблеми процесів, ролі тощо. Щоб відповісти на наші дослідницькі запитання, ми задавали питання про позицію та роль співрозмовників у проєктах розвитку та про завдання, які вони виконують, використовуючи завдання-описи з [16] та [17]. Для кожного завдання була зібрана інформація про те, чи було і як воно було виконано, чи спостерігались проблеми при його виконанні, які взаємодії відбуватимуться, з якими акторами тощо. Відкриті запитання використовувались для з’ясування додаткової інформації, не охопленої через питання в настанові для співбесіди.

Аналіз зібраних даних: Для кластеризації даних ми використовували відкрите кодування, де один код може бути присвоєний багатьом фрагментам тексту, а один фрагмент тексту може бути призначений більше ніж одному коду [18]. Після

початкового кодування ми розглянули групи кодових фраз та об’єднали їх у концепції та пов’язали з роллю, становищем та взаємодією архітекторів. Коди та поняття виникли під час аналізу і не були визначені заздалегідь. Оскільки дані були зібрані в рамках тематичного дослідження та є контекстно-залежними, ми провели ітераційний аналіз вмісту, щоб зробити висновки зі зібраних даних у його контексті [19]. Аналіз якісних даних інтерв’ю вимагає інтеграції даних, коли різні опитані могли використовувати терміни та поняття з різним значенням або різні терміни та поняття, щоб виразити одне і те ж. Для вирішення цієї проблеми було використано взаємні трансляції.

РОЗДІЛ 2
ОГЛЯД ТЕХНОЛОГІЇ SCRUM ТА ПРОЕКТУВАННЯ АРХІТЕКТУРИ У ГНУЧКИХ ПРОЄКТАХ

Парадигма гнучкого розвитку програмного забезпечення (Agile Software Development – ASD) була широко прийнята сотнями великих і малих компаній, намагаючись зменшити витрати та збільшити їх здатність справлятися зі змінами в динамічних ринкових умовах. На основі принципів в Agile Manifesto , практикуючі Перевірений запропонував кілька методів і підходи, такі як SCRUM, функція керованого розвитку [20], екстремальне програмування [21], а також розробки через тестування. У цьому розділі ми називаємо їх усіма методами ASD. Поки що немає сумнівів в тому, що мало місце багаторазове збільшення в прийнятті методів ASD всіх видів компаній, там завжди був зростаючий скептицизм з приводу надійності, ефективності.

Широко визнано, що програмна архітектура (Software Architecture – SA) може бути ефективним інструментом зменшення витрат і часу на розвиток та еволюцію, а також для підвищення концептуальної цілісності та якості системи [22]. Однак послідовники методів ASD розглядають архітектурно-орієнтовані підходи як частину планової парадигми розвитку. На їх думку, попереднє проектування та оцінка SA як високих церемоніальних заходів, ймовірно, забирає багато часу та зусиль без забезпечення цінних результатів для клієнтів (тобто кодом функцій). Прихильники SA вважають, що не можна дотримуватися розумних архітектурних практик, використовуючи гнучкі підходи.

Можна стверджувати, що така ситуація виникла з двох крайніх поглядів на методи ASD та SA-центричні методи. Прихильники архітектурно-орієнтованих підходів виявляються менш впевненими в тому, що будь-яку програмно-інтенсивну систему значних розмірів можна успішно побудувати та розвивати, не приділяючи достатньої уваги архітектурним питанням, особливо в таких сферах, як автомобільна,

телекомунікаційна, фінансова та медична пристроїв. Прихильники методів ASD, схоже, застосовують інший підхід.

На їх думку, рефакторинг може допомогти вирішити більшість програмно-інтенсивних структурних проблем системи. Стверджувалося, що рефакторинг є вартісним до тих пір, поки дизайн високого рівня достатньо хороший, щоб обмежити потребу у масштабному рефакторингу [23, 24, 25]. І багато практичних прикладів показують, що масштабний рефакторинг часто призводить до значних дефектів, які дуже дорого вартують пізніше у життєвому циклі розробки.

У більшості описів методів ASD дуже мало уваги приділяється загальним заходам архітектурного проектування, таким як архітектурний аналіз, архітектурний синтез, архітектурна оцінка та типи артефактів, пов’язані з цією діяльністю. Більшість методів ASD, як правило, припускають, що архітектурний дизайн – це проект високого рівня без явних структурних сил, таких як атрибути якості.

Рефакторинг є основним методом розвитку архітектури у Agile. Первинна практика поступового проектування другого видання книги [21] стверджує, що архітектура може з’являтися в щоденному дизайні. Новий дизайн означає, що архітектура покладається на пошук потенційно поганих архітектурних рішень у реалізованому коді та покращення архітектури, коли це потрібно за допомогою фабрики. Відповідно до цього підходу, архітектура виникає з коду, а не з якоїсь попередньої структури.

Починається визнавати факт, що обидві дисципліни (тобто методи ASD та архітектурно-орієнтовані підходи) відіграють важливу й додаткову роль у розробці програмного забезпечення та еволюційній діяльності. Хоча методи ASD обіцяють дозволити компаніям досягти ефективності, якості та гнучкості для пристосування змін, критично важливо дотримуватися надійних архітектурних практик для великих проектів з розробки програмного забезпечення. Також зростає визнання важливості приділяти більше уваги архітектурним аспектам у гнучких підходах.

Ця ситуація стимулювала кілька зусиль, спрямованих на виявлення механіки та передумов інтеграції відповідних архітектурно-орієнтованих принципів та практик у

методи ASD. Однією з основних цілей цих зусиль є допомогти практикуючим зрозуміти контекстуальні фактори та причини звернення уваги на роль і значення системного SA при впровадженні методів ASD, дослідники та практики також визначили технічні та організаційні проблеми, пов'язані з інтеграцією підходів Agile у традиційні методи розробки програмного забезпечення [26, 27].

Далі ми коротко описуємо деякі відомі архітектурно-орієнтовані концепції та підходи, а також їх походження та контекст застосовності. Можна стверджувати, що концепції та принципи, пов'язані з SA, описані в цьому розділі, можуть бути адаптовані та інтегровані в методи ASD.

2.1 Архітектура програмного забезпечення

Архітектура програмного забезпечення є важливою підгалуззю програмної інженерії. Хоча важлива роль SA у досягненні цілей щодо якості програмно-інтенсивної системи набула популярності протягом 1990-х, ідея забезпечення якості програмного забезпечення за допомогою дизайнерських рішень високого рівня виникла в 1970-х. Парнас показав, як модуляризацію та приховування інформації можна використовувати як засіб підвищення гнучкості та зрозумілості системи [28]. Незабаром Стівенс та ін. представив ідею зчеплення та згуртованості модулів як характеристик якісного дизайну програмного забезпечення [29]. Однак інженери програмного забезпечення усвідомлювали важливість взаємозв'язку між нефункціональними вимогами (Non Functional Requirements – NFR) та дизайном SA лише на початку 1990-х. Практика використання шаблонів дизайну та архітектурних стилів для виготовлення якісних конструкцій протягом коротких періодів часу дала поштовх для нового інтересу до вирішення питань якості на рівні архітектури.

Архітектура програмного забезпечення може означати різні речі для різних людей. Важко стверджувати, що саме є загальновизнаним визначенням SA в галузі програмного забезпечення [30]. Одне з перших визначень SA було надано Перрі та Вольфом у їх широко цитованій роботі [31]. Вони визначають SA наступним чином:

SA = {Елементи, форма, обґрунтування} (2.1)

Відповідно до цього визначення SA являє собою комбінацію (2.1) набору архітектурних елементів (тобто обробки, передачі даних та з'єднання), а також форми цих елементів як принципів, що визначають взаємозв'язок між елементами та їх властивостями, і обґрунтування вибору елементів та їх форми певним чином. Це визначення послужило основою для початкових досліджень у галузі SA. Недавня тенденція опису SA як сукупності проектних рішень та обґрунтування, що лежить в основі цих проектних рішень, підкреслила важливість обґрунтування у прийнятті та описі дизайнерських рішень [32].

Автори у [33] визначили SA таким чином: архітектура програмного забезпечення системи – це сукупність структур, необхідних для розуміння системи, які включають елементи програмного забезпечення, відносини між ними та властивості обох.

Структури в SA представляють рішення щодо розділення та взаємодії, прийняті для розподілу відповідальності за задоволення вимог між набором компонентів та визначення взаємозв'язку компонентів між собою. Структурне розділення керується конкретними вимогами та обмеженнями програми. Одним з головних міркувань під час прийняття рішення про розділення є створення вільно поєднаної d-архітектури з набору високозв’язних компонентів для мінімізації залежностей між компонентами.

Керуючи непотрібними залежностями, локалізується ефект змін у різних компонентах. Архітектурні структури масштабних програмно-інтенсивних систем вважаються критично важливими для задоволення багатьох NFR. Кожна архітектурна структура може допомогти архітекторам міркувати про різну якість системи та її властивості. Архітектурні споруди документуються з використанням різних архітектурних точок зору.

2.1.1 Процес проєктування архітектури програмного забезпечення та життєвий цикл архітектури

Також важливо добре розуміти процес проєктування SA та так званий життєвий цикл SA. Зазвичай вважається, що дизайн архітектури – це творча діяльність без чітко визначеного процесу. Це можна вважати правильним припущенням для проектування архітектури великої кількості систем. Однак при серйозній спробі розробити та оцінити SA для широкомасштабної складної системи важливо, щоб існував дисциплінований процес, який може підтримати творчість за допомогою більш контрольованого та рефлексивного підходу. Більше того, як і будь-який інший артефакт, SA також має життєвий цикл, який проходить різні етапи та дії. Кожна фаза життєвого циклу архітектури має свої передумови для використання та застосовності.

Для підтримки процесу SA було розроблено та просунуто кілька моделей та методів процесу. Деякі з найвідоміших – це метод проектування на основі атрибутів (ADD) [33], процес та організація бізнес- архітектури [34], Перегляди 4 + 1 Rationale Unified Process [35], Siemens '4 Views [36], та архітектурне відокремлення проблем [37]. Для того, щоб раціоналізувати варіанти, доступні менеджерам проектів програмного забезпечення та архітекторам, розробники п’яти відомих методів проектування архітектури вирішили розробити нову загальну модель дизайну SA.

Оригінальна загальна модель проектування архітектури складалася з трьох видів діяльності. Ця загальна модель була розширена Тангом та його колегами [39] для охоплення архітектурної матеріалізації та еволюції. Кожна з дій у загальній моделі архітектурного проекту коротко описана нижче (див. рисунок 2.1):

1. Аналіз предметної області: Ця діяльність складається з кількох підзаходів та завдань. Ця діяльність спрямована на визначення проблем, що підлягають вирішенню. Деякі з основних видів діяльності можуть бути вивчення архітектурних вимог (або навіть виявлення та уточнення архітектурних вимог), перегляд проблем та контексту зацікавлених сторін, щоб відокремити та визначити пріоритетні архітектурно значущі вимоги (ASR) від тих, що не мають архітектурного значення.

2. Проектування та опис архітектурних рішень: Ця діяльність спрямована на прийняття ключових архітектурних проектних рішень на основі ASR. Архітектор може розглянути кілька доступних варіантів дизайну, перш ніж вибрати ті, які здаються найбільш доцільними та оптимальними. Архітектор також відповідає за документування спроектованої архітектури з використанням відповідних позначень документації та шаблонів.

3. Архітектурна оцінка: Ця діяльність має на меті підтвердити, що архітектурні рішення, обрані під час попереднього процесу, є правильними. Отже, запропоновані архітектурні рішення оцінюються на відповідність ASR.

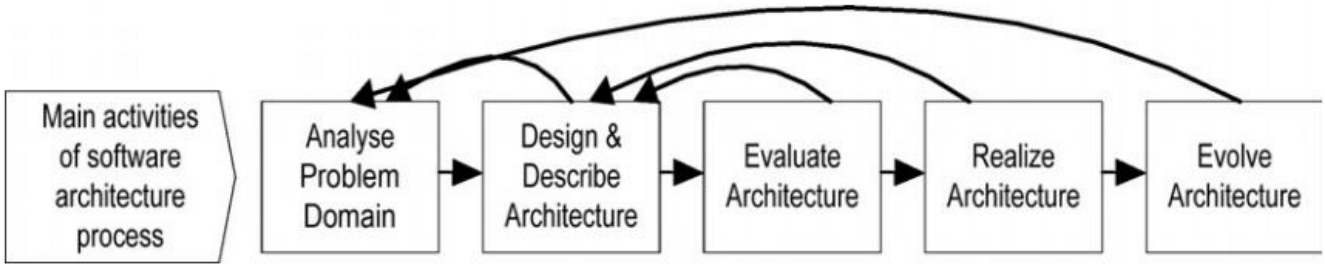

Рисунок 2.1 – Модель процесу архітектури програмного забезпечення.

4. Реалізація архітектури: Це фаза, на якій спроектована архітектура деконструюється в детальний проект і реалізується. На цьому етапі розробники програмного забезпечення приймають кілька десятків рішень, які потрібно узгодити з рішеннями архітектури високого рівня. Це означає, що розробники програмного забезпечення повинні переконатися, що їх рішення відповідають архітектурі, розробленій архітектором.

5. Розвиток архітектури: Це передбачає внесення архітектурних змін у міру розвитку архітектури через вимоги до вдосконалення та обслуговування, які ставлять кілька нових вимог до архітектури, яка лежить в основі системи. З точки зору управління знаннями, попередні проектні рішення переоцінюються на предмет потенційного впливу необхідних змін, і приймаються нові рішення, щоб врахувати необхідні зміни без шкоди архітектурній цілісності.

Слід зазначити, що вищезазначені заходи не йдуть послідовно, як модель водоспаду. Швидше, ці заходи виконуються досить ітеративно / еволюційно, і завдання, пов'язані з однією конкретною діяльністю, можуть бути виконані та / або переглянуті під час виконання будь-якої іншої діяльності. У наступних підрозділах ми коротко обговорюємо різні методи та прийоми, призначені для підтримки процесу SA, описаного в цьому розділі.

2.1.2 Архітектурно значущі вимоги

Вимоги до програмного забезпечення в основному поділяються на функціональні вимоги та NFR. Функціональні вимоги відповідають бажаним особливостям системи; у той час як NFR вказують необхідні властивості системи. Для NFR використовуються різні терміни, такі як атрибути якості, обмеження, цілі та вимоги до поведінки. Нещодавно визнано, що всі ці терміни можна використовувати для ASR, проте ASR можуть включати також функціональні вимоги.

Очевидними прикладами ASR є надійність, модифікація, продуктивність та зручність використання. ASR зазвичай є суб'єктивними, відносними та взаємодіючими [40, 41]. Вони є суб'єктивними, оскільки їх можна розглядати, інтерпретувати та аналізувати по-різному різними людьми та в різному контексті. ASR також відносні, оскільки важливість кожного ASR часто визначається щодо інших ASR в даному контексті.

Вважається, що ASR взаємодіють у тому сенсі, що спроба досягнення певної ASR може в свою чергу мати позитивний чи негативний вплив на інші ASR. ASR часто визначаються основними зацікавленими сторонами системи, такими як кінцеві користувачі, розробники, менеджери та супровідники. ASR використовуються для керівництва процесами проектування та аналізу SA.

ASR менш зрозумілі та керовані, ніж функціональні вимоги. Ця ситуація, коли ASR не приділяють достатньої уваги заздалегідь, досить поширена, незалежно від використовуваної парадигми розробки програмного забезпечення, будь то Agile чи nonAgile. Однією з основних причин цього стану є кількість та визначення ознак

якості, які можна розглянути для системи. У [41] перелічено 161 атрибут якості, який, як стверджується, не є вичерпним списком. Більше того, існування численних класифікацій атрибутів якості є ще однією перешкодою для повного розуміння значення атрибутів якості.

Більше того, не існує універсального визначення так званих атрибутів якості (таких як продуктивність, доступність, можливість модифікації та зручність використання), які зазвичай складають ядро будь-якого набору ASR. Однак точна специфікація ASR є важливою для полегшення ретельного аналізу. Для вирішення цієї ситуації дослідники SA запропонували декілька підходів, таких як сценарії для характеристики атрибутів якості, підхід до характеристики ASR, заснований на структурі, та архітектурно підковані персони. Всі ці три підходи не тільки доповнюють один одного, коли йдеться про отримання та уточнення ASR для проектування та оцінки архітектури, але також можуть бути легко інтегровані в методи ASD для розгляду ASR як першокласних об’єктів.

Підхід, заснований на архітектурно підкованих персонах, був описаний у розділі 4 цієї книги. Чен та його колеги надали основу доказів для систематичної характеристики ASR. Очікується, що система задовольнить різні потреби зацікавлених сторін у виявленні, уточненні та розумінні ASR для проектування та оцінки архітектурних проектних рішень. Сценарії вже давно використовуються в декількох областях різних дисциплін (військова та ділова стратегія, прийняття рішень,). Очікується, що сценарії будуть ефективним засобом визначення атрибутів якості процесів SA, оскільки вони є дуже конкретними, що дозволяє користувачеві легко та точно зрозуміти їх детальний ефект. Сценарій – це текстова, незалежна від системи специфікація атрибута якості [22]. У добре структурованому сценарії повинно бути чітко зазначено ASR з точки зору стимулу та реакції. Важливо, що сценарій має чітко визначені заходи реагування для успішного аналізу SAS. В [22] забезпечив структуру (показано в таблиці 2.1) для структурування сценаріїв.

Таблиця 2.1– Схема створення сценарію для шести елементів

Елементи	Короткий опис
Стимул	Умова, яку потрібно враховувати, коли вона надходить у систему
Відповідь	Діяльність, що здійснюється після надходження стимулу
Джерело стимулу	Суб'єкт (людина, система або будь-який виконавчий механізм), який генерує стимул
Навколишнє середовище	Стан системи, коли виникає подразник, наприклад, перевантажений, працює тощо.
Стимульований артефакт	Деякий артефакт, який стимулюється; може бути ціла система або її частина
Захід реагування	Реакція на стимул має бути деяким чином вимірюваною, щоб вимога могла бути перевірена

Структура формування сценаріїв, показана в таблиці 2.1, вважається досить ефективною для виявлення та структурування сценаріїв, зібраних від зацікавлених сторін. Стверджується, що ця система забезпечує відносно суворий та систематичний підхід до охоплення та документування сценаріїв, що враховують якість, і які можуть бути використані для вибору відповідної системи міркувань для аналізу SA. Сценарії можуть бути абстрактними або конкретними. Абстрактні сценарії використовуються для сприяння виявленню сценаріїв знизу вгору. Абстрактні сценарії незалежні від системи та орієнтовані на ASR. Конкретний сценарій – це текстова специфікація ASR для певної системи.

2.1.3 Методи проектування архітектури програмного забезпечення

Спільнота архітектури програмного забезпечення розробила кілька методів та прийомів для підтримки процесу проектування архітектури. Одним з ключових диференціальних аспектів методів проектування, розроблених дослідниками та практиками SA, є те, що вони підвищують ASR від майже повністю ігнорування до важливої уваги під час проектування SA. Кожен з архітектурно-орієнтованих методів проектування має свої сильні та слабкі сторони. Одним із способів залучення їх сильних сторін та подолання слабких місць є вибір різних підходів та методів з різних методів та їх застосування на основі контекстуальних вимог.

Дослідники з Інституту програмної інженерії (SEI) розробили кілька методів підтримки дизайну архітектури – наприклад, ADD [37] та стилі архітектури на основі атрибутів [42]. Аль-Наєм та ін. [43] запропонували архітектурну основу підтримки прийняття рішень для проектування архітектури, яка складається з дизайнерських рішень, вже оцінених щодо бажаних атрибутів якості та організаційних обмежень.

З цього короткого аналізу відомих архітектурно-центричних методів проектування стає зрозуміло, що метод проектування архітектури повинен не тільки допомогти визначити відповідні дизайнерські рішення щодо ASR, але також повинен включати діяльність, щоб визначити, чи має запропонований дизайн архітектури потенціал для виконання необхідних ASR. Більшість існуючих методів проектування намагаються використати підходи, засновані на знаннях, з точки зору застосування шаблонів дизайну та архітектурних стилів. Однак більшість існуючих архітектурно-орієнтованих методів вважаються важкими та непридатними до швидкого реагування на зміни.

Отже, їх потрібно належним чином адаптувати та контекстуалізувати для середовищ ASD. Кілька дослідницьких зусиль спрямовані на надання вказівок щодо адаптації методів проектування та оцінки архітектури для спритних методів [39, 40].

2.1.4 Документування архітектури програмного забезпечення

Загальновизнано, що архітектура є засобом спілкування між зацікавленими сторонами. Отже, це повинно бути описано однозначно та в достатній кількості деталей, що може надати відповідну інформацію кожному типу зацікавлених осіб [44]. Архітектурна документація також є життєво важливим артефактом для кількох ключових видів діяльності, таких як аналіз рішень архітектури , розподіл робіт та технічне обслуговування після розгортання [1]. Архітектурна документація може споживати велику кількість ресурсів, які потребують обґрунтованого розподілу. Ось чому архітектурна документація зазвичай не практикується загалом, зокрема, в гнучких і худих світах. Важливим питанням архітектурної документації є вибір відповідного засобу опису архітектури, який може слугувати основним цілям

(наприклад, комунікація, аналіз, впровадження та обслуговування) документування SAS.

Останнім часом зростає акцент на документування SAS з використанням різних точок зору, придатних для різних зацікавлених сторін [45]. Архітектурний вигляд – це представлення системи з відповідного набору проблем, що є важливим для різних зацікавлених сторін. Отже, кожна точка зору стосується проблем одного або декількох зацікавлених сторін системи. Термін "вигляд" використовується для вираження архітектури системи щодо певної точки зору t. Відповідно до стандартів IEEE для опису SA [45], архітектурний опис організований за різними поглядами. Одним з найпопулярніших підходів на основі поглядів називають погляди "4 + 1" [27]. Модель подання 4 + 1 має на меті описати SA за допомогою п’яти паралельних переглядів. Кожен з них вирішує певний набір проблем.

– Логічний вигляд позначає розділи функціональних вимог на логічні сутності в архітектурі. Цей вигляд ілюструє об'єктну модель дизайну в об'єктно-орієнтованому дизайні, що відповідає.

– Перегляд процесу використовується для представлення деяких типів ASR, таких як паралельність та продуктивність. Цей погляд можна описати на декількох рівнях абстракції, кожен з яких стосується окремого питання.

– Подання розробки ілюструє організацію вбудованих програмних модулів у середовищі розробки програмного забезпечення. Ця точка зору також представляє внутрішні властивості, такі як багаторазове використання, простота розробки, тестованість та спільність. Зазвичай він складається з підсистем, які організовані в ієрархію шарів. Цей погляд також підтримує розподіл вимог та розподіл робіт, оцінку витрат, планування, моніторинг прогресу та міркування щодо повторного використання, портативності та безпеки.

– Фізичний вигляд являє собою відображення архітектурних елементів, зафіксованих у логічних, технологічних та розвивальних уявленнях, на мережі комп’ютерів. Цей погляд враховує коефіцієнти корисної дії (наприклад, доступність,

надійність (стійкість до несправностей), продуктивність (пропускна здатність) та масштабованість).

– Сценарії використовуються, щоб продемонструвати, що елементи інших поглядів можуть безперешкодно працювати разом. Цей п’ятий погляд складається з невеликого підмножини важливих сценаріїв і має дві основні цілі: драйвер проектування та перевірку / ілюстрацію.

Клементс та його колеги запропонували інший підхід, який називається Views and Beyond (V&B) [44], для документування SA за допомогою поглядів. Як і IEEE Std 1471, їх підхід базується на філософії, згідно з якою замість виписування фіксованого набору поглядів, як Kruchten, SA слід документувати, використовуючи будь-які подання, корисні для розробленої системи. Головний внесок V&B полягає у відображенні конкретних архітектурних стилів у видах та наданні шаблонів для збору відповідної інформації. Окрім підходів до архітектурної документації, спільнота SA запропонувала кілька ADL, які вважаються офіційними підходами до опису SA.

Проведено два порівняльних дослідження мов архітектурного опису (ADL), про які повідомляється в [46, 47]. Уніфікована мова моделювання (UML) [48] стала де-факто стандартним позначенням для документування програмного забезпечення для будь-яких типів середовища розробки програмного забезпечення, гнучкого або неактивного. До серйозного оновлення UML 2.0 UML мав дев'ять діаграм: діаграму класів, діаграму об'єктів, діаграму використання, діаграму послідовностей , діаграму співпраці, діаграму стану, діаграму діяльності, діаграму компонентів та схему розгортання. UML 2.0 усунув основну свою слабкість, надавши нові схеми для опису структури та поведінки системи.

2.1.5 Оцінка архітектури програмного забезпечення

Оцінка архітектури програмного забезпечення є важливою діяльністю в процесі архітектури програмного забезпечення. Основною метою оцінки архітектури є оцінка потенціалу запропонованої / обраної архітектури для забезпечення системи, здатної відповідати необхідним вимогам до якості, та виявлення будь-яких потенційних

ризиків [10, 11]. Дослідники та практики запропонували велику кількість методів оцінки архітектури, для яких також запропонована система класифікації та порівняння [13]. Найбільш широко використовуваними методами оцінки архітектури є метод на основі сценаріїв. Ці методи називаються сценарійними, оскільки сценарії використовуються для характеристики атрибутів якості, необхідних системі. Вважається, що аналіз на основі сценаріїв підходить для атрибутів якості часу розробки (таких як ремонтопридатність та зручність використання), а не для атрибутів якості виконання (таких як продуктивність та масштабованість), які можна оцінити за допомогою кількісних методів, таких як моделювання або математичні математичні моделі.

Серед відомих методів оцінки архітектури, заснованих на сценаріях, є метод аналізу SA (SAAM) [12], метод аналізу компромісного рішення архітектури (ATAM) [13], аналіз рівня ремонтопридатності архітектури (ALMA) [10].

SAAM – це найдавніший метод, запропонований для аналізу архітектури за допомогою сценаріїв. Аналіз декількох архітектур-кандидатів вимагає застосування SAAM до кожної із запропонованих архітектур, а потім порівняння результатів. З точки зору часу та зусиль це може коштувати дуже дорого, якщо кількість архітектур, що підлягають порівнянню, велика. SAAM був розширений на ряд методів, таких як SAAM для складних сценаріїв, розширення SAAM шляхом інтеграції в домен-орієнтований процес та процес повторного використання, і SAAM для еволюції та повторного використання. ATAM виріс із SAAM. Ключовими перевагами ATAM є чіткі способи розуміння того, як архітектура підтримує безліч конкуруючих атрибутів якості, та проведення аналізу торгівлі. ATAM використовує як якісні методи, такі як сценарії, так і кількісні методи для вимірювання якостей архітектури.

Деякі методи використовують одну або комбінацію різних методів аналізу (тобто сценарії, моделювання, математичне моделювання або міркування на основі досвіду). Усі ці методи використовують сценарії для характеристики ознак якості. Потрібні сценарії накладаються на архітектурні компоненти для оцінки спроможності архітектури підтримувати ці сценарії або визначати зміни, необхідні для обробки цих

сценаріїв. PASA – це метод аналізу архітектури, що поєднує сценарні та кількісні методи.

PASA використовує сценарії для визначення цілей продуктивності системи та застосовує принципи та прийоми з програмного забезпечення інженерії продуктивності (SPE), щоб визначити, чи здатна архітектура підтримувати сценарії продуктивності. PASA включає архітектурно-чутливі до виконання архітектурні стилі та анти-шаблони як інструменти аналізу та формалізує діяльність з аналізу архітектури процесу інженерної діяльності.

2.2 Розробка програмного забезпечення Agile і архітектура

Швидкі методи розробки програмного забезпечення обіцяють підтримувати постійний зворотний зв'язок та враховувати зміни у вимогах до програмного забезпечення протягом усього життєвого циклу розробки програмного забезпечення, підтримувати тісну співпрацю між замовниками та розробниками, а також забезпечувати раннє та часте надання програмних функцій, необхідних для системи [10]. Методи ASD базуються на Agile Manifesto, який був опублікований групою розробників програмного забезпечення та консультантів у 2001 р. Згідно з Agile Manifesto:

- Основна роль команді, а не процесам інструментам.
- Робочий програмний продукт має вищий пріоритет над документацією.
- Співпраця з клієнтами протягом всього часу створення продукту.
- Оперативні реакції на зміни плану.

Цей маніфест описує основні цінності, що лежать в основі поглядів спритної спільноти на різні аспекти процесів розробки програмного забезпечення, людей, практики та артефакти. Згідно з цим маніфестом, методи ASD розроблені та впроваджені таким чином, що узгоджуються з основними цінностями ASD, такими як індивідууми та взаємодія над процесом та інструментами, робоче програмне забезпечення над всебічною документацією, співпраця із клієнтами під час

узгодження контракту та реагування на зміни слідуючи плану [13]. Agile Alliance також заручився низкою загальних принципів гнучких процесів, включаючи задоволеність споживачів завдяки ранній і постійній доставці програмного забезпечення, активну участь клієнтів, що розміщується разом, здатність впоратися зі змінами навіть пізно в процесі роботи – це життєвий цикл розробки, простота процесів розробки програмного забезпечення, короткі цикли зворотного зв'язку, взаємна довіра та спільне володіння кодом [13].

Деякі з добре відомих методів ASD – це екстремальне програмування (XP) [13], Crystal Clear [64] та SCRUM [13]. Існує велика кількість книг та дослідницьких робіт з описом деталей кожного з добре відомих і широко практикуваних методів АСД. Ми можемо віднести читача цього розділу до двох хороших джерел, щоб отримати вступну інформацію про різні методи ASD, такі як SCRUM, динамічний метод розробки програмного забезпечення, адаптивна розробка програмного забезпечення, екстремальне програмування та методології кристалів. Оскільки існує велика різноманітність ASD та практик, видається доцільним зберегти наші погляди та обговорення, зосереджені на інтеграції архітектурних підходів у декілька відомих та широко прийнятих методів ASD.

Отже, у цьому розділі коротко торкнуться двох відомих методів ASD – SCRUM, гнучкого методу управління проектами, та XP. Обмежуючи кількість гнучких методів для обговорення з урахуванням архітектурних принципів і практик, ми розраховуємо надати точне, але більш послідовне і глибоке обговорення того, як змусити ASD-методи та архітектурно-орієнтовані практики працювати в гармонії, використовуючи переваги обох дисципліни для розробки високоякісного та економічно ефективного програмного забезпечення ітеративно та поступово, без зайвих затримок проекту та ризиків.

2.2.1 SCRUM-методологія

SCRUM став одним із провідних (якщо не провідним) методом ASD, який був розроблений для управління проектами з розробки програмного забезпечення.

SCRUM – це термін, який використовується в грі в регбі, де він означає "повернення м'яча поза грою назад у гру" завдяки командним зусиллям. У розробці програмного забезпечення SCRUM є ітеративним та додатковим підходом до управління проектами, який забезпечує просту перевірку та адаптацію системи, а не конкретні методи. Проекти, засновані на SCRUM, надають програмне забезпечення з кроком, який називається спринтом (зазвичай 3-4 тижневі ітерації).

Кожен спринт починається з планування, під час якого історії користувачів беруться із відставань на основі пріоритетів, і закінчується оглядовим оглядом. Очікується, що діяльність з планування триватиме кілька годин (наприклад, 4 години) і не надто довго. Спрінт-оглядова зустріч також може тривати близько 4 годин. Очікується, що всі ключові зацікавлені сторони візьмуть участь у плануванні спринту та оглядових зустрічах на початку та завершенні кожного спринту.

Команда SCRUM проводить коротку зустріч (наприклад, максимум 15 хвилин) на початку кожного дня. Ця зустріч називається "щоденною зустріччю SCRUM" і спрямована на те, щоб дати змогу кожному учаснику команди задати лише три запитання: "Що я робив учора, що буду робити сьогодні та які виставники в моїй роботі?" Очікується, що кожен проект SCRUM матиме щонайменше три артефакти: відставання продуктів, відставання спринтів та діаграми вигорання. Спільнота архітектури програмного забезпечення також запозичила термін "відставання" і запропонувала архітектурному процесу зберігати архітектурне відставання, коли архітектура розробляється та оцінюється ітеративно. Нагромадження SCRUM складається з вимог, які необхідно впровадити протягом поточного або майбутнього циклів спринту. Ітераційний та поступовий підхід до архітектури також включає концепцію архітектурних відставань [7]. Третім артефактом є щоденна діаграма вигорання, яка спрямована на надання звіту про стан з точки зору кумулятивної роботи, яку ще потрібно зробити.

2.2.2 Екстремальне програмування

Екстремальне програмування – це ще один популярний спритний підхід, який був розроблений на основі принципів і практики, доведених до екстремальних рівнів. Як і інші методи ASD, XP також виступає за короткі ітерації та часті випуски робочого коду з метою підвищення продуктивності, але все ще пристосовуючи зміни вимог. XP був розроблений для спільних команд з восьми до десяти розробників, що працюють з об'єктно-орієнтованою мовою програмування. Цей підхід швидко набув популярності серед розробників програмного забезпечення, яких не влаштовували такі традиційні підходи до розробки програмного забезпечення, як водоспад. Нижче наведено деякі ключові практики XP.

– Планування: Рекомендується тісна взаємодія між клієнтами та розробниками для оцінки та визначення пріоритетів вимог до наступного випуску. Ці вимоги враховуються як розповіді користувачів про сюжетних картах. Очікується, що програмісти планують і передають лише історії користувачів, узгоджені з клієнтами.

– Невеликі релізи: Початкова версія системи випущена в експлуатацію через кілька ітерацій. Нові функції надаються в наступних релізах щодня або щотижня.

– Метафора: Команда розробників та замовники розробляють набір метафор для моделювання системи, яку потрібно розробити.

– Простий дизайн: XP заохочує розробників максимально спрощувати дизайн системи. За словами Бека, сказати все один раз і лише один раз.

– Тести: Принцип першого тестування означає, що розробники пишуть приймальні тести для свого коду перед тим, як писати сам код. Клієнти пишуть функціональні тести для кожної ітерації, і в кінці кожної ітерації очікується, що всі тести будуть успішно запущені.

– Рефакторинг: Дизайн системи еволюціонував шляхом трансформації існуючого дизайну системи таким чином, щоб усі тест-кейси успішно працювали.

– Програмування в парах: програмний код пишеться двома розробниками, які сидять поруч.

– Постійна інтеграція: весь новий код інтегрований у систему якомога частіше. Усі функціональні тести все ще повинні пройти після інтеграції або новий код буде відхилено.

– Колективна власність: Усі розробники, що працюють над системою, спільно володіють кодом. Це означає, що будь-який розробник може вносити зміни в будь-яке місце коду в будь-який час, коли він вважає це необхідним.

– Підготовка клієнта на місці: Клієнт постійно сидить із командою розробників. Замовник на місці відповідає на запитання, проводить приймальні випробування та забезпечує прогрес у розробці.

– Сорока-годинний тиждень: якщо комусь із команди розробників доведеться працювати понаднормово протягом двох тижнів поспіль, це ознака великої проблеми. Вимоги слід обирати для кожної ітерації таким чином, щоб розробники не потребували надмірного часу.

– Відкритий робочий простір: Розробники мають загальний робочий простір, обладнаний невеликими кабінками по периферії та загальну машину розробки в центрі для парних програмістів.

– Прості правила: Члени команди підписуються на набір правил. Правила можуть бути змінені в будь-який час, доки існує консенсус щодо того, як оцінювати наслідки змін.

2.3 Використання архітектурних і Agile-підходів

В Agile-підходах зростає визнання важливості приділяти більше уваги архітектурним аспектам. Отже, зростає кількість зусиль, спрямованих на виявлення технічних та організаційних проблем, пов'язаних з інтеграцією гнучких підходів у традиційні методи розробки програмного забезпечення [5,8]. Результатом цих зусиль

було декілька пропозицій щодо поєднання переваг основних елементів гнучкого та архітектурно-орієнтованого підходів.

Комбінації стали можливими завдяки тому, що RUP та XP поділяють основи ітеративного, поступового та еволюційного розвитку. Можна стверджувати, що однією з важливих передумов подолання розриву між гнучким та архітектурним підходами є створення та розповсюдження доказової сукупності знань про суперечності та примирення між гнучкими та архітектурними принципами, практиками та їх прихильниками. погляди.

Така сукупність знань також повинна включати виклики, з якими стикаються команди розробників програмного забезпечення, коли вони намагаються слідувати архітектурно підкованим принципам у гнучкому цеху розробки, а також проблеми та ризики, які можуть виникнути в гнучких проектах, що не включають принципи, зосереджені на архітектурі. Ми застосували емпіричний підхід для отримання та розповсюдження розуміння викликів та рішень для поєднання гнучких та архітектурно-орієнтованих підходів. Ми дійшли кількох висновків щодо поєднання гнучких архітектурних підходів. Таблиця 2.2 представляє наше розуміння розміщення деяких добре відомих маневрених практик разом із архітектурними практиками, щоб показати, що багато маневрених практик мають еквівалентні принципи або практики в архітектурних дисциплінах, і їх можна легко адаптувати та застосовувати в гнучких умовах.

Очікується, що архітектори програмного забезпечення виступатимуть як фасилітатори в цілих проектах з розробки програмного забезпечення та як представники загальних атрибутів якості системи. Інші ролі архітекторів ПЗ:

– Архітектор повинен добре розуміти Agile-підходи.

– Архітектор повинен знати, як подати ключове дизайнерське рішення власникам продуктів у конфліктних ситуаціях.

– Архітектор проекту повинен знати загальну архітектуру, необхідні функції та статус релізу.

– Архітектор повинен вміти задокументувати та передати архітектуру всім зацікавленим сторонам.

– Архітектор повинен бути готовим виконувати кілька архітектурних ролей – архітектор рішень, архітектор програмного забезпечення та архітектор впровадження, або повинен мати можливість переконати свою організацію у встановленні різних архітектурних ролей залежно від природи проекту.

Таблиця 2.2 – Поєднання Agile та архітектурних практик між собою

Деякі практики ASD	**Частота використання**
Спринт	Ітераційний характер загальної моделі проектування архітектури програмного забезпечення (SA) з беклогом архітектурних проблем, які слід вирішити
Планування спринту	Пріоритетність архітектурно значущих вимог до кожної ітерації
Огляд спринту	Архітектурний огляд
Щоденні зустрічі	Обмін обґрунтуванням архітектури та знань на засіданнях груп архітектури
Контактна особа від замовника	Залучення ключових зацікавлених сторін до якомога більшої кількості фаз архітектури життєвого циклу
Неперервна інтеграція	Інтеграція на рівні архітектури та взаємодія – підходи до атрибутів якості
Рефакторинг	Рефакторинг на рівні архітектури з використанням шаблонів проектування та архітектурних стилів
Простий дизайн	Дизайн на основі шаблону проектування, щоб дизайн був простим і добре відомим
Спільний доступ до репозиторію з кодом	Робота із стейкхолдерами щодо ключових архітектурних проектних рішень
Стандарти кодування	Архітектурні шаблони та стандарти для підтримки загальних цілей та стандартів
TDD	Тестування на основі архітектури

Важливо пам’ятати, що архітектори програмного забезпечення зазвичай проектують архітектуру, але саме розробники матеріалізують спроектовану архітектуру. Отже, розробники програмного забезпечення повинні нести однакову відповідальність за те, щоб розглядати SA як першокласну організацію, що забезпечує концепцію всієї системи. Ось чому ми стверджували, що роль розробників програмного забезпечення однаково важлива для успішного поєднання гнучких та

архітектурних підходів; команда розробників повинна вирішити, як використовувати різні архітектурні артефакти та документи.

Однак відомостей про те, як команди ASD сприймають та використовують SA існує небагато. Ці знання слід вважати важливими, оскільки, якщо команда ASD вважатиме SA відповідальним для їхніх завдань, не буде потрібно багато зусиль, щоб переконати їх застосувати архітектурні принципи та практики, які можуть мати відношення до їх проекту та контексту.

Agile-розробники використовували архітектурні артефакти для спілкування членів команди, надавали інформацію щодо подальших проектних рішень, задокументованих припущень щодо проектування та оцінювали альтернативних варіантів проектування.

Отже, існує важлива і нагальна потреба зрозуміти важливість, можливості та виклики, пов'язані зі створенням архітектурно-орієнтованих та гнучких підходів до спільної роботи над розробкою програмно-інтенсивних систем та служб. Однією з ключових цілей є створення на основі фактичних даних сукупності знань шляхом виявлення та розуміння основних моментів суперечностей при поєднанні гнучкості та архітектури, а також те, як ці суперечності можна перетворити на переваги на вимог проекту.

РОЗДІЛ 3
РЕЗУЛЬТАТИ ДОСЛІДЖЕННЯ

3.1 Огляд результатів та первинних спостережень

У таблиці 2.1 ми наводимо огляд кейсів дослідження. "Розмір" у таблиці 3.1 стосується кількості працівників, "багатонаціональний" вказує, чи має організація сайти / працівників у різних країнах. "Охоплення" вказує, чи орієнтована організація на глобальний або локальний ринки. Нижче ми надаємо початкові спостереження щодо кожного випадку стосовно ролі архітектора (архітекторів).

Таблиця 3.1 – Огляд судових організацій

Справа / Організація	Домен	Розмір	Опитувані	(Багато-) національний	Досягти
1	Електронна комерція	Середній (<150)	Провідний розробник	Багатонаціональний	Глобальний
2	Програмні рішення	Маленький (~ 50)	Провідний розробник	Національний	Місцеві
3	Фінанси	Великий (~ 3500)	Старший архітектор	Національний	Місцеві
4	Консультація	Великий (> 11000)	Старший архітектор	Багатонаціо-нальний	Глобальний
5	Навігаційні системи	Великий (~ 4500)	Старший архітектор, архітектор програмного забезпечення	Багатонаціональний	Глобальний
6	Побутова техніка	Великий (> 100 000)	Власник дизайну	Багатонаціональний	Глобальний

– У випадку 1 у компанії відсутня чітка роль архітектора. Обговорення / рішення щодо архітектури залучають всю команду SCRUM. Провідний розробник є найвищою технічною особою, і тому останнє слово в архітектурному рішенні.

– При організації справи 2 один проєкт зазвичай виконує одна команда SCRUM. Кожна команда має провідного розробника, який відповідає за архітектурну

діяльність, але вся команда працює над архітектурою та завданнями, що стосуються архітектури.

– В організації випадку 3 старший архітектор, який проживає за межами команди SCRUM, розробляє архітектуру високого рівня та пояснює її керівнику групи (який також виконує роль майстра SCRUM). Архітектор розробляє детальну архітектуру під час проєкту (і при необхідності змінює архітектуру). Він підтримує команду з питань архітектури протягом усього проєкту.

– Case 4 – це консультаційна компанія, яка пропонує клієнтам спеціалізованих архітекторів. Архітектори проживають за межами команд SCRUM і надають свої послуги та знання різним командам SCRUM.

– У організації справи 5 є старший архітектор програмного забезпечення, який не є частиною команди SCRUM, та інший архітектор програмного забезпечення, який є частиною команди SCRUM. Враховуючи масштаби проєктів, кілька команд SCRUM знову беруть участь в окремих проєктах. Тому SCRUM застосовується інтегровано в Scaled Agile Framework (SAFe) із командами SCRUM на рівні “Team” SAFe та старшим архітектором програмного забезпечення на рівні “Program” (на якому команди працюють над корпоративною місією).

– У випадку 6 команда архітекторів, яка знаходиться за межами команд SCRUM, підтримує еталонну архітектуру, яку повинні застосовувати команди SCRUM. Довідкова архітектура та реалізація прототипу надається команді SCRUM на початку проєкту. На додаток до архітектурної команди, системний архітектор, що знаходиться за межами команди SCRUM, підтримує специфікацію вимог, зосереджуючи увагу на правових та нормативних аспектах програмного забезпечення. Крім того, команда SCRUM має власника дизайну, який впорядковує архітектурні дії в команді та розробляє, коли це необхідно, детальні проєкти.

Щоб відповісти на RQ1 (позиція архітекторів у SCRUM), ми проаналізували дані інтерв’ю та характеристики шести випадків. Ми визначили наступні три загальні сценарії (ці сценарії більш докладно описані в підрозділах 4.2-4.4):

- Сценарій “Внутрішній rheitect” у випадку 1 та випадку 2.
- Сценарій "Зовнішній архітектор" у випадку 3 та випадку 4.
- Сценарій “Внутрішній та зовнішній архітектор” у випадку 5 та 6.

Ми визнаємо, що три сценарії все ще мають високий рівень, і кожен сценарій має свій власний “смак” залежно від організації (як описано в підрозділах 4.2-4.4). Крім того, ці сценарії можуть не стати несподіванкою або не з’явитися новим. Однак під час нашого дослідження ми вивели ці сценарії з емпіричного вивчення галузевої практики, а не з анекдотичних доказів з особистого досвіду. Дослідницькі дослідження дають картину реальності, і реальність рідко буває дивною або суперечливою [20].

Що стосується до RQ2 (взаємодії архітекторів з іншими ролями в SCRUM), ми виявили, що взаємодія залежить від положення архітекторів в процесі і навички власників продукту. Так само, як і в проєктах, що не спритні, архітектору в проєктах SCRUM потрібно задокументувати та передати архітектуру; розуміти вимоги, загальну архітектуру та стан її реалізації; а також сприяти та пропонувати послуги командам [21]. Загалом, архітектори взаємодіють з усіма іншими типами ролей та зацікавленими сторонами у проєктах SCRUM. Ми обговорюємо RQ2 та взаємодії більш докладно в наступних розділах, де ми представляємо три сценарії, знайдені для RQ1.

3.2 Сценарій “Внутрішній архітектор”

У випадку 1 та 2 ми знайшли сценарій “внутрішнього архітектора” (рис. 3.1), де архітектор є частиною команди розробників. На рис. 1 показано агрегування взаємодій з двох випадків (тобто ми витягли спільні риси із випадку 1 та випадку 2). На рис. 3.1. Суб’єкти „Команди” вказують ролі, а не окремих осіб.

Архітектор може виконувати кілька ролей, наприклад, виконувати також функції розробника та сприяти кодуванню. Архітектор визначає проєкт архітектури разом з іншими членами команди, який підтримується, оновлюється та поширюється

протягом усього проєкту. Якщо в команді немає спеціального архітектора (як у випадку 1), роль архітектора може взяти на себе вся команда.

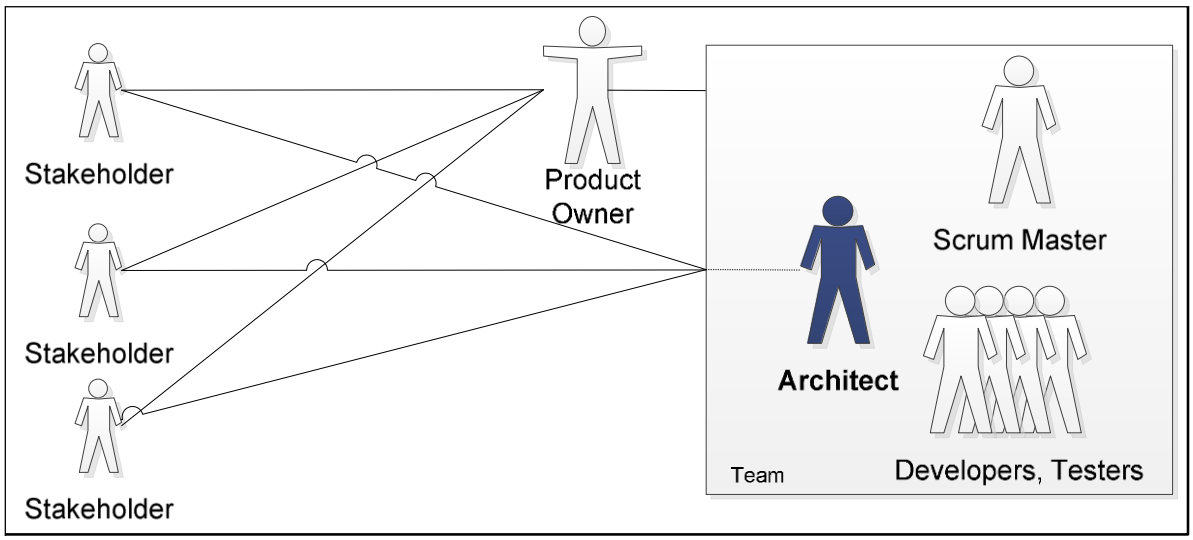

Рисунок 3.1 – Сценарій “Внутрішній архітектор”

Що стосується взаємодій (RQ2), ми виявили у випадку 1, що власники продуктів, які не мають архітектурних знань дуги, хоча і відкриті для архітектурної діяльності, не можуть сприяти архітектурній роботі. “Внутрішні” архітектори повинні безпосередньо спілкуватися із зовнішніми зацікавленими сторонами щодо питань архітектури, що збільшує їх робоче навантаження, і певною мірою перешкоджає меті ролі Власника продукту в SCRUM. Як ми виявили у випадку 2, власники продуктів надають архітекторам та колективу неповну інформацію. Тому (і подібно до випадку 1) зовнішні зацікавлені сторони взаємодіють з командою та дуговим хітектом безпосередньо і навпаки (позначено пунктиром на рис. 3.1). Деякі власники продуктів у випадку, якщо 2 ображаються за те, що їх не залишають у спілкуванні, і тому можуть виникнути конфлікти між власниками продуктів та архітекторами. Слід зауважити, що на рис. 3.1 власник продукту може спілкуватися з будь-яким членом команди (включаючи архітектора), і ми не ілюструємо спілкування в команді (усі члени команди спілкуються між собою).

3.3 Сценарій „Зовнішній архітектор”

У випадках 3 та 4 ми визначили сценарій „зовнішнього архітектора” (рис. 3.2). На рис. 3.2 показано агрегування взаємодій двох випадків (тобто ми об’єднали випадок 3 та випадок 4 у загальний сценарій). У цьому сценарії архітектор не входить до команди SCRUM, а також не існує “внутрішнього” архітектора. Архітектор розробляє початковий проєкт високого рівня і представляє його колективу. Архітектор доступний під час проєкту для уточнення проєкту, вирішення проблем, адаптації або розширення архітектури або прийняття нових архітектурних рішень. Архітектор також контролює відповідність виробу архітектурному дизайну.

Що стосується RQ2 (взаємодії), ми виявили у випадку 3, що архітекторам потрібно навчити зацікавлених сторін проєкту поза командою про дугову архітектуру та значення архітектури. Однак через часові обмеження Власника продукту в цьому випадку архітектор не може навчити Власника Продукту питанням архітектури або залучати Власника Продукту до архітектурних рішень.

Окрім того, оскільки архітектор є зовнішнім для команди і відповідає за більше ніж одну команду, архітектор не завжди доступний для команд і не завжди в курсі подій команди. Тому архітектор залучає керівників команд виступати в ролі «довіреної особи», щоб представляти архітектора в команді. У випадку 4, як і у випадку 3, архітектор не спілкується з організацією замовлення з питань архітектури, оскільки організація проєкту не має архітектурних знань (пунктирна лінія на рис. 3.2). На відміну від випадку 3, у випадку 4 архітектор доступний до команди більш регулярно.

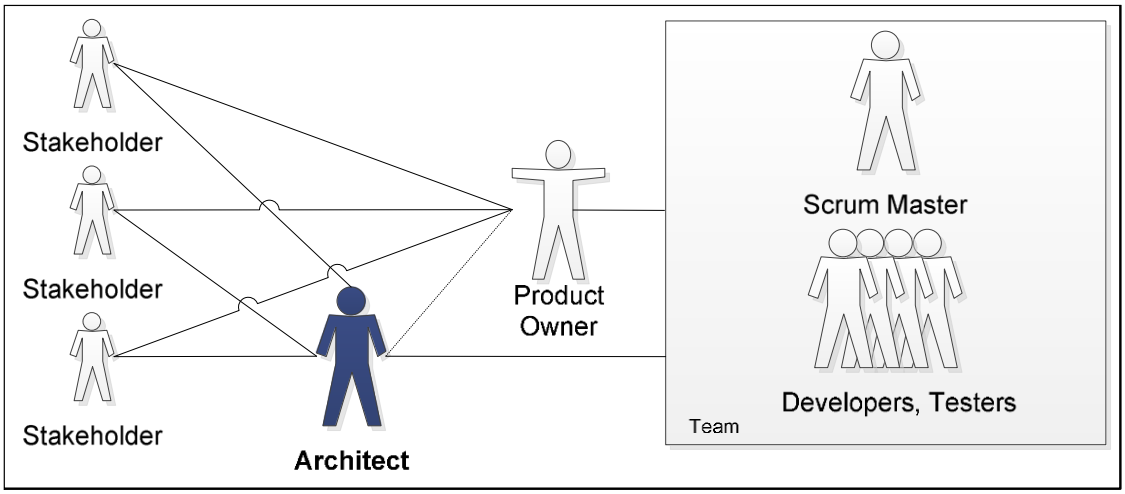

Рисунок 3.2 – Сценарій „Зовнішній архітектор”

3.4 Сценарій “Внутрішні та зовнішні архітектори”

У випадках 5 та 6 ми визначили сценарій „внутрішнього та зовнішнього архітекторів” (рис. 3.3 знову показує узагальнений сценарій для випадку 5 та 6). У цьому сценарії існує зовнішній та внутрішній архітектор. “Внутрішній” архітектор займається завданнями, пов’язаними з повсякденною роботою команди, тоді як “зовнішній” архітектор займається рішеннями вищого рівня, які потенційно можуть вплинути і на інші команди. У випадку 6 зовнішній архітектор є членом архітектурної команди і поєднується з командою SCRUM.

Що стосується взаємодій (RQ2), ми виявили, що зовнішній архітектор взаємодіє із зовнішніми зацікавленими сторонами, а комунікація між зовнішнім архітектором та командою здійснюється через внутрішнього архітектора. У випадку 5 зовнішній архітектор наділений повноваженнями переносити історії зі списку спринтів, який Власник продукту не вказав достатньо, розміщуючи архітектора ієрархічно вище, ніж Власник продукту.

Крім того, внутрішнім архітекторам у різних командах пропонується спілкуватися між собою за різними архітектурними рішеннями, що стосуються інших команд або частин проєкту (не зображено на рис. 3.3, оскільки це специфічний випадок 5). Як ми виявили у випадку 6, між зовнішніми архітекторами та зовнішніми

зацікавленими людьми часто існує односпрямоване спілкування. Це означає, що архітектори отримують вклади від зацікавлених сторін, але не перевіряють у зацікавлених сторін якість своїх рішень.

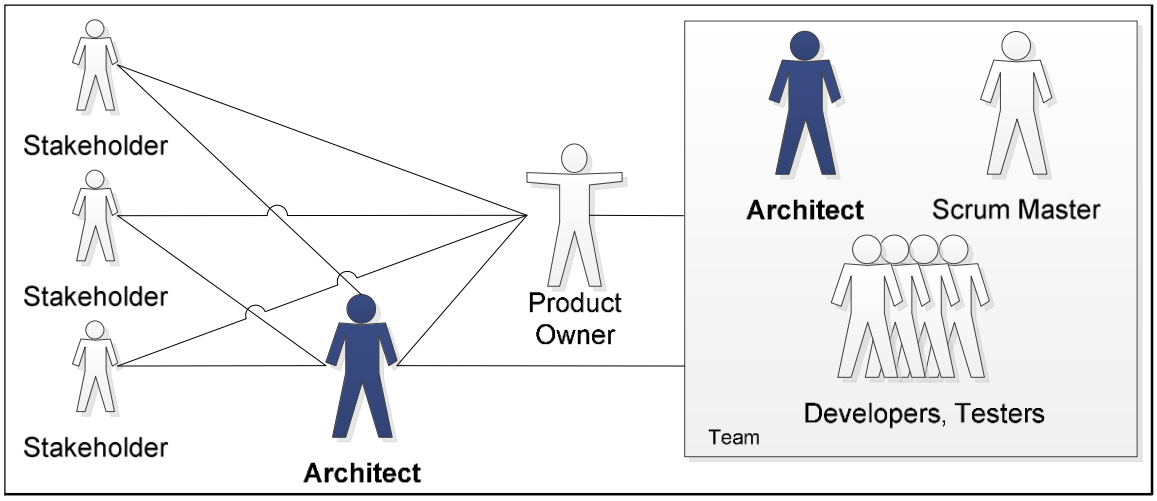

Рисунок 3.3 – Сценарій “Внутрішні та зовнішні архітектори”

Крім того, у випадку 6 зовнішні архітектори можуть не надати достатнього вкладу для розробки часів проєкту під час проєкту, що є вирішальним у гнучких проєктах, де архітектурні рішення можуть змінюватися з часом. Щоб пом'якшити цю проблему, зовнішній архітектор тимчасово приєднується до команди розробників на початку проєкту. У випадку 6, є додатковий архітектор системи поза командами SCRUM та архітектури. Архітектор системи взаємодіє з командою незалежно від “зовнішнього” архітектора, створюючи додатковий канал зв'язку до команди та накладаючи на команду додаткові вимоги до документації.

РОЗДІЛ 4
ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ

4.1 Впровадження в Україні світового досвіду щодо покращення умов і безпеки праці в ІТ-компаніях

Поява та впровадження нових інформаційно-комунікаційних технологій зумовлює необхідність подальшого вдосконалення охорони праці фахівців ІТ-індустрії. З метою належного правового забезпечення необхідно розширити ти доповнити перелік основних професій комп'ютерної галузі у національному класифікаторі ДК-003-2010, а також підготувати відповідний випуск у кваліфікаційному довіднику посад фахівців ІТ-індустрії, що сприятиме вирішенню питань їх соціального захисту, пенсійного забезпечення, атестації робочих місць основних професій за умовами праці на предмет подальших певних видів пільг та компенсацій за важкі шкідливі і небезпечні умови праці. Важливим напрямом стосовно визначення професійної придатності фахівців з інформаційних технологій є проведення психофізіологічної експертизи відповідно до 5 статті Закону України «Про охорону праці».

Робота з комп'ютерами нового покоління характеризується певним психофізіологічними перенавантаженнями, втомою зорового аналізатора, гіпокінезією, відсутність диференційованих норм праці при роботі з новою комп'ютерною технікою в залежності від віку, статі, категорії зорової роботи, режимів праці і відпочинку (протягом робочого дня, тижня, щорічного режиму відпусток).Все це потребує розробки нових нормативно-правових актів з регламентації праці та відпочинку фахівців ІТ-індустрії і стандартів підприємств, центрів комп'ютерної техніки, центрів інформаційних технологій, сучасних комп'ютерних класів.

Особлива роль з точки зору збереження та відновлення здоров'я працюючих в комп'ютерні' галузі належить попереднім та періодичним наглядам з подальшої

психофізіологічної експертизи і встановленням професійної придатності при роботі з комп'ютерами нового покоління, який супроводжується виникненням певних факторів професійного ризику електро-травматизму при їх ремонті та обслуговуванні. В цьому зв'язку необхідне запровадження експертизи на предмет безпечної експлуатації ПЕОМ, тобто офіційне підтвердження фактичних параметрів електробезпеки, їх відповідності вимогам нормативної документації фахівців, які проводять таку експертизу повинні пройти навчання і перевірку знань відповідно до вимог ДНАОП 0.00-8.20-99.

За результатами експертизи повинні прийматися рішення про відповідність ПЕОМ нормам безпеки, терміни чергової експертизи, оформлюються протоколи вимірювань і випробувань, проведені у разі потреби розрахунки та експертний висновок. Для підвищення розумової працездатності то зорової роботи повинна здійснюватися ергономічна оптимізація в рамках системи «оператор-термінал», яка сприятиме результативній фізичній та інтелектуальний працездатності і відновленню психосоматичного здоров'я фахівців ІТ-індустрії [49].

Заслуговує на увагу зарубіжний досвід створення у приміщеннях та в зоні їх розміщення на територіях підприємств спеціальних візуальних комфортних умов та забезпечення вимог виробничої естетики, дотримання норм рівнів виробничого шуму та акустичної тиші за межами офісу. Також дуже важливим є використання в офісних приміщеннях та кабінетах психофізіологічного розвантаження функціональної музики, яка сприяє попередженню перевтоми і підтриманню необхідного рівня розумової працездатності фахівців комп'ютерної галузі [50], [51].

В цьому напрямі заслуговує на увагу створення при великих центрах інформаційних технологій кімнат (кабінетів) психофізіологічного розвантаження працівників галузі (на 5 місць).Зарубіжний досвід охорони праці при використанні новітніх інформаційних технологій та сучасного комп'ютерного обладнання передбачає з метою попередження наслідків монотонної праці, підвищення рівня рухової активності і покращення розумової працездатності

фахівців ІТ-індустрії під час технологічних перерв участь у спеціальних облаштованих приміщеннях необхідним спортивним інвентарем та різними тренажерами відповідних фізичних вправ, індивідуальних тренінгових завдань відповідно до віку, статі та категорії зорової роботи.

Такий підхід дозволяє зняти надлишкове психофізіологічне перевантаження, підвищити працездатність центральної нервової системи, попередити перевтому зорового аналізатора. Показана ефективність проведення різноманітних за своєю спрямованістю вправ робітників цієї галузі (приблизно на 5-30%).

Всі наведені заходи щодо вдосконалення охорони праці фахівців ІТ-індустрії повинні контролюватися службою охорони праці та комісією з охорони праці підприємства. Особливе значення у соціальному захисті цієї категорії працівників належить прийняття комплексного договору, який може забезпечити фахівців додатковими пільгами та компенсаціями.

4.2 Вплив електромагнітного імпульсу (ЕМІ) ядерного вибуху на елементи виробництва та заходи захисту

Ядерні вибухи в атмосфері й більш високих шарах призводять до виникнення потужних електромагнітних полів з довжиною хвиль від 1 до 1000 м і більше. Ці поля через короткочасне існування називають електромагнітним імпульсом (ЕМІ). ЕМІ виникає при ядерному вибусі у воєнний час, у мирний час – при випробуванні ядерної зброї або ядерних аваріях і катастрофах в атмосфері й космосі.

Основною причиною виникнення ЕМІ тривалістю менше 1 с вважають взаємодію гамма-променів і нейтронів ядерного вибуху з атомами газів повітря, внаслідок чого з них вибиваються електрони (ефект Комнтона) і хаотично розлітаються в середовищі позитивно заряджених атомів газів. Важливе значення має також виникнення асиметрії в розподілі просторових електричних зарядів, пов'язаних з особливостями поширення гамма-променів і утворення електронів.

Гамма-промені, які випускаються із зони вибуху в напрямі поверхні землі, поглинаються в більш щільних шарах атмосфери, вибиваючи з атомів повітря швидкі електрони, які летять у напрямку гамма-променів зі швидкістю світла, а позитивні іони (залишки атомів) залишаються на місці. У результаті поділу і переміщення позитивних і негативних зарядів у цій області й у зоні вибуху, а також при взаємодії зарядів з геомагнітним полем Землі утворюються елементарні й результуючі електричні та магнітні поля ЕМІ, які досягають поверхні землі в зоні радіусом кількох сотень кілометрів. Виникають сильні поперечні токи і утворюється подібність великої "плоскої антени", яка випромінює потужний ЕМІ з часом наростання порядна 10 не і тривалістю більше 230 не; зі смугою частот від 10 кГц до 100 МГц. Залежно від висоти ядерного вибуху за інших однакових умов змінюються характер, інтенсивність ЕМІ і дальність його поширення.

При наземному і низькому повітряному вибуху уражаюча дія НМІ спостерігається на відстані кількох кілометрів від центру вибуху. Під час ядерного вибуху на висотах від 3 до 25 км утворюється симетричне джерело генерації, але радіус поширення ЕМІ залишається обмеженим внаслідок сильного поглинання гамма-випромінювання в щільних шарах атмосфери.

Найбільшу уражаючу дію має ЕМІ, що виникає при екзоатмо-сферному вибуху (більше 40 км). Зі збільшенням висоти вибуху збільшується і район джерела генерації ЕМІ, досягаючи в діаметрі тисячі кілометрів і товщини 20–40 км. Так, під час вибуху на висоті 80 км.

ЕМІ буде поширюватися на площі радіусом 960 км, а під час вибуху на висоті 160 км – на площі радіусом 1400 км. Екзоатмосферний ЕМІ характеризується дуже малим часом наростання (декілька сотень наносекунд), високою інтенсивністю електричного поля (більше 50 кВ/хв) і магнітного поля (близько 130 А/хв). Розряд блискавки порівняно з ЕМІ має значно більшу тривалість зростання і спаду (5–300 мКе), створює дуже потужні поля (близько 100 кВ/хв), несе значно більшу енергію, але спектр частот становить близько 10 МГц, тоді як для ЕМІ він більше – 100 МГц. Пікове значення ЕМІ може досягти 50 000 В/хв, що дорівнює всій енергії

яка випромінюється в радіочастотній частині спектра. Уражаюча дія ЕМІ обумовлена виникненням напруги і струмів у провідниках різної довжини, розміщених у повітрі, землі.

ЕМІ захвачують спектр частот від десятків до кількох сотень мегагерц, тобто діапазон, в якому працюють установки електропостачання, зв'язку і радіолокації. Напруженість електромагнітного поля, створюваного ЕМІ, досягає 50 000 В/м, тоді як у радіолокації вона не перевищує 200 В/м, а у зв'язку – 10 В/м.

У серпні 1958 р. у момент заатмосферного термоядерного вибуху, проведеного США над островом Джонсон, на Гавайях, які знаходяться за 1000 км від епіцентру вибуху, погасло освітлення на вулицях. Це сталося в результаті дії ЕМІ на повітряні лінії електропередач, які відіграли роль протяжних антен.

Величина ЕМІ залежно від ступеня асиметрії вибуху може бути різною – від десятків до сотень кіловольт на метр антени, тоді як чутливість звичайних УДК-приймачів становить кілька десятків або сотень мікровольт. Так, у разі наземного вибуху потужністю 1 Мт напруженість поля на відстані 3 км становить близько 50 кВ/м, а на відстані 16 км – 1 кВ/м. А у разі заатмосферного вибуху такої ж потужності напруженість поля становитиме тисячі кіловольт на метр площі в кілька тисяч квадратних кілометрів земної поверхні.

Уражаюча дія ЕМІ в приземній області й на землі пов'язана з акумулюванням його енергії довгими металевими предметами, рамними і каркасними конструкціями, антенами, лініями електропередачі та зв'язку, в них виникають сильні наведені струми, які руйнують підключене електронне та інше чутливе устаткування. У районі дії ЕМІ безпосередній контакт людини зі струмопровідними предметами небезпечний.

ЕМІ уражає радіоелектронну і радіотехнічну апаратуру. В провідниках індукуються високі напруги і струми, які можуть призвести до постійних або тимчасових пошкоджень ізоляції кабелів, відключення реле і переривників, пошкодження елементів зв'язку, магнітних запам'ятовуючих пристроїв у ЕОМ і системах передачі даних тощо. Найбільш уразливими елементами обладнання є

напівпровідникові прилади – транзистори, діоди, випрямлячі, інтегруючі кола, цифрові процесори, управляючі й контрольні прилади. Чутливі до пошкодження ЕМІ транзистори звукової частоти, перемикаючі транзистори, інтегруючі кола та ін.

Особливо чутливими до впливу ЕМІ є 6 основних груп об'єктів і систем:

1) системи передачі електроенергії: повітряні ЛЕП, кабельні лінії, різні види з'єднувальних ліній і повітряна електропроводка;

2) системи виробництва, перетворення і накопичення енергії: електростанції, генератори постійного і змінного струму, трансформатори, перетворювачі струмів і напруг, комутатори і розподільні пристрої, електричні батареї і акумулятори, паливні, сонячні й термоелементи;

3) системи регулювання і управління: електромеханічні й електронні датчики та інші елементи автоматики, комп'ютерні установки, м і к ро п роцесори;

4) системи споживання електроенергії: електродвигуни і електромагнітні, нагрівальні, холодильні, вентиляційні, освітлювальні установки та кондиціонери;

5) системи електротяги: електроприводи, напівпровідникові та інші типи перетворювачів;

6) системи радіозв'язку, передачі, зберігання і накопичення інформації: антени, хвилеводи, коаксильні кабелі, електронні прилади, радіопередавачі, радіоприймачі, установки автономного електропостачання, змішувачі, телефонні апарати, телеграфні установки, заземлені кабелі й проводи, АТС.

Найбільш стійкі до ЕМІ вакуумні електронні прилади, які виходять із ладу при енергії 1 Дж. Величина енергії ЕМІ залежить від ширини періоду частот антенних систем.

Більшість систем зв'язку працюють у діапазоні частот від середніх до ультрависоких і будуть пошкодженими залежно від робочого періоду частот. Радіолокаційні системи менше пошкоджуються від ЕМІ, тому що вони працюють у періоді частот, де щільність енергії ЕМІ невелика. Іскріння, яке виникає під впливом високого електричного поля ЕМІ, може спричинити спалахування парів бензину та інших налив у сховищах.

Якщо ядерний вибух стався поблизу лінії електропостачання, зв'язку великої довжини, то наведені в них напруги можуть поширюватися по проводах на багато кілометрів, пошкоджувати апаратуру й уражати людей, які знаходяться на безпечній відстані відносно інших уражаючих факторів ядерного вибуху.

ЕМІ небезпечний і за наявності міцних споруд, розрахованих на стійкість проти ударної хвилі наземного ядерного вибуху, проведеного на відстані кількох сотень метрів.

Сучасний рівень знань про природу і властивості ЕМІ дає можливість розробити захист від нього і впровадити заходи захисту до яких входять схеми, стійкі до електромагнітної інтерференції, радіоелектронні елементи стійкі до ЕМІ, екранування окремих пристроїв або цілих електронних систем.

ВИСНОВКИ

Загальна думка про те, що архітектори можуть з’являтися на різних посадах у гнучких проєктах, обговорювались в різних роботах. Дана кваліфікаційна робота виходить за рамки загальної ідеї і відрізняється тим, що ми виводимо три конкретні сценарії для команд SCRUM. Тому ми співвідносимо сценарії та взаємодії архітекторів, які ми виявили, до існуючої літератури.

– Сценарій “внутрішнього архітектора” каже, що всі члени команди беруть участь у архітектурній діяльності, але жоден член команди не грає ролі або несе повної відповідальності “Архітектор”.

Архітектурні роботи повинні виконуватися або цілою командою, або принаймні деякими кваліфікованими членами команди – для обміну знаннями між командою. В Agile-проектах роль архітектора може взяти на себе і вся команда. Роль архітектора в команді (тобто її спільна між усіма членами команди або виконувана одним членом команди) суттєво сприяє успішним проєктам. Крім архітекторів, власник продукту та команда розробників також бере участь у процесі прийняття рішення про архітектуру та інколи відповідає за нього.

– У сценарії «зовнішнього архітектора» архітектор може працювати з багатьма гнучкими командами та співпрацювати з іншими архітекторами (наприклад, як команда архітекторів проєктів або як член архітектурної ради, де узгоджуються та впроваджуються загальні архітектурні практики).

„Зовнішній” архітектор фокусується на архітектурних схемах високого рівня, архітектурному виборі та атрибутах якості. Архітектор спілкується із зовнішніми зацікавленими сторонами та тісно співпрацює з Власником продукту та командою для визначення та розвитку архітектури.

Загалом, наше дослідження сприяє висвітленню різних сценаріїв взаємодії відповідальних за архітекторську діяльність у SCRUM. Навіть незважаючи на те, що архітектор може не мати явної та офіційно визнаної ролі, відповідні завдання та

можливості існують у налаштуваннях SCRUM. У цьому розділі ми узагальнюємо потенційні та загальні висновки досліджень:

– У сценаріях, в яких беруть участь зовнішні архітектори (тобто сценарії „зовнішнього архітектора“ та „внутрішніх та зовнішніх архітекторів “), ми помітили, що зовнішні архітектори не надають достатньої інформації зовнішнім зацікавленим сторонам про прийняті архітектурні рішення та командам, включаючи “внутрішніх” архітекторів.

– У сценарії «зовнішнього архітектора» потенційні виклики для архітектури можуть бути спричинені віддаленістю між архітектором та командою розробників. Оскільки не існує «внутрішнього» архітектора, архітектурно важливий внесок від команди може бути важче охопити на початку проєкту. Це може спричинити проблеми, зокрема у критичних проєктах. Крім того, зовнішній архітектор може не мати можливості постійно підтримувати та навчати команду під час проєкту.

– Сценарій "зовнішнього архітектора", судячи з усього, суперечить ідеї міжфункціональної команди у SCRUM та парадигмі SCRUM, тоді як сценарій "внутрішнього архітектора" – це те, що очікує та підтримує SCRUM.

– У сценарії «внутрішніх та зовнішніх архітекторів» синхронізація ідей та розуміння між зовнішніми та внутрішніми архітекторами може бути складною і навіть призвести до конфліктів. Це особливо вірно там, де не існує чіткої ієрархічної структури між зовнішніми архітекторами, командою (включаючи архітектора команди, майстра SCRUM) та власником продукту.

– Завдання архітектора в SCRUM подібні до завдань архітекторів у проєктах, що не є SCRUM. Однак у всіх випадках нашого дослідження ми спостерігали, що архітектори в проєктах SCRUM повинні мати певні навички, які виходять за рамки "традиційного" архітектурного проєктування, такі як розуміння Agile-практик, можливість спілкування та переконання Власника продукту у своїх рішеннях.

Крім того, архітектори SCRUM можуть виконувати архітектурну діяльність по-різному, наприклад, заздалегідь створювати менші частини проєкту, взаємодіючи з різноманітним колом зацікавлених сторін, застосовувати специфічні практики, такі як гнучке моделювання. Визначення архітектурних навичок в гнучких умовах, які потенційно можуть відрізнятися від навичок, необхідних для «традиційного» попереднього архітектурного проєктування, є темою для майбутніх досліджень.

Теоретично власник продукту міг би істотно полегшити роботу архітектора. Власник продукту може надати єдину точку зв'язку для інформації, наданої зовнішніми зацікавленими сторонами, спрощуючи зв'язок між архітекторами та зовнішніми зацікавленими сторонами. Це означає, що Власнику продукту необхідно збирати та надавати архітектурно важливу інформацію, повідомляти архітектурні рішення зацікавленим сторонам і навіть брати участь у архітектурній діяльності (наприклад, оглядах архітектури). Усі заходи, пов’язані з „отримання вхідних даних” для проєктування та „надання інформації” зацікавленим сторонам, які виходять за межі команди, будуть спрямовані на Власника продукту, а не на зовнішніх зацікавлених сторін.

Однак ця додаткова відповідальність вимагає більше зусиль і часу від Власника продукту, а також компетентності в архітектурній діяльності. Власники продуктів повинні бути навчені, щоб не перешкоджати спілкуванню між Власником продукту та командою через відсутність знань, пов'язаних з архітектурою (і, отже, неможливість надати команді інформацію, що відповідає архітектурі від зацікавлених сторін).

ПЕРЕЛІК ПОСИЛАНЬ

1. Bass, Len, Paul Clements, and Rick Kazman. Software architecture in practice. Addison-Wesley Professional, 2003.

2. Abrahamsson, Pekka, Muhammad Ali Babar, and Philippe Kruchten. "Agility and architecture: Can they coexist?." IEEE Software 27.2 (2010): 16-22.

3. Yang, Chen, Peng Liang, and Paris Avgeriou. "A systematic mapping study on the combination of software architecture and agile development." Journal of Systems and Software 111 (2016): 157-184.

4. Eloranta, Veli-Pekka, and Kai Koskimies. "Lightweight architecture knowledge management for agile software development." Agile Software Architecture. Morgan Kaufmann, 2014. 189-213.

5. One, Version. "9th annual state of agile survey." Survey. Accessed online 15 (2015).

6. Runeson, Per, and Martin Höst. "Guidelines for conducting and reporting case study research in software engineering." Empirical software engineering 14.2 (2009): 131-164.

7. Schwaber, Ken. Agile project management with Scrum. Microsoft press, 2004.

8. Schwaber, K., and M. Beedle. "Agile software development with Scrum Prentice Hall PTR Upper Saddle River." NJ, USA (2001).

9. Faber, Roland. "Architects as service providers." IEEE software 27.2 (2010): 33-40.

10. Ihor, Bodnarchuk, et al. "Multicriteria Choice of Software Architecture Using Dynamic Correction of Quality Attributes." International Conference on Computer Science, Engineering and Education Applications. Springer, Cham, 2019.

11. Kharchenko, Alexander, Ihor Bodnarchuk, and Vasyl Yatcyshyn. "The method for comparative evaluation of software architecture with accounting of trade-offs." American Journal of Information Systems 2.1 (2014): 20-25.

12. Харченко, Олександр Григорович, Василь Володимирович Яцишин, and Ігор Едуардович Райчев. "Інструментальний засіб розробки та комунікації вимог якості до програмних систем." (2010).

13. A Kharchenko, I Bodnarchuk, I Halay, V Yatcyshyn. An Optimal Trade-off Solusion of the Software Architecture Choice Problem // Journal of Information and Computing Science, 2016. PP. 281 – 290.

14. Ihor, Bodnarchuk, et al. "Multicriteria Choice of Software Architecture Using Dynamic Correction of Quality Attributes." International Conference on Computer Science, Engineering and Education Applications. Springer, Cham, 2019.

15. Barbara Kitchenham and Shari Lawrence Pfleeger. 2003. Principles of survey research part 6: data analysis. SIGSOFT Softw. Eng. Notes 28, 2 (March 2003), 24 – 27. DOI: https://doi.org/10.1145/638750.638758

16. Kruchten, Philippe. "What do software architects really do?." Journal of Systems and Software 81.12 (2008): 2413-2416.

17. Fowler, Martin. "Who needs an architect?." IEEE SOFTWARE 20.5 (2003): 11-13.

18. Miles, Matthew B., A. Michael Huberman, and Johnny Saldana. "Qualitative data analysis: A methods sourcebook." (2014).

19. Krippendorff, Klaus. Content analysis: An introduction to its methodology. Sage publications, 2018.

20. Palmer SR, Felsing JM. A practical guide to feature-driven development. USA: Prentice Hall; 2002.

21. Beck K. Extreme programming explained: embrace change. Reading, MA: Addison Wesley Longman, Inc.; 2000.

22. Bass L, Clements P. Kazman R. Software architecture in practice. 2nd ed. Boston, MA: Addison-Wesley, 2003.

23. Ihme, Tuomas, and Pekka Abrahamsson. "Agile architecting: The use of architectural patterns in mobile java applications." International Journal of Agile Manufacturing 8.2 (2005): 97-112.

24. Kruchten, Philipe. "Situated agility." Proceedings of the 9th International Conference on Agile Processes and eXtreme Programming in Software Engineering, Limerick, Ireland. 2008.

25. Boehm, Barry. "Get ready for agile methods, with care." Computer 35.1 (2002): 64-69.

26. Angelov, S., Meesters, M., & Galster, M. (2016, November). Architects in SCRUM: What challenges do they face?. In European Conference on Software Architecture (pp. 229-237). Springer, Cham.

27. Boehm, Barry, and Richard Turner. "Management challenges to implementing agile processes in traditional development organizations." IEEE software 22.5 (2005): 30-39.

28. Parnas, David L. "On the criteria to be used in decomposing systems into modules." Software pioneers. Springer, Berlin, Heidelberg, 2002. 411-427.

29. Stevens, Wayne P., Glenford J. Myers, and Larry L. Constantine. "Structured design." *IBM systems journal* 13.2 (1974): 115-139.

30. Gorton, Ian. Essential software architecture. Springer Science & Business Media, 2006.

31. Perry, Dewayne E., and Alexander L. Wolf. "Foundations for the study of software architecture." ACM SIGSOFT Software engineering notes 17.4 (1992): 40-52.

32. Babar, Muhammad Ali, et al. Software architecture knowledge management. Berlin: Springer, 2009.

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

34. America, Pierre, Eelco Rommes, and Henk Obbink. "Multi-view variation modeling for scenario analysis." International Workshop on Software Product-Family Engineering. Springer, Berlin, Heidelberg, 2003.

35. P. B. Kruchten, "The 4+1 View Model of architecture," in IEEE Software, vol. 12, no. 6, pp. 42-50, Nov. 1995, DOI: 10.1109/52.469759.

36. Hofmeister, Christine, Robert Nord, and Dilip Soni. Applied software architecture. Addison-Wesley Professional, 2000.

37. Ran, Alexander, M. Jazayeri, and F. van der Linden. "ARES conceptual framework for software architecture." Software Architecture for Product Families Principles and Practice (2000): 1-29.

38. Hofmeister, Christine, et al. "A general model of software architecture design derived from five industrial approaches." Journal of Systems and Software 80.1 (2007): 106-126.

39. Tang, Antony, et al. "A comparative study of architecture knowledge management tools." Journal of Systems and Software 83.3 (2010): 352-370.

40. Chung, Lawrence, et al. Non-functional requirements in software engineering. Vol. 5. Springer Science & Business Media, 2012.

41. Chen, Long, et al. "Theoretical kinetic investigation of thermal decomposition of methylcyclohexane." Computational and Theoretical Chemistry 1026 (2013): 38-45.

42. Klein, Mark H., et al. "Attribute-based architecture styles." Working Conference on Software Architecture. Springer, Boston, MA, 1999.

43. Al-Naeem, Tariq, et al. "A quality-driven systematic approach for architecting distributed software applications." Proceedings of the 27th international conference on Software engineering. 2005.

44. Clements, Paul, et al. "Documenting software architectures: views and beyond." 25th International Conference on Software Engineering, 2003. Proceedings.. IEEE, 2003.

45. Jen, Lih-ren, and Yuh-jye Lee. "Working Group. IEEE recommended practice for architectural description of software-intensive systems." IEEE Architecture. 2000.

46. Clements, Paul C. "A survey of architecture description languages." Proceedings of the 8th international workshop on software specification and design. IEEE, 1996.

47. Medvidovic, Nenad, and Richard N. Taylor. "A classification and comparison framework for software architecture description languages." IEEE Transactions on software engineering 26.1 (2000): 70-93.

48. Fowler, Martin. UML distilled: a brief guide to the standard object modeling language. Addison-Wesley Professional, 2004.

49. Вдосконалення охорони праці в IT-індустрії. // Харківський національний дорожний університет. [Електронний ресурс]. – Режим доступу: https://www.khadi.kharkov.ua/fileadmin/P_vcheniy_secretar/%D0%9E%D0%A5%D0%9E%D0%A0%D0%9E%D0%9D%D0%90_%D0%9F%D0%A0%D0%90%D0%A6%D0%86/R_IT-INDUSTRIA.pdf

50. Сьогодні UA [Електронний ресурс]: [Веб-сайт]. – Електронні дані. – Режим доступу: https://www.segodnya.ua/lifestyle/fun/pochti-kak-u-google-chem-udivlyayut-ofisy-ukrainskih-it-kompaniy-764025.html

51. MRPL.CITY [Електронний ресурс]: [Веб-сайт]. – Електроні дані. – Режим доступу: https://mrpl.city/news/view/mariupolskaya-konditerka-stanet-biznes-tsentrom-foto-plusvideo

ДОДАТКИ

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ТЕРНОПІЛЬСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
ІМЕНІ ІВАНА ПУЛЮЯ

М А Т Е Р І А Л И

VIII НАУКОВО-ТЕХНІЧНОЇ КОНФЕРЕНЦІЇ

«ІНФОРМАЦІЙНІ МОДЕЛІ, СИСТЕМИ ТА ТЕХНОЛОГІЇ»

9–10 грудня 2020 року

ТЕРНОПІЛЬ
2020

УДК 004.415
А.І. Вовк, Н.-П. Б.Оберванюк
(Тернопільський національний технічний університет імені Івана Пулюя)

ПРОБЛЕМИ ПРОЕКТУВАННЯ ПРОГРАМНОЇ АРХІТЕКТУРИ ДЛЯ AJILE-ПРОЄКТІВ

UDC 004.415
A.I. Vovk, N.-P.B. Obervaniuk

PROBLEMS OF SOFTWARE ARCHITECHTURE DESIGN FOR AJILE-PROJECTS

Сучасні технології програмування (гнучке та екстремальне програмування, технологія SCRUM та інші) по суті є ітераційними. При виконанні поточної ітерації можуть вноситися зміни у вимоги або обмеження, що потребуватиме внесення відповідних змін у розділи проекту, в тому числі і в розділ архітектури. Вибір варіанта архітектури здійснюється з множини альтернатив, які конструюються на основі функціональних вимог із стандартних компонентів (патернів). Для підвищення обґрунтованості прийнятих рішень та автоматизації процесу використовуються методи оптимізації та багатокритеріального вибору[1]. Для оцінювання альтернатив по окремих критеріях якості найбільш ефективним є метод аналізу ієрархій (MAI) або його модифікований варіант. В цих методах відносна оцінка альтернатив визначається з використанням експертної інформації, і при включенні в розгляд нових альтернатив потрібно повторно проводити експертне оцінювання та розрахунки ваг альтернатив. Для оцінювання та вибору архітектури по множині критеріїв, як правило, використовують лінійну згортку [2].

Для вирішення цієї задачі можна також застосувати нелінійну скалярну згортку, в якій реалізований принцип «далі від обмежень». Однак, тут теж виникає проблема збіжності до оптимуму процедури симплекс-планування при визначенні ваг критеріїв якості. Для уникнення перерахованих проблем бажано побудувати цільову функцію для вибору архітектури в аналітичному вигляді, структура і параметри якої визначалися б об’єктивно, на основі експерименту, а не постулювались.

Пропонується для побудови цільової функції вибору архітектури використати МГУА в поєднанні з MAI. В цьому методі, для вибору моделі цільової функції генеруються різні структури моделей в обраному класі. Селекція моделей і прийняття рішення про завершення процесу відбувається за значенням критерію, обчисленого на послідовності експертних значень цільової функції, які визначаються методом аналізу ієрархій. Оскільки експертам необхідно визначати оцінки альтернатив по сукупності критеріїв якості, то критерій неузгодженості матриці парних порівнянь може перевищувати допустиме значення і отримані оцінки виявляться некоректними. Тому пропонується використовувати модифікований метод аналізу ієрархій, в якому ваги альтернатив визначаються з умови мінімізації неузгодженості матриці парних порівнянь.

Література.

1. Kharchenko, A.; Halay, I.; Bodnarchuk, I. Multicriteria architecture choice of software system under design and reengineering. 2016 XIth International Scientific and Technical Conference Computer Sciences and Information Technologies (CSIT). Anais. In: CSIT. Lviv, Ukraine: IEEE, 4–8, set. 2016.
2. Bodnarchuk, Ihor, et al. 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). IEEE, 2018. p. 292–297.