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: Методи реалізації процесів забезпечення якості в Agile-проєктах з використанням технології CI/CD.

Виконав(ла): студент(ка) 6 курсу, групи СНнм-61
спеціальності 122 Комп’ютерні науки
(шифр і назва спеціальності)

(підпис) Осідак Н.І. (прізвище та ініціали)

Керівник (підпис) Марценко С.В. (прізвище та ініціали)

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

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

Рецензент (підпис) (прізвище та ініціали)

Тернопіль
2025

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

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

Кафедра комп'ютерних наук
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Боднарчук І.О.
(підпис) (прізвище та ініціали)
"____" _____ 202__ р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня Магістр
(назва освітнього ступеня)

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

студенту Осідак Назарій Ігорович
(прізвище, ім'я, по батькові)

1. Тема роботи **Методи реалізації процесів забезпечення якості в Agile-проєктах з використанням технології CI/CD**

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

Затверджені наказом ректора від "*29*" *листопада* 2024 року № *4/7-1139*

2. Термін подання студентом завершеної роботи 20 травня 2025 р.

3. Вихідні дані до роботи Літературні джерела з тематики роботи

4. Зміст роботи (перелік питань, які потрібно розробити) ВСТУП 1 ОГЛЯД ДЖЕРЕЛ 1.1 Основні елементи практик CI та CT 1.2 Безперервна інтеграція 1.3 Безперервне тестування 1.4 Гнучкі методології розробки ПЗ 1.5 Удосконалення процесів для Agile-ПЗ 2 ЗАБЕЗПЕЧЕННЯ ЯКОСТІ 2.1 Поняття якості для гнучких методологій розробки 2.2 Модель часової динаміки визначення методів забезпечення якості 2.3 Ітерація процесу забезпечення якості 2.4 Забезпечення якості на рівні релізу 2.5 Покращення гнучкого тестування 3 ГНУЧКІ МЕТОДИ РОЗРОБКИ ПЗ 3.1 Безперервна інтеграція програмного забезпечення 3.2 Методи безперервної інтеграції, що використовуються розробниками 3.3 Елементи якості в процесі розробки 3.4 Проблеми та рішення практик безперервної інтеграції з використанням методів гнучкої розробки програмного забезпечення 3.5 Огляд практик тестування для гнучких технологій розробки 3.6 Практики безперервного тестування для гнучкого забезпечення якості 3.7 Проблеми та рішення для безперервного тестування в Agile Quality Assurance 4 ОХОРОНА ПРАЦІ ТА БЕЗПЕКА ВИСНОВКИ СПИСОК ДЖЕРЕЛ ДОДАТКИ

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

6. Консультанти розділів роботи

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

7. Дата видачі завдання _____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1.	Ознайомлення з завданням до кваліфікаційної роботи	14.04.25-15.04.25	*Виконано*
2.	Підбір наукових джерел по темі роботи	16.04.25-20.04.25	*Виконано*
3.	Переклад та опрацювання наукових джерел	20.04.25-24.04.25	*Виконано*
	по темі кваліфікаційної роботи		
4.	Виконання дослідження щодо теми	24.04.25-10.05.25	*Виконано*
	Кваліфікаційної роботи		
5.	Оформлення першого розділу	04.05.25-05.10.25	*Виконано*
6.	Оформлення другого розділу	05.05.25-13.05.25	*Виконано*
7.	Оформлення третього розділу	13.05.25-14.05.25	*Виконано*
8.	Виконання завдання до підрозділу "Охорона праці"	08.05.25-09.04.25	*Виконано*
9.	Виконання завдання до підрозділу "Безпека в	10.05.25-05.05.25	*Виконано*
	надзвичайних ситуаціях"		
10.	Оформлення кваліфікаційної роботи	05.05.25-31.05.25	*Виконано*
11.	Нормоконтроль	14.05.25-15.05.25	*Виконано*
12.	Перевірка на плагіат	15.05.25	*Виконано*
13.	Попередній захист кваліфікаційної роботи	16.05.25	*Виконано*
14.	Захист кваліфікаційної роботи	29.05.2025	

Студент _____ Осідак Н.І.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Марценко С.В.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

"Методи реалізації процесів забезпечення якості в Agile-проєктах з використанням технології CI/CD" // Кваліфікаційна робота освітнього рівня "Магістр" // Осідак Назарій Ігорович // Тернопільський національний технічний університет ім. І. Пулюя, факультет комп'ютерно-інформаційних систем і програмної інженерії, кафедра комп'ютерних наук, група СНнм-61 // Тернопіль, 2025 // с. – 70, рис. – 2, табл. – 5, джерел – 61.

Ключові слова: Agile, тестування програмного забезпечення, забезпечення якості, безперервна інтеграція, безперервне тестування

З огляду на трансформацію процесу розробки програмного забезпечення, фахівці з якості змушені адаптуватися до нових умов. Основна ідея гнучкого підходу полягає в тому, що якість продукту оцінюють як розробники, так і користувачі, що сприяє покращенню загальної якості системи. Хоча цей підхід потенційно підвищує якість кінцевого продукту, він водночас передбачає зменшення ролі команди з забезпечення якості. Agile-методологія націлена на прискорення процесу створення програмного забезпечення та зниження витрат.

У контексті гнучкого забезпечення якості (QA) та розробки ключовими методами є безперервна інтеграція (CI) та безперервне тестування. Програмісти регулярно застосовують, перевіряють та тестують зміни в коді протягом усього циклу розробки. У гнучкому забезпеченні якості CI та безперервне тестування передбачають рутинну інтеграцію коду, автоматизоване тестування та оперативний зворотний зв'язок, що забезпечує швидкий та ітеративний розвиток високоякісного програмного забезпечення.

ANNOTATION

“Implementing of Quality Assurance Processes Methods in Agile Projects Using CI/CD Technology” // Master’s degree qualification paper // Osidak Nazarii // Ternopil Ivan Puluj National Technical University, Faculty of Computer Information Systems and Software Engineering, Computer Science Department, group СНнм-61 // Ternopil, 2025 // p. – 70, fig. – 2, tables – 5, references – 61.

Key words: Agile, software testing, quality assurance, continuous integration, continuous testing.

Given the transformation of software development processes, quality specialists are forced to adapt to new conditions. The main idea of the agile approach is that product quality is assessed by both developers and users, contributing to the overall improvement of the system's quality. Although this approach potentially enhances the final product's quality, it simultaneously implies a reduced role for the quality assurance team. The Agile methodology aims to speed up the software development process and reduce costs.

In the context of agile quality assurance (QA) and development, the key methods are continuous integration (CI) and continuous testing. Programmers regularly apply, verify, and test code changes throughout the entire development cycle. In agile QA, CI and continuous testing involve routine code integration, automated testing, and prompt feedback, ensuring rapid and iterative development of high-quality software.

ЗМІСТ

ВСТУП

Актуальність задачі. В сучасних умовах розробки програмного забезпечення, що характеризуються швидкими змінами вимог та необхідністю частого випуску нових версій, гнучкі (Agile) методології набули значного поширення. Вони дозволяють командам швидко реагувати на зворотний зв'язок та адаптуватися до нових викликів. Однак, висока динаміка Agile-процесів ставить нові вимоги до забезпечення якості програмних продуктів. Задача впровадження ефективних процесів контролю якості в гнучкі технології розробки є надзвичайно актуальною, і ключову роль у її вирішенні відіграють технології неперервної інтеграції та неперервного розгортання (CI/CD).

Гнучкі методології, такі як Scrum чи Kanban, зосереджені на ітеративній та інкрементній розробці. Це означає, що програмний продукт постійно змінюється та доповнюється новою функціональністю. У таких умовах традиційні підходи до контролю якості, що передбачають тривалі фази тестування наприкінці циклу розробки, стають неефективними та можуть гальмувати процес. Виникає потреба в інтеграції процесів забезпечення якості безпосередньо в цикл розробки, роблячи їх невід'ємною частиною кожного етапу.

Важливим аспектом у цьому контексті є проєктування та оцінювання програмної архітектури. Як зазначається у дослідженнях, при гнучких методах управління проєктами питання проєктування архітектури програмних систем набуває особливого значення [5]. Архітектура має бути достатньо гнучкою та адаптивною, щоб витримувати постійні зміни та еволюцію системи. При цьому виникає задача багатокритеріального вибору програмної архітектури, що враховує динамічну корекцію атрибутів якості [1]. Необхідні адаптивні методи для оцінювання та вибору архітектури в умовах гнучких технік проєктування [2]. Це підкреслює, що якість програмного продукту закладається ще на етапі його архітектурного проєктування, і цей процес має бути тісно пов'язаний із

загальною системою забезпечення якості. Задача проєктування програмної архітектури є важливою складовою процесів забезпечення якості [4].

Технології CI/CD є потужним інструментом для реалізації ефективного контролю якості в Agile-середовищі. Неперервна інтеграція передбачає часте злиття коду від різних розробників у спільне сховище, після чого автоматично запускаються процеси збірки та тестування. Це дозволяє швидко виявляти та виправляти інтеграційні проблеми та дефекти на ранніх етапах розробки. Неперервне розгортання автоматизує процеси доставки та розгортання програмного забезпечення, що дає можливість оперативно надавати користувачам нові версії продукту, отримуючи швидкий зворотний зв'язок.

Інтеграція автоматизованого тестування (модульного, інтеграційного, функціонального) у CI/CD конвеєри стає стандартом для забезпечення високої якості в Agile-проєктах. Це дозволяє виконувати тести після кожної зміни коду, гарантуючи, що внесені зміни не порушили наявну функціональність та не внесли нових дефектів. Такий підхід, підтриманий глобальними процесами проєктування програмного забезпечення [3], сприяє створенню стабільнішого та надійнішого продукту.

Таким чином, актуальність задачі впровадження процесів контролю якості в гнучкі технології розробки програмних продуктів з використанням технологій CI/CD зумовлена необхідністю забезпечення високої якості програмного забезпечення в умовах швидких ітерацій та постійних змін, характерних для Agile. CI/CD надає необхідну автоматизацію та інструменти для безперервного тестування та моніторингу якості, що є критично важливим для успіху сучасних Agile-проєктів.

Мета роботи.

Метою цього дослідження є виявлення проблем та рішень, пов'язаних із практичним впровадженням безперервної інтеграції та тестування в умовах гнучкої розробки програмного забезпечення. У роботі здійснено огляд гнучких методів, особливостей гнучкого забезпечення якості, практик CI у гнучкій розробці, а також розглянуто труднощі та шляхи їх подолання. Додатково

проаналізовано практики безперервного тестування у гнучкому QA, а також пов’язані з ними виклики та можливі рішення.

На основі сформульованої актуальності та поданої мети дослідження, можна визначити наступні задачі, вирішення яких дозволить досягти поставленої мети:

1. Провести аналіз гнучких методологій розробки програмного забезпечення та визначити особливості інтеграції процесів забезпечення якості на різних етапах їх життєвого циклу. Ця задача сфокусована на теоретичних засадах Agile та їх взаємозв'язку з якістю, що є фундаментом для розуміння подальших практичних аспектів. Вона включає огляд гнучких методів та особливостей гнучкого забезпечення якості.

2. Дослідити сучасні практики та інструменти впровадження безперервної інтеграції (CI) в Agile-проєктах, виявивши типові проблеми та перешкоди на шляху їх практичної реалізації. Ця задача заглиблюється у практичну сторону CI в Agile, як зазначено в меті, включаючи практики CI та пов'язані з ними труднощі.

3. Систематизувати підходи та практики безперервного тестування в контексті гнучкого забезпечення якості (Agile QA), проаналізувавши пов'язані з ними виклики та обмеження. Ця задача деталізує аспекти безперервного тестування в Agile QA, як того вимагає мета дослідження.

4. Розробити набір рекомендацій та можливих шляхів подолання проблем, пов'язаних із практичним впровадженням безперервної інтеграції та тестування, з метою підвищення ефективності процесів забезпечення якості в умовах гнучкої розробки програмного забезпечення. Ця задача є синтезуючою і спрямована на пошук рішень та рекомендацій для практичного впровадження CI/CD та тестування, що безпосередньо відповідає меті дослідження – виявлення проблем та рішень.

Об’єкт дослідження: процеси гнучкої (Agile) розробки програмного забезпечення.

Предмет дослідження: методи та засоби впровадження процесів забезпечення якості в Agile-розробку з використанням технологій CI/CD.

Наукова новизна отриманих результатів.

Проведено огляд літературних джерел за тематикою кваліфікаційної роботи з метою формування фреймворку впровадження процесів забезпечення якості в гнучкій розробці програмних продуктів на основі використання технологій CI/CD. Отримані результати дають цінні рекомендації практикам SPI щодо розробки відповідних стратегій впровадження SPI. Таким чином, отримав подальший розвиток метод впровадження практик DevOps для гнучких методологій розробки програмних продуктів.

Практичне значення отриманих результатів.

У цій роботі розглянуті питання, що загалом стосуються практик DevOps з застосування безперервного тестування, і котрі є дуже важливими для забезпечення якості продукту. Воно дозволяє швидше розробляти програмне забезпечення, знижує ймовірність виникнення проблем під час виробництва та покращує загальну якість програми. Безперервне тестування в DevOps передбачає вибір відповідних інструментів автоматизації тестування, розробку ефективної системи автоматизації тестування, інтеграцію тестування в конвеєр DevOps та дотримання найкращих практик DevOps. Це кроки, які необхідно зробити для впровадження безперервного тестування..

Апробація результатів та особистий внесок здобувача.

Основні положення роботи доповідались, розглядались та обговорювались на науковій конференції Тернопільського національного технічного університету імені Івана Пулюя. Результати кваліфікаційної роботи опубліковані у тезах студентської наукової конференції "Актуальні задачі сучасних технологій – 2024", яка проводилась у ТНТУ.

1 ОГЛЯД ЛІТЕРАТУРНИХ ДЖЕРЕЛ ЗА ТЕМОЮ РОБОТИ

1.1 Основні елементи практик безперервної інтеграції та безперервного тестування

Фактор якості може бути ключовим елементом та ознакою розвитку та успіху гнучкого методу розробки програмного забезпечення, або ж ознакою невдачі у його виробництві. Для досягнення вищого рівня якості вкрай важливо стежити за складовою успіху [2].

Сьогодні багато компаній розглядають різні процедури забезпечення якості, намагаючись знайти рішення, яке дозволить їм подолати труднощі, пов'язані з підтримкою якості в гнучких середовищах [4]. Незважаючи на те, що в минулому було доведено їхню ефективність, вони більше не підходять для гнучкого середовища. Наприклад, огляд того, що можна досягти на кожному етапі життєвого циклу розробки програмного забезпечення (SDLC), є однією з областей, на якій необхідно зосередитися, щоб підтримувати необхідний рівень якості. Більш формальні та ретельні технічні оцінки, такі як інспекції та перегляд дизайну тощо, повинні бути включені до цього огляду. Ця оцінка гарантує, що дефекти та інші проблеми будуть виявлені та виправлені на ранній стадії, покращуючи якість кінцевого результату.

Найновіші методи програмування можна розглядати як еволюційні, ітеративні та інкрементні. Це включає такі методи, як, наприклад, Enterprise Unified Process (EUP), Rapid Application Advancement (RAD), Extreme Programming (XP) та Rational Unified Process (RUP). Багато сучасних процесів також є гнучкими. Експерти з якості повинні адаптуватися, коли змінюється основна природа розробки програмування. Це дослідження пояснює поширені гнучкі методи розробки програмного забезпечення та показує, як їх впровадження створює програмне забезпечення, яке володіє рівнем значно вищої якості, ніж те, що зазвичай створюють традиційні команди розробників програмного забезпечення.

Згідно [7], створення підходів, орієнтованих на тестування, може допомогти гнучким методам досягти мети створення високоякісного програмного забезпечення. За словами автора цієї роботи, «якість є невід'ємним фактором гнучкого методу», що видається розумним, враховуючи значний тиск, який чиниться на команди, щоб писати тест-кейси перед створенням коду [7]. Тестування та якість йдуть пліч-о-пліч, оскільки основною метою тестування є виявлення дефектів у продукті до його відправлення клієнтам. Це надає розробнику продукту можливість покращити його якість, вирішуючи будь-які виявлені проблеми. Це свідчить про те, що ключовим завданням для процесів гнучкого забезпечення якості є надання протестованого, функціонального та сертифікованого клієнтом програмного забезпечення після завершення кожного нового релізу.

Потреба в широкому асортименті програмних продуктів зростає разом зі складністю програмного забезпечення щодня. Це вимагає надання потужного інструменту, який може збалансувати результат та якість. Практика застосування метрик програмного забезпечення до процесу його розробки та до програмного продукту є критично важливим завданням, яке вимагає навчання та дисципліни, і яке надає знання про стан процесу розробки програмного забезпечення та/або продукту стосовно цілей, які мають бути досягнуті. Ця дисципліна відома як забезпечення якості, і вона є основним фактором успіху для кожного проекту з розробки програмного забезпечення. Заходи із забезпечення якості дозволяють розробляти програмне забезпечення з мінімальним набором вимог та сприяють частим змінам цих потреб. Гнучка методологія наразі є одним з основних підходів, що використовуються більшістю галузей розробки програмного забезпечення. Навіть якщо ця процедура може швидко створити продукт, ми не можемо гарантувати його якість, доки не впровадимо заходи із забезпечення якості (SQA) у процес [6].

Процес створення програмного забезпечення, яке може адаптуватися та реагувати на зміни відповідно до змінних вимог клієнта, можна назвати гнучким забезпеченням якості. Це показує, що ключовим елементом забезпечення якості

в гнучкій методології є регулярна поставка програмного забезпечення, яке було оцінено, перевірено на функціональність та схвалено клієнтом наприкінці кожної ітерації. Гнучкий підхід до забезпечення якості перевершує традиційні методи забезпечення якості програмного забезпечення, вирішуючи проблеми якості більш просунутим способом.

1.2 Безперервна інтеграція

Практика безперервної інтеграції раніше привертала значну увагу, і зараз вона добре сформована та зрозуміла. Численні інші операції, пов'язані з управлінням початковим кодом (Source Code Management – SCM), потребують постійної інтеграції, і ми очікуємо побачити їх розвиток відповідно до наших очікувань. Це призведе до створення нових найкращих практик, декотрі з яких можуть бути пов'язані з SCM, для уточнення найкращих практик. Автори роботи [7] пропонують у своїй статті першу спробу визначити практики SCM, що підходять для гнучкого середовища. Ми також сподіваємося, що ці практики розвинуться в більш практично придатні практики.

Кент Бек вперше представив безперервну інтеграцію як метод розробки програмного забезпечення у своїй книзі «Пояснення екстремального програмування». Цей метод часто називають абревіатурою CI [8] та про прискорення співпраці розробників. Програма часто модифікується різними розробниками, і кожна інтеграція гарантує функціональність продукту шляхом запуску автоматизованих тестів та вжиття інших запобіжних заходів. Згідно з літературою з програмної інженерії та прикладами, було показано, що використання безперервної інтеграції покращує якість програмного забезпечення, зусилля та швидкість тестування, частоту релізів програмного забезпечення та інші фактори.

Для здійснення безперервної інтеграції використовується власне сервер безперервної інтеграції (CI). Коли розробник публікує зміни, CI-сервери негайно запускають збірку безперервної інтеграції, показують результати та, за

бажанням, повідомляють автора про ці результати. Це допомагає розробникам практикувати безперервну інтеграцію. Хоча більшість програмних проектів, які виконують безперервну інтеграцію, роблять це за допомогою CI-серверів, цю дію також можна виконувати вручну.

Розробник вносить кілька змін до вихідного коду, і ці модифікації часто відбуваються кілька разів на тиждень або щодня. Крок інтеграції коду вважається найважливішою частиною всього життєвого циклу DevOps. Процес постійної інтеграції нового коду в старий включає створення нових кодів, які дозволяють додавати нові можливості. Помилки виявляються досить рано у вихідному коді під час цього раунду тестування. Інструменти для модульного тестування, перевірки коду, інтеграційного тестування, компіляції та упаковки будуть використовуватися розробниками під час написання нового коду для програми. Функціональність програми збільшиться завдяки цьому додатковому коду. Постійна інтеграція цього нового коду в існуючий вихідний код допомагає відобразити будь-які зміни, з якими кінцеві користувачі можуть зіткнутися в результаті оновлення коду. Jenkins – це надійний інструмент DevOps, який часто використовується для отримання найновішого вихідного коду та перетворення збірок у виконувані форми. Ці переходи відбуваються гладко, а змінений код упаковується перед відправкою на наступний крок, який є або продакшн-сервером, або сервером, що використовується для тестування [9].

1.3 Безперервне тестування

Розробники зазвичай виконують крок безперервного тестування перед фазою безперервної інтеграції. Залежно від змін, внесених до коду програми протягом життєвого циклу DevOps, цю фазу можна змістити так, щоб вона йшла після фази безперервної інтеграції. Тепер, коли програма створена, вона постійно тестується для виявлення будь-яких проблем. Docker-контейнери використовуються для моделювання тестового середовища. Автоматизоване тестування – це інструмент, що економить час і зусилля розробників. Процес

оцінки тестів виграє від звітів, створених автоматичним тестуванням. Тепер набагато простіше аналізувати тестові випадки, які були невдалими. Остаточний набір тестів не містить жодних помилок після завершення процедури тестування користувачами (User Acceptance Testing – UAT). Використовуючи ці інструменти, можна запланувати виконання тестових випадків протягом певного періоду часу. Для оновлення вихідного коду протестований код зрештою повертається на фазу безперервної інтеграції. Це робиться для того, щоб код працював без помилок. Дві ключові процедури, які необхідно виконати, щоб гарантувати, що код програми буде постійно оновлюватися, – це безперервне тестування та інтеграція. Під час фази безперервного зворотного зв'язку ці покращення оцінюються [10].

1.4 Гнучкі методології розробки ПЗ

Від початку, коли вимоги отримані, до самого кінця, коли програмний продукт доставляється замовникові, тестується та збирається інформація від користувача, гнучка методологія дотримується вибраної моделі життєвого циклу розробки програмного забезпечення. Менше документації, що надається в проекті, та більший акцент на кодуванні – це те, що відрізняє гнучкі методи від інших методологій [3].

Scrum, екстремальне програмування (XP) та інші гнучкі методології базуються на застосуванні перевірених часом методів, які широко визнані для підвищення якості розробки програмного забезпечення. Можна стверджувати, що використання найкращих практик робиться для того, щоб полегшити інтеграцію процесів забезпечення якості програмного забезпечення (SQA) у проект. Ключову структуру підтримки проекту забезпечують заходи із забезпечення якості (QA), які відбуваються під час процесу розробки програмного забезпечення [11]. З 1990-х років гнучка методологія отримала суттєве обговорення в різних академічних публікаціях, включаючи книги, есе та

журнали. Однак, досліджень на тему контролю якості в гнучких методологіях розробки програмного забезпечення було проведено не так багато.

Коротко кажучи, це фреймворк для розробки програмного забезпечення, який базується на високій частоті та швидкості ітерацій, щоб як забезпечити програму, так і врахувати модифікації, що надходять від замовника. Гнучкі методи стають дедалі популярнішими в сучасний час як засіб подолання динаміки корпоративного розширення. Це пов'язано з тим, що гнучкі методи дозволяють швидко реагувати на постійно мінливі вимоги та забезпечують продукт вищої якості за коротший час [12]. Завдяки своїй здатності керувати мінливими потребами, акценту на співпраці клієнта та розробника, а також ранній поставці програмного забезпечення, гнучкі методології набули популярності в комерційному світі з кінця 1990-х років. Індустрія розробки програмного забезпечення здебільшого використовує гнучкі підходи, які пропонують певні переваги в управлінні часом та вимогами до системи.

Численні переваги гнучких підходів включають підвищення задоволеності клієнтів, підвищення успішності проектів та вищу якість розробки. Гнучкі підходи до розробки покращують координацію між командами. Ці методи сприяють швидкому виходу на ринок; однак процес доставки уповільнюється через відсутність координації між розробниками та операційною командою. Гнучкі компанії самостійно здійснюють розробку, тестування та розгортання. Ці завдання займають багато часу, що призводить до затримки процесу випуску. Командам експлуатації та тестування не приділяється значної уваги в гнучких методах. Процеси тестування, доставки та розгортання є відповідальністю цих команд. Продукт розробляється командами розробників значно швидше, що призводить до відставання інших команд, що може призвести до затримки процесу доставки. В результаті проблеми та помилки проявляються на ранніх етапах процесу виробництва продукту.

Методологія DevOps покращує комунікацію, доставку, продуктивність та інтеграцію між операційним персоналом, тестувальниками та розробниками. Метою DevOps є підвищення задоволеності клієнтів шляхом підвищення якості

та безперебійної доставки. Використання DevOps кількома європейськими банками призвело до помітного 25-відсоткового покращення надання ними онлайн-послуг [13]. Впровадження DevOps спирається на використання автоматизованого конвеєра розгортання, що зменшує повторювані ручні завдання, пов'язані з безперервною інтеграцією. Розробка та експлуатація – це просто дві частини процесу DevOps [14]. Однак інші зацікавлені сторони, включаючи розробників, тестувальників, аналітиків, а також персонал, відповідальний за управління базами даних та їхню безпеку, активно беруть участь у впровадженні практик DevOps.

DevOps може допомогти подолати труднощі безперервної доставки. Цифрові гіганти, такі як Amazon та Netflix, вже використовують DevOps для доставки на ринок точно розроблених, орієнтованих на клієнта програмних рішень [15**Error! Reference source not found.**]. Застосування концепцій DevOps для реалізації безперервної доставки усуває традиційне тестування, моніторинг та інтеграцію коду, прискорюючи випуск продукту користувача. Використовуючи елементи багаторазового використання та сприяючи широкому використанню систем управління програмним забезпеченням, парадигма DevOps сприяє безперервній доставці.

Принципи безперервної доставки є основою практики DevOps. Швидкий випуск скорочує час, необхідний для розробки програми та доставки високоякісного програмного забезпечення. Тестування на більш загальному рівні проводиться в компаніях-розробниках програмного забезпечення для оцінки мети та встановлення якості програми. З іншого боку, швидка перевірка якості не повинна передбачатися для безперервної інтеграції. Тестування проводиться вручну, і немає автоматизації жодного з тестових випадків, тому швидкість виявлення помилок буде поступово зростати. Через обсяг роботи та часу, необхідних для цієї процедури, доставка продукту буде затримуватися.

Безперервне тестування – це одна зі стратегій, яка може бути використана для вирішення цих труднощів. Безперервне тестування дозволяє надавати своєчасний зворотний зв'язок без участі людини. Автоматизоване тестування є

компонентом безперервного тестування. Цей моніторинг та покращення якості тестових випадків здійснюється за допомогою автоматизації [16]. Стратегії безперервного тестування використовуються в тестовій діяльності, яка є частиною стратегії DevOps. Це допомагає виявляти недоліки та помилки в різних програмних компонентах. Тестування відбувається інакше, ніж традиційне тестування, завдяки принципам DevOps, які можна розглядати як сукупність методів і засобів управління програмним прдуктом. Тестування було одним із етапів життєвого циклу розробки програмного забезпечення до появи DevOps. Однак традиційне тестування не проводиться протягом усієї фази проекту, яка триває від початку до завершення. У DevOps тестування має бути впроваджено з самого початку, щоб гарантувати високу якість програмного забезпечення та розділити відповідальність за цю якість між командою розробників та операційною командою [17].

1.5 Удосконалення процесів для Agile-програмного забезпечення

Ретельне вивчення гнучких процесів призводить до висновку, що гнучкі підходи – це сукупність процедур та дій, які скорочують час, необхідний для створення програмного забезпечення, та пропонують передові методи для врахування швидкозмінних бізнес-вимог. Ці відносно нові методології забезпечення якості зрештою повинні перерости у встановлені стандарти розробки програмного забезпечення. Однак, як показало [18], саме забезпечення можливості повторного використання в гнучкій розробці програмного забезпечення дозволяє швидше завершити процес розробки програмного забезпечення.

Усі гнучкі підходи мають разюче схожі процеси в своїх численних ітераціях, оскільки вони базуються на одному й тому ж наборі принципів. Цікаво відзначити, що навіть розробники гнучких методологій зараз приймають використання методів з інших гнучких методологій, якщо вони підходять для конкретної ситуації [19]. Поглиблений аналіз гнучких методологій показує, що

ці методи використовують низку різних моделей, заснованих на реальних сценаріях, для вирішення тих самих проблем.

Екстремальне програмування – це парадигма, яка розглядає процес розробки програмного забезпечення як групову діяльність, де розробники співпрацюють. Методологія розробки адаптивних систем, також відома як ASD, використовується для підходу до проектів розробки програмного забезпечення з точки зору складних самоадаптивних систем [20]. Невелика кількість доступних гнучких методологій була використана для демонстрації процесу оцінювання. Вибір методологічних елементів залежить від великої суб'єктивності. Метою цієї роботи не є надання вичерпної таксономії підходів. Для більш ретельної таксономії було використано невелику кількість доступних гнучких методологій. Як результат, об'єкти, включені сюди, були обрані для демонстрації подібності між різними гнучкими методологіями. Виявлення цих подібностей дозволить розробникам, які не впевнені, яку гнучку техніку обрати, бути краще підготовленими. Хоча використання правильної методики у проекті розробки програмного забезпечення може не миттєво призвести до успіху проекту або створення високоякісного продукту, це може призвести до провалу проекту.

2 ЗАБЕЗПЕЧЕННЯ ЯКОСТІ ПРИ ГНУЧКІЙ РОЗРОБЦІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Поняття якості для гнучких методологій розробки програмних продуктів

У роботі [7] якість описується, як прямий результат власне гнучких методологій з точки зору гнучкої розробки. Через ці характеристики гнучкої якості, вичерпна документація більше не є необхідною; проте це призводить до того, що слово «гнучка якість» стає дещо розпливчастим і його стає складнішим для визначення. Оскільки більшість гнучких методів пропонують лише невелику кількість пропозицій щодо того, як слід перевіряти та підтримувати різні атрибути якості, інтеграція підходів до тестування з гнучкими операціями може бути складною. Це вірно, навіть якщо найпоширенішою діяльністю SQA в наш час є оцінка численних атрибутів якості. За даними [21] були виділені основні задачі, які гнучкі принципи ставлять перед тестуванням з традиційної точки зору. Автори наголосили на необхідності проведення тестування в короткі цикли часу, уникаючи перевищення дозволеної тривалості тестування, як одне з питань, яке необхідно вирішити, щоб задовольнити критерії гнучкої концепції безперервної доставки корисного програмного забезпечення.

Найскладнішою частиною «реагування на зміни навіть на пізніх стадіях розробки» є те, що тестування не може проводитися на основі реалізованих потреб. Це створює проблему. Через те, що гнучкий метод залежить від особистісної взаємодії, інженери та бізнес-учасники процесу тестування можуть зіткнутися з комунікаційними збоями, які можуть призвести до розбіжностей. Крім того, найважливішою ознакою успіху є наявність робочого програмного забезпечення, тому інформація про якість необхідна як на ранніх етапах процесу розробки, так і протягом усього процесу.

Автори [21] спостерігали конфлікти в гнучкому тестуванні, враховуючи такі концепції, коли йдеться про основні принципи тестування. Наприклад,

однією з основних характеристик тестування є незалежність. Однак, за допомогою гнучких підходів розробники створюють тести для своїх програм, а тестувальники або міняються ролями з розробниками, або самі стають розробниками. Це ознака того, що тестувальники недостатньо незалежні, оскільки вони інтегровані в команду розробників. Крім того, їм слід утримуватися від тестування власного коду, оскільки важко знайти помилки, а тестування не визначає, чи задовольняє кінцевий продукт потреби клієнта або чи розуміє їх.

Крім того, тестування вимагає від людей певного набору навичок і знань, щоб бути успішними та ефективними. З іншого боку, споживче тестування є життєздатним для досягнення якості, якщо вони мають унікальний досвід обробки або проведення тестування, що, на жаль, не завжди так. Клієнти можуть, тим не менш, тестувати продукти, щоб забезпечити якість. Пошук правильного результату тестування та виявлення невідповідностей програми – це процеси, які називаються «проблемою оракула». Це створює ще одну складність. Гнучкі підходи використовують великий відсоток автоматизованих тестів, що ставить питання про те, чи достатньо цих тестів для виявлення дефектів у коді.

Деструктивний підхід, який зосереджений на дослідженні та виявленні проблем, є одним із найвідоміших методів тестування програмного забезпечення. Згідно з [22], тестування на відповідність проводиться для того, щоб переконатися, що об'єкт відповідає певній вимозі та/або законодавчій вимогі. Метою цього дослідження є оцінка ефективності програмної системи у виконанні необхідних специфікацій та визначення її несправності.

Гнучкі методи можуть призвести до системних збоїв, незважаючи на проходження модульних тестів, оскільки вони надають більшого пріоритету опису якостей продукту, ніж дослідженню фундаментальних причин. Однак процедура тестування не лише шукає недоліки, але й збирає дані, які можна використовувати для покращення якості продукту. Хоча вона не розкриває досягнутий рівень якості продукту та не полегшує оцінку досягнутої якості,

гнучкий метод значною мірою спирається на дотримання встановлених протоколів та перевірку їх дотримання.

2.2 Модель часової динаміки визначення методів забезпечення якості в гнучких підходах розробки

Можна стверджувати, що впровадження додаткових методів тестування зрештою призводить до впровадження процедур гнучкої розробки. У цьому розділі проаналізуємо підходи, використані в роботі [21], щоб надати більш ґрунтовне пояснення та ясність щодо недоліків і проблем при використанні традиційних методів тестування в гнучких методологіях розробки. Виходячи з концепції використаних в цій роботі понять «горизонтів часу серцебиття, ітерації та релізу» на основі виділених в [22] циклів контролю (Cycles of Control – CoC), автори використовували модель часової динаміки для визначення методів забезпечення якості в сучасних гнучких підходах.

Фреймворк CoC є загальним прикладом фреймворку, який може бути використаний для зображення поступової та ітеративної розробки програмного забезпечення. Через це фреймворк CoC може бути використаний для ілюстрації гнучкої розробки. Дизайн базується на ідеї темпу часу, в якому заздалегідь визначений період часу поділяється на кілька періодів різної тривалості, кожен з яких має бюджет і термін виконання. Контрольна точка розташована наприкінці кожного окремого сегмента і використовується для проведення аналізу процесу.

Зміни до планів також можуть бути внесені, якщо буде визначено, що вони необхідні. Оскільки коригування можна вносити лише в певних контрольних точках, можливі як стійкість, так і створення гнучкості, що дозволяє одночасні зміни плану та реагування на зміну умов середовища через заздалегідь визначені інтервали часу. Потік розробки продукту визначається цими інтервалами, які також називають часовими рамками. Графік часового блоку (дата виконання) визначається відповідно до ідеї темпу часу, але його обсяг (створення функціональності) – ні. У випадку, якщо неможливо реалізувати всі критерії

продукту до встановленої кінцевої дати, обсяг проекту буде скорочено, щоб гарантувати його готовність до встановленої кінцевої дати. Як наслідок, вимоги необхідно ранжувати (пріоритезувати), щоб команда могла самостійно підібрати відповідний обсяг робіт. Ілюстрацію циклів елементів формування контролю можна побачити на рисунку 2.1.

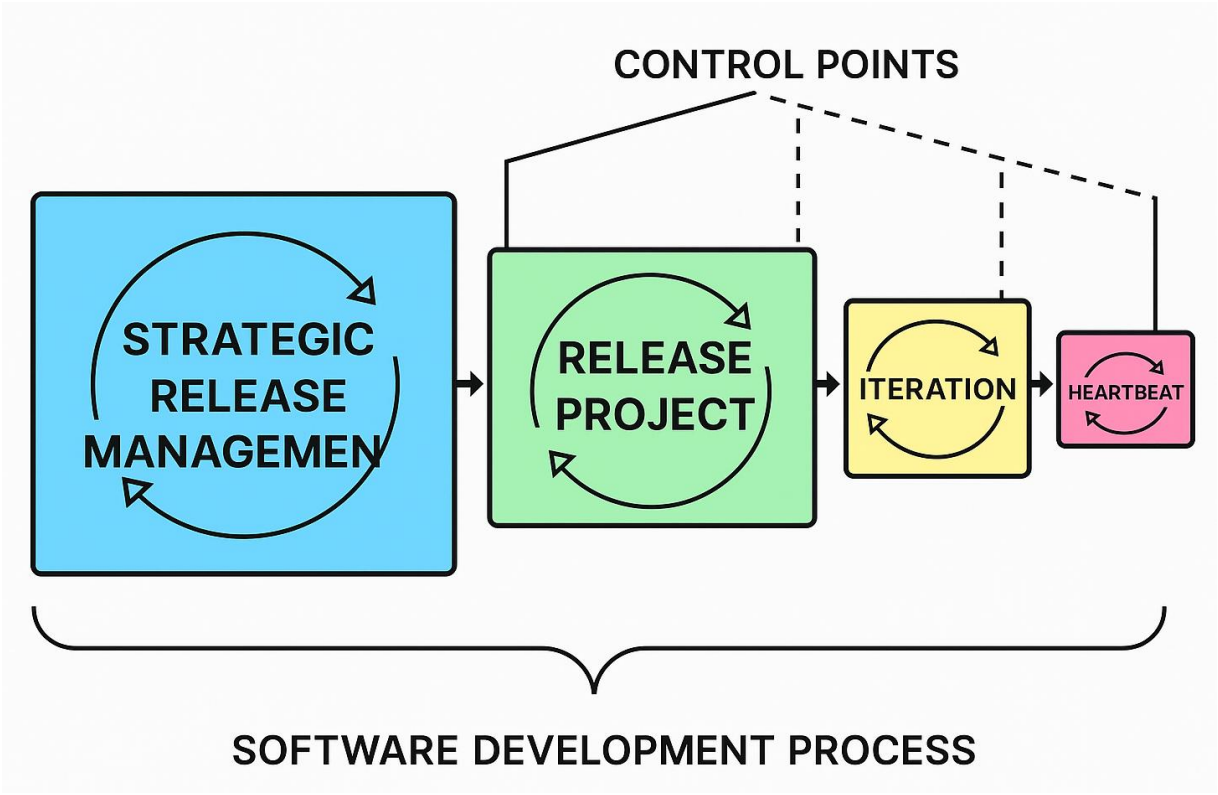

Рисунок 2.1 – Цикли управління якістю в гнучких проектах

Створення довгострокових стратегій для портфелів продуктів та проектів компанії включено до стратегічного управління релізами. Воно виступає життєво важливою ланкою між фактичним процесом створення продукту та прийняттям стратегічних рішень. Кожен реліз продукту успішно управляється в межах циклу проекту релізу як «проект з часовими рамками», який виконується. Кожен проект організований у ітерації з часовими рамками, де створюється частина кінцевого релізу продукту. Цикли служать таймером та регулятором для щоденної діяльності [22]. Рисунок 2.2 ілюструє це, показуючи цикли у вигляді часової шкали. Це показує, як часовий горизонт стратегічного управління

релізами охоплює два окремі проекти релізу, включає три окремі взаємодії та синхронізує роботу з щоденними циклами.

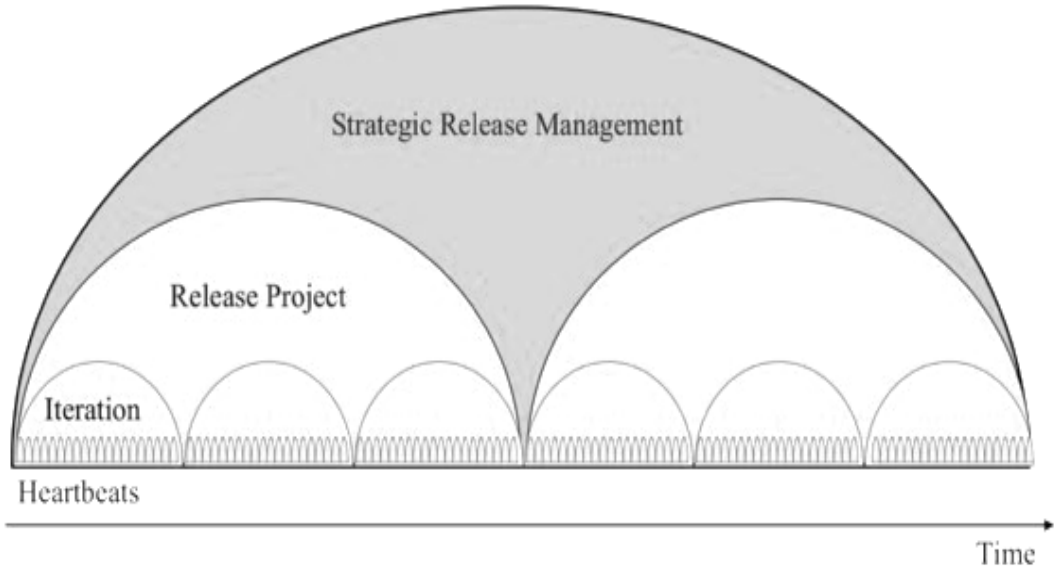

Рисунок 2.2 – Хронологія циклів управління якістю в гнучких проектах

Жодна додаткова фаза тестування не використовується для відкладення операцій забезпечення якості на рівні heartbeat. Натомість ці завдання виконуються, як частина обов'язків з проектування та кодування на етапі впровадження. Це стосується кожного учасника команди незалежно від того, хто відповідає за виконання цих дій: розробник чи тестувальник. Ці дії надають розробникам негайний зворотний зв'язок, дозволяючи їм керувати процесом розробки у правильному напрямку.

Практики забезпечення якості heartbeat включають методи, які «вносять якість у функціональність під час її впровадження», що означає, що завдання впровадження не вважаються завершеними, доки ці процедури забезпечення якості не будуть виконані. Ці процедури включені до обов'язків з проектування та кодування [21]. Вони служать міцною основою для процесу гнучкої розробки та ставлять за мету гарантувати ефективне виконання кожного завдання розробки, оскільки ці практики надають розробникам негайний зворотний

зв'язок. Автоматизоване модульне тестування виступає еталоном забезпечення якості.

Таблиця 2.1 – Процедури забезпечення якості для гнучких підходів на часових горизонтах

	Extreme Programming	Feature Driven Development	Crestal Clear	DSDM
Ітерація релізу	Оцінка результатів приймальних випробувань	Тестування системи	–	• Інтеграційне, системне та приймальне тестування всередині кожного таймбоксу • Тестування користувачами‖ • Еволюційне прототипування» • Огляди документів
heartbeat	• Розробка на основі тестування • Безперервна інтеграція» • Парне програмування» • Приймальні випробування‖ • Колективне володіння кодом» • Стандарти кодування • Простий дизайн та безперервний рефакторинг • Клієнт на місці	• Модульне тестування • Звичайні збірки • Перевірка проекту • Перевірка коду • «Індивідуальне володіння кодом»	• Автоматизовані тести та часта інтеграція • Парне програмування» • Осмотична комунікація • Легкий доступ до досвідчених користувачів	• Модульне тестування • Зворотні зміни • Активна участь користувачів

Кожен фрагмент коду, написаний розробниками, має пройти модульне тестування, а просування вперед розглядається лише після завершення та успішного проходження тестів. Кожне цикл heartbeat також має бути успішним, щоб багато функцій могли взаємодіяти одна з одною та координувати свої дії. Коли йдеться про гнучкі підходи на часовому горизонті ітерації, забезпечення якості (QA) є дуже потужною дисципліною. Часовий горизонт heartbeat

забезпечує основу для більшості стратегій забезпечення якості, як зазначено в таблиці 2.1. Це пов'язано з акцентом на модульному тестуванні, частих збірках, коді та дизайні.

На горизонті часу ітерації забезпечення якості охоплює процедури, спрямовані на досягнення цілей ітерації. Це охоплює всі фази впровадження, тестування та перевірки, які не застосовуються до кожної окремої функції heartbeat. Горизонти часу ітерації відповідають за велику кількість завдань, що виконуються професійними тестувальниками. Ці обов'язки включають оцінку надійності, продуктивності та інших характеристик якості системи.

Функціональне тестування – це лише один з видів тестування, який може бути виконаний у системі. Загалом, діяльність, якою займаються професійні тестувальники, лише опосередковано пов'язана з процесом розробки. Ці експерти відповідають за написання та проведення тестування для всієї ітерації, і вони повинні координувати свої зусилля з розробниками. Важливо відстежувати зростання, прогрес роботи та постійне надання високоякісної інформації, оскільки завдання ітерації обмежені в часі.

Один зі способів зробити це – через зустрічі з питань heartbeat. Як зазначено в таблиці 2.1, на горизонті часу ітерації гнучких методологій набагато менше практик забезпечення якості, ніж на горизонті часу heartbeat. На цьому рівні не так багато процесів чи дій, спеціально визначених для цілей забезпечення якості. Наприклад, екстремальне програмування (XP) залежить від надійних імпульсних методів, а часовий горизонт ітерацій відповідає виключно за вимірювання прогресу [23].

Забезпечення якості продукту на часовому горизонті релізу є основною метою процесу забезпечення якості релізу. Приклади завдань включають тестування, яке неможливо завершити протягом відведеного часу для ітерації, завдання, надані іншій групі тестування, та тестування в різних середовищах. Гарною загальною стратегією для залучення релізів із забезпеченням якості є наявність окремих ітерацій стабілізації. Оскільки ітерація стабілізації оцінює

якість роботи, виконаної на попередніх ітераціях, наприкінці проекту релізу, ця фаза стабілізації не вважається ітераційним забезпеченням якості.

Крім того, це означає, що деякі проблеми з якістю не виявляються до останньої ітерації процесу.

Таблиця 2.1 показує, що складно визначити будь-які процеси забезпечення якості в гнучких підходах на горизонті часу релізу. Метод розробки динамічних систем (DSDM) стверджує, що поза ітераціями можуть бути випадки, коли окремі/розділені процеси приймального тестування необхідні, наприклад, на точному горизонті часу релізу, у величезних проектах або через контрактні обмеження [24].

2.3 Ітерація процесу забезпечення якості

Забезпечення якості стосується завдань та дій, які не виконуються для кожної реалізованої функції окремо в певному ритмі на горизонті часу ітерації. Натомість, за допомогою цілеспрямованого підходу, ці дії контролюються та відстежуються протягом ітераційного горизонту часу. Це включає всі практики впровадження, тестування та перевірки, необхідні для забезпечення достатньої якості кінцевого продукту ітерації. Завдання ітерації обмежені в часі, оскільки кожна ітерація має заздалегідь визначену тривалість. Протягом ітерації тестувальники повинні розставляти пріоритети своїх завдань. Як результат, вкрай важливо стежити за виконанням завдання та постійно ділитися достовірною інформацією, наприклад, через зустрічі.

Без актуальних знань важко визначити обсяг завдань забезпечення якості ітерації, які необхідно виконати, щоб досягти бажаної якості продукту до кінця ітерації. У гнучких методологіях на горизонті часу ітерації набагато менше процедур забезпечення якості, ніж на горизонті часу контролю. Таблиця 2.1 показує, що існує лише кілька добре налагоджених методів забезпечення та оцінки якості програмного інкременту, створеного під час кожної ітерації. Процедури на горизонті часу ітерації менш визначені, ніж ті, що стосуються

часового горизонту heartbeat. Деякі методології, такі як XP, майже повністю покладаються на обґрунтовані практики heartbeat та вважають моніторинг прогресу важливим лише протягом усього часового горизонту ітерації.

Інші методології, які мають менш суворі практики heartbeat, визнають необхідність використання системного тестування для оцінки якості, досягнутої протягом ітерації, але вони не дають жодних конкретних інструкцій щодо того, як це зробити. Наприклад, єдина рекомендація FDD щодо досягнення цієї мети полягає в тому, щоб вирішити, які збірки та як часто подавати на незалежне системне тестування [25].

2.4 Забезпечення якості на рівні релізу

Забезпечення якості релізу використовується для забезпечення підтримки якості продукту протягом усього періоду релізу. Це включає перегляд результатів тестування та будь-яких інших високоякісних даних, зібраних з різних ітерацій, а також способи спрямування проекту розробки на основі отриманих знань (наприклад, планування майбутніх ітерацій). На період релізу відповідальність покладається на окрему групу тестування, тестування в різних умовах та тестування, яке неможливо завершити в межах графіка ітерацій.

Типовою практикою, яка використовується для включення забезпечення якості релізу, є окрема стабілізаційна ітерація після завершення проекту релізу. Оскільки стабілізаційна ітерація аналізує якість роботи, виконаної в попередніх ітераціях, це не є забезпеченням якості для ітерації. Через це деякі ризики проблеми в реалізації вимог якості не виявляються до самої останньої ітерації процесу.

Досить важко визначити будь-які процеси забезпечення якості в гнучких методологіях на період релізу. Наприклад, нам не вдалося знайти жодної процедури забезпечення якості серед зразків методів, наведених у таблиці 2.1, які б підходили для періоду релізу. У деяких випадках, згідно з DSDM, можуть бути необхідні незалежні процедури приймального тестування поза ітераціями,

тобто поза часовим горизонтом релізу. Ці обставини можуть виникати внаслідок контрактних обмежень або у випадку надзвичайно великих проектів.

2.5 Покращення гнучкого тестування для забезпечення якості

Гнучкі методи залежать від залучення користувачів, але вони не передбачають значного досвіду в деструктивному тестуванні. У деяких підходах для тестування використовуються лише тести прийняття користувачами, такі як XP, який пропонує набір ефективних заходів розробки з метою створення якісного програмного забезпечення. Деякі підходи, з іншого боку, не надають такого переліку завдань і, як наслідок, визнають необхідність специфічних методів тестування не лише на рівні інтеграції, але й на системному рівні та рівні прийняття. Це призводить до висновку, що процедури забезпечення якості heartbeat мають потенціал для вдосконалення, наприклад, шляхом додавання ролі незалежного тестувальника, який тестує кожну завершену функцію разом із розробником.

Гнучке тестування добре представлене прикладом сесійного дослідницького тестування (Exploration Testing – ET) на ітераційному часовому горизонті. Дослідницьке тестування (ET) – це неформальний підхід до розробки тестів, де тестувальник активно формує дизайн тестів під час їх виконання [26].

Тестувальник використовує знання, отримані в процесі тестування, для постійного вдосконалення та оновлення тестових випадків. Це новий підхід, побудований на досвіді, де розробка, навчання та виконання тестів здійснюються одночасно, а результати цих процесів негайно застосовуються до процесу створення більшої кількості тестів. Як результат, це не залежить від попередньо розроблених тестових випадків. Тестувальники використовують свої знання двома різними способами [27]: для розробки тестів та для виявлення збоїв. Це вказує на те, наскільки добре працюють знання тестувальників під час сесій дослідницького тестування. Використання сесійного дослідницького тестування дозволяє ефективно керувати тестуванням у стислі терміни, що робить його

придатним для коротких ітерацій. Крім того, як зазначалося раніше, суть дослідницького тестування сприяє культивуванні необхідного негативного мислення, необхідного для ефективного тестування. У деяких сценаріях існує потреба в тестуванні протягом періоду релізу. Однак часто вигідніше включати кілька методів забезпечення якості (QA) протягом усього періоду ітерації та фази heartbeat. Такий підхід забезпечує доступність якісної інформації на більш ранній стадії та зменшує ризики, пов'язані з низькою якістю на перших етапах розробки.

Гнучкі методології приділяють значну увагу участі клієнтів або користувачів і, як правило, виключають низку негативних процедур тестування. За винятком тестів прийняття користувачами, які належать до компетенції клієнта, деякі підходи, такі як XP, пропонують ретельний набір практик розробки, які пріоритезують виробництво програмного забезпечення задовільної якості без опори на значне тестування. Однак слід підкреслити, що іншим гнучким підходам бракує ретельного набору стандартів і вони не враховують вимогу щодо спеціалізованих практик тестування на рівнях інтеграції, системи та приймального тестування.

Метод динамічної розробки систем (DSDM), добре відомий приклад, вимагає впровадження тестування на кількох рівнях протягом кожної ітерації. Виходячи з існуючої літератури, очевидно, що бракує комплексних порад щодо ефективної інтеграції процедур деструктивного та незалежного тестування в гнучкі процеси розробки. Наше дослідження показало, що використання часових горизонтів для опису процесу розробки та використовуваних процедур сприяє визначенню областей, де процеси тестування можуть бути покращені.

Розуміння часового охоплення heartbeat та пов'язаних з ним методологій забезпечує необхідну узгодженість дій розробників та тестувальників. Існує потенціал для покращення методів забезпечення якості (QA), пов'язаних з "heartbeat". Одним із таких удосконалень є залучення незалежного тестувальника, який співпрацює з розробником для тестування кожної завершеної функції. Спираючись на незалежні деструктивні тести, а не виключно

на конструктивні методи розробника, ця функція забезпечує миттєвий зворотний зв'язок щодо досягнутої якості.

Дослідницьке тестування – це метод тестування, який втілює принципи гнучкого тестування, оскільки він не спирається на певні тестові випадки. Найкраще використовувати сесійне дослідницьке тестування в поєднанні з короткими ітераціями, оскільки це дозволяє керувати тестуванням у стислі терміни. За допомогою дослідницького методу можна оцінити всю систему або, наприклад, взаємодію між кількома окремими частинами. Наше дослідження демонструє цінність дослідницького тестування для тестування додатків з точки зору кінцевого користувача та швидкого виявлення критичних недоліків.

Дослідницький характер тестування полегшує розвиток руйнівного менталітету, необхідного для ефективного тестування. Дослідницьке тестування також може бути використане для переконання співробітників підприємств або інших осіб з глибокими знаннями предметної області взяти участь у тестуванні. За деяких обставин завдання тестування можуть бути необхідними на горизонті часу релізу. Гнучкі методології для часових рамок релізу ще не знайдено. У багатьох випадках може бути корисним включити якомога більше процедур контролю якості на часових рамках ітерації та heartbeat, щоб запропонувати якісну інформацію на ранній стадії та допомогти уникнути ризиків, пов'язаних з якістю.

3 ГНУЧКІ МЕТОДИ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ З БЕЗПЕРЕРВНОЮ ІНТЕГРАЦІЄЮ

3.1 Безперервна інтеграція програмного забезпечення

Інтеграція програмного забезпечення – це процес об'єднання різних підсистем або компонентів програмного забезпечення для формування цілісної системи [28]. Включення життєвого циклу розробки програмного забезпечення є важливим компонентом у сфері програмної інженерії, оскільки процес розробки часто охоплює багато етапів і передбачає співпрацю між групою інженерів. Інтеграція програмного забезпечення часто виконується як окремий етап у рамках традиційного життєвого циклу розробки програмного забезпечення, що відбувається після завершення впровадження програмного забезпечення. Один з дванадцяти керівних принципів екстремального програмування (XP), безперервна інтеграція (CI), – це методологія розробки програмного забезпечення, яка спочатку була запроваджена у 1997 році [29], що містить такі характерні особливості:

- 40-годинний робочий тиждень.
- Стандарти кодування.
- Колективна робота з програмним кодом.
- Безперервна інтеграція.
- Залучення клієнта (замовника).
- Парне програмування.
- Рефакторинг.
- Простий дизайн.
- Невеликий реліз.
- Тестування.
- Планування в ігровій формі.

Це просто означає, що кожен розробник регулярно об'єднує свою роботу принаймні раз на день. Такий підхід гарантує, що другорядні компоненти

включаються, як тільки вони завершені та готові стати частиною системи, запобігаючи виникненню складності.

Важливо автоматично створювати та додавати тестові випадки для всієї системи, включаючи нещодавно введені компоненти, щоб виконати стратегію безперервної інтеграції. Крім того, програма повинна бути зібрана та протестована автоматично, і розробник повинен отримувати швидкий зворотний зв'язок щодо нового коду, перш ніж додавати його [30]. Будь-які коментарі на цьому етапі будуть розглянуті розробником, як тільки вони будуть отримані. Коли програміст ще знайомиться зі своїми програмами, це допомагає у процесі виявлення недоліків та проблем.

Щоб отримати вигоду від впровадження безперервної інтеграції, розробники повинні змінити свою типову щоденну практику розробки програмного забезпечення. Людям часто потрібно утримуватися від внесення коду в репозиторій, негайно виправляти проблемні збірки, створювати автоматизовані тести, переконатися, що всі тести та перевірки проходять успішно, виконувати приватні збірки та уникати отримання дефектного коду.

Зміни коду, внесені різними розробниками, часто об'єднуються один раз на день під час використання безперервної інтеграції. Оскільки програма створюється та принаймні модульно тестується при кожній інтеграції, можна гарантувати принципово адекватний рівень якості. Крім того, оскільки розробники постійно отримують найновіші модифікації від інших розробників, більш імовірно, що вони зіткнуться з труднощами з цими змінами, які потім можна буде усунути швидше порівняно з тим, коли зміни будуть доступні іншим розробникам лише через кілька днів або тижнів [8].

Безперервна інтеграція – це один із методів, який можна використовувати для покращення ручного тестування програмного забезпечення, зокрема тривалості циклу тестування. Коли безперервна інтеграція відсутня, тестувальники повинні чекати, поки програма буде створена, перш ніж вони зможуть її протестувати, потім вони повинні розгорнути продукт у своєму тестовому середовищі, запустити програмне забезпечення та, нарешті, дістатися

до розділу програмного забезпечення, який потрібно протестувати. Усі ці фази можуть призвести до невдачі, змушуючи тестувальників чекати, поки розробники вирішать проблеми, що, у свою чергу, збільшує час, необхідний для тестування та схвалення модифікацій програмного забезпечення. Тестерам просто потрібно розгорнути найновішу збірку програмного забезпечення, і вони можуть мати більший рівень впевненості в тому, що програма працюватиме до того моменту, коли вона досягне розділу, який потрібно протестувати, оскільки безперервна інтеграція гарантує, що програмне забезпечення завжди перебуває в загальному функціональному стані.

Якщо програма також автоматично доставляється до тестового середовища як частина збірки безперервної інтеграції, то фазу ручного розгортання тестувальники також можуть пропустити. Збірки безперервної інтеграції – це вид збірки, який виконується безперервно. Тестери можуть зосередити свої зусилля на тих видах тестування, які найефективніше виконуються вручну людьми, таких як нефункціональне тестування та дослідницькі тести. Якщо відомо, що програма функціонує належним чином в цілому в результаті безперервної інтеграційної збірки, яка включає широкий набір тестів, тоді програмне забезпечення може бути протестовано. Окрім тестування нефункціональних вимог, таких як ємність та безпека, тестери також можуть підтвердити зручність використання програми. Комплексне автоматизоване тестування в поєднанні з ручним тестуванням програми в цілому має потенціал для покращення якості програмного забезпечення, ніж просте виконання ручних тестів.

3.2 Методи безперервної інтеграції, що використовуються розробниками

У цьому розділі будуть розглянуті основні кроки безперервної інтеграції разом з окремими компонентами кожного кроку. Вважається, що компанія

впроваджує безперервну інтеграцію в процес розробки програмного забезпечення в результаті практичного впровадження цих концепцій.

Часте додавання коду: центральний момент процесу безперервної інтеграції. Коли справа доходить до додавання своєї роботи до спільного репозиторію коду, розробникам не слід чекати більше одного дня, перш ніж це зробити. Нижче наведено деякі варіанти, доступні розробникам для спрощення цього процесу: замість того, щоб вносити масштабні зміни до кількох компонентів одночасно, внесіть лише невеликі корективи; Візьміть на себе зобов'язання після завершення кожної окремої дії, припускаючи, що завдання можна виконати за кілька годин кожне.

Не слід комітити пошкоджений код – програмування, яке спричиняє помилки будь-якого роду під час включення до збірки безперервної інтеграції, називається «пошкодженим кодом». Розробники повинні спочатку зібрати та протестувати свій код локально на власних ПК, перш ніж вносити зміни до спільного репозиторію коду. Вони повинні зачекати, поки всі тести та перевірки пройдуться, перш ніж додавати будь-який код.

Негайно слід виправляти помилкові збірки: проблема з розгортанням, базою даних або компіляцією може призвести до помилкової збірки. Головним пріоритетом проекту має бути виправлення помилкової збірки, і відповідний розробник повинен негайно реагувати на такі проблеми.

Варто писати автоматизовані тести для розробників. Тести мають бути автоматизовані, щоб ефективно працювати в середовищі неперевершеної інтеграції. Вони також повинні охоплювати весь вихідний код.

Усі автоматизовані тести та перевірки повинні бути успішно виконані, щоб збірка була успішною: найважливіша вимога CI для якості програмного забезпечення полягає в наступному. Інтеграційна збірка включає інструменти покриття, які використовуються для виявлення вихідного коду, якому бракує відповідного тестового випадку. Для запуску автоматизованих перевірок та перевірки загальних стандартів дизайну та кодування також використовуються додаткові інструменти.

Запуск приватних збірок. Завдяки технологіям неперервної інтеграції (CI) програмісти можуть зберігати локальну копію програмного забезпечення своїх робочих станцій зі спільного репозиторію коду. Щоб переконатися, що він не виходить з ладу, вони можуть спочатку використовувати нещодавню збірку інтеграції локально, перш ніж об'єднувати її з основним сервером.

Слід уникати отримання пошкодженого коду, використовуючи технології неперервної інтеграції (CI), які допомагають повідомляти про помилки. Однією з найважливіших особливостей CI є її здатність надавати розробникам швидкий зворотний зв'язок. Стан коду постійно оновлюється, і якщо він пошкоджений, жоден розробник не повинен перевіряти його зі спільного репозиторію коду. Відповідальна особа повинна розпочати виправлення, щойно зворотний зв'язок покаже, що код має недоліки. Якщо ні, то негайний внесок, який пропонує CI, втрачається.

Ступінь впровадження безперервної інтеграції значно варіюється між програмними проектами. Збірка безперервної інтеграції може включати лише кілька процесів, таких як автоматизована збірка програмного забезпечення та автоматизовані модульні тести, але вона також може включати багато додаткових фаз, таких як автоматизоване приймальне тестування, автоматизоване розгортання або обчислення метрик програмного забезпечення. Інші змінні, такі як час, необхідний для завершення збірки безперервної інтеграції, або частота інтеграції змін програмного забезпечення, також можуть впливати на реалізацію безперервної інтеграції та її вплив на якість програмного забезпечення та час циклу тестування.

Наступні заходи спрямовані на забезпечення відмінної внутрішньої та зовнішньої якості програмного забезпечення, а також швидкого часу циклу тестування. Частота інтеграції оновлень програмного забезпечення є першим критичним елементом. Коли модифікації впроваджуються частіше, недоліки в цих змінах виявляються та усуваються швидше. Крім того, при більш регулярній інтеграції розмір змін менший, що знижує ризик кожної інтеграції та сприяє ефективному виявленню проблем.

Розробники більш схильні інтегрувати свої модифікації рідше, якщо тривалість збірки безперервної інтеграції велика. Як результат, також важливо, щоб збірки безперервної інтеграції завершувалися швидко. Для забезпечення високої якості програмного забезпечення та швидкого циклу тестування також вирішальним є виконання автоматизованих тестів як компонента безперервної інтеграційної збірки та їх обсяг. Тестувальники програмного забезпечення можуть зосередитися на спеціалізованих завданнях тестування, таких як дослідницьке тестування, замість повторюваних тестових дій, таких як регресійне тестування, використовуючи автоматизовані тести [15].

Безперервне обчислення метрик програмного забезпечення, часто відоме як автоматизована перевірка, допомагає покращити внутрішню якість програмного забезпечення. Такі метрики, як покриття коду або кількість дублікатів коду, можуть виявити проблеми з якістю вихідного коду та зробити цю якість очевидною. Також простіше вжити коригувальних заходів, якщо джерела проблем коду вже були виявлені за допомогою автоматизованої перевірки. Автоматизоване розгортання програмного забезпечення в тестових середовищах може допомогти забезпечити відмінну якість програмного забезпечення, пропонуючи перевірений метод розгортання програм. Ймовірність того, що розгортання програмного забезпечення буде виконуватися по-різному кожного разу або для різних середовищ, якщо це робиться вручну, досить висока. Ризик розгортання у виробничому середовищі зменшується, пропонуючи автоматизований механізм розгортання та тестуючи його, запускаючи його як частину безперервної інтеграційної збірки.

3.3 Елементи якості в процесі розробки

На які характеристики якості впливає безперервна інтеграція програмного забезпечення, було визначено за допомогою стандартного визначення характеристик ISO [31], який класифікував критерії якості відповідно до того, як вони пов'язані з різними етапами життєвого циклу розробки програмного

забезпечення. Оскільки безперервна інтеграція є добре відомою методологією розробки, це дослідження значною мірою зосереджено на характеристиках якості, які підпадають під її охоплення. Повний перелік ознак та супутніх їм вимірювань наведено в таблиці 3.1.

Таблиця 3.1 – Елементи якості в процесі розробки та їх вимірювання

Атрибут	Вимірювання
Час розробки	Час завершити реалізацію функції та підготувати її до тестування.
Введені помилки	Загальна кількість помилок, знайдених у щойно написаному коді або у пошкоджених старих частинах коду.
Час доставки	Час завершити процес впровадження та тестування функції, а потім зробити її доступною для використання клієнтами.
Якість тесту	Тести повинні охоплювати всі функції та виявляти більшість (якщо не всі) проблем.
Документація	Необхідно, щоб документ містив усю роботу, виконану розробниками.
Управління змінами	Зміни до специфікацій мають бути прийнятними та не вимагати надмірної кількості часу та праці.
Модель витрат	Він розраховується шляхом порівняння витрат на підготовку нового налаштування з сумою грошей, яка буде заощаджена в результаті впровадження нових коригувань.

У наступній таблиці представлено вичерпний огляд традиційних підходів до інтеграції та подальші численні аспекти якості, пов'язані з методологіями інтеграції програмного забезпечення.

3.4 Проблеми та рішення практик безперервної інтеграції з використанням методів гнучкої розробки програмного забезпечення

У методології Agile тестувальники беруть на себе різні обов'язки, які відрізняються від тих, що були в старіших методах. У цьому контексті тестувальник активно взаємодіє з усіма зацікавленими сторонами.

Таблиця 3.2 – Якість до та після використання безперервної інтеграції

Критерії порівняння	До безперервної інтеграції	Після безперервної інтеграції
Час розробки	Вимоги до всього релізу опрацьовуються розробниками.	Робота розробників зосереджена на певній вимозі.
Введені помилки	Коли тестування проводиться лише після завершення значної частини коду, кількість проблем набагато вища, ніж коли тестування проводиться після реалізації лише невеликих частин коду.	Необов'язково чекати, поки буде готовий весь реліз, щоб кожну функцію було протестовано та виправлено після завершення розробки.
Час доставки	Після завершення релізу та його ретельного тестування	Після виконання однієї попередньої умови та перевірки її дотримання,
Якість тесту	• Усі релізи в цілому проходять тестування. • Середовище, яке використовується для тестування, відрізняється від середовища, що використовується у продакшені.	Після того, як функція інтегрована, тестування проводиться на її окремих компонентах, а результати тестування надсилаються розробникам, щойно вони стають доступними. Тестове середовище та робоче середовище майже не відрізняються одне від одного.

Продовження таблиці 3.2

Документація	• Процес тестування охоплює кожен пункт вимог • Кожен з релізів у повному обсязі. • Середовище, яке використовується під час тестування, не ідентичне тому, яке використовується під час виробництва.	• Є документація щодо вимог. • Кількість створених документів невелика. • Автоматизовані інструменти створюють статистику та дані про розроблені функції на основі вимог, результатів тестування та згенерованих дефектів тощо.
Управління змінами	• Зміни прийнятні лише після проходження тривалої процедури та отримання дозволів • Модифікації впроваджуються за допомогою абсолютно нового патча або релізу.	Будь-які зміни можуть бути впроваджені в будь-який момент, але власник повинен спочатку повідомити про щойно окреслені потреби командам бізнес-аналізу та розробки програмного забезпечення.
Модель витрат	Кількість часу та зусиль, які знадобляться для ручного виконання розробки та інтеграції програмного забезпечення	Витрати, пов'язані з придбанням сервера та інструментів для безперервної інтеграції

У розподіленому контексті управління тестовими випадками стає проблемою, коли кількість спринтів зростає, що призводить до пропорційного зростання розміру набору тестів. Вивчення існуючих наукових робіт [32] показує, що вищезгадані проблеми потребують уваги щодо практики тестування в умовах Agile.

3.5 Огляд практик тестування для гнучких технологій розробки

У цьому розділі проаналізуємо проблеми, що виникають при виконанні тестування в рамках гнучких моделей життєвого циклу розробки програмних продуктів та запропонуємо способи вирішення цих проблем.

Проблема: стосується здатності Agile-тестувальника займатися іншими завданнями одночасно зі своїми обов'язками з тестування. У доступній літературі бракує вичерпного опису послідовності дій, включаючи регресійне тестування, в рамках життєвого циклу Agile-тестування [33, 34].

Рішення: Запропонований життєвий цикл гнучкого тестування передбачає активну взаємодію між тестувальником та іншими зацікавленими сторонами. Позиція тестувальника визначається його зосередженістю на проведенні тестових операцій, спрямованих на забезпечення високоякісного продукту для клієнта. Крім того, було визнано значний вплив регресійного тестування.

Проблема: метод парного програмування використовується у впровадженні гнучкого тестування для отримання високоякісного результату. Йдеться про методологію проведення тестування в ситуаціях, коли члени команди або клієнти фізично не знаходяться в одному місці. Ще однією проблемою є процес формування пар для цілей парного програмування або тестування [35].

Рішення: пропонується концептуальна структура для розподіленого середовища [36], яка включає відповіді на проблеми, з якими стикаються клієнти та члени розподіленої команди. Зокрема, для управління розподіленими перешкодами, дотримуючись основних принципів проектування, було рекомендовано рефакторинг [37]. Крім того, проблему пов'язаних труднощів у розподілених середовищах можна вирішити за допомогою шаблону, який має розроблене рішення на основі найкращих поточних рішень. Крім того, було запропоновано самоцентричну стратегію (спільні риси членів команди) для

парного програмування та парного тестування, щоб допомогти членам команди створити пару.

Проблема: Agile часто передбачає адаптацію до змін, тому це може чинити значний вплив на залежні модулі, і в результаті кількість тестових випадків зростає. Тому регресійне тестування є більш необхідним. Тому для керування більшою кількістю тестових випадків потрібен окремий підхід до регресійного тестування.

Рішення: представлено методологію вибору регресійних тестів [38], модель, що базується на графі користувацького наративу та включає зв'язки між історіями користувача. Крім того, для пошуку ідеального виходу з усіх шляхів, включених до графу користувацького наративу, включаючи шляхи, яких ще не існує, та шляхи, які вже існують, було використано дві метрики, відомі як середнє значення шляху та середня довжина шляху.

Проблема: завдання полягає в управлінні тестовими випадками для спринту в розподіленому середовищі, коли часто виникає потреба реагувати на зміни, і коли поточні користувацькі історії зазнають впливу в результаті цих змін.

Рішення: Для визначення пріоритетів тестових випадків було запропоновано лінгвістичну стратегію [39], яка базується на кількості пауз, виявлених у користувацьких історіях, для визначення пріоритету наративу, та кількості іменників та дієслів, визначених у користувацьких історіях, для визначення пріоритетів тестових випадків. Крім того, як метод визначення пріоритетів тестових випадків було запропоновано підхід, що базується на ризиках та ідентифікації історій з високим рівнем ризику.

3.6 Практики безперервного тестування для гнучкого забезпечення якості

Пропонується розуміти під терміном безперервного тестування (БТ) спосіб скорочення часу, що витрачається на виконання тестів. Крім того, у книзі

беззаперечних авторитетів у програмній інженерії [40] вони назвали автоматичне виконання всіх тестів проекту щоразу, коли проект збирається (функція, відома як «автоматичне тестування») однією з переваг ідеї. Часте виконання тестів є однією з цілей TDD. Однак розробнику часто доводиться залишати свою роботу, щоб фізично виконувати тести. Ви можете продовжувати розробку кодову, використовуючи сучасні IDE, такі як Eclipse або Visual Studio. Безперервна компіляція (також відома як автоматизована компіляція або автоматизована збірка) є загальним терміном для цього. Дозволяючи IDE виконувати збірку у фоновому режимі, поки розробник пише та/або зберігає файл, цей метод усуває витрати на ручну компіляцію коду після написання вихідного коду. Виконуючи тести під час роботи розробника, БТ просуває цей підхід. Запуск тестів не вимагає зупинки того, що ви робите. Розробник отримує оперативний зворотний зв'язок, оскільки тести виконуються автоматично у фоновому режимі.

Практика безперервного тестування (БТ) спрощується завдяки широкому розмаїттю плагінів, які легко доступні на ринку та адаптуються до різних інтегрованих середовищ розробки (IDE) та інших інструментів. Більшість інструментів надаються як розширення для сучасних інтегрованих середовищ розробки (IDE). Найперші інструменти були створені спеціально для мови програмування Java та інтегрованого середовища розробки (IDE) Eclipse. Для платформ Visual Studio та.NET були розроблені аналогічні утиліти. Крім того, доступні інструменти для мов програмування, таких як Ruby. Кілька організацій створили різні інструменти безперервного тестування (БТ). Ці інструменти включають Contester, плагін Eclipse, розроблений студентами Товариства програмної інженерії Вроцлавського технологічного університету, JUnit Max, плагін Eclipse для БТ, NCrunch, комерційний плагін Visual Studio, Autotest, інструмент безперервного тестування Ruby, Continuous Testing for VS, комерційний плагін Visual Studio, AutoTest.NET та Mighty Moose, пакетну версію AutoTest. Розробники інтегрованих середовищ розробки (IDE) сьогодні все більше усвідомлюють важливість БТ. Microsoft Visual Studio включає функцію безперервного тестування. Як результат, галузь усвідомлює важливість практики

БТ. Наша мета — спиратися на це розуміння та використовувати будь-яку потенційну синергію, яка може виникнути в результаті поєднання ідей TDD та БТ у контексті гнучкої розробки програмного забезпечення. Крім того, ми хочемо отримати емпіричні докази, щоб підтвердити корисність застосування цієї запропонованої практики в промислових умовах.

У сфері розробки програмного забезпечення тестування є неймовірно важливим. З процесом тестування програмного забезпечення пов'язані значні витрати. Для створення та запуску тестів процес тестування вимагає значних витрат часу та зусиль. Вартість розробки зазвичай зростає вдвічі через виконання процесів тестування. Для вирішення цих проблем можна використовувати безперервне тестування. Концепцію безперервного тестування вперше розробили [41]. Був проведений експеримент, щоб продемонструвати, що використання безперервного тестування може призвести до скорочення загального часу розробки приблизно на 15 відсотків. Проведений експеримент демонструє, що використання безперервного тестування може ефективно зменшити втрати в процесі розробки, зокрема з точки зору мінімізації часу очікування. Безперервне тестування передбачає використання методів автоматизації та стратегічне визначення пріоритетів процесу тестування. Безперервне тестування є більш ефективним та результативним підходом до ідентифікації та виявлення помилок, що вимагає меншої кількості часу та зусиль. Воно безперервно запускає тести, щоб переконатися, що код має належний рівень якості. Ці тести запускаються автоматично, коли програмісти створюють новий код. Безперервне тестування може забезпечувати безперервний зворотний зв'язок, запускаючи тести у фоновому режимі без залучення інженерів. Екстремальне програмування та безперервна компіляція можуть розглядатися як розширення безперервного тестування. Безперервна компіляція забезпечує швидкий зворотний зв'язок щодо помилок компіляції. Безперервне тестування – це практика частого проведення тестів на ранніх стадіях процесу розробки. Безперервна доставка залежить від безперервного тестування, щоб покращити якість продукту та забезпечити його безпомилковість.

3.7 Проблеми та рішення для безперервного тестування в Agile Quality Assurance

3.7.1 Виклики безперервного тестування в Agile Quality Assurance

Тестування на більш загальному рівні проводиться в компаніях-розробниках програмного забезпечення для оцінки рівня якості програми. З іншого боку, швидка перевірка якості не повинна передбачатися для безперервної інтеграції. Тестування проводиться вручну, і немає автоматизації жодного з тестових випадків, тому швидкість виявлення помилок поступово зростатиме. Процес триватиме довше, ніж очікувалося, що призведе до затримки у доставці елементів. Безперервне тестування – це одна зі стратегій, яка може бути використана для вирішення цих труднощів. Безперервне тестування дозволяє надавати своєчасний зворотний зв'язок без участі людини. Автоматизоване тестування є компонентом безперервного тестування. Цей моніторинг та покращення якості тестових випадків здійснюється за допомогою автоматизації. Стратегії безперервного тестування використовуються в тестовій діяльності, яка є частиною стратегії DevOps. Це допомагає виявити недоліки та помилки в різних компонентах програмного забезпечення. До тестування підходять інакше, ніж до традиційного тестування, оскільки DevOps можна розглядати як набір принципів [42]. До DevOps тестування було одним із етапів життєвого циклу розробки програмного забезпечення, але воно не виконується протягом усього проекту. Але з DevOps тестування має бути присутнім з самого початку та гарантувати якість продукту завдяки спільній відповідальності між командами розробки та супроводу.

Багато досліджень висвітлювали перешкоди, з якими стикається IT-компанія під час впровадження методів DevOps.

Принципи DevOps можуть створювати перешкоди для компаній.

Невідповідна архітектура, ручне тестування та небажання змін є перешкодами для впровадження безперервної доставки. Через обмеження

інструментів безперервний розподіл було призупинено. Сучасні технології створюють ризики для безпеки під час розгортання та забезпечують неадекватний зворотний зв'язок під час тестування.

Впровадження таких підходів DevOps, як безперервна доставка, вимагає змін у культурі компанії, посаді та обов'язках, пов'язаних з релізом. Багато компаній вважають складним створити процедури, які працюють, тому слід проводити співбесіди для визначення переваг та проблем. Автори [43] виявили, що архітектура системи, тестування та інструменти інтеграції є ключовими перешкодами для впровадження безперервної доставки за принципами DevOps. Вони стверджують, що стратегії компаній повинні змінитися, коли вони переходять на DevOps та CD. У роботі [44] було визначено ризик безпеки конвеєра безперервної доставки. Через злом та крадіжку даних з конвеєра CD існує ризик захисту даних з Інтернету. Автори запропонували використовувати контейнерні технології, такі як Docker, для вирішення цієї проблеми.

У роботі [42] пояснюється цінність відкритих каналів комунікації та тісних робочих відносин між командами розробки та супроводу в контексті забезпечення якості програмного забезпечення. DevOps надає найкращі рішення для тестування та доставки. Коли справа доходить до тестування, принципи та функції DevOps використовують низку методів, які відрізняються від тих, що використовуються у звичайному тестуванні. Гарантується відсутність затримок доставки завдяки використанню метрик та визначенню пріоритетів тестових випадків. Проблему впровадження DevOps можна вирішити за допомогою команд тестування в бізнесі. Для полегшення автоматизованого розгортання, релізів та моніторингу інфраструктури безперервного тестування можна розділити на групи тестування. Це полегшує швидке отримання відгуків про якість програмного забезпечення.

Тестування можна розглядати по-різному завдяки DevOps, яка може допомогти командам розробки та супроводу тісно пов'язати себе. Існує низка підходів до організації методів тестування DevOps, таких як перемикання функцій, тестування інфраструктури, деструктивне тестування тощо. Команди з

розвитку бізнесу та супроводу ПЗ можуть вирішити використовувати ці стратегії залежно від своїх потреб. Тестування у продакшені в рамках DevOps надає команді розробників постійний зворотний зв'язок. Три основні методи– це альфа/бета-тестування, моніторинг як тестування та бета-тестування для тестування у продакшені. Переваги DevOps включають моніторинг виробничого середовища в режимі реального часу та управління аномаліями, як тільки вони з'являються. Інструменти тестування, що використовуються в методології DevOps, допомагають у розробці ефективних методологій генерації тестів, як показано в [46].

Фахівці з DevOps мають проблеми з пошуком точних інструментів для виконання завдань безперервної інтеграції та доставки. Вони також втрачають час та енергію, намагаючись вибрати відповідний інструмент тестування. Багато компаній використовують процеси DevOps для таких послуг тестування, як модульне тестування, тестування, орієнтоване на розробку, та поведінкове тестування. Конвеєр тестування одразу отримує код для подальшого покращення якості. Тестування має бути частиною DevOps з самого початку процесу розробки. Тестування до завершення проекту впливає на якість цього проекту.

Досягнення більшої продуктивності під час безперервного тестування DevOps відбувається завдяки вибору відповідних інструментів та технологій. Інструменти економлять час та зусилля, автоматично виявляючи проблеми під час тестування. Покращення продуктивності тестувальників вимагає вивчення нових інструментів та технологій, співпраця між командами розробки та тестування підвищує ефективність. Крім того, впровадження DevOps усуває прогалини в знаннях та покращує якість коду.

Команди тестування можуть подолати труднощі в процесі трансформації DevOps. Команда розробників може забезпечити швидкий зворотний зв'язок, якщо вони встановлять структуру для безперервного тестування. Інфраструктура для безперервного тестування та співпраця між розробкою та експлуатацією забезпечують швидкий реліз та розширене охоплення тестуванням. Однак щодо заходів, які допомагають командам покращити свою роботу з тестування,

доступної інформації небагато. Усі процеси, від розробки до релізу, залежать від тестування DevOps або охоплюються ним. DevOps сприяє тестуванню безпеки та продуктивності. Моніторинг може бути корисним у тестуванні для отримання негайного зворотного зв'язку щодо технічних процесів. Тестування DevOps покращує якість та покращує комунікацію між усіма зацікавленими сторонами. В академічних або наукових статтях, включаючи тематичні дослідження, немає багато систематичних досліджень тестування DevOps.

Міцна та надійна архітектура забезпечує тестованість та розгортання. Вона сприяє отриманню швидкого зворотного зв'язку від операційних команд та команди розробників. Незважаючи на це, існує мало досліджень щодо того, як DevOp впливає на архітектуру програмного забезпечення. Методології DevOps спрямовані на швидке впровадження змін у продакшн для досягнення високої якості [47]. Ці методи позбавляються структурних перешкод для трансформації. На думку багатьох фахівців з програмного забезпечення, монолітні та інші архітектурні стилі/типи не застосовні до DevOps та CD [48]. DevOps не підходить для високозв'язаного дизайну.

Застаріла або успадкована архітектура може впливати на впровадження DevOps. Збір даних з різних технологій ускладнений через проблеми інтеграції в цих застарілих інфраструктурах. Дослідження [48] стверджує, що еталонний дизайн полегшує співробітникам спільну роботу та вирішення бізнес-питань у середовищі DevOps. Обсяг операційних зусиль зменшився на 50% завдяки взаємодії команд розробки та супроводу. Коли нові модулі додаються до виробничого конвеєра, DevOps зменшує кількість збоїв на 3%. За словами автора, команди спонукають розробляти нові програми через меншу кількість помилок, а не через зосередження на виправленні дефектів. Цикл тестування та розгортання значно уповільнюється в мікросервісних системах через безперервну переробку коду.

Завдяки своїй здатності сприяти швидкій доставці та надавати корисний зворотний зв'язок, значна кількість компаній впроваджує стратегію DevOps. Щодо трансформації DevOps було проведено численні дослідження. Хоча це

дослідження зосереджено на безперервному тестуванні та тому, як архітектура впливає на перехід до DevOps, воно не виходить за рамки цих двох областей. Що стосується безперервного тестування (БТ) та безперервної доставки (CD), існує брак досліджень, які б точно вивчали вплив архітектури на виробничі умови. Існує мало ґрунтовних досліджень, які дають глибоке розуміння безперервного тестування та доставки в контексті екосистеми DevOps.

3.7.2 Розв'язання проблем впровадження безперервного тестування в гнучкі технології розробки

Це дослідження має на меті визначити потенційні процедури або методи, які можна було б використовувати для вирішення проблем, пов'язаних з безперервним тестуванням та DevOps.

Надамо список можливих практик, що дозволять подолати проблеми безперервного тестування в гнучкі технології розробки.

1. Участь замовника у підтвердженні мети та вимог.
2. Збір відповідних вимог.
3. Використання конкретних/специфічних інструментів.
4. Використання лише некомерційних інструментів.
5. Раннє та поточне тестування.
6. Постійне планування спринтів.
7. Комунікація та співпраця.
8. Автоматизація.
9. Отримання протоколів при реалізації процесів.
10. Розуміння відповідальності та обов'язків.
11. Міжфункціональне середовище.
12. Правильне виконання тестових випадків та чіткі результати.
13. Немає страху невдачі чи змін.
14. Програма обміну знаннями між командами.
15. Семінари, тренінги та майстер-класи.

16. Використовуються Jenkins, Selenium та інструментів контролю версій.
17. Потрібні навички повноцінної розробки та прийняття рішень.
18. Допомога керівництва та членів групи.
19. Розвиток інфраструктури.
20. Виконання верифікаційних тестів.
21. Використання хмарних сервісів.

Тестування гарантує якість послуг та програмного забезпечення в DevOps. Клієнти все частіше вимагають швидкої відповіді та відмінних додатків. Крім того, автоматизація є найважливішим етапом для гарантування постійного зворотного зв'язку. Команди прагнуть максимально автоматизувати роботу згідно з принципами DevOps. Автоматизоване тестування забезпечує швидкий зворотний зв'язок, вказуючи на недоліки програмного забезпечення. Наприклад, тестування необхідне для кожного додаткового випуску продукту за новим гнучким способом роботи. Щоб мінімізувати трудовитрати та людські помилки, автоматизуйте якомога більше тестів, включаючи модульні, інтеграційні та приймальні тести. Безперервне тестування та моніторинг були необхідні для створення спринтів та їх випуску для клієнтів. Крім того, безперервне тестування перевіряє та підтверджує, що програма не містить помилок, а метою є задоволення вимог клієнта.

Проблема комунікації, очевидно, виникає внаслідок географічного розділення між групами тестування та розробки. Програма обміну навичками покращить знання про тестування та DevOps. Впровадження DevOps передбачає використання спеціальних інструментів тестування та покращених каналів зв'язку. Коли тестувальники та розробники працюють разом, ефективність модульних та інтерфейсних тестів може бути підвищена. Наприклад, Jenkins автоматизує тестування коду за допомогою сервера безперервної інтеграції, щоб заощадити значну кількість часу та роботи. Співпраця між командами тестування та розробки, а також автоматизація, є важливими для застосування концепцій DevOps.

3.7.3 Управління тестовими даними

Проблеми – Управління та підтримка відповідних тестових даних, які ефективно імітують реальні ситуації, можуть бути складними.

Рішення – Дані можна зробити більш релевантними та такими, що відображають реальні ситуації, використовуючи синтетичні тестові дані.

Проблеми – Створення та керування тестовими даними, які враховують різноманітні умови, може бути складним.

Рішення – для забезпечення реалістичних тестових даних, які відповідають нормам конфіденційності, можна використовувати такі методи, як маскування даних, анонімізація даних та розробка синтетичних даних. У цьому відношенні можуть бути корисними такі технології, як Delphix та інші інструменти TDM (керування тестовими даними).

3.7.4 Управління тестовим середовищем

Проблеми – особливо складним є завдання забезпечити надійність та узгодженість середовища тестування під час роботи із складними додатками. Рішення – використання технологій контейнеризації, таких як Docker, є одним із способів гарантувати надійне та узгоджене середовище тестування.

Проблеми – налаштування та підтримка тестових середовищ, які точно відображають робоче середовище, може бути складною та трудомісткою.

Рішення – Автоматизація налаштування та конфігурації середовища може бути досягнута за допомогою використання інфраструктури як коду (IaaC) та технологій контейнеризації (таких як Docker). Kubernetes та інші інструменти можуть допомогти в управлінні контейнерними середовищами.

3.7.5 Підтримка автоматизації тестування

Проблеми – Скрипти для автоматизації тестування можуть легко стати складними та важкими в обслуговуванні, особливо якщо застосунок постійно оновлюється.

Рішення – Підтримка актуальності та актуальності скриптів автоматизації тестування вимагає регулярного обслуговування та переробки.

Проблеми – зі збільшенням розміру програми може стати складніше підтримувати та масштабувати автоматизовані набори тестів.

Рішення – Використання модульного дизайну тестів та використання таких фреймворків, як Selenium, Appium або TestNG, може покращити зручність обслуговування та масштабованість скриптів автоматизації тестування. Крім того, доцільно розглянути хмарні рішення для тестування як засіб підвищення масштабованості.

3.7.6 Інтеграційне тестування

Проблеми – Складність тестування зв'язків між різними компонентами та системами може створювати труднощі в Agile-контекстах.

Рішення – Для перевірки взаємодії API та мікросервісів рекомендується тестування контрактів та тестування контрактів, керованих споживачем (CDC). Такі інструменти, як Pact та Spring Cloud Contract, можуть допомогти у досягненні бажаних результатів.

4 ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ

В умовах прискореного розвитку інформаційних технологій, професії у сфері ІТ займають ключове місце в економічній структурі та соціальному устрої. Відповідно, набуває актуальності забезпечення адекватних умов праці та охорони здоров'я для спеціалістів цієї галузі. Попри загальноприйняту думку про порівняно низький ризик в роботі програмістів порівняно з іншими професіями, ця сфера містить специфічні ризики та особливості, що вимагають індивідуального підходу до питань охорони праці.

Аналіз робочого середовища ІТ-спеціалістів показує, що, незважаючи на зовнішню зручність, існують недоліки з точки зору ергономіки, психологічного навантаження та інших важливих аспектів. У цьому контексті важливим є вивчення основних аспектів охорони праці програмістів, аналіз потенційних ризиків та розробка рекомендацій щодо оптимізації умов праці для фахівців у галузі інформаційних технологій.

4.1 Питання щодо охорони праці

Фактори трудового середовища можуть істотно впливати на здоров'я та працездатність розробника програм. Неправильно організоване робоче місце може викликати проблеми із хребтом, шиєю, зап'ястям та іншими частинами тіла. Довгий час роботи за комп'ютером може призвести до тунельного синдрому зап'ястного каналу.

4.1.1 Ергономіка

У сучасних умовах трудової діяльності значуще місце займає організація робочого місця користувачів персональних комп'ютерів. Законодавство України наголошує на необхідності забезпечення безпечних та комфортних умов праці. Зокрема, відповідно до КЗпП та Закону України «Про охорону праці»,

роботодавці мають обов'язки стосовно забезпечення працівників належними умовами.

При цьому велику увагу слід приділити ергономічності робочого простору. Згідно з нормативами (ДСТУ 8604:2015, ДСТУ 7299:2013, ДСТУ EN 60447:2022), елементи робочого місця мають бути правильно розташовані, враховуючи особливості трудової діяльності. Наприклад, відстані між комп'ютерами, розміри робочого столу, а також вільний простір для ніг користувача визначені з урахуванням оптимального комфорту та доступності.

При організації робочих місць із персональними комп'ютерами важливо забезпечити відстані між боковими частинами столів не менше ніж 1,2 м, а також враховувати, що відстань між задньою частиною одного комп'ютера та екраном іншого має бути 2,5 м. Структура робочого столу повинна бути розроблена відповідно до ергономічних стандартів, дозволяючи ефективно розташовувати необхідне обладнання, таке як дисплей, клавіатура, принтер, а також робочі документи.

Щодо параметрів робочого столу: його висота повинна знаходитися у діапазоні від 680 до 800 мм. Що стосується ширини та глибини, то вони повинні бути такими, щоб працівник міг з легкістю працювати в зоні доступності рук (для цього рекомендовані показники: ширини від 600 до 1400 мм, глибини від 800 до 1000 мм). Також необхідно передбачити комфортний простір для ніг користувача: висота – мінімум 600 мм, ширина – мінімум 500 мм, глибина на рівні колін – 450 мм і на рівні витягнутої ноги – не менше 650 мм.

Стілець на робочому місці повинен мати підйомно-поворотні характеристики, можливість регулювання по висоті, куту нахилу сидушки і спинки, а також відстані від спинки до зовнішнього краю сидіння. Поверхня сидіння має бути рівною, а її зовнішній край має мати округлу форму. Налаштування по кожному параметру повинно бути індивідуальним, інтуїтивним та надійно фіксуватися.

Інтервал регулювання частин стільця: для лінійних розмірів – 15-20 мм, для кутових 2-5°. Зусилля для регулювання не має перевищувати 20 Н. Висота

сидіння повинна бути від 400 до 500 мм, а її ширина і глибина – не менше 400 мм. Сидіння має мати можливість нахилу до 15° вперед і до 5° назад. Висота спинки стільця – 300±20 мм, ширина – не менше 380 мм, радіус кривизни горизонтально – 400 мм. Кут нахилу спинки може регулюватися в діапазоні 1-30° від вертикального положення. Відстань від спинки до сидіння – від 260 до 400 мм. Для зменшення напруження в руках рекомендується використовувати підлокітники довжиною від 250 мм, шириною 50-70 мм, що налаштовуються по висоті від 230 до 260 мм та по відстані між ними від 350 до 500 мм. Матеріал сидіння і спинки має бути напівтвердим, антиковзним, що пропускає повітря і легко миється, а також не збирає статичний заряд.

Робоча зона повинна бути обладнана підніжкою шириною мінімум 300 мм, глибиною не менше 400 мм, з можливістю регулювання по висоті до 150 мм і нахилом до 20°. На підніжці має бути рельєфна поверхня і маленький борт по зовнішньому краю висотою 10 мм. Екран комп'ютера слід розміщувати на оптимальній відстані від користувача, що варіюється в межах 500-700 мм, але не ближче ніж 500 мм, з урахуванням легкості читання тексту і зображень. Екран повинен бути розташований так, щоб забезпечити комфорт при спостереженні, кутом у 30° від вертикалі.

4.1.2 Освітлення

Ефективне та грамотне виробниче світло підвищує якість зорової діяльності, зменшує втомленість, стимулює зростання продуктивності, сприяє комфортному робочому середовищу, додаючи спокій та позитив працівникам, а також підсилює безпеку роботи, зменшуючи ризик травм. Недостатнє або надто яскраве освітлення може спричинити зорове перевтомлення і головний біль.

Недолік світла може призводити до перевантаження очей, зниження уваги та передчасної втоми. Занадто яскраве світло викликає сліпоту, незадоволення та відчуття дискомфорту в очах. Неправильне розташування світла може утворювати відблиски, контрастні тіні та дезорієнтувати працівника. Ці фактори

можуть спричинити нещасний випадок або професійні захворювання, тому важливо правильно розраховувати інтенсивність світла.

Розрізняють три типи світла - природне, штучне та комбіноване.

Природне світло – це варіант освітлення, при якому денне світло проникає через вікна або інші прозорі елементи приміщення. Інтенсивність природного світла може сильно коливатися в залежності від доби, сезону та інших факторів. Штучне світло використовують у вечірній час або коли природне світло недостатнє. Якщо природне світло доповнюється штучним, таке освітлення називають змішаним.

За своїм призначенням штучне світло може бути робочим, евакуаційним, аварійним чи охоронним. Робоче світло поділяється на загальне та місцеве. При загальному освітленні світильники розташовані рівномірно по приміщенню. У системі комбінованого освітлення до загального додається додаткове місцеве світло.

Згідно норм «ДБН В.2.5-28:2018 Природне і штучне освітлення», в комп'ютерних залах слід застосовувати комбінований тип освітлення. Для виконання робіт високої зорової точності (об'єкт розрізнення 0,3 ... 0,5 мм) інтенсивність природного світла має бути не менше 1,5%. Для робіт середньої точності (об'єкт розрізнення 0,5 ... 1,0 мм) - не менше 1,0%. Як джерела штучного світла часто використовують лампи типу ЛБ або ДРЛ, об'єднані в світильниках над робочими столами.

Вимоги до освітленості для робочих місць з комп'ютерами такі: при високій точності зорової роботи – 300 лк (загальне) та 750 лк (комбіноване); при середній точності – 200 та 300 лк відповідно.

Також важливо, щоб усе поле зору було якісно освітлене. Світлова інтенсивність приміщення та яскравість екрану комп'ютера повинні бути близькими, адже надлишкове світло може збільшувати втомленість очей.

4.1.3 Параметри мікроклімату

Для створення комфортних умов праці в приміщеннях з комп'ютерною технікою важливо дотримуватися певних стандартів мікроклімату. В таблиці 4.1 наведені параметри мікроклімату для таких приміщень

Таблиця 4.1 – Параметри мікроклімату для приміщень, де встановлені комп'ютери

Параметри мікроклімату	Літній період	Зимовий період
Температура повітря, °С	20 – 24	22 – 24
Відносна вологість, %	40 – 60	40 – 60
Швидкість руху повітря, м/с	0,1 – 0,2	0,1 – 0,2
Рівень шуму, дБ	до 50	До 50

4.1.4 Електромагнітне і іонізуюче випромінювання

Більшість науковців переконані, що короткочасна та тривала експозиція випромінюванням від екрана комп'ютера не шкодить здоров'ю працівників, які використовують персональні комп’ютери (ПК). Але повних даних про потенційну шкоду від такого випромінювання для людей, що працюють за комп'ютерами, ще немає, тому дослідження в цій сфері тривають.

Дозволені показники для неіонізуючого електромагнітного випромінювання від екрана ПК наведені в таблиці 4.2.

Зазвичай максимальна інтенсивність рентгенівського випромінювання на робочому місці користувача комп'ютера не перевищує 10 мкбер/год. Щодо ультрафіолетового та інфрачервоного випромінювання від екрану, їх інтенсивність знаходиться у діапазоні від 10 до 100МВт/м ^{2}.

4.1.1 Емоційна психогігієна

У сфері ІТ охорона праці включає не лише фізичне, але й психічне здоров'я працівників. Фактори, що впливають на психіку працівників у галузі інформаційних технологій, є різноманітними та мають велике значення у створенні здорового робочого середовища.

Таблиця 4.2 – Допустимі значення параметрів неіонізуючого електромагнітного випромінювання

Найменування параметра	Допустимі значення
Напруженість електричної складової електромагнітного поля на відстані 50см від поверхні монітора	10 В / м
Напруженість магнітної складової електромагнітного поля на відстані 50см від поверхні монітора	0,3 А / м
Напруженість електростатичного поля	20кВ / м

До них належать:

– тривалість та інтенсивність робочого дня: довгі години роботи без відпочинку можуть призвести до перевтоми та стресу, впливаючи на психічне здоров'я;

– терміни виконання завдань і робочий тиск: нереалістичні терміни виконання завдань або надмірний робочий тиск можуть викликати стрес та тривогу;

– міжособистісні взаємодії: конфлікти в команді або нездорова робоча атмосфера можуть спричиняти емоційне вигорання;

– робота в умовах ізоляції або віддалена робота: відсутність соціальної взаємодії або підтримки може вплинути на почуття ізоляції та самотності;

– work-life баланс : нездатність знайти баланс між роботою та особистим життям може призвести до стресу та емоційного вигорання;

– значне розумове навантаження: постійна потреба в освоєнні нових технологій та адаптації до змін може бути стресовою;

– відсутність автономії або контролю над роботою: обмежений контроль над робочими процесами та відсутність автономії можуть викликати почуття безпорадності та фрустрації.

Для ефективної профілактики психоемоційних проблем, пов'язаних з роботою в ІТ-галузі, необхідно розробити комплексний підхід, який враховує різноманітність факторів, що впливають на психічне здоров'я працівників. Значущість цього підходу полягає в його спрямованості на зменшення стресорів у робочому середовищі та підвищення ресурсів для психічного відновлення.

Першочергово, акцентується увага на регулюванні тривалості робочого дня та інтенсивності роботи. Це включає розробку ефективних графіків роботи, що передбачають достатні перерви для відновлення, та уникнення перевантаження завданнями. Крім того, розглядається важливість балансу між роботою та особистим життям, який може бути підтриманий через гнучкі робочі години та можливість дистанційної роботи.

Ще одним важливим аспектом є оптимізація робочого середовища з точки зору ергономіки. Покращення умов роботи на робочому місці, включаючи забезпечення якісного обладнання та комфортних умов, сприятиме зниженню фізичного та психологічного дискомфорту.

Ключову роль відіграє також розвиток корпоративної культури, яка підтримує психологічне благополуччя. Це передбачає створення відкритого, підтримуючого співробітництва та комунікації середовища, заохочення взаємодопомоги між колегами, та реалізацію програм з управління стресом.

Для забезпечення довгострокового психічного благополуччя, розглядається імплементація регулярних тренінгів та навчальних програм, спрямованих на розвиток навичок управління стресом та підвищення особистісної стійкості. Також важливим є застосування систематичного моніторингу психічного стану працівників з використанням психометричних інструментів для раннього виявлення потенційних проблем.

Розуміння та належне врахування цих факторів є ключовим для забезпечення психічного благополуччя працівників у галузі ІТ, що, у свою чергу, позитивно впливає на їх продуктивність та загальне задоволення роботою.

4.2 Питання щодо безпеки в надзвичайних ситуаціях

В умовах швидкого розвитку інформаційних технологій та зростання залежності бізнес-процесів від ІТ-систем, питання безпеки в сфері ІТ набуває особливої актуальності. Розглядаючи безпеку праці в ІТ, особливу увагу необхідно приділити розробці та імплементації процедур реагування на надзвичайні ситуації, що включають технічні збої, кібератаки, витік конфіденційної інформації, фізичні загрози обладнанню та перебої в електропостачанні, а також інші аварійні стани, здатні порушити звичний хід робочих процесів.

Відповідальність за реалізацію ефективної системи безпеки лежить на ІТ-спеціалістах, керівництві компаній та інших зацікавлених сторонах. Необхідно забезпечити встановлення чітких правил та інструкцій, які регламентують дії персоналу у випадку виникнення надзвичайних ситуацій. Це включає визначення швидких комунікаційних каналів, розробку планів відновлення роботи систем, а також проведення регулярних тренінгів та інструктажів з охорони праці для всіх працівників.

Плани евакуації та реагування мають бути адаптовані до можливих ІТ-ризиків. Вони повинні включати дії для захисту життя та здоров'я працівників, а також процедури збереження даних та обладнання. Плани мають бути детальними та зрозумілими для кожного члена команди, з чіткими інструкціями щодо дій в кризових ситуаціях.

Регулярні тренінги з охорони праці, які охоплюють надзвичайні ситуації в ІТ, є необхідними для підготовки персоналу до ефективного реагування на інциденти. Навчальні програми повинні включати імітації надзвичайних ситуацій, навчання з використанням спеціалізованого обладнання та вивчення

найкращих практик у сфері кібербезпеки. Систематичний аналіз ризиків є важливим для ідентифікації потенційних вразливостей системи та розробки відповідних заходів запобігання. Встановлення систем моніторингу, які можуть виявити ознаки надзвичайних ситуацій на ранніх стадіях, допомагає зменшити потенційні збитки та забезпечити оперативне реагування.

Після кожної надзвичайної ситуації необхідно проводити детальний аналіз її причин, ефективності дій персоналу та наявних процедур реагування. На основі цього аналізу слід вносити корективи у плани надзвичайних дій, поліпшувати процедури безпеки та організовувати додаткове навчання. Ефективне управління надзвичайними ситуаціями в IT-сфері вимагає всебічного підходу, який поєднує в собі як технічні, так і організаційні аспекти. Відповідальність за це лежить на всіх рівнях організації, починаючи з IT-фахівців і закінчуючи вищим керівництвом. Завдяки встановленню та дотриманню відповідних процедур можна мінімізувати ризики та забезпечити безперебійну роботу в умовах, що постійно змінюються.

ВИСНОВКИ

У цій кваліфікаційній роботі ми дійшли до висновку, що навіть попри те, що деякі гнучкі практики існують вже довгий час, самі гнучкі методології є відносно новими та здобули широке поширення у світі бізнесу. Серед розробників існує величезна розбіжність у знаннях щодо рівня програмного забезпечення, яке створюється. Розробники повинні бути обізнані з тим, як їхні гнучкі підходи можна оновлювати або адаптувати, щоб забезпечити найвищу якість роботи.

Було показано, що безперервна інтеграція значно покращує якість програмного забезпечення в цілому, що спонукає виробників програмного забезпечення адаптувати свої методи розробки в цьому напрямку. Значна кількість небезпек, пов'язаних з процесом інтеграції програмного забезпечення, була зменшена завдяки безперервній інтеграції частин розробленого програмного забезпечення, щойно вони стають доступними.

Метою цього дослідження було дослідити рівень охоплення забезпеченням якості, що пропонується методологіями Lean для гнучкої розробки програмного забезпечення. Було розглянуто паралелі та розбіжності між технічними підходами гнучкої та планово-орієнтованої методології. Теоретичні основи якості програмного забезпечення були розроблені з акцентом на перспективу якості ISO-9126 та основи забезпечення якості, при цьому традиційна точка зору на якість використовувалася як відправна точка для наближення до цієї мети.

Потім, зіставляючи гнучкі концепції з відповідними їм викликами та зіставляючи фундаментальні принципи тестування з непослідовними практиками в ASWD, стало можливим виявити проблеми та слабкі сторони ASWD у зв'язку зі забезпеченням якості (SQA). Щоб з'ясувати, чи відповідає ASWD SQA, були проведені обидва ці порівняння.

Наприклад, було зазначено, що гнучкі методології не застосовують явних показників якості, коли йдеться про їхні керівні принципи. Ця відсутність прямих показників якості була визнана потенційною слабкістю методу. Крім того, було визначено, що певні навички, деструктивний підхід та незалежність тестування суперечать тому, як зараз використовуються гнучкі підходи. Включення додаткових методів тестування, які спочатку не розглядаються в самих гнучких методологіях, може покращити операції ASWD.

Для рівня часу ітерації було запропоновано позицію незалежного тестувальника, а для рівня часу реалізації окремої вимоги (так званий heartbeat) розглядається дослідницьке (explanatory) тестування на основі сесій. Після ретельного розгляду було визначено, що методи тестування повинні бути охоплені протягом цих двох періодів.

У цій роботі перераховано наступні проблеми, пов'язані з гнучким тестуванням. Щоб визначити, чи є покращення дійсно необхідним, враховуючи відмінності між окремими методами, які більш-менш включають явні дії з тестування у свій порядок денний, спочатку необхідно надати докази достатності існуючих практик SQA, що використовуються для існуючих ASWD. Це робиться для того, щоб оцінити, чи дійсно потрібне покращення. По-друге, необхідні додаткові дослідження, щоб з'ясувати, чи є методи тестування ефективними в гнучкій розробці, а також чи здатні вони відповідати вимогам якості, що висуваються традиційними способами ведення бізнесу. Наприклад, два потенційні підходи, які слід розглянути, – це розробка на основі поведінки (BDD) та розробка на основі приймального тестування (ATDD).

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Ihor, B., Oleksii, D., Aleksandr, K., Nataliia, K., Oleksandr, M., & Volodymyr, P. (2019, January). Multicriteria choice of software architecture using dynamic correction of quality attributes. In International Conference on Computer Science, Engineering and Education Applications (pp. 419-427). Cham: Springer International Publishing.
2. Bodnarchuk, I., Lisovyi, V., Kharchenko, O., & Galai, I. (2018, September). Adaptive method for assessment and selection of software architecture in flexible techniques of design. In 2018 IEEE 13th International Scientific and Technical Conference on Computer Sciences and Information Technologies (CSIT) (Vol. 1, pp. 292-297). IEEE.
3. Kharchenko, A., Raichev, I., Bodnarchuk, I., & Matsiuk, O. (2021, October). The Survey of Global Software Design Processes. In 2021 IEEE 8th International Conference on Problems of Infocommunications, Science and Technology (PIC S&T) (pp. 291-294). IEEE.
4. Волович, В., Береженко, Б. М., & Боднарчук, І. О. (2022). Задача проєктування програмної архітектури в процесах забезпечення якості. Матеріали X науково-технічної конференції „Інформаційні моделі, системи та технології “Тернопільського національного технічного університету імені Івана Пулюя, 104-106.
5. Боднарчук, І., Харченко, О., Хоміцький, Б., & Шимчук, Г. (2019). Проектування архітектури програмних систем в проектах з гнучкими методами управління. Матеріали XXI наукової конференції Тернопільського національного технічного університету імені Івана Пулюя, 46-48.
6. Fontana, R. M., Meyer Jr, V., Reinehr, S., & Malucelli, A. (2015). Progressive Outcomes: A framework for maturing in agile software development. Journal of Systems and Software, 102, 88-108.
7. Ambler, S., (2005), Quality in an Agile World, Software Quality Professional, Vol. 7, No. 4, pp. 34-40.

8. Humble, Jez, and David Farley. Continuous delivery: reliable software releases through build, test, and deployment automation. Pearson Education, 2010.
9. Poornalinga, K. S., & Rajkumar, P. (2016). Survey on Continuous Integration, Deployment and Delivery in Agile and DevOps Practices. International Journal of Computer Sciences and Engineering, 4(4), 213-216.
10. Lai, S. T., & Leu, F. Y. (2016, July). A Version Control-based Continuous Testing Frame for Improving the IID Process Efficiency and Quality. In 2016 10th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS) (pp. 464-469). IEEE.
11. Sagheer, M., Zafar, T., & Sirshar, M. (2015). A framework for software quality assurance using agile methodology. International Journal of Scientific & Technology Research, 4(2), 44-50.
12. Boehm, B., & Turner, R. (2003). Using risk to balance agile and plan-driven methods. Computer, 36(6), 57-66.
13. Gregory, J., & Crispin, L. (2014). More agile testing: learning journeys for the whole team. Addison-Wesley Professional.
14. A. Dyck, R. Penners, and H. Lichter - Towards definitions for release engineering and devops. In Release Engineering (RELENG), 2015 IEEE/ACM 3rd International Workshop on, pages 3–3, May 2015
15. Zhu, L., Bass, L., & Champlin-Scharff, G. (2016). DevOps and its practices. IEEE software, 33(3), 32-34.
16. Fitzgerald, B., & Stol, K. J. (2014, June). Continuous software engineering and beyond: trends and challenges. In Proceedings of the 1st International Workshop on rapid continuous software engineering (pp. 1-9).
17. Faber, Frank. "Testing in DevOps." The Future of Software Quality Assurance. Springer, Cham, 2020. 27-38.
18. S. Sukhpal and C. Inderveer. (2012). Enabling Reusability in Agile Software Development.
19. Manifesto, A. (2001). Manifesto for agile software development.

20. Mnkandla, E., & Dwolatzky, B. (2006, October). Defining agile software quality assurance. In 2006 International Conference on Software Engineering Advances (ICSEA'06) (pp. 36-36). IEEE.
21. Itkonen, J., Rautiainen, K. &Lassenius, C. (2005). Towards understanding quality assurance in agile software development. In H.E. Andersin, E. Niemi & V. Hirvonen (Ed.), ICAM 2005. Proceedings of the International Conference on Agility (pp. 201-207). Helsinki, Finnland
22. Rautiainen, K. (2004). Cycles of Control: A temporal pacing framework for software product development management (Licentiate thesis). Helsinki University of Technology, Helsinki, Finland
23. Beck, K. 1999. Embracing change with extreme programming. Computer, 32(10), 70-77.
24. Stapleton, J. (1997). Dynamic systems development method. Harlow, England: Addison-Wesley
25. Palmer, S.R., J.M. Felsing 2002. A Practical Guide to Feature Driven Development. Upper Saddle River: Prentice-Hall.
26. Veenendaal, E. (2018). Test techniques for the test analyst [e-book]. Retrieved from http://www.erikvanveenendaal.nl/site/wpcontent/uploads/Test-Techniques-for-the-Test-AnalysteBook.pdf
27. Itkonen, J., Mäntylä, M. V. & Lessenius, C. (2012). The role of the tester's knowledge in exploratory software testing. IEEE Transactions on Software Engineering, 39(5).
28. Northrop, L., Clements, P., Bachmann, F., Bergey, J., Chastek, G., Cohen, S., & O'Brien, L. (2007). A framework for software product line practice, version 5.0. SEI. – 2007. http://www.sei.cmu.edu/productlines/index.html.
29. Campos, J., Arcuri, A., Fraser, G., & Abreu, R. (2014). Continuous test generation: enhancing continuous integration with automated test generation. In Proceedings of the 29th ACM/IEEE international conference on Automated software engineering, 55-66.

30. Beaumont, O., Bonichon, N., Courtès, L., Hanin, X., &Dolstra, E. (2012, May). Mixed data-parallel scheduling for distributed continuous integration. In Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International (pp. 91- 98). IEEE
31. Харченко, О., Боднарчук, І., & Галай, І. (2013). Метод для отримання множини показників якості архітектури програмного забезпечення.
32. Pettichord, B., – Agile Testing Challenges, Pacific Northwest Software Quality Conference 2004.
33. Amit Juyal, Umesh Kumar Tiwari, Lata Nautiyal, Shashidhar G. Koolagudi, ―Agile Plus Comprehensive model for Software Development, In International Journal Computer Technology& Applications, Volume 3 (4), 1378-1383
34. Bhalerao, S., D. Puntambekar and Ingle, M., Generalizing Agile Software Development Life Cycle, In International Journal on Computer Science and Engineering Vol.1(3), 2009, 222-226.
35. Kohl, J., ― Pair Testing. Better Software, Jan 2004 [66] Anita, Naresh Chauhan, ― a Framework for Quality Improvement in Distributed Agile Environment, IEEE International Conference on Research and Development Prospects on Engineering and Technology, March 2013, E. G. S. Pillay Engineering College, Tamilnadu, India
36. Anita, Naresh Chauhan, ― a Framework for Quality Improvement in Distributed Agile Environment‖, IEEE International Conference on Research and Development Prospects on Engineering and Technology, March 2013, E. G. S. Pillay Engineering College, Tamilnadu, India.
37. Anita, Naresh Chauhan, ― an Object Oriented Design Approach for Modification of Rotten Code Using Regression Testing & Refactoring‖, ― an Object Oriented Design Approach for Modification of Rotten Code Using Regression Testing & Refactoring‖ Volume 4, Number 7, 2014, pp, 681- 686
38. Anita, Naresh Chauhan, ― A Regression Test Selection Technique by Optimizing User Stories in an Agile Environment, 4th IEEE International Advanced Computing Conference, IACC 2014 (21st -22nd Feb 2014), in ITM University, Gurgaon, India

39. Anita, Naresh Chauhan, ― A Linguistic approach for TCP in an Agile Environment, 13th Annual International Software Testing Conference (4th - 5 th Dec 2013), Crossing the Chasm: From Assurance to Confirmation, Bangalore, India.
40. Gamma, E., & Beck, K. (2004). Contributing to Eclipse: principles, patterns, and plug-ins. Addison-Wesley Professional.
41. Saff, D. and Ernst, M. D. (2004). An experimental evaluation of continuous testing during development. In ISSTA 2004, Proceedings of the 2004 International Symposium on Software Testing and Analysis, pages 76 – 85, Boston, MA, USA
42. Roche, J. (2013). Adopting DevOps practices in quality assurance. Communications of the ACM, 56(11), 38-43.
43. E. Laukkanen, J. Itkonen, and C. Lassenius, "Problems, causes and solutions when adopting continuous delivery — A systematic literature review, Inf. Softw. Technol., vol. 82, pp. 55–79, 2017.
44. Faheem Ullah, Adam Johannes Raft, Mojtaba Shahin, Mansooreh Zahedi, and Muhammad Ali Babar. 2017. Security Support in Continuous Deployment Pipeline. In Proc. 12th International
45. Mäntylä, M. V., Adams, B., Khomh, F., Engström, E., & Petersen, K. (2015). On rapid releases and software testing: a case study and a semi-systematic literature review. Empirical Software Engineering, 20, 1384-1425.
46. L. Lwakatare, T. Karvonen, T. Sauvola, P. Kuvaja, H. Olsson, J. Bosch and M. Oivo, ― Towards DevOps in the Embedded Systems Domain: Why is It so Hard?, 2016 49th Hawaii International Conference on System Sciences, 2016.
47. Garousi, V., Felderer, M., Mäntylä, M.V. (2019) Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. Information Software Technology 106: 101-121.
48. Sturtevant, D. (2017) Modular Architectures Make You Agile in the Long Run. IEEE Software, 35(1): p. 104-108.
49. Taibi, D., Lenarduzzi, V., & Pahl, C. (2019). Continuous architecting with microservices and devops: A systematic mapping study. In Cloud Computing and

Services Science: 8th International Conference, CLOSER 2018, Funchal, Madeira, Portugal, March 19-21, 2018, Revised Selected Papers 8 (pp. 126-151). Springer International Publishing.

50. Nybom, Kristian, Jens Smeds, and Ivan Porres. ― On the impact of mixing responsibilities between devs and ops. In International Conference on Agile Software Development, pp. 131-143. Springer, Cham, 2016.
51. Lwakatare, Lucy Ellen, Terhi Kilamo, Teemu Karvonen, Tanja Sauvola, Ville Heikkilä, Juha Itkonen, Pasi Kuvaja, Tommi Mikkonen, Markku Oivo, and Casper Lassenius. ― DevOps in practice: A multiple case study of five companies. Information and Software Technology 114 (2019): 217-230.
52. Sujay Honnamane. Rameshkumar Bar Jumpstarting DevOps with Continuous Testing 2015.
53. Jones, S., Noppen, J. and Lettice, F., 2016, July. Management challenges for DevOps adoption within UK SMEs. In Proceedings of the 2nd International Workshop on quality-aware DevOps (pp. 7-11).
54. Chen, Lianping. ― Continuous delivery: Overcoming adoption challenges. Journal of Systems and Software 128 (2017): 72-86.
55. Ibrahim, Mahmoud Mohammad Ahmad, Sharifah Mashita Syed-Mohamad, and Mohd Heikal Husin. ― Managing quality assurance challenges of DevOps through analytics. In Proceedings of the 2019 8th International Conference on Software and Computer Applications, pp. 194-198. 2019.
56. Wang, Y., Pyhäjärvi, M. and Mäntylä, M.V., 2020. Test Automation Process Improvement in a DevOpsTeam: Experience Report. arXiv preprint arXiv:2004.06381
57. Leite, Leonardo, Carla Rocha, Fabio Kon, Dejan Milojicic, and Paulo Meirelles. ― A survey of DevOps concepts and challenges. ACM Computing Surveys (CSUR) 52, no. 6 (2019): 1-35.
58. Mascheroni, Maximiliano A., and Emanuel Irrazábal ― Continuous testing and solutions for testing problems in continuous delivery: A systematic literature review. Computación y Sistemas 22, no. 3 (2018): 1009-1038.

59. Surendra Naidu Mullaguru — Changing Scenario of Testing Paradigms Using DevOps–A Comparative Study with Classical Models, Global Journal of Computer Science and Technology. 2015.
60. Virmani, M., 2015, May. Understanding DevOps and bridging the gap from continuous integration to continuous delivery. In Fifth International Conference on the Innovative Computing Technology (INTECH 2015) (pp. 78-82). IEEE.
61. Hemant Madaan, (2023). Continuous Testing: The Key to Quality Assurance in the DevOps Era.https://devops.com/continuous-testing-the-key-toquality-assurance-in-the-devops-era/ (10.05.2025).

ДОДАТКИ

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Тернопільський національний технічний університет імені Івана Пулюя (Україна)
Університет імені П'єра і Марії Кюрі (Франція)
Маріборський університет (Словенія)
Технічний університет у Кошице (Словаччина)
Вільнюський технічний університет ім. Гедимінаса (Литва)
Наукове товариство ім. Т.Шевченка

АКТУАЛЬНІ ЗАДАЧІ СУЧАСНИХ ТЕХНОЛОГІЙ

Збірник
тез доповідей

XIII Міжнародної науково-практичної конференції молодих учених та студентів
11-12 грудня 2024 року

УКРАЇНА
ТЕРНОПІЛЬ – 2024

СЕКЦІЯ: КОМП'ЮТЕРНО-ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ ТА СИСТЕМИ ЗВ'ЯЗКУ

УДК 004.05; 004.42; 004.9
Назар Осідак, Олександр Цвігун
Тернопільський національний технічний університет імені Івана Пулюя

ЗАСТОСУВАННЯ ШІ ДЛЯ ЗАДАЧ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ

Nazar Osidak, Oleksandr Tsvihun

APPLICATION OF AI FOR AUTOMATED TESTING TASKS

Штучний інтелект (ШІ) має великий потенціал у автоматизації тестування програмного забезпечення, але його застосування залежить від конкретних завдань. Існують типові ситуації, коли варто використовувати ШІ для автоматизації тестування, а є також випадки, коли традиційні методи можуть бути більш ефективними [1].

Штучний інтелект найкраще підходить для автоматизації тестів, де є потреба в адаптації, обробці великих обсягів даних, або в ситуаціях, коли специфікація тестів неповна або має суперечності. Традиційні методи автоматизації залишаються ефективними для стабільних, стандартизованих тестів.

Опишемо коротко завдання, які можна вирішити за допомогою ШІ. В першу чергу це тестування на основі поведінки користувача (User Behavior Testing): ШІ може аналізувати дані поведінки користувачів та автоматично генерувати тести, що імітують реальні сценарії взаємодії з продуктом. Наприклад, використання алгоритмів машинного навчання для створення сценаріїв тестування, що адаптуються під реальні патерни поведінки.

ШІ може автоматично визначати, які тестові сценарії виконувати в першу чергу, щоб мінімізувати час тестування при максимальному покритті. Таким чином вирішується задача оптимізації сценаріїв тестування. Алгоритми, зокрема на основі навчання з підкріпленням, можуть оптимізувати вибір тестів, залежно від результатів попередніх запусків і змін у коді. У випадках, коли програма працює з великими обсягами даних або складними системами, ШІ може застосовувати методи аналізу великих даних для пошуку потенційних дефектів у системах, що важко протестувати вручну. ШІ може допомогти в тестуванні програм, де специфікація неповна або містить суперечності. Застосування технологій, таких як нейронні мережі, може допомогти адаптувати тестування до змін у поведінці програми.

Існує також не менший список задач, які краще вирішувати без ШІ. Регресійне тестування для випадків, коли програма має добре визначену структуру та стабільний функціонал. Тоді традиційні методи автоматизації тестів (наприклад, за допомогою Selenium або JUnit) можуть бути більш ефективними. Це дозволяє виконувати стандартні перевірки без додаткових затрат на навчання моделей ШІ.

Статичні тести на навантаження або стрес-тести, які вимірюють стабільність і продуктивність програми, можуть бути виконані без ШІ. Вони потребують чітких та вимірних результатів, які краще піддаються автоматизації без залучення складних технологій. У випадку функціонального тестування прості функціональні тести, коли потрібно перевірити, чи працюють конкретні функції програми відповідно до вимог, можуть бути виконані без ШІ за допомогою традиційних інструментів автоматизації. У таких випадках ШІ не дасть значного покращення в продуктивності.

Література

1. Цісарук Д. В. Методи і засоби тестування програмного забезпечення комп'ютерних систем з використанням алгоритмів машинного навчання : кваліфікаційна робота магістра за спеціальністю „123 — комп'ютерна інженерія“ / Д. В. Цісарук. – Тернопіль, 2021. – 90 с.