Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту
(повна назва)
Кафедра Інформатики
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

ВИВЧЕННЯ ЗАСОБІВ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ ДЛЯ ОЦІНЮВАННЯ ПОКАЗНИКІВ ЯКОСТІ ПРОГРАМНОГО ПРОДУКТУ
(тема)

Виконав:
студент 2 курсу, групи СІм-20-1
Василенко І. С.
(прізвище, ініціали)

Спеціальності 122 Комп’ютерні науки
(код і повна назва спеціальності)

Тип програми освітньо-професійна

Освітня програма Соціальна інформатика
(повна назва освітньої програми)

Керівник проф. Гороховатський В.О.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри ____________ Кобилін О.А.
(підпис) (прізвище, ініціали)

2021 р.

Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту
(повна назва)

Кафедра Інформатики
(повна назва)

Рівень вищої освіти другий (магістерський)

Спеціальність 122 Комп'ютерні науки
(код і повна назва)

Тип програми освітньо-професійна

Освітня програма Соціальна інформатика
(повна назва освітньої програми)

ЗАТВЕРДЖУЮ:
Зав. кафедри
____
(підпис)
«____»______________ 2021 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Василенко Ірині Сергіївні
(прізвище, ім'я, по батькові)

1. Тема роботи Вивчення засобів автоматизованого тестування для оцінювання показників якості програмного продукту
затверджена наказом по університету від «_22_» жовтня 2021 року № 1573Ст.
2. Термін подання студентом роботи до екзаменаційної комісії _26_ листопада_ 2021 р.
3. Вихідні дані до роботи методології тестування: каскадна модель, V-подібна модель, ітераційна інкрементальна модель, спіральна модель, гнучка модель; вимоги програмного застосунку, тестове покриття вимог, види тестування програмного продукту.

4. Перелік питань, що потрібно опрацювати в роботі
1. Огляд методологій тестування.
2. Вимоги програмного застосунку.
3. Тестове покриття вимог.
4. Види тестування програмного продукту.
5. Оцінка ефективності автоматизованого тестування.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри) актуальність, автоматизоване тестування, постановка задачі, тестовий код автоматизованого теста.

6. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата
Консультант з дотримання діючих стандартів та норм	Доцент Бєлова Н.В.		

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Отримання завдання на кваліфікаційну роботу	22.10.2021	
2	Аналіз завдання, підбір літератури	22.10.21-04.11.21	
3	Аналіз літератури з досліджуваної проблеми	05.11.21-15.11.21	
4	Аналіз технічних засобів	17.11.21-26.11.21	
5	Розробка методу	27.11.21-31.11.21	
6	Програмна реалізація	01.11.21-07.11.21	
7	Оформлення пояснювальної записки	08.11.21-15.11.21	
8	Перевірка на плагіат	20.11.2021	
9	Рецензування	28.11.2021	
10	Підготовка презентації та доповіді	29.11.2021	
11	Занесення роботи в електронний архів	03.11.2021	
12	Попередній захист кваліфікаційної роботи	06.11.2021	

Дата видачі завдання __22____ _жовтня__ 2021 р.

Студент ____________________________________
(підпис)

Керівник роботи ___________________________ __проф. Гороховатський В. О.__
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ/ABSTRACT

Пояснювальна записка до кваліфікаційної роботи: 95 с., 4 табл., 16 рис., 1 дод., 44 джерела.

АВТОМАТИЗОВАНЕ ТЕСТУВАННЯ, МАНУАЛЬНЕ ТЕСТУВАННЯ, ВИДИ ТЕСТУВАННЯ ПРОГРАМНОГО ПРОДУКТУ, ТЕСТОВЕ ПОКРИТТЯ ВИМОГ, МЕТОДОЛОГІЇ ТЕСТУВАННЯ.

Об’єктом дослідження є забезпечення якості та надійності програмного забезпечення.

Метою дослідження є вивчити засоби автоматизованого тестування для оцінювання якості програмного продукту.

Використано методології тестування та види тестування програмного продукту. Проведено дослідження видів тестування програмного продукту. Проведено аналіз переваг та недоліків автоматизованого тестування у порівнянні з мануальним тестуванням. Розроблена програма автоматизованого тестування та декілька мануальних тестів.

У результаті роботи здійснена програмна реалізація автоматизованого тесту для сайту.

AUTOMATED TESTING, MANUAL TESTING TYPES OF SOFTWARE TESTING, TEST COVERING REQUIREMENTS, TESTING METHODOLOGY.

The object of research is to ensure the quality and reliability of software.

The aim of the study is to examine the means of automated testing to assess the quality of the software product.

Testing methodologies and types of software product testing are used. A study of the types of software product testing was conducted. The analysis of advantages and disadvantages of automated testing in comparison with manual testing is carried out. An automated testing program and several manual tests have been developed.

As a result of work the program realization of the automated test for a site is carried out.

ЗМІСТ

Перелік умовних позначень, символів, одиниць, скорочень і термінів 7
Вступ 8
1 Огляд існуючих засобів автоматизації тестування 11
1.1 Тестування на рівні коду 11
1.2 Тестування програм через графічний інтерфейс 14
1.3 Рекомендації для ефективної автоматизації тестування 18
1.4 Рекомендації до вибору інструмента автоматизації 19
1.5 Детальніше про найпопулярніші інструменти автоматизації тестування 20
1.6 Постановка задачі дослідження 23
2 Організація тестування програмного продукту 25
2.1 Процеси тестування 25
2.2 Методології тестування 29
2.2.1 Каскадна модель 29
2.2.2 V-подібна модель 30
2.2.3 Ітераційна інкрементальна модель 31
2.2.4 Спіральна модель 32
2.2.5 Гнучка модель 34
2.2.6 Аналіз якості програмного забезпечення 37
2.2.7 Вимоги програмного застосунку 41
3 Види тестування програмного продукту 52
3.1 Функціональні види тестування 52
3.2 Нефункціональні види тестування 54
3.3 Пов’язані зі змінами види тестування 56
3.4 Впровадження автоматизованих тестів 58
3.5 Оцінка ефективності автоматизації тестування 67
3.6 Оцінка повноти тестування 71

3.7 Структурні критерії 72
3.8 Критерії повноти на основі структури вхідних даних 78
3.9 Критерії повноти на основі вимог 79
3.10 Критерії повноти на основі припущень про помилки..................... 81
Висновки 83
Перелік джерел посилання... 84
Додаток А Лістинг програми 89

ПЕРЕЛІК УМОВНИХ ПОЗНЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ПЗ – програмне забезпечення

Графічний інтерфейс користувача (ГІП), графічний інтерфейс користувача (ДПІ) (англ. graphical user interface, GUI) – система засобів для взаємодії користувача з електронними пристроями, заснована на представленні всіх доступних користувачеві системних об’єктів і функцій у вигляді графічних компонентів екрану піктограм, меню, кнопок, списків тощо)

ВСТУП

Програмне забезпечення грає все більшу роль з розвитком обчислювальних машин для вирішення завдань, що пов’язані з інформаційними технологіями. Коли були ще ранні етапи розвитку інформаційних технологій, створити програму було дуже дорогим і складним процесом. Це вимагало обладнання, що дорого коштувало та високої кваліфікації працівників. Тож це обладнання могли придбати лише великі підприємства і інститути. Але на цей день практично кожен має доступ до великих обчислювальних потужностей [1].

Вже зараз створена велика кількість мов програмування, деякі з них мають своєю метою навчання програмуванню. Також є різноманітні підходи та методики, які роблять легшим та структурують створення програм. Раніше багато задач вирішувалися лише за допомогою використання дорогого спеціального і важкого в освоєнні обладнання [2]. Зараз це вирішується завдяки більш дешевим програмно-апаратним комплексам. Після цього значно більша кількість людей змогла почати займатись програмуванням.

Можна виділити три основні групи: люди, які програмують професійно; люди, які займаються програмуванням, як допоміжним інструментом; люди, які програмують тому що їм цікаво. Велика кількість програм закінчує своє існування так і не закінченими а в інших програмах обов’язково існують помилки. Ці помилки можуть потягти за собою різноманітного ступеня тяжкості в залежності від того в якій сфері використовується. Чим важливіша ця сфера, тим більш строгі мають бути вимоги до якості цієї програми. Неякісна програма не буде конкурентна на ринку і не принесе бажаного прибутку. Тож важливим буде забезпечити необхідний рівень якості, якщо програма буде виконуватись дуже повільно або при виконанні видавати невірні данні на якихось комбінаціях вхідних даних, її використання не буде доцільним.

Для визначення якості програмного продукту використовують набір атрибутів, що можуть показати на скільки продукт відповідає вимогам зацікавлених сторін. Але якість може визначатися по різному для різних зацікавлених сторін. Тож для забезпечення якості продукту потрібно визначити всі зацікавлені сторони, також їх думки з приводу якості та знайти рішення яке буде задовольняти усі сторони. Однаково ефективних методів забезпечення якості продукту немає. Та при підготовці будь-якого програмного продукту можливо знайти і впровадити практики і підходи, які будуть ефективними для показання якості, вартості та часу розробки.

Тестування є одним з основних способів забезпечення якості програмного продукту. Тестування програмного продукту полягає у виконанні програми на деякій множині даних і зіставлення отриманих результатів з очікуваними (еталонними) з метою встановити чи задовольняє додаток до покладених завдань. Будучи одним з основних етапів розробки програмного продукту, тестування вносить значний вклад в загальну трудомісткість процесу розробки [2].

При автоматизованому тестуванні для виконання тестів і перевірки результатів використовуються програмні засоби. При використанні методології гнучкого програмування TDD (Test Driven Development, Розробка через тестування) тести пишуться до написання коду програми, і програма вважається закінченою, коли проходить всі тести. У цьому підході автоматизація тестування закладена в самій методології. Автоматизація тестування при правильному застосуванні дозволяє значно знизити час, необхідний на тестування. Воно не може повністю замінити ручне тестування, але зате високо ефективно при модульному, регресійому, навантажувальному, інсталяційному і деяких інших видах тестування.

Найперша «автоматизація» з’явилася в епоху операційних систем DOS і CP / M. Тоді вона полягала у віддачі з додатком команд через командний рядок і аналізі результатів. Трохи пізніше додалися віддалені виклики через API для роботи по мережі. Вперше про автоматизованому

тестуванні згадується в книзі Фредеріка Брукса «Міфічний людино-місяць», де йдеться про перспективи використання модульного тестування. Але по-справжньому автоматизація тестування стала розвиватися тільки в 80-х роках.

Існує два основні підходи до автоматизації тестування: тестування на рівні коду і GUI-тестування. До першого типу відносяться, зокрема, модульне тестування. До другого – імітація дій користувача за допомогою спеціальних тестових фреймворків [2, 3].

Найбільш поширеною формою автоматизації є тестування додатків через графічний користувальницький інтерфейс. Популярність такого виду тестування пояснюється двома факторами: по-перше, застосунок тестується тим же способом, яким його буде використовувати людина, по-друге, можна застосунок, не маючи при цьому доступу до вихідного коду.

Важливо правильно приймати рішення щодо введення автоматизації конкретних стадій тестування. Наприклад, при вирішенні автоматизувати тестування взаємодії з графічним інтерфейсом користувача, тести придеться коригувати при кожній зміні інтерфейсу. Може виявитися так, що витрати на підтримку таких тестів в актуальному стані потрібно більше ресурсів, ніж на виконання тестування вручну.

Також не можна забувати, що будь-яке тестування і є невід’ємною частиною життєвого циклу програми, тільки воно не може гарантувати високу якість. Особливу увагу слід приділяти на стадіях підготовки вимог і створення архітектури продукту. Багато авторів схиляються до того, що чим раніше знайдений дефект, тим дешевше його усунути.

У даній роботі розглянуті моделі життєвого циклу програмного продукту, наведені критерії якості – це міра, яка дозволяє отримати числове значення деяких властивостей ПЗ, відображає оперативну інформацію про його поточний стан, описані існуючі рішення для автоматизації тестування, проаналізовано їх переваги та недоліки, оцінена ефективність автоматизації тестування, надано рекомендації щодо впровадження автоматизованого тестування в процес розробки програмних продуктів [3].

1 ОГЛЯД ІСНУЮЧИХ ЗАСОБІВ АВТОМАТИЗАЦІЇ ТЕСТВАННЯ

1.1 Тестування на рівні коду

Цей підхід використовується в основному при модульному і регресійному тестуванні.

Модульне тестування перевіряє функціональність і шукає дефекти в частинах застосунку, які доступні і можуть бути протестовані по-окремо (модулі програм, об'єкти, класи, функції і т.д.). Зазвичай компонентне (модульне) тестування проводиться викликаючи код, який необхідно перевірити і при підтримці середовищ розробки, таких як фреймворки (frameworks – каркаси) для модульного тестування або інструменти для налагодження [4]. Всі знайдені дефекти, як правило виправляються в коді без формального їх опису в системі менеджменту багів (Bug Tracking System).

Переваги модульного тестування включають:

– заохочення змін. Модульне тестування пізніше дозволяє програмістам проводити рефакторинг, будучи впевненими, що модуль як і раніше працює коректно (регресійні тестування). Це заохочує програмістів до змін коду, оскільки досить легко перевірити, що код працює і після змін;

– спрощення інтеграції. Модульне тестування допомагає усунути сумніви з приводу окремих модулів і може бути використано для підходу до тестування «знизу вгору»: спочатку тестуються окремі частини програми, потім програма в цілому. Такий підхід значно спрощує інтеграційне тестування. Варто відзначити, що навіть ретельно побудована ієрархія модульних тестів не тотожна інтеграційного тестування. Воно поки що не піддається повній автоматизації і вимагає участі людей тестувальників;

– документування коду. Модульні тести можна розглядати як «живий документ» для тестованого класу. Клієнти, які не знають, як

використовувати даний клас, можуть використовувати модульний тест як приклад;

– відділення інтерфейсу від реалізації. Оскільки деякі класи можуть використовувати інші класи, тестування окремого класу часто поширюється на пов'язані з ним. Наприклад, клас користується базою даних; в ході написання тесту програміст виявляє, що тесту доводиться взаємодіяти з базою. Це помилка, оскільки тест не повинен виходити за кордон класу. В результаті розробник абстрагується від з'єднання з базою даних і реалізує цей інтерфейс, використовуючи свій власний mock-об'єкт (об'єкт, що реалізують задані аспекти модельованого програмного оточення) [4]. Це призводить до менш пов'язаному коду, мінімізуючи залежності в системі.

Також, у цього виду тестування є певні обмеження і недоліки:

– не дозволяє виявити всі помилки. Це випливає з практичної неможливості трасування всіх можливих шляхів виконання програми, за винятком найпростіших випадків. Крім того, відбувається тестування кожного з модулів окремо. Це означає, що помилки інтеграції, системного рівня, функцій, виконуваних в декількох модулях не будуть визначені. Крім того, дана технологія марна для проведення тестів на продуктивність. Таким чином, модульне тестування більш ефективно при використанні в поєднанні з іншими методиками тестування;

– великий обсяг тестового коду. Тестування програмного забезпечення – комбінаторна задача. Наприклад, кожне можливе значення булевою змінної зажадає двох тестів: один на варіант TRUE, інший – на варіант FALSE. В результаті на кожен рядок вихідного коду потрібно 3-5 рядків тестового коду;

– суворе дотримання технологій тестування. Необхідно зберігати не тільки записи про всі проведених тестах, але і про всі зміни вихідного коду у всіх модулях. З цією метою слід використовувати систему контролю версій ПЗ. Таким чином, якщо пізніша версія ПЗ не проходить тест, який був успішно пройдений раніше, буде нескладним звірити вихідний код

варіантів і усунути помилку. Також необхідно переконатися в незмінному відстеження та аналізу невдалих тестів. Ігнорування цієї вимоги призведе до лавиноподібного збільшення невдалих тестових результатів.

Інструментарій для полегшення виконання модульного тестування розроблений для широкого спектра мов. У мовах D і Cobra модульне тестування інтегровано в саму граматику мови і доступно без використання додаткових бібліотек [4, 5].

Для Java:

– JUnit;

– TestNG;

– JavaTESK.

NUnit – для мов платформи .NET: C #, Visual Basic .NET і ін.

Для C:

– CUnit cunit;

– CTESK;

– cfix.

API Sanity Autotest – для динамічних C / C ++ бібліотек в Unix-подібних ОС.

Для Objective-C – OCUnit.

Для C++:

– CPPUnit;

– Boost Test;

– Google C++ Testing Framework.

Symbian – фреймворк для Symbian OS всіх версій.

API Sanity Autotest – для динамічних C / C ++ бібліотек в Unix-подібних ОС.

DUnit – для Delphi.

Для Perl:

– Test;

– Test Simple;

– Test Unit;

– Test Unit Lite.

Для PHP:

– SimpleTest;

– PHPUnit.

Для Python:

– PyUnit;

– yTest;

– Nose.

Для T-SQL:

– TSQLUnit;

– SPUnit.

Для ActionScript 2.0 – мова сценаріїв, який використовується віртуальною машиною Adobe Flash Player версії 7 і 8 (AsUnit, AS2Unit).

Для ActionScript 3.0 – скриптова мова, яка використовується віртуальною машиною Adobe Flash Player версії 9(FlexUnit, AsUnit).

Test Unit – для Ruby. JsUnit – для JavaScript.

1.2 Тестування програм через графічний інтерфейс

Популярність такого виду тестування пояснюється двома факторами: по-перше, застосунок тестується тим самим способом, яким його використовуватиме людина, по-друге, можна тестуватизастоснок, не маючи при цьому доступу до вихідного коду [5].

UI-автоматизація розвивалася протягом 4 поколінь інструментів та технік:

– утиліти запису та відтворення (capture/playback tools) записують дії тестувальника під час ручного тестування. Вони дозволяють виконувати тести без прямої участі людини протягом тривалого часу, значно

збільшуючи продуктивність та усуваючи повторення одноманітних дій під час ручного тестування. У той же час, будь-яка мала зміна програмного забезпечення, що тестується, вимагає перезапису ручних тестів. Тому це перше покоління інструментів не є ефективним і не масштабованим;

– сценарії (Scripting) – форма програмування мовами, спеціально розроблених для автоматизації тестування ПЗ – пом'якшує багато проблем capture/playback tools. Але розробкою займаються програмісти високого рівня, які працюють окремо від тестувальників, які безпосередньо запускають тести. До того ж, скрипти найбільше підходять для тестування GUI і не можуть бути впровадженими, пакетними або взагалі якимось чином об'єднані в систему. Нарешті, зміни в ПЗ вимагають складних змін у відповідних скриптах, і підтримка все зростаючої бібліотеки тестуючих скриптів стає врешті-решт непереборним завданням;

– data-driven testing – методологія, яка використовується для автоматизації тестування. Особливістю є те, що тестові скрипти виконуються та верифікуються на основі даних, що зберігаються в центральному сховищі даних або БД. Роль БД можуть виконувати ODBC-ресурси, CSV або xls файли і т.д. Data-driven testing – це поєднання кількох взаємодіючих тестових скриптів та їх джерел даних у фреймворк, що використовується в методології. У цьому фреймворку змінні використовуються як для вхідних значень, так і для вихідних значень перевірок: у тестовому скрипті зазвичай закодовані навігація за застосунком, читання джерел даних, ведення логів тестування. Таким чином, логіка, яка буде виконана у скрипті, також залежить від даних;

– keyword-based автоматизація передбачає поділ процесу створення тесових випадків на 2 етапи: етап планування та етап реалізації [6].

Популярні програми для такого виду тестування включають:

Комерційні:

– HP LoadRunner, HP QuickTest Professional, HP Quality Center;

– Segue SilkPerformer;

– IBM Rational FunctionalTester, IBM Rational PerformanceTester, IBM Rational TestStudio;

– AutomatedQA TestComplete.

З відкритим вихідним кодом:

– Selenium;

– WATIR.

Також варто виділити навантажувальне тестування, яке теж піддається автоматизації. Навантажувальне тестування (Load Testing) або тестування продуктивності (Performance Testing) – це автоматизоване тестування, що імітує роботу певної кількості бізнес-користувачів на якомусь загальному (розділеному ними) ресурсі [6-8].

Основними цілями навантажувального тестування є:

– оцінка продуктивності та працездатності застосунка на етапі розробки та передачі в експлуатацію;

– оцінка продуктивності та працездатності програми на етапі випуску нових релізів, патч-сетів;

– оптимізація продуктивності програми, включаючи налаштування серверів та оптимізацію коду;

– підбір відповідної для цієї програми апаратної (програмної платформи) та конфігурації сервера.

Зауважимо, що в рамках однієї мети можуть використовуватися різні види тестів продуктивності та навантаження, наприклад, для першої, другої та третьої мети потрібно проводити тестування продуктивності так і тестування стабільності. Але при плануванні навантажувального тестування логічніше все ж таки відштовхуватися від технічних цілей (а не комерційних, перерахованих вище), які досягаються в результаті тестування та класифікувати тести з них:

– якщо цікавить дослідження продуктивності програми, а саме часи відгуку для операцій на різних навантаженнях в досить широких діапазонах,

включаючи стресові навантаження, то це все-таки тестування продуктивності (Performance Testing);

– якщо метою є розуміння наскільки програма стійка в режимі тривалого використання (виключення витоків пам’яті, некоректних конфігураційних налаштувань і т.д.), то проводиться довгий тест навантаження – це тестування стабільності (Stability Testing). При цьому аналіз часів відгуку може мати місце, але не бути першим пріоритетом, головне, щоб система «не впала»;

– стрес тестування (Stress Testing) має на меті перевірити чи повертається система після позамежного навантаження (і коли) до нормального режиму, а також цілями стресового тестування можуть бути перевірки поведінки системи у випадках коли, один із серверів програми в пулі перестає працювати, аварійно змінилася апаратна конфігурації сервера бази даних тощо. Зазначимо також, що при стресовому тестуванні перевіряється не продуктивність системи, а її здатність до регенерації після понаднавантаження [8].

Комерційні інструменти для автоматизованого тестування навантаження:

– Hewlett-Packard (Mercury Interactive);

– HP Performance Center (включає HP LoadRunner);

– IBM Rational;

– Rational Performance Tester;

– Borland (Segue);

– Borland (Segue);

– AutomatedQA Corp;

– TestComplete;

– Microsoft;

– MS Web Application Stress Tool.

Окремим пунктом виділимо безкоштовні інструменти для автоматизованого тестування навантаження:

– Jmeter;

– Grinder.

У цілому нині можна скласти зведену таблицю 1.1 найпопулярніших засобів автоматизації.

Таблиця 1.1 – Найпопулярніші засоби автоматизації в IT компаніях

Розробник	Функціональне	Навантажувальне	Якість коду	Керування тестами
IBM	+	+	+	+
Borland	+	+	–	+
AutomatedQA	+	+	–	+
HP	+	+	+	+
Open-Source	Abbot, Selenium, Watir	Grinder, Jmeter, OpenSTA	GCT, NCover, Cobertura	FitNesse, TestLink

1.3 Рекомендації для ефективної автоматизації тестування

Рекомендації для ефективної автоматизації тестування:

– обсяг автоматизації необхідно детально визначити на початок проєкту. Це дозволить переконатися, що очікування від автоматизації буде виправдано;

– визначте правильний інструмент автоматизації: інструмент не повинен вибиратися на основі його популярності, він повинен відповідати вимогам автоматизації на конкретному проєкті;

– виберіть потрібний фреймворк;

– стандарти створення сценаріїв. При написанні сценаріїв для автоматизації необхідно дотримуватись стандартів. Ось деякі з них: створіть єдині скрипти, коментарі та відступи коду; розробіть правила найменування

тестових сценаріїв; прикладайте необхідні документи, якщо, наприклад, складно зрозуміти проходження тестового сценарію без скріншота та/або специфікації;

– визначте метрики та стежте за ними. Успіх автоматизації не можна визначити лише шляхом порівняння витрачених зусиль, на той чи інший вид тестування. Ось основні показники: відсоток виявлених дефектів; час, необхідне тестування автоматизації випуску кожного нового циклу; мінімальний час, необхідний для випуску; індекс задоволеності клієнтів; покращення продуктивності.

Наведені вище рекомендації, якщо їх дотримуватись, дозволять якісно виконати автоматизацію тестування [9].

1.4 Рекомендації до вибору інструмента автоматизації

Вибір відповідного інструменту може виявитися складним завданням. Наступні критерії допоможуть вам вибрати найкращий інструмент для ваших вимог:

– підтримка довкілля;

– легкість використання;

– тестування бази даних;

– ідентифікація об’єкта;

– тестування зображень;

– тестування відновлення після помилок;

– відображення об’єктів;

– використовується мова сценаріїв;

– підтримка різних типів тестування, у тому числі функціонального, тестового керування, мобільного тощо;

– підтримка кількох фреймворків тестування;

– легко налагоджувати сценарії програмного забезпечення автоматизації;

– вміння розпізнавати предмети у будь-якому середовищі;

– великі звіти про випробування та його результаты;

– мінімізація витрат на навчання вибраним інструментам.

Вибір інструменту – одна з найсерйозніших проблем, яку необхідно вирішити, перш ніж приступати безпосередньо до автоматизації.

Спершу визначте вимоги, вивчіть різні інструменти та їх можливості, встановіть очікування від інструменту та зробіть Proof Of Concept (підтвердження своєї ідеї) [9].

1.5 Детальніше про найпопулярніші інструменти автоматизації тестування

Selenium – це середовище тестування для тестування вебзастосунку у різних браузерах та платформах, таких як Windows, Mac та Linux. Selenium допомагає тестувальникам писати тести різними мовами програмування, таких як Java, PHP, C#, Python, Groovy, Ruby та Perl. Selenium пропонує функції запису та відтворення для написання тестів без вивчення Selenium IDE.

TestingWhiz це інструмент автоматизації тестування зі сценаріями без коду від Cygnet Infotech, постачальника IT рішень 3-го рівня CMMi. Редакція Enterprise інструменту TestingWhiz пропонує повний пакет різних рішень для автоматизованого тестування, таких як вебтестування, тестування програмного забезпечення, тестування баз даних, тестування API, тестування мобільних застосунків, обслуговування набору регресійних тестів, оптимізація та автоматизація та міжбраузерне тестування [10].

TestingWhiz пропонує різні функції, такі як:

– тестування на основі ключових слів, даних розподіленого тестування;

– тестування розширення браузера;

– object Eye Внутрішній рекордер;

– SMTP інтеграція;

– інтеграція з інструментами відстеження помилок, такими як Jira, Mantis, TFS та FogBugz;

– централізоване сховище об’єктів;

– інтеграція системи контролю версій;

– індивідуальне правило запису.

HP QuickTest Professional було перейменовано на HPE Unified Functional Testing. HPE UFT пропонує автоматизацію тестування для функціонального та регресійного тестування для програмних програм.

Мова сценаріїв Visual Basic Scripting Edition використовується цим інструментом для реєстрації процесів тестування та керування різними об’єктами та елементами керування під час тестування програм.

QTP пропонує різні функції, такі як:

– інтеграція з Mercury Business Process Testing та Mercury Quality Center;

– унікальне розпізнавання смарт-об’єктів;

– механізм обробки помилок;

– створення параметрів для об’єктів, контрольних точок та таблиць, керованих даними;

– автоматизована документація.

TestComplete – це функціональна платформа тестування, яка пропонує різні рішення для автоматизації тестування настільних, мобільних програм компанією SmartBear Software [10].

TestComplete пропонує такі функції:

– тестування GUI;

– підтримка мови сценаріїв – JavaScript, Python, VBScript, JScript, DelphiScript, C++ Script та C# Script;

– тестовий візуалізатор;

– скриптове тестування;

– тестовий запис та відтворення.

Ranorex Studio пропонує інструменти автоматизації тестування, які охоплюють тестування всіх десктопних та мобільних програм.

Ranorex пропонує такі функції:

– розпізнавання графічного інтерфейсу користувача;

– багаторазові тестові коди;

– виявлення помилок;

– інтеграція з різними інструментами;

– запис та відтворення.

Sahi – інструмент для автоматизації тестування вебзастосунків. Sahi з відкритим вихідним кодом написаний мовами програмування Java та JavaScript [10].

Sahi надає такі можливості:

– проводить мультибраузерне тестування;

– підтримує ExtJS, ZK, Dojo, YUI та ін.;

– запис та відтворення на тестуванні браузера.

Watir – це інструмент тестування з відкритим вихідним кодом, що складається з бібліотек Ruby, для автоматизації тестування вебзастосунків. Це вимовляється як «вода» [11].

Watir пропонує такі функції:

– тестує мовну вебзастосунок;

– крос-браузерне тестування;

– сумісний з бізнес-інструментами розробки, такими як RSpec, Cucumber та Test / Unit;

– перевіряє кнопки, форми, посилання та їх відповіді на вебсторінках.

Katalon Studio – це безкоштовне рішення для автоматизації тестування, розроблене компанією Katalon LLC. Програмне забезпечення побудовано на основі середовищ автоматизації з відкритим кодом Selenium, Appium зі спеціалізованим інтерфейсом IDE для тестування API, вебзастосунків і мобільних пристроїв. Цей інструмент включає повний пакет потужних функцій, які допомагають подолати загальні проблеми в автоматизації тестування вебінтерфейсу [11].

Katalon Studio складається з наступних функцій:

– вбудований репозиторій об’єктів, XPath, повторна ідентифікація об’єкта;

– підтримує мови сценаріїв Java/Groovy;

– вбудована підтримка тестування на основі зображень;

– підтримка інструментів безперервної інтеграції, таких як Jenkins та TeamCity;

– підтримує інтерфейс Duel-редактора;

– робочий процес виконання, що налаштовується.

1.6 Постановка задачі дослідження

Робота присвячена дослідженню важливої галузі у сучасній інформатиці – автоматизоване тестування програмного забезпечення. Даний вид діяльності активно застосовується у процесі життєвого циклу розробки ПЗ та безпосередньо впливає на забезпечення якості ПЗ. Тому вивчення проблем автоматизації тестування та розробка інструментарію для його проведення є актуальним завданням.

Об’єктом дослідження є забезпечення якості та надійності програмного забезпечення.

Метою дослідження є вивчити засоби автоматизованого тестування для оцінювання якості програмного продукту.

Предмет дослідження: інструмент – фреймворк для тестування, що служить для забезпечення зручності розробки автоматизованих тестів для широкого кола фахівців відділів/управлінь тестування.

Для досягнення мети необхідно вирішити такі завдання:

– провести аналіз існуючих методологій тестування;

– провести аналіз існуючих видів тестування програмного продукту;

– розробити мануальні та автоматизовані тести.

2 ОРГАНІЗАЦІЯ ТЕСТУВАННЯ ПРОГРАМНОГО ПРОДУКТУ

2.1 Процеси тестування

При впровадженні автоматизованого тестування у розробку, щоб уникнути багатьох частих проблем пов'язаних з цим процесом, рекомендується використовувати методологію життєвого циклу автоматизованого тестування (ATLM) [12].

При використанні такого структурного підходу організації можуть виконувати дії, пов'язані з тестуванням з максимальним можливим у межах ресурсів покриттям. Цей підхід використовується паралельно до життєвого циклу розроблюваної системи.

Цей підхід складається з шести основних процесів або компонентів. Кожен із них ділиться на підпроцеси, як показано нижче.

Перша фаза ATLM: рішення автоматизувати тестування. Під час цієї фази відповідальні за тестування мають визначити свої очікування від автоматизації тестування. Виділити потенційну користь при правильному впровадженні. Також необхідно висунути рішення про інструменти, які передбачається використати [12].

Автоматизована генерація тестових планів. На даний момент в продажу не існує інструмента, що забезпечує автоматичне створення тестових планів, водночас проєктування та виконання тестів. Часто буває так, що старший менеджер проєкту міг почути переваги автоматизованого тестування та попросив одного з інженерів підготувати огляд існуючих коштів, які можуть у цьому допомогти. Але він може сформувати помилкове враження про можливості таких інструментів. Важливо дати зрозуміти, що для досягнення автоматизації все ще потрібні кваліфіковані фахівці в цій галузі, оскільки інструмент не може замінити людину. Він може розглядатись як додаткова частина, що підтримує випуск хорошого продукту.

Миттєве скорочення термінів тестування. Як було сказано вище, насправді час, що витрачається на тестування, зросте через те, що доведеться або доповнювати існуючий процес тестування, або розробити і впровадити новий. Вся команда тестувальників та розробників повинна ознайомитися з новими процесами тестування та дотримуватися їх. Після закінчення цього періоду очікується зросла продуктивність і зниження витрат за тестування [12, 13].

Переваги автоматизованого тестування. При правильному впровадженні процесів автоматизованого тестування можна виділити такі переваги: підвищена надійність продукту, підвищення тестового покриття, зниження тимчасових витрат на тестування.

Отримання інструменту для автоматизованого тестування. Інженер вибирає інструменти, що найбільш підходять під оточення, становить критерії оцінки інструменту. Визначається домен оцінювання, на якому проведеться пілотний запуск інструменту. Після цих кроків інструмент поставляється і вся команда тестувальників оцінює його за складеними критеріями.

Друга фаза ATLM: застосування автоматизованого тестування.

Аналіз процесу тестування Проведення аналізу процесу тестування гарантує, що весь процес і стратегія тестування визначені і при необхідності можуть бути модифіковані з цілого успішного впровадження автоматизованого тестування. Тестувальник визначає та збирає оцінки процесу тестування, що дозволить удосконалити процес. Повинні бути встановлені цілі, об’єкти та стратегії тестування, а процес тестування має бути документований та доведений до відома команди тестувальників. На цій фазі визначаються види тестування, які застосовуються до технічного середовища, а також тести, що підтримуються автоматизованими засобами. Оцінюються плани підключення користувачів, та аналізуються навички тестувальників на предмет відповідності вимогам та запланованим роботам із тестування. Раннє участь команди тестувальників особливо важливе,

оскільки дозволяє оновити специфікації вимог і висловити їх у термінах, які можуть бути адекватно протестовані, а також дає команді більш повне уявлення про вимоги та архітектуру програми [12-14].

Третя фаза ATLM: розгляд інструментального засобу тестування На фазі розгляду інструментального засобу тестування тестувальник визначає, чи буде корисним для проєкту включення засобів автоматизованого тестування до роботи з тестування. При цьому враховуються вимоги до тестування в проєкті, існуюче середовище тестування та людські ресурси, середовище користувача, платформа та функції програми, що підлягають тестуванню. Перевіряється план-графік проєкту на наявність достатнього часу для встановлення засобів тестування та розробки ієрархії вимог. Потенційні засоби тестування порівнюються з вимогами до тестування. Перевіряється сумісність засобів тестування з застосунком та середовищем. Досліджуються альтернативні рішення, що дозволяють подолати проблеми, виявлені під час перевірки сумісності [14].

Четверта фаза ATLM: планування, проєктування та розробка тестування.

Планування тестування. Фаза планування тестування включає огляд довгострокових робіт з планування тестування. На цьому етапі команда тестувальників визначає стандарти та основні напрямки створення процедур тестування; апаратні, програмні та мережеві засоби, необхідні для підтримки тестового середовища; вимоги до даних для тестування; попередній план-графік тестування; вимоги щодо оцінки продуктивності; процедуру управління конфігурацією та середовищем тестування; процедуру відстеження дефектів та засоби для його проведення.

До плану тестування входять результати всіх попередніх фаз структурованої методології тестування (ATLM). Він визначає ролі та обов’язки, план-графік тестування в проєкті, роботи з планування та проєктування тестування, підготовку тестового середовища, ризики та непередбачувані обставини при тестуванні, прийнятний рівень якості

продукту (тобто критерій завершення тестування). Застосунки до плану тестування можуть включати процедури тестування, опис угод з іменування, стандарти формату процедур тестування і матрицю залежності процедури тестування [15].

Установка середовища тестування – це частина планування тестування. Команда тестувальників повинна планувати, відстежувати та керувати роботами із встановлення середовища тестування, що може зайняти чимало часу. Тестувальники повинні скласти план-графік і керувати роботою по встановленню середовища; встановити апаратні, програмні та мережеві ресурси середовища тестування; інтегрувати та встановити ресурси середовища тестування; отримати та оновити тестові бази даних; розробити скрипти для встановлення середовища та скрипти для тестового стенду.

Проєктування тестування. На цій фазі визначаються кількість тестів, які потрібно виконати, способи, за допомогою яких можна отримати доступ до тесту (наприклад, шляхи або функції) та умови тестування, які повинні дотримуватися. Необхідно визначити та дотримуватися стандартів проєктування тестування [15-17].

Розробка тестування. Щоб автоматизоване тестування можна було повторно використовувати, повторювати і супроводжувати, необхідно визначити і дотримуватися стандартів розробки тестування.

П’ята фаза ATLM: виконання тестів та управління тестами.

Команда тестувальників зобов’язана виконувати скрипти тестування та вдосконалювати скрипти комплексного тестування відповідно до графіка виконання процедури тестування. Крім того, вона має оцінити результати виконання тестування так, щоб уникнути неправильних позитивних чи негативних оцінок. Системні проблеми документуються у звітах про системні проблеми. Необхідно домогтися розуміння з боку розробника системних проблем, програмного забезпечення, а також реплікації цієї

проблеми. І, нарешті, команда має виконати регресійне тестування та всі інші тести та завершити відстеження проблем.

Шоста фаза ATLM: Огляд та оцінка програми тестування

Робота з критичного перегляду, або інспекції та оцінки повинна проводитись протягом усього життєвого циклу тестування. Протягом усього життєвого циклу тестування та наступних робіт з виконання тестування повинні оцінюватись вимірювання та проводитись роботи з остаточного критичного перегляду та оцінки, що дозволить удосконалити процес [18].

2.2 Методології тестування

Вибір моделі розробки ПЗ серйозно впливає на процес тестування, визначаючи вибір стратегії, розклад, потрібні ресурси тощо. Моделей розробки програмного забезпечення багато, але в загальному випадку класичними можна вважати водопадну, v-подібну, ітераційну інкрементальну, спіральну та гнучку [18].

Розуміти і знати моделі розробки ПЗ потрібно для того, щоб розуміти що відбувається на проєкті з початку роботи в компанії. Чим краще людина розуміє, що відбувається на проєкті, тим ясніше буде видно внесок однієї людини у загальну роботу та сенс її роботи.

2.2.1 Каскадна модель

Каскадна модель (англ. Waterfall model, іноді перекладають як модель «Водоспад») – модель процесу розробки програмного забезпечення, в якій процес розробки виглядає як потік, що послідовно проходить фази аналізу вимог, проєктування, реалізації, тестування, інтеграції та підтримки [19] (рис. 2.1).

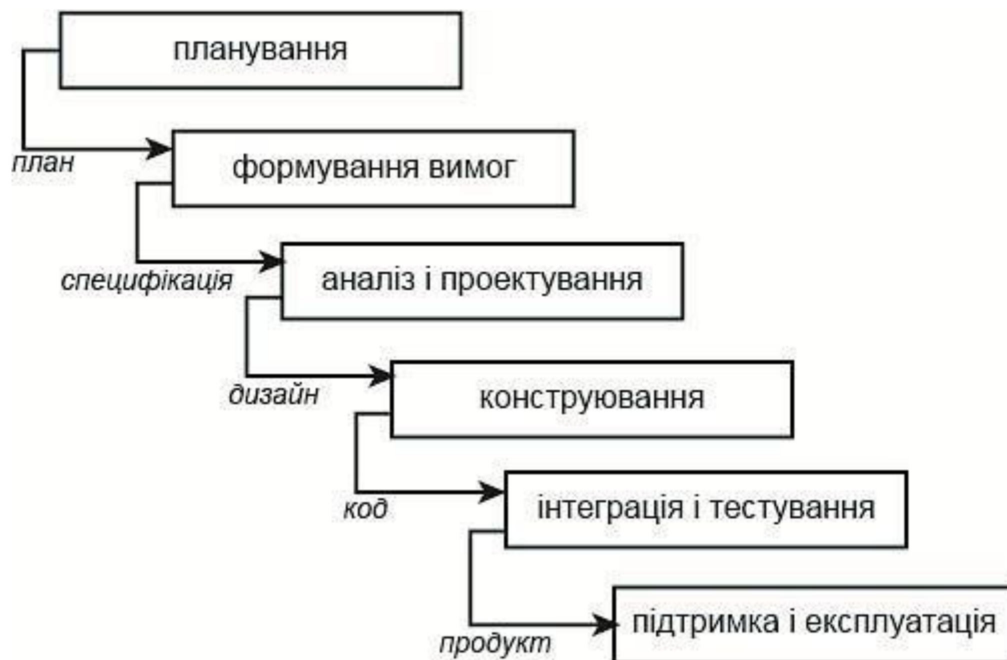

Рисунок 2.1 – Каскадна модель процесу розробки програмного забезпечення

До недоліків водоспадної моделі прийнято відносити той факт, що участь користувачів ПЗ у ній або не передбачено взагалі, або передбачено лише побічно на стадії одноразового збору вимог. З точки зору тестування ця модель погана тим, що тестування в явному вигляді з’являється тут лише з середини розвитку проєкту, досягаючи свого максимуму наприкінці.

2.2.2 V-подібна модель

V-подібна модель (V-model) є логічним розвитком водоспаду. Можна помітити (рис. 2.2.), що у випадку як водоспадна, і v подібна моделі життєвого циклу ПЗ можуть містити той самий набір стадій, але принципова відмінність у тому, як ця інформація використовується у процесі реалізації проєкту.

Дуже спрощено можна сказати, що при використанні v-подібної моделі на кожній стадії «на узвозі» потрібно думати про те, що і як відбуватиметься відповідної стадії «на підйомі». Тестування тут з’являється вже на ранніх стадіях розвитку проєкту, що дозволяє мінімізувати ризики, а

також виявити та усунути безліч потенційних проблем до того, як вони стануть реальними проблемами [19].

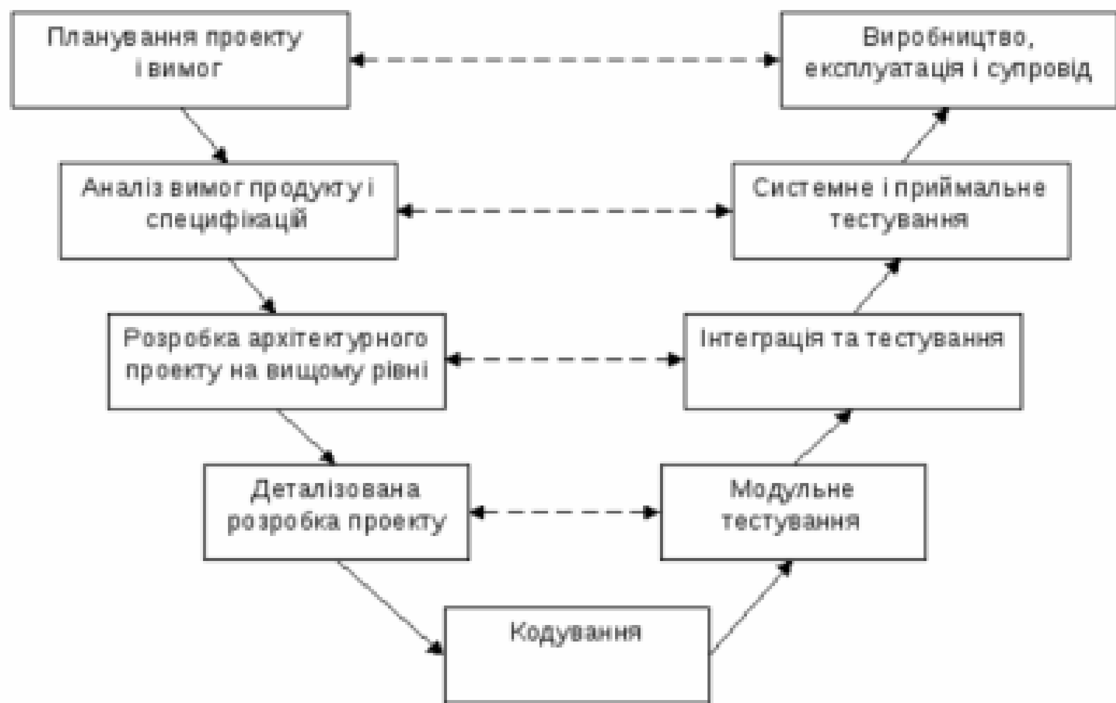

Рисунок 2.2 – V-подібна модель розробки ПЗ

2.2.3 Ітераційна інкрементальна модель

Ітераційна інкрементальна модель (iterative model), incremental model) є фундаментальною основою сучасного підходу до розробки ПЗ. Як випливає з назви моделі, їй властива певна двоїстість (а ISTQB-глосарій навіть не наводить єдиного визначення, розбиваючи його на окремі частини):

– з погляду життєвого циклу модель є ітераційною, т.к. під розуміє багаторазове повторення тих самих стадій;

– з точки зору розвитку продукту (прирощення його корисних функцій) модель є інкрементальною.

Ключовою особливістю даної моделі є розбиття проєкту на відносно невеликі проміжки (ітерації), кожен із яких у випадку може включати всі

класичні стадії, властиві водоспадній і v-подібної моделям (рис. 2.3). Підсумком ітерації є збільшення (інкремент) функціональності продукту, виражене у проміжному білді (build) [19, 20].

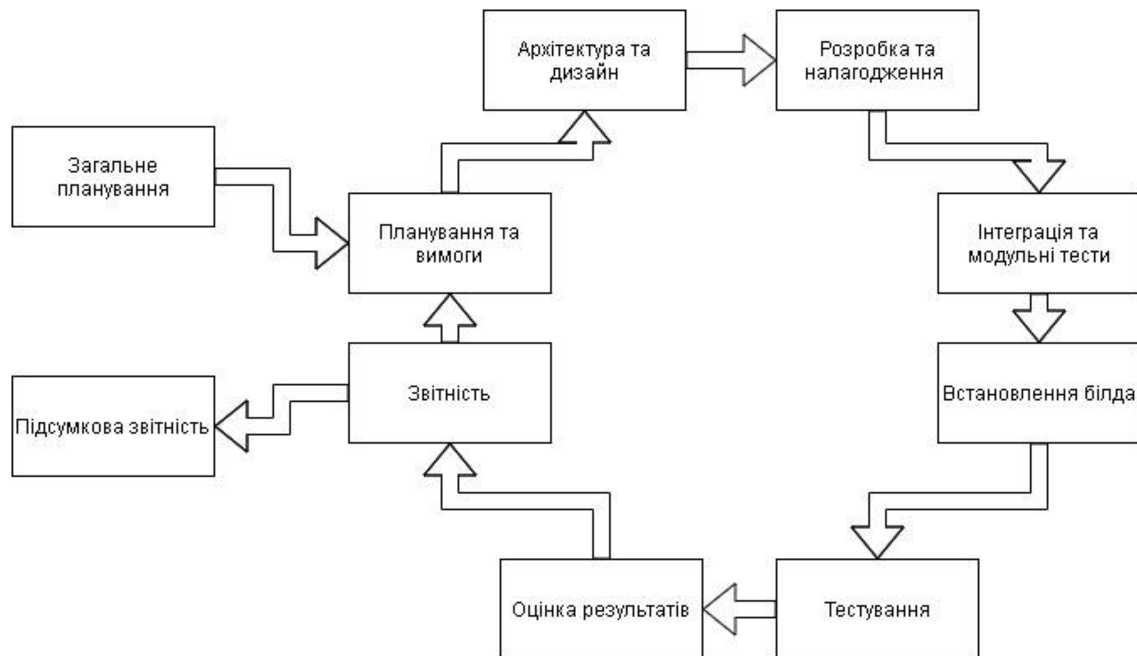

Рисунок 2.3 – Ітераційна інкрементальна модель розробки ПЗ

Довжина ітерацій може змінюватися в залежності від безлічі факторів, однак сам принцип багаторазового повторення дозволяє гарантувати, що і тестування, і демонстрація продукту кінцевому замовнику (з отриманням зворотної активно застосовуватиметься з самого початку і протягом всього часу розробки проєкту.

2.2.4 Спіральна модель

Спіральна модель (spiral model) є окремим випадком ітераційної інкрементальної моделі, в якій особлива увага приділяється управлінню ризиками, що особливо впливають на організацію процесу розробки проєкту та контрольні точки.

Схематично суть спіральної моделі представлена рис. 2.4. Зверніть увагу на те, що тут явно виділено чотири ключові фази:

– опрацювання цілей, альтернатив та обмежень;

– аналіз ризиків та прототипування;

– розробка (проміжної версії) продукту;

– планування наступного циклу.

З точки зору тестування та управління якістю ризикам є відчутною перевагою при використанні спіральної моделі для розробки концептуальних проєктів, в яких вимоги є природним чином складними і нестабільними (можуть багаторазово змінюватися під час виконання проєкту) [21].

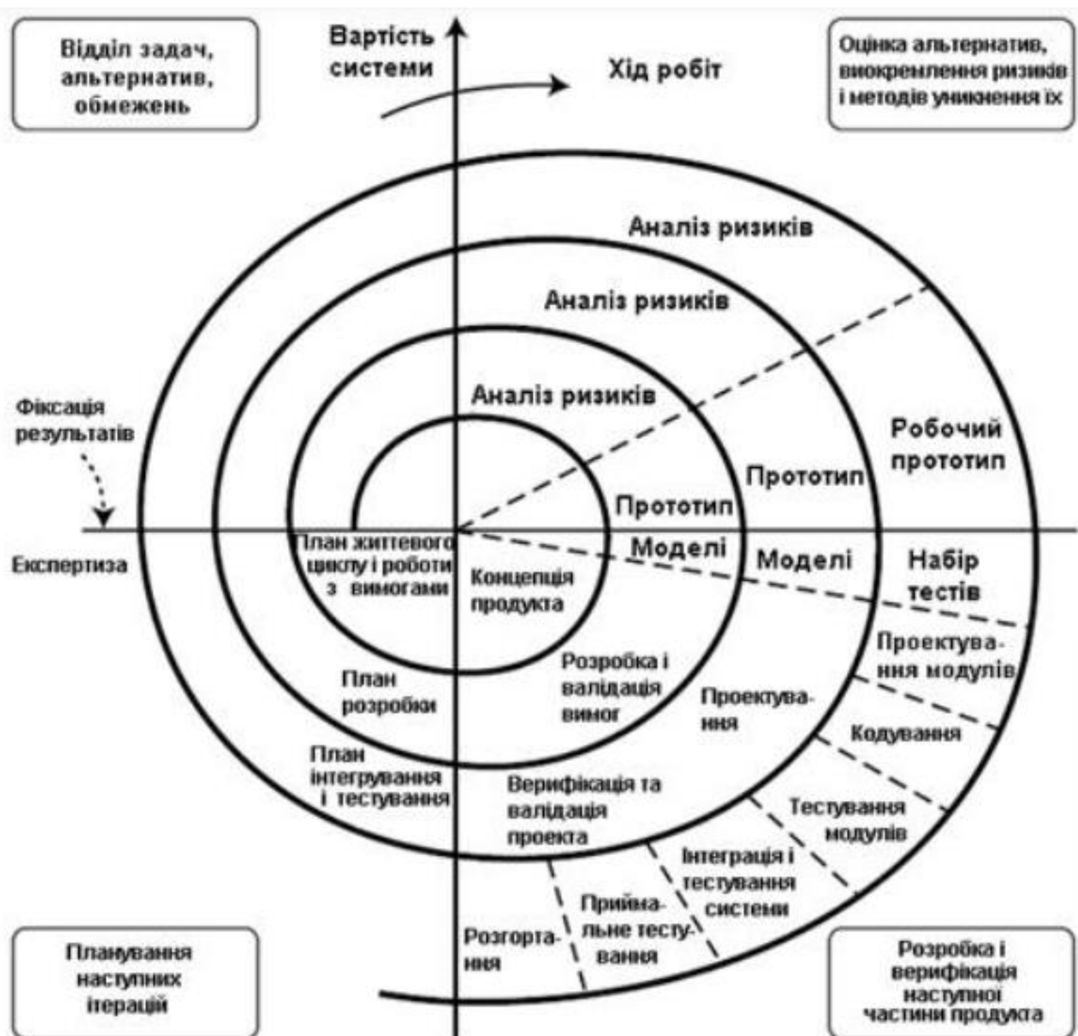

Рисунок 2.4 – Спіральна модель розробки ПЗ

2.2.5 Гнучка модель

Гнучка модель (agile model) є сукупністю різних підходів до розробки ПЗ та базується на т.зв. «agile-маніфесті»:

– люди та взаємодія важливіші за процеси та інструменти;

– працюючий продукт важливіший за вичерпну документацію;

– співпраця із замовником важливіша за погодження умов контракту;

– готовність до змін важливіша за дотримання початкового плану.

Як нескладно здогадатися, покладені в основу гнучкої моделі підходи є логічним розвитком і продовженням всього того, що було за десятиліття створено та випробувано у водопадній, v-подібній, ітераційній інкрементальній, спіральні та інші моделі. Причому тут вперше було досягнуто відчутного результату в зниженні бюрократичної складової та максимальної адаптації процесу розробки ПЗ до миттєвих змін ринку та вимог замовника [21, 22] (рис. 2.5).

Рисунок 2.5 – Гнучка модель розробки ПЗ

Головним недоліком гнучкої моделі вважається складність її застосування до великих проєктів, і навіть часте хибне використання її підходів, викликане нерозуміння фундаментальних принципів моделі.

Проте можна стверджувати, що дедалі більше проєктів починають використовувати гнучку модель розробки.

Наприклад: я працюю автоматизатором на проєкті. В нас є вебверсія, а також mobile (android iOS) версії застосунку. Цей застосунок потрібен для того, щоб користувачі мали змогу створити свій застосунок на сайті та використовувати його далі. Працюємо по гнучкій(SCRUM) методології. Спринти в нас по два тижні. Також маємо обов'язкові дзвінки, які називаються scrum meeting. Вони бувають різні та направлені на різні цілі.

Та перед цим потрібно дати визначення такого слова, як спринт [23].

Спринт – проміжок часу, достатній до виконання запланованої сукупності операцій SCRUM, метою якої є створення інкременту (інкремент продукту є готовою до використання частиною продукту, яка повинна бути реалізована до завершення спринту) бізнес-продукту. Жорстко фіксований за часом. Тривалість одного спринту від 1 до 4 тижнів. Чим коротшим є спринт, тим гнучкішим є процес розробки, релізи виходять частіше, швидше надходять відгуки від споживача, менше часу витрачається на роботу в неправильному напрямку, але багато часу витрачається на мітинги планування спринту, ретроспективи. З іншого боку, при більш тривалих спринтах команда (SCRUM Team) зменшує витрати на наради, демонстрації товару тощо. Тож які дзвінки в нас є теперь можно навести [24].

Планування спринта (Sprint Planning Meeting). Проходить на початку нової ітерації спринта:

– із Product Backlog обираються задачі, зобов'язання по виконанню яких за спринт бере на себе команда;

– на основі обраних задач створюється Sprint Backlog. Кожна задача оцінюється у ідеальних людино-годинах;

– розв’язання задачі не повинне займати більше 12 годин або одного дня. За необхідності задачу розбивають на підзадачі;

– обговорюється та визначається, яким чином буде реалізовано цей об’єм робіт;

– тривалість наради обмежена зверху 4-8 годинами в залежності від тривалості ітерації, досвіду команди тощо;

– (перша частина наради) Беруть участь Product Owner + Команда: обирають задачі із Product Backlog;

– (друга частина наради) Бере участь лише команда: обговорюють технічні деталі реалізації, наповнюють Sprint Backlog.

Щоденна нарада (Daily Scrum Meeting). Відбувається кожен день протягом спринта. Є «пульсом» ходу спринта. Нараді властиві наступні обмеження:

– починається точно вчасно;

– всі можуть спостерігати, але говорять тільки по черзі;

– триває не більш ніж 15 хвилин;

– проводиться в одному і тому ж місці протягом одного спринта.

Протягом наради кожен член команди відповідає на 3 запитання:

«Що зроблено з моменту попередньої щоденної наради?»

«Що буде зроблено з моменту поточної наради до наступної?»

«Які проблеми заважають досягненню цілей спринта?» (Над рішенням цих проблем працює ScrumMaster. Зазвичай це рішення проходить за рамками щоденної наради і у складі осіб, що безпосередньо займаються даною перешкодою) [25].

Демонстрація (Sprint Review Meeting):

– проходить у кінці ітерації (спринта);

– команда демонструє внесок функціональності до продукту всім зацікавленим особам;

– залучається максимальна кількість глядачів;

– усі члени команди беруть участь у демонстрації (одна людина на демонстрацію або кожен показує, що зробив за спринт);

– обмежена 4-ма годинами в залежності від тривалості ітерації і змін у продукті.

Ретроспектива (Sprint Retrospective). Це один із заходів скраму, що дає скрам-команді можливість провести інспекцію своєї роботи та створити план покращень на наступний спринт. Ретроспектива проходить після огляду спринту, перед плануванням наступного спринту. У ньому бере участь вся скрам-команда [26]. У ході ретроспективи:

– члени команди висловлюють свою думку про минулий спринт;

– виконують покращення процесу розробки (вирішують питання та фіксують вдалі рішення);

– обмежена 1-3ма годинами.

А також відповідають на два основних запитання:

«Що було зроблено добре у минулому спринт?»

«Що потрібно покращити в наступному?»

2.2.6 Аналіз якості програмного забезпечення

Задоволенню вимогам замовника сприяє створення якісного продукту.

Терміном якість програмного забезпечення позначаємо сукупність властивостей, що визначають спроможність забезпечення задовольнити запити замовника, які він висловив як вимоги до розробки.

Якість програмного забезпечення – це відносне поняття, що має сенс тільки при врахуванні реальних умов його застосування, тому вимоги, які пред’являються до якості програмного забезпечення, ставляться відповідно до умов і конкретної галузі їхнього застосування [27].

Якщо провести моніторингове опитування серед користувачів програмного забезпечення по питанню якості програмних продуктів, то дуже часто звучатимуть відповіді:

– програмне забезпечення легко використовується;

– у програмному забезпеченні реалізовано багато можливостей;

– практично немає помилок;

– захищає інформацію користувача;

– можливість використання на різних платформах;

– працює безвідмовно потрібний проміжок часу;

– має добре організовану довідкову службу;

– задовольняє потреби користувачів та ін.

Всі наведені відповіді дійсно мають відношення до поняття якості програмного забезпечення, але вказують на характеристики важливі для конкретного користувача, замовника або групи вказаних осіб [28].

Аналіз якості в програмній інженерії орієнтований на:

– досягнення необхідної якості програмного забезпечення відповідно до встановлених критеріїв;

– верифікацію і валідацію на етапах життєвого циклу та оцінку якості виробленого програмного продукту [28];

– забезпечення надійності як основної характеристики гарантії якості програмного забезпечення [29].

Визначені напрями програмної інженерії розглядаються на всіх етапах життєвого циклу програмного продукту, тобто аналіз і забезпечення якості проводиться за всіма видами діяльності у вирішенні задач планування, розробки і підтримки процесів створення програмного забезпечення.

Для створення якісного програмного продукту інженер програмного забезпечення повинен користуватися різноманітними технологіями та стандартами побудови якісного програмного забезпечення

Для проведення аналізу якості, а також верифікації і валідації програмного забезпечення програмісту потрібні знання про організацію

управління якістю, про процеси аналізу якості, метою яких є забезпечення додаткових гарантій досягнення заданої якості та її поліпшення. Слід звернути увагу, що під поняттям валідація розуміємо забезпечення відповідності розробки вимогам її замовників, а верифікація – це перевірка правильності трансформації проєкту в код реалізації.

Аналіз якості програмного забезпечення включає початкову оцінку, яка відповідає процесам виробництва якісного програмного продукту: моніторингу, плануванню, виконанню, зміні або підтримці.

Досягнення якості програмного забезпечення залежить від процесу проєктування, який повинен бути планованим і включати систематизований набір дій із забезпечення адекватності й довіри до продукту, створеного відповідно до поставлених технічних вимог. При цьому верифікація та валідація програмного забезпечення належать до управління якістю і є важливими процесами забезпечення якості програмного продукту на етапах його життєвого циклу. Життєвий цикл є моделлю створення і використання програмного забезпечення, яка відображає його різні стани, починаючи з моменту виникнення необхідності в даному програмному продукті і закінчуючи моментом його повного виходу із вжитку у всіх користувачів. Існує багато моделей життєвого циклу програмного забезпечення, але в міжнародних стандартах як фундаментальні класифікуються три з них [29]: каскадна (водоспадна), інкрементна (поетапна), еволюційна (спіральна). Якість програмного забезпечення визначається вже на етапі проєктування спіральною моделлю життєвого циклу програмного забезпечення за допомогою розрахункових та експертних методів вимірювання.

Основні стандартні положення зі створення якісного продукту й оцінки рівня досягнутої якості зафіксовано в міжнародних [29] та вітчизняних стандартах [28-30]. Залежно від специфіки програмних продуктів стандарти пропонують термінологію та склад показників (атрибутів) якості. Вони утворюють базові знання і визначають планування, проєктування, аналіз, вимірювання та поліпшення якості.

Якість програмного забезпечення визначається набором загальних характеристик, на формулювання яких спрямовані процеси інженерії вимог як складової частини програмної інженерії. Визначення характеристик якості відображає погляд користувача на якість програмного забезпечення. Відповідно до стандарту ISO-9126 [30] визначено шість характеристик:

– функціональність;

– надійність;

– зручність;

– ефективність;

– супроводжуваність;

– переносність.

Функціональність – це сукупність властивостей, які визначають спроможність програмного забезпечення виконувати в заданому середовищі перелік функцій відповідно вимогам до обробки і загальносистемним засобам [30].

Надійність – це множина атрибутів, котрі вказують на спроможність програмного забезпечення перетворювати вхідні дані на результати при умовах, що залежать від періоду часу (зношення й старіння програмного забезпечення не враховуються) [30].

Зручність – це множина атрибутів, котрі вказують на необхідні або сприятливі умови для використання програмного забезпечення визначеним колом користувачів [30].

Ефективність – це множина атрибутів, які показують взаємозв’язок між рівнем виконання програмного забезпечення і кількістю використовуваних ресурсів у початкових умовах [30].

Супроводжуваність – це множина властивостей, які вказують на зусилля, котрі необхідно витратити на проведення модифікацій, таких як коригування, удосконалення й адаптацію програмного забезпечення зі зміною середовища, вимог або функціональних специфікацій [29, 30].

Переносність – це множина показників, котрі вказують на здатність програмного забезпечення пристосовуватися до роботи при зміні середовища виконання [30].

Для проведення оцінки якості програмного забезпечення попередньо визначаються базові значення показників для аналогу, прийнятого за еталон про розробці даного програмного забезпечення. Значення базових показників мають відбивати сучасний світовий рівень розробок. На роль аналогів вибирається реальне програмне забезпечення того самого функціонального призначення, що й порівнюване, з такими ж основними параметрами, що й дана структура, і схожими умовами експлуатації.

Таким чином, аналіз якості є діяльністю, що включає процеси управління, інфраструктуру програмної інженерії, тестування, інженерію вимог. Характеристики якості визначають споживчі властивості і мають вартісний вираз, що включає оцінку витрат на процес розробки та експлуатації, оцінку економічних вигод застосування вказаних програм порівняно з іншими засобами вирішення відповідного прикладного завдання, а також перспектив подальшого використання даного програмного забезпечення в умовах зміни середовища функціонування.

2.2.7 Вимоги до програмного застосунку

Ця група критеріїв дозволить оцінити, наскільки пропрацювали вимоги (user story) до ПЗ, визначити вразливі місця та найбільш складні, потенційно проблемні фічі ПЗ, зрозуміти, де потрібний особливий контроль.

User Story(вимоги) – це одне або кілька речень написаних повсякденною або бізнес-мовою користувача або клієнта, які описують те, чого користувач або клієнт хоче досягти за допомогою програмного забезпечення. Це короткий опис того, як користувач, клієнт або інша особа буде використовувати систему, і яку вигоду при цьому отримає від

конкретної функціональності. Таким чином, замість складних для розуміння клієнта Requirements Specifications, створюються короткі розповіді – User Stories – які забезпечують власнику продукту зрозумілий контекст і дозволяють ефективно керувати і приорітизувати список завдань [31].

Тестове покриття вимоги. Іншими словами, це кількість тестів на 1 вимогу.

Призначення критерію: виявити слабкі місця у тестовому покритті, підвести ризики.

Тестове покриття = (кількість вимог, покритих тест-кейсами/загальна кількість вимог) *100%.

Даний критерій буде працювати, тільки якщо вимоги добре декомпозовані і більш менш рівнозначні. Зрозуміло це завжди можливо, але якщо виходить зробити вимоги досить атомарними, то цей критерій покаже відхилення покриття кожної вимоги від середнього рівня. Чим більше значення відрізняється від 1, тим менше тестів написано для однієї вимоги, ніж зазвичай.

Щоб перевірити таке тестове покриття, вимоги необхідно розбити на пункти і кожен пункт зв’язати з тест-кейсами, які його перевіряють. У тест-кейсах немає необхідності, якщо вони не перевіряють будь-які вимоги в продукті. У свою чергу пункти вимог, для яких не написано жодного тест-кейсу, не будуть протестовані, а значить не можна буде з точністю сказати, чи реалізовані дані вимоги в продукті і як ці функції працюють [30, 31].

Всі зв’язки між тест-кейсами і пунктами вимог називаються матрицею трасування. Якщо простежити і проаналізувати ці зв’язки, можна зрозуміти, які тест-кейси перевіряють які вимоги і для яких вимог потрібно написати/збільшити кількість тест-кейсів. Для деяких же вимог, можливо, існують зайві тест-кейси.

Найважливіше звернути увагу на вимоги, для яких коефіцієнт дорівнюватиме або близький до 0. Для них потрібно розглянути можливість додавання тестів.

Якщо вимоги не атомарні, то цей критерій дозволить переконатись лише в тому, що для кожної вимоги є хоча б один тест. Для цього коефіцієнт завжди повинен бути більшим за 1.

Критерій обчислюється як середня кількість зв'язків кожної вимоги з рештою вимог.

Призначення критерію: дати підстави для оцінки термінів тестування та врахування можливих ризиків. Знаючи ступінь взаємного впливу вимог один на одного можна, наприклад, запланувати додатковий час та кейси для наскрізного тестування, опрацювати регресійні перевірки, подивитися у бік інтеграції тощо.

Тут складно вводити якісь обмеження для значень даного коефіцієнта, залежить від специфіки функціоналу, архітектури, технологій. Однак, за своїм досвідом можу сказати, що добре, коли рівень зв'язаності не перевищує 0,2-0,3. В іншому випадку доробка в рамках однієї з вимог вестиме до ланцюжка змін, а значить і можливих помилок, у значній частині продукту.

Коефіцієнт стабільності вимог. Призначення критерію: показати, скільки вже реалізованих вимог доводиться переробляти від релізу до релізу розробки нових фіч.

Зрозуміло, повністю ізольованого функціоналу немає, але кількість нових вимог має переважати над змінюваними а коефіцієнт бажано бути менше 0,5. У цьому випадку впроваджуємо нових фічі вдвічі більше, ніж переробляємо існуючих.

Якщо коефіцієнт вище 0,5, особливо якщо більше 1, це швидше за все означає, що раніше зробили те, що виявилося непотрібним. Команда фокусується не на створенні нових цінностей для бізнесу, а на переробці раніше випущених фіч [30, 31].

Також критерій дає уявлення про те, як легко масштабується функціонал системи, додаються нові можливості.

Якість продукту. Як випливає з назви, ця група критеріїв демонструє якість ПЗ, а також якість самої розробки.

Щільність дефектів. Обчислюється частка дефектів, що припадає на окремий модуль протягом ітерації чи релізу.

Призначення критерію: виділити, яка частина є найбільш проблемною. Ця інформація допоможе при оцінці та плануванні робіт з даним модулем, а також при аналізі ризиків [31] (рис. 2.6).

$$\frac{\text{Кількість дефектів в окремому модулі}}{\text{Загальна кількість дефектів в ПЗ}}$$

Рисунок 2.6 – Формула критерію щільності дефектів

Причини великої кількості дефектів до одного конкретному модулі (коефіцієнт більше 0,3) можуть бути різні: неякісні вимоги, кваліфікація розробника, технічна складність і т.д. У будь-якому випадку цей критерій відразу зверне нашу увагу на проблемну зону.

Коефіцієнт повторно відкритих дефектів. Призначення критерію: дати оцінку якості розробки та виправлення дефектів, а також складності продукту чи окремого модуля [31] (рис. 2.7).

$$\frac{\text{Кількість повторно виявлиних дефектів}}{\text{Загальна кількість помлок, включаючи ранні виправлені та нові}}$$

Рисунок 2.7 – Формула критерію коефіцієнту повторного відкритих дефектів

Цей критерій можна розраховувати для всього програмного забезпечення, окремого модуля або функціональності. Чим отримане значення ближче до 0, тим менше під час розробки повторюються старі помилки.

Якщо коефіцієнт вийшов більше 0,2-0,3, це може говорити або про технічну складність модуля та високу пов'язаність вимог у ньому, або про кострубату архітектуру, або про те, що попередній фікс був зроблений неякісно.

Середня вартість виправлення дефекту. Відношення суми витрат понесених командою під час роботи з усіма дефектами (наприклад, в межах релізу) до загальної кількості дефектів [31].

Призначення критерію: показати як дорого нам обходитися виявлення та виправлення кожного дефекту. Це дасть можливість порахувати вигоду від скорочення числа помилок, що допускаються, оцінити доцільність відповідних технік (рис. 2.8).

$$\frac{\text{Вартість з обробки та виправлення ефектів}}{\text{Кількість дефектів}}$$

Рисунок 2.8 – Формула критерію середньої вартості виправлення дефекту

Будь-яких правильних значень тут звичайно немає, все визначатиметься специфікою конкретної ситуації.

Кількість дефектів у коді конкретного розробника. Призначення критерію: підсвітити можливі складнощі у команді розробки, кому з фахівці в не вистачає досвіду, знань чи часу, потрібна допомога [31] (рис. 2.9).

$$\frac{\text{Кількість дефектів у коді конкретного розробника}}{\text{Загальна кількість дефектів}}$$

Рисунок 2.9 – Формула критерію кількості дефектів у коді конкретного розробника

Якщо, наприклад, 50% всіх дефектів припадає на 1 розробника, а всього в команді їх 5, то тут є проблема. З цього не випливає, що цей

програміст погано працює, але сигналізує обов’язково розібратися в подібній ситуації.

Критерій також може бути індикатором особливо складного для розробки та підтримки модуля/функціоналу/системи.

Можливості та ефективність команди QA. Основне завдання цієї групи критеріїв у тому, щоб висловити у цифрах, потім здатна команда тестування. Ці показники можна розраховувати та порівнювати на регулярній основі, аналізувати тенденції, спостерігати за ними, як на роботу команди впливають ті чи інші зміни [31].

Швидкість роботи (velocity) команди QA. Розраховується як відношення реалізованих story points (або вимог або user stories) за кілька, наприклад, 4-5 ітерацій (Sprint) до кількості обраних ітерацій.

Призначення критерію: чисельно висловити можливості, швидкість роботи команди для подальшого планування обсягу робіт та аналізу трендів розвитку (рис. 2.10).

$$\frac{\text{Кількість story points за N ітерцій}}{\text{N}}$$

Рисунок 2.10 – Формула критерію швидкості роботи QA

Критерій дозволяє стежити за швидкістю роботи QA, спостерігати, які внутрішні процеси чи зовнішні впливу на команду можуть на цю швидкість вплинути.

Середній час життя дефекту. Загальний час, протягом якого були відкриті дефекти, знайдені в рамках ітерації або релізу до суми дефектів.

Призначення критерію: показати, скільки в середньому часу йде на роботу з одним дефектом: на його реєстрацію, виправлення та відтворення. Даний показник дозволить оцінити час, необхідний тестування, виділити області ПО з якими виникають найбільші труднощі [31] (рис. 2.11).

$$\frac{\text{Сумарний час виправлення знайдених дефектів}}{\text{Кількість дефектів}}$$

Рисунок 2.11 – Формула критерію середнього часу життя дефекту

Зазвичай час життя дефекту це весь час від його створення (статус Created) до закриття (Closed) за вирахуванням всіх можливих Postponed і Hold. Будь-який баг-трекер дозволяє розрахувати та вивантажити цю інформацію для окремого спринту або релізу.

Також середній час життя дефекту можна розраховувати для різних модулів та функцій програмного забезпечення, або, що найцікавіше, окремо для кожного з тестувальників та розробників з команди. Так, є шанс виявити особливо складні модулі або слабку ланку в команді ПЗ.

Якість роботи команди тестування. Завдання цього набору критеріїв оцінити, наскільки якісно тестувальники виконують свої завдання, визначити рівень компетенцій та зрілості команди QA. Володіючи таким набором показників можна порівнювати команду з нею самою в різні моменти часу або з іншими, зовнішніми групами тестування [31].

Ефективність тестів та тестових наборів. Призначення критерію: показати, як багато помилок у середньому дозволяють виявити наші кейси. Цей критерій відображає якість тест дизайну та допомагає стежити за тенденцією його зміни (рис. 2.12).

$$\frac{\text{Кількість знайдених помилок}}{\text{Кількість кейсів у тестовому наборі}}$$

Рисунок 2.12 – Формула критерію ефективності тестів та тестових наборів

Найкраще розраховувати цей критерій для всіх наборів тестів: для окремих груп функціональних перевірок, регресійного набору, Smoke тестування тощо.

Даний показник «забійності» тестів дозволяє моніторити ефективність кожного набору, як вона змінюється з часом і доповнювати їх «свіжими» тестами.

Коефіцієнт помилок, пропущених на продуктів. Кількість помилок виявлених після випуску релізу / загальна кількість помилок у ПЗ виявлених у процесі тестування і після випуску.

Призначення критерію: продемонструвати якість тестування та ефективність виявлення помилок – яка частка дефектів була відфільтрована, а яка пройшла продуктів [31, 32] (рис. 2.13).

$$\frac{\text{Кількість помилок виявлених після випуску релізу}}{\text{Загальна кількість помилок випущених до та після релізу}}$$

Рисунок 2.13 – Формула критерію коефіцієнт помилок, пропущених на продуктів

Допустимий відсоток помилок, які були пропущені на продуктів, звичайно ж залежатиме від багатьох факторів. Однак, якщо коефіцієнт вийшов >0,1, це погано. Це означає, що кожен десятий дефект не був виявлений під час тестування і привів до проблем, які вже передані користувачам.

Реальний час роботи команди QA. Відношення часу витраченого командою безпосередньо на QA активності до загальної кількості годин.

Призначення критерію: по-перше, збільшити точність планування, а по-друге, відстежувати та керувати ефективністю роботи тієї чи іншої команди [32] (рис. 2.14).

$$\frac{\text{Час витрачений на цільові QA активності}}{\text{Загальна кількість робочих годин команди}}$$

Рисунок 2.14 – Формула критерію реального часу роботи команди QA

Цільові активності, це аналіз, дизайн, оцінки, тестування, робочі зустрічі та багато іншого. Можливі побічні речі – це простий через блокери, проблеми у комунікаціях, недоступність ресурсів тощо.

Звичайно, даний коефіцієнт ніколи не буде дорівнює 1. Практика показує, що для ефективних команд він може становити 0,5-0,6.

Точність оцінки часу за областями/видами/типами робіт.

Призначення критерію: дозволяє використовувати коефіцієнт поправки для наступних оцінок (рис. 2.15).

$$\frac{\text{Оцінний час роботи}}{\text{Фактичний час роботи}}$$

Рисунок 2.15 – Формула критерію точності оцінки часу за областями/видами/типами робіт

Ступінь точності оцінки можна визначити для всієї команди або окремих тестувальників, всієї системи або окремих модулів ПЗ.

Частка непідтверджених дефектів. Призначення критерію: показати скільки дефектів було заведено «вхолосту» [32] (рис. 2.16).

$$\frac{\text{Число дефектів не прийнятих до виправлення}}{\text{Загальна кількість зареєстрованих дефектів}}$$

Рисунок 2.16 – Формула критерію частки непідтверджених (відхилених) дефектів

Якщо частка дефектів, які були відхилені, перевищує 20%, то в команді може спостерігатися розсинхронізація в розумінні, що є дефектом, а що ні.

Зворотній зв’язок та задоволеність користувачів. І насамкінець, група критеріїв, що показує, як продукт було прийнято кінцевими користувачами,

наскільки він відповідав їхнім очікуванням. Але важливим є не тільки зворотний зв'язок про ПЗ: ще одне важливе завдання цієї групи критеріїв – показати, чи задоволені користувачі процесом взаємодії з командою IT загалом і QA зокрема.

Задоволеність користувачів IT-сервісом. Регулярне опитування задоволеності користувачів сервісом IT із виставленням балів.

Призначення критерію: показати, чи довіряють користувачі команді IT, чи розуміють, як і чому організовано її роботу, наскільки ця робота виправдовує очікування.

Критерій може бути індикатором того, що необхідно сфокусуватися на оптимізації процесу або зробити його зрозумілішим і прозорішим для користувачів.

Розрахунок показника задоволеності можна проводити з урахуванням результатів опитування за підсумками релізу. Збираємо всі оцінки та вважаємо середній бал. Далі можна повторно розрахувати такий бал після того, як будуть зроблені зміни в процесі [32].

Задоволеність користувачів продуктом. Регулярне опитування користувачів про те, наскільки вони задоволені продуктом.

Призначення критерію: визначити, наскільки продукт, що розробляється, відповідає очікуванням користувачів, в тому напрямку рухаємося, чи правильно визначаємо важливість фіч і вибираємо варіанти рішень.

Для розрахунку цього критерію також проводимо опитування користувачів та обчислюємо середній бал. Розраховуючи такий показник на регулярній основі (наприклад після кожного релізу) можна стежити за трендом задоволеності користувачів.

Залучення стейкхолдерів. Кількість ініціатив та пропозицій щодо покращення процесу та продукту, що надійшли протягом ітерації (релізу) з боку стейкхолдерів [32].

Призначення критерію: визначити рівень участі зовнішніх стейкхолдерів у роботі над продуктом. Маючи на руках такий критерій можна зорієнтуватися, де потрібно отримати зворотний зв’язок, щоб одного разу не зіткнутися із зневагою та ненавистю проблемами та нерозумінням.

Цікавими є ініціативи, що надходять від усіх стейкхолдерів. Це можуть бути ідеї з боку бізнесу або користувачів щодо спрощення інтерфейсу продукту, розширення його можливостей. Також це можуть бути пропозиції від інфраструктури або команди підтримки щодо технічних аспектів системи: надійності, продуктивності, швидкості установки, моніторингу роботи [32].

3 ВИДИ ТЕСТУВАННЯ ПРОГРАМНОГО ПРОДУКТУ ТА АВТОМАТИЗАЦІЯ ПРОЦЕСІВ ТЕСТУВАННЯ

У цьому розділі на конкретному прикладі перевірятиметься гіпотеза про доцільність автоматизації тестування в компанії. Також будуть написані автоматизовані тести для перевірки цієї гіпотези на реальних даних.

Залежно від цілей види тестування можна умовно розділити на такі типи:

– функціональні;

– нефункціональні;

– пов'язані зі змінами.

3.1 Функціональні види тестування

Функціональні тести базуються на функціях та особливостях, а також взаємодії з іншими системами, і можуть бути представлені на всіх рівнях тестування: компонентному або модульному (Component/Unit testing), інтеграційному (Integration testing), системному (System testing) та приймальному (Acceptance testing)). Функціональні види тестування розглядають зовнішню поведінку системи. Далі перераховані одні з найпоширеніших видів функціональних тестів:

– функціональне тестування (Functional testing);

– тестування безпеки (Security and Access Control Testing);

– тестування взаємодії (Interoperability Testing).

Тестування безпеки – це стратегія тестування, що використовується для перевірки безпеки системи, а також для аналізу ризиків, пов'язаних із забезпеченням цілісного підходу до захисту застосунків, хакерів, вірусів, несанкціонованого доступу до конфіденційних даних. Тестування безпеки

може виконуватися як автоматизовано і в ручну, включаючи перевірку як позитивних, і негативних тестових випадків. Ґрунтується на трьох основних принципах – це конфіденційність, цілісність та доступність.

Конфіденційність – це приховування певних ресурсів чи інформації. Під конфіденційністю можна розуміти обмеження доступу до ресурсу певної категорії користувачів, або іншими словами, за яких умов користувач має доступ до даного ресурсу [29-32].

Існує два основних критерії щодо поняття цілісності:

– довіра. Очікується, що ресурс буде змінено лише відповідним способом певною групою користувачів;

– пошкодження та відновлення. Якщо дані ушкоджуються або неправильно змінюються авторизованим або не авторизованим користувачем, ви повинні визначити, наскільки важливою є процедура відновлення даних.

Доступність є вимогами про те, що ресурси повинні бути доступні авторизованому користувачеві, внутрішньому об’єкту або пристрою. Як правило, чим критичніший ресурс тим вищий рівень доступності має бути.

Тестування взаємодії (Interoperability Testing) – це функціональне тестування, що перевіряє здатність програми взаємодіяти з одним і більше компонентами або системами і включає тестування сумісності (compatibility testing) і інтеграційне тестування (integration testing).

Програмне забезпечення з хорошими характеристиками взаємодії може бути легко інтегровано з іншими системами, не вимагаючи серйозних модифікацій.

У цьому випадку кількість змін і час, необхідний на їх виконання, можуть бути використані для вимірювання можливості взаємодії [31, 32].

3.2 Нефункціональні види тестування

Нефункціональне тестування описує тести, необхідні визначення характеристик програмного забезпечення, які можна виміряти різними величинами. Загалом це тестування того, як система працює. Далі перераховані основні види нефункціональних тестів:

– усі види тестування продуктивності:

– навантажувальне тестування (Performance and Load Testing);

– стресове тестування (Stress Testing);

– тестування стабільності чи надійності (Stability / Reliability Testing);

– об’ємне тестування (Volume Testing;

– тестування установки (Installation testing);

– тестування зручності користування (Usability Testing);

– тестування на відмову та відновлення (Failover and Recovery Testing);

– конфігураційне тестування (Configuration Testing).

Тестування установки спрямоване на перевірку успішної інсталяції та налаштування, а також оновлення або видалення програмного забезпечення.

Зараз найбільш поширена установка програмного забезпечення за допомогою інсталяторів (спеціальних програм, які самі по собі так само вимагають належного тестування).

У реальних умовах інсталяторів може бути. У цьому випадку доведеться самостійно виконувати установку програмного забезпечення, використовуючи документацію у вигляді інструкцій або readme файлів, що крок за кроком описують всі необхідні дії та перевірки [31, 32].

У розподілених системах, де програма розгортається на вже працюючому оточенні, простого набору інструкцій може бути мало. Для цього часто пишеться план установки (Deployment Plan), що включає не тільки кроки по інсталяції програми, але і кроки відкату (roll-back) до попередньої версії, у разі невдачі. Сам по собі план установки також має

пройти процедуру тестування, щоб уникнути проблем при видачі в реальну експлуатацію. Особливо це актуально, якщо установка виконується на системи, де кожна хвилина простою – це втрата репутації та великої кількості коштів, наприклад: банки, фінансові компанії чи навіть банерні мережі. Тому тестування установки можна назвати одним із найважливіших завдань щодо забезпечення якості програмного забезпечення.

Саме такий комплексний підхід із написанням планів, покроковою перевіркою встановлення та відкату інсталяції, повноправно можна назвати тестуванням установки [32].

Тестування зручності користування – це метод тестування, спрямований на встановлення ступеня зручності використання, навчання, зрозумілості та привабливості для користувачів продукту, що розробляється в контексті заданих умов.

Тестування зручності користування дає оцінку рівня зручності використання програми за такими пунктами:

– продуктивність, ефективність (efficiency) – скільки часу та кроків знадобиться користувачеві для завершення основних завдань програми, наприклад, розміщення новини, реєстрації, купівля тощо? (менше – краще);

– правильність (accuracy) – скільки помилок зробив користувач під час роботи з застосунком? (менше – краще);

– активізація у пам’яті (recall) – як багато користувач пам’ятає роботу програми після призупинення роботи з ним на тривалий час? (повторне виконання операцій після перерви має відбуватися швидше, ніж у нового користувача);

– емоційна реакція (emotional response) – як користувач почувається після завершення завдання – розгублений, відчув стрес? Чи рекомендує користувач систему своїм друзям? (позитивна реакція – краще).

Тестування на відмову та відновлення – це тестування того, наскільки добре програма може відновлюватися після падіння (помилки у застосунку, зависання), апаратної проблеми та інших схожих випадків. При цьому

тестуванні програма навмисне виводиться з ладу різними шляхами для перевірки правильності відновлення.

Приклади тестування відновлення:

– під час роботи програми комп’ютер раптово перезавантажується. Після перезавантаження перевіряється правильність та цілісність даних програми;

– від’єднайте кабель мережі під час отримання даних від мережі. Через деякий час приєднати кабель назад і проаналізувати здатність програми продовжувати отримувати дані з того моменту, як зникло з’єднання з мережею;

– перезапустити системи в той час як інтернет-браузер має кілька активних сесій. Після перезавантаження перевірити, що оглядач може відновити їх усі.

Конфігураційне тестування це різновид тестування продуктивності, коли замість тестування з позиції навантаження, тестуються ефекти зміни конфігурації докладання на продуктивність і поведінка [33].

3.3 Пов’язані зі змінами види тестування

Після проведення необхідних змін, таких як виправлення дефекту, програмне забезпечення має бути протестовано повторно для підтвердження того факту, що проблема була справді вирішена. Нижче перелічені види тестування, які необхідно проводити після встановлення програмного забезпечення, для підтвердження працездатності програми або правильності здійсненого виправлення дефекту:

– димове тестування (Smoke Testing);

– перевірка узгодженості/справності (Sanity Testing);

– регресійне тестування (Regression Testing).

Поняття димове тестування пішло з інженерного середовища. При введенні в експлуатацію нових апаратних компонентів вважалося, що тестування пройшло успішно, якщо з установки не пішов дим. В області тестування програмного забезпечення, воно спрямоване на поверхневу перевірку всіх модулів програми на предмет працездатності і наявність критичних і блокуючих дефектів, що швидко знаходяться. За результатами димового тестування робиться висновок про те, чи приймається ні встановлена версія програмного забезпечення в подальше тестування. Для полегшення роботи, економії часу та людських ресурсів рекомендується автоматизувати димові випробування [33, 34].

Перевірка справності також є швидким тестуванням і визначає, чи доцільно переходити до наступних етапів тестування. Основна відмінність від димового тестування в тому, що димове тестування визначає, чи можливо продовжувати тестування, у той час як перевірка справності визначає доцільність. Бажано, щоб тест справності тестував найменшу кількість функцій програми, необхідне визначення правильної роботи основний логіки програми. Тести справності та димові тести – дві техніки для передчасної зупинки тестування та економії часу та сил на більш трудомісткі етапи.

Регресійне тестування – вид тестування спрямований на перевірку змін, зроблених у застосунку або навколишньому середовищі (лагодження дефекту, злиття коду, міграція на іншу операційну систему, базу даних, вебсервер або сервер програми), для підтвердження того факту, що існуюча раніше функціональність працює як і колись. Регресійними можуть бути тести як функціональні, так і нефункціональні [34].

Як правило, для регресійного тестування використовуються тестові випадки, написані на ранніх стадіях розробки та тестування. Це дає гарантію того, що зміни у новій версії програми не пошкодили вже існуючу функціональність. Рекомендується виконувати автоматизацію регресійних

тестів для прискорення подальшого процесу тестування та виявлення дефектів на ранніх стадіях розробки програмного забезпечення.

Сам по собі термін «Регресійне тестування», залежно від контексту використання, може мати різний зміст. Сем Канер, наприклад, описав три основних типи регресійного тестування:

– регресія багів (Bug regression) – спроба довести, що виправлена помилка насправді не виправлена;

– регресія старих багів (Old bugs regression) – спроба довести, що зміна коду чи даних зламало виправлення старих помилок, тобто. старі баги почали знову відтворюватись;

– регресія побічного ефекту (Side effect regression) – спроба довести, що недавня зміна коду або даних зламала інші частини програми, що розробляється [34].

3.4 Впровадження автоматизованих тестів

Автоматизація функціональних тестів вебпрограм зазвичай відбувається за допомогою спеціальних програмних фреймворків, що дозволяють симулювати поведінку реальних користувачів з програмного середовища. Ці фреймворки зазвичай складаються з 2-х частин: програми або надбудови над браузером, яка дозволяє керувати браузером та виконувати команди всередині нього та програмного API, що надає зручні функції для контролю цієї програми. Нижче наведено основні функції, які надають подібні фреймворки:

– загальні функції браузера (Відкриття нової вкладки або вікна та контролю їх розмірів);

– навігація між вебсторінками;

– пошук вебелемента на сторінці із зазначеними параметрами;

– функції очікування певних подій (наприклад, очікування повного завантаження сторінки або появи певного елемента);

– симуляція дій користувача, наприклад натискання кнопки миші на певний елемент або введення послідовності символів з клавіатури;

– для більш складних дій можна виконувати JavaScript команди на сторінці.

Автоматизувати тести можна за допомогою різних програмних фреймворків. Є велика кількість інструментів для автоматизації функціонального тестування основні з яких Selenium WebDriver, Watij HtmlUnit Jamaleon.

Мною був обраний Selenium WebDriver, оскільки він на відміну від інших фреймрівків дозволяє вибрати мову програмування для реалізації тестів (більшість інших фреймворків дозволяють використовувати тільки Java), здатний працювати з усіма браузерами і має максимально багатий функціонал з точки зору функціонального тестування. З можливих мов програмування для реалізації тестів (Java, C#, Ruby, Python), що надаються Selenium WebDriver, був обраний C#, оскільки на момент написання даної роботи мною був накопичений більший досвід використання мов Java та JavaScript, а розвиток у іншій мові буде для мене досвідом , який я зможу використати у майбутньому.

При практичному застосуванні автоматизованих функціональних тестів дуже швидко виявляється що тестування кожного окремого вебзастосунку має свою специфіку. Це підштовхує до створення додаткового програмного шару між фреймворком для тестування та самими автотестами. Такий підхід дозволяє мінімізувати кількість коду, що повторюється шляхом винесення його в функції цього шару, що позитивно позначається на швидкості створення автотестів, знижує ймовірність виникнення помилок і покращує зручність читання коду. Зазвичай такі проміжні API містять утиліти та класи, що дозволяють оперувати більш високорівневими сутностями, що існують у контексті конкретного

вебзастосунку. Часто в автотестах може знадобитися протестувати функціонал, для якого необхідно авторизуватися, знайти типовий для всіх сторінок програми елемент, зробити прямий запит до бази даних, щоб звірити дані з представленими на сайті тощо. Такі дії регулярно виконуватимуться з автотестів, при цьому якщо не винести їх в окреме API вони лише захаращуватимуть код, оскільки займають багато місця і не мають прямого відношення до конкретного тест кейсу.

Цей тест перевіряє роботу технічної підтримки сайту. Для перевірки цього функціоналу необхідно надіслати повідомлення до служби підтримки сайту та перевірити, що надіслане повідомлення відображається в особистому кабінеті. Далі йде перевірка на можливість надіслати повідомлення до служби підтримки зі сторінки особистого кабінету та можливість надіслати друге повідомлення за раніше створеним. Цей тест-кейс також має перевірку на коректну роботу пошуку на сторінці технічної підтримки в особистому кабінеті користувача (табл. 3.1).

Таблиця 3.1 – Ручний тест для перевірки технічної підтримки сайту

№	**Кроки**	**Очікувана реакція**	**Статус виконання**
1	**2**	**3**	**4**
1	Надсилання звернення. Перехід до особистого кабінету за кнопкою Повернутися до списку	Відображається список	Пройдено
2	Відображення даних на вкладці особистого кабінету «Техпідтримка»	Дані відображаються	Пройдено

Продовження таблиці 3.1

1	2	3	4
3	Створення нового повідомлення з вкладки особистого кабінету «Техпідтримка»	Повідомлення створено	Пройдено
4	Відправка другого повідомлення по прикладу раніше створеного	Повторне повідомлення створено	Пройдено
5	Перевірка пошуку на вкладці особистого кабінету «Техпідтримка»	Пошук працює коректно	Пройдено
Тип виконання	Вручну		
Час виконання (хвилин)	10,00		
Пріоритет	Medium		
Деталі виконання			
Версія (складання)	Тестове складання		
Тестувальник	Vasylenko Iryna		
Execution Result	Пройден		
Execution Mode	Вручну		

Цей тест перевіряє працездатність усіх вкладок на сторінці «Стрічка повідомлень» в особистому кабінеті користувача, працездатність фільтра та пошуку на сторінці стрічки повідомлень (табл. 3.2).

Таблиця 3.2 – Ручний тест для перевірки вкладок на сторінці «Стрічка повідомлень»

№	Кроки	Очікувана реакція	Статус виконання
1	Працездатність вкладок, відображення правильної інформації на кожній з них	Вкладки працюють коректно, відображається вірна інформація	Пройдено
2	Відображення заявок, чернеток	Відображаються коректно	Пройдено
3	Працездатність пошуку, фільтру	Пошук та фільтр працюють коректно	Пройдено
Тип виконання	Вручну		
Час виконання (хвилини)	10,00		
Пріоритет	Medium		
Деталі виконання			
Версія (складання)	Тестове складання		
Тестувальник	Vasylenko Iryna		
Execution Result	Пройдено		
Execution Mode	Вручну		

Далі представлено реалізацію описаних вище тестів.

Тест, який перевіряє надсилання повідомлення до служби підтримки та відображення цього повідомлення до особистого кабінету користувача:

– перехід на сторінку «Стрічка повідомлень», вкладка «Техпідтримка»:

```
WebDriver.Chrome.NavigateToUrl("http://lk-dev.test.gosuslugi.ru/notifications?type=FEEDBACK");
```

– очікування появи елемента «Створити нове повідомлення» та наступне натискання на нього:

```
WebDriver.Chrome.WaitUntilElementVisible(By.CssSelector("#content>div.ng-scope>div>ng-include>div>fieldset>ul>li.feedback-button.ng-scope>a")).SafeClick();
```

– заповнення форми зворотного зв’язку:

```
WebDriver.Chrome.WaitUntilElementVisible(By.Id("_epgu_el2")).SafeClick();
WebDriver.Chrome.WaitUntilElementVisible(By.XPath("//*[@id=\"feedbackForm\"]/div/div/div/div[2]/ul/li[4]")).SafeClick();
WebDriver.Chrome.WaitUntilElementVisible(By.XPath("//*[@id=\"feedbackForm\"]/div/div/ng-include/epgu-input[1]/div/textarea")).SendKeys("TEST");
WebDriver.Chrome.WaitUntilElementVisible(By.XPath("//*[@id=\"feedbackForm\"]/div/div/ng-include/epgu-input[1]/div/textarea")).Submit();
```

– фіксація часу надсилання повідомлення:

```
DateTime time = DateTime.Now;
```

– очікування появи елемента «Повернутися до списку» та наступне натискання на нього:

```
WebDriver.Chrome.WaitUntilElementVisible(By.XPath("//*[@id=\"content\"]/div[2]/div/div/div/div[4]/a")).SafeClick();
```

– очікування появи повідомлення з відповідним часом відправки та наступне натискання на нього:

WebDriver.Chrome.RefreshUntil(_ => _.FindElements(By.ClassName("advice-datetime")).Any(a => a.Text.Equals(time.ToString("dd.MM.yyyy HH:mm"))));

WebDriver.Chrome.FindElements(By.ClassName("advice-datetime")).First(_ => _.Text.Equals(time.ToString("dd.MM.yyyy HH:mm"))).SafeClick();

– створення унікального повідомлення для надсилання додаткового повідомлення:

Guid guid = Guid.N;

– надсилання додаткового повідомлення:

WebDriver.Chrome.WaitUntilElementVisible(By.XPath("//*[@id=\"content\"]/div[2]/div/ng-include/ng-include/div/div[2]/form/div/div[1]/div[1]/textarea")).SendKeys(guid.ToString()); WebDriver.Chrome.WaitUntilElementVisible(By.XPath("//*[@id=\"content\"]/div[2]/div/ng-include/ng-include/div/div[2]/form/div/div[1]/div[1]/textarea")).Submit();

– очікування відображення надісланого повідомлення у вікні листування:

WebDriver.Chrome.WaitUntilElementVisible(By.ClassName("event-text"));

– перевірка збігу тексту повідомлення з відправленим раніше:

Assert.IsTrue(WebDriver.Chrome.FindElements(By.ClassName("event-text")).Any(_ => _.Text == guid.ToString()), "No message recieved");}.

Тест, який перевіряє функцію пошуку на сторінці техпідтримки:

– перехід на сторінку «Стрічка повідомлень», вкладка «Техпідтримка»:

WebDriver.Chrome.NavigateToUrl("http://lk-dev.test.gosuslugi.ru/notifications?type=FEEDBACK");

– введення тексту в полі фільтрації повідомлень:

string searchText = "отзыв";

WebDriver.Chrome.WaitUntilElementVisible(By.XPath("//*[@id=\"content\"]/div[2]/div/ng-include/div/fieldset/ul/li[1]/dl[2]/dd/input")).SendKeys(searchText);

– очікування поки що всі відсортовані повідомлення не міститимуть введений у фільтр текст:

WebDriver.Chrome.WaitUntil(_ => _.FindElements(By.ClassName("highlighted")).All(a => a.Text == searchText));}.

Тест, який перевіряє працездатність усіх вкладок Особистого кабінету:

– перехід на сторінку «Стрічка повідомлень», вкладка «Всі»:

WebDriver.Chrome.NavigateToUrl("http://lk-dev.test.gosuslugi.ru/notifications?type=all");

– перехід на всі вкладки на сторінці «Стрічка повідомлень»:

WebDriver.Chrome.NavigateToUrl("http://lk-dev.test.gosuslugi.ru/notifications?type=PAYMENT");

WebDriver.Chrome.WaitUntilElementExists(By.ClassName("advice-item"));

WebDriver.Chrome.NavigateToUrl("http://lk-dev.test.gosuslugi.ru/notifications?type=ORDER");

WebDriver.Chrome.WaitUntilElementExists(By.ClassName("advice-item"));

WebDriver.Chrome.NavigateToUrl("http://lk-dev.test.gosuslugi.ru/notifications?type=DRAFT");

WebDriver.Chrome.WaitUntilElementExists(By.ClassName("advice-item"));

WebDriver.Chrome.NavigateToUrl("http://lk-dev.test.gosuslugi.ru/notifications?type=FEEDBACK");

WebDriver.Chrome.WaitUntilElementExists(By.ClassName("advice-item"));");

– перехід на вкладку «Всі» та введення в поле фільтрації певного тексту:

```
WebDriver.Chrome.NavigateToUrl("http://lk-dev.test.gosuslugi.ru/notifications?type=all
WebDriver.Chrome.WaitUntilElementVisible(By.XPath("//*[@id=\"content\"]/div[2]/div/ng-include/div/fieldset/ul/li/dl[2]/dd/input")).SendKeys("35");
```

– очікування появи результатів фільтрації:

```
WebDriver.Chrome.WaitUntilElementExists(By.ClassName("advice-item"));
```

– перевірка правильності фільтрації повідомлень:

```
Assert.IsTrue(WebDriver.Chrome.FindElements(By.ClassName("highlighted")).All(_ => _.Text == "35"));
```

– очищення поля фільтрації:

```
WebDriver.Chrome.WaitUntilElementVisible(By.XPath("//*[@id=\"content\"]/div[2]/div/ng-include/div/fieldset/ul/li/dl[2]/dd/input")).Clear();
```

– фільтрування повідомлень за датою:

```
WebDriver.Chrome.WaitUntilElementVisible(By.XPath("//*[@id=\"_epgu_el1\"]/div[1]")).SafeClick();WebDriver.Chrome.WaitUntilElementVisible(By.XPath("//*[@id=\"content\"]/div[2]/div/ng-include/div/fieldset/ul/li/dl[1]/dd/div/div[2]/ul/li[5]")).SafeClick();
WebDriver.Chrome.WaitUntilElementExists(By.ClassName("advice-datetime"));
```

– перевірка відповідності дати на всіх елементах із датою фільтрації:

```
Assert.IsTrue(WebDriver.Chrome.FindElements(By.ClassName("advice-datetime")).All(_ => (DateTime.Now - DateTime.Parse(_.Text)).Duration().Days == 0)).
```

Після того як було автоматизовано кілька тестів, описаних вище, можна дійти невтішного висновку про доцільність автоматизації

Розробка автоматизованих тестів складалася з двох етапів: розробка проміжного шару між фреймворком для тестування та самими авто тестами,

та розробка автоматизованих тестів. На перший етап пішло близько 16 години робочого часу, а на другий близько 8. При цьому було розроблено 2 масштабні тести.

3.5 Оцінка ефективності автоматизації тестування

Розглянемо приклад із практики використання автоматизованого тестування, що відображає підсумки науково-дослідної роботи, проведеної в Європі, щодо збору вимірювань результатів автоматизованого тестування з метою вивчення переваг методів автоматизованого тестування порівняно з тестуванням вручну.

Вивчення проводилося компанією imbus GmbH за спонсорської підтримки європейської комісії. Європейська ініціатива в галузі створення інформаційних систем та програмного забезпечення (ESSI), створена Європейською Комісією, сприяла експерименту з удосконалення процесів (PIE) компанії imbus в галузі автоматизованого тестування графічних інтерфейсів користувача. PIE складався з двох частин: основний проєкт та експеримент [34, 35].

Основний проєкт був із розробкою програмного забезпечення, проведеної певними компаніями. Він полягав у розробці інтегрованого засобу програмного забезпечення ПК, що застосовується для підтримки обладнання базової радіостанції. Ця програма надавала графічний інтерфейс користувача для введення в експлуатацію, визначення параметрів, діагностики апаратних засобів, завантаження програмного забезпечення, створення бази даних обладнання та польової та автономної діагностики багатьох типів базових станцій для радіозв’язку GSM, включаючи графічний редактор для визначення обладнання [35].

В експерименті використовувався новий метод проведення тестування – автоматизоване тестування. Для з’ясування, чи цей метод

надає якісь переваги, його результати порівняли з тими, що були отримані за допомогою старого методу – ручного тестування, яке проводилося паралельно з основним проєктом.

Мета проєкту – оптимізація процесу тестування графічного інтерфейсу користувача (GUI) і підвищення ступеня використання автоматизованого тестування за рахунок застосування відповідних інструментальних засобів. Для отримання порівняльних числових оцінок результатів тестування дослідники проводили тестування, використовуючи методи ручні в одному прикладі і автоматизовані в іншому. Вимоги до тестування, визначені у специфікації тестування, були виконані тестувальником вручну в основному проєкті [35].

Одночасно PIE-команда використовувала вимоги до тестування для розробки процедур автоматизованого тестування та подальшого виконання тестів та регресійного тестування за допомогою автоматизованого засобу WinRunner.

Для визначення того, наскільки економічнішим є автоматизоване тестування GUI-інтерфейсу в порівнянні з тестуванням вручну, вимірювалися і порівнювалися витрати при використанні обох методів в експерименті PIE. Першим розглядалося питання, скільки разів було повторено певний тест перш, ніж автоматизоване тестування стало дешевшим за ручне. Результати даного експерименту щодо вдосконалення процесу представлені у таблиці 3.3.

Тест і: тести, визначені у межах специфікацій із тестування в основних проєктах.

V_m : Витрати створення специфікації тестування.

V_a : витрати на створення специфікації тестування та на реалізацію.

D_m : витрати на виконання одного тесту вручну.

Таблиця 3.3 – Результати експерименту щодо удосконалення процесу розробки

Підготовка V			**Виконання D**		**N**	**Витрати E виконання n автоматизованих тестів**			
Тест	Ручний	Автоматизований	Ручний	Автоматизований		1	5	10	20
1	16	56	24	1	1,74	143%	45%	26%	32%
2	10	14	2	0,1	2,11	118%	73%	50%	20%
3	10	16	4,5	0,2	1,40	112%	52%	33%	64%
4	20	28	1,5	0,2	6,15	131%	105 %	86%	57%
5	10	15	1	0,1	5,56	137%	103 %	80%	43%
6	10	15	1,5	0,1	3,57	131%	89%	64%	54%
7	10	11,5	0,75	0,1	2,31	108%	87%	71%	54%
8	10	11,5	0,5	0,1	3,75	110%	96%	83%	68%
9	10	14	3	0,1	1,38	108%	58%	38%	23%
10	10	10,6	0,5	0,1	1,50	102%	89%	77%	63%
Разом	116	191,6	39,25	2,1	2,03	125%	65%	42%	26%

D_a: Витрати реалізацію тесту після початку автоматизованого тестування. Час на тестування не враховувалося, тому що воно виконувалося без контролю за допомогою CR-засобу.

V і D вимірюються в годинах.

$$E_n = A_a / A_m = \left(V_a + n * D_a\right) / \left(V_m + n * D_m\right). \qquad (3.1)$$

N: значення показнику економічної ефективності.

Підготовка специфікації вимог щодо тестування завантаження програмного забезпечення (завантаження програмного забезпечення була однією з функцій продукту і проводилася при натисканні кнопки ЗАВАНТАЖИТИ) зайняла 10 годин (Ручний V у рядку Тест 2 у таблиці 3.3). Програмування цих тестів зайняло ще 4 години, що увійшло у повний час автоматизованої підготовки до тестування (Автоматизований V), що дорівнює 14 годин (результати Тесту 2). Ручне виконання цих тестів зайняло 2 години (Ручний D) на противагу 0,1 години (Автоматизований D), оскільки тестувальнику потрібно було перевірити та проаналізувати звіти, згенеровані засобами тестування при автоматичному виконанні тестів [36]. На основі вимірювань, отриманих в результаті одноразового виконання тестування, можна обчислити витрати, які були б потрібні для повторного тестування 5, 10 або 20 разів. Наприклад, у разі п'яти автоматизованих виконань тестів коефіцієнт зменшення витрат на тестування порівняно з проведенням тестування вручну становитиме:

$$E_5 = A_a / A_m = \left(V_a + 5 * D_a\right) / \left(V_m + 5 * D_m\right) = \\ = \left(14 + 5 * 0.1\right) / \left(10 + 5 * 2\right) = 14.5 / 20 = 0.725 = 73\%. \quad (3.2)$$

У таблиці 3.3 показник економічної ефективності представлений коефіцієнтом N відповідно до рівності $E = A_a / A_m = 100\%$,

де E – це відносні витрати;

A – абсолютні витрати (A_a – абсолютні витрати на автоматизацію, A_m – абсолютні витрати при тестуванні вручну);

N – значення показника економічної ефективності;

V – підготовка;

D – виконання;

Вимірювання, виконані в ході експерименту, показують, що економічної ефективності можна досягти вже під час проведення другого циклу регресійного тестування (*n*total = 2,03). Ця рентабельність, однак,

вимагає дотримання двох умов: (1) тестування виконується без втручання людини (наприклад, вночі), і (2) не потрібно жодних подальших модифікацій скриптів для того, щоб повторно виконати їх стосовно остаточних версій продукту. Як згадувалося, ці необхідні умови нелегко дотриматися [36].

Низькоякісне програмування, здійснене на початку тестування, призводить до необхідності підтримки тестових скриптів при кожному повторному тестуванні. З іншого боку, якщо команда тестувальників встановить чіткі межі автоматизованого тестування GUI (і CR-засіб з'явиться наріжним каменем, а не засобом для вирішення всіх проблем), зниження вартості до 40% для типового циклу тестування продукту (E_{10}) цілком реальне [35, 36].

3.6 Оцінка повноти тестування

Набір тестів, що використовується під час тестування, завжди кінцевий і, більше, обмежений міркуваннями економічної ефективності розподілу ресурсів між різними видами діяльності під час розробки ПЗ. Тому вкрай важливо будувати його так, щоб використовувані тести перевіряли якнайбільше різних аспектів функціональності системи в якомога більшому розмаїтті ситуацій. Щоб систематичним чином перебирати ситуації, що істотно відрізняються один від одного, використовують критерії повноти тестування або критерії адекватності тестування. Тестовий набір, що відповідає заданому критерію повноти, називають повним за цим критерієм [36].

Найчастіше для визначення критерію повноти деякі з можливих тестових ситуацій розглядають як еквівалентні та визначають кількість класів нееквівалентних тестових ситуацій, які зустрілися або «покриті» під час тестування. Такі критерії повноти називають критеріями тестового

покриття. При цьому визначається і числовий критерій тестового покриття – частка покритих класів ситуацій серед усіх можливих. Критерій повноти може використовувати різні значення критерію, наприклад, він може вимагати, щоб повний тестовий набір завжди покривав 100% виділених класів ситуацій, або вважати достатнім покриття 85% класів ситуацій. Оскільки для одного критерію покриття можна визначити багато критеріїв повноти, далі мова найчастіше йде про різні критерії тестового покриття.

Повноту тестування можна визначати по-різному, але в основі будь-якого критерію повноти лежить уявлення про можливі помилки в системі, що тестується. Різні способи класифікації ситуацій, що відображають їхню різноманітність з точки зору тестування, перераховані нижче. Кожен з них і будь-яке їхнє підмножина спільно можуть використовуватися для визначення критерію тестового покриття [36]. Класифікувати ситуації можна так:

– на основі структурних елементів системи, що тестується, які виконуються або задіяні в ході тестування;

– на основі структури вхідних даних, які використовуються під час тестування;

– на основі елементів вимог, що перевіряються під час виконання тестів;

– на основі явно сформульованих припущень про помилки, виявлення яких мають забезпечити тести.

3.7 Структурні критерії

Критерії повноти тестування і критерію тестового покриття, засновані на структурі системи, що тестується, називаються структурними, а тестування, що проводиться з їх використанням – структурним тестуванням.

В основі структурних критеріїв повноти лежить проста ідея: якщо помилка знаходиться в якійсь конструкції коду, в якомусь компоненті системи, що тестується, то виконавши цю конструкцію або змусивши працювати цей компонент, ми, швидше за все, зможемо її виявити. Відповідно, якщо у двох ситуаціях виконуються одні й ті самі елементи коду, така помилка буде виявлятися в обох ситуаціях, або не виявлятися в жодній, тому їх можна оголосити еквівалентними і перевіряти завжди тільки одну з таких ситуацій [37].

Це припущення рідко виконується практично, проте як евристика визначення критеріїв тестового покриття, воно досить корисно. Далі для деяких конкретних структурних критеріїв будуть наведені приклади простих програм, у яких виконання однієї й тієї конструкції в деяких випадках розкриває помилку, а в деяких – ні.

Структурні критерії покриття різняться залежно від обсягу елементів системи, що використовуються їх визначенні. Можна виділити три рівні структурних критеріїв – рівень окремої функції або окремого методу класу, рівень компонента або класу, що включає кілька операцій, і рівень підсистеми або системи в цілому, у складі яких може бути багато компонентів.

Незалежно від рівня структурні критерії можуть бути засновані на інформації двох видів – на інформації про передачу управління між різними елементами системи, що виконуються, або на інформації про використання та запис даних. Критерії першого типу називаються заснованими на потоці управління, другого типу – заснованими на потоках даних [36, 37].

Важливою перевагою структурних критеріїв покриття є можливість їхнього автоматизованого обчислення за наявності доступу до коду або схем архітектури системи, що тестується. Істотним недоліком є відсутність урахування вимог – можливо покрити усі елементи структури, але не виявимо, що якусь вимогу просто забули реалізувати.

Структурні критерії покриття для однієї функції або методу на основі потоку управління базуються на елементах коду цієї функції або цього методу, що виконуються в ході тесту.

Найбільш простий з таких критеріїв – критерій покриття інструкцій (statement coverage), що дорівнює частці виконаних під час тестування інструкцій коду функції стосовно всіх її інструкцій. Оскільки у більшості сучасних мов програмування прийнято писати не більше однієї інструкції у рядку, цей критерій найчастіше корелює з критерієм покриття рядків вихідного коду. Однак завжди при розмові про рядки коду варто уточнювати, наскільки вони відповідають інструкціям, тому що частина рядків містить декларативну інформацію, яка не виконується, і інформація про інструкції дозволяє точніше оцінити ситуацію. У разі, якщо частина інструкцій недосяжна, тобто. не може бути виконана за жодних умов, частку покритих інструкцій визначають тільки по відношенню до всіх досяжних інструкцій [37].

Певний код може бути помилково пропущений, а покриття інструкцій, природно, не гарантує виявлення пропущених інструкцій, оскільки воно обчислюється лише з наявних. Щоб вирішити цю проблему, використовують покриття гілок.

Для визначення гілок необхідно розглянути граф потоку управління програми. Кожен умовний оператор (як і оператор циклу чи вибору) має кілька ребер графа, які з нього, відповідно до можливим ходом виконання інструкцій, тобто визначає розгалуження потоку управління. Кожне ребро, що виходить з вершини графа, з якої виходять інші ребра, називається гілкою.

Умовному оператору відповідають два виходять з його вершини ребра, оператору перевірки умови циклу – теж два (вийти з циклу чи ні), а оператору вибору може відповідати багато ребер, що виходять – за кількістю зазначених варіантів значень виразу, за яким здійснюється вибір.

Проте вірно, що, якщо тести дають 100% покриття гілок, вони завжди дадуть і 100% покриття інструкцій. Для величин покриття, менших за 100%, жодних загальних співвідношень між покриттям інструкцій і гілок вказати не можна. З одного боку, кількість інструкцій може значно перевищувати кількість гілок, з іншого боку, половина гілок програми може містити інструкцій. Тому покриття 95% інструкцій може відповідати лише 5% покритих гілок і, навпаки, покривши 95% гілок, можна покрити лише 5% інструкцій [37].

Але те що, 100% покриття гілок автоматично означає 100% покриття інструкцій, вже досить важливий. Він показує, що критерій покриття гілок «не грубіше» критерію покриття інструкцій, що виділяє не менше різноманітних ситуацій. Якщо 100% покриття по одній метриці тестового покриття тягне за собою 100% покриття по іншій метриці, кажуть, що перший критерій сильніший або тонший. Відповідно, другий слабше або грубіше за перший. Якщо один критерій сильніший за інший, а той теж сильніше за перший, вони еквівалентні.

Ще сильнішим критерієм є критерій комбінацій умов. Комбінація умов визначається наступним чином. Виділимо умови всіх розгалужень у коді програми, які визначаються умовними операторами, операторами циклу або операторами вибору. Ці умови є предикатами, складеними за допомогою логічних операцій (заперечення «не», кон’юнкції «і», диз’юнкції «або», рівності, нерівності або виключає «або») з елементарних умов – логічних формул, які вже не розкладаються на складові логічні формули. Виділивши всі елементарні умови з розгалужень у коді заданої програми, можемо складати комбінації з цих умов та їх заперечень, або, що те саме, з їх значень true і false (1 і 0) [35-38].

Комбінація значень елементарних умов покривається тестом, якщо під час виконання цих умов у певний момент мають у точності ці значення.

Критерій покриття комбінацій умов (multiple condition coverage) визначається як частка покритих текстами комбінацій значень елементарних

умов, що беруть участь в умовах розгалужень програми, по відношенню до загальної кількості комбінацій значень елементарних умов.

У випадку питання про здійсненності різних комбінацій досить непростий, оскільки елементарні умови може бути пов'язані неявно. Наприклад, вони можуть використовувати глобальні змінні, значення яких підпорядковуються нетривіальним обмеженням, скажімо, одна зі змінних може представляти список якихось об'єктів, а друга індекс одного з об'єктів у цьому списку, або –1, якщо список порожній. У випадку визначити здійсненність комбінації значень довільних формул виявляється досить складно, оскільки цього потрібно знати зміст використовуваних умовах змінних і функцій. Тому обчислення тестового покриття за критерієм покриття комбінацій умов у загальному випадку не автоматизується [35-38].

Інша проблема практичного використання покриття за цім критерієм пов'язан з можливим експоненційним зростанням кількості ситуацій, що виділяються для програми при зростанні її розміру. Якщо в програмі використовується n операторів розгалуження, умова кожного їх елементарно і між цими умовами немає жодних зв'язків, виникає лише $2n$ гілок, але $2n$ можливих комбінацій умов. Тому на практиці використовуються критерії тестового покриття, сильніші, ніж покриття гілок, але що приводять до меншої кількості різних ситуацій, ніж покриття комбінацій умов.

Найбільш слабкий з таких критеріїв – критерій покриття умов та гілок (condition/decision coverage або condition/branch coverage). Вважається вона в такий спосіб. Для кожної елементарної умови визначається, скільки значень воно може приймати за всіх можливих сценаріїв виконання програми. Зазвичай, якщо немає специфічних обмежень, може приймати два значення. Але іноді зустрічаються постійні умови, які завжди рівні true, або завжди рівні false, при всіх виконаннях програми. Визначається загальна кількість можливих значень елементарних умов, у яких кожна звичайна умова вносить 2 значення, а постійна умова – 1. Критерій покриття умов і гілок

обчислюється як відношення загальної кількості значень, які всі елементарні умови приймали в ході тесту, складеного до кількості виконаних гілок сумі загального можливого числа можливих значень умов та числа досяжних гілок [37, 38].

Якщо при цьому не враховувати покриття гілок, вийде критерій покриття умов (condition coverage), який, однак, може виявитися не сильнішим за критерій покриття гілок – те, що всі елементарні умови набували всіх можливих значень, ще не гарантує, що всі гілки були покриті.

За наявності n елементарних умов у програмі критерію покриття умов та гілок визначає не більше 2*n* ситуацій. Він, як легко бачити, сильніше за критерій покриття гілок. Однак у нашому прикладі тести, що дають 100% покриття гілок, дають повне покриття умов і гілок, не виявляючи внесену помилку.

Більш сильний критерій покриття – критерію модифікованого покриття умов та гілок (modified condition/decision coverage, MC/DC). Набір тестів вважається таким, що досягає 100% покриття за цім критерієм, якщо:

– кожна досяжна гілка покривається цим набором;

– кожна непостійна елементарна умова набуває обох можливих значень при виконанні цього набору;

– для кожного складового умови розгалуження і кожного елементарного умови, що входить до нього, зміна значення якого здатна змінити значення всієї умови, є два тести, в яких всі інші елементарні умови, що входять в цю складову умову, мають одні і ті ж значення, а дана елементарна умова і складова Умова загалом у цих тестах мають різні значення [39].

Простіше кажучи, у повному тесті за критерієм MC/DC має бути продемонстровано, що кожна елементарна умова, здатна впливати на результуюче значення, що включає умови розгалуження, дійсно змінює його значення незалежно від інших елементарних умов.

3.8 Критерії повноти на основі структури вхідних даних

Часто при тестуванні не можна скористатися інформацією про пристрій системи, що тестується, оскільки її вихідний код недоступний. При цьому вся відома інформація про її структуру – це структура вхідних даних – загальний список доступних ззовні інтерфейсів та структури даних параметрів кожного інтерфейсу.

Щоб визначити критерій покриття вхідних даних, потрібно розбити їх на підмножини еквівалентних з точки зору даних. Є два основні способи визначення такої еквівалентності:

– виділення різних підтипів даних одного типу на основі практичних міркувань. Наприклад, ряд міркувань дозволяє розділити цілі числа на позитивні, негативні та 0. Наслідуючи інші міркування можна розбити їх на парні та непарні числа;

– поділ значень за певними правилами, що стосуються їхньої структури. Наприклад, цілі числа зазвичай видаються у двійково-додатковому форматі, всі негативні мають перший (останній) біт, що дорівнює 1, на відміну від невід’ємних. Тому розбиття на позитивні та негативні можна обґрунтувати структурою уявлення числа в машині. Можна ділити цілі числа на великі та невеликі за абсолютною величиною відповідно до того, чи є хоча б один біт рівний 1 серед перших 16. Тобто числа ≥ 65536 за абсолютною величиною оголошуються більшими, а менші – невеликими [39].

Для визначення класів еквівалентних складних даних зазвичай використовують їхню структуру та класи еквівалентності даних простих типів, з яких вони побудовані. Так, визначаючи розбиття на класи для об’єктів, у яких два цілих поля, досить природно оголосити еквівалентними об’єкти, у яких відповідні поля мають один знак або одночасно рівні нуль виходить лише 9 класів таких об’єктів.

Для більш складних даних використовують опис їхньої структури у вигляді граматики, щоб виділити класи еквівалентності або відповідні критерії покриття.

Наприклад, для документів, що описуються деякою контекстно вільною граматикою, можна визначити такі критерії покриття.

Критерій покриття правил – частка правил граматики, використаних для побудови тестових даних серед усіх її правил.

Критерій покриття альтернатив для кожного правила визначається, скільки є можливих альтернатив його розкриття, і замість 1 у визначенні попередніх критеріїв для всіх правил враховується це число, а для покритих – тільки кількість реалізованих альтернатив [39].

3.9 Критерії повноти на основі вимог

Базова ідея критеріїв покриття на основі вимог – якщо два тести перевіряють виконання тих самих обмежень, визначених у вимогах, швидше за все, вони або обидва виявлять помилку, або обидва виконаються успішно. Тому їх можна вважати еквівалентними.

Це неправда, як і базова ідея структурних критеріїв, але також дозволяє запровадити досить зручні критерії покриття.

Зазвичай вимоги оформляються як неформального тексту, організованого ієрархічно, тобто. з виділеними пунктами та підпунктами. У цьому випадку можна визначити критерії покриття вимог як частку тестових наборів, що перевіряються, найбільш детальних виділених пунктів вимог серед тих, які взагалі можна перевірити (іноді частину таких вимог неможливо перевірити взагалі, або за час, обмежений рамками проєкту).

Більше формальне визначення можна дати метриці покриття тверджень. Для її визначення з вимог виділяються елементарні твердження, що перевіряються, виконання кожного яких, взагалі кажучи, не пов’язане з

виконанням інших. Критерій визначається як частка перевірених в тестах таких тверджень стосовно всіх [38, 39].

Наприклад, нехай у вимогах до системи є така фраза: «Система повинна підтримувати виконання операцій читання, додавання, видалення та модифікації запису про клієнта, причому, при паралельній роботі кількох операторів тільки один з них у своїй сесії може видаляти та модифікувати вже наявний запис».

У ньому можна назвати такі твердження:

– «оператор може прочитати дані запису про клієнта»;

– «оператор може додати запис про клієнта»;

– «оператор може видалити запис про клієнта, якщо працює з нею один»;

– «оператор може модифікувати запис про клієнта, якщо працює з нею один»;

– «якщо кілька операторів працює із записом про клієнта, тільки один може його видалити»;

– «якщо кілька операторів працює із записом про клієнта, тільки один може її модифікувати».

Відповідно, вимірювати повноту покриття тестів можна ставленням кількості перевірених із цих тверджень до загальної кількості виділених.

Випадок, що часто зустрічається, – оформлення вимог у вигляді набору правил (або бізнес-правил). У цьому випадку критерій покриття правил вважають частку перевірених тестами правил серед усіх існуючих.

Це дозволяє визначити критерії покриття умов правил приблизно так, як це робиться для покриття умов, що використовуються в коді. Можна використовувати аналоги критеріїв покриття елементарних умов, умов та розгалужень, комбінацій умов або MC/DC. Використовувати аналог критерію покриття коротких диз’юнктів не можна, якщо тільки не впевнені, що в коді ці умови використовуються рівно в тому ж порядку, що практично ніколи не виконано [40].

В цілому критерії покриття на основі вимог мають важливу перевагу вже тому, що вони дозволяють враховувати вимоги при оцінці повноти тестування, так що неперевірена вимога автоматично означає не цілком повне тестування. Однак, на відміну від структурних критеріїв, для організації вимірювання покриття за вимогами потрібні серйозні додаткові зусилля, наприклад, вказівки кожного тесту, які вимоги він перевіряє [40].

3.10 Критерії повноти на основі припущень про помилки

Найбільш зручний на практиці спосіб виміру повноти тестування на основі явних гіпотез про можливі помилки – це метод визначення повноти тестів на основі виявлених мутантів.

В межах цього методу мови програмування, у якому написана тестована програма, визначається досить повний набір операторів мутації. Кожен такий оператор змінює текст програм, наприклад, видаляючи певну інструкцію, вставляючи нову інструкцію, замінюючи змінні у виразах інші змінні того ж типу або на константні вирази того ж типу, замінюючи оператори арифметичних дій +, –, *, / один на одного, замінюючи оператори логічних операцій один на одного та ін. [41]. Важливо, що після застосування будь-якого з операторів мутації синтаксично та семантично коректна програма залишається коректною.

Програма, одержувана із тестованої застосуванням одного оператора мутації, називається мутантом. При застосуванні більшої кількості операторів виходять мутанти другого і вищих порядків, але вони зазвичай не використовуються, тому що їх кількість навіть для невеликої програми дуже велика [42].

Ті мутанти, які еквівалентні за поведінкою вихідної програми, тобто. поводяться так само у всіх ситуаціях, викидають з отриманої безлічі мутантів. Після цього використовується критерій повноти тестів, що

визначається як частка мутантів, що виявляються тестами, серед тих, що залишилися.

На практиці набори мутантів зазвичай виходять досить великими, і доводиться витрачати значні зусилля, щоб відсіяти їх еквівалентні вихідної програмі, оскільки виявлення еквівалентності не можна автоматизувати повністю. В результаті критерій повноти на основі частки виявлених мутантів досить сильний, але її застосування є дуже трудомістким.

Ще одним аргументом проти використання мутантів служить те, що вони допомагають виявляти лише невеликі та випадкові помилки-друкарські помилки. Серйозна помилка в розумінні вимог найчастіше призводить не до зміни одного знака або пропуску однієї інструкції, а до втрати цілої групи інструкцій, які мали спрацьовувати в певних умовах, разом з умовами їх виконання. Виявити таку помилку за допомогою мутацій дуже важко.

ВИСНОВКИ

У рамках кваліфікаційної роботи розглядаються існуючі програмні інструменти автоматизації Також складене зразкове уявлення про існуючі рішення і почато розглядати можливість впровадження цих інструментів у цикл розробки. Здійснено докладний розгляд теоретичної основи створення програмних продуктів, зокрема з використанням технологій автоматизованого тестування. Описується організація процесу впровадження та підтримки циклу автоматизованого тестування. Проведено зіставлення циклу автоматизованого тестування циклу розробки програмного продукту.

Наведено огляд існуючих критеріїв якості та надійності програмних продуктів відповідно до ISO/IEC 9126. Також описано критеріїв якості тестування. Також йде огляд видів тестування. Для кожного виду дається характеристика застосування в умовах автоматизованого тестування. Можна помітити, що деякі важливі види тестування потребують ручного тестування та не піддаються автоматизації.

Описані дослідження, проведені з метою оцінити ефективність автоматизації тестування в розробці. Даються чисельні показники та умови, за яких такі показники ефективності досяжні. Визначено критерії повноти тестового покриття. Ці критерії дозволяють отримати певні гарантії щодо складених тестів.

Протягом усієї роботи даються рекомендації щодо впровадження автоматизованого тестування. Позначаються моменти, які слід розглянути докладніше і чого слід очікувати під час впровадження. Отже, основні завдання роботи було виконано.

Результати роботи апробовано у вигляді 2 тез доповідей під час VIII Міжнародної наукової конференції «Наука і практика, актуальні проблеми, інновації» [43] та IX Міжнародної наукової конференції «Тенденції розвитку сучасної науки і практика» [44].

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Куліков, С. С. (2015). Тестування програмного застосунку. Базовий курс.

2. М’ясников, А. В. (2020). Побудова моделі інформаційної системи для автоматизації тестування проникнення. Проблеми інформаційної безпеки. Комп’ютерні системи, (3), 32-39.

3. Шувалов, М. Е. (2021). Автоматизація тестування: підход до розробки та проблеми супроводу. Фундаментальні та прикладні дослідження молодих вчених (pp. 384-387).

4. Д’яков, Н. В., & Сахно, В. В. (2021). Аналіз процесу автоматизації тестування програмного продукту, (4-1), 437-439.

5. Кокарева, О. В., Жуков, Н. Н., & Сайпулаєва, Ж. С. (2021). Підготовка інженерів забезпечення якості для автоматизації тестування вебзастосунків. Сучасна освіта: традиції та, (S2-1), 93-97.

6. Анкудінов, И. Г. (2020). Автоматизація тестування проєктних комптецій персоналу. In Сучасні освітні технології у підготовці фахівців для мінерального сировинного комплексу (pp. 576-581).

7. Ніконов, В. А. (2020). Стратегії автоматизації тестування. In Innovation-2020 (pp. 294-298).

8. Тулфоров, Д. М. (2020). Автоматизація тестування вебзастосунків, використовуючи класифікатор типів елементів машинного навчання. Міжнародний журнал гуманітарних та природничих наук, (6-2).

9. Пожидаєв, Ю. К., & Міткевич, М. А. (2020). Автоматизація тестування мобільних застосунків зв допомогою. Інтернаука (19-1), 47-49.

10. Бєлянчиков, І. А. (2020). Автоматизація тестування користувального інтерфейсу програмного забезпечення на прикладі методу скриншотів. In Сучасні технології: актуальні питання, здобутки та інновації (pp. 68-72).

11. Корольова, Д. Ст (2020). Огляд методів автоматизації тестування та інструментів його. In Сучасні технології: актуальні питання, здобутки та інновації (pp. 30-34).

12. Китайгородський, П. С. (2021). Автоматизація тестування ETL-процесів та структури сховища даних.

13. Смолякова, К. В. (2021). Технологія RPA стосовно автоматизації тестування програмного забезпечення.

14. Анісімов, Ст І., Васильєв, С. А., Євдокимов, І. А., & Тарасова, О. Б. (2020). Огляд методів автоматизації тестування та документування серверного інтерфейсу, основаногго на архітектурі. Інформаційні технології у проєктуванні та виробництві, (2), 45-48.

15. Богер, Ст Ст (2020). Застосування інструментів для автоматизації тестування вебзастосунків.

16. Нестерова, О. А. (2020). Метдика автоматизації тестування мобільних затосунків на scrum-проєктах. Студентський форум (16-1), 52-56.

17. Баглюк, С. І., & Нечай, А. А. (2020). До питання про вибір вихідних даних при автоматизації тестування програм. Вісник українського нового університету. Серія: Складні системи: моделі, аналіз та управління, (4), 103-107.

18. Тулфоров, Д. М. (2020). Автоматизація тестування вебзастосунку використовуючи классифікатор типов елементів машиного навчання. Впровадження сучасних конструкцій та передових технологій у колійне господарство, 16(16), 122-129.

19. Таран, С. Н., & Щербина, І. О. (2020). Технології автоматизації тестування та Їх впровадження в процес створення ігрових. Вісник Адигейського державного університету. Серія 4: Природно-математичні та технічні науки, (4 (271)).

20. Сівков, Д. І. (2020). Програма для автоматизації тестування теоретичних знань шляхом повторення.

21. M. Ayaz Ahmad, Volodymyr Gorokhovatskyi, Iryna Tvoroshenko, Nataliia Vlasenko, Syed Khalid Mustafa (2021) The Research of Image Classification Methods Based on the Introducing Cluster Representation Parameters for the Structural Description, International Journal of Engineering Trends and Technology, 69(10), pp. 186-192

22. Марков, А. А., Скуднєв, Д. М., Ісаєва, А. В. (2020). Комплексна система тестування Web ресурсів. У пріоритетних напрямах розвитку російської науки (pp. 31-36).

23. Марінчук, О. С. (2020). Тестування різних сервісів API за допомогою інструмента Postman. Постулат (1).

24. Гороховатський В.О., Гадецька С.В. (2020) Статистичне оброблення та аналіз даних у структурних методах класифікації зображень (монографія), Харків, ФОП Панов А.Н., 128 с.

25. Гороховатський, В. О., Власенко, Н. В., & Рибалка, М. О. (2021). Застосування засобів хешування даних для прискорення класифікаційних рішень у структурних методах розпізнавання зображень.

26. Лоскутов, А. І., Кликов, Ст А., Ряхова, Є. А., Столяров, А. Ст, & Шестопалова, О. Л. (2021). Нейромереживий підхід до кластеризації контрольованих параметрів, як одного з етапів автоматизації процесу ідентифікації складних технічних об’єктів. Вимірювання. Моніторинг. Управління. Контроль, (1(35)).

27. Гороховатський, В. О., Пупченко, Д. В., & Стяглик, Н. І. (2020). Дослідження трансформацій простору даних при навчанні мережі кохонена у методах структурної класифікації зображень.

28. Gadetska, S. V., & Gorokhovatskyi, V. O. (2018). Statistical measures for computation of the image relevance of visual objects in the structural image classification methods. Telecommunications and Radio Engineering, 77(12).

29. Якушина, Ст Є., & Лінарт, А. Ст (2020). Розвиток технологій тестування програмного забезпечення. Особливості проведення у сучасних

проєктах автоматизації. In Фундаментальні та прикладні дослідження молодих вчених (pp. 335-338).

30. Гороховатський, В. О., Гадецька, С. В., Стяглик, Н. І., & Власенко, Н. В. (2020). Класифікація зображень на підставі ансамблю статистичних розподілів за класами еталонів для компонентів структурного опису.

31. Гороховатський, В. О., Стяглик, Н. І., & Царевська, В. В. (2021). Комбінаційний метод прискореного метричного пошуку даних у задачах класифікації зображень.

32. Саїтов, А. Ст, & Блохін, М. Ст (2020). Автоматизоване тестування у «1С: Підприємство 8.3». Наукові обрії, (4), 252-258.

33. Бахвалова, З. А., & Камишова, Є. А. (2020). РОЛЬ ФОРМАЛІЗАЦІЇ ВИМОГ У ТЕСТУВАННІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ. Інформаційні та математичні технології в науці та управлінні, (1 (17)).

34. Шедько, В. В. (2020). Тестирование, верификация и аттестация программного обеспечения.

35. Новгородцева, Т. Ю., Бурдуковська, А. Ст, Іванова, Є. Н., Дядькін, Ю. А., & Лісников, І. Н. (2021). Моделювання структурних елементів контингенту студентів з урахуванням кластерного аналізу. Сучасна педагогічна освіта, (5), 75-80.

36. Гороховатский В.А. Распознавание изображений в условиях неполной информации/ В.А. Гороховатский. Х.: ХНУРЭ, 2003. 112 с.

37. Kobylin O., Gorokhovatskyi V., Tvoroshenko I., and Peredrii O. (2020) The application of non-parametric statistics methods in image classifiers based on structural description components, *Telecommunications and Radio Engineering*, 79(10), pp. 855-863.

38. Daradkeh, Y.I., Tvoroshenko, I., Gorokhovatskyi, V., Latiff, L.A., and Ahmad, N. (2021) Development of Effective Methods for Structural Image

Recognition Using the Principles of Data Granulation and Apparatus of Fuzzy Logic, *IEEE Access*, 9, pp. 13417-13428.

39. Daradkeh, Y.I., Gorokhovatskyi, V., Tvoroshenko, I., Gadetska, S., and Al-Dhaifallah, M. (2021) Methods of Classification of Images on the Basis of the Values of Statistical Distributions for the Composition of Structural Description Components, *IEEE Access*, 9, pp. 92964-92973.

40. Gorokhovatskyi, V.O., Tvoroshenko, I.S., and Peredrii O.O. (2020) Image classification method modification based on model of logic processing of bit description weights vector, *Telecommunications and Radio Engineering*, 79(1), pp. 59-69.

41. Gorokhovatskyi Volodymyr, and Tvoroshenko Iryna (2020) Image Classification Based on the Kohonen Network and the Data Space Modification. In CEUR Workshop Proceedings: Computer Modeling and Intelligent Systems (CMIS-2020). 2608. pp. 1013-1026.

42. Gorokhovatskyi V. Quantization of the Space of Structural Image Features as a Way to Increase Recognition Performance / Gorokhovatskyi Volodymyr, Putyatin Yevgenyi, Gorokhovatskyi Oleksii, Peredrii Olena // The Second IEEE International Conference on DataStream Mining & Processing 21-25 August 2018, Lviv, Ukraine. – pp. 464 – 467.

43. Василенко І. С. (2021). Аналіз способів автоматизації для тестування програмного забезпечення, The VIII International Science Conference «Science and practice, actual problems, innovations», November 09 – 12, 2021, Amsterdam, Netherlands. 525 p.

44. Василенко І. С. (2021). Порівняльний аналіз типів тестування програм, The IX International Science Conference «Trends of development modern science and practice», November 16 – 19, 2021, Stockholm, Sweden. 588 p.