Міністерство освіти і науки України

Харківський національний університет імені В.Н. Каразіна

Факультет комп’ютерних наук

**КУРСОВА РОБОТА**

з дисципліни «Математичні методи та технології тестування і верифікації програмного забезпечення»

Тема «Дослідницьке тестування (Exploratory testing)»

Виконав студент 2 курсу

групи КС-21 спеціальності:

122 – Комп’ютерні науки

Безрук Юрій Русланович

Перевірив:

Доцент Нарєжній О. П. .......... ...........

Харків – 2020

**ЗМІСТ**

[ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ 3](#_Toc42266312)

[ВСТУП 5](#_Toc42266313)

[РОЗДІЛ 1 ОСНОВНІ ПОЛОЖЕННЯ. ІНТУІТИВНЕ ТЕСТУВАННЯ 7](#_Toc42266314)

[1.1 Проблеми автоматизації 7](#_Toc42266315)

[1.2 Інтуїтивне тестування 7](#_Toc42266316)

[РОЗДІЛ 2 ДОСЛІДНИЦЬКЕ ТЕСТУВАННЯ 10](#_Toc42266317)

[РОЗДІЛ 3 ПЕРЕВАГИ ТА МЕТОДИ ДОСЛІДНИЦЬКОГО ТЕСТУВАННЯ 13](#_Toc42266318)

[3.1 Переваги 13](#_Toc42266319)

[3.2 Методи дослідницького тестування 14](#_Toc42266320)

[РОЗДІЛ 4 ВИКОРИСТАННЯ ДОСЛІДНИЦЬКОГО ТЕСТУВАННЯ 17](#_Toc42266321)

[4.1 Методології 17](#_Toc42266322)

[4.2 Вимоги до тестувальників 18](#_Toc42266323)

[РОЗДІЛ 5 ПРИКЛАД ЗАСТОСУВАННЯ. ПРОГРАМНА РЕАЛІЗАЦІЯ 20](#_Toc42266324)

[ВИСНОВКИ 25](#_Toc42266325)

[ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 26](#_Toc42266326)

[ДОДАТОК А ПРИКЛАД КОДУ РЕАЛІЗАЦІЇ 27](#_Toc42266327)

# ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ, ТЕРМІНІВ

ПЗ – програмне забезпечення.,

QA – quality assurance,

Тестування – це процес, що складається з усіх статичних та динамічних заходів життєвого циклу, пов'язаний із плануванням, підготовкою та оцінкою компонента чи системи та пов'язаних з ними робочих продуктів, щоб визначити, що вони відповідають визначеним вимогам, продемонструвати, що вони відповідають цілі та виявити дефекти [1].

Debugging – процес пошуку, аналізу та усунення причин збоїв у компоненті чи системі [1].

Забезпечення якості – діяльність зосереджена на забезпеченні впевненості, що вимоги до якості будуть виконані [1].

Скрипт – послідовність інструкцій щодо виконання тесту.

Тест-кейс – сукупність передумов, входів, дій (де це можливо), очікуваних результатів та пост-умов, розроблених на основі умов тесту [1].

Фідбек (з анг., feedback – зворотній зв’язок) – форма спілкування, чи отримання повідомлень, відповідь, звіт.

Баг (з анг., bug – жук) – помилка, недосконалість або дефект робочого виробу, коли він не відповідає його вимогам чи характеристикам.

Спринт - ітерація в SCRUM, в ході якої створюється інкремент бізнес-продукту. Жорстко фіксований за часом. Тривалість одного спринту від 1 до 4 тижнів.

Тест план (Test Plan) – це документ, що описує весь обсяг робіт з тестування, починаючи з опису об'єкта, стратегії, розкладу, критеріїв початку і закінчення тестування, до необхідного в процесі роботи обладнання, спеціальних знань, а також оцінки ризиків з варіантами їх вирішення [1].

Тест дизайн – це етап процесу тестування ПО, на якому проектуються і створюються тестові сценарії (тест кейси), відповідно до визначених раніше критеріями якості та цілями тестування.

Test-driven development – методика розробки програмного забезпечення, в якій тестові приклади розробляються і часто автоматизуються, а потім програмне забезпечення розробляється поступово для передачі цих тестових випадків [1].

# ВСТУП

Протягом десятиліть розвитку розробки ПО до питань *тестування* і забезпечення якості підходили дуже і дуже по-різному. Саме значення поняття тестування змінювалося з часом, змінювали і методики, що призвело до появи різноманітних методів роботи з тестуванням програмного забезпечення і забезпеченням контролю якості. Взагалі ж, можна виділити кілька основних «епох тестування ».

У 50-60-х роках минулого століття процес тестування був дуже формалізований, відділений від процесу безпосередньої розробки ПО і «математизований». Фактично тестування являло собою швидше налагодження програм (англ. *debugging*). Існувала концепція так званого «Вичерпного тестування (англ. exhaustive testing)» - перевірки всіх можливих шляхів виконання коду з усіма можливими вхідними даними. Але дуже скоро було з'ясовано, що вичерпне тестування неможливо, тому що кількість можливих шляхів і вхідних даних стрімко зростає, а також при такому підході складно знайти проблеми в документації.

У 70-х роках фактично народилися дві фундаментальні ідеї тестування: тестування спочатку розглядалося як процес доказу працездатності програми в деяких заданих умовах (англ. positive testing), а потім - строго навпаки: як процес докази непрацездатності програми в деяких заданих умовах (англ. negative testing). Це внутрішнє протиріччя не тільки не зникло з часом, але і в наші дні багатьма авторами абсолютно справедливо відзначається як дві взаємодоповнюючі мети тестування.

Відзначимо, що «процес доказу непрацездатності програми» цінується трохи більше, тому що не дозволяє закривати очі на виявлені проблеми.

У 80-х роках відбулося ключова зміна місця тестування в розробці програмного забезпечення (ПЗ): замість однієї з фінальних стадій створення проекту тестування стало застосовуватися протягом всього циклу розробки, що дозволило в дуже багатьох випадках не тільки швидко виявляти й усувати проблеми, але навіть передбачати і запобігати їх появі.

У 90-х роках відбувся перехід від тестування як такого до більш всеосяжного процесу, який називається «*забезпечення якості»* (англ. quality assurance, QA), який охоплює весь цикл розробки ПЗ і зачіпає процеси планування, проектування, створення і виконання тест-кейсів, підтримку наявних тест-кейсів і тестових оточень. Тестування вийшло на якісно новий рівень, який природним чином привів до подальшого розвитку методологій, появи досить потужних інструментів управління процесом тестування і інструментальних засобів автоматизації тестування, вже цілком схожих на своїх нинішніх нащадків.

У нульові роки нинішнього століття розвиток тестування тривав в контексті пошуку все нових і нових шляхів, методологій, технік і підходів до забезпечення якості. Серйозний вплив на розуміння тестування мала поява гнучких методологій розробки та таких підходів, як «розробка за допомогою керуванням тестуванням (англ., *test-driven development*, TDD)». Автоматизація тестування вже сприймалася як звичайна невід'ємна частина більшості проектів, а також стали популярні ідеї про те, що найголовніша частина процесу тестування – не відповідність програми вимогам, а її здатність надати кінцевому користувачеві можливість ефективно вирішувати свої завдання.

Тому не дивно, що сучасне тестування можна описати так: гнучкі методології і гнучке тестування, глибока інтеграція з процесом розробки, широке використання автоматизації, колосальний набір технологій та інструментальних засобів, кросфункціональность команди (коли тестувальник і програміст у чому можуть виконувати роботу один одного) [2].

Але поряд із розвитком технологій і методологій, виникали і нові проблеми і питання, що потребували вирішення. І автоматизоване тестування не могло їх усі вирішити. Адже дії виключно за *тест-планом* не можуть спрогнозувати усі варіанти розвитку подій.

# **РОЗДІЛ 1 ОСНОВНІ ПОЛОЖЕННЯ. ІНТУІТИВНЕ ТЕСТУВАННЯ**

## 1.1 Проблеми автоматизації

Автоматизоване тестування часто стикається із проблемою того, що обмежені, заздалегідь передбачені сценарії за *тест-дизайном* не можуть обробити усі можливі критичні ситуації під час роботи з програмним забезпеченням, більш того, передбачити велику кількість різноманітних помилок на вже готовому додатку і протестувати їх автоматично взагалі дуже складно. Натомість, пряма взаємодія тестувальника з програмою так, як це буде робити користувач, дарує свободу взаємодії і тестування з можливістю отримання більш детального розуміння тих чи інших аспектів продукту.

Отже, об’єктом даної роботи вважаємо процес тестування і проблеми, що з’являються під час його виконання, а також шляхи їх вирішення.

Предметом даної роботи є інтуїтивні способи тестування (Ad-hoc, explorary), які надають тестувальнику більших повноважень у вирішенні проблем, які часто не може вирішити автоматизація.

Тому, метою даної роботи є ознайомлення та поглиблення знать у області ручного тестування, вивчення сучасних дослідницьких методів контролю якості.

## 1.2 Інтуїтивне тестування

Перший крок у напрямку дослідження програмного продукту під час налагодження робить так зване інтуїтивне тестування. (англ., ad-hoc testing).

Аd-hoc testing – це вид тестування, який виконується без підготовки до тестів, без визначення очікуваних результатів, проектування тестових сценаріїв. Це неформальне, імпровізаційне тестування. Він не вимагає ніякої документації, планування, процесів яких слід дотримуватися у виконанні. Також на даний вид тестування не пишуться тест-кейси, що в свою чергу може викликати певні труднощі в спробах відтворити дефект в системі. Такий вид найчастіше може дати відразу більше результату ніж тестування за заздалегідь визначеними сценаріями. Це обумовлено тим, що тестувальник на перших кроках приступає до тестування основного функціоналу і виконує нестандартні перевірки, точніше деякі з його перевірок будуть нестандартними [3].

Найчастіше таке тестування виконується коли мало часу на точне і послідовне тестування. При цьому тестувальник покладається на своє загальне уявлення про програму і здоровий глузд



Рисунок 1.1 – основні принципи інтуїтивного тестування

Тестування ad-hoc має сенс тільки в разі якщо тестувальник володіє загальною інформацією про продукт. Якщо людина зовсім не знатиме продукт, то витратить час на його вивчення, особливо якщо проект дуже складний і великий. Тому потрібно мати гарне уявлення про цілі проекту, його призначення, основні функції і можливості. А далі вже можна приступати до ad-hoc testing.

Ad-hoc-тести проводяться після формального тестування програми. Ad-hoc методи є найменш формальним типом тестування, оскільки це НЕ структурований підхід. Отже, дефекти, виявлені за допомогою цього методу, важко відтворити, оскільки для цих сценаріїв немає узгоджених тестових випадків.

Тестувальник тестує в довільному порядку без дотримання специфікацій / вимог. Отже, успіх тестування Ad-hoc залежить від можливостей тестувальника, який проводить тестування. Тестер повинен знаходити дефекти без будь-якого належного планування та документації, ґрунтуючись виключно на інтуїції [3].

# РОЗДІЛ 2 ДОСЛІДНИЦЬКЕ ТЕСТУВАННЯ

Але інтуїтивне тестування також не завжди є хорошим рішенням. Адже воно безкорисне, якщо тестувальник погано знає програмний продукт, чи він занадто складний. У цьому випадку, тестувальнику потрібно не просто намагатися зламати систему, шукаючи випадкові помилки, а досліджувати програму, поглиблюватись у її вивченні, виконуючи послідовні висновки і додаючи нові тести. Так виникає нова галузь тестування – так зване «дослідницьке тестування».

Дослідницьке тестування (англ. Explorary testing) – це підхід до тестування програмного забезпечення, який підкреслює залучення тестера, нескриптованість тестування та свободу тестера у виборі шляху тестування. Вповноважені вибирати те, на чому зосередитись, і як оцінити програмне забезпечення, ці тестери можуть проводити випробування в тих областях, які мають потенційні проблеми або які вони вважають (на основі питань в інших областях), слід перевірити [4].

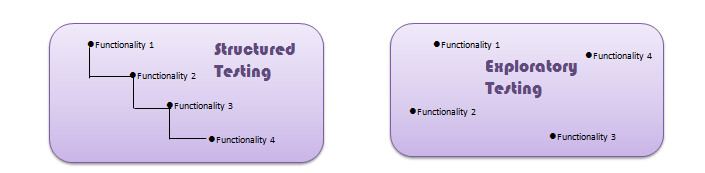


Рисунок 2.1 – відмінності підходів скриптованого та дослідницького тестування [5]

Тестовий дизайн, точні кроки тестування та очікувані результати дослідницьких випробувань не визначаються заздалегідь. Таким чином, поки *скриптові* тестувальники слідують за набором *тест-кейсів*, а шляхи в програмному забезпеченні, яке вони перевіряють, були заздалегідь розглянуті та прокладені для них, дослідницькі тестувальники отримують свої результати, не дотримуючись заздалегідь визначеного маршруту [4].

Різниця між ad hoc і exploratory testing в тому, що теоретично, ad hoc може провести будь-хто, а для проведення exploratory необхідно майстерність і володіння певними техніками. Зверніть увагу, що певні техніки це не тільки техніки тестування.

Цей підхід до тестування оцінює досвід, інтелект, творчість, навчання та навіть інтуїцію тестувальників під час пошуку тестового дизайну, виконання тесту та тестування однією людиною одночасно. Дослідницькі тестувальники звільняються від документування своїх кроків заздалегідь, що дає їм можливість включити те, що вони дізналися, і як вони це дізналися безпосередньо в наступний тест.

Звичайно, це не означає, що тестери роблять все, що завгодно. Дослідницьке тестування - це не випадкове натискання кнопок. Однак, усуваючи строгі сценарії тестування, дослідницьке тестування дозволяє досягти більш цілісного підходу до проектування, виконання та аналізу тестів [4] .

Дослідження - це систематичне розслідування з метою встановлення фактів, тобто це процес вивчення чого-небудь.

Тому, кажучи про досвід, ми дуже часто маємо на увазі саме знання чого-небудь. Знання - це вже вміння чогось, але дослідження - це вивчення. Говорячи про дослідницький тестуванні ми повинні відкинути всі наші знання тестування, досвід роботи з системою, технік тест-дизайну і дивитися на системи новим, абсолютно не «замилений» поглядом.

Дослідницьке тестування - це в першу чергу вміння мислити, бажання звертати увагу на різні аспекти системи, відкинувши наші знання і досвід роботи з даною системою. Адже саме досвід часто не дозволяє нам бачити здавалося б такі прості речі, на які звичайний тестувальник відразу вкаже при тестуванні [4].

Основними пунктами, що використовуються у дослідницькому тестуванні є 4 циклічно повторюваних дії на кожному кроці:

1. Learning. Тестувальник набирається знань про продукт, який досліджує, що допомагає йому краще розуміти його можливості.
2. Design. Тестувальник встановлює план тест-дизайну на поточний крок.
3. Execution. Тестувальник виконує поточний тест-план в одну дію.
4. Feedback. Тестувальник отримує результати роботи та необхідну інформацію про поведінку продукту у даній ситуації.

Далі, на основі отриманої інформації тестувальник починаю новий цикл тестування за тими ж кроками, але вже з новими цілями, визначити які йому допомогла нова інформація.



Рисунок 2.2 – основні частини дослідницького тестування [6]

# РОЗДІЛ 3 ПЕРЕВАГИ ТА МЕТОДИ ДОСЛІДНИЦЬКОГО ТЕСТУВАННЯ

## 3.1 Переваги

Дослідницьке тестування розвивається в ході процесу, оскільки тестери мають можливість для вивчення сценаріїв, пов’язаних із пристроями та платформами, які вони використовують, в момент тестування. Завдяки цьому тестери можуть шукати всілякі можливі проблеми та помилки, пов’язані з платформою програмного забезпечення. Наприклад, програма, яка функціонує, як очікувалося, у мобільному браузері Chrome на iOS, може не так добре працювати на пристрої Android (навіть того ж браузера). І це лише один погляд на всі можливі проблеми з платформою, на які продукт може зіткнутися на різних мобільних або веб-програмах, програмному забезпеченні smartwatch чи будь-де ще [4].

Дослідницький тестер може і враховує проблеми кожного браузера, версію операційної системи, конфігурацію обладнання та інші важливі фактори. Цей тип тестування є чудовим у виявленні проблем на платформі.

До інших переваг дослідницького тестування відносяться:

* Скорочений термін підготовки тесту - Оскільки ця методика тестування потребує меншої кількості документації, тестувальник може скоріше почати випробування.
* Одночасне навчання - кваліфіковані тестувальники можуть використовувати свої особисті навички, досвід та знання в процесі тестування та дізнаватися про поведінку системи під час тестування. Це означає, що вони можуть розробляти, виконувати та навіть вдосконалювати сценарії тестування в режимі реального часу, спостерігаючи, як сценарії або дії пов'язані зі швидкістю виконання, яку неможливо повторити автоматизованим тестуванням.
* Динамічний *фідбек* - тестери звільняються від традиційного підходу, що базується на тест-кейсах, вони в змозі зосередити свої зусилля на ключових або проблемних областях за потребою. Це може дати розробникам дуже необхідний внесок у якість продукції або можливі проблеми на постійній та більш негайній основі.
* Виявляє критичні *баги* - сценарійні тести певною мірою обмежені у цьому плані. Розробники створюють кейси на основі того, що, на їхню думку, може піти не так. З іншого боку, дослідницькі тестуваньники думають інакше і вільніше досліджують проект, виявляючи важкодоступні дефекти.

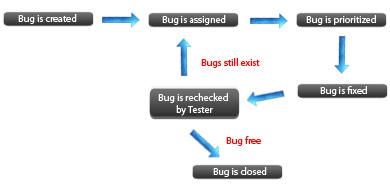


Рисунок 3.1 – життєвий цикл багу

## 3.2 Методи дослідницького тестування

Дослідницьке тестування підтримує випуск програмного забезпечення вищої якості. Як? Витягуючи тестерів та програмне забезпечення з обмежень автономного тестування лабораторії. Без обмеженого контрольованого середовища тестування дослідники-тестери можуть гарантувати, що програми та веб-сайти працюють у реальних умовах – і таким чином, досліджують, як реальні користувачі хотіли б ними користуватися.

Підтвердження того, що програма або програмне забезпечення працює без проблем всередині вашого офісу, не гарантує, що вона працюватиме так весь час і для всіх. Ваші користувачі можуть використовувати ваш продукт вдома, на роботі, в дорозі, з поганим з’язком, у сільській чи міській місцевості, на мобільних, ноутбуках, настільних пристроях або навіть пристроях Інтернету речей (IoT).

Ось чому методи розвідувальних випробувань зосереджені на виявленні:

- Мети продукту: що є основною задачею, яку виріб повинен вирішувати?

- Основних функцій - які основні функції виконує виріб?

- Сфер потенційної нестабільності - які функції, швидше за все, виходять з ладу, порушують стандарти стабільності, є ненадійними або мають негативні побічні ефекти?

Далі йде фактичне тестування кожної функції та запис проблем, що зустрічаються. Мета - не витрачати багато часу на вивчення всіх можливих шляхів, поки не будуть виявлені дефекти. Старанний тестувальник здійснить глибше занурення у виявленні проблем, які складніше знайти інакше. Починаючи з ознайомлення з усім, що може піти не так під час розробки програмного забезпечення, ці тестувальники зосереджуються на тому, як користувачі можуть перервати, пошкодити, зірвати, або зламати систему. Тим не менш, дослідницькі випробування - це що завгодно, крім того, щоб зробити безпідставні здогадки. Як писав Джеймс Бах у часто цитованому документі про дослідницькі випробування, головна директива цієї методології - «бути продуманою та методичною».

На рисунку 3.2 зображено, як у результаті роботи дослідницького тестувальника сегментується та вивчається задача на прикладі веб-сайту.

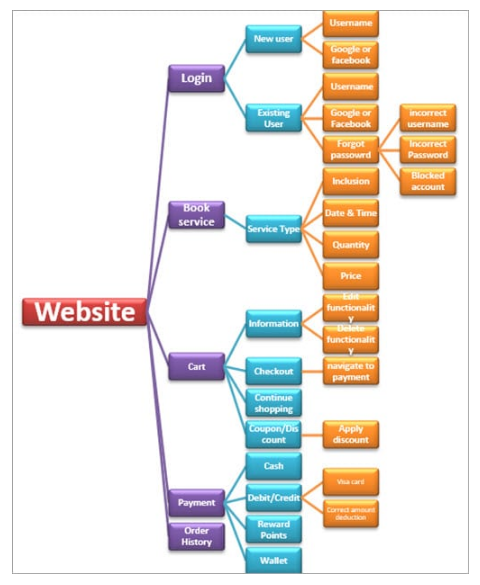


Рисунок 3.2 – приклад огляду задачі дослідницьким тестувальником [7]

# РОЗДІЛ 4 ВИКОРИСТАННЯ ДОСЛІДНИЦЬКОГО ТЕСТУВАННЯ

## 4.1 Методології

Бах разом із Джемом Канером був одним із піонерів концепції дослідницьких випробувань у 90-х роках. Вони раніше за інших побачили цінність процедур коли тестери знайомилися з продуктом та отримували свободу, необхідну для розробки тестових випадків, про які розробники та традиційні методи тестування можуть не замислюватися перед розробкою автоматизованого програмного забезпечення для тестування. Як результат, коли дослідницькі методи випробувань доповнюють інші методи, поліпшується якість тестування. Це, в свою чергу, означає, що випускається продукція більш високої якості та з кращим досвідом користувача, що призводить до успіху програмного забезпечення.

Дослідницьке тестування підходить для гнучких методологій. Вичерпне автоматизоване тестування складно здійснити за шкалою стиснених *спринтів*. Це не означає, що розвідувальні роботи повинні замінити автоматизовані тестування - одне може ефективно підтримувати інше, але розвідувальні випробування частіше здійснюють, коли зарано використовувати автоматизацію випробувань.

В методології дослідницького тестування замість того, щоб вимагати повної документації та письмових сценаріїв тест-кейсів, експерт-тестувальник може зануритися в розробку продукту та отримати загальне уявлення про його функціональність. Тестувальник здатний проектувати і тестувати одночасно. Це визнає його здатність впливати на правильні тести, які потрібно зробити в потрібний час [4].

Це творчий вид тестування, який може краще йти в ногу з вимогами спритного процесу. Ця методологія охоплює постійне покращення та вдосконалення. Тестер розробляє завдання, розробляє тести для його виконання та проводить тестування. Якщо питання тестування залишаються без відповіді, тестер може налаштувати процес, поки тести не повернуть результати, пропонуючи зрозуміти первісну мету.

## 4.2 Вимоги до тестувальників

Дослідницьке тестування не стосується того, щоб мимоволі клікати по програмі. Що робить певних людей чудовими тестувальниками? Існує кілька атрибутів:

Запал - дослідницьке тестування не визначає заздалегідь що буде тестуватися та як, тому вправні дослідники завжди зацікавлені. Вони хочуть дізнатися все, хочуть знати, що стоїть за цією кнопкою або що ховається в цій функції. Їх потрібно спонукати до того, щоб дізнатися більше, оскільки вони займаються самостійно, коли справа стосується перевірки різних частин веб-сайту чи програми та розслідування його куточків.

Гнучкість - вправні дослідницькі тестувальники - це грамотні розробники тестів, які повинні поглиблюватись у вивченні додатку, який тестують: якщо розслідування певної частини програмного забезпечення виявить помилку, їм потрібно мати можливість імпровізувати, створюючі нові тести, які шукають пов'язані проблеми та помилки. дослідницькі тестувальник, який добре розробляє експромтні додаткові тести, також може визначити масштаб помилки або принаймні описати її вплив на додаток, що полегшує виправлення розробником [4].

Спостережливість – допитливість недостатня сама по собі. тестувальники повинні не тільки бути зацікавленими, але й мати можливість помічати будь-які відхилення та документувати їх.

Цілісність – дослідницьке тестування повинно враховувати та записувати все, що може впливати на процес досліду. Це може бути ряд факторів, таких як: wifi чи сотовий зв’язок, потужність сигналу, рівень заряду акумулятора або навіть шум навколишнього середовища. Вправний дослідницький тестувальник розуміє потребу в додатковому дослідженні і повністю враховує всі шляхи, які слід розглянути, а не приймає рішення за одним сценарієм [4].

Відмінний комунікатор – хтось може бути розумним, допитливим та вмілим у пошуку дефектів, але цього мало, якщо він не може повідомити розробникам свої результати тестів. Без окремого тест-дизайнера, що взаємодіє з менеджером продукту, щоб заздалегідь визначити окремі етапи тестування та очікувані результати, тестери несуть відповідальність за пояснення ідеї даного тесту, обставин, що його оточують, а потім результатів, які вони отримують.

Кращі дослідницькі тестувальники знають, коли вживати слова, коли використовувати скріншоти та коли знімати відео. Вони враховують точку зору людей, які будуть використовувати їх роботу: розробників, QA-інженерів і продукт-менеджерів і надають інформацію, яка потрібна цим людям [4].

# РОЗДІЛ 5 ПРИКЛАД ЗАСТОСУВАННЯ. ПРОГРАМНА РЕАЛІЗАЦІЯ

Припустимо, що нашою задачею є протестувати роботу системи порівняння сайту pn.com.ua [8]. Зайдемо на сторінку з ноутбуками. Згідно з основними засадами explorary testing, ми повинні спочатку дослідити область. Тож, маємо: до механізму порівняння на сайті відноситься кнопка «додати у порівняння» під кожним товаром. Тому спочатку тестуємо її. Що відбудеться під час її натискання? Як зміниться сама кнопка?

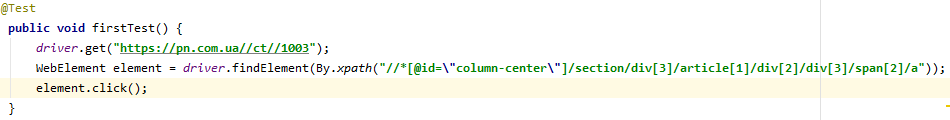


Рисунок 5.1 – початкове дослідження елементу

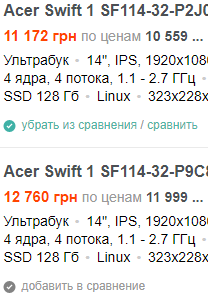


Рисунок 5.2 – порівняння нажатої та ненажатої конпоки.

Тут у тестувальника одразу виникає питання: чи працюють обидві ці кнопки? І чи можна додати до порівняння лише один товар? Записуємо це наступними тестами.

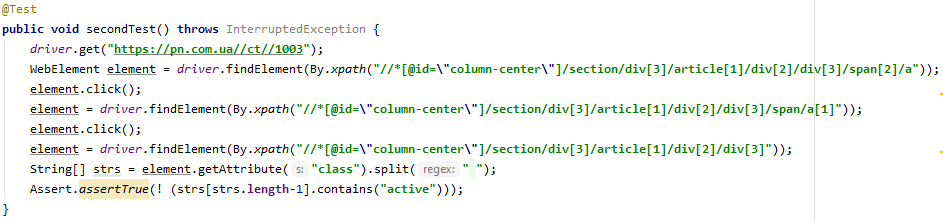


Рисунок 5.3 – другий тест системи

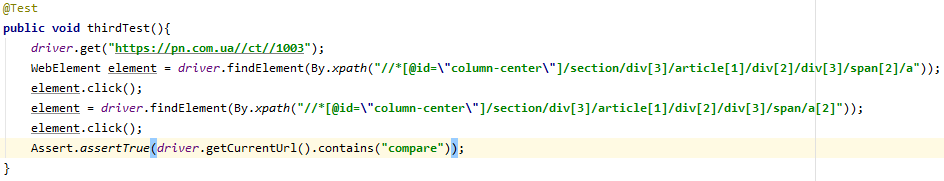


Рисунок 5.4 – третій тест системи



Рисунок 6.4 – лише один товар у порівнянні, за умовою

Обидва тести завершилися позитивно. Крім того ми побачили ще один функціонал, пов’язаний з порівнянням – сторінка порівняння. Потрібно також протестувати і її. Основні кнопки працюють як і повинні. Але головна задача дослідницького тестування – перевіряти особливі варіанти роботи, які зазвичай не передбачені тест-планами. Наприклад, ситуація з порівнянням одного елементу. Те що таке порівняння може існувати на сайті, ми перевірили. А щодо прихованого функціоналу? Наприклад, перевіримо, чи справді звільняється у пам’яті масив товарів порівняння, коли ми видаляємо з нього усі товари? Для цього додаємо до порівняння один товар, заходимо на порівнюючу сторінку, обираємо товар та натискаємо «видалити». Після цього перевіряємо розмір масиву – він повинен дорівнювати нулю.

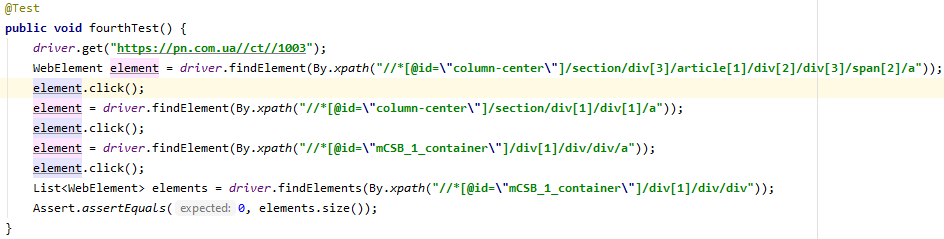


Рисунок 5.5 – четвертий тест-кейс

Після цього помічаємо кнопку сортування. Перевірити слід і її. Тобто, чи змінять елементи свої позиції при зміні порядку сортування. Для цього обираємо 4 товари, переходимо до порівняння. Зберігаємо список елементів до переліку, змінюємо порядок сортування з «за назвою» на «від дешевих» і зберігаємо інший перелік елементів. А потім циклічно перевіряємо елементи масиву. Отримаємо результат: елементи змінили розташування.



Рисунок 5.6 – п’ятий тест-кейс

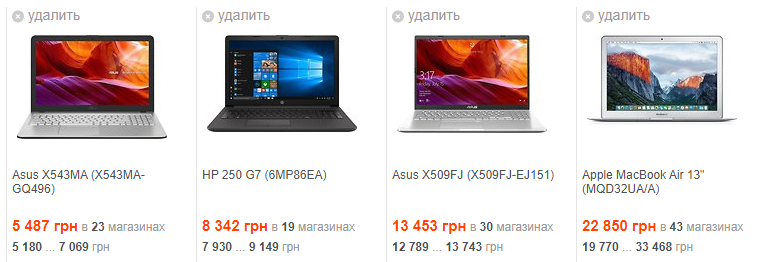
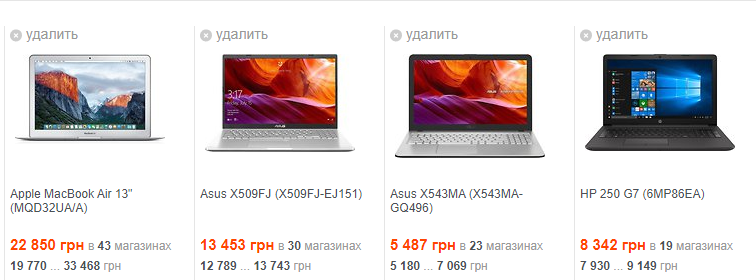


Рисунок 5.7 – елементи до та після зміни сортування

Ось таким чином дослідницьке тестування і відрізняється від скриптового – прямо по ходу виконання тестування вирішували, яким буде наш наступний крок, оцінюючи можливості продукту, які відкриваються на попередньому. Повний код тестів наведений у додатку А.

# ВИСНОВКИ

Таким чином, у ході виконання даної роботи було розглянуто основні положення, теоретичні та практичні використання дослідницького тестування. Докладно розглянутий процес розвитку тестування як методики через десятиліття, функціональний розвиток та розвиток підходів а також понять. Розібрано основні переваги та недоліки вільного тестування (інтуїтивного, дослідницького) перед скриптово-автоматизованим. Також наголошено на основних відмінностях між Ad-hoc та Explorary тестуванням, показано різницю у їх заповадженні.

У роботі були розглянуті основні вимоги до дослідницьких тестувальників і пропозиції по їх самовдосконаленню, наведено характерні риси та вміння, якими вони повинні володіти.

Під час написання роботи у якості прикладу дослідницького тестування було створено відповідне програмне забезпечення для тестування обраного ресурсу та продукту (розглядалося на прикладі веб-сторінки pn.com.ua), яким було вибрано систему порівняння товарів веб-ресурсу с усіма її можливостями. На мою думку, на базі створених тестів у майбутньому можна створити більш розширену тестову систему програмного забезпечення, з використанням описаних у роботі необхідних умов.

# ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. ISTQB Glossary // https:// glossary.istqb.org/en/search/test-driven%20 development. Дата звернення: 01.06.2020.
2. Куліков С.С., Тестирование программного обеспечения. Базовый курс, 2-е издание, М.: ОДО «Четыре четверти», 2017, 8 с. Дата звернення: 01.06.2020.
3. Ad-Hoc testing – QA Evolution. // https:// qaevolution.ru/testirovanie-po/

vidy-testirovaniya-po/ad-hoc-testing/ Дата звернення: 02.06.2020.

1. Exploratory testing – Test IO. // https:// test.io/exploratory-testing/. Дата звернення: 03.06.2020 .
2. What is Exploratory testing? – Guru99 // https://www.guru99.com/

exploratory-testing.html. Дата звернення: 02.06.2020.

1. Исследовательское тестирование и исследовательские методы – Software -testing. // https:// www.software-testing.by/blog/

exploratory-testing-exploratory-tours/ Дата звернення: 03.06.2020.

1. What is Exploratory testing in software testing? – SoftwareTestingHelp. // https:// www.softwaretestinghelp.com/what-is-exploratory-testing/. Дата звернення: 05.06.2020.
2. Все цены Харькова (прайс-навигатор) // https:// pn.com.ua/. Дата звернення: 04.06.2020.

# ДОДАТОК А ПРИКЛАД КОДУ РЕАЛІЗАЦІЇ

Лістинг 1 – код класу тестування.

**import** org.junit.AfterClass;  
**import** org.junit.Assert;  
**import** org.junit.BeforeClass;  
**import** org.junit.Test;  
**import** org.openqa.selenium.By;  
**import** org.openqa.selenium.WebDriver;  
**import** org.openqa.selenium.WebElement;  
**import** org.openqa.selenium.opera.OperaDriver;  
  
**import** java.util.List;  
  
**public class** WebTesting {  
 **static** WebDriver *driver*;  
  
 @BeforeClass  
 **public static void** start(){  
 System.*setProperty*(**"webdriver.opera.driver"**, **"operadriver\_win64\\operadriver.exe"**);  
 *driver* = **new** OperaDriver();  
 }  
 @Test  
 **public void** firstTest() {  
 *driver*.get(**"https://pn.com.ua//ct//1003"**);  
 WebElement element = *driver*.findElement(By.*xpath*(**"//\*[@id=\"column-center\"]/section/div[3]/article[1]/div[2]/div[3]/span[2]/a"**));  
 element.click();  
 }  
 @Test  
 **public void** secondTest() **throws** InterruptedException {  
 *driver*.get(**"https://pn.com.ua//ct//1003"**);  
 WebElement element = *driver*.findElement(By.*xpath*(**"//\*[@id=\"column-center\"]/section/div[3]/article[1]/div[2]/div[3]/span[2]/a"**));  
 element.click();  
 element = *driver*.findElement(By.*xpath*(**"//\*[@id=\"column-center\"]/section/div[3]/article[1]/div[2]/div[3]/span/a[1]"**));  
 element.click();  
 element = *driver*.findElement(By.*xpath*(**"//\*[@id=\"column-center\"]/section/div[3]/article[1]/div[2]/div[3]"**));  
 String[] strs = element.getAttribute(**"class"**).split(**" "**)ж  
 Assert.*assertTrue*(! (strs[strs.**length**-1].contains(**"active"**)));  
 }  
 @Test  
 **public void** thirdTest(){  
 *driver*.get(**"https://pn.com.ua//ct//1003"**);  
 WebElement element = *driver*.findElement(By.*xpath*(**"//\*[@id=\"column-center\"]/section/div[3]/article[1]/div[2]/div[3]/span[2]/a"**));  
 element.click();  
 element = *driver*.findElement(By.*xpath*(**"//\*[@id=\"column-center\"]/section/div[3]/article[1]/div[2]/div[3]/span/a[2]"**));  
 element.click();  
 Assert.*assertTrue*(*driver*.getCurrentUrl().contains(**"compare"**));  
 }  
 @Test  
 **public void** fourthTest() {  
 *driver*.get(**"https://pn.com.ua//ct//1003"**);  
 WebElement element = *driver*.findElement(By.*xpath*(**"//\*[@id=\"column-center\"]/section/div[3]/article[1]/div[2]/div[3]/span[2]/a"**));  
 element.click();  
 element = *driver*.findElement(By.*xpath*(**"//\*[@id=\"column-center\"]/section/div[1]/div[1]/a"**));  
 element.click();  
 element = *driver*.findElement(By.*xpath*(**"//\*[@id=\"mCSB\_1\_container\"]/div[1]/div/div/a"**));  
 element.click();  
 List<WebElement> elements = *driver*.findElements(By.*xpath*(**"//\*[@id=\"mCSB\_1\_container\"]/div[1]/div/div"**));  
 Assert.*assertEquals*(0, elements.size());  
 }  
 @Test  
 **public void** fifthTest() {  
 *driver*.get(**"https://pn.com.ua//ct//1003"**);  
 String Listxpath = **"//\*[@id=\"column-center\"]/section/div[3]/article/div[2]/div[3]/span[2]/a"**;  
 List<WebElement> elements = *driver*.findElements(By.*xpath*(Listxpath));  
 **for**(**int** i=0; i<4; i++)  
 elements.get(i).click();  
 WebElement element = *driver*.findElement(By.*xpath*(**"//\*[@id=\"column-center\"]/section/div[1]/div[1]/a"**));  
 element.click();  
 Listxpath = **"//\*[@id=\"mCSB\_1\_container\"]/div[1]/div/div"**;  
 elements = *driver*.findElements(By.*xpath*(Listxpath));  
 element = *driver*.findElement(By.*xpath*(**"/html/body/div[1]/div[2]/div/div/div[2]/section/div[1]/ul/li[2]/a"**));  
 element.click();  
 element = *driver*.findElement(By.*xpath*(**"/html/body/div[1]/div[2]/div/div/div[2]/section/div[1]/ul/li[2]/ul/li[2]/a"**));  
 element.click();  
 List<WebElement> changed = *driver*.findElements(By.*xpath*(Listxpath));  
 **boolean** isChanged = **false**;  
 **for**(**int** i=0; i<4; i++)  
 **if**(elements.get(i) != changed.get(i))  
 isChanged=**true**;  
 Assert.*assertTrue*(isChanged);  
 }  
 @AfterClass  
 **public static void** finish(){  
  
 }  
}