Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

Магістерська робота

освітній ступінь – магістр

на тему: **«ВИКОРИСТАННЯ КОМП'ЮТЕРНОГО ЗОРУ В АВТОМАТИЗАЦІЇ ТЕСТУВАННЯ ГРАФІЧНОГО ІНТЕРФЕЙСУ КОРИСТУВАЧА»**

Виконала: студентка 2-го року навчання
Спеціальності
121 Інженерія програмного забезпечення

Манжура Анна Володимирівна

Керівник Бучко О.А.
кандидат технічних наук, доцент

Рецензент _____

Магістерська робота захищена з оцінкою _____

Секретар ЕК _____
“____” _____2022 р.

Київ 2022

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ

Керівник магістерської роботи

_____доцент Бучко О.А.

(підпис)

«____»_____2022 р

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на магістерську роботу

студентка 2-го року навчання, освітній ступінь магістр, факультету інформатики
Манжурі Анні Володимирівні

ТЕМА: Використання комп'ютерного зору в автоматизації тестування графічного інтерфейса користувача

ЗАДАЧА: Створити систему для автоматизації тестування графічного інтерфейса користувача використовуючи технології комп'ютерного зору

ЗМІСТ ТЧ до магістерської роботи:

Зміст
Анотація
Вступ
1. Тестування. Теоретична частина
2. Візуальне тестування
3. Практична частина. Побудова фреймворку для візуального регресійного тестування
Висновки
Список літератури
Додатки

Дата видачі «___» _____ 2022 р. Керівник _____

(підпис)

Завдання отримала _____

(підпис)

Календарний план виконання роботи:

№ п/п	Назва етапу дипломного проекту (роботи)	Термін виконання етапу	Примітка
1.	Отримання завдання на курсову роботу.	21.11.2021	
2.	Узгодження теми, структури та термінів виконання з керівником.	15.12.2021	
3.	Огляд технічної літератури за темою роботи.	20.01.2022	
4.	Ознайомлення з теорією.	21.02.2022	
5.	Розробка практичної частини.	10.03.2022	
6.	Написання текстової частини курсової роботи.	14.03.2022	
7.	Надання текстової частини керівнику для попередньої перевірки.	30.06.2022	
8.	Створення слайдів для доповіді та написання доповіді.	01.07.2022	
9.	Захист курсової роботи.	07.07.2022	

Студент *Манжура А. В.*

Керівник *Бучко О.А.*

“_____”_____

ЗМІСТ

Анотація

Метою магістерської роботи є дослідження використання комп’ютерного зору у забезпеченні візуального тестування веб-застосунків, а також розробка такої системи, яка інтегрована у функціональні автоматизовані набори тестів. Таким чином здійснюється моніторинг та аналіз візуальних змін у графічному інтерфейсі тестованого додатка. Запропонований інструмент має вирішити існуючі проблеми традиційного візуального тестування знімків. Було розглянуто основні теоретичні поняття у тестуванні, його типах, фреймворках. Також виконано огляд сучасних проблем у візуальному тестуванні, наведено шляхи вирішення деяких. В роботі надано аналіз та тематичне дослідження низки алгоритмів обробки та аналізу зображень, як-от сегментація, індекс структурної схожості, що використовуються для візуального порівняння зображень. При розробці автоматизованої системи візуального тестування веб-інтерфейсів були використані мова програмування Python, бібліотеки Keras та OpenCV, платформа тестування Pytest.

Вступ

Тестування графічного інтерфейсу користувача (GUI) є дуже важливим етапом тестування для контролю якості програмних додатків. GUI є центральним вузлом тестової програми, звідки можна отримати доступ до всіх функцій. Таким чином, важко ретельно тестувати програми за допомогою їх графічного інтерфейсу, особливо тому, що вони призначені для роботи з людьми, а не з машинами. Крім того, GUI за своєю суттю є нестатичним інтерфейсом, схильним до постійних змін, спричинених оновленням функціональності, покращеною зручністю використання, зміною вимог або зміненим контекстом. Це ускладнює розробку та обслуговування тестових випадків, не вдаючись до тривалого та дорогого ручного тестування.

Автоматизоване візуальне тестування — це процес забезпечення якості, призначений для автоматичної перевірки візуального відображення інтерфейсу. Найбільша проблема з автоматизованим візуальним тестуванням полягає в тому, що люди і машини сприймають пікселі по-різному. Два зображення інтерфейсу користувача можуть здаватися абсолютно ідентичними для людини, але відрізнятися на рівні пікселів. Алгоритми згладжування для зміни розміру зображень, різні відеокарти створюють відмінності в пікселях, а це призводить до великої кількості помилкових дефектів.

Предметом дослідження є методи аналізу та порівняння очікуваних і реальних зображень інтерфейсу. Актуальність даної роботи визначається тим, що у зв'язку зі стрімким розвитком автоматизованого візуального тестування виникла потреба в розробці системи, яка дозволить більш точно аналізувати інтерфейс і зменшити кількість хибних результатів.

Метою роботи є дослідження інструментів і технологій комп’ютерного зору для аналізу, обробки та порівняння зображень графічного інтерфейсу, отриманих у процесі виконання тестів автоматичних функцій регресії, та створення системи, яка дозволить відстежувати помилки інтерфейсу веб-додатків і генерувати звіт.

РОЗДІЛ 1. ТЕСТУВАННЯ. ТЕОРЕТИЧНА ЧАСТИНА.

1.1 Тестування програмних застосунків

1.1.1 Визначення

Тестування програмного забезпечення – це спосіб оцінити якість програмного забезпечення та зменшити ризик збою програмного забезпечення в роботі. Поширене помилкове сприйняття тестування полягає в тому, що воно складається лише з виконання тестів, тобто виконання програмного забезпечення та перевірки результатів. Проте насправді це процес, який включає багато різних видів діяльності, і виконання тестів (включно з перевіркою результатів) є лише одним із цих видів діяльності. Процес тестування також включає в себе такі дії, як планування тестування, аналіз, проектування та впровадження тестів, звітування про хід і результати тестування, оцінка якості тестового об’єкта, перевірка робочих продуктів, таких як вимоги і вихідний код. Тестові активності організовуються та виконуються по-різному – залежно від життєвого циклу проекту.

1.1.2 Основні цілі тестування

Для будь-якого проекту цілі тестування можуть включати такі:

а) запобігання дефектам, оцінюючи робочі продукти, такі як вимоги, історії користувачів, дизайн та код;

б) перевірка, чи були виконані всі зазначені вимоги;

в) перевірка, чи є тестовий об'єкт завершеним, і перевірити, чи працює він так, як очікують користувачі та інші зацікавлені сторони;

г) формування впевненості у рівні якості тестового об'єкта;

ґ) знаходження дефектів та збоїв, знижуючи рівень ризику неналежної якості програмного забезпечення;

д) надання достатньої інформації зацікавленим сторонам, щоб дозволити їм приймати обґрунтовані рішення;

е) гарантування дотримання договірних, юридичних або нормативних вимог чи стандартів та перевірка відповідності об'єкта тестування таким вимогам або стандартам [9].

1.1.3 Види тестування

Тестування можна поділити на декілька груп типів залежно від певних атрибутів, як-от: наскільки великі частини програми тестуються, які знання про об’єкт тестування мають тестувальники, спосіб виконання тесових активностей тощо.

Залежно від глибини розуміння тестувальником тестованого додатка виділяють такі види:

а) білого ящика;

б) чорного ящика;

в) сірого ящика.

Залежно від рівнів тестування існують такі види:

а) компонентне тестування;

б) інтеграційне тестування;

в) системне тестування;

г) приймальне тестування.

За способом проведення тестування виокремлюють мануальне (ручне) та автоматизоване тестування. Мануальне включає в себе ручні завдання, такі як: налаштування тестового середовища, виконання перевірочної функціональності, збір і перегляд результатів і запис знайдених проблем. Цей процес можна виконати, дотримуючись плану тестування або, як альтернатива, шляхом дослідницького тестування [9].

Повну класифікацію з багатьма іншими видами та підвидами тестування можна побачити на Рисунку 1.1.

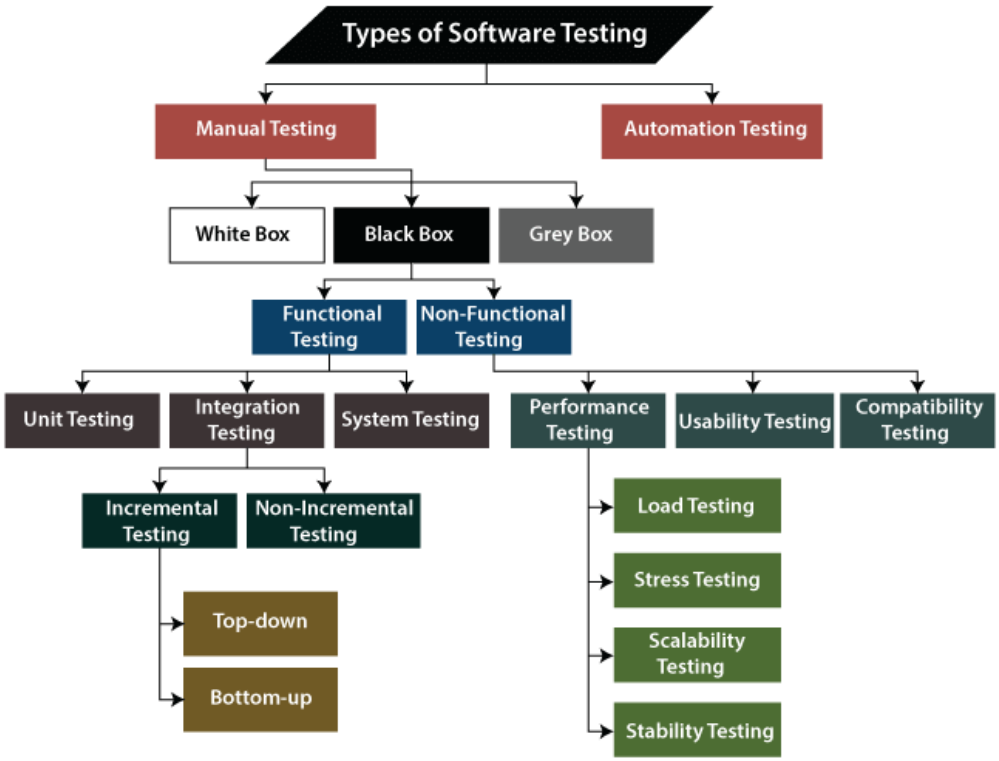


Рисунок 1.1 – Типи тестування програмного забезпечення [13]

1.2 Автоматизоване тестування

1.2.1 Основні поняття та цілі

У тестуванні програмного забезпечення автоматизація тестування — це одне або кілька з наступних завдань:

а) використання спеціально створених програмних засобів для контролю та налаштування тестових передумов;

б) виконання тестів;

в) порівняння фактичних результатів з очікуваними;

Хорошою практикою є відокремлення програмного забезпечення, яке використовується для тестування, від самої системи, що тестується (SUT), щоб

мінімізувати втручання. Існують винятки, наприклад, вбудовані системи, де програмне забезпечення потрібно розгорнути на SUT.

Автоматизація тестування допомагає послідовно і багаторазово виконувати велику множину тестових випадків на різних версіях SUT та/або середовищ. Але це більше, ніж механізм для запуску набору тестів без взаємодії з людиною. Автоматизування включає в себе процес розробки, а саме такого:

а) власне програмне забезпечення;

б) документація;

в) тестові випадки;

г) тестові середовища;

ґ) дані тесту.

Ця сукупність тестового програмного забезпечення необхідне для таких активностей, як-от: впровадження автоматизованих тестових випадків, моніторинг і контроль виконання тестів, а також інтерпретація, звітування та запис результатів.

Серед основних цілей можна виокремити такі:

а) підвищення ефективності тестування;

б) забезпечення ширшого охоплення функцій;

в) зменшення загальної вартості тесту;

г) виконання тестів, які не піддаються мануальному тестуванню;

ґ) скорочення періоду виконання тесту;

д) збільшення частоти виконання тестів та скорочення часу, необхідного для тестових циклів.

1.2.2 Переваги, недоліки та обмеження

Однією з найбільш очевидних переваг є те, що автоматизоване тестування значно економить час та зусилля. Скриптування автоматичних перевірок може зайняти деякий час. Однак, під час виконання, вони, як правило, швидкі й можуть

проходити різні етапи набагато швидше, ніж людина. Тому вони допомагають швидко надавати зворотний зв’язок команді розробників щодо якості тестованого додатка або певного компонента.

Тестування займає значну частину загального життєвого циклу розробки програми. Це свідчить про те, що навіть найменше покращення загальної ефективності може істотно змінити загальні часові рамки проекту. Хоча спочатку налаштування займає більше часу, автоматизовані тести з часом потребують значно менше часу. Їх можна запускати практично без нагляду, залишаючи результати моніторингу до кінця процесу.

Найкращим використанням автоматизованих перевірок є регресійні тести, тобто такі тести, що виконуються щоразу після того як в програму були внесені модифікації і є ризик появи дефектів. Автоматизація регресійних тестів звільняє час тестувальників, тому вони можуть більше зосередитися на дослідницькому тестуванні нових функцій. З точки зору бізнесу, початкові інвестиції в написання автоматизованих тестів в довготривалій перспективі також виправдовують себе, адже економлять гроші та ресурси на одноманітні повторювані тестові завдання.

Ще одним безперечним плюсом є те, що автоматичні перевірки зазвичай пишуться тією ж мовою, що й тестована програма. З цієї причини відповідальність за написання, підтримку та виконання тестів стає спільною. Кожен в команді розробників може зробити свій внесок, а не тільки тестувальники.

Автоматизація тестування при правильному виконанні може мати багато переваг і бути надзвичайно корисною для проекту та організації. Проте є деякі підводні камені або недоліки, які варто враховувати.

По-перше, автоматизоване тестування може надати помилкове відчуття якості. Автоматична перевірка перевіряє лише те, що було запрограмовано для перевірки. Усі тести в наборі можуть успішно пройти, але при цьому можуть залишитись непоміченими серйозні недоліки. Причина цього полягає в тому, що автоматична перевірка не була закодована для «пошуку» цих збоїв. Розв’язок цієї проблеми полягає в тому, щоб переконатися, що було розроблено вдалі тестові

сценарії, перш ніж їх автоматизувати. Ефективність автоматизованої перевірки напряму залежить від якості дизайну тесту. Також завжди треба доповнювати автоматичні перевірки ручним та дослідницьким тестуванням.

По-друге, автоматичні перевірки можуть бути ненадійними через багато факторів. Якщо вони продовжують давати збій через проблеми, відмінні від справжніх дефектів, то виходять хибні результати. Наприклад, автоматична перевірка може порушитися через зміну інтерфейсу користувача, відключення служби або проблеми з мережею. Ці проблеми не пов’язані безпосередньо з тестовою програмою, але можуть вплинути на результати автоматизованих тестів.

По-третє, автоматизовані тести вимагають постійної підтримки. У міру того, як програма, що тестується, розвивається, мають розвиватися й автоматизовані перевірки. Якщо регресійні набори тестів не оновлюються, виникає все більше різних видів збоїв. Деякі перевірки можуть бути вже не актуальними, а деякі – вже не справжнім відображенням нових реалізацій. Ці збої будуть забруднювати результати. Початок автоматизації тестування — це не разова робота. Щоб отримати максимальну віддачу від нього, потрібно постійно тримати тести оновленими та відповідними до останніх програмних збірок. Це вимагає багато часу, зусиль і ресурсів. Оскільки фактор підтримки тестів є постійною діяльністю, необхідно приділити час розробці хорошої основи: використовувати модулі багаторазового використання, відокремлювати тести від фреймворка та використовувати принципи хорошого проектування, щоб полегшити витрати на обслуговування [3].

Окрім переваг та недоліків, є ще певні обмеження автоматизованого тестування. Не всі мануальні тести можуть бути автоматизованими – перевірити можна лише ті результати, які машина може інтерпретувати. Тобто, наприклад, жодним чином не вдасться автоматизовано оцінити досвід користувача щодо простоти використання, зручності графічного інтерфейсу, зовнішнього вигляду та естетичної узгодженості в додатку. З цього випливає наступний висновок – автоматизоване тестування не є повною заміною ручного та дослідницького.

Щоб правильно та повністю протестувати програму, завжди потрібен людський інтелект. Тестування вимагає знання предметної області, зосередженого розуму та бажання вивчати застосунок. Це не просто виконання набору попередньо визначених кроків та порівняння фактичних результатів з очікуваними. Тому, абсолютно необхідно поєднувати та доповнювати автоматичні перевірки ручним та дослідницьким тестуванням.

1.2.3 Підходи до автоматизованого тестування

Вибір правильного підходу до автоматизації є життєво важливим для забезпечення належного тестування, мінімізації витрат на розробку та підтримку. Існує багато підходів до автоматизації тестування. Залежно від типу взаємодії з системою, що тестується, загальноприйнятими є такі:

а) API-тестування – тестування через загальнодоступні інтерфейси до класів, модулів або бібліотек, які валідуються за допомогою різноманітних вхідних аргументів для перевірки правильності результатів, що повертаються;

б) GUI-тестування – тестування через графічний інтерфейс користувача, яке генерує події, такі як натискання клавіш і клацання миші, і спостерігає за змінами в інтерфейсі користувача, щоб підтвердити правильність спостережуваної поведінки програми.

Інший варіант класифікації підходів залежать від способу перетворення тест-кейсів в послідовність дій, які виконуються проти системи, що тестується. Добре зарекомендовані підходи включають наступні:

а) capture/playback;

б) лінійні скриптові;

в) структуризовані скриптові;

г) керовані даними;

ґ) на основі ключових слів;

д) на основі моделі.

Підхід capture/playback – це найпростіший підхід до автоматизованого тестування з обмеженою гнучкістю та довгостроковою корисністю. Тут певні інструменти використовуються для фіксації (запису) взаємодії з системою під час виконання послідовності дій, визначеної процедурою тестування. Під час відтворення подій існують різні можливості ручної та автоматичної перевірки вихідних даних. Головним недоліком цього підходу є те, що такі записи важко підтримувати та розвивати, оскільки їх виконання сильно залежить від версії системи, в якій було записано сценарій. Наприклад, зміни в макеті графічного інтерфейсу можуть вплинути на тестовий скрипт, навіть якщо це лише зміна позиціювання елемента.

В підході лінійного написання скриптів кожен тест виконується вручну, поки тестовий інструмент записує послідовність дій, а в деяких випадках фіксує видимий результат на екран. Зазвичай це призводить до створення одного великого сценарію для кожної процедури тестування. Записані сценарії можна редагувати, щоб покращити читабельність (наприклад, додаючи коментарі, щоб пояснити, що відбувається в ключових моментах) або додавати додаткові перевірки за допомогою мови сценаріїв інструмента. Скрипти потім можуть відтворюватися інструментом, змушуючи інструмент повторювати ті самі дії, які зробив тестувальник, коли сценарій був записаний. Незважаючи на те, що це можна використовувати для автоматизації тестів GUI, це не найкраща техніка для автоматизації великої кількості тестів, які потрібні для багатьох релізів програмного забезпечення. Це пов’язано з високими витратами на технічне обслуговування, які зазвичай викликаються змінами в SUT.

Основна відмінність між технікою структурованих сценаріїв і технікою лінійного написання сценаріїв полягає у введенні у використання бібліотеки сценаріїв. Вони містять багаторазові сценарії, які виконують послідовності інструкцій, що зазвичай потрібні в ряді тестів. Хорошими прикладами таких сценаріїв є ті, які взаємодіють, наприклад, з операціями інтерфейсів системи, що тестується. Переваги цього підходу включають значне скорочення необхідних змін у технічному обслуговуванні та зниження витрат на автоматизацію нових

тестів (оскільки вони можуть використовувати вже існуючі сценарії, а не створювати їх усі з нуля) [11].

Техніка написання сценаріїв, керованих даними, заснована на техніці написання структурованих сценаріїв. Найбільш істотна відмінність полягає в тому, як обробляються тестові вхідні дані. Вони витягуються зі скриптів і поміщаються в один або кілька окремих файлів (як правило, їх називають файлами даних). Це означає, що основний тестовий сценарій можна повторно використовувати для реалізації кількох тестів. Зазвичай цей сценарій багаторазового використання називається сценарієм контролю. Контрольний сценарій містить послідовність інструкцій, необхідних для виконання тестів, але зчитує вхідні дані файл даних. Один контрольний тест може використовуватися для багатьох тестів, але зазвичай його недостатньо для автоматизації широкого кола тестів. Ця техніка надає більш глибоке тестування в певній області та може збільшити охоплення тестами [11].

Техніка написання сценаріїв на основі ключових слів базується на техніці написання сценаріїв, керованих даними. Є дві основні відмінності: файли даних тепер називаються файлами «тестового визначення»; і є лише один сценарій контролю. Файл визначення тесту містить опис тестів у спосіб, який має бути легшим для розуміння аналітиками тестів (простіше, ніж еквівалентний файл даних). Зазвичай він містить дані, як і файли даних, але файли ключових слів також містять інструкції високого рівня («слова дії»). Ключові слова мають бути вибрані так, щоб вони були значущими для аналітика тестування, описуваних тестів та програми, що тестується. Вони в основному (але не виключно) використовуються для представлення бізнес-взаємодій високого рівня із системою. Кожне ключове слово являє собою ряд детальних взаємодій із системою, що тестується. Наприклад, «створити обліковий запис», «розмістити замовлення», «перевірити статус замовлення» — це всі можливі дії для програми онлайн-покупок, кожна з яких включає ряд детальних кроків. Коли один тестовий аналітик описує системний тест іншому тестовому аналітику, він, ймовірно, буде говорити в термінах цих дій високого рівня, а не детальних

кроків. Таким чином, мета підходу, керованого ключовими словами, полягає в тому, щоб реалізувати ці дії високого рівня та дозволити тестам бути визначеними в термінах дій високого рівня без посилання на детальні кроки. Сфера відповідальності тестових аналітиків включає створення та підтримку файлів ключових слів. Це означає, що після впровадження допоміжних сценаріїв аналітики тестування можуть додавати «автоматичні» тести, просто вказавши їх у файлі ключових слів (як у сценаріях на основі даних).

Тестування на основі моделі відноситься до технік автоматизованого створення тестових випадків на відміну від розглянутих вище, де автоматизоване було тільки виконання тестів. Цей підхід дозволяє шляхом абстракції зосередитися на суті тестування (з точки зору бізнес-логіки, даних, сценаріїв, конфігурацій тощо, що підлягають тестуванню), а також генерувати тести для різних цільових систем і цільових технологій, щоб моделі, які використовуються для генерації тестів, являли собою безпечне у майбутньому представлення тестового програмного забезпечення, яке можна повторно використовувати та підтримувати в міру розвитку технології. У разі зміни вимог необхідно адаптувати лише тестову модель, а повний набір тестових випадків генерується автоматично.

1.2.4 Загальна архітектура автоматизації тестування

Будь-який проект автоматизації тестування потрібно налаштовувати як проект розробки програмного забезпечення, і він вимагає спеціального управління. На Рисунку 1.2 зображена схема загальної архітектури такого проекту.

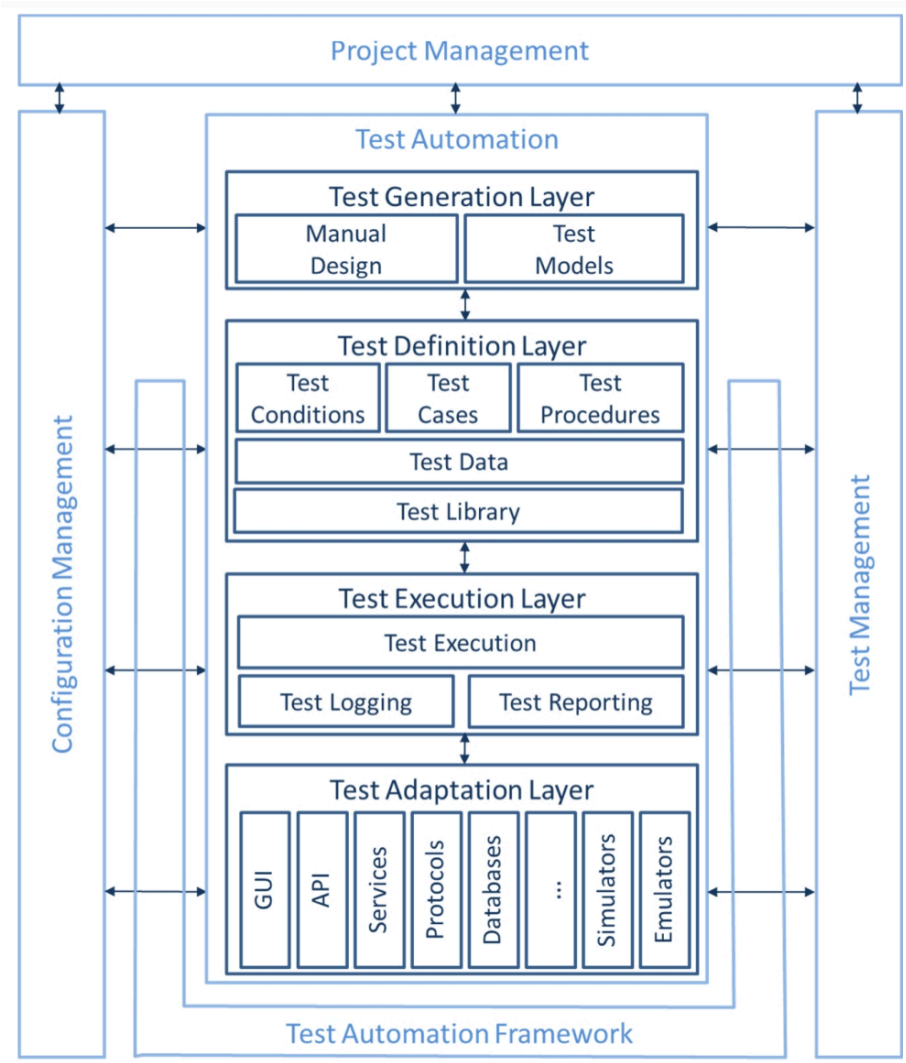


Рисунок 1.2 – Загальна архітектура автоматизації тестування [2]

Рівень генерації тестів складається з інструментів, які підтримують мануальну розробку тестових випадків, збір та отримання даних тестування, а також автоматичне генерування тест кейсів з моделей, якщо використовується відповідний підхід. Компоненти цього рівня використовуються для редагування та переміщення по структурах тестового набору, зв’язки тестових випадків з цілями тестування та вимогами, а також для документування тест дизайну.

Рівень визначення тестів складається з інструментів для підтримки визначення тестових випадків (на високому та/або низькому рівні), тестових даних, процедур тестування, тестових сценаріїв, а також тут відбувається надання доступу до тестових бібліотек за потреби (наприклад, у підходах, керованих ключовими словами).

Рівень виконання тесту забезпечує автоматичне виконання тестів, логування цього виконання та звітування про результати. Тут відбувається параметризування налаштувань тесту, інтерпретування тестових даних та тестових випадків та перетворення їх у виконувані сценарії, аналізування відповідей системи під час виконання тесту та порівняння отриманих результатів з очікуваним.

Рівень адаптації тестів забезпечує посередництво між технологічно незалежними визначеннями тестів та специфічними технологічними вимогами системи та тестових пристроїв. Інструменти цього рівня підтримують взаємодію з системою, що тестується, її моніторинг та симуляцію чи імітацію середовища для неї. Тут застосовуються різні технологічні адаптери, а також відбувається розподіл виконання тесту між кількома тестовими пристроями чи тестовими інтерфейсами, або ж забезпечується виконання тестів локально.

Ці чотири рівні формують разом концепцію фреймворку автоматизованого тестування [2], який буде розглянуто детальніше далі. Окрім цього, ця структура має бути підсилена конфігураційним управлінням, яке гарантує сумісність та відповідність версій програмного забезпечення та фреймворку. Більше того, як і будь-який інший проект, автоматизоване тестування має керуватись з точки зору виконання завдань для всіх етапів встановленої методології життєвого циклу при розробці, а також має бути нагляд за тим, щоб інформацію про стан (метрики) можна було легко витягти або автоматично повідомити керівництву проекту [2].

1.3 Фреймворки автоматизованого тестування

1.3.1 Поняття фреймворку автоматизації та їх типи

Інженер з автоматизації тестування займається проектуванням, розробкою, впровадженням та підтримкою рішень з автоматизації тестування. У міру розробки кожного рішення необхідно виконувати подібні завдання, відповідати на подібні питання, а також розставляти подібні проблеми та розставляти їх пріоритети. Ці повторювані концепції, кроки та підходи в автоматизації

тестування стають основою загальної архітектури. Вона в свою чергу складається як із тестового середовища (і його артефактів), так і з наборів тестових випадків (включаючи тестові дані). Для реалізації цього необхідно використовувати певне програмне рішення, яку часто називають фреймворком автоматизації. Ця платформа забезпечує підтримку для реалізації тестового середовища та надає інструменти, тестові засоби або допоміжні бібліотеки. Фреймворк повинен бути простий у використанні та підтримці. Окрім цього, він має мати впроваджені засоби звітності, які б забезпечували залучених тестувальників, менеджерів з тестування, розробників, керівників проектів та інших зацікавлених сторін тестовими звітами з інформацію про якість системи, що тестується. Також необхідно реалізувати простий спосіб усунення несправностей у невдалих тестах, які могли провалитися і через реальні виявлені несправності в системі, що тестується, і через збої самого фреймворку, і через проблеми з самими тестами або тестовим середовищем. В результаті цього фреймворк також повинен мати можливість швидко відновитися, пропустити поточний провалений тестовий випадок і продовжити тестування з наступного.

Важливо використовувати фреймворк для автоматизованого тестування, оскільки він однозначно покращує ефективність та швидкість тестування команди, а також допомагає підвищити точність, різко зменшує витрати на обслуговування тестів і знижує ризики. Завдяки численним перевагам фреймворків, автоматизація тестування стала фундаментальним компонентом сучасних методів Agile і DevOps.

Існує сім поширених типів фреймворків автоматизації тестування, кожна зі своєю власною архітектурою та різними перевагами і недоліками. Більшість з цих типів напряму корелюють з підходами автоматизованого тестування, які було описано в розділі 1.2.3. На Рисунку 1.3 можна побачити цю класифікацію.

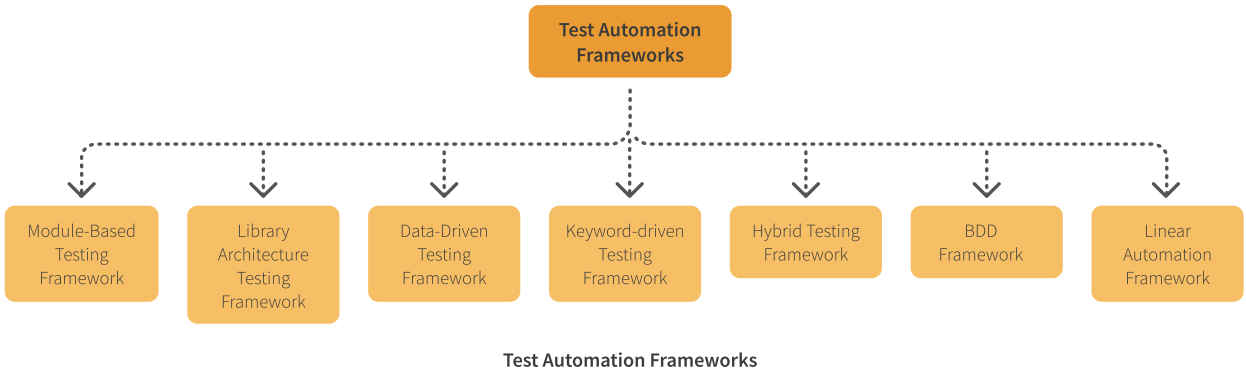


Рисунок 1.3 – Типи фреймворків автоматизації [13]

Linear Scripting Framework — це фреймворк автоматизації тестування базового рівня, також відома як фреймворк «Запис і відтворення». Тестерам не потрібно писати власне код для створення функцій, а кроки записуються в послідовному порядку. У цьому процесі записується кожен крок, наприклад навігація, введення користувачем або контрольні точки, а потім автоматично відтворюється сценарій для проведення тесту. Це один із найшвидших способів створення тестових сценаріїв, оскільки їх можна легко записати за мінімальний час. Проте скрипти, розроблені за допомогою цього фреймворка, не можна використовувати повторно, адже дані жорстко закодовані, і сценарії потрібно буде змінити, якщо дані будуть змінені. Підтримка також є клопітною, оскільки будь-які зміни в програмі вимагатимуть багато переробки. Ця модель не особливо масштабована по мірі розширення області тестування.

Реалізація модульного фреймворку вимагатиме від тестувальників розділити програму, що тестується, на окремі блоки, функції або розділи, кожен з яких буде тестуватися окремо. Після розбиття програми на окремі модулі для кожної частини створюється тестовий сценарій, які потім об’єднуються, щоб побудувати більші тести в ієрархічній формі. Ключовою стратегією використання цього типу фреймворку є створення рівня абстракції, щоб будь-які зміни, внесені в окремих розділах, не впливали на загальний модуль.

Фреймворк, що засновано на бібліотеках, базується на попередньому типу, але має деякі додаткові переваги. Замість того, щоб ділити тестований додаток

на різні сценарії, які потрібно запустити, подібні завдання всередині сценаріїв ідентифікуються, а потім групуються за функціями, тому програма в кінцевому підсумку розбивається за загальними цілями. Ці функції зберігаються в бібліотеці, яка може бути викликана тестовими сценаріями, коли це необхідно.

Використання типу фреймворку, керованого даними, відокремлює дані тесту від логіки сценарію, тобто тестувальники можуть зберігати дані ззовні. Дуже часто виникають ситуації, коли необхідно кілька разів перевірити ту саму функцію з різними наборами даних. У цих випадках важливо, щоб дані тестування не були жорстко закодовані в самому сценарії, що відбувається з лінійними або модульними фреймворками тестування. Налаштування системи тестування на основі даних дозволить тестувальнику зберігати та передавати вхідні/вихідні параметри тестовим сценаріям із зовнішнього джерела даних, наприклад, електронних таблиць Excel, текстових файлів, файлів CSV, таблиць SQL або репозиторіїв ODBC.

У фреймворці, що керується ключовими словами, кожна функція програми, що тестується, викладена в таблиці з серією інструкцій у послідовному порядку для кожного тесту, який потрібно запустити. Ключові слова також зберігаються у зовнішній таблиці даних, що робить їх незалежними від інструменту автоматичного тестування, який використовується для виконання тестів. Ключові слова — це частина сценарію, що представляє різні дії, що виконуються для перевірки графічного інтерфейсу програми. Вони можуть бути позначені просто як «клацніть» або «увійти». Після того, як таблиця буде налаштована, тестувальникам залишається лише написати код, який підкаже необхідну дію на основі ключових слів. Коли тест виконується, дані тесту зчитуються та вказуються на відповідне ключове слово, яке потім виконує відповідний сценарій.

Behavior Driven Development (BDD) фреймворк – дозволяє всім учасникам команди (наприклад, бізнес-аналітикам, розробникам, тестувальникам тощо) брати активну участь у процесі розробки та аналізу тестів. Тут для створення

сценаріїв використовується нетехнічна, природна мова для підвищення взаємодії та колаборації на проекті.

Як і більшість сучасних процесів тестування, фреймворки автоматизації почали інтегруватися і перекривати одна одну. Як випливає з назви, гібридний фреймворк — це комбінація будь-якого з вищезгаданих типів, створених для використання переваг одних і пом'якшення недоліків інших. Кожна програма відрізняється, як і процеси, які використовуються для їх тестування. Оскільки все більше команд переходять до гнучкої моделі, створення гнучкого каркаса для автоматизованого тестування має вирішальне значення. Гібридну структуру можна легше адаптувати, щоб отримати найкращі результати тестування.

1.3.2 Аналіз та порівняння існуючих фреймворків

Для аналізу та порівняння обрано фреймворки, які використовують мову Python. Ця мова програмування з відкритим вихідним кодом є інтуїтивно зрозумілою, простою, елегантною й менш загромадженою, аніж деякі інші. І коли справа доходить до тестування, такі якості є обов'язковими, особливо якщо в команді є мануальні тестувальники без навичок програмування, які зараз переходять на автоматизацію та можуть доволі швидко навчитися Python для ефективного написання скриптів.

1.3.2.1 Robot Framework

Цей фреймворк є таким, що керується ключовими словами, тобто він дозволяє легко створювати тестові кейси за допомогою зрозумілих для людини ключових слів, не вимагаючи при цьому досвіду програмування. Robot Framework підтримує всі операційні системи (Windows, Linux або MacOS) і всі типи програми (веб, мобільні та настільні). Також він надає чіткі та зручні дані звітів HTML (включаючи знімки екрана). Robot Framework має багату екосистему з великою кількістю API, що робить її дуже розширюваною

платформою та дозволяє інтегруватися з будь-яким іншим інструментом сторонніх розробників.

До мінусів фреймворка належать усі ті, що стосуються цього типу фреймворків загалом. Якщо є намір реалізувати підхід, керований ключовими словами, який дозволить тестувальникам і бізнес-аналітикам створювати автоматизовані тести, RF — це правильне рішення, яке надає різноманітні розширення та бібліотеки, і є простим у використанні. Однак, якщо необхідно вести розробку складних сценаріїв, чи є необхідність потрібно складної конфігурації, то RF – не найкращий варіант.

1.3.2.2 Pytest

Pytest — це платформа тестування з відкритим кодом, яка, можливо, є однією з найбільш широко використовуваних фреймворків тестування Python. Pytest також підтримує модульне тестування, функціональне тестування та тестування API. За допомогою Pytest можна писати компактні, прості набори тестів. Він має велику спільноту та велику підтримку, уможливлює охоплення всіх комбінацій параметрів без необхідності переписувати тестові випадки. При цьому Pytest має певні проблеми з сумісністю, а це означає, що, хоча тестові випадки легко писати в цьому фреймворку, та буде неможливо використовувати їх в інших платформах тестування.

Класичним застосуванням цього фреймворку є саме написання модульних тестів, проте він є дуже розширюваний за допомогою багатьох доступних плагінів, тому стає можливим інтеграція з іншими інструментами, що уможливлюють написання тестів для графічного веб-інтерфейсу користувача для різних браузерів.

1.3.2.3 Behave

Behave є однією з широко використовуваних платформ тестування Selenium Python для тестування BDD. Він використовує мову Gherkin для

розробки сценаріїв і файлів функцій. Оскільки Gherkin використовує просту мову для читання для розробки тестових випадків, тести також можна створювати для нетехнічного персоналу в команді, що дозволить розробку, орієнтовану на бізнес. Використання цього фреймворку забезпечує кращу комунікацію між членами команди, оскільки збільшується співпраця між інженерами, менеджерами, аналітиками з якості, менеджерами з розвитку бізнесу тощо. Він підтримує інтеграцію з іншими веб-фреймворками та має чудову допоміжну документацію. Головним недоліком Behave є відсутність вбудованої підтримки паралельного виконання тестів, що є однією з основних вимог для автоматизації тестування браузера.

РОЗДІЛ 2. ВІЗУАЛЬНЕ ТЕСТУВАННЯ

2.1 Візуальне тестування

2.1.1 Визначення

Як вже зазначалось, існує багато видів тестування програмного забезпечення, але певно основним поділом є на функціональне та нефункціональне. Усі перевірки, що спрямовані на виявлення невідповідностей між реальною поведінкою реалізованих функцій і очікуваною поведінкою відповідно до специфікації і вимог, називаються функціональними. А окрім цього, існує ще безліч інших характеристик програмного забезпечення, які необхідно тестувати, як-от: надійність, міцність, зручність використання, стабільність, локалізацію тощо. Усе це є складниками нефункціонального тестування. Однією з найбільш помітних нефункціональних характеристик програмного забезпечення для користувачів є якість графічного інтерфейсу (GUI). Візуальне тестування — це спроба виявити нефункціональні помилки, які можуть з’являтися через зміну графічного стану програми, що тестується. Оцінюється саме видимий результат програми, який порівнюється з результатами, очікувані за дизайном. Іншими словами, це допомагає виявити «візуальні помилки» у зовнішньому вигляді сторінки або екрана, які відрізняються від суто функціональних помилок.

Типовим прикладом може бути веб-додаток, в якому графічний інтерфейс зазвичай програмується комбінацією мови гіпертекстової розмітки (HTML) і каскадних таблиць стилів (CSS). HTML часто використовується для визначення вмісту веб-програми (наприклад, сторінка містить таблицю, зображення тощо), тоді як CSS визначає структуру та зовнішній вигляд веб-програми (наприклад, колір шрифту, абсолютне розташування елементів веб-сторінки). Отриманий веб-додаток являє собою набір правил (CSS і HTML), що застосовуються до статичного вмісту (наприклад, зображення, відео, текст). Комбінація правил має вирішальне значення, і незначна зміна може повністю змінити візуальний стан веб-програми. Такі зміни дуже важко, іноді навіть неможливо виявити за

допомогою автоматизованого функціонального тестування програми, тому що функціональні тести перевіряють лише бажану функціональність веб-програми та ігнорують характеристики веб-сторінки, такі як червоний колір заголовка, пробіл між двома абзацами тощо. Саме тому візуальне тестування є необхідним. Це робиться або вручну, коли тестер переглядає всі варіанти використання веб-додатка та перевіряє, що програма візуально не зламалася. Або це виконується автоматично, шляхом виконання сценаріїв, які підтверджують візуальний стан програми. Автоматизовані інструменти візуального тестування можуть допомогти прискорити це та зменшити кількість помилок, які виникають під час перевірки вручну. У цій дипломній роботі ми зосередимося саме на візуальному тестуванні веб-додатків.

2.1.2 Важливість

Постає закономірне питання: чому візуальне тестування є настільки важливим? Ми проводимо візуальне тестування, тому що візуальні помилки трапляються частіше, ніж можна собі уявити. В програмних продуктах усіх найбільших компаніях час від часу користувачі знаходять візуальні помилки різного ступеня впливовості на задоволення потреб клієнтів. Десь текст «з'їхав», десь інтегрована реклама перекрилася картинкою, десь елемент пересунувся на хибне місце. Але в усіх випадках, це не просто косметичні проблеми, це речі, які частково або повністю блокують дохід компаніям. Реальні приклади візуальних помилок, які призводять до збитків, зображено на Рисунку 2.1.

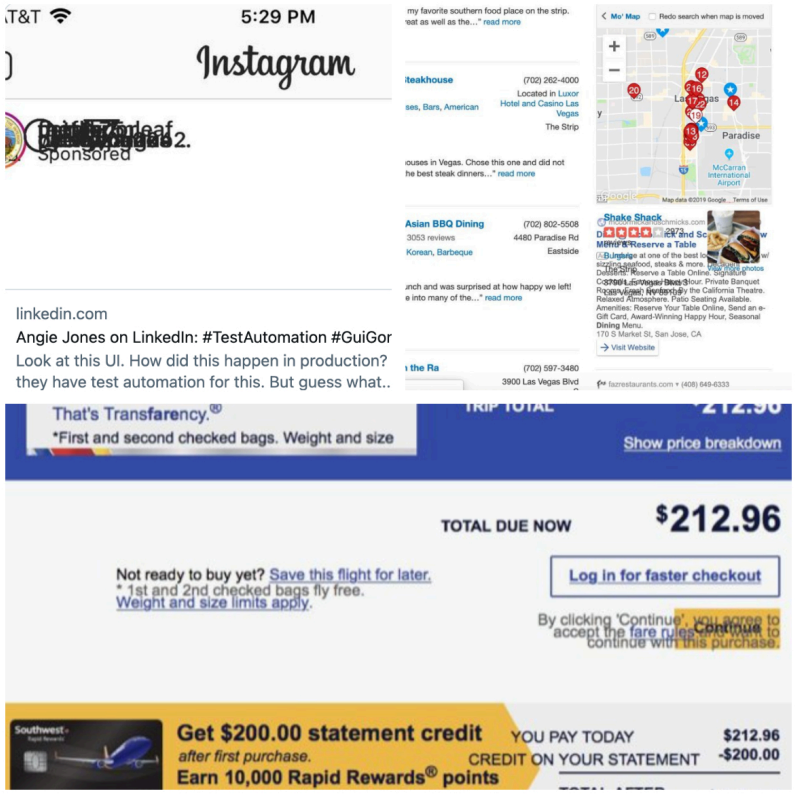


Рисунок 2.1 – Приклади реальних візуальних помилок великих компаній

Ще одне питання: чому функціональні тести не можуть ловити ці візуальні помилки? Насправді скрипти функціональних тестів можуть перевірити розмір, положення та колірну схему візуальних елементів. Але при цьому це призведе до значного збільшення тестових сценаріїв в розмірах через роздутість перевірок. Наприклад, допустимо на веб-сторінці є 21 візуальний елемент – різноманітні іконки, текст тощо. Якщо використовувати традиційні перевірки в інструментах функціонального тестування, як-от Selenium Webdriver, Cypress, WebdriverIO або Appium, то доведеться перевірити наступне для кожного з цих 21 візуальних елементів: чи є елемент видимим, верхні ліві координати x та y, висота, ширина та колір фону. Тобто, перемноживши кількість усіх елементів на кількість перевірок, ми отримаємо 105 рядків перевірочного коду. Та навіть з усім цим, не можна буде виявити абсолютно всі візуальні помилки, наприклад, чи можливо отримати доступ до візуального елемента, якщо він перекритий іншим. Більше того, якщо зміниться операційна система, браузер, орієнтація екрана, розмір екрана або розмір шрифту – в результаті зміниться і зовнішній

вигляд програми. Це означає, що потрібно буде написати ще 105 рядків тверджень функціонального тесту, а саме для кожного поєднання цих параметрів. У підсумку можна отримати тисячі рядків коду, будь-який з яких, можливо, доведеться змінити з новим релізом програмного забезпечення. Намагатися підтримувати це було б надзвичайно складно і дуже сильно ресурсозатратно.

2.1.3 Автоматизація візуального тестування

Оскільки інструменти автоматичного функціонального тестування погано підходять для пошуку візуальних помилок, компанії зазвичай знаходять візуальні збої за допомогою мануальних тестувальників [15]. Для них це перетворюється на гру «Знайди різницю». Це є абсолютно неефективно і займає величезну кількість часу, адже деякі відмінності важко помітити, а в інших випадках наші очі обманюють нас, щоб знайти відмінності, яких не існує. Візуальне тестування вручну означає порівняння двох знімків екрана, одного з відомого правильного базового зображення, а іншого з останньої версії тестованої програми. Для кожної пари зображень необхідно витратити час, щоб переконатися, що вловили всі проблеми, особливо якщо сторінка довга або містить багато візуальних елементів. Очевидним стає необхідність автоматизації цього процесу.

Автоматизоване візуальне тестування використовує програмне забезпечення для автоматизації процесу порівняння візуальних елементів у різних комбінаціях екрану для виявлення візуальних дефектів. Воно накладається до вже існуючих сценаріїв функціонального тестування, які виконуються в таких інструментах, як Selenium Webdriver, Cypress, WebdriverIO, Appium або PyTest. Оскільки сценарій керує додатком, то програма створює вебсторінки зі статичними візуальними елементами. Функціональне тестування змінює візуальні елементи, тому кожен крок функціонального тесту створює новий стан інтерфейсу користувача, який можна візуально перевірити. Автоматичне візуальне тестування походить від функціонального тестування.

Але замість того, щоб писати сотні і тисячі тверджень, щоб перевірити властивості кожного візуального елемента, спеціалізовані інструменти перевіряють зовнішній вигляд всього екрана лише одним перевірочним оператором. Таким чином, можна створювати тестові сценарії, які значно простіші та легші в обслуговуванні та підтримці.

2.1.4 Тестування скриншотів

Як вже було зазначено вище, тестування розмітки та CSS напряму є неефективним через значне збільшення коду методів перевірок, складності підтримки через постійну зміну назв стилів, а також неможливість перевірки ряду візуальних дефектів, як-от зміщення елемента на сторінці. Тому цілком природно перейти до тестування знімків з екранів повністю, тобто до тестування скриншотів.

Найбільш примітивний спосіб тестування знімків екрану полягає в фіксуванні растрового зображення, тобто сітки пікселів, в різних точках тестового запуску з подальшим порівнянням його з базовим зображенням. Алгоритми тестування моментальних знімків є дуже спрощеними: перебираємо кожну пару пікселів, а потім перевіряємо, чи збігається шістнадцятковий код кольору. Якщо коди кольорів відрізняються, то фіксуємо це як візуальну помилку. На Рисунку 2.2 зображено приклад такого попіксельного порівняння.

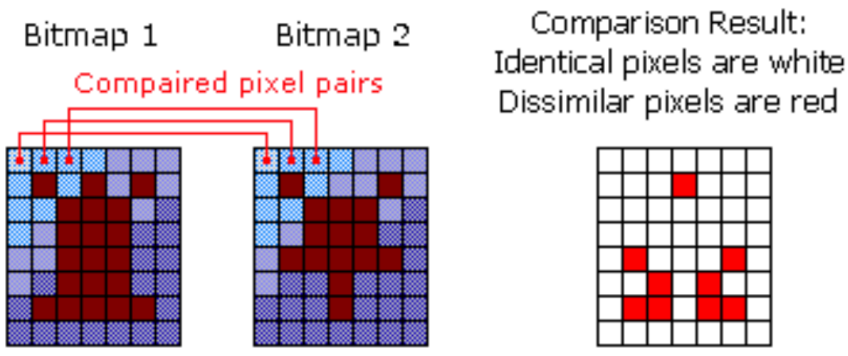


Рисунок 2.2 – Прямолінійне порівняння пар пікселів [15]

Оскільки такі алгоритми створити можна відносно легко, існує ряд комерційних інструментів для тестування знімків з відкритим кодом. На відміну від мануальних тестувальників, ці інструменти можуть швидко і послідовно виявляти відмінності в пікселях. Узагальнено на ринку їх називають інструменти Pixel Perfect.

В такого підходу, однак, є безліч недоліків. По-перше, на жаль, пікселі не є візуальними елементами самі по собі. Алгоритми згладжування шрифтів, зміна розміру зображення, графічні карти і навіть алгоритми візуалізації створюють різницю в пікселях. Це спричинює те, що в результаті інструмент, який очікує точного збігу пікселів між двома зображеннями, може бути заповнений піксельними відмінностями. Можна спробувати це трохи нівелювати, надавши користувачеві параметри, які можна налаштувати, наприклад, кількість пікселів, які можуть відрізнятися між двома зображеннями. Але цього замало. Існує багато прикладів, коли зображення можуть здаватися ідентичними, але їхні пікселі при цьому не збігатимуться. Тому порівняння зображень буде не точним, причин для цього багато, як-от:

а) наведення миші. Навіть якщо курсор миші не захоплений на знімку екрана, він насправді може бути наведений на елемент, на який це може візуально вплинути. Оскільки змінювати зовнішній вигляд елемента в CSS, JavaScript, JQuery, React, Angular, Vue та інших технологіях розробника інтерфейсу неважко, цей шаблон досить поширений у веб-додатках. На Рисунку 2.3 можна побачити приклад цього: на верхньому знімку екрана курсор миші не наводиться на кнопку пошуку Google, а на нижньому скриншоті – наведений. Це спричиняє візуальну різницю в елементах і відповідно в пікселях, проте з точки зору інтерфейсу обидва знімки є непомилковими і при тестовому порівнянні було б добре, якби це не позначати як візуальний збій.

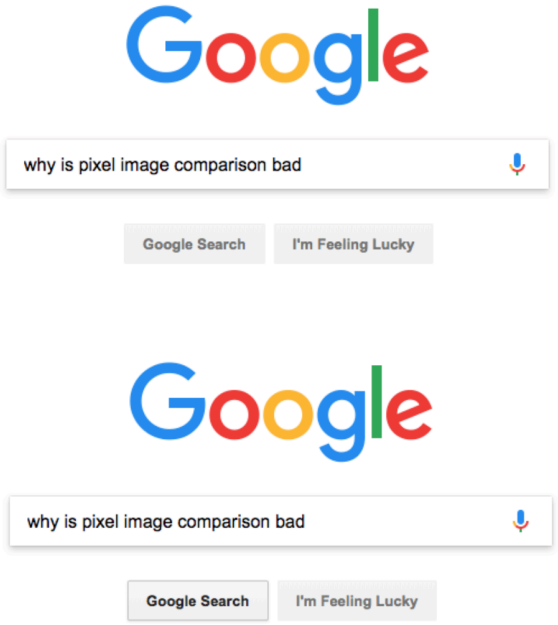


Рисунок 2.3 – Візуальний вплив наведення курсора на елемент [15]

б) каретка введення. Якщо є текстовий елемент, який знаходиться у фокусі, у ньому може бути миготливий курсор. Припустимо, що під час виконання базового тесту було зроблено знімок екрана. Потім, при проведенні тесту, знімок екрана ловить сторінку в той момент, коли курсор невидимий. Спрощене порівняння пікселів видасть помилку в цьому випадку, однак інтерфейс користувача є абсолютно коректним. Рисунок 2.4 зображає приклад цього – на верхньому знімку екрана видно курсор, а в нижній частині – ні.

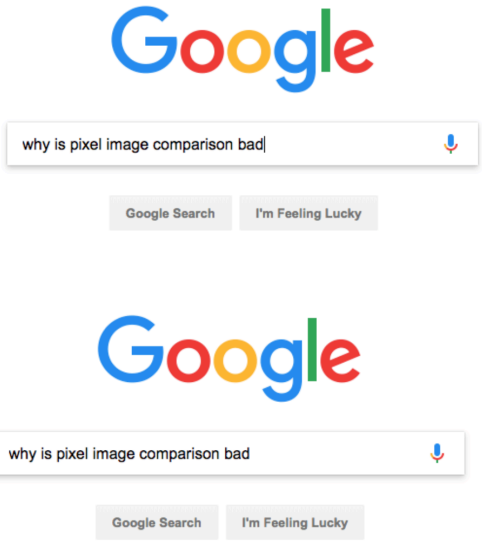


Рисунок 2.4 – Вплив каретки введення на візуальну різницю [15]

в) згладжування шрифтів. Кожна операційна система відтворює шрифти по-різному, щоб зробити їх більш гладкими і легшими для читання. Питання не тільки в різних підходах Mac та Windows; навіть оновлення операційної системи може змінити згладжування шрифтів. Наприклад, Windows використовує технологію ClearType для цього, а конкретні методи варіюються залежно від версії. Крім того, тестувальники самі можуть змінити їх в налаштуваннях, не усвідомлюючи, як це вплине на візуальне тестування. Рисунок 2.5 зображає, як різні налаштування дисплея для згладжування шрифтів впливають на прямолінійне піксельне порівняння, спричинюючи появу величезної кількості хибно позитивних відповідей під час візуального автоматизованого тестування.

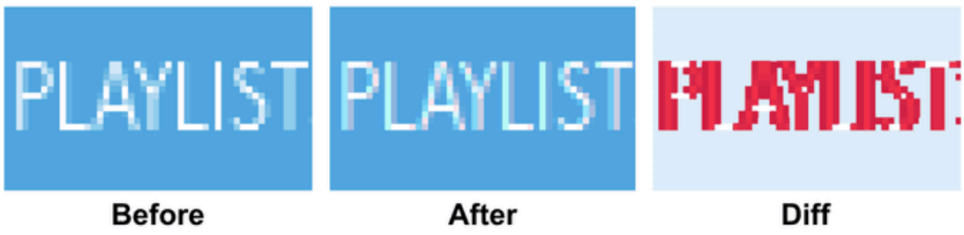


Рисунок 2.5 – Приклад хибно позитивного результату при порівнянні скриншотів [4]

г) різні дисплеї. Екрани з високою роздільною здатністю, як-от Apple Retina Display, мають більшу щільність пікселів – настільки, що пікселі не видно людському оку. Знімки екрана, зроблений на екрані Retina та моніторі з меншою щільністю пікселів, не зможе порівняти просте піксельне порівняння, навіть якщо для людського ока немає видимої різниці.

Отже, необхідні інші методи порівняння двох зображень. Для цього знімки зображень потрібно попередньо обробити і проаналізувати, використовуючи алгоритми комп’ютерного зору.

2.2 Методи обробки та аналізу зображень

Сучасне покоління автоматизованого візуального тестування використовує клас алгоритмів штучного інтелекту, які називаються комп'ютерним зором, як основний механізм для візуального порівняння. Він поєднує алгоритм навчання для інтерпретації зв'язку між відтвореною сторінкою та передбачуваним відображенням візуальних елементів із реальними візуальними елементами та місцеположеннями. Як і піксельні інструменти, автоматизоване візуальне тестування на основі комп'ютерного зору робить знімки сторінок під час виконання функціональних тестів. Проте в процесі порівняння використовуються певні алгоритми, щоб визначити, коли виникли помилки.

Вибираючи метод обробки знімка екрану графічного інтерфейсу, слід враховувати такі фактори: згладжування шрифту; різні ситуації, в яких стилі та елементи відображаються різними браузерами; можливість порівнювати зображення з різною роздільною здатністю.

2.2.1 Сегментація зображень

Сегментація зображень – це підобласть комп'ютерного зору та цифрової обробки зображень, яка спрямована на групування подібних областей або сегментів зображення під відповідними мітками класів. Можна сказати, що завдання формування сегментів є еквівалентним до групування пікселів. Сегментація є розширенням задачі класифікації зображень, де, окрім неї, ми виконуємо ще і локалізацію. Таким чином, сегментація зображень є надмножиною класифікації зображень, в якій модель точно вказує, де присутній відповідний об'єкт, шляхом окреслення його меж. Мета цього процесу полягає в тому, щоб змінити представлення зображення на щось, що є більш значущим і легшим для аналізу [7].

Існує багато алгоритмів сегмантації зображень – всіх їх можна умовна розділити на декілька груп, залежно від того, на чому вони засновані: на порозі, на регіонах, на ребрах, на вододолі, на кластерах [7]. Розглянемо деякі з цих алгоритмів.

2.2.1.1 K-Means алгоритм

Алгоритм K-Means (K-середніх) був запропонований в 1979 році, але залишається актуальним і сьогодні. Його ідея досить проста, а алгоритм працює досить ефективно. Проте оптимальність рішень, отриманих за допомогою алгоритму K-Means, не гарантується. Крім того, недоліком алгоритму є необхідність заздалегідь знати кількість кластерів, тобто задавати той самий параметр k.

Формально вирішення проблеми кластеризації полягає в тому, щоб «позначити» кожен із існуючих об’єктів – присвоїти йому номер певного класу. Алгоритм K-середніх передбачає, що об’єкти поділяються на класи таким чином, що відмінності («відстані») між об’єктами одного класу мінімізуються, а відмінності між об’єктами різних класів максимізуються [8]. Поточний алгоритм є основою майже всіх алгоритмів нечіткої кластеризації, і його детальний аналіз допомагає краще зрозуміти принципи, закладені в більш складні алгоритми.

Узагальнено алгоритм являє собою ітераційну процедуру з такими кроками:

1) вибрати кількість кластерів K;
2) вибрати у випадковому порядку K точок центроїди (не обов’язково з усього набору даних);
3) призначити кожну точку даних найближчому центроїду, який утворює K кластерів;
4) обчислити і помістити новий центроїд кожного кластера;

5) перепризначити кожну точку даних новому найближчому центроїду. Якщо якесь перепризначення відбулося, перейдіть до кроку 4, інакше – модель готова.

Мета кластеризації K-Means — мінімізувати суму квадратів відстаней між усіма точками та центром кластера [8]. Більш математичний опис цього алгоритму наведено нижче:

1) ініціалізувати початкову матрицю розбиття U випадковим чином і вибирати точність δ, яка буде використана для завершення алгоритму, встановити номер ітерації l = 0 ;
2) щоб визначити центри кластерів використовується така формула:

$$c_l^{(i)} = \frac{\sum_{j=1}^{d} u_{ij} m_j}{\sum_{j=1}^{d} u_{ij}}, 1 \le \mathrm{i} \le \mathrm{c}\,, \tag{2.1}$$

де $c_l^{(i)}$– центри кластерів, *d* – розміри об'єкту, *c* – кількість кластерів, *l* – кількість на поточній ітерації, *u* – матриця розбиття об'єкта *m*, *m* – об'єкт, який досліджується;

3) оновити розбиття матриці за формулою:

$$u_{ij}^{(l)} = \begin{cases} 1, & \text{якщо } d(m_j, c_i) = \min\limits_{l \le k \le c} d\,(m_j, c_k) \\ 0, & \text{в інших випадках} \end{cases}, \tag{2.2}$$

де $u_{ij}^{(l)}$ – елемент розбиття матриці, $d(m_j, c_i)$ – обрана метрика, *c* – кластер, *m* – об'єкт, який досліджується;

4) перевірити обмеження $\left\|U^{(l)} - U^{(l-1)}\right\| < \delta$, де *U* – матриця розбиття, а δ – обрана точність. Якщо обмеження задоволено, то закінчити процес, в іншому випадку – повернутись на крок 2 з номером ітерації *l=l+1*.

Якщо мова йде про сегментацію зображень, які представляються у вигляді матриці з R, G, B вимірами, то спочатку ми мусимо змінити його форму до масиву розміром Mx3 (M — кількість пікселів у зображенні). І після кластеризації застосовуємо значення центроїдів (це також R,G,B) до всіх пікселів, так щоб отримане зображення мало певну кількість кольорів.

Основним недоліком цього алгоритму через дискретність елементів матриці розбиття є великий розмір просторового розбиття. Щоб усунути цей недолік, необхідно представити елементи матриці розбиття числами від 0 до 1. Тобто належність елемента даних до кластера визначається функцією належності, таким чином елемент даних може бути членом кількох кластерів з різним ступенем. Іншими недоліками та обмеженнями алгоритму є необхідність заздалегідь знати кількість кластерів і чутливість до вибору початкових центрів кластерів. Класичний варіант реалізує випадковий вибір кластерів, що дуже часто є джерелом помилок і не справляється із завданням, коли об'єкт належить до різних кластерів однаково або не належить жодному [8].

2.2.1.2 Mean Shift алгоритм

Головний недолік попереднього алгоритму можна було б вирішити, якщо не задавати кількість кластерів заздалегідь. А це цілком реально, адже її можливо визначити на основі вхідних даних. Напрямок до центроїда найближчого скупчення визначається тим, де розташована більшість найближчих точок.

Mean Shift – є ітераційним методом, що дозволяє вирішити цю проблему. Середнє зміщення було запропоновано Фукунага та Хостетлером і розширено для використання в інших областях, таких як комп’ютерний зір. Метод розглядає функціональний простір як емпіричну функцію щільності ймовірності. Якщо вхідними даними є набір точок, то метод «середнього зсуву» розглядає їх як вибірку з основної функції щільності ймовірності. Якщо у функціональному просторі є щільні області, вони належать до форми функції щільності

ймовірності [5]. Кластери можна ідентифікувати, пов’язані з цією формою, використовуючи саме метод Mean Shift.

Кожен піксель пов’язується з піком щільності ймовірності зображення. Цей пік обчислюється шляхом визначення околу пікселя, а потім обчислення середнього значення пікселя, що лежить в цьому околі. Потім він зміщується до середнього, і аналогічні кроки повторюються до конвергенції [5]. Результат алгоритму параметризується лише розміром ядра (пропускною здатністю) і тому вимагає менше ручного втручання в порівнянні з іншими алгоритмами. Однак визначити відповідну пропускну здатність – непросте завдання: занадто велика або мала може призвести до надмірної або недостатньої сегментації.

Фунція щільності визначається таким чином:

$$f(s) = \frac{1}{Nh^d} \sum_{i=1}^{N} K\left(\frac{s - s_i}{h}\right), \tag{2.3}$$

де s_i $(i = 1, 2, .., n)$ – послідовність точок зображення в d-розмірному просторі, h – окіл точки (пропускна здатність), N – кількість точок (пікселів).

Максимуми функції розташовані там, де точки зображення ущільнюються в просторі. Пікселі, які належать до одного локального максимуму, об’єднуються в один сегмент. Виявляється, щоб знайти, до якого з центрів ущільнення належить піксель, потрібно зробити крок уздовж градієнта, щоб знайти найближчий локальний максимум (див. Рисунок 2.6).

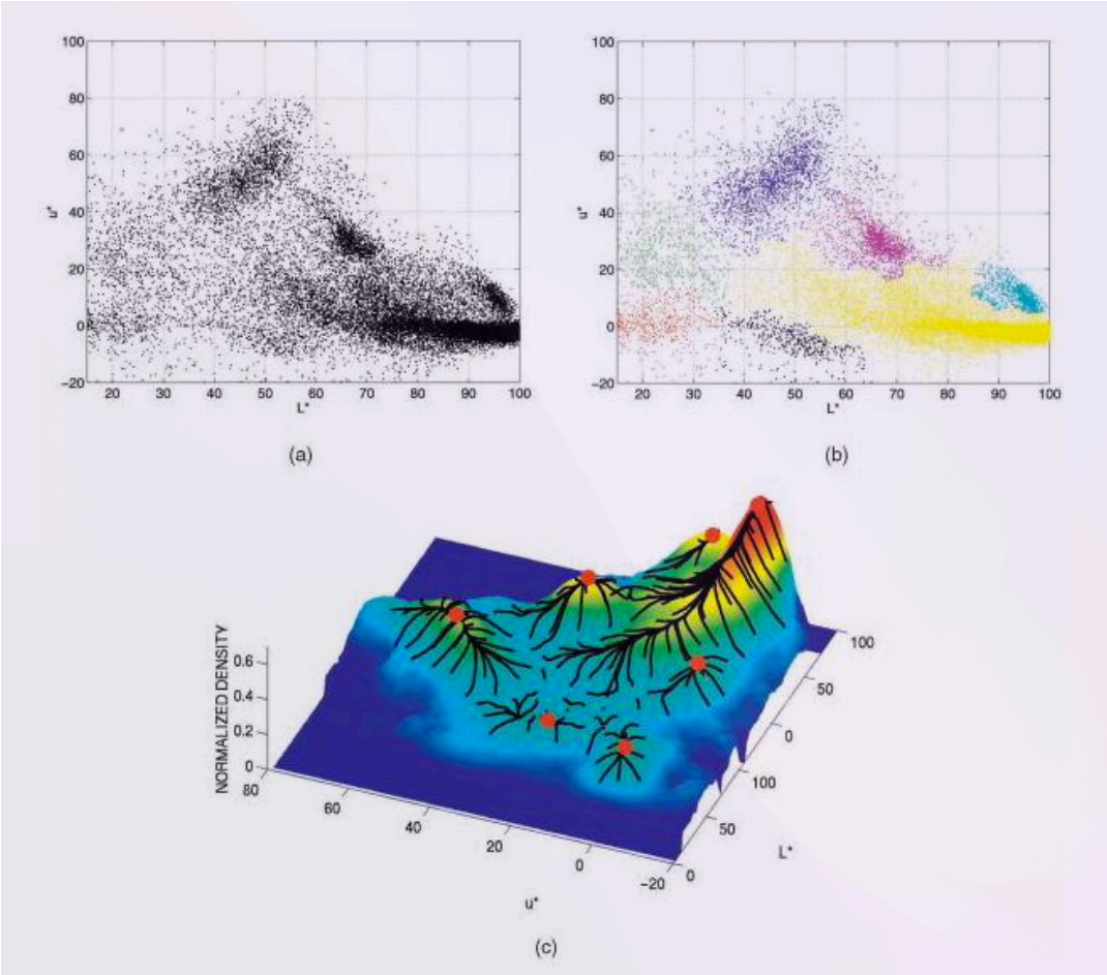


Рисунок 2.6 – пікселі зображення: (a) пікселі в двовимірному просторі, (b) пікселі, які досягають одного локального максимуму, пофарбовані в той самий колір, (c) функція щільності, максимуми відповідають місцям найбільшої концентрації точок [5].

Цей метод також є корисним для послаблення затінення або тональності в локалізованих об’єктах. Дослідження показують, що найбільш продуктивним він є, коли пропускна здатність h становить від 0,03 до 0,06.

2.2.2 Порівняння зображень

Індекс структурної схожості (SSIM) — це метод вимірювання подібності двох зображень. Структурна інформація — це ідея, що пікселі мають сильні взаємозалежності, особливо коли вони просторово близькі. Ці залежності несуть

важливу інформацію про структуру об’єктів у візуальній сцені. Маскування яскравості — це явище, при якому спотворення зображення (у цьому контексті) мають тенденцію бути менш помітними в яскравих областях, тоді як контрастне маскування — це явище, при якому спотворення стають менш помітними там, де є значна активність або «текстура» зображення [6]. Значення SSIM може змінюватися від -1 до 1, де 1 вказує на повну схожість двох зображень.

Цей метод був розроблений як покращення алгоритму MSE (Mean Squarred Method), головним недоліком якого був факт того, що великі відстані між інтенсивністю пікселів не обов’язково означають, що вміст зображень різко відрізняється. Головною перевагою SSIM є те, що він порівнює два вікна (тобто невеликі підвибірки), а не ціле зображення, як у MSE [6]. Це проводить до більш надійного підходу, який може врахувати зміни в самій структурі зображення.

MSE та SSIM – традиційні методи комп’ютерного зору та обробки зображень для порівняння зображень. Вони, як правило, найкраще працюють, коли зображення майже ідеально вирівняні (інакше розташування і значення пікселів не збігаються, даючи хибну оцінку подібності).

2.2.3 Пошук та видалення тексту з зображень

Інколи в процесі обробки чи аналізу зображення необхідно знайти всі текстові елементи і видалити їх, залишивши при цьому відсутні частини заповненими, щоб відсутність тексту була максимально непомітною. Найкраще для цієї задачі підходить технологія оптичного розпізнавання символів (OCR). Це технологія, що використовується для розрізнення друкованих або рукописних символів тексту всередині цифрових зображень фізичних документів, таких як сканований паперовий документ. Основний процес OCR включає вивчення тексту документа та переклад символів у код, який можна використовувати для обробки даних.

Першим кроком OCR є використання сканера для обробки фізичної форми документа. Після копіювання всіх сторінок OCR перетворює документ у

двоколірну або чорно-білу версію. Відскановане (або растрове) зображення аналізується на наявність світлих і темних областей, де темні області визначаються як символи, які потрібно розпізнати, а світлі ділянки визначаються як фон. Темні ділянки потім обробляються, щоб знайти букви алфавіту або цифри. Програми розпізнавання тексту можуть відрізнятися за своїми техніками, але зазвичай передбачають націлювання на один символ, слово або блок тексту за раз. Потім символи ідентифікуються за допомогою певних алгоритмів.

Весь процес видалення тексту і зафарбовування пустих частин зображення зображено на Рисунку 2.7 і виглядає приблизно так:

1) визначити текст на зображенні та отримати координати рамки кожного тексту за допомогою натренованої моделі OCR;
2) для кожної обмежувальної рамки застосувати маску, щоб вказати алгоритму, яку частину зображення необхідно замалювати;
3) застосувати алгоритм зафарбовування, щоб зафарбувати замасковані ділянки зображення для отримання зображення без тексту.

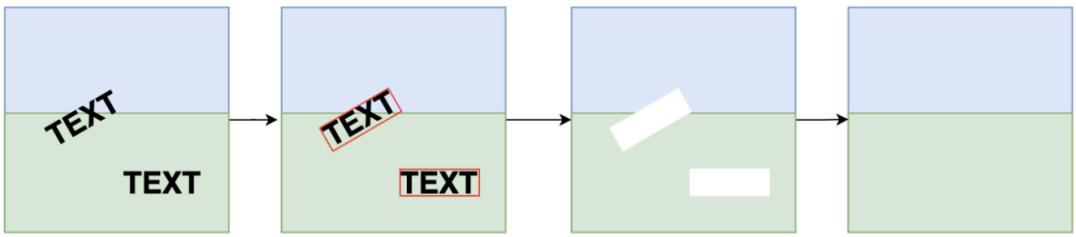


Рисунок 2.7 – Ілюстрація процесу видалення і зафарбовування тексту з зображення

Базова ідея алгоритмів зафарбовування досить проста: замінити пікселі бажаних областей сусідніми пікселями, щоб вони виглядали як околиці. Конкретні імплементації можуть різнитися, найпопулярнішими є такі:

а) метод швидкого маршування. Алгоритм починається від кордону регіону, який треба зафарбувати, і йде всередину області, поступово заповнюючи спочатку все на кордоні. Для зафарбовування потрібен невеликий окіл навколо пікселя. Цей піксель замінюється нормалізованою зваженою сумою всіх відомих

пікселів по сусідству. В цьому процесі вибір ваги є дуже важливим. Більша вага надається тим пікселям, які лежать поблизу точки, близько до нормалі кордону, і тим, що лежать на контурах кордону. Після того, як піксель зафарбовано, він переміщається до наступного найближчого пікселя [10].

б) другий алгоритм заснований на диференціальних рівняннях і динаміці рідини. Основний принцип - евристичний. Спочатку він рухається по краях від відомих областей до невідомих, оскільки краї мають бути безперервними. Він продовжує лінії, що з'єднують точки з однаковою інтенсивністю (так само, як контури з'єднують точки з однаковою висотою), узгоджуючи вектори градієнта на кордоні області малювання. Для цього використовуються деякі методи з гідродинаміки. Коли вони отримані, колір заповнюється, щоб зменшити мінімальну дисперсію в цій області [1].

РОЗДІЛ 3. ПРАКТИЧНА ЧАСТИНА. ПОБУДОВА ФРЕЙМВОРКУ ДЛЯ ВІЗУАЛЬНОГО РЕГРЕСІЙНОГО ТЕСТУВАННЯ

3.1 Гіпотеза

Висуваємо гіпотезу, що повинно бути достатньо мати лише функціональний набір тестів для наскрізного тестування програми. Скрипти функціонального тестування взаємодіють із програмою достатньою мірою, тому інший сценарій, що робить знімки екрана під час такої взаємодії, є зайвим. Ця надлишковість коштує дорого, оскільки командам із забезпечення якості потрібно підтримувати більше наборів тестів. Зміни в програмі потрібно було б вносити в більшій кількості місць, що дозволило б помилкам проникати в тестові набори з більшою вірогідністю.

У той же час вважаємо, що сценарій функціонального тестування має бути максимально абстрагований від візуального тестування. Це означає, що явні виклики методів, які роблять знімки екрана, мають бути необов’язковими. Функціональний сценарій тестування повинен автоматично робити знімки екрана під час тестування в кожному важливому стані програми. Таким чином ми досягнемо кращої читабельності скриптів функціональних тестів.

Безумовно, має бути можливість явного створення скриншотів, однак потреба сприяти такому варіанту має бути спорадичною. Це може бути досягнуто завдяки змінам в параметрах конфігурації. Тестер повинен мати можливість роботи налаштування на глобальному рівні, тобто для всіх тестів з одного місця, а також на рівні окремого індивідуального тесту, визначаючи в яких ситуаціях потрібно зробити знімок екрана.

База скриншотів, яка буде служити орієнтиром для правильного стану вебдодатка, буде створена автоматично під час першого запуску створеного фреймворку. Скриншоти мають бути доступними всім зацікавленим сторонам (розробникам, тестувальникам тощо).

Життєздатним рішенням є також впровадження веб-інтерфейсу як системи для управління результатами візуального порівняння тестування. У цій системі користувач повинен мати можливість приймати рішення про результати візуального тестування – точніше оцінити результати, чи є помилка в програмі чи ні. Цей веб-інтерфейс сприятиме співпраці між зацікавленими сторонами.

Дотримуючись вищезгаданих принципів, ми досягнемо більшої ефективності команди із забезпечення якості, в тому числі значно скоротимо кількість часу, який їм потрібно витрачати на ручне тестування.

3.2 Процес

Беручи до уваги всі мотиви, пояснені раніше, варто розглянути те, як ми можемо побудувати автоматизований процес впровадження візуальних регресійних тестів у наші проекти та деякі передові практики, пов’язані з цим.

Отже, побудований фреймворк має мати робочий процес, що виконується у такій послідовності:

1. Тестувальники пишуть сценарії тестування, визначають, що треба знімати на скриншотах і де їх робити з програми. Крім того, ці сценарії повинні обробляти будь-яку необхідну взаємодію користувача.
2. Виконання набору написаний тестів уперше для створення шаблонів, тобто знімків екрану з якими будуть порівнюватися нові під час виконання наступних тестів.
3. Виконання набору візуальних тестів вдруге і більше разів, щоб створити нові знімки екрана для порівняння з шаблонами, згенерованими під час першого запуску.
4. Інструмент запускає алгоритм для порівняння нещодавно зроблених скриншотів іх базовими шаблонами, які були зроблені під час першого запуску.
5. Створення звіту, що містить усі відмінності, виявлені під час порівняння.

6. Ревізор аналізує звіт і визначає, чи очікувалися внесені зміни, якщо такі були. Якщо зміни у графічному інтерфейсі є очікуваними, ми повертаємося до кроку 2.
7. Після того, як усі необхідні зміни були застосовані та кінцевий результат відповідає очікуванням, шаблонні (базові) зображення необхідно оновити для майбутніх тестів.

Загальний процес можна описати наступними підпроцесами.

3.2.1 Перше виконання функціональних тестів

На Рисунку 3.1 показано кроки, необхідні для створення шаблонів. Це обов’язкова умова для візуального тестування.

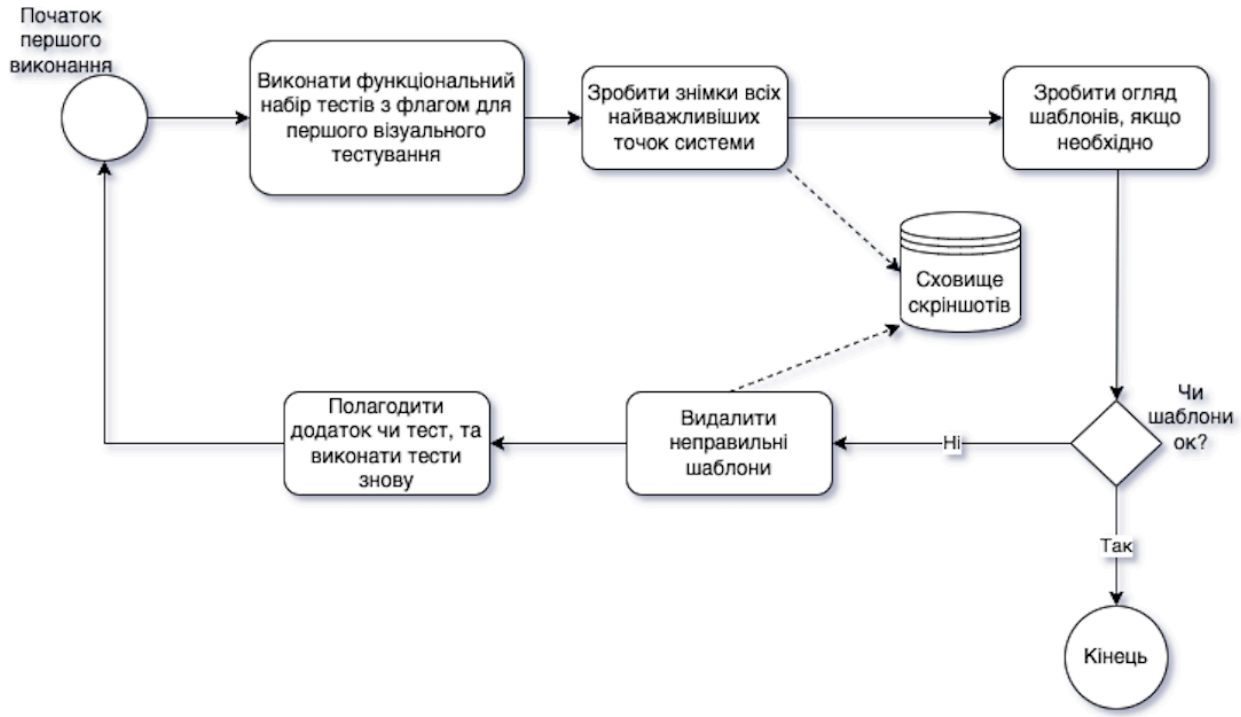


Рисунок 3.1 – Процес створення шаблонів під час першого виконання функціональних тестів

Під час виконання тестів створюються скриншоти. Якщо всі функціональні тести пройшли, ці знімки екрана можуть бути заявлені як

шаблони та завантажені в сховище. Скриншоти, створені в тестах, які пройшли невдало, ігноруються: вони не включаються у візуальне тестування. Такі функціональні тести необхідно змінити та полагодити, зробити прохідними, а потім вже включити їх до візуального тестування.

Додатковою частиною є перегляд скриншотів. Якщо є якісь проблеми з шаблонами, їх слід викинути, а тести повторити. Якщо шаблони правильні, тестувальник може перейти до наступних запусків набору тестів, щоб почати фактичне візуальне тестування.

3.2.2 Подальше виконання візуального тестування

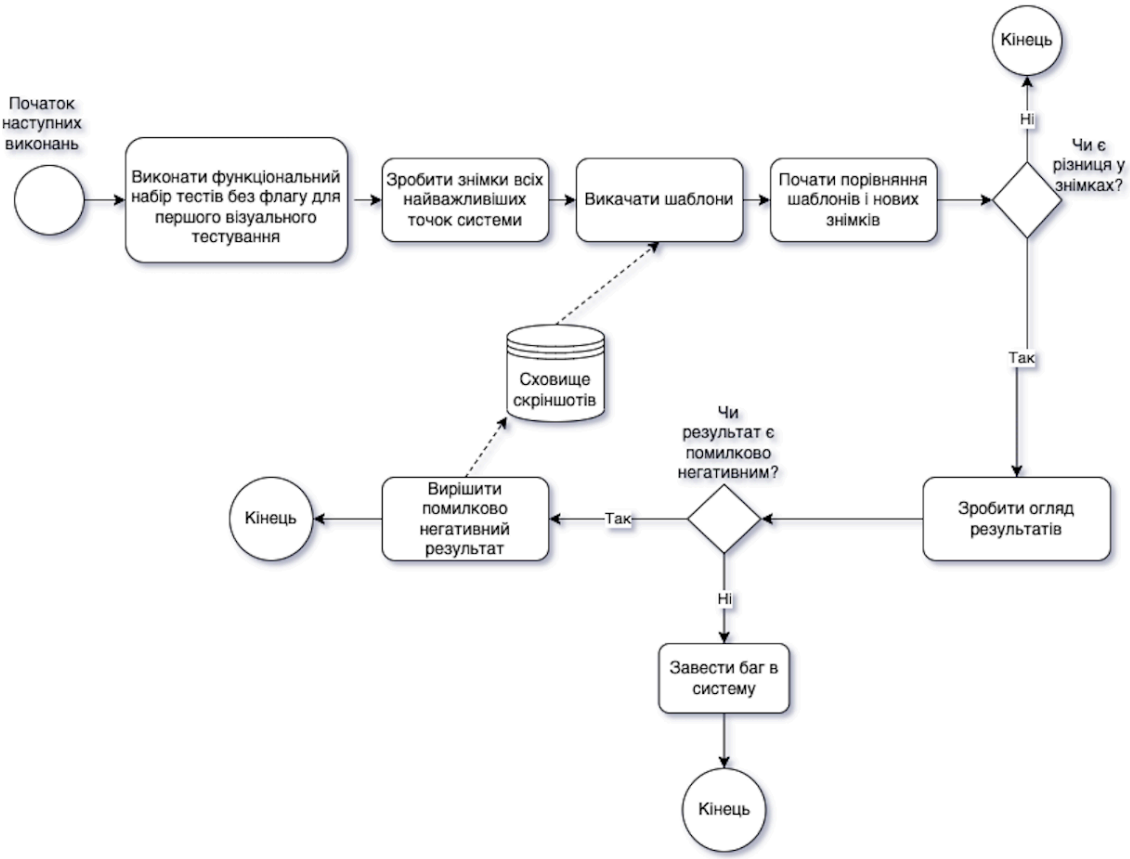


Рисунок 3.2 – Подальше виконання візуального тестування разом з функціональними тестами

На Рисунку 3.2 показано, як буде виглядати подальше виконання візуального тестування разом із функціональним тестуванням. Перший крок такий же, як і в попередньому процесі, виконуються функціональні тести та

робляться скриншоти. По-друге, шаблони завантажуються зі сховища, і після цього можна почати власне візуальне тестування.

Щойно створені знімки екрана за певним алгоритмом порівнюються із завантаженими шаблонами. Результат зберігається у файлі дескриптора, а відмінності між конкретним шаблоном і зразком візуально зберігаються у згенерованому зображенні. Якщо виявлено якісь відмінності, створені зображення разом із відповідними зразками та шаблонами завантажуються на віддалене сховище.

Користувачі також повинні мати можливість переглядати результати, де вони знайдуть конкретний запуск відповідно до відмітки часу, коли тестовий набір був виконаний. Вони повинні вміти оцінювати результати з відображеним триплетом, що складається з шаблону, знімка екрану та їх відмінностях. Також має бути можливість позначити, чи є це хибнонегативний результат тесту, і в такому випадку потрібно мати можливість вжити заходів, щоб запобігти таким результатам у майбутньому.

Інструмент має бути легко налаштовуваним, щоб такі хибнонегативні результати були досить рідкісними, натомість невдалі тести візуального порівняння повинні виявити помилку в програмі, що тестується. У цьому випадку користувач несе відповідальність за подання такої помилки у відповідну систему відстежування помилок. Згенерований шаблон, знімок екрану і зображення-відмінність можна використовувати як корисне вкладення до звіту про проблему, який краще описує фактичний і очікуваний результат.

3.3 Технології розробки фреймворку

В основі будь-якого проекту автоматизації тестування лежить тестове «ядро» – базовий інструмент, що оброблятиме структуру тестового випадку, забезпечуватиме виконання тесту та звітування про результати. Це основа, на яку можна додавати додаткові пакети, плагіни та інший код для розширення функціональності.

Для початку розробки власного фреймворку візьмемо за основу готовий інструмент з відкритим кодом для написання коду на Python – Pytest. З усіх проаналізованих фреймворків він виявився найбільш універсальним, легко масштабованим та простим у налаштуванні. Різноманітні плагіни можуть покращувати покриття коду, надавати гарні звіти та забезпечувати паралельне виконання.

Pytest також може інтегруватися з іншими фреймворками. І найпершим розширенням, необхідним для реалізації UI тестування веб-додатків у браузерах, є Selenium WebDriver. Це є реалізацією стандарту WebDriver – програмованого інтерфейсу для взаємодії з реальними веб-браузерами. Він дає змогу автоматизувати тестування, щоб відкривати браузер, надсилати кліки, вводити клавіші, очищати текст і завершувати сесію в браузері. Інтерфейс WebDriver є рекомендацією W3C.

WebDriver працює по протоколу JSON Wire Protocol. Мовні прив’язки кодують кожну взаємодію за допомогою JSON і надсилають їх як запити REST API до драйвера браузера. Для того, щоб WebDriver мав змогу автоматизувати користувацькі взаємодії з якимсь браузером необхідно також встановити драйвер цього браузера. Драйвер є окремим виконуваним файлом на тестовій машині. Він діє як проксі-сервер між абонентом взаємодії та самим браузером. Він отримує запити JSON на взаємодію та надсилає їх у браузер за допомогою HTTP. Для кожного браузера також потрібен свій власний тип драйвера, встановлений на тій же машині, що й браузер, і доступний із системного шляху. Наприклад, для Google Chrome потрібен ChromeDriver. На Рисунку 3.4 схематично зображено як працює Selenium WebDriver.

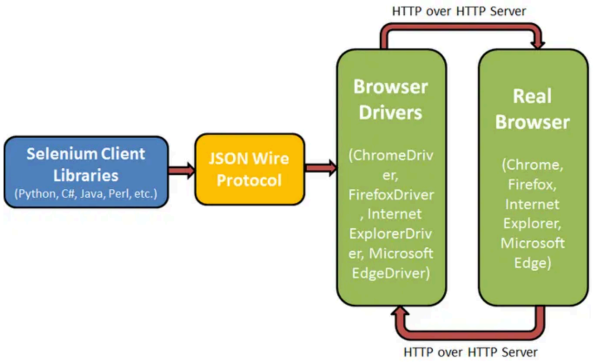


Рисунок 3.4 – Схема роботи Selenium WebDriver [14]

Для аналізу та обробки знімків екрану будемо використовувати бібліотеку для комп'ютерного зору OpenCV. Це величезна бібліотека з відкритим вихідним кодом, яка використовується в областях, що працюють на основі алгоритмів штучного інтелекту або машинного навчання, а також для виконання завдань, які потребують обробки зображень. Після інтеграції з іншими бібліотеками, такими як NumPy, Python може обробляти структуру масиву OpenCV для аналізу. Ідентифікація шаблонів зображення та кількох його ознак потребує використання векторного простору та виконання математичних операцій над цими ознаками.

Також для деяких задач будемо використовувати бібліотеку Keras. Це найпопулярніша бібліотека, яка використовується для глибокого навчання, і надає API для різних навчальних програм. Він забезпечує ефективний і зручний спосіб навчання моделей і програм. Перевагою Keras в нашому випадку є також те, що він повністю написаний на Python.

3.4 Опис реалізації

3.4.1 Page Objects

В Pytest тестові випадки оформлені як функції, а не класи. Через це втрачається структуризація тестів, особливо якщо веб-застосунок для тестування складається з десятків сторінок, які містять сотні різних елементів і веб-компонентів. Техніка Page Objects Pattern забезпечує рішення для роботи з

декількома веб-сторінками, запобігає небажаному дублюванню коду та забезпечує його читабельність та можливість повторного використання. Загалом, кожна сторінка в нашій програмі буде представлена власним унікальним класом, і перевірка елементів сторінки буде реалізована саме там. Кожен клас представлений атрибутами локаторів (селекторів CSS чи XPath) елементів та методами взаємодії з ними. А замість того, щоб робити напряму виклики WebDriver, тест викликає Page Objects метод. Цей патерн є одним з найпоширеніших, які використовуються для автоматизації тестування веб-інтерфейсів.

Використовуючи Chrome DevTools ми визначаємо локатори необхідних нам елементів, для взаємодії з ними, як-от кліки, заповнення полів тощо (див. Рисунок 3.5).

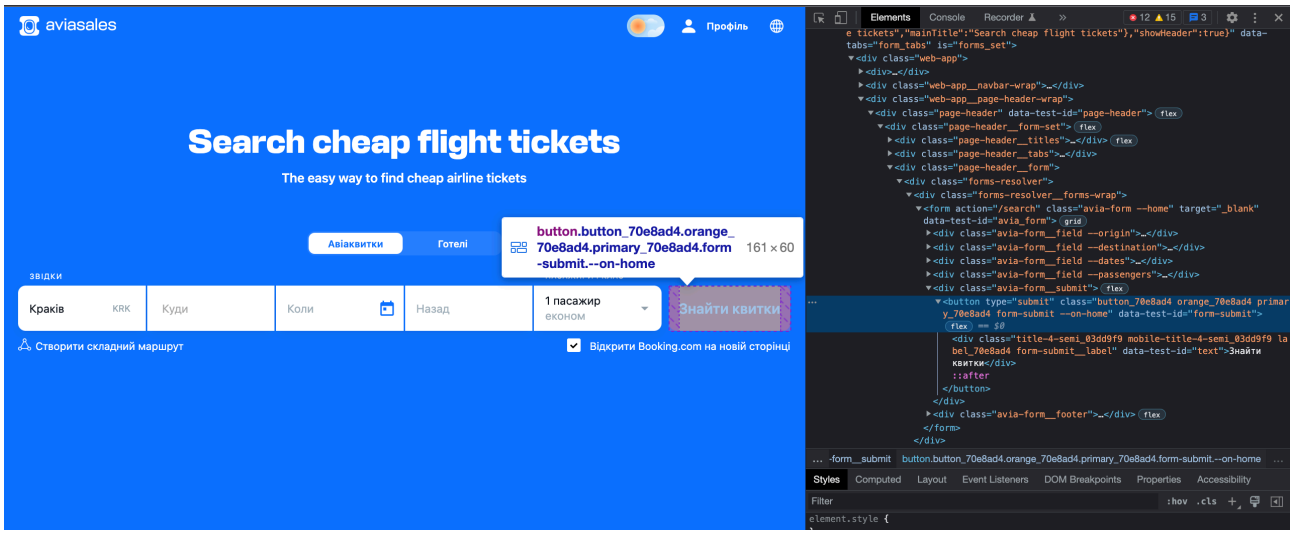


Рисунок 3.5 – Процес визначення локаторів необхідних елементів за допомогою Chrome DevTools

Після цього ми включаємо усі необхідні нам локатори у клас об’єкту саме цієї головної сторінки веб-сайту, і прописуємо методи взаємодії з ними, які необхідні для перевірок очікуваних і реальних результатів поведінки системи (див. Додаток А). Кожна сторінка сайту, яка тестується, матиме свій окремий клас. Таким чином код стає чистішим, структурованішим та його можна легко повторно використовувати.

3.4.2 Порівняння знімків

Як зазначено в розділі 3.2, ми маємо мати можливість запускати написані тести з певним флагом. Якщо флаг виключено, то функціональні тести відпрацьовують без візуального тестування, при цьому просто роблячи скриншоти тестованої системи в певних точках, які ми заздалегідь визначаємо. Знімки сторінок, що були зроблені, вважатимуться базовими або шаблонами, з якими ми будемо порівнювати скриншоти в наступних запусках тестів уже із включеним флагом.

Для порівняння будемо використовувати алгоритм SSIM, описаний в розділі 2.2.2. Реалізацію алгоритму на Python можна побачити в Додатку Ж. Як вже зазначалось, цей алгоритм повертає певне значення схожості двох зображень. Якщо це значення вище за якусь константу – то наш фреймворк видаватиме негативний результат тестування і звітуватиме про це, прикріпивши 3 зображення для розгляду користувачу – очікуваний знімок (наш шаблон, з яким ми порівнюємо), реальний знімок (зроблений в поточному запуску тесту), та різницю між ними, підкреслену червоними прямокутниками.

Постає питання, як порівнювати ці скриншоти так, щоб зробити кількість хибно негативних результатів максимально невеликою? Як вже зазначалось в розділі 2.1.4 існує ряд проблем, через які прямолінійне попіксельне порівняння є неефективним. Знімки екранів мають пройти певний аналіз та обробку перед тим, як порівнюватись.

Запропонований в цій роботі алгоритм полягає в тому, що спочатку необхідно позбутися усіх текстових елементів на сторінці. Немає сенсу тестувати його саме візуально через проблеми із згладжуванням шрифтів, а також через те, що текст – це зазвичай найбільш динамічно змінювана частинка веб-сайту. До того ж, це дозволить тестувати сайти незалежно від їх локалізації тобто мови перекладу. Проте якщо у тестувальника є об’єктивна причина тестувати саме текстові написи, то це краще робити звичайним функціональним тестуванням за рахунок видобування тексту з веб-елементів за їхніми локаторами.

Використовуватимемо Keras OCR для виявлення тексту всередині зображень, і OpenCV для зафарбовування – процесу, коли відсутні частини фотографії заповнюються для створення повного зображення, – щоб видалити знайдений текст. Keras OCR пропонує вже навчену модель з визначеними вагами для детектора і розпізнавача тексту. При проходженні зображення через цю модель, воно поверне кортеж (слово, поле), де поле містить координати (x, y) чотирьох кутів поля слова. Маючи ці координати ми можемо застосувати алгоритм зафарбовування з бібліотеки OpenCV. Бібліотека має два можливі алгоритми малювання та дозволяє застосовувати прямокутні, кругові або лінійні маски. В нашому випадку використовуватимемо лінійні маски, оскільки вони більш гнучкі для покриття тексту з різною орієнтацією: прямокутні маски будуть добре працювати лише для слів, паралельних або перпендикулярних осі x, а кругові маски охоплюють більшу область, ніж необхідно. Щоб застосувати маску, потрібно вказати координати початкової та кінцевої точок лінії, а також товщину лінії. Початкова точка буде серединою між верхнім лівим кутом і нижнім лівим кутом поля, а кінцевою точкою буде середина між верхнім правим кутом і нижнім правим кутом. Для товщини обчислимо довжину лінії між верхнім лівим кутом і нижнім лівим кутом. Зрештою, для зафарбовування зображення використовуватимемо алгоритм з OpenCV, що заснований на диференціальних рівняннях і динаміці рідини, який описано в розділі 2.2.3. Код повної реалізації видалення тексту на Python наведено у Додатку К.

Для того, щоб отримати чітко виділені візуальні елементи на сторінці та ті самі результати для зображень з різною роздільною здатністю, використовуватимемо алгоритми сегментації, описані в розділі 2.2.1. Код алгоритмів K-Means та Mean Shift на Python наведено у Додатку Л.

3.4.3 Звітування про результати тестування

Для того, щоб тестувальник міг проінспектувати результати запуску тестів, та ознайомитись зі знімками екрану, якщо очікувані відрізняються від реальних,

оснастимо наш фреймворк функціями звітування на окремій веб-платформі. Для цього використовуватиме плагін Report Portal – інструментальну панель автоматизації тестування. За допомогою цього порталу звітів ми маємо доступ до виконання всіх автоматичних тестів у проекті та легко знаходимо інформацію про будь-який тестовий елемент (див. Рисунок 3.6). Можна вручну визначити причини збою тестового випадку та встановити для нього тип дефекту: помилка продукту, системна проблема або помилка фреймворку, задля того, аби візуалізувати структуру усіх збоїв. Також, великою перевагою є те, що Report Portal можна інтегрувати із системою відстежування помилок, і пов’язати створення карток з завданнями для розробників напряму з цього інструменту для звітування про запуски автоматизованого тестування.

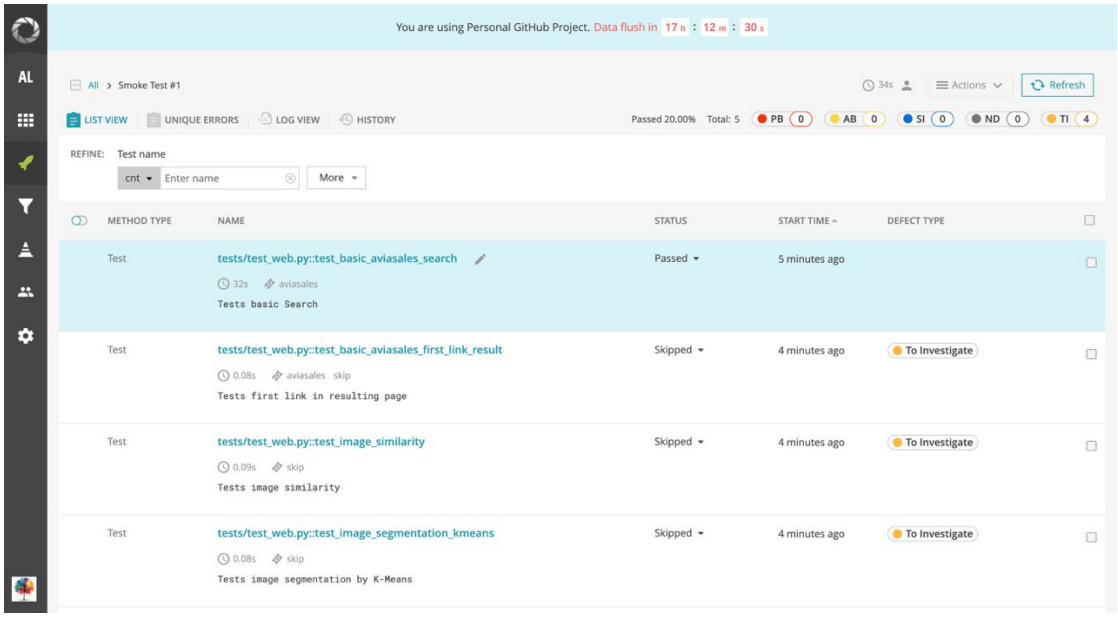


Рисунок 3.6 – Приклад звітів у Report Portal

3.5 Тестування розробленого фреймворку

Для тестування фреймворку візьмемо один користувацький «флоу», тобто сценарій. Користувач заходить на сайт пошуку авіаквитків, заповнює поля з містом вильоту, містом прильоту, датою вильоту. Обирає опцію без зворотного квитка та натискає на кнопку «Знайти квитки». Функціональна перевірка полягає

в тому, щоб переконатись, що у результаті відкриється сторінка з підходящими квитками. Для цього нам знадобиться ще один клас Page Object, що уособлюватиме сторінку з результатами пошуку (див. Додаток Б). Маючи локатор для окремого знайденого квитка, ми можемо перевірити, чи знайдено на сторінці хоч 1 такий елемент. Також можемо перевірити, чи в цьому елементі містами відправлення та призначення є ті самі міста, що ми ввели у пошуку. Цього всього ми досягнемо звичайними функціональними перевірками за допомогою реалізації Selenium Webdriver. Також, не вказуючи цього напряму, ми перевіримо факт існування необхідних елементів форми пошуку на головній сторінці, а також те, чи вони поводять себе як очікувано – поле вводу заповнюється, дата на календарі обирається, кнопка натискається. При бажанні, можемо навіть перевірити якийсь важливий текст, наприклад слоган сайту вгорі головної сторінки.

Для швидкої перевірки основного функціоналу цього достатньо, проте для візуального тестування – аж ніяк, бо ми не впевнились жодним чином, що протестовані елементи знаходяться в правильних позиціях, мають правильний колір тощо. Тому, ми маємо вказати в коді точки, де необхідно зробити скриншот для перевірки поточної стану системи та порівнянні її з тією, що є очікуваною. Ми обрали декілька моментів для знімків екрану, а саме: початок сценарію, коли всі поля є пустими; момент перед натисканням на кнопку пошуку, коли всі поля заповнені певними даними; момент, коли сторінка з результатами повністю провантажилась. При першому запуску нашого тесту на побудованому фреймворку, були зроблені 3 скриншоти. Після перевірки тестувальником і упевнення в тому, що вони є вірними, ці скриншоти стають базовими, або шаблонами. При подальших запусках тесту будуть робитись ті самі 3 знімки. І їх треба порівнювати з нашими шаблонами.

Почнемо із простого та прямолінійного варіанту подій – якщо є реальна відмінність між очікуваним і реальним знімком, наприклад, якщо «з’їхав» значок під першим полем або якщо зник значок пори дня згори (див. Рисунок 3.7).

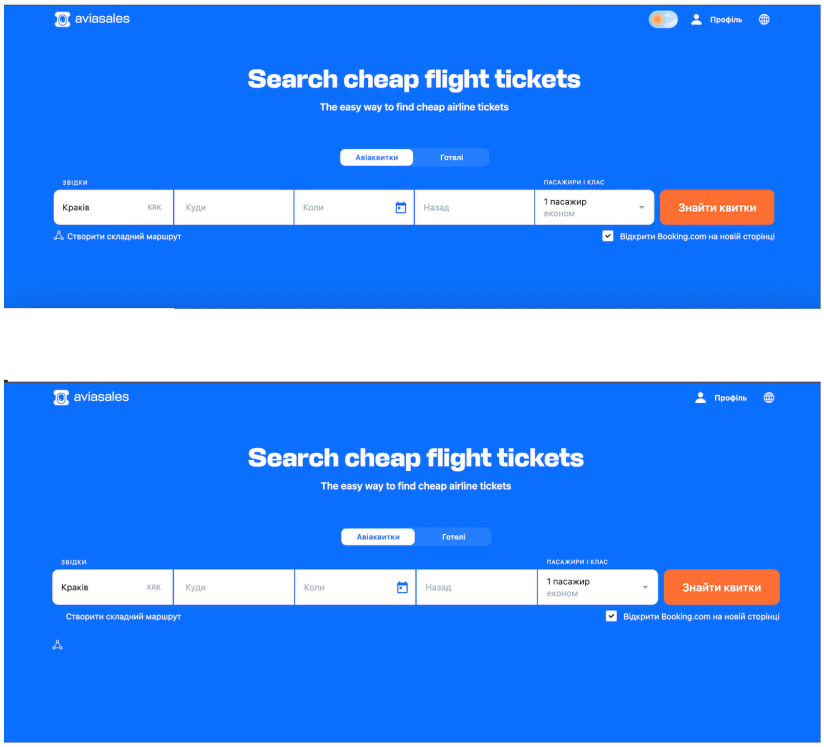


Рисунок 3.7 – Очікуваний (згори) та фактичний (знизу) знімки екрану

Результат запуску тесту буде виглядати таким чином у Report Portal (див. Рисунок 3.8).

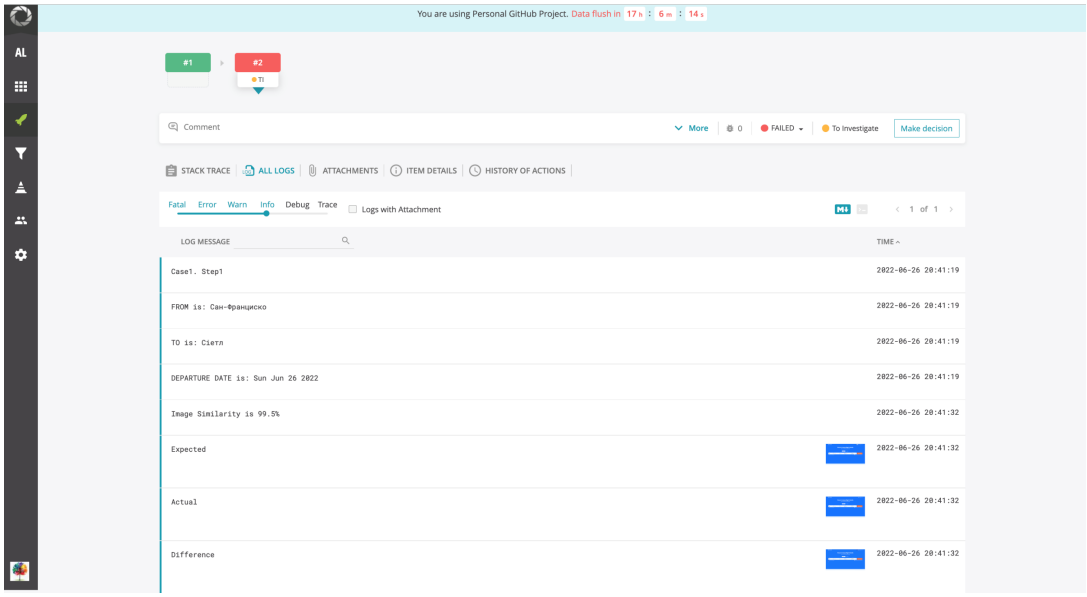


Рисунок 3.8 – Звіт по запуску тесту

Тут ми бачимо покроково, які дані вводились у форму, яке значення подібності знімків, та самі знімки – а саме три штуки: очікуваний результат, фактичний, та різниця між ними. Також видно, що тест видав негативний результат, отже значення подібності знімків не стовідсоткове. На Рисунку 3.9 показано третій знімок, що прикріплений до звіту: на ньому показано відмінності у зображеннях.

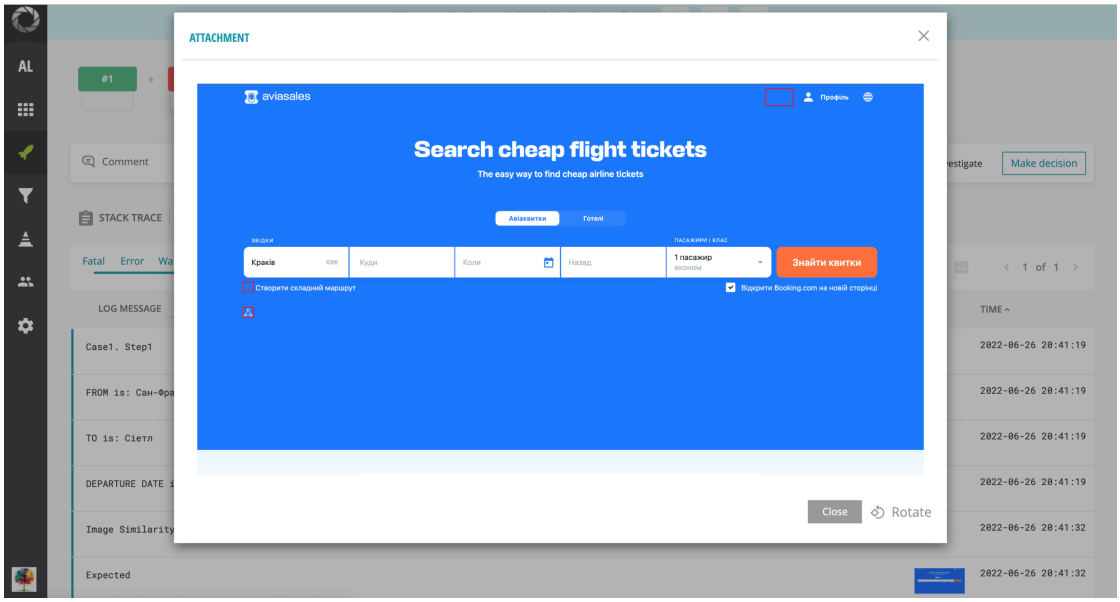


Рисунок 3.9 – Відмінності у скриншотах, що відображається у звіті

Як бачимо, алгоритм успішно знайшов та проілюстрував усі відмінності. Також, за потреби, як показано на Рисунку 3.10, прямо у Report Portal ми можемо позначити цей збій тесту як продуктову помилку (тобто реальну помилку, яку розробникам необхідно виправляти), чи як хибно негативну помилку, чи як очікувану помилку (наприклад якщо дизайн веб-застосунку був змінений цілеспрямовано і результат є очікуваним). При останньому варіанті необхідно буде замінити поточний шаблон на цей.

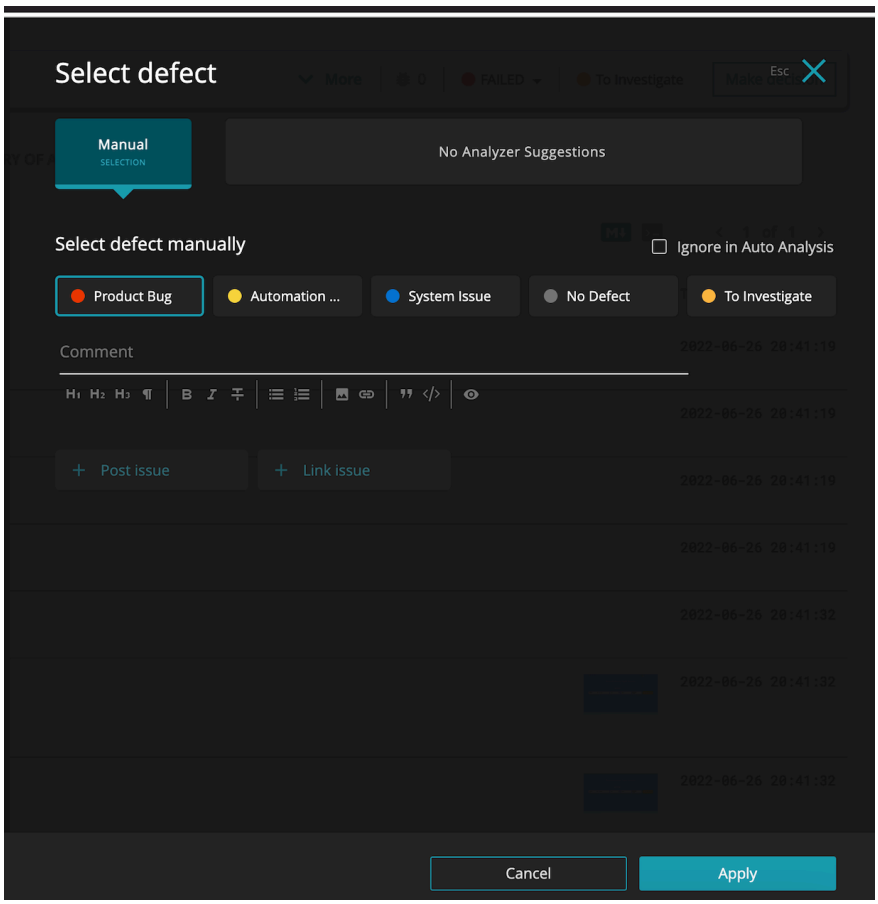


Рисунок 3.10 – Процес позначення збою як продуктової помилки

Отже, з прямолінійною задачею, фреймворк справляється добре. Наступною перевіркою буде, як він відреагує на згладжування шрифтів. Додамо відповідний CSS атрибут до тексту кнопки (див. Рисунок 3.11).

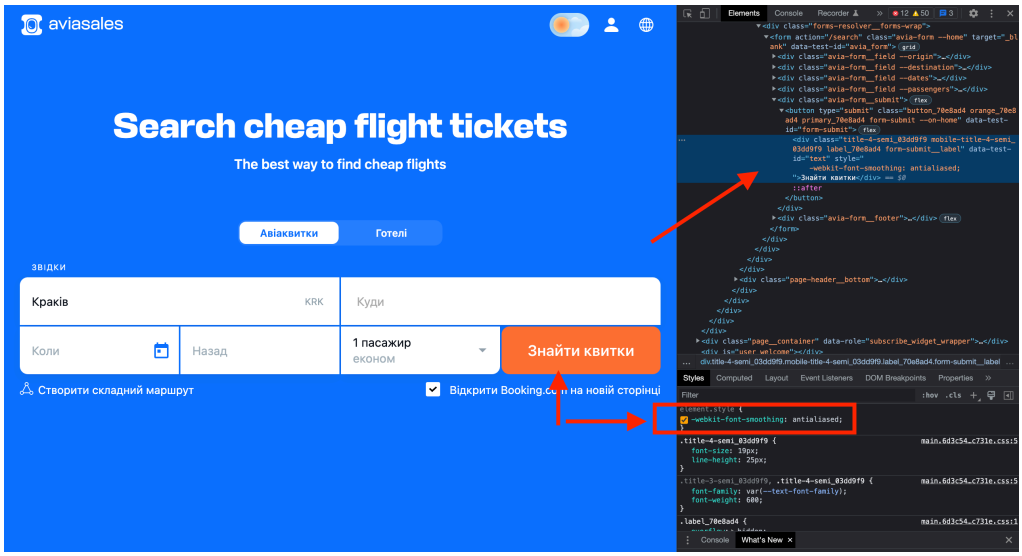


Рисунок 3.11 – Додавання CSS атрибуту згладжування шрифту до кнопки

Наочно відмінності у зображеннях будуть виглядати так (див. Рисунок 3.12). Для кінцевого користувача це зовсім непомітна різниця, тому не має братися до уваги в процесі порівняння, проте при прямолінійному попіксельному порівнянні фактична відмінність буде зафіксована і подібність скриншотів не буде стовідсотковим (див. Рисунок 3.13).

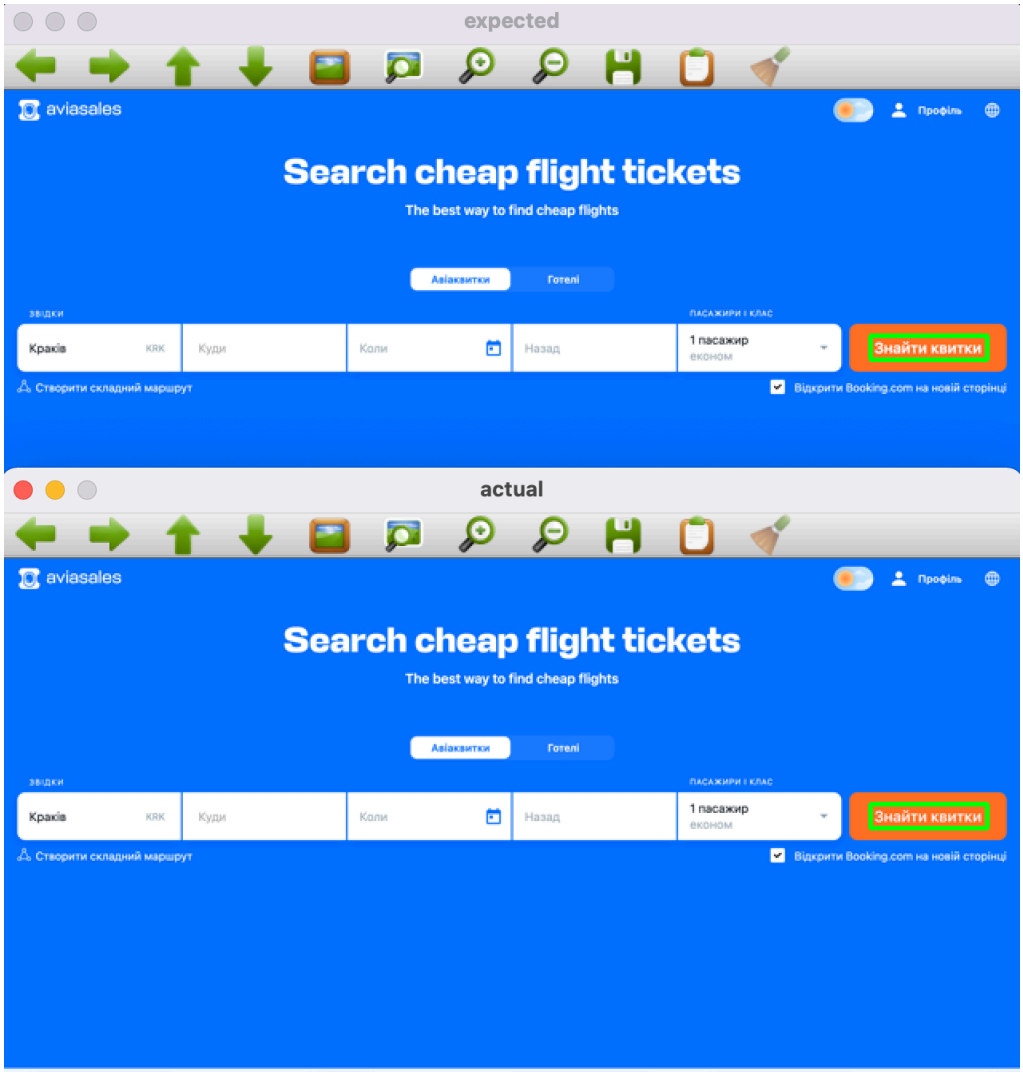


Рисунок 3.12 – Непомітна різниця між текстом при згладженні

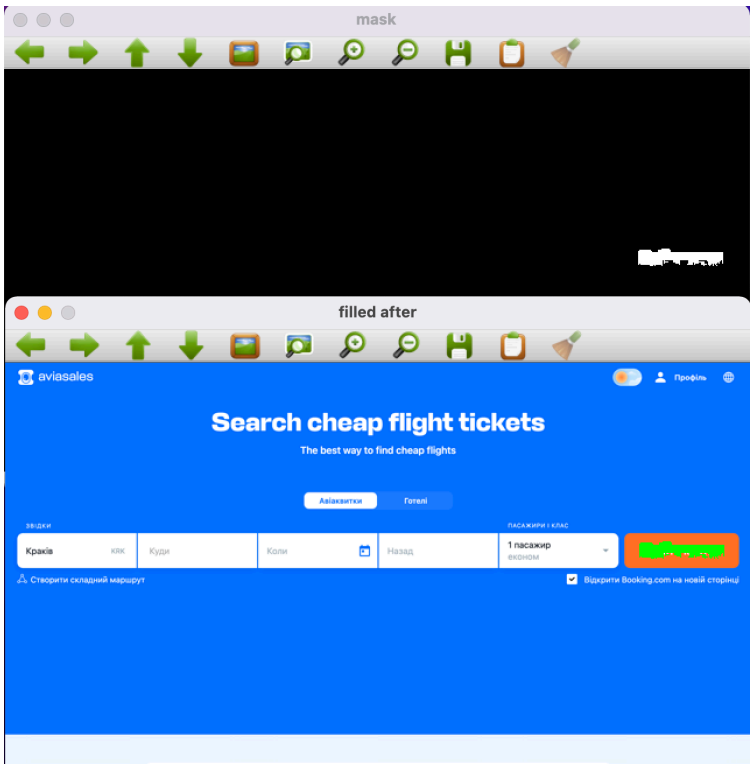


Рисунок 3.13 – Попіксельна різниця між текстом при згладженні

Прибирання всього тексту зі скриншота здатне вирішити цю проблему, при цьому не зменшивши цінність візуального тестування, адже важливий текст може бути легко перевірений функціонально, як вже зазначалось вище. Застосувавши алгоритми комп'ютерного зору, описані в розділі 2.2.3, ми отримаємо таке зображення (див. Рисунок 3.14).

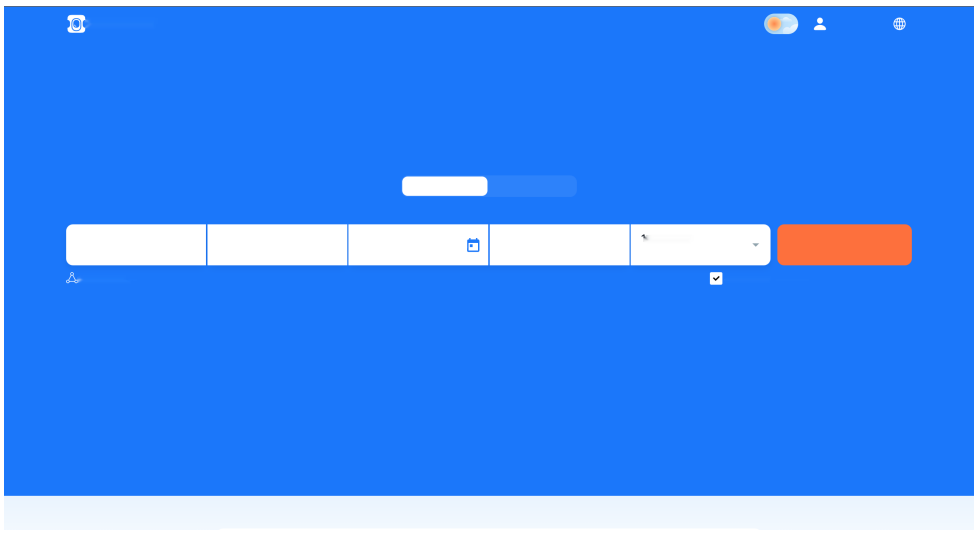


Рисунок 3.14 – Скриншот з видаленим текстом

Як бачимо вдалось повністю прибрати весь текст, і зафарбувати ці місця. Проте видно, що подекуди залишися сліди цього – трохи нечіткість фону, залишки тексту. Для того, щоб очистити це, а також для того, аби забезпечити однаковий схематичний вигляд будь-якої веб-сторінки на дисплеях з різною роздільною здатністю, використовуватимемо кластеризацію зображення за допомогою якогось методу сегментації. Ми можемо звернутись до MeanShift або ж K-Means алгоритму. В першому випадку ми б програли по швидкості роботи, а в другому – по необхідності вручну задавати параметр k (кількість кластерів). В цій роботі я продемонструю результат роботи саме методу K-Means з 4 кластерами. На Рисунку 3.15 видно, що ми маємо чітко виділені візуальні елементи на сторінці, відсутність тексту на скриншоті та будь-яких залишків зафарбовування від попереднього алгоритму. Знімок готовий до порівнянь з базовим зображенням. У Додатках В та Г можна побачити застосування такого ж підходу в попередній обробці зображення зі сторінкою результатів пошуку.

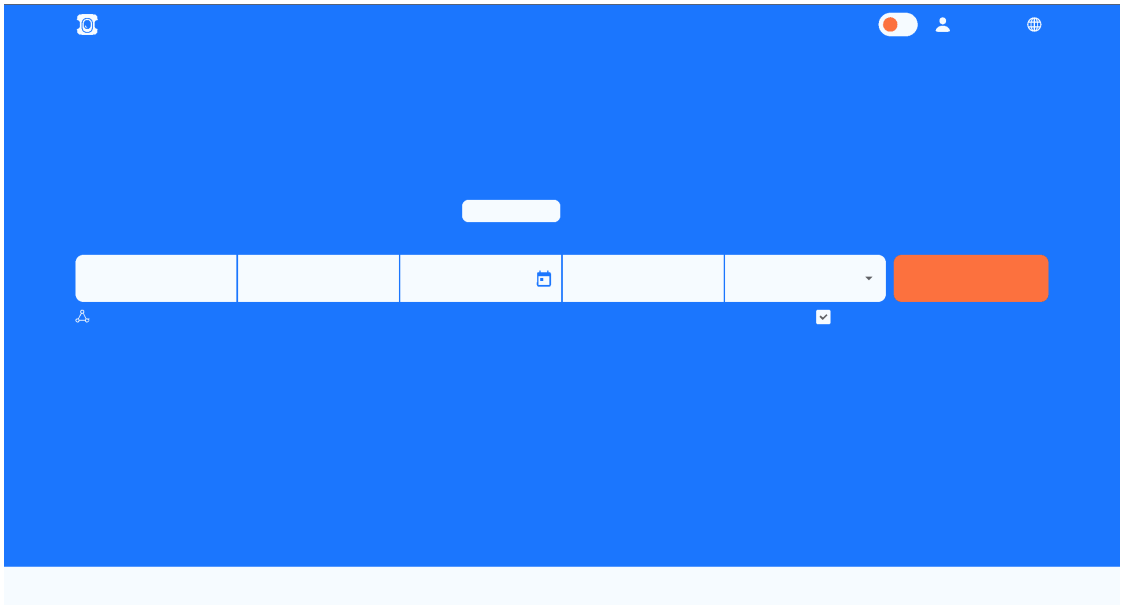


Рисунок 3.15 – Зображення після застосування алгоритму K-Means

Отже, припустимо на головній сторінці веб-сайту є певні візуальні збої. По-перше, абсолютно об’єктивна зміна в графічному інтерфейсі відбулась в розміщенні іконки для перемикання мови – вона опустилась на декілька пікселів

донизу. По друге, був змінений розмір першого поля для вводу місця вильоту і кнопка пошуку. По-третє, текст першого поля відрізняється від шаблонного, адже місто вильоту заповняється автоматично залежно від геолокації користувача, тобто це динамічна змінна, яка не має впливати на результати тесту. І зрештою остання відмінність полягає в щільності пікселів і відтворенні шрифтів через те, що базовий знімок був зроблений на машині з операційною системою Mac на дисплеї Apple Retina Display, а тестування нашим фреймворком було запущено на Windows машині на екрані з іншою роздільною здатністю. На Додатку Д і Е можна побачити початковий вигляд базового (шаблонного) зображення і поточного знімка. За допомогою розробленого фреймворку протестуємо той же самий користувацький сценарій з введенням у форму пошуку певні дані. Результат візуального тестування виглядатиме таким чином (див. Рисунок 3.16).

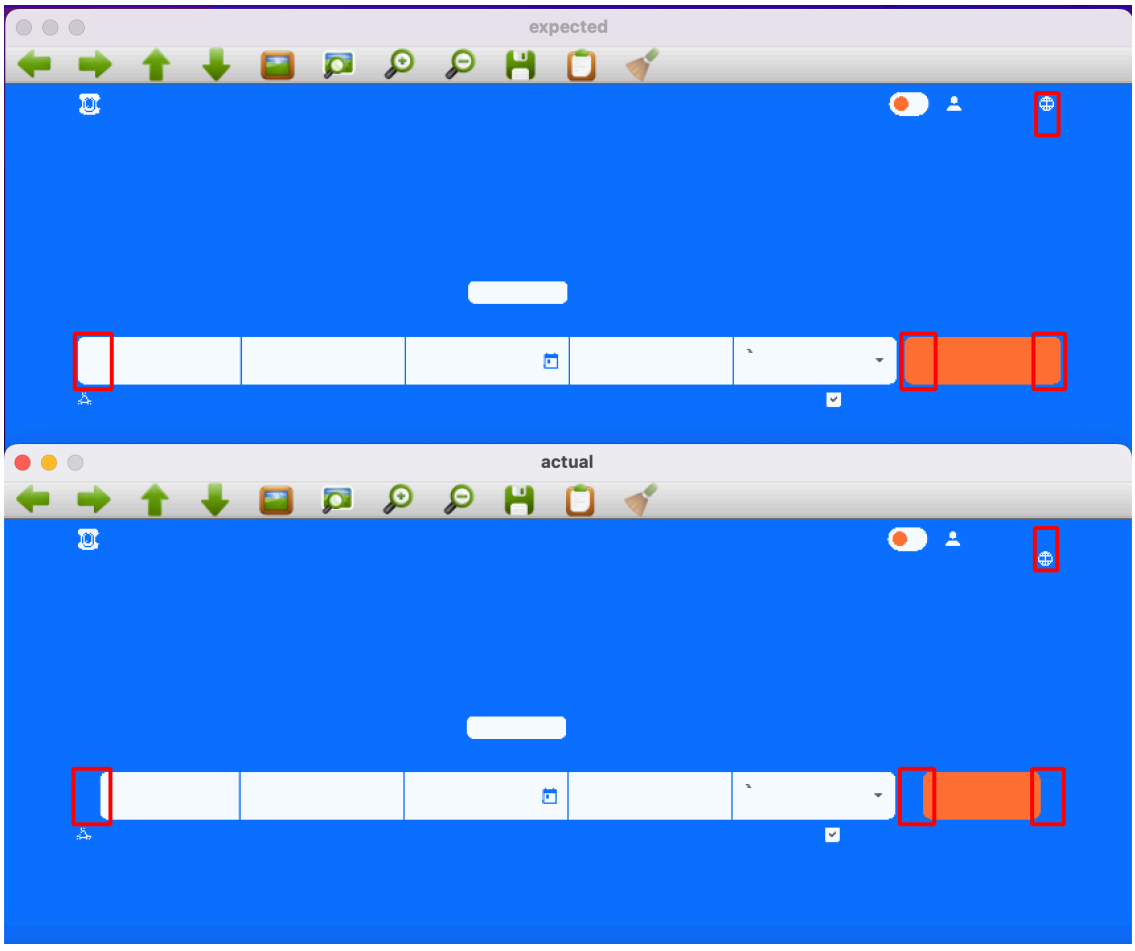


Рисунок 3.16 – Візуальні відмінності, зафіксовані фреймворком

Зрештою, фреймворк абсолютно правильно визначив важливі візуальні відмінності у зображеннях, які несуть цінність для тестування веб-сайту, знехтувавши текстом та роздільною здатністю. Значно зменшилась кількість хибно негативних візуальних збоїв, а отже збільшилась ефективність тестування.

Висновки

Тестування графічного інтерфейсу є дуже важливим етапом тестування для контролю якості програмного забезпечення. Незважаючи на те, що розроблено багато засобів і методів автоматизованого тестування, вони все ще не вирішують всіх проблем, наприклад, відстеження візуальних змін в інтерфейсі. У зв’язку зі стрімким розвитком автоматизованого візуального тестування виникла потреба в розробці системи, яка дозволить виконати більш точний аналіз інтерфейсу та зменшити кількість помилкових спрацьовувань.

Перша частина роботи присвячена огляду теоретичних основ мануального та автоматизованого візуального тестування – було розглянено існуючі методи та програмне забезпечення.

У другому розділі був проведений детальний аналіз основних методів і підходів до автоматизованого візуального тестування, в тому числі методів перевірки стилю елементів і порівняння піксельних зображень. Було проаналізовано методи кластеризації (K-Means і Mean Shift) для сегментації зображень, вимірювання подібності, знаходження і видалення тексту зі знімків. При виборі методів обробки зображень було проведено детальний огляд та порівняльний аналіз.

Створення фрейморку (програмного рішення) автоматизованого візуального тестування, а саме основний результат роботи, описаний в третій частині. Для розв'язання цієї задачі були виконані такі завдання: розглянуто особливості та етапи розробки фреймворків для тестування, підібрано програмне забезпечення для розробки відповідної системи, застосовані методи комп’ютерного зору, визначено головні процеси, розроблено систему візуального тестування та перевірено її працездатність. При розробці автоматизованої системи візуального тестування веб-інтерфейсів були використані мови програмування Python, бібліотеки Keras та OpenCV, платформа тестування Pytest.

Список використаної літератури

1. Bertalmio M. Navier-stokes, fluid dynamics, and image and video inpainting [Електронний ресурс] / M. Bertalmio, A. L. Bertozzi, G. Sapiro // 2001 IEEE computer society conference on computer vision and pattern recognition. CVPR 2001, Kauai, HI, USA. – [Б. м.]. – Режим доступу: https://doi.org/10.1109/cvpr.2001.990497 (дата звернення: 01.07.2022). – Назва з екрана.
2. Fewster M. Test automation engineering: guide to the ISTQB advanced level certification / Mark Fewster, Ina Schieferdecker, Andrew L. Pollner. – [Б. м.] : Rocky Nook, 2018. – 300 с.
3. Ghahrai A. Test automation advantages and disadvantages [Електронний ресурс] / Amir Ghahrai // DevQA.io - For Developers and QAs. – Режим доступу: https://devqa.io/test-automation-advantages-and-disadvantages/ (дата звернення: 27.06.2022). – Назва з екрана.
4. Giroto L. Visual regression testing [Електронний ресурс] / Leonardo Giroto // Medium. – Режим доступу: https://medium.com/loftbr/visual-regression-testing-eb74050f3366 (дата звернення: 27.06.2022). – Назва з екрана.
5. Huiyu Z. Mean shift and its application in image segmentation [Електронний ресурс] / Zhou Huiyu, Wang Xun, Schaefer Gerald // SpringerLink. – Режим доступу: https://doi.org/10.1007/978-3-642-17934-1_13 (дата звернення: 27.06.2022). – Назва з екрана.
6. Image quality assessment: from error visibility to structural similarity [Електронний ресурс] / Z. Wang [та ін.] // IEEE transactions on image processing. – 2004. – Т. 13, № 4. – С. 600–612. – Режим доступу: https://doi.org/10.1109/tip.2003.819861 (дата звернення: 27.06.2022). – Назва з екрана.

7. Introduction to Image Segmentation with K-Means clustering - KDnuggets [Електронний ресурс] // KDnuggets. – Режим доступу: https://www.kdnuggets.com/2019/08/introduction-image-segmentation-k-means-clustering.html (дата звернення: 27.06.2022). – Назва з екрана.
8. Image segmentation: deep learning vs traditional [guide] [Електронний ресурс] // V7 - AI Data Platform for Computer Vision Annotation. – Режим доступу: https://www.v7labs.com/blog/image-segmentation-guide (дата звернення: 01.07.2022). – Назва з екрана.
9. Software testing: an ISTQB-BCS certified tester foundation guide / Peter Morgan [та ін.]. – [Б. м.] : BCS Learning & Development Limited, 2019. – 283 с.
10. Test automation frameworks [Електронний ресурс] // smartbear.com. – Режим доступу: https://smartbear.com/learn/automated-testing/test-automation-frameworks/ (дата звернення: 27.06.2022). – Назва з екрана.
11. Telea A. An image inpainting technique based on the fast marching method [Електронний ресурс] / Alexandru Telea // Journal of graphics tools. – 2004. – Т. 9, № 1. – С. 23–34. – Режим доступу: https://doi.org/10.1080/10867651.2004.10487596 (дата звернення: 01.07.2022). – Назва з екрана.
12. Top 8 python testing frameworks in 2021 - testproject [Електронний ресурс] // TestProject. – Режим доступу: https://blog.testproject.io/2020/10/27/top-python-testing-frameworks/ (дата звернення: 27.06.2022). – Назва з екрана.
13. Types of test automation frameworks | everything you should know [Електронний ресурс] // Software Testing Material. – Режим доступу: https://www.softwaretestingmaterial.com/types-test-automation-frameworks/#What-is-a-framework (дата звернення: 27.06.2022). – Назва з екрана.
14. What is selenium webdriver? [complete guide] [Електронний ресурс] // Hackr.io. – Режим доступу: https://hackr.io/blog/what-is-selenium-webdriver (дата звернення: 30.06.2022). – Назва з екрана.

15. What is visual testing? [Електронний ресурс] // Automated Visual Testing | Applitools. – Режим доступу: https://applitools.com/blog/visual-testing/ (дата звернення: 27.06.2022). – Назва з екрана.
16. Zhou H. Mean shift and its application in image segmentation [Електронний ресурс] / Huiyu Zhou, Xun Wang, Gerald Schaefer // Innovations in intelligent image analysis. – Berlin, Heidelberg, 2011. – С. 291–312. – Режим доступу: https://doi.org/10.1007/978-3-642-17934-1_13 (дата звернення: 27.06.2022). – Назва з екрана.

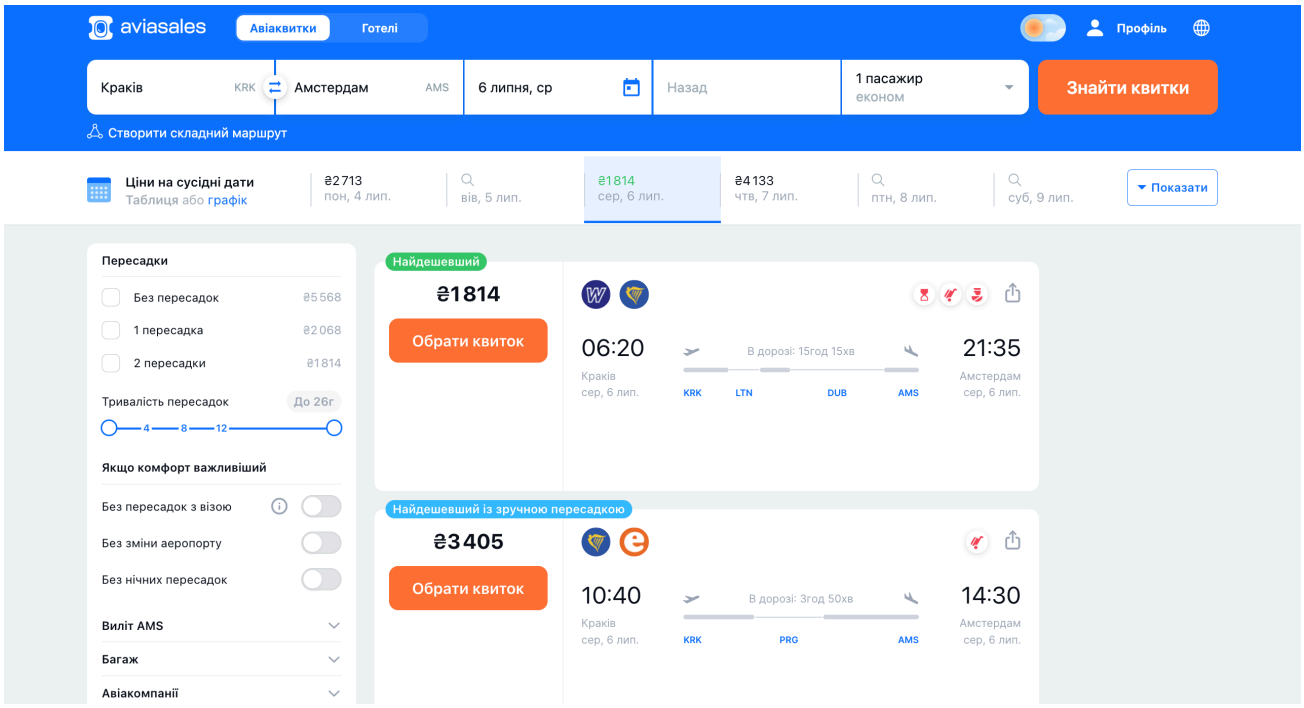
Додаток А (обов'язковий) Код класу об'єкта головної сторінки тестованого веб-сайту

```
class AviaSalesSearchPage:
  URL = 'https://www.aviasales.ua/'
  SEARCH_INPUT_FROM = (By.CSS_SELECTOR, '[data-test-id="autocomplete-origin"] input')
  SEARCH_INPUT_TO = (By.CSS_SELECTOR, '[data-test-id="autocomplete-destination"] in-
put')
  SEARCH_BUTTON = (By.CSS_SELECTOR, 'button[data-test-id="form-submit"]')
  NO_RETURN_BUTTON = (By.CSS_SELECTOR, 'button[data-test-id="no-return-ticket"]')

  @classmethod
  def DATE_CELL_CALENDAR(cls, date):
    selector = f'.calendar__week [aria-label="{date}"]'
    return (By.CSS_SELECTOR, selector)

  def __init__(self, browser):
    self.browser = browser
  def load(self):
    self.browser.get(self.URL)

  def search_no_return(self, from_city, to, depart_date):
    search_input_to = self.browser.find_element(*self.SEARCH_INPUT_TO)
    search_input_to.send_keys(to + Keys.RETURN)
    time.sleep(2)
    departure_input = self.browser.find_element(*self.DEPARTURE_INPUT)
    departure_input.click()
    time.sleep(2)
    self.select_date(self.DEPARTURE_INPUT, depart_date)
    time.sleep(1)
    self.select_no_return()
    time.sleep(1)

  def select_no_return(self):
    no_return = self.browser.find_element(*self.NO_RETURN_BUTTON)
    no_return.click()

  def select_date(self, selector, date):
    date_cell = self.browser.find_element(*self.DATE_CELL_CALENDAR(date))
    date_cell.click()

  def click_search(self):
    button = self.browser.find_element(*self.SEARCH_BUTTON)
    button.click()
```

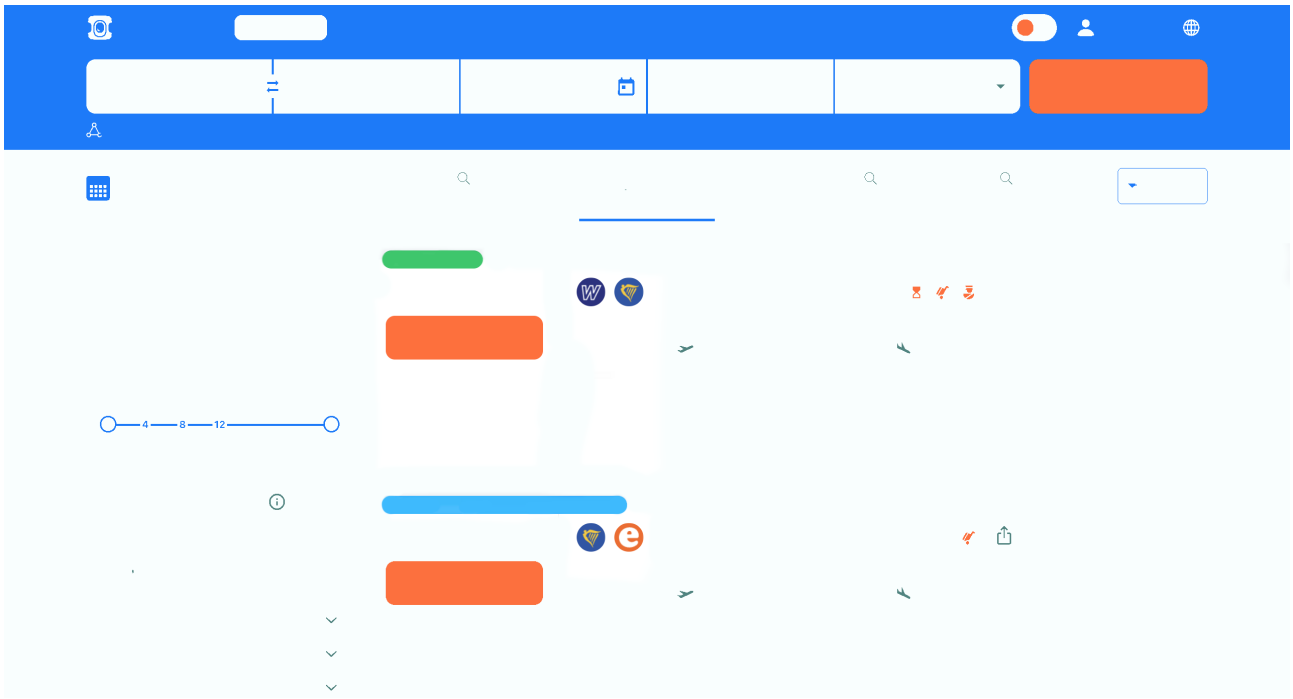
Додаток Б (обов'язковий) Код класу об'єкта сторінки результатів пошуку

```
class AviasalesResultPage:
  TICKET_ITEM = (By.CSS_SELECTOR, '.product-list__item[data-ticket-sign]')
  FILTERS_SIDE_BAR = (By.CSS_SELECTOR, '.app__sidebar')
  TOP_PANEL_PREDICTION = (By.CSS_SELECTOR, '.prediction')
  BUTTON_SHOW_IN_TOP_PANEL = (By.CSS_SELECTOR, '.prediction-header__right-col
button')

  def __init__(self, browser):
    self.browser = browser
    browser.switch_to.window(browser.window_handles[1])

  def tickets_found_count(self):
    link_divs = self.browser.find_elements(*self.TICKET_ITEM)
    return len(link_divs)

  def click_show_more_prediction(self):
    more_btn = self.browser.find_element(*self.BUTTON_SHOW_IN_TOP_PANEL)
    more_btn.click()
```

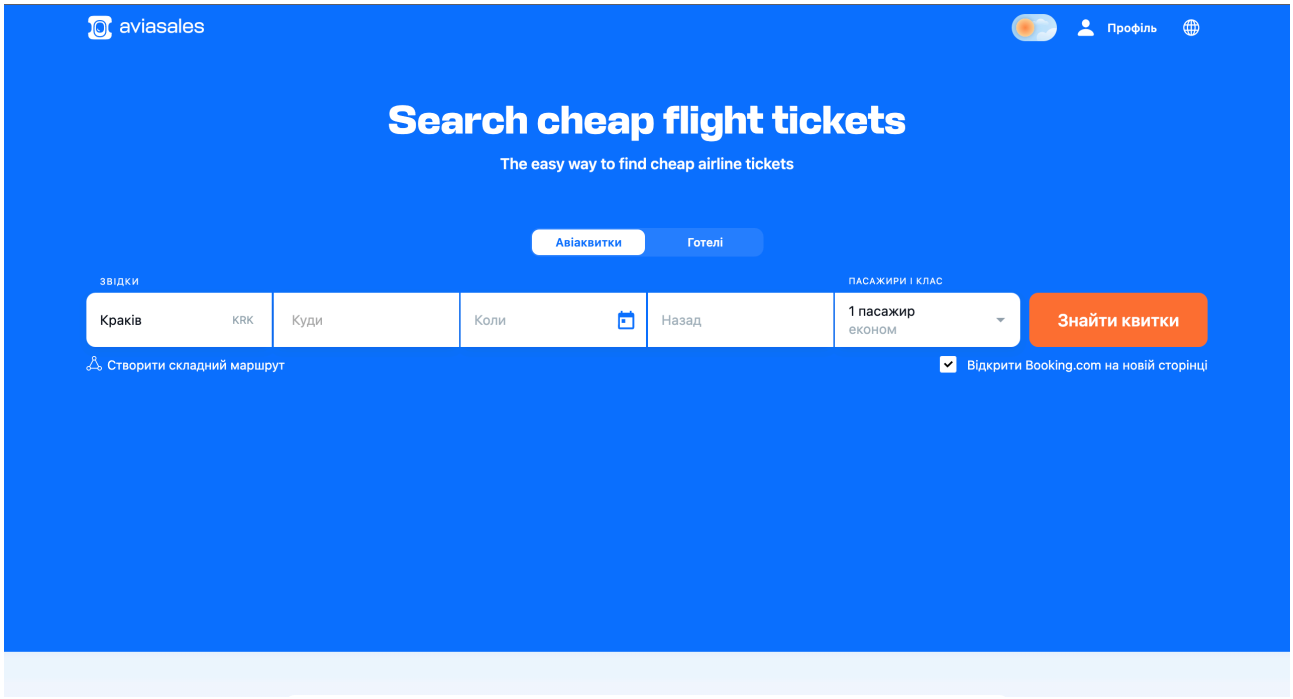
Додаток В (обов’язковий) Скриншот сторінки результатів пошуку до обробки за допомогою комп’ютерного зору



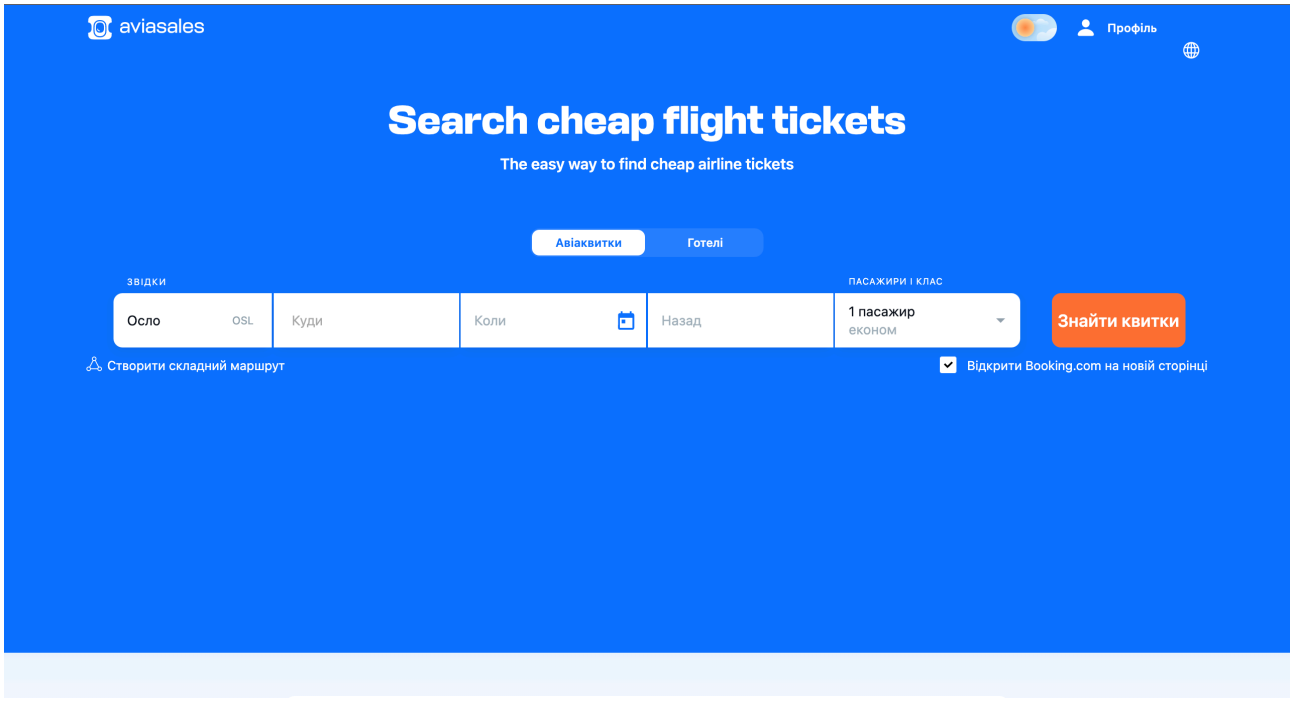
Додаток Г (обов’язковий) Скриншот сторінки результатів пошуку після обробки за допомогою комп’ютерного зору



Додаток Д (обов’язковий) Початковий базовий (шаблонний) знімок екрану



Додаток Е (обов'язковий) Знімок екрану з певними візуальними змінам



Додаток Ж (обов'язковий) Код алгоритму знаходження індексу структурної подібності двох зображень

```
from skimage.metrics import structural_similarity
import cv2
import numpy as np

def scale_images(image1, image2, width=640, height=480):
    ## Setting constant dimensions, with arbitrary default values of 640x480
    dim = (width, height)

    image1 = cv2.resize(image1, dim)
    image2 = cv2.resize(image2, dim)

    return image1, image2



expected = cv2.imread('test_images/diploma1_notext_kmeans4.png')
actual = cv2.imread('test_images/diploma12_notext_kmeans4_.png')
expected, actual = scale_images(expected, actual, 853, 460)

# Converting to grayscale
expected_gray = cv2.cvtColor(expected, cv2.COLOR_BGR2GRAY)
actual_gray = cv2.cvtColor(actual, cv2.COLOR_BGR2GRAY)

# Compute SSIM
(score, difference) = structural_similarity(expected_gray, actual_gray, full=True)
print("Image Similarity: {:.4f}%".format(score * 100))

# Converting the array to 8-bit unsigned integers in range [0,255]
difference = (difference * 255).astype("uint8")
difference_box = cv2.merge([difference, difference, difference])

# Threshold the difference image
thresh = cv2.threshold(diff, 0, 255, cv2.THRESH_BINARY_INV | cv2.THRESH_OTSU)[1]
```

```
# Find contours for obtaining regions that are different
contours = cv2.findContours(thresh, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
contours = contours[0] if len(contours) == 2 else contours[1]

mask = np.zeros(expected.shape, dtype='uint8')
filled_after = actual.copy()
for c in contours:
    area = cv2.contourArea(c)
    if area > 40:
        x,y,w,h = cv2.boundingRect(c)
        cv2.rectangle(expected, (x, y), (x + w, y + h), (0,0,255), 2)
        cv2.rectangle(actual, (x, y), (x + w, y + h), (0,0,255), 2)
        cv2.rectangle(difference_box, (x, y), (x + w, y + h), (0,0,255), 2)
        cv2.drawContours(mask, [c], 0, (255,255,255), -1)
        cv2.drawContours(filled_after, [c], 0, (0,0,255), -1)
```

Додаток К (обов'язковий) Код реалізації алгоритму видалення тексту з зображення

```
import matplotlib.pyplot as plt
import keras_ocr
import cv2

def midpoint(x1, y1, x2, y2):
    x_mid = int((x1 + x2) / 2)
    y_mid = int((y1 + y2) / 2)
    return (x_mid, y_mid)

def inpaint_text(img_path, pipeline):
    # read image
    img = keras_ocr.tools.read(img_path)
    # generate (word, box) tuples
    prediction_groups = pipeline.recognize([img])
    mask = np.zeros(img.shape[:2], dtype="uint8")
    for box in prediction_groups[0]:
        x0, y0 = box[1][0]
        x1, y1 = box[1][1]
        x2, y2 = box[1][2]
        x3, y3 = box[1][3]

        x_mid0, y_mid0 = midpoint(x1, y1, x2, y2)
        x_mid1, y_mi1 = midpoint(x0, y0, x3, y3)
        thickness = int(math.sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2))
        cv2.line(mask, (x_mid0, y_mid0), (x_mid1, y_mi1), 255,
                 thickness)
        img = cv2.inpaint(img, mask, 7, cv2.INPAINT_NS)
    return (img)

pipeline = keras_ocr.pipeline.Pipeline()
img = inpaint_text(root_path + name_img_to_process + '.png', pipeline)
img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
```

Додаток Л (обов'язковий) Код реалізації K-Means та MeanShift алгоритмів сегментації

```
def mean_shift(img_path):
    img = cv2.imread(img_path)
    # reduce noise
    img = cv2.medianBlur(img, 3)
    # flattening
    flat_image = np.float32(img.reshape((-1, 3)))
    # calculating gaussian kernel
    bandwidth = estimate_bandwidth(flat_image, quantile=.04)
    ms = MeanShift(bandwidth=bandwidth, max_iter=100, bin_seeding=True, cluster_all=True)
    ms.fit(flat_image)
    labeled = ms.labels_
    # getting number of segments
    segments = np.unique(labeled)
    print('Number of segments: ', segments.shape[0])
    # getting the average color of each segment
    total = np.zeros((segments.shape[0], 3), dtype=float)
    count = np.zeros(total.shape, dtype=float)
    for i, lab in enumerate(labeled):
        total[lab] = total[lab] + flat_image[i]
        count[lab] += 1
    avg = total / count
    avg = np.uint8(avg)
    # casting the labeled image into matcing average color
    res = avg[labeled]
    result = res.reshape((img.shape))
    return result

def kmeans(img_path, k):
    image = cv2.imread(img_path)
    # reshaping the image to a 2D array of pixels and RGB values, then converting to float
    pixel_values = np.float32(image.reshape((-1, 3)))
    # criteria when to stop
```

```
    criteria = (cv2.TERM_CRITERIA_EPS, 10, 0.1)
    # number of clusters (K)
    _, labels_e, (centers_e) = cv2.kmeans(pixel_values, k, None, criteria, 10,
cv2.KMEANS_RANDOM_CENTERS)
    # converting back to 8 bit values
    centers_e = np.uint8(centers_e)
    # flattening the labels array
    labels_e = labels_e.flatten()
    # converting all pixels to the color of the centroids
    segmented_image_e = centers_e[labels_e]
    # returning original img dimension
    segmented_image_e = segmented_image_e.reshape(image.shape)
    return segmented_image_e
```