math_2022_042

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЧЕРНІВЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ЮРІЯ ФЕДЬКОВИЧА

Факультет математики та інформатики
кафедра математичного моделювання

ОРГАНІЗАЦІЯ ТЕСТУВАННЯ ПРОЕКТУ «РЕАБІЛІТАЦІЙНИЙ ЦЕНТР «ОСОБЛИВА ДИТИНА»»

Кваліфікаційна робота

Рівень вищої освіти – другий (магістерський)

Виконала:
студентка 6 курсу, 607 групи
Петрусяк Марія Павлівна
Керівник*:*
доцент Піддубна Л.А.

До захисту допущено
на засіданні кафедри
протокол № 8 від 6 грудня 2022 р.
Зав. кафедрою ____________ проф. Черевко І.М.

Чернівці – 2022

АНОТАЦІЯ

У магістерській роботі розроблений тестовий фреймворк із автоматизованими тестами для сайту взаємодії між фахівцями Буковинського реабілітаційного центру «Особлива дитина» та батьками, у яких є діти з інвалідністю віком до 18 років. Проект створений для допомоги клієнтам і для зменшення інформаційного та організаційного навантаження фахівців.

The master's thesis developed a test framework with automated tests for interaction between specialists of the Bukovyna Rehabilitation Center "Special Child" and parents who have children with disabilities under the age of 18. The project was created to help clients and to reduce the information and organizational load of specialists.

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів наукових досліджень інших авторів мають посилання на відповідне джерело.

____

(підпис)

ЗМІСТ

ВСТУП……………………………………………………………..4

§1. Теоретичний опис проробленої роботи…………..……… .6

1.1 Написання чек-листів на їхня роль в тестуванні ..……..6

1.2 Мануальне тестування проекту та знайдені баги ..……..8

1.3 BDD підхід для написання автоматизованих тестів …..12

1.4 E2E тестування в автоматизації ……………………....14

§2. Реалізація фреймворку ……………………………………..17

2.1 Опис архітектури проекту, використовуючи POM патерн проектування ……………………………………………17

2.2 Налаштування проекту (Gemfile) ……………………….23

2.3 Cucumber та Capybara. Особливості бібліотек ………...28

2.4 Інструкція для користування. Запуск тестів …………..33

Висновки ……………………………………………………….37

Список використаної літератури ………………………………..38

Додатки ………………………………………………………...40

ВСТУП

Тестування або забезпечення контролю якості – це постійний і впорядкований процес, який допомагає інженерам-тестувальникам дізнатися, чи відповідає програмне забезпечення конкретним і точним вимогам.

Для реалізації даної роботи вирішили тестувати проект для взаємодії між фахівцями Буковинського реабілітаційного центру «Особлива дитина» та батьками, у яких є діти з інвалідністю віком до 18 років.

Загалом, Буковинський центр комплексної реабілітації для дітей з інвалідністю «Особлива дитина» – реабілітаційна установа, цільовим призначенням якої є здійснення комплексу реабілітаційних заходів, спрямованих на створення умов для всебічного розвитку дітей з інвалідністю.

Робота центрів реабілітації особливих людей/дітей не проста, потребує спеціальної професійної підготовки, сконцентрованої віддачі в роботі. Щоб якось полегшити виконання такої важливої справи, варто використовувати комп`ютерну техніку для забезпечення виконання типових, рутинних операцій. Тому важливо, аби наш проект працював коректно і без помилок.

Завдання роботи:

організація процесу мануального тестування та автоматизації тестування програмного забезпечення.

Мета роботи:

забезпечити якість програмного продукту і тим самим полегшити безперешкодне виконання типових, рутинних операцій.

При створенні тестового фреймворку були використані наступні технології:

- мова програмування Ruby;
- бібліотеки Cucumber та Capybara;
- бібліотека SitePrism для реалізації POM патерна;
- Selenium Web Driver – інструмент для імітації дій користувача в програмі.

Для організації роботи були використані наступні програми для розробки:

- інтегроване середовище розробки Ruby Mine;
- Chrome Driver – середовище для тестування;
- Trello – для організації та керування задачами;
- GitHub – для контролю версій та зберігання проекту.

§1. Теоретичний опис проробленої роботи

1.1 Написання чек-листів на їхня роль в тестуванні

Оскільки дану програму потрібно було тестувати з нуля, то для більшої відсотковості покриття протестованого функціоналу було вирішено писати чек-листи.

Чек-лист є основним елементом тестування програмного забезпечення. Він містить ряд тестів, перевірок, які допомагають визначити, чи готовий продукт до розгортання для кінцевих користувачів. А якщо ні, це допомагає дізнатися, які компоненти потрібно переробити та покращити. Без контрольного списку перевірок неможливо бути повністю впевненим у якості продукту.

І ось чому:

Ви можете провести вічність, тестуючи програму, і все одно не зможете сказати, чи готова вона до випуску, чи ні зі 100% упевненістю. Щоб цього не сталося, вам потрібно мати фіксований набір тестів, які повністю охоплюють функціональність майбутнього програмного забезпечення.

Ви не зможете зробити висновок про готовність програмного забезпечення до розгортання. Лише за допомогою контрольного списку перевірок (чек-листів) ви зможете побачити відсоток функцій додатка, які працюють належним чином.

Не забуваймо про «людський фактор». Майже неможливо все запам'ятати і назвати ті компоненти продукту, які вже перевірені, а які ще ні.

Ви не можете надати точну оцінку часу для тестування без фіксованого набору тестів.

Будь-який контрольний список створюється на основі Специфікації вимог до програмного забезпечення (SRS документації).

При створенні чек-листів слід дотримуватися трьох основних правил:

- Включені тести мають охоплювати всі функції продукту. Жодна вимога не повинна залишатися без уваги.

- Необхідно звести до мінімуму кількість тестів. Чим більше вимог ви можете перевірити за допомогою одного тесту, тим краще.
- Набір тестів повинен не повторювати вимоги, а перевіряти їх.

Отже, Чек-листи влаштовані дуже просто. Будь-який містить перелік блоків, секцій, сторінок, інших елементів, які слід протестувати.

Переваги використання чек-листів:

- покращити уявлення про систему загалом, бачити статус її готовності;
- розуміти обсяг виконаної та майбутньої роботи з тестування;
- не повторюватися в перевірках і не пропустити нічого важливого в процесі тестування.

1.2 Мануальне тестування проекту та знайдені баги

Як вже було зазначено раніше, проект «Особлива дитина» було спочатку мануально протестовано, тобто вручну. Це був підготовчий етап для написання автоматизованих тестів, адже спочатку потрібно було зрозуміти, як працює система в цілому, на що краще звернути увагу при написання авто тестів.

Тому для ручного тестування було обрано підхід тестування чорного ящика (black-box testing) та типи даного підходу, такі як: димне тестування (smoke testing), функціональне тестування та приймальне тестування.

Тестування чорної скриньки передбачає перевірку системи без попереднього знання її внутрішньої роботи. Тестувальник надає вхідні дані та спостерігає за результатами, згенерованими системою, що тестується. Це дає змогу визначити, як система реагує на очікувані та несподівані дії користувача, час відповіді, проблеми зручності та надійності.

Тестування чорної скриньки є потужною технікою тестування, оскільки вона наскрізно перевіряє систему. Подібно до того, як кінцевим користувачам «байдуже», як закодована чи архітектурна система, і вони очікують отримати відповідну відповідь на їхні запити, тестувальник може моделювати діяльність користувача та перевіряти, чи виконує система свої обіцянки. В той же час тест чорної скриньки оцінює всі відповідні підсистеми, включаючи UI/UX дизайн, веб-сервер або сервер додатків, базу даних, залежності та інтегровані системи.

Димне тестування — це процес тестування програмного забезпечення, який визначає, чи є розгорнуте програмне забезпечення стабільне чи ні. Димне тестування є підтвердженням для тестувальника продовжити подальше тестування програмного забезпечення.

Воно складається з мінімального набору тестів, які виконуються на кожній збірці для перевірки функцій програмного забезпечення. Димне тестування також відоме як «Тестування верифікації» або «Тестування достовірності».

Переваги димного тестування:

Усі проблеми в системі будуть ідентифіковані за допомогою димного тестування.

Димне тестування проводиться після випуску збірки для контролю якості. За допомогою димового тестування більшість дефектів виявляються на початкових етапах розробки програмного забезпечення.

За допомогою димного тестування ми спрощуємо виявлення та виправлення основних дефектів.

За допомогою димного тестування команда контролю якості може виявити дефекти у функціональності програми, які могли виникнути через новий код.

Тестування «чорної скриньки» може перевірити певні функції або особливості програмного забезпечення, яке тестується. Наприклад, перевірити, чи можна ввійти в систему за допомогою правильних облікових даних користувача та неможливо ввійти за допомогою неправильних облікових даних.

Функціональне тестування може бути зосереджено на найбільш критичних аспектах програмного забезпечення, на інтеграції між ключовими компонентами (тип - тестування інтеграції) або на системі в цілому (системне тестування).

Регресійне тестування – це тип тестування програмного забезпечення для підтвердження того, що нещодавня зміна програми чи коду не вплинула негативно на наявну функціональність. Регресійне тестування — це не що інше, як повний або частковий вибір уже виконаних тестів, які виконуються повторно, щоб переконатися, що існуючі функції працюють відповідно до вимог.

Це тестування проводиться, щоб переконатися, що нові зміни коду не мають побічних ефектів на існуючий функціонал. Це гарантує, що старий код продовжує працювати після внесення останніх змін у коді.

Регресійне тестування можна застосувати до функціональних аспектів програмного забезпечення (наприклад, певна функція більше не працює

належним чином у новій версії) або нефункціональних аспектів (наприклад, операція, яка виконується добре, виконується дуже повільно в новій версії).

В ході тестування були знайдені дефекти (загалом 21).

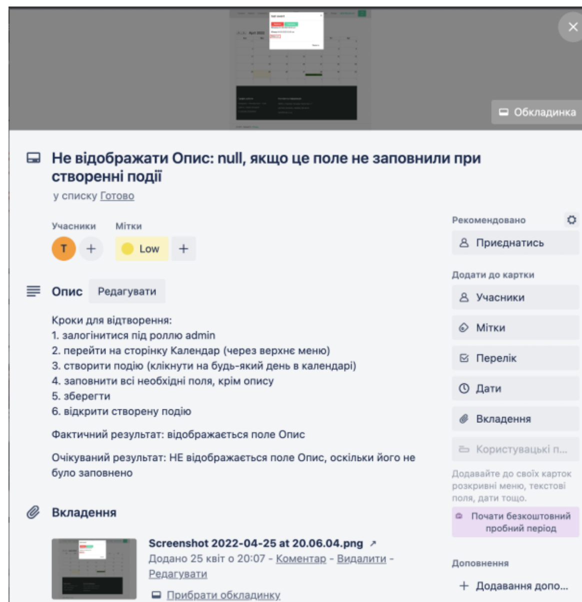

Рис.1. Приклад знайденого дефекту

Отже, як можна побачити на Рис.1, створені дефекти були зарепортовані за стандартом ISO 24765 і містять необхідні поля, такі як: назва (заголовок), пріоритет, кроки для відтворення, фактичний та очікуваний результат, скріншот із помилкою та відповідальна людина за виправлення дефекту.

Всі знайдені дефекти були записати на Trello дошку.

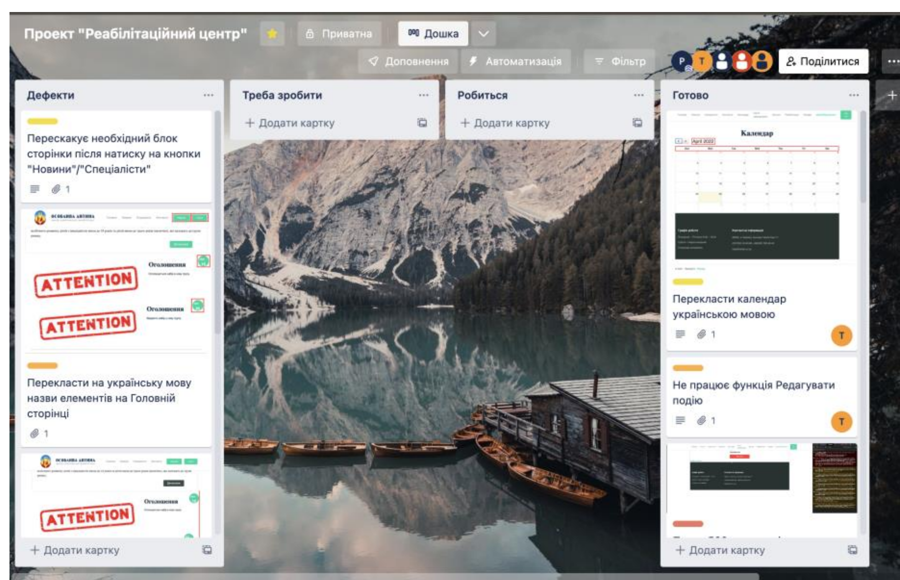

Рис.2. Trello дошка

Trello — популярний, простий і легкий у використанні інструмент для спільної роботи команди, який дозволяє організовувати проекти та все, що з ними пов'язано, у дошки. За допомогою Trello можна знайти будь-яку інформацію, необхідну для розробки, як-от:

- Над чим працюється?
- Хто над чим працює?
- Який прогрес досягає проект, чи є зрушення?

1.3 BDD (Behaviour-Driven Development) підхід для написання автоматизованих тестів

Розробка, орієнтована на поведінку (BDD) — це частина розробки, орієнтованої на тестування (TDD). BDD використовує зрозумілі людині описи вимог користувача програмного забезпечення як основу для тестування програмного забезпечення. Як і Domain Driven Design (DDD), раннім кроком у BDD є визначення спільної мети між зацікавленими сторонами, експертами в галузі та інженерами. Цей процес передбачає визначення сутностей, подій і результатів, які цікавлять користувачів, і надання їм рішень, з якими всі можуть погодитися.

Люди, що використовують даних підхід BDD потім застосовують цей так званий словник для створення специфічної для домену мови, яку вони можуть використовувати для кодування системних тестів, таких як тести прийнятності користувача (UAT).

Кожен тест базується на історії користувача (User Story), написаній офіційно визначеною загальноприйнятою мовою на основі англійської.

BDD – це спосіб роботи команд (які працюють над розробкою програмного забезпечення), який усуває прірву між людьми зорієнтованими на бізнес та технічними людьми за допомогою:

- Заохочення співпраці між різними ролями у команді для формування спільного розуміння проблеми, яку потрібно вирішити

- Швидкі невеликі ітерації для збільшення зворотного зв'язку та потоку цінностей
- Створення системної документації, яка автоматично перевіряється на поведінку систем.

Також варто зазначити, що поряд із BDD процесом, часто використовується TDD (Test Driven Development) підхід.

Проте, зазвичай інженери стикаються з різними проблемами під час виконання тестової розробки (TDD), такі як:

- З чого почати
- Що тестувати, а що ні
- Скільки тестувати за один раз
- Як назвати тести
- Як зрозуміти, чому тест провалився

Рішенням усіх цих проблем є розробка, орієнтована на поведінку (BDD). Вона розвинувся на основі усталених гнучких практик і розроблена, щоб зробити їх більш доступними та ефективними для команд, які не знайомі з гнучкою розробкою програмного забезпечення. З часом BDD розширився, щоб охопити ширшу картину гнучкого аналізу та автоматизованого приймального тестування.

Основна відмінність між TDD і BDD полягає в тому, що:

- TDD описує, як працює програмне забезпечення.
- З іншого боку, BDD:
- Описує, як кінцевий користувач використовує програмне забезпечення.
- Сприяє співпраці та спілкуванню.
- Наголошує на прикладах поведінки системи.
- Націлений на виконувані специфікації, отримані з прикладів.

1.4 E2E (End to End) тестування в автоматизації

Наскрізне тестування (тестування E2E) — це метод тестування програмного забезпечення, який передбачає тестування робочого процесу програми від початку до кінця. Цей метод спрямований на повторення сценаріїв реального користувача для перевірки системи на інтеграцію та цілісність даних.

По суті, тест проходить кожну операцію, яку може виконати програма, щоб перевірити, як програма взаємодіє з апаратним забезпеченням, підключенням до мережі, зовнішніми залежностями, базами даних та іншими програмами. Зазвичай тестування E2E виконується після завершення функціонального та системного тестування.

Наведені нижче дії необхідні для початку та завершення будь-якого E2E тесту:

- аналіз вимог, майже чітке уявлення про те, як програма має працювати в кожному аспекті;
- налаштування тестового середовища відповідно до всіх вимог;
- аналіз вимог до програмного та апаратного забезпечення;
- створити перелік, як кожна система повинна реагувати на дії користувача;
- перелічити методи тестування, необхідні для перевірки дій, відповідей системи;
- створити чіткі описи стандартів (мова, інструменти тощо), яких слід дотримуватися в кожному тесті;
- спроектувати тестові випадки;
- виконання тестів, вивчення та збереження результатів.

Звичайно, виявлення помилок у складному робочому процесі спричиняє труднощі. Нижче наведено два основних:

- *Створення робочих процесів*: щоб перевірити робочий процес програми, тестові випадки в наборі тестів E2E потрібно запускати в певній послідовності. Ця послідовність має відповідати шляху кінцевого користувача

під час навігації програмою. Створення пакетів тестів, які відповідають цьому робочому процесу, може бути важким, особливо тому, що вони зазвичай передбачають створення та виконання тисяч тестів.

- *Доступ до тестового середовища*: тестувати програми в середовищах для розробників легко. Однак кожну програму потрібно тестувати в клієнтському або робочому середовищах. Швидше за все, робочі середовища не завжди доступні для тестування. Тестувальники повинні інсталювати локальні аплікації та входити у віртуальні машини, навіть якщо вони є. Тестувальники також повинні підготуватися до таких проблем, як оновлення системи, які можуть перервати виконання тесту, і запобігати їм. Найкращий спосіб отримати доступ до ідеального тестового середовища – це тестувати на реальній хмарі пристрою.

Чому автоматизація тестування є надзвичайно важливою?

Не дивлячись на той факт, що ніщо не може замінити те, що може запропонувати людський розум, ручне тестування часом може бути виснажливим, повторюваним і нудним. Більше того, оскільки додатки з часом стають лише більшими, їхнє тестування вручну стає ще складнішим, забираючи продуктивний час і зусилля команди.

Оскільки ручне тестування також схильне до помилок, рівень впровадження дефектів значно вищий, що призводить до низької якості програмного забезпечення. Згідно зі звітом CISQ (Консорціум з якості IT-програмного забезпечення), вартість неякісного програмного забезпечення в США склала приблизно 2,84 трильйона доларів. На детальному рівні, якщо виправлення помилки на етапі розробки коштує 100 доларів США, то виправлення тієї ж помилки на продакшені, тобто у кінцевого користувача, коштує 10 000 доларів США.

Автоматизація тестування зменшує випадки людських помилок, щоразу генеруючи послідовні та надійні результати. Це означає вищу якість, менші

витрати на виправлення помилок, менше зусиль і вищу продуктивність для команд.

Загалом, автоматизоване тестування — це швидший спосіб отримати відгук про невдачі, ніж тестування вручну, відповідно до принципів «Fail fast, Fail early» («Чим швидше – тим краще»). Це допомагає забезпечити якість так, як ніколи не зможе ручне тестування.

§2. Реалізація фреймворку

2.1 Опис архітектури проекту, використовуючи POM (Page Object Model) паттерн проектування

POM — це шаблон проектування, який зазвичай використовується в Selenium для автоматизації тестових випадків. Цей шаблон проектування можна використовувати з будь-якою структурою, як-от керована ключовими словами, керована даними, гібридна структура тощо.

Об'єкт «Сторінка» — це об'єктно-орієнтований клас, який діє як інтерфейс для сторінки вашої програми, що тестується. Клас сторінки містить веб-елементи та методи для взаємодії з веб-елементами. Автоматизуючи тестові випадки, ми створюємо об'єкт цих класів сторінок і взаємодіємо з веб-елементами, викликаючи методи цих класів.

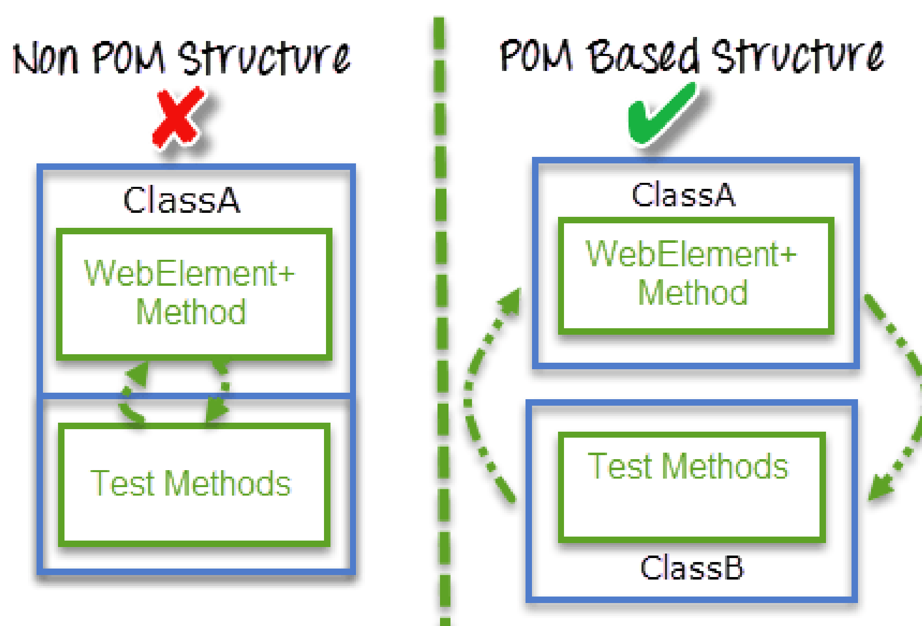

Рис.3. Схематичний розбір POM архітектури

Цей шаблон дизайну використовується в Selenium, де веб-сторінки представлені відповідним класом, а веб-елементи представлені змінними класу, а всі взаємодії можливі через методи або, скажімо, функції класу.

Переваги використання даного патерну:

Можливість повторного використання: за потреби ми можемо повторно використовувати клас сторінки в різних тестах, що означає, що нам не потрібно писати код для ідентифікації веб-елементів і методів взаємодії з ними для кожного тесту.

Ремонтопридатність: як ми бачимо з наведеного вище зображення, тестовий приклад і клас сторінки відрізняються один від одного, що означає, що ми можемо легко оновити код, якщо додається новий веб-елемент або оновиться існуючий.

Читабельність: як ми бачимо на зображенні вище, код сторінки розділений від тестового коду, що допомагає покращити читабельність коду.

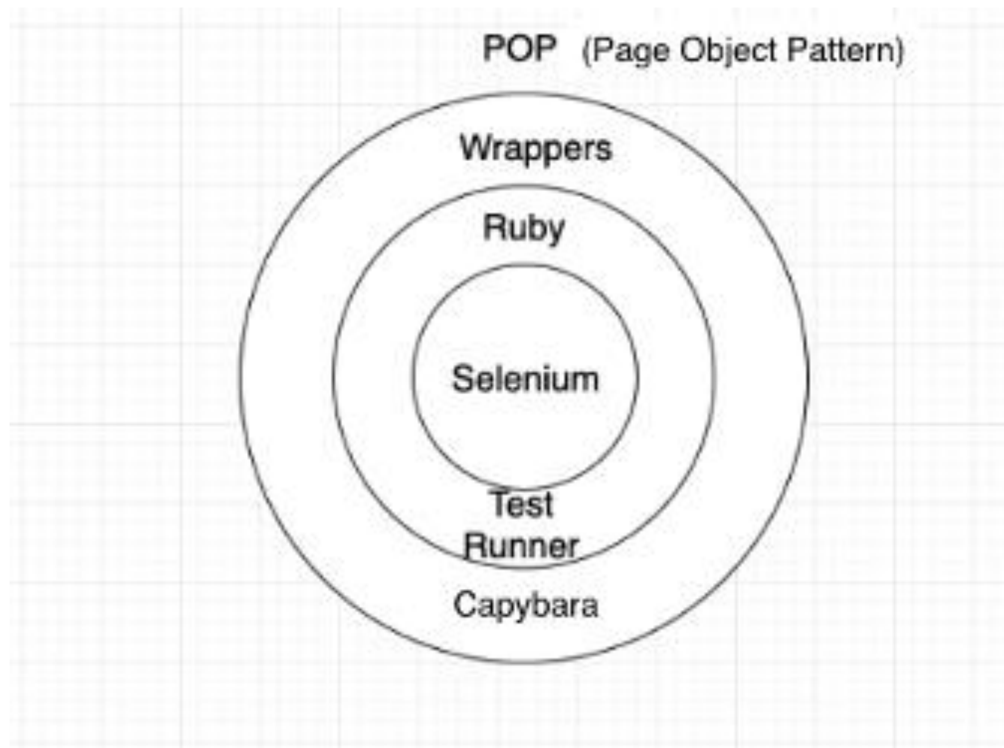

Рис. 4. Схема фреймворку

Для розуміння всього процесу, розглянемо вищенаведену діаграму: Отже, в основі всього лежить Selenium - інструмент для роботи із веб елементами, далі використовуємо мову програмування Ruby і Test Runner (у даному випадку Cucumber) для написання і рану тестів. Для того, аби взаємодіяти із Selenium, використовуємо Wrappers, а саме – Capybara.

Page Object, незважаючи на його назву, застосовується не лише до сторінок, але й до будь-яких значущих елементів або групи елементів на сторінці. Завдяки сучасній тенденції створення односторінкових додатків уся сторінка складається з розділів, таких як верхня частина (тобто хедер), нижня частина (футер), меню, розділ вмісту тощо.

Page Object є класичним прикладом інкапсуляції, тобто насправді приховують деталі структури інтерфейсу користувача від тесту. Як і будь-яка інкапсуляція, вона дає вам дві переваги:

- вона маніпулює інтерфейсом користувача в одному місці та робить тестовий код легшим для розуміння.
- оскільки логіка полягає в намірах тесту, він не захаращений деталями інтерфейсу.

Одним із найпопулярніших бібліотек для роботи із об'єктами сторінки для приймального тестування в спільноті Ruby є бібліотека site_prism.

WEB автоматизовані тести стають все більш популярними, оскільки вони включені в Rails як системні тести. Однак багато розробників намагаються уникати їх через повільний час виконання та високу вартість обслуговування. Зі швидкістю нічого не можна зробити — тести веб-переглядача ніколи не будуть такими швидкими, як модульні тести. Це компроміс, на який готові піти деякі клієнти, оскільки модульні тести взагалі не дозволяють перевірити роботу кінцевого користувача.

З іншого боку, зручність підтримки актуальності тестів — це те, що завжди можна покращити. Відповідь на питання «Чому ви не пишете наскрізні тести?» те, що часто чує тестувальник від колег-розробників, звучить так: «Зміни в розмітці HTML порушують збірки, тому ми надто часто змінюємо тести». Це також вірно для більшості наборів тестів: навіть якщо зміна HTML/CSS/JS не вплине на взаємодію з користувачем, це може змусити тести впасти, не вдало пройти.

Одним із основних класів у site_prism є Page class. Він описує елементи та розділи на певній сторінці.

```
class LoginPage < SitePrism::Page
  set_url page_url 'https://rehabcv20211216123152.azurewebsites.net/Account/Login'

  element name :login_field, *find_args '#Email'
  element name :password_field, *find_args '#Password'
  element name :submit_button, *find_args ':nth-child(4) > input'
end
```

Рис. 5. Приклад описаної сторінки

Метод set_url дозволяє надати URL-адресу вашої сторінки для навігації. Також site_prism використовує адресовану бібліотеку, а set_url метод можна параметризувати.

У більшості випадків об'єкт сторінки створюється як контейнер для елементів або групи елементів. Переходимо до визначення елемента. Перший параметр - ім'я (див. Рис.5). Інші параметри є аргументами пошуку capybara. У нашому випадку selenium шукатиме за css селектором ':nth-child(4) > input' або за id = '#Email'.

Тепер, якщо наша розмітка зміниться, нам потрібно лише змінити клас сторінки, не потрібно змінювати тести. Ще одна важлива перевага полягає в тому, що наші сторінки можуть бути модульними, складатися з кількох розділів. І ці розділи можна використовувати повторно.

Після визначення елемента його можна використовувати таким чином (див. Рис.6):

```
And(/^I go to login page$/) do
  @main_page.log_in_link.click
end
```

Рис.6. Приклад написаного кроку для тесту

Сам тест у нашому випадку має наступний вигляд:

```
Feature: Sign in
  As a visitor
  I want to be able to sign in
  In order to became a user

  Background: Way to login page
#    Given there is a registered user
    When I am on Special Child site
    And I go to login page

Scenario: sign in (UI checking Sign in)
  Then login page is opened
  And login page should contain login field
  And login page should contain password field
  And login page should contain submit button
```

Рис.7. Приклад написаного тесту на авторизацію (UI перевірка)

Отже, SitePrism бібліотека робить тестування функцій більш захоплюючим і забезпечує хороший баланс між зручністю читання і функціональними можливостями. Особливо корисно, що є можливість використовувати HTML-сторінку, користуючись об'єктом, де взаємодія зі сторінкою відбувається за допомогою методів.

Ще одна річ, яка є цікавою в об'єктах сторінки – це інкапсуляція, яку надає дана бібліотека. Наприклад, якщо необхідно буде змінити селектор CSS input#Email на input#Login, тоді просто потрібно змінити селектор CSS у відповідному методі element та на відповідній сторінці.

2.2 Налаштування проекту (Gemfile)

Що таке Ruby Gem?

Gem — це пакет, бібліотека, яку можна завантажити та встановити. Коли перед розробниками постає задача додати новий функціонал до програми, необхідно встановити певну бібліотеку, тобто інсталювати gem.

Gem-и дозволяють:

- Додати додаткову функціональність до програми Ruby;
- Легко працювати із зовнішніми додатками (наприклад, API);
- Створення веб-додатку.

Це лише деякі приклади. Кожен gem має свою мету.

Мета використання gem-ів?

- Спосіб обміну бібліотеками та інструментами в Ruby;
- Файлова структура та формат gem-ів дозволяє легко зрозуміти, як вони працюють;
- Файл специфікації (".spec"), який постачається з кожним gem-ом, описує залежності (інші необхідні gem-и), тому код має все необхідне для роботи.

Завдяки RubyGems у нас є багата екосистема корисних бібліотек, які можна встановити лише використовуючи одну команду gem install.

Список бібліотек, необхідних для певного проекту, можна вказати в спеціальному файлі під назвою «Gemfile», щоб Bundler міг їх автоматично встановити.

Що таке Bundler?

Це менеджер управління залежностями gem-ів в Ruby додатках. Ця утиліта дозволяє легко встановлювати необхідні gem-и для програми, при цьому зовсім не залежатиме від встановлених в системі. Цю утиліту можна використовувати для будь-якого Ruby фреймворку.

Встановлюємо цю утиліту як інший gem, використовуючи команду: gem install bundler.

Далі необхідно створити файл Gemfile, який є конфігом для Bundler-а. Для цього потрібно перейти в директорію проекту та виконати команду: bundle init.

У цьому файлі задаються всі необхідні залежності gem-ів. Розглянемо які саме можливості надає цей конфіг файл.

Спочатку задається ресурс, звідки за замовчуванням будуть встановлюватися gem-и: source 'https://rubygems.org' .

Далі вже йде список gem-ів, які потрібні для роботи програми. Тут варто зазначити, що gem-и можна об'єднувати в групи і далі встановлювати лише певні групи. За замовчуванням усі gem-и включаються у групу default. Якщо потрібна саме конкретна версія gem-а можна задати її номер версії. Є можливість задати ім'я файлу, який буде підключений під час підключення бібліотеки Bundler-ом. За замовчуванням це ім'я gem-а, тому в більшості випадків нічого ставити не потрібно. Опція яка дозволяє задати ім'я файлу для підключення називається require, використовується вона наступним способом, наприклад: require 'capybara/cucumber'.

Gemfile, який використовується у даному фреймворку:

```
source 'https://rubygems.org'

gem 'byebug'
gem 'hashie'
gem 'nokogiri'
gem 'capybara'
gem 'cucumber'
gem 'rubocop'
gem 'require_all'
gem 'rspec-expectations'
gem 'selenium-webdriver'
gem 'site_prism'
gem 'typhoeus'
gem 'xpath'
```

Рис. 8. Приклад Gemfile-у

Важливим ще є той факт, що gem-и, які записані в Gemfile, повинні записувати в алфавітному порядку (за правилами написання Ruby коду).

Коротко про кожен gem, що використовується у проекті:

1. Byebug — це налагоджувач Ruby. Основний компонент забезпечує підтримку коду, на який можуть опиратися інтерфейси. Також він забезпечує обробку точок зупинки, а також має простий у використанні інтерфейс командного рядка;
2. Hashie — це набір класів, які додають хешам нові функціональності;
3. Nokogiri – дозволяє легко працювати з XML та HTML в Ruby. Він забезпечує розумний, простий для розуміння API для читання, написання, зміни та запитів документів. Він швидкий і відповідає стандартам, покладаючись на власні парсери;
4. Capybara — це інструмент тестування інтеграції для веб-додатків. Він імітує дію користувача у веб-сайті;
5. Cucumber – BDD інстумент;
6. Rubocop — це інструмент перевірки стилю та форматування коду Ruby. Він спрямований на забезпечення дотримання стилю написання коду на Ruby;
7. Require_all – простий спосіб завантажити код і збільшити облість видимості коду;
8. Rspec-expectations – надає простий, читабельний API для вираження очікуваних результатів прикладу коду;
9. Selenium_wendriver – реалізовує протокол W3C WebDriver для автоматизації популярних браузерів. Він має на меті імітувати поведінку реального користувача під час взаємодії з HTML програми. Насамперед він призначений для тестування веб-додатків, але й будь-яке веб-завдання можна автоматизувати;
10. Site_prism надає простий, зрозумілий і семантичний DSL для опису сайту. SitePrism реалізовує шаблон Page Object Model поверх Capybara;

11. Typhoeus - запускає HTTP-запити паралельно, чітко інкапсулюючи логіку обробки;
12. Xpath — це Ruby DSL для створення виразів XPath.

Ще один не менш важливий файл для налаштування фреймовку – файл env.rb .

```ruby
require 'capybara/cucumber'
require 'selenium-webdriver'
require 'site_prism'
require 'require_all'

require_all 'page_objects'
require_relative 'feature_helper'

def options
  Selenium::WebDriver::Chrome::Options.new(args: %w[window-size=1800,1000])
end

Capybara.default_driver = :selenium
Capybara.register_driver name :selenium do |app|
  Capybara::Selenium::Driver.new(app, **options { browser: :chrome, options: options } )
end

World(FeatureHelper)
```

Рис. 9. Файл env.rb

В даному файлі викликаються всі основні gem-и, встановлюється Selenium Webdriver за замовчуванням, розширення вікна, де будуть проходити тести та реалізовується основа Page Object паттерну.

2.3 Cucumber та Capybara. Особливості бібліотек

Що таке Cucumber?

Cucumber — це інструмент тестування, який підтримує розробку Behavior Driven Development (BDD). Він пропонує спосіб написання тестів, який може зрозуміти кожен, незалежно від його технічних знань. У BDD користувачі (бізнес-аналітики, власники продуктів) спочатку пишуть сценарії або приймальні тести, які описують поведінку системи з точки зору клієнта, для перегляду та підписання власниками продукту, перш ніж розробники напишуть свої коди. Фреймворк Cucumber використовує мову програмування Ruby.

Переваги Cucumber бібліотеки:

- Корисно залучати бізнес-стейкхолдерів, які не вміють легко читати код, але хочуть контролювати процес розробки;
- Інструмент Cucumber тестування зосереджений на досвіді кінцевого користувача;
- Стиль написання тестів дозволяє легше використовувати код у тестах;
- Швидке та просте налаштування та виконання тестів.

Для кожного Cucumber проекту існує окрема папка у корені проекту під назвою «features». Тут будуть розташовані всі Cucumber тести. У цій папці можна знайти додаткові підпапки, наприклад, step_definition і support.

Що таке Feature файл?

Feature файл містить високорівневий опис тестового сценарію простою мовою. Він відомий як Cucumber. Cucumber - це звичайна англійська мова

Feature файл Cucumber складається з таких компонентів:

- Feature: описує поточний тестовий сценарій, який потрібно виконати.
- Scenario: Сценарій описує кроки та очікуваний результат для конкретного тестового випадку.
- Схема сценарію: той самий сценарій можна виконати для кількох наборів даних за допомогою схеми сценарію.

- Background: блок тесту, який буде виконуватися перед кожним запуском сценарію.
- Given: визначає контекст тексту, який буде виконано. Використовуючи таблиці даних “Given”, крок також можна параметризувати.
- When: “When” вказує тестову дію, яку потрібно виконати. Фактичні кроки для тесту.
- Then: Очікуваний результат тесту може бути представлений у блоці «Then».

```
Feature: Sign in
  As a visitor
  I want to be able to sign in
  In order to became a user

  Background: Way to login page
    Given there is a registered user
    When I am on Special Child site
    And I go to login page

Scenario: sign in (UI checking Sign in)
  Then login page is opened
  And login page should contain login field
  And login page should contain password field
  And login page should contain submit button
```

Рис. 10. Приклад тесту із використанням всіх необхідних блоків

Описані вище блоки тесту – це тести написані на мові програмування Cucumber. Використовуючи методологію Gherkin.

Gherkin — це мова, зрозуміла для не технічних людей, яка допомагає описати поведінку (тобто бізнес логіку), не вдаючись у деталі реалізації. Це доменно-спеціальна мова для визначення тестів у форматі Cucumber для специфікацій, функціональностей. Він використовує зрозумілу мову для опису тестових випадків використання та дозволяє користувачам видаляти логічні деталі з тестів поведінки.

Синтаксис Gherkin:

Gherkin — це рядково-орієнтована мова, як YAML і Python. Кожен рядок називається кроком і починається ключовим словом, а термінали закінчуються зупинкою. Для відступу використовується табуляція або пробіл.

У цьому сценарії коментар можна додати будь-де, але він має починатися зі знака #. Компілятор зчитує кожен рядок після видалення ключових слів Gherkin, таких як: given, when, then, тощо.

Сценарії Gherkin: поєднує людську концепцію причини та наслідку з програмною концепцією введення/процесу/виведення.

Документ Gherkin має розширення .feature, так би мовити просто тестовий файл із привабливим розширенням. Cucumber читає документ Gherkin і виконує тест, щоб перевірити, чи програмне забезпечення поводиться відповідно до синтаксису Gherkin.

Кращі практики використання Gherkin:

- Кожен сценарій слід виконувати окремо;
- Кожна функція повинна виконуватися одночасно;
- Інформацію про кроки слід розділити окремо від тестів;
- Пов'яжіть сценарій із вимогами;
- Створюйте модульні та прості для розуміння кроки;
- Спробуйте поєднати всі спільні сценарії.

Що таке «Step Definition»?

Визначення кроку (Step Definition) відображає кроки тестового випадку у файлах функцій (введених Given/When/Then) у код. Він виконує кроки програми, що тестується, і перевіряє результати на очікувані результати. Щоб визначення кроку було виконано, воно має відповідати даному компоненту в об'єкті. Визначення кроку визначено у файлах ruby у розділі 'features/step_definitions/*_steps.rb'.

Для написання фактичного коду тестів, було вирішено обрати бібліотеку Capybara.

Capybara — це веб-платформа автоматизації, яка використовується для створення функціональних тестів, які імітують, як користувачі взаємодіятимуть із програмою.

Capybara — це бібліотека/gem, корисний для використання веб-драйвера. Крім того, він пропонує зручну для користувача мову DSL (Domain Specific Language), яка корисна для опису дій, які виконує базовий веб-драйвер. Важливо, сторінка завантажується за допомогою DSL (і веб-драйвера). Крім того, Capybara намагатиметься знайти відповідний елемент у DOM (модель об’єкта документа) і виконати дію, наприклад натиснути кнопку, посилання тощо.

Capybara базується на припущенні, що в сучасній веб-програмі потенційно все може відбуватися асинхронно. Щоразу, коли ви перевіряєте наявність певного вмісту або виконання певної умови, Capybara за замовчуванням чекає, поки вміст з’явиться або умова стане істинною. Тайм-аут для цього, звичайно, можна налаштувати.

Ключові переваги:

- Для додатків Rails і Rack налаштування не потрібні. Працює за замовчуванням;
- Інтуїтивно зрозумілий API, який імітує мову реального користувача;
- Потужні функції синхронізації означають, що вам ніколи не доведеться вручну чекати завершення асинхронних процесів.

```
And(/^I fill (login|password) field with my (login|password)$/) do |credentials|
  if credentials == 'login'
    @login_page.login_field.set(@admin_login)
  else
    @login_page.pass_field.set(@admin_pass)
  end
end
```

Рис.11. Приклад визначеного кроку

2.4 Інструкція для користування. Запуск тестів

Для написання та запуску тестів було вирішено використовувати середовище для розробки (IDE) – RubyMine.

Якщо підійти до питання формально, то IDE – це інтегроване середовище розробки. Іншими словами, це робоче місце розробника, всередині якого у нього під рукою будуть усі необхідні йому інструменти: редактор файлів, консоль, перегляд папок проекту тощо.

Основні функції IDE:

По-перше, це текстовий редактор. Як і у будь-якій IDE, тут є підсвітка коду, а також перевірка його на помилки через неуважність.

По-друге, в IDE, як правило, вбудована консоль: під час роботи не доведеться переключатися між вікнами, втрачаючи час. Файли зберігаються автоматично, а запуск програми — натисканням однієї клавіші. Плюс , звичайно ж, гарячі клавіші, щоб все вийшло ще швидше.

По-третє, є можливість прямо в IDE бачити структуру програми чи фреймворку. Без цього просто нікуди, якщо програма у складна. Не потрібно переключатися в менеджер файлів і шукати потрібний файл. Весь проект постійно у під рукою.

Ієрархія папок у створеному фреймворку буде мати наступний вигляд:

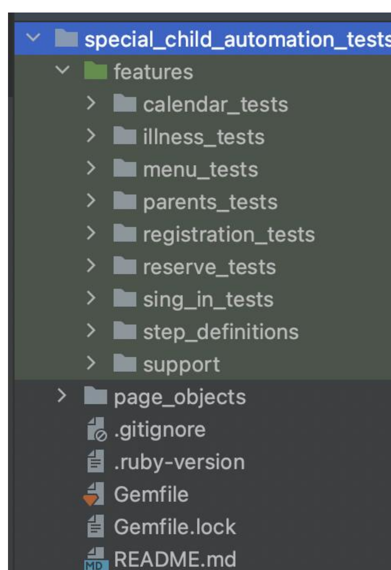

Рис.12. Структура проекту

Як вже було зазначено раніше, всі тести зосереджені в директорії features, розбиті власне по своїх директоріях, в залежності від функціоналу, що перевіряється. Також в директорії features розташована під-папка step_definitions із усіма фактичними реалізованими кроками. Варто звернути увагу на під-папку support, де зосереджені всі допоміжні файли, такі як: файл із константами, що використовуються в тестах, файл env.rb із налаштуваннями для фреймворку, файл hooks.rb, де розташовано всі Before/After хуки (дії, які потрібно виконувати перед та після кожного сценарію, наприклад, вилогінення із сайту після проходження тесту).

Також на рівні із директорією features, розташована папка page_objects. У ній зібрано всі файли із описами сторінок, необхідними локаторами та посиланнями на сторінки проекту «Особлива дитина».

Варто зазначити, що також у даному фреймворку має місце файл .ruby_version. Для даної роботи було вирішено обрати Ruby – 3.1.2, найбільш стабільну.

Запуск тестів:

Наразі є два шляхи запуску тестів: через термінал (командний рядок) або використовуючи RubyMine.

Для того, щоб запустити тести із терміналу необхідно перейти в директорію проекту, використовуючи команду: cd <шлях до папки проекту>. Щоб запустити всі тести, які є в проекті, потрібно виконати команду cucumber. Інтерпретатор сам знайде всі сценарії проекту та запустить їх по черзі. Щоб запустити тільки один конкретний файл із тестами, потрібно виконати наступну команду: cucumber <шлях до файлу>. Для того, аби запустити конкретний файл і один тест із необхідного файлу, потрібно ввести: cucumber <шлях до файлу:номер лінійки, з яких починається потрібний тест>. Наприклад, cucumber features/sing_in_tests/sign_in.feature:20

```
[mpetrus@FVFGD1S0Q05N special_child_automation_tests % cucumber
@web @UI @functional @e2e
Feature: Sign in
    As a visitor
    I want to be able to sign in
    In order to became a user

  Background: Way to login page     # features/sing_in_tests/sign_in.feature:8
    #      Given there is a registered user
    When I am on Special Child site # features/step_definitions/main_page_steps.rb:1
    And I open login page           # features/step_definitions/main_page_steps.rb:6

  Scenario: sign in (UI checking Sign in)        # features/sing_in_tests/sign_in.feature:13
    Then login page should contain login field    # features/step_definitions/login_page_steps.rb:1
    And login page should contain password field # features/step_definitions/login_page_steps.rb:1
    And login page should contain submit button  # features/step_definitions/login_page_steps.rb:10
#
#  Scenario: sign in (functional positive flow)
#    And I fill 'login' field with my 'login'
#    And I fill 'password' field with my 'password'
#    And I click on 'submit' button
#    Then I am sign in the site
#
#  Scenario: sign in (functional negative flow with non-valid login)
#    And I fill 'login' field with my 'random login'
#    And I fill 'password' field with my 'password'
#    And I click on 'submit' button
#    Then I should see error message
#
#  Scenario: sign in (functional negative flow with non-valid password)
#    And I fill 'login' field with my 'login'
#    And I fill 'password' field with my 'random password'
#    And I click on 'submit' button
#    Then I should see error message

1 scenario (1 passed)
5 steps (5 passed)
0m5.119s
```

Рис.13. Приклад запуску тесту з терміналу

Ще один спосіб запуску тестів – використовуючи RubyMine IDE. Насправді, все просто та прозоро. Відкриваючи файл із розширенням feature, навпроти ключових слів Feature та Scenario з’являються зелені кнопки. Після натиску на них з’являється випадаючий список, де є можливість обрати функцію запуску свієї feature чи конкретного сценарію.

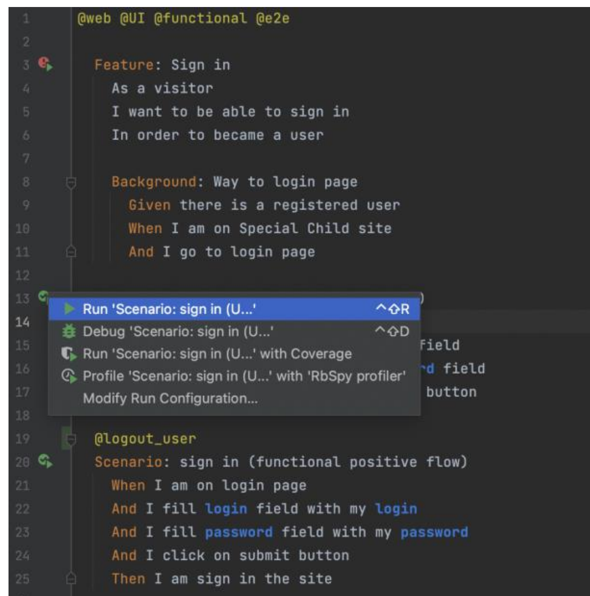

Рис. 14. Приклад запуску тесту із IDE

Під час проходження тесту в IDE відкривається вбудований термінал, де відображається вся інформація про тест: які кроки проходять, чи тест пройшов успішно, чи зазнав невдачі. Також виводяться докладні логи про проходження тесту.

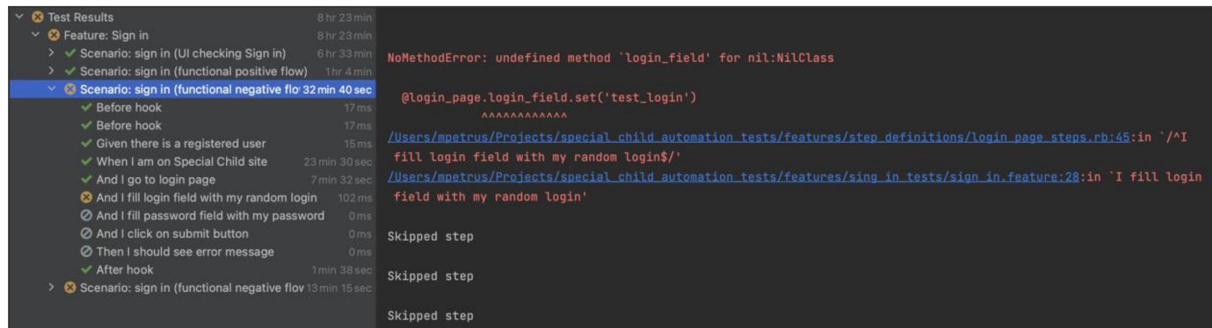

Рис.15. Приклад невдалого тесту

Висновки

Робота центрів реабілітації особливих людей/дітей не проста, потребує спеціальної професійної підготовки, сконцентрованої віддачі в роботі. Щоб якось полегшити виконання такої важливої справи, варто використовувати комп`ютерну техніку для забезпечення виконання типових, рутинних операцій. Тому важливо, аби наш проект працював коректно і без помилок.

Під час виконання дипломної роботи було мануально протестовано та розроблено фреймворк для автоматизації тестування проекту «Особлива дитина». В процесі виконання дипломної роботи були вирішені всі поставлені задачі:

- мануально протестовано проект «Особлива дитина»;
- вивчено нові технології: Ruby, Cucumber, Capybara, gem SitePrism;
- створено фреймворк для автоматизації тестування, використовуючи вивчені технології;
- автоматизовано найважливіші тестові випадки, забезпечуючи правильність роботи головної бізнес логіки.

Всі знайдені дефекти в ході тестування було записано в трекенговій системі Trello.

Отже, тестування – це важливий етап в розробці програми, тому що саме тестувальник бере на себе роль кінцевого користувача та може заздалегідь виявити дефект, що дозволяє в подальшому випустити на кінцевий ринок якісний продукт. Адже саме для проекту «Особлива дитина» надзвичайно важливо, щоб програмний продукт був чітким, зрозумілим та простим у користуванні.

Список використаної літератури

1. Офіційна документація Cucumber [Електронний ресурс]. – Режим доступу: **https://cucumber.io/docs/cucumber/**
2. Вступ до написання тестів на Cucumber [Електронний ресурс]. – Режим доступу: https://www.browserstack.com/guide/learn-about-cucumber-testing-tool
3. Офіційна документація Capybara [Електронний ресурс]. – Режим доступу: https://github.com/teamcapybara/capybara
4. Офіційна документація Capybara [Електронний ресурс]. – Режим доступу: https://rubygems.org/gems/capybara/versions/2.7.1
5. Офіційна документація SitePrism [Електронний ресурс]. – Режим доступу: https://github.com/site-prism/site_prism
6. Офіційна документація Ruby [Електронний ресурс]. – Режим доступу: https://ruby-doc.org/
7. Ruby tutorial [Електронний ресурс]. – Режим доступу: https://www.tutorialspoint.com/ruby/ruby_syntax.htm
8. Behavior Driven Development (BDD) and Functional Testing [Електронний ресурс]. – Режим доступу: https://medium.com/javascript-scene/behavior-driven-development-bdd-and-functional-testing-62084ad7f1f2
9. James Whittaker, Jason Arbon, Jeff Carollo: How Google Tests Software. - Addison-Wesley Professional; 1st edition (March 23, 2012). – Режим доступу до ресурсу: https://www.amazon.com/Google-Tests-Software-James-Whittaker/dp/0321803027
10. Matt Wynne, Aslak Hellesoy: The Cucumber Book: Behaviour-Driven Development for Testers and Developers. – Pragmatic Bookshelf; 1st edition (February 21, 2012). – Режим доступу до ресурсу: https://www.amazon.com/Cucumber-Book-Behaviour-Driven-Development-Programmers/dp/1934356808

11. Gemfile [Електронний ресурс]. – Режим доступу: https://bundler.io/guides/gemfile.html
12. Прикладна математика та інформаційні технології. – Чернівці, 2022 [с.269-271].
13. Петрусяк М. Тестування програмного забезпечення у проекті «Реабілітаційний центр «Особлива дитина»»: Матеріали міжнар. наук. конф., присвяченої 60-річчю кафедри прикладної математики та інформаційних технологій (м. Чернівці, 22-24 вересня 2022 р.). Чернівці, 2022. С. 269-271. URL: https://archer.chnu.edu.ua/xmlui /handle/123456789/6242

Додатки

1. Приклади тестів:

@web @UI @functional @e2e

Feature: Sign in

As a visitor

I want to be able to sign in

In order to became a user

Background: Way to login page

Given there is a registered user

When I am on Special Child site

And I go to login page

Scenario: sign in (UI checking Sign in)

Then login page is opened

And login page should contain **login** field

And login page should contain **password** field

And login page should contain submit button

@logout_user

Scenario: sign in (functional positive flow)

When I am on login page

And I fill **login** field with my **login**

And I fill **password** field with my **password**

And I click on submit button

Then I am sign in the site

Scenario: sign in (functional negative flow with non-valid login)

And I fill login field with my random login

And I fill **password** field with my **password**

And I click on submit button

Then I should see error message

Scenario: sign in (functional negative flow with non-valid password)

And I fill **login** field with my **login**

And I fill password field with my random password

And I click on submit button

Then I should see error message

2. Приклади реалізації кроків

```
When(/^I am on login page$/) do
  @login_page = LoginPage.new
end
Then(/^login page is opened$/) do
  expect(current_url).to include('/Account/Login')
end

And(/^login page should contain (login|password) field$/) do |field_name|
  case field_name
  when 'login'
    expect(page).to have_content('Електронна пошта/Логін')
  when 'password'
    expect(page).to have_content('Пароль')
  end
end

And(/^login page should contain submit button$/) do
  expect(page).to have_button('Увійти')
end

Given(/^there is a registered user$/) do
  @admin_login = 'admin@gmail.com'
  @admin_pass = 'P@$$w0rd'
end

And(/^I fill (login|password) field with my (login|password)$/) do |field, credentials|
  if field == 'login' && credentials == 'login'
    @login_page.login_field.set(@admin_login)
```

```
  else
    @login_page.pass_field.set(@admin_pass)
  end
end

And(/^I click on submit button$/) do
  @login_page.submit_button.click
end

Then(/^I am sign in the site$/) do
  expect(page).to have_css(':nth-child(6) > a')
  expect(page).to have_content(@admin_login)
end

And(/^I fill login field with my random login$/) do
  @login_page.login_field.set('test_login')
end

Then(/^I should see error message$/) do
  within(@login_page.error_message) do
    expect(page).to
have_content(SupportData::Errors.LOGIN_ERROR_MESSAGE)
  end
end

And(/^I fill password field with my random password$/) do
  @login_page.pass_field.set('test_pass')
end
```

3. Приклад опису сторінки:

```
class LoginPage < SitePrism::Page
  set_url 'https://rehabcv20211216123152.azurewebsites.net/Account/Login'

  element :login_field, '#Email'
  element :pass_field, '#Password'
  element :submit_button, ':nth-child(4) > input'
  element :error_message, '.validation-summary-errors'
end
```

4. Приклад реалізації допоміжних кроків

```
When /^I debug with byebug$/ do
  ##
  # The first `byebug' is always skipped.
  #
  byebug

  byebug
end

Then (/^I pause$/) do
  binding.pry
end

# I wait 2 seconds
# I wait 0.5 seconds
# I wait 0.55 seconds
Given /^I wait (\d+(?:\.\d+)?) seconds?$/ do |sec|
  sleep(sec.to_f)
end
And /^I (reload|refresh) the page$/ do |action|
  case action
  when "reload"
    visit current_path
  when "refresh"
    page.evaluate_script("window.location.reload()");
  end
  step 'I wait for the page to load'
end
```

5. Налаштування фреймворку

```
require 'capybara/cucumber'
require 'selenium-webdriver'
require 'site_prism'
require 'require_all'

require_all 'page_objects'
require_relative 'feature_helper'

def options
Selenium::WebDriver::Chrome::Options.new(args: %w[window-size=1800,1000])
end

Capybara.default_driver = :selenium
Capybara.register_driver :selenium do |app|
Capybara::Selenium::Driver.new(app,  browser: :chrome, options: options)
end

World(FeatureHelper)
```

6. Приклад файлу із допоміжними константами

```
# frozen_string_literal: true

module SupportData
  module Errors
    LOGIN_ERROR_MESSAGE = 'Невірний логін або пароль'
  end
end
```