Informatyka techniczna i sztuczna inteligencja
Technical informatics and Artificial Intelligence
Engineer of XXI Century'2024

Automation of software processes and tests using IT tools

Michał Niemczyk [1], Sławomir Herma[2,*]

1 *mgr inż., V Liceum Ogólnokształcące w Bielsku-Białej, ul. J.Lompy 10, 43-300 Bielsko-Biała, Polska, michal.niemczyk@lo5.bielsko.pl*
2 *dr inż., Uniwersytet Bielsko-Bialski, Wydział Budowy Maszyn i Informatyki, sherma@ubb.edu.pl*
* *Corresponding author, sherma@ubb.edu.pl****Abstract:***

Software process and test automation plays a key role in ensuring the quality of information systems and improving the software lifecycle. The article provides an overview of the tools and methods used in automating business processes and functional, performance and regression testing. Special attention is given to open-source platforms such as Selenium, TestNG and Jenkins, which are gaining popularity due to their flexibility and broad functionality. The importance of CI/CD tools and test data management techniques in agile environments is also discussed. The purpose of the article is to show the practical applications of IT tools in optimizing testing processes and their impact on software quality and shortening time to market.

Keywords: Process automation; software testing; Selenium; TestNG; CI/CD; software quality; IT tools;

Automatyzacja procesów i testów oprogramowania z wykorzystaniem narzędzi informatycznych

Michał Niemczyk [1], Sławomir Herma[2,*]

1 *mgr inż., V Liceum Ogólnokształcące w Bielsku-Białej, ul. J.Lompy 10, 43-300 Bielsko-Biała, Polska, michal.niemczyk@lo5.bielsko.pl*
2 *dr inż., Uniwersytet Bielsko-Bialski, Wydział Budowy Maszyn i Informatyki, sherma@ubb.edu.pl*
* *Corresponding author, sherma@ubb.edu.pl*

Streszczenie: Automatyzacja procesów i testów oprogramowania odgrywa kluczową rolę w zapewnianiu wysokiej jakości systemów informatycznych oraz usprawnianiu cyklu życia oprogramowania. Artykuł przedstawia przegląd narzędzi oraz metod stosowanych w automatyzacji procesów biznesowych i testów funkcjonalnych, wydajnościowych oraz regresyjnych. Szczególną uwagę poświęcono platformom open-source, takim jak Selenium, TestNG oraz Jenkins, które zyskują na popularności ze względu na elastyczność i szeroką funkcjonalność. Omówiono również znaczenie narzędzi CI/CD oraz technik zarządzania danymi testowymi w środowiskach zwinnych. Celem artykułu jest ukazanie praktycznych zastosowań narzędzi informatycznych w optymalizacji procesów testowych oraz ich wpływu na jakość oprogramowania i skrócenie czasu dostarczania produktu na rynek.

Słowa kluczowe: automatyzacja procesów; testy oprogramowania; Selenium; TestNG; CI/CD; jakość oprogramowania; narzędzia informatyczne;

1. Wprowadzenie

Automatyzacja testów pełni fundamentalną rolę w współczesnych procesach wytwarzania oprogramowania, oferując znaczącą redukcję kosztów oraz czasu potrzebnego na przeprowadzanie testów, a także minimalizację ryzyka wystąpienia błędów. Jest to szczególnie istotne w środowiskach Agile[1] i DevOps, gdzie kluczowe znaczenie ma szybkie dostosowywanie się do zmieniających się wymagań biznesowych i technologicznych. Narzędzia takie jak Selenium[2], Playwright, rozszerzone o JUnit, TestNG, Allure czy Jenkins dla procesów CI/CD umożliwiają zautomatyzowanie testów funkcjonalnych, wydajnościowych oraz regresyjnych, jednocześnie wspierając procesy

DOI: https://doi.org/10.53052/9788367652292.35

ciągłej integracji i dostarczania oprogramowania. Dzięki automatyzacji możliwe jest również przeprowadzanie testów w różnych środowiskach i konfiguracjach systemowych, co zwiększa ich wszechstronność i wartość dla organizacji.

Jednym z najistotniejszych atutów testów automatycznych jest ich powtarzalność. Automatyczne wykonywanie testów eliminuje błędy ludzkie, które często pojawiają się podczas wielokrotnego powtarzania tych samych czynności w testach manualnych. Przykładowo, regresyjne testowanie funkcji po każdej zmianie w kodzie może być czasochłonne i podatne na pomyłki, jeśli wykonywane jest ręcznie. Automatyzacja pozwala na precyzyjne odtworzenie kroków testowych w sposób niezawodny, zwiększając tym samym spójność wyników oraz dokładność testów.

Jednak automatyzacja wiąże się także z pewnymi wyzwaniami. Początkowy koszt wdrożenia narzędzi i stworzenia odpowiednich testów może być wysoki, szczególnie w przypadku dużych i skomplikowanych systemów. Ponadto, utrzymanie zestawów testów wymaga specjalistycznej wiedzy, ponieważ każda zmiana w aplikacji może wymagać aktualizacji skryptów testowych.

Współczesne trendy technologiczne wskazują na coraz szersze zastosowanie sztucznej inteligencji w procesie automatyzacji testów. Algorytmy AI pomagają w generowaniu przypadków testowych, identyfikowaniu optymalnych scenariuszy oraz wykrywaniu regresji. Narzędzia wspomagane AI mogą również analizować wyniki testów, wskazując potencjalne obszary problematyczne, co znacząco przyspiesza proces identyfikacji i rozwiązywania błędów.

Celem niniejszego artykułu jest dogłębne omówienie korzyści płynących z automatyzacji testów oraz wyzwań związanych z jej wdrażaniem, a także przedstawienie wpływu tej technologii na jakość i efektywność wytwarzania oprogramowania. Praca ta bazuje na najnowszych publikacjach naukowych, studiach przypadków oraz praktycznych wdrożeniach narzędzi, co pozwala na kompleksowe spojrzenie na aktualny stan wiedzy w tej dziedzinie.

2. Materiały i Metody

Automatyzacja testów znacząco wpływa na efektywność i jakość procesu wytwarzania oprogramowania, szczególnie w przypadku aplikacji webowych, które muszą spełniać wysokie wymagania w zakresie wydajności, funkcjonalności i kompatybilności. Narzędzia takie jak Selenium oraz TestNG odgrywają kluczową rolę w realizacji zautomatyzowanych testów aplikacji webowych, zapewniając wsparcie na różnych etapach testowania.

Testy automatyczne stanowią wszechstronne narzędzie wspierające proces zapewniania jakości oprogramowania (QA), umożliwiając szybkie, dokładne i powtarzalne weryfikowanie działania aplikacji. Mogą być stosowane do testowania wielu elementów systemów informatycznych, takich jak interfejs użytkownika (UI), logika biznesowa, warstwa aplikacyjna, API, bazy danych, a także wydajność i zgodność aplikacji z przeglądarkami czy wymaganiami regulacyjnymi. Testy UI pozwalają na sprawdzanie wyglądu, układu oraz interakcji elementów, takich jak formularze, przyciski czy dynamiczne komponenty, co umożliwia kompleksową weryfikację doświadczenia użytkownika. Testy logiki biznesowej i aplikacyjnej sprawdzają poprawność przepływów pracy oraz operacji, takich jak kalkulacje czy walidacja danych. Testy API pozwalają na weryfikację komunikacji między komponentami systemu, zapewniając poprawność odpowiedzi i zgodność z dokumentacją. Automatyzacja testów obejmuje także operacje na bazach danych, weryfikując poprawność zapisów i odczytów oraz ich zgodność z wymaganiami aplikacji. Testy wydajności i obciążenia umożliwiają symulację wielu użytkowników, co pozwala ocenić stabilność systemu pod dużym obciążeniem. Testy regresyjne weryfikują, czy wprowadzone zmiany nie wpłynęły negatywnie na istniejącą funkcjonalność, a testy zgodności pozwalają na sprawdzanie działania aplikacji w różnych przeglądarkach, systemach operacyjnych oraz pod kątem zgodności z regulacjami, takimi jak WCAG czy RODO.

Korzyści wynikające z zastosowania testów automatycznych są znaczące. Automatyzacja przyspiesza czas testowania poprzez wielokrotne, szybkie uruchamianie testów, eliminując konieczność manualnej weryfikacji powtarzalnych scenariuszy. Dzięki swojej powtarzalności testy automatyczne eliminują błędy ludzkie, zapewniając spójność i wiarygodność wyników. Skalowalność automatyzacji pozwala na testowanie aplikacji o dużej złożoności oraz w różnych środowiskach, co jest kluczowe w przypadku współczesnych, rozproszonych systemów. Regularne uruchamianie testów w pipeline'ach CI/CD umożliwia wczesne wykrywanie błędów i ich szybką naprawę, co obniża koszty wprowadzania poprawek. Automatyzacja testów generuje szczegółowe raporty, które ułatwiają analizę wyników i identyfikację źródeł problemów, co wspiera szybkie podejmowanie decyzji. W dłuższym okresie

zastosowanie testów automatycznych prowadzi do znaczących oszczędności, mimo początkowych kosztów związanych
z wdrożeniem i utrzymaniem infrastruktury testowej. Elastyczność tego podejścia pozwala na jego skuteczne wykorzystanie
w środowiskach Agile i DevOps, wspierając iteracyjne modele pracy oraz szybkie wprowadzanie zmian[3,4].

Selenium to narzędzie typu open-source, które umożliwia automatyzację testów interfejsów użytkownika (UI) aplikacji webowych w różnych przeglądarkach (m.in. Chrome, Firefox, Edge). Pozwala ono na symulowanie interakcji użytkownika
z aplikacją, takich jak wypełnianie formularzy, klikanie przycisków czy nawigacja po stronach. Dodatkową zaletą Selenium jest jego kompatybilność z wieloma językami programowania, w tym Java, Python, C#, oraz integracja z narzędziami do zarządzania testami i CI/CD[5].

TestNG[6] to framework testowy inspirowany JUnit, który rozszerza jego możliwości, oferując zaawansowane funkcje, takie jak:

- Grupowanie testów, co umożliwia uruchamianie wybranych zestawów testów w określonej kolejności.
- Tworzenie asercji, które precyzyjnie weryfikują poprawność działania aplikacji.
- Generowanie raportów w czasie rzeczywistym, co ułatwia analizę wyników testów.
- Wsparcie dla testów równoległych, co pozwala na szybsze wykonanie dużych zestawów testów.

Stos technologiczny Selenium + TestNG[7]

Połączenie Selenium z TestNG jest powszechnie stosowane w testowaniu aplikacji webowych, ponieważ oba narzędzia doskonale się uzupełniają. Selenium odpowiada za interakcje z przeglądarką, a TestNG zapewnia strukturalne zarządzanie przypadkami testowymi oraz analizę wyników. Przykładowy proces automatyzacji testów wygląda następująco:

1. Konfiguracja środowiska: Instalacja Selenium WebDriver oraz konfiguracja TestNG w projekcie opartym na Maven lub Gradle.
2. Tworzenie skryptów testowych: Używanie Selenium do interakcji z elementami interfejsu aplikacji webowej (np. przyciski, pola tekstowe) oraz TestNG do definiowania scenariuszy testowych z adnotacjami (@Test, @BeforeSuite, @AfterTest itp.).
3. Raportowanie wyników: TestNG automatycznie generuje raporty HTML, które przedstawiają szczegółowe wyniki wykonania testów z wykorzystaniem odpowiednich bibliotek raportujących.
4. Integracja z CI/CD: Jenkins lub GitLab CI/CD mogą być użyte do automatycznego uruchamiania testów przy każdej aktualizacji kodu, co pozwala na wczesne wykrywanie regresji i błędów.

Zalety stosu technologicznego Selenium + TestNG:

Obsługa wielu przeglądarek i systemów operacyjnych - Selenium WebDriver zapewnia pełne wsparcie dla popularnych przeglądarek internetowych, takich jak Chrome, Firefox, Edge, Safari, oraz możliwość uruchamiania testów na różnych systemach operacyjnych, w tym Windows, macOS, i Linux. Dzięki temu można zapewnić wszechstronne testowanie aplikacji
w szerokim zakresie środowisk.

Łatwa integracja z narzędziami CI/CD i repozytoriami kodu - TestNG i Selenium doskonale integrują się z narzędziami CI/CD, takimi jak Jenkins, GitLab CI/CD czy GitHub Actions, co pozwala na automatyczne uruchamianie testów przy każdym wdrożeniu lub aktualizacji kodu. Daje to możliwość szybkiego wykrywania regresji i zapewnia ciągłą kontrolę jakości.

Możliwość wykonywania testów równoległych - Dzięki wsparciu TestNG dla testów równoległych, można znacząco skrócić czas wykonania dużych zestawów testowych. TestNG pozwala na konfigurowanie równoczesnych testów na różnych przeglądarkach
i urządzeniach, co jest kluczowe w przypadku złożonych projektów.

Rozbudowane opcje raportowania z użyciem Allure i ExtentReports

- Allure: Umożliwia generowanie atrakcyjnych i szczegółowych raportów, które wizualizują wyniki testów w przystępny sposób. Raporty Allure oferują funkcje takie jak podsumowanie testów, szczegółowe opisy kroków, zrzuty ekranu
- z niepowodzeń, a także logi, które ułatwiają analizę błędów.
- ExtentReports: Dodatkowe narzędzie do generowania raportów, które wspiera personalizację wizualizacji wyników oraz integrację z logami i zrzutami ekranu. ExtentReports oferuje wsparcie dla interaktywnego śledzenia wyników testów, co zwiększa przejrzystość i ułatwia komunikację z interesariuszami projektu.

Bogata społeczność użytkowników i zasoby - Selenium oraz TestNG mają szeroką społeczność użytkowników, co oznacza łatwy dostęp do dokumentacji, poradników, tutoriali oraz gotowych przykładów kodu. Istnieją również liczne fora dyskusyjne i grupy wsparcia technicznego, które pomagają w rozwiązywaniu problemów.

Wady i wyzwania:
Wysoka krzywa uczenia się, szczególnie dla początkujących - Nowi użytkownicy mogą mieć trudności z opanowaniem zaawansowanych funkcji Selenium i TestNG, takich jak obsługa dynamicznych elementów, synchronizacja czy konfiguracja testów równoległych.

Konieczność częstego aktualizowania testów w przypadku dynamicznych interfejsów aplikacji - Aplikacje webowe, które korzystają z nowoczesnych frameworków, takich jak React czy Angular, często zmieniają swoje struktury DOM, co wymaga regularnego aktualizowania skryptów testowych.

Złożoność konfiguracji narzędzi raportowania - Chociaż Allure i ExtentReports są bardzo funkcjonalne, ich konfiguracja, integracja z istniejącymi projektami oraz optymalizacja raportów może być czasochłonna. Wymaga to również znajomości dodatkowych technologii, takich jak wtyczki Maven/Gradle czy konfiguracja logowania.

Wymaganie zaawansowanej wiedzy technicznej - Konfiguracja środowiska testowego, integracja z CI/CD, tworzenie testów równoległych czy obsługa dynamicznych elementów wymaga zaawansowanej wiedzy w zakresie programowania i narzędzi automatyzacji.

Wyższy koszt początkowy - Choć Selenium i TestNG są narzędziami open-source, czasochłonność wdrożenia i szkolenia zespołu w zakresie zaawansowanej automatyzacji może generować znaczne koszty początkowe.

Wykorzystanie ChatGPT do generowania danych testowych stanowi innowacyjne podejście w automatyzacji testów z użyciem Selenium i TestNG. ChatGPT, dzięki swoim zaawansowanym możliwościom przetwarzania języka naturalnego, może być używany do tworzenia realistycznych danych wejściowych, takich jak nazwy użytkowników, adresy e-mail, hasła, a także bardziej złożonych zestawów danych, np. scenariuszy zakupów w aplikacjach e-commerce czy formularzy kontaktowych. Dane te mogą być dynamicznie generowane w czasie rzeczywistym i bezpośrednio wykorzystywane w skryptach testowych Selenium, co eliminuje potrzebę ręcznego przygotowywania dużych zbiorów danych testowych.

Integracja ChatGPT z Selenium i TestNG odbywa się poprzez API OpenAI, które umożliwia generowanie danych i ich przechowywanie w plikach JSON lub bazach danych, skąd mogą być pobierane przez skrypty testowe. Takie podejście pozwala na zwiększenie różnorodności danych testowych, co poprawia wykrywalność błędów i lepiej odzwierciedla rzeczywiste warunki użytkowania aplikacji. Dodatkowo, ChatGPT może wspierać testy eksploracyjne, generując nietypowe dane lub przypadki krańcowe, co zwiększa pokrycie testowe.

Choć generowanie danych testowych przy użyciu ChatGPT niesie wiele korzyści, wymaga odpowiedniego przemyślenia pod kątem ochrony danych wrażliwych oraz zgodności z regulacjami prawnymi, takimi jak RODO czy HIPAA, w przypadku testowania aplikacji przechowujących dane użytkowników.

3. Wyniki

Testy automatyczne dla aplikacji webowych z wykorzystaniem Selenium WebDriver i TestNG umożliwiają wszechstronne testowanie funkcjonalności stron internetowych w różnych przeglądarkach i środowiskach.
Konfiguracja projektu testowego w Selenium + TestNG zgodnie z metodologią Page Object Model (POM) wymaga odpowiedniego przygotowania środowiska oraz organizacji struktury kodu. Taki projekt, poza podstawowymi funkcjami automatyzacji, jest rozbudowany o zaawansowane narzędzia i rozwiązania zwiększające skalowalność oraz czytelność kodu.

3.1 Struktura projektu i klasy bazowe

Projekt jest podzielony na odpowiednie foldery, które zwiększają czytelność i ułatwiają zarządzanie kodem:

- base: zawiera klasy bazowe, takie jak BaseTest i BasePage, zapewniające wspólne funkcjonalności dla wszystkich testów i stron.
- pages: przechowuje klasy Page Object Model, odpowiadające za mapowanie elementów i logikę działania stron.
- drivers: zawiera implementację wzorca singleton dla zarządzania WebDriverem (DriverSingleton).
- utils: przechowuje klasy narzędziowe, takie jak ConfigReader oraz pomocnicze metody.
- data: przechowuje pliki JSON z danymi testowymi dla konkretnych stron aplikacji.
- tests: zawiera klasy testowe, które używają POM oraz danych testowych.

3.2 Konfiguracja Maven i plik POM

W pliku pom.xml definiujemy zależności dla bibliotek używanych w projekcie.

```
<dependencies>
    <!-- Selenium -->
    <dependency>
        <groupId>org.seleniumhq.selenium</groupId>
        <artifactId>selenium-java</artifactId>
        <version>4.12.0</version>
    </dependency>

    <!-- TestNG -->
    <dependency>
        <groupId>org.testng</groupId>
        <artifactId>testng</artifactId>
        <version>7.8.0</version>
        <scope>test</scope>
    </dependency>

    <!-- WebDriverManager -->
    <dependency>
        <groupId>io.github.bonigarcia</groupId>
        <artifactId>webdrivermanager</artifactId>
        <version>5.5.0</version>
    </dependency>

    <!-- ExtentReports -->
    <dependency>
        <groupId>com.aventstack</groupId>
        <artifactId>extentreports</artifactId>
        <version>5.0.9</version>
    </dependency>

    <!-- Jackson Databind -->
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
        <version>2.15.2</version>
    </dependency>

    <!-- Awaitility -->
    <dependency>
        <groupId>org.awaitility</groupId>
        <artifactId>awaitility</artifactId>
        <version>4.2.0</version>
    </dependency>
</dependencies>
```

DOI: https://doi.org/10.53052/9788367652292.35

Powyższy fragment przedstawia sekcję dependencies pliku pom.xml, który definiuje zależności niezbędne do realizacji projektu automatyzacji testów oprogramowania z wykorzystaniem narzędzia Maven. Zależności te obejmują kluczowe biblioteki, które wspierają różne aspekty procesu testowego, takie jak interakcja z przeglądarką, zarządzanie WebDriverem, generowanie raportów czy przetwarzanie danych.

Pierwszą zdefiniowaną zależnością jest Selenium, kluczowe narzędzie w automatyzacji testów aplikacji webowych. Wersja 4.12.0 zapewnia wsparcie dla nowoczesnych przeglądarek oraz funkcjonalności związanych z protokołem WebDriver, co umożliwia precyzyjną interakcję z elementami stron internetowych. Kolejną zależnością jest TestNG, framework testowy w wersji 7.8.0, który wspiera strukturalne zarządzanie testami, takie jak parametryzacja, grupowanie, a także zaawansowane opcje raportowania i asercji. TestNG jest skonfigurowany z zakresem test, co oznacza, że będzie wykorzystywany wyłącznie podczas uruchamiania testów.

Trzecia zależność dotyczy WebDriverManager, biblioteki w wersji 5.5.0, która automatyzuje zarządzanie sterownikami przeglądarek. Dzięki niej unika się ręcznego pobierania i konfiguracji sterowników, co znacząco upraszcza proces integracji
i uruchamiania testów w różnych środowiskach. Kolejna biblioteka to ExtentReports w wersji 5.0.9, narzędzie służące do generowania szczegółowych raportów testowych. Raporty ExtentReports są przejrzyste i zawierają informacje o wynikach testów, w tym zrzuty ekranu i logi, co wspiera analizę i diagnostykę problemów.

Następna zależność to Jackson Databind, biblioteka w wersji 2.15.2, służąca do obsługi danych w formacie JSON. Jackson Databind umożliwia łatwe odczytywanie, przetwarzanie i serializację danych testowych, co wspiera proces parametryzacji testów
i zarządzania danymi wejściowymi. Ostatnia zdefiniowana zależność to Awaitility w wersji 4.2.0, biblioteka wspierająca implementację warunków oczekiwania w testach asynchronicznych. Awaitility jest szczególnie przydatna w sytuacjach, gdy wynik operacji zależy od zmiennych czasowych lub złożonych zależności między komponentami aplikacji.

3.3 Implementacja klas bazowych

BaseTest - Klasa odpowiedzialna za konfigurację WebDrivera i ustawienia globalne.

```
package baseTests;

import com.aventstack.extentreports.ExtentReports;
import com.aventstack.extentreports.ExtentTest;
import com.aventstack.extentreports.Status;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeOptions;
import org.openqa.selenium.support.ui.WebDriverWait;
import org.testng.ITestResult;
import org.testng.annotations.*;
import utils.ExtentManager;
import utils.ConfigReader;
import utils.DriverSingleton;
import utils.ScreenshotUtil;

import java.lang.reflect.Method;
import java.time.Duration;

public class TestBase {

    protected WebDriver driver;
    protected ConfigReader configReader;
    protected WebDriverWait wait;

    protected ExtentReports extent;
    protected ExtentTest test;

    @BeforeSuite
    public void setupSuite() {

    }
```

```
    @BeforeClass
    public void setUp() {
        ChromeOptions options = new ChromeOptions();
        configReader = new ConfigReader();
        driver = DriverSingleton.getDriver();
        wait = new WebDriverWait(driver, Duration.ofSeconds(5));
        extent = ExtentManager.getInstance();
    }

    @BeforeMethod
    public void setupMethod(Method method) {
        test = extent.createTest(method.getName());
    }

    @AfterMethod
    public void afterMethod(ITestResult result) {
        if(result.getStatus() == ITestResult.FAILURE){
            test.log(Status.FAIL, "Test Case Failed: " + result.getName());
            test.log(Status.FAIL, "Reason: " + result.getThrowable());

            String screenShotPath = ScreenshotUtil.captureScreenshot(driver, result.getName());
            try{
                test.addScreenCaptureFromPath(screenShotPath);
            }catch(Exception e){
                test.log(Status.FAIL, "Screenshot: " + e.getMessage());
            }

        }else if (result.getStatus() == ITestResult.SUCCESS){
            test.log(Status.PASS, "Test Case Passed: " + result.getName());

            String screenShotPath = ScreenshotUtil.captureScreenshot(driver, result.getName());
            try{
                test.addScreenCaptureFromPath(screenShotPath);
            }catch(Exception e){
                test.log(Status.FAIL, "Screenshot: " + e.getMessage());
            }
        }else if (result.getStatus() == ITestResult.SKIP){
            test.log(Status.SKIP, "Test Case Skipped: " + result.getName());
        }
    }

    @AfterSuite
    public void tearDownSuite() {

        if (extent != null) {
            extent.flush();
        }
        DriverSingleton.quitDriver();
        extent.flush();

    }

    @AfterClass
    public void tearDown() {
     driver.quit();
    }
}
```

Klasa TestBase pełni fundamentalną rolę w architekturze projektu automatyzacji testów, zapewniając centralną konfigurację WebDrivera, obsługę raportowania za pomocą ExtentReports oraz zarządzanie cyklem życia testów. Jest zaprojektowana jako klasa bazowa, z której dziedziczą wszystkie klasy testowe, co pozwala na zachowanie modularności, reużywalności oraz spójności kodu w projekcie.

Główne zmienne instancji zdefiniowane w klasie obejmują driver, który zarządza instancją WebDrivera, configReader, służący do odczytywania ustawień konfiguracyjnych, oraz wait, umożliwiający stosowanie dynamicznych mechanizmów oczekiwania na elementy w aplikacji. Dodatkowo, zmienne extent i test są wykorzystywane do zarządzania raportami generowanymi przez ExtentReports, co pozwala na szczegółowe śledzenie wyników testów.

Metody z adnotacjami TestNG, takie jak @BeforeSuite, @BeforeClass i @BeforeMethod, definiują różne etapy przygotowania środowiska testowego. Metoda @BeforeSuite jest zarezerwowana dla operacji, które muszą zostać wykonane przed rozpoczęciem całego zestawu testów, natomiast @BeforeClass odpowiada za konfigurację specyficzną dla każdej klasy testowej, w tym inicjalizację WebDrivera przy użyciu DriverSingleton, konfigurację WebDriverWait oraz przygotowanie narzędzia ExtentReports. Metoda @BeforeMethod jest wykonywana przed każdą metodą testową i tworzy obiekt ExtentTest, który rejestruje szczegóły dotyczące bieżącego testu, w tym jego nazwę oraz status.

Metoda @AfterMethod jest odpowiedzialna za zarządzanie zdarzeniami po zakończeniu każdego testu. Sprawdza status testu
i rejestruje go w raporcie ExtentReports. W przypadku niepowodzenia testu rejestrowana jest nazwa testu, przyczyna błędu oraz wykonywany jest zrzut ekranu za pomocą klasy ScreenshotUtil. Zrzuty ekranu są następnie dołączane do raportu, co ułatwia diagnostykę problemów. Dla testów zakończonych sukcesem również generowane są zrzuty ekranu, co zwiększa kompletność dokumentacji wyników testowych. W przypadku testów pominiętych rejestrowany jest ich status jako SKIP.

Metody @AfterSuite i @AfterClass zamykają cykl życia testów. @AfterSuite kończy działanie narzędzia ExtentReports i zapewnia zapis ostatecznego raportu. Dodatkowo wywoływana jest metoda DriverSingleton.quitDriver(), która zamyka instancję WebDrivera i zwalnia zasoby systemowe. @AfterClass odpowiada za zamknięcie przeglądarki dla danej klasy testowej.

Integracja z ExtentReports pozwala na generowanie szczegółowych raportów zawierających wyniki testów, zrzuty ekranu oraz szczegóły dotyczące przyczyn błędów. Centralne zarządzanie WebDriverem przez DriverSingleton zapewnia, że w danym czasie istnieje tylko jedna aktywna instancja przeglądarki, co eliminuje potencjalne konflikty i zwiększa stabilność testów.

BasePage - Bazowa klasa dla wszystkich stron, zawiera wspólne metody dla elementów stron:

```
package pages;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.interactions.Actions;
import org.openqa.selenium.support.PageFactory;
import org.openqa.selenium.support.ui.WebDriverWait;

import java.time.Duration;

public class BasePage {
    protected WebDriver driver;
    protected Actions actions;
    protected WebDriverWait wait;


    public BasePage(WebDriver driver) {
        this.driver = driver;
        this.actions = new Actions(driver);
        this.wait = new WebDriverWait(driver, Duration.ofSeconds(3));
        PageFactory.initElements(driver, this);
    }
}
```

Klasa BasePage jest kluczowym elementem architektury automatyzacji testów, zaprojektowanym zgodnie z metodologią Page Object Model (POM). Jej głównym celem jest zapewnienie wspólnych mechanizmów i narzędzi dla wszystkich klas reprezentujących strony aplikacji webowych, co wspiera modularność, reużywalność kodu oraz ułatwia jego utrzymanie. Klasa definiuje trzy podstawowe obiekty: driver, actions oraz wait. Obiekt driver, będący instancją WebDrivera, umożliwia interakcję

DOI: https://doi.org/10.53052/9788367652292.35

z przeglądarką oraz elementami DOM. Actions zapewnia zaawansowane funkcje manipulacji elementami, takie jak najeżdżanie kursorem, przeciąganie i upuszczanie oraz kliknięcia wielokrotne, natomiast wait reprezentuje dynamiczny mechanizm oczekiwania na spełnienie określonych warunków w trakcie działania aplikacji.

Konstruktor klasy BasePage przyjmuje obiekt WebDriver jako parametr, co pozwala na zachowanie spójności i reużywalności konfiguracji w całym projekcie. Konstruktor inicjalizuje obiekty driver, actions oraz wait z domyślnym czasem oczekiwania wynoszącym 3 sekundy, co pozwala na elastyczne radzenie sobie z opóźnieniami w ładowaniu elementów strony. Dodatkowo konstruktor wykorzystuje mechanizm PageFactory.initElements, który automatycznie mapuje elementy DOM na pola w klasach POM. Dzięki temu klasy dziedziczące BasePage mogą bezpośrednio korzystać z gotowych mechanizmów interakcji
z elementami strony.

Zastosowanie klasy BasePage w projekcie umożliwia centralizację wspólnych operacji i mechanizmów, co znacznie redukuje redundancję kodu w klasach dziedziczących. Dzięki temu każda klasa POM może automatycznie korzystać z już zdefiniowanych narzędzi i funkcjonalności, eliminując konieczność ich powielania. Klasa ta może być również rozbudowywana
o dodatkowe metody wspierające interakcje z elementami stron, takie jak sprawdzanie widoczności elementów, przewijanie do określonych sekcji strony czy obsługa wyjątków związanych z interakcjami użytkownika.

Pod względem architektonicznym klasa BasePage wspiera modularność i skalowalność projektu, co jest kluczowe w przypadku dużych i złożonych systemów testowych. Dzięki jej implementacji dodawanie nowych funkcji do istniejących stron czy obsługa zmian w strukturze aplikacji wymaga minimalnych modyfikacji w kodzie, co znacząco obniża koszty utrzymania projektu. W konsekwencji klasa BasePage stanowi fundament efektywnej automatyzacji testów, zapewniając jednocześnie spójność, niezawodność i łatwość zarządzania kodem w złożonych środowiskach testowych.

3.4 Singleton dla WebDrivera

DriverSingleton - Klasa zarządzająca WebDriverem

package utils;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.chrome.ChromeOptions;

```
public class DriverSingleton {
    private static WebDriver driver;
    private static ConfigReader configReader;
    private DriverSingleton() {
    }

    public static WebDriver getDriver() {
        if (driver == null) {
            ChromeOptions options = new ChromeOptions();
            configReader = new ConfigReader();

            if(configReader.isNoDefalutBrowserCheck()){
                options.addArguments("--no-default-browser-check=new");
            }
            if(configReader.isNoFirstRun()){
                options.addArguments("--no-first-run=new");
            }
            if(configReader.isIncognito()){
                options.addArguments("--incognito");
            }
            options.addArguments("--window-size=1920,1080");

            options.addArguments(configReader.getChromeConf());
            driver = new ChromeDriver(options);
```

```
            }
            return driver;
        }

        public static void quitDriver() {
            if (driver != null) {
                driver.quit();
                driver = null;
            }
        }
    }
```

Klasa DriverSingleton implementuje wzorzec projektowy Singleton, który zapewnia istnienie tylko jednej instancji obiektu WebDriver w całym cyklu działania aplikacji testowej. Jest to kluczowe rozwiązanie w automatyzacji testów, ponieważ centralizuje zarządzanie WebDriverem, redukuje redundancję kodu i minimalizuje ryzyko konfliktów wynikających z wielokrotnej inicjalizacji przeglądarki.

Główną funkcją klasy jest metoda getDriver, odpowiedzialna za inicjalizację WebDrivera w przypadku, gdy instancja jeszcze nie istnieje. Proces ten obejmuje konfigurację przeglądarki Google Chrome za pomocą klasy ChromeOptions. Do ChromeOptions dodawane są różne argumenty konfiguracyjne, takie jak --no-default-browser-check, wyłączający monit
o ustawieniu Chrome jako domyślnej przeglądarki, --no-first-run, eliminujący komunikaty powitalne podczas pierwszego uruchomienia, czy --incognito, aktywujący tryb incognito, co pozwala na izolację testów od danych użytkownika. Ponadto, ustawienie rozdzielczości okna przeglądarki na 1920x1080 zapewnia jednolite warunki testowe dla wszystkich przypadków testowych. Dodatkowe opcje mogą być przekazywane za pomocą metody getChromeConf() z klasy ConfigReader, co pozwala na dynamiczne dostosowanie konfiguracji do specyficznych wymagań środowiska testowego.

Metoda quitDriver zamyka instancję WebDrivera za pomocą driver.quit() oraz resetuje jej referencję do null. Takie podejście umożliwia poprawne zwalnianie zasobów przeglądarki i ponowne tworzenie jej instancji, jeśli zajdzie taka potrzeba
w kolejnych testach. Dzięki temu aplikacja testowa pozostaje efektywna, a system nie jest obciążany przez niepotrzebne procesy przeglądarki.

Klasa DriverSingleton integruje się z narzędziem ConfigReader, które umożliwia elastyczne zarządzanie konfiguracją przeglądarki. Na przykład, parametry takie jak tryb incognito czy dodatkowe ustawienia Chrome mogą być zdefiniowane w pliku konfiguracyjnym i automatycznie odczytywane podczas inicjalizacji WebDrivera. Taka modularność ułatwia dostosowanie testów do różnych środowisk, takich jak testowe, stagingowe czy produkcyjne, bez konieczności zmiany kodu źródłowego.

Zastosowanie DriverSingleton w projektach testowych pozwala na centralizację zarządzania WebDriverem, poprawia czytelność kodu oraz zapewnia większą stabilność testów. Klasa ta minimalizuje potencjalne problemy wynikające z wielokrotnego uruchamiania przeglądarki w jednej sesji testowej i wspiera elastyczność konfiguracji, co czyni ją kluczowym elementem architektury automatyzacji testów. Dzięki jej implementacji testy stają się bardziej skalowalne, efektywne i łatwe w utrzymaniu, co jest szczególnie istotne w złożonych i dynamicznych projektach.

3.5 Konfiguracja plików JSON i ich odczyt

ConfigReader - Klasa do odczytywania danych konfiguracyjnych:

```
package utils;

import com.fasterxml.jackson.core.type.TypeReference;
import com.fasterxml.jackson.databind.ObjectMapper;
import org.testng.annotations.DataProvider;

import java.io.File;
import java.io.IOException;
import java.util.List;
import java.util.Map;
```

```
public class JsonDataProvider {

    // Uniwersalna metoda do wczytywania danych z określonego pliku
    public Object[][] getDataFromFile(String fileName) throws IOException {
        ObjectMapper objectMapper = new ObjectMapper();
        File file = new File(fileName);

        // Wczytanie danych z pliku JSON jako lista map
        List<Map<String, String>> data = objectMapper.readValue(file, new
TypeReference<List<Map<String, String>>>() {});

        // Konwersja danych do formatu Object[][]
        Object[][] testData = new Object[data.size()][1];
        for (int i = 0; i < data.size(); i++) {
            testData[i][0] = data.get(i);
        }

        return testData;
    }

    // Przykładowe DataProvider dla LoginPageTest
    @DataProvider(name = "testLoginData")
    public Object[][] getLoginData() throws IOException {
        return getDataFromFile("src/test/testData/loginPageData.json");
    }
```

Klasa JsonDataProvider pełni funkcję narzędzia umożliwiającego dynamiczne odczytywanie danych testowych z plików JSON
i ich dostarczanie do testów w formacie zgodnym z wymaganiami frameworka TestNG. Jest to przykład elastycznego podejścia do parametryzacji testów, które pozwala na oddzielenie danych testowych od logiki testów, co zwiększa modularność oraz reużywalność kodu.

Główną metodą w tej klasie jest getDataFromFile, która przyjmuje jako parametr nazwę pliku JSON i zwraca dane w formacie Object[][]. Wykorzystuje ona bibliotekę Jackson (ObjectMapper) do odczytu i deserializacji zawartości pliku JSON do listy map (List<Map<String, String>>). Taka struktura umożliwia przechowywanie dowolnej liczby klucz-wartość w jednym elemencie, co jest szczególnie przydatne w złożonych scenariuszach testowych.

- Proces odczytu danych obejmuje następujące kroki:
- Zainicjalizowanie obiektu ObjectMapper oraz odczytanie zawartości pliku JSON.
- Przekształcenie danych z JSON do listy map, gdzie każda mapa odpowiada jednemu zestawowi danych testowych.
- Konwersję listy map do dwuwymiarowej tablicy obiektów Object[][], wymaganej przez TestNG. Każdy wiersz tablicy reprezentuje jeden zestaw danych testowych, co pozwala na ich iteracyjne przetwarzanie
- w ramach testów.

Metoda getLoginData, oznaczona adnotacją @DataProvider, jest dedykowanym dostawcą danych dla testów związanych ze stroną logowania. Adnotacja @DataProvider w TestNG umożliwia parametryzację testów poprzez dostarczenie wielu zestawów danych. Metoda ta odwołuje się do getDataFromFile, podając ścieżkę do pliku JSON zawierającego dane testowe dla strony logowania ("src/test/testData/loginPageData.json"). Dzięki temu dane są dynamicznie wczytywane podczas uruchamiania testów, co eliminuje potrzebę ręcznego definiowania danych w kodzie.

3.6 Pliki JSON dla danych testowych

Dane testowe są przechowywane w plikach JSON, np. loginPage.json

```
[
  {
    "username": "JanTestowy",
    "password": "TestoweHasloZakodowane",
    "email": "testowyEmail@TestowyEmail.pl"
  }
```

```
]
```

JSON reprezentuje zestaw danych testowych, który może być wykorzystywany w procesie automatyzacji testów aplikacji, takich jak testy logowania, rejestracji czy innych operacji związanych z uwierzytelnianiem użytkownika. JSON, będący popularnym formatem wymiany danych, jest szeroko stosowany w automatyzacji testów ze względu na swoją prostotę, czytelność oraz kompatybilność z różnymi narzędziami i językami programowania.

Struktura danych w przedstawionym fragmencie składa się z jednej encji zawartej w tablicy, co pozwala na łatwe rozszerzenie zestawu o dodatkowe przypadki testowe w przyszłości. Każda encja jest obiektem zawierającym trzy klucz-wartości:
username: Przechowuje nazwę użytkownika testowego, w tym przypadku "JanTestowy", co umożliwia symulację operacji wykonywanych przez użytkownika w systemie.

password: Zawiera hasło użytkownika w postaci zakodowanej lub jawnej. Wartość "TestoweHasloZakodowane" stanowi dane testowe wykorzystywane w procesach uwierzytelniania lub weryfikacji poprawności obsługi haseł.

email: Przechowuje adres e-mail powiązany z kontem użytkownika, tutaj "testowyEmail@TestowyEmail.pl". Pole to może być używane w testach walidacji formatów adresów e-mail, odzyskiwania hasła czy weryfikacji unikalności konta.

Dane te mogą być dynamicznie wczytywane do skryptów testowych, np. w Selenium lub TestNG, przy użyciu bibliotek do obsługi JSON, takich jak Jackson w języku Java. Dzięki temu możliwe jest oddzielenie logiki testów od danych testowych, co poprawia modularność i czytelność kodu oraz ułatwia jego utrzymanie.

JSON wspiera integrację z nowoczesnymi narzędziami do automatyzacji testów, takimi jak frameworki TestNG czy JUnit. Pozwala także na dynamiczne parametryzowanie testów, dzięki czemu zestawy testowe mogą być łatwo skalowane. Dane zapisane w pliku JSON mogą być wykorzystywane w scenariuszach testowych, takich jak weryfikacja poprawności logowania czy symulacja działań użytkowników w różnych warunkach testowych.

Zastosowanie tego fragmentu JSON w procesie testowania przyczynia się do zwiększenia elastyczności, reużywalności oraz efektywności scenariuszy testowych, jednocześnie zapewniając łatwość modyfikacji i rozszerzania danych testowych w projekcie.

3.7 Budowa przypadku testowego – fragment

Poniżej opisano fragmenty kodu dla utworzenia przypadku testowego dla logowani a się do aplikacji

3.7.1 Plik LoginBase z katalogu BaseTest

```
package baseTests;
import org.openqa.selenium.support.PageFactory;
import org.testng.annotations.BeforeClass;
import pages.LoginPage;

public class LoginBase extends TestBase {
    protected LoginPage loginPage;

    @BeforeClass
    public void setUp() {
        super.setUp();
        driver.get(configReader.getAccountLoginURL());
        loginPage = PageFactory.initElements(driver, LoginPage.class);
    }

}
```

Powyższy kod przedstawia implementację klasy bazowej LoginBase, która jest częścią architektury testowej opartej na frameworkach Selenium i TestNG. Klasa ta rozszerza TestBase, dziedzicząc jej konfigurację oraz metody wspierające,

co pozwala na ujednolicenie i ponowne wykorzystanie podstawowych operacji testowych w projekcie. Głównym zadaniem klasy LoginBase jest zapewnienie specyficznej konfiguracji środowiska testowego dla scenariuszy związanych z testowaniem strony logowania.
W metodzie setUp, oznaczonej adnotacją @BeforeClass, realizowana jest konfiguracja przed rozpoczęciem dowolnego przypadku testowego w klasach dziedziczących. Dzięki zastosowaniu adnotacji @BeforeClass, metoda ta jest wykonywana tylko raz przed wszystkimi testami w danej klasie testowej, co zwiększa efektywność i zmniejsza obciążenie środowiska testowego.

Metoda setUp wywołuje metodę o tej samej nazwie z klasy nadrzędnej (super.setUp()), co pozwala na inicjalizację wspólnych zasobów i ustawień skonfigurowanych w TestBase. Następnie otwierana jest strona logowania za pomocą driver.get(configReader.getAccountLoginURL()), gdzie configReader dostarcza dynamiczny URL na podstawie pliku konfiguracyjnego, co umożliwia łatwe dostosowanie środowiska testowego do różnych scenariuszy i środowisk (np. testowych, stagingowych czy produkcyjnych).

Kolejnym krokiem jest inicjalizacja obiektu LoginPage za pomocą wzorca PageFactory. Metoda PageFactory.initElements(driver, LoginPage.class) wiąże elementy strony logowania z ich odpowiednikami w klasie LoginPage, umożliwiając prostą i intuicyjną interakcję z elementami strony w testach. Dzięki zastosowaniu tej techniki, wszelkie zmiany
w strukturze strony wymagają jedynie aktualizacji klasy Page Object, co znacznie upraszcza proces utrzymania testów.

Klasa LoginBase odgrywa kluczową rolę w modularnej architekturze projektu, ułatwiając testowanie funkcji związanych
z logowaniem. Jej zastosowanie zgodne z metodologią Page Object Model (POM) oraz wzorcem hierarchii klas bazowych wspiera czytelność i skalowalność projektu testowego. Struktura ta pozwala na łatwe rozszerzenie funkcjonalności oraz efektywne zarządzanie testami, minimalizując powtarzalność kodu i zwiększając jego reużywalność. W konsekwencji, takie podejście przyczynia się do optymalizacji procesu automatyzacji testów oraz zmniejszenia kosztów utrzymania projektu.

3.7.2 Plik LoginTest z katalogu Tests

```
package tests;

import baseTests.LoginBase;
import com.aventstack.extentreports.Status;
import org.openqa.selenium.By;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.ui.ExpectedConditions;
import org.testng.Assert;
import org.testng.annotations.Test;
import utils.JsonDataProvider;
import java.util.Map;


public class LoginPageTest extends LoginBase {

    // Przypadki testowe z krokami testowymi
    @Test(dataProvider = "testLoginData", dataProviderClass = JsonDataProvider.class)
    public void loginTest(Map<String, String> testData){
        String username = testData.get("username");
        String password = testData.get("password");

        test.log(Status.INFO, "Login Test");
        loginPage.sendUserNameField(username);
        loginPage.sendUserPasswordField(password);
        loginPage.clickSubmitLoginButton();

    }
```

Kod przedstawia implementację testu automatycznego dla strony logowania, zrealizowanego za pomocą frameworków Selenium i TestNG, zgodnie z metodologią Page Object Model (POM). Klasa LoginPageTest znajduje się w pakiecie tests

i rozszerza klasę bazową LoginBase, co pozwala na dziedziczenie wspólnych metod oraz konfiguracji testowej, takich jak inicjalizacja WebDrivera czy dostęp do obiektów POM. Struktura ta wspiera modularność i przejrzystość kodu, ułatwiając zarządzanie dużymi projektami testowymi.

Metoda loginTest jest oznaczona adnotacją @Test i parametryzowana przy użyciu dataProvider. Dzięki zastosowaniu klasy JsonDataProvider dane testowe są dostarczane w formie mapy, co umożliwia dynamiczne definiowanie scenariuszy testowych bez konieczności modyfikowania kodu testowego. Takie podejście zwiększa elastyczność i pozwala na łatwe dodawanie nowych przypadków testowych poprzez modyfikację plików JSON.

Test wykonuje podstawowe kroki logowania, takie jak wprowadzenie nazwy użytkownika i hasła oraz kliknięcie przycisku logowania. Interakcje te są realizowane za pomocą metod dostępnych w obiekcie loginPage, który implementuje wzorzec Page Object Model. Dzięki temu zmiany w strukturze strony wymagają jedynie aktualizacji klasy POM, bez konieczności ingerencji
w logikę testów, co znacząco upraszcza utrzymanie projektu.

Zintegrowanie z ExtentReports pozwala na rejestrowanie szczegółowych informacji o przebiegu testów. W metodzie loginTest używana jest funkcja test.log(Status.INFO, "Login Test"), która zapisuje informacje o rozpoczęciu testu do raportu. Dodatkowo, mechanizm realizacji zrzutów ekranu zaimplementowany w klasie bazowej BaseTest umożliwia automatyczne rejestrowanie stanu aplikacji w trakcie wykonania testu lub w momencie wystąpienia błędów. Taki mechanizm zwiększa przejrzystość raportów
i ułatwia analizę problemów, zwłaszcza w sytuacjach, gdy test zakończy się niepowodzeniem.

3.7.3 Wywołanie przypadku testowego

Wywołanie przypadku testowego realizowane jest przez odpowiedni plik xml:

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">

<suite name="Login Page test">
    <test name="Login Page">
        <classes>
            <class name="tests.LoginPageTest">
                <methods>
                    <include name="loginTestAndLogout"/>
                    <include name="loginTestRememberMe"/>
                </methods>
            </class>
        </classes>
    </test>
</suite>
```

Plik XML dla TestNG przedstawia podstawową konfigurację zestawu testów oraz wybranych scenariuszy, umożliwiając precyzyjne zarządzanie procesem automatyzacji w projektach testowych. Dokument rozpoczyna się od deklaracji wersji XML oraz kodowania UTF-8, co stanowi standard w tego typu plikach. Następnie, za pomocą deklaracji typu dokumentu (DTD - Document Type Definition), określa się zgodność pliku z wersją 1.0 specyfikacji TestNG, co zapewnia poprawne jego interpretowanie przez framework.

Sekcja <suite> definiuje zestaw testów, którego nazwa została określona jako "Login Page test". Jest to logiczna jednostka grupująca powiązane testy i pozwalająca na uruchamianie ich jako całości. Wewnątrz zestawu znajduje się sekcja <test>, opisująca pojedynczy test o nazwie "Login Page". W ramach tego testu zdefiniowano jedną klasę testową za pomocą elementu <class>, wskazując na pełną kwalifikowaną nazwę klasy tests.LoginPageTest, co jednoznacznie identyfikuje jej lokalizację w strukturze projektu.

Sekcja <methods> w ramach klasy umożliwia precyzyjne określenie, które metody testowe powinny zostać uruchomione. W tym przypadku wybrano dwie konkretne metody: loginTestAndLogout oraz loginTestRememberMe.

DOI: https://doi.org/10.53052/9788367652292.35

Dzięki temu możliwe jest uruchamianie tylko wybranych scenariuszy testowych, co znacząco poprawia efektywność procesu testowania, szczególnie
w projektach o dużej skali.

Taka konfiguracja charakteryzuje się wysokim poziomem modularności, pozwalając na precyzyjne grupowanie
i zarządzanie testami. Zastosowanie filtrów metod za pomocą elementów <include> umożliwia kontrolowanie zakresu uruchamianych testów, co jest szczególnie istotne w sytuacjach, gdy nie wszystkie scenariusze muszą być testowane podczas każdego cyklu.

Plik XML dla TestNG doskonale integruje się z narzędziami CI/CD, takimi jak Jenkins, GitLab CI/CD czy GitHub Actions, umożliwiając dynamiczne i zautomatyzowane uruchamianie testów w procesie ciągłej integracji. Możliwość dodawania parametrów w sekcji <parameter> zwiększa elastyczność testów, pozwalając na dynamiczne przekazywanie wartości konfiguracyjnych. Ponadto, istnieje opcja użycia elementów <exclude> w celu pomijania wybranych metod, co dodatkowo zwiększa kontrolę nad wykonywanymi testami.

3.8 Korzyści z POM i konfiguracji

Modularność i łatwość utrzymania: Page Object Model pozwala oddzielić logikę testów od elementów strony, co ułatwia utrzymanie kodu przy zmianach w interfejsie.

- Czytelność: Jasna struktura projektu oraz wykorzystanie narzędzi raportowania (ExtentReports, Allure) poprawiają czytelność wyników testów.
- Skalowalność: Singleton dla WebDrivera i zarządzanie danymi JSON pozwalają na łatwe dodawanie nowych testów oraz obsługę różnych środowisk testowych.

Przedstawione rozwiązanie jest kompleksowym przykładem referencyjnym dla projektów automatyzacji testów aplikacji webowych, integrującym kluczowe narzędzia i technologie w celu stworzenia skalowalnego, modularnego oraz efektywnego środowiska testowego. Stanowi ono fundament, na którym można budować złożone procesy testowe, uwzględniając dynamicznie zmieniające się potrzeby projektowe oraz wymagania biznesowe. Rozwiązanie obejmuje zaawansowaną konfigurację zależności, implementację wzorców projektowych (takich jak Singleton oraz Page Object Model), dynamiczne zarządzanie danymi testowymi, a także generowanie przejrzystych i szczegółowych raportów testowych. Taka struktura nie tylko zwiększa efektywność procesu testowania, ale również minimalizuje ryzyko błędów oraz upraszcza utrzymanie projektu w dłuższej perspektywie.

Kluczowym elementem tego rozwiązania jest modularność, która pozwala na oddzielenie logiki testowej od interakcji z elementami aplikacji, co znacząco zwiększa czytelność, reużywalność i łatwość utrzymania kodu testowego. Dzięki zastosowaniu wzorca projektowego Page Object Model (POM), każda strona aplikacji jest reprezentowana przez dedykowaną klasę, która przechowuje elementy strony oraz metody obsługujące interakcje z nimi. Takie podejście minimalizuje redundancję kodu, umożliwiając centralne zarządzanie logiką interfejsu użytkownika. W przypadku zmian w strukturze aplikacji modyfikacje ograniczają się wyłącznie do odpowiednich klas reprezentujących zmienione strony, co znacząco redukuje czas i koszty utrzymania testów oraz minimalizuje ryzyko wprowadzenia nowych błędów.

Centralizacja konfiguracji WebDrivera za pomocą wzorca Singleton dodatkowo wzmacnia efektywność i stabilność środowiska testowego. Singleton zapewnia, że w całej sesji testowej istnieje tylko jedna instancja WebDrivera, eliminując problemy związane z wielokrotnym inicjowaniem przeglądarki. Dzięki temu zasoby systemowe są lepiej wykorzystywane, a ewentualne konflikty wynikające z konkurencyjnych instancji WebDrivera są całkowicie wyeliminowane. To rozwiązanie jest szczególnie korzystne
w środowiskach, gdzie testy są uruchamiane równolegle lub w ramach złożonych procesów CI/CD, ponieważ zapewnia spójność
i poprawność działania nawet w dynamicznych warunkach.

Architektura oparta na modularności wspiera również optymalizację zasobów poprzez lepsze zarządzanie czasem wykonania testów oraz ich niezależnością. Struktura testów staje się bardziej skalowalna, co pozwala na łatwe dodawanie nowych funkcjonalności bez konieczności przepisania istniejącego kodu. Dzięki rozdzieleniu odpowiedzialności za logikę testową, interakcje z elementami aplikacji oraz konfigurację środowiska testowego,

zespoły mogą pracować równolegle nad różnymi aspektami projektu, co zwiększa produktywność i skraca czas wdrożenia nowych testów.

Dodatkowo, podejście to wspiera implementację zaawansowanych technik testowych, takich jak dynamiczne generowanie danych testowych czy automatyczne dostosowywanie testów do zmian w aplikacji. Możliwość łatwego rozszerzania funkcjonalności
w ramach istniejącej struktury kodu sprawia, że rozwiązanie jest szczególnie przydatne w dynamicznych środowiskach Agile, gdzie wymagania i funkcje aplikacji często się zmieniają. Centralizacja konfiguracji i modularność kodu ułatwiają integrację
z narzędziami do raportowania, takimi jak ExtentReports, oraz z zaawansowanymi mechanizmami oczekiwania, które zwiększają niezawodność testów w dynamicznych aplikacjach opartych na technologiach takich jak React czy Angular.

Dynamiczne zarządzanie danymi testowymi, oparte na integracji z biblioteką Jackson Databind, umożliwia odczyt danych z plików JSON, co znacząco ułatwia parametryzację testów oraz dostosowanie ich do różnych scenariuszy użytkowania. Takie podejście wspiera testowanie funkcji w wielu wariantach, pozwalając na symulację zróżnicowanych przypadków testowych bez potrzeby wprowadzania zmian w kodzie testowym. Ponadto generowanie raportów testowych za pomocą ExtentReports pozwala na szczegółowe śledzenie wyników testów, dostarczając informacji o przebiegu testów, przyczynach błędów oraz wizualizując wyniki za pomocą zrzutów ekranu.

Włączenie sztucznej inteligencji (AI) do środowiska testowego otwiera zupełnie nowe perspektywy, które znacząco zmieniają tradycyjne podejście do automatyzacji testów i optymalizacji całego procesu testowego. AI w testowaniu oprogramowania przekształca czasochłonne, manualne zadania w zautomatyzowane, inteligentne operacje, które nie tylko zwiększają efektywność, ale również pozwalają na bardziej kompleksowe testowanie w krótszym czasie. Narzędzia oparte na sztucznej inteligencji mogą być wykorzystywane w różnych aspektach cyklu życia testów, takich jak generowanie kodu, automatyczne tworzenie danych testowych, analiza wyników testów, identyfikacja potencjalnych problemów, a nawet przewidywanie awarii w przyszłych wdrożeniach.

Generowanie skryptów testowych za pomocą AI, takich jak ChatGPT, radykalnie przyspiesza proces implementacji testów. AI może automatycznie generować fragmenty kodu oparte na wzorcach projektowych, takich jak Page Object Model, dostarczając gotowe klasy POM, które są dostosowane do specyficznych struktur aplikacji. Dzięki temu zespół testerski może skupić się na złożonych aspektach testowania, podczas gdy powtarzalne i standardowe komponenty są tworzone automatycznie. Ponadto, wykorzystanie AI w generowaniu kodu zmniejsza ryzyko błędów ludzkich, takich jak literówki czy niewłaściwa implementacja wzorców, co przekłada się na większą niezawodność i spójność skryptów testowych.

W zakresie tworzenia danych testowych AI odgrywa kluczową rolę, dostarczając dynamicznie generowane, realistyczne dane wejściowe dostosowane do różnych scenariuszy testowych. Algorytmy AI mogą tworzyć zestawy danych, które uwzględniają nietypowe przypadki brzegowe, potencjalne nadużycia systemu, a także dane zgodne z regulacjami prawnymi, takimi jak RODO. Automatyzacja tego procesu pozwala na szybkie dostosowanie danych testowych do zmieniających się wymagań biznesowych i technicznych bez potrzeby ręcznej edycji.

Analiza wyników testów jest kolejnym obszarem, w którym sztuczna inteligencja wprowadza istotne innowacje. Algorytmy AI mogą automatycznie identyfikować wzorce w wynikach testów, takie jak często powtarzające się błędy czy problemy wydajnościowe. Mogą także grupować błędy na podstawie ich podobieństw i wskazywać priorytety ich rozwiązania, co wspiera efektywne zarządzanie procesem naprawy usterek. W bardziej zaawansowanych zastosowaniach AI może przewidywać potencjalne awarie na podstawie analizy historycznych danych testowych i sugerować proaktywne działania, które zapobiegną problemom
w przyszłości.

Jednym z najbardziej obiecujących zastosowań AI w testowaniu jest jej zdolność do automatycznej adaptacji testów w przypadku zmian w aplikacji. Algorytmy AI mogą analizować strukturę aplikacji, wykrywać zmiany w modelu DOM oraz automatycznie aktualizować klasy Page Object Model, co znacząco zmniejsza koszty i czas utrzymania testów. Dzięki temu proces testowania staje się bardziej odporny na zmiany w aplikacji, a zespół testerski może skupić się na testach o wyższej wartości biznesowej.

Wpływ tego rozwiązania na środowisko testowe jest znaczący, zarówno pod względem efektywności, jak i elastyczności. Jego implementacja pozwala na stworzenie systemu, który jest łatwy w adaptacji do nowych wymagań, wspiera procesy CI/CD oraz umożliwia szybkie dostosowanie do zmieniających się potrzeb biznesowych. W rezultacie przedstawiony model jest idealnym rozwiązaniem dla dynamicznych środowisk Agile i DevOps, które wymagają ciągłej integracji, częstych aktualizacji oraz wysokiej jakości dostarczanego oprogramowania. Dzięki połączeniu tradycyjnych technik automatyzacji z możliwościami oferowanymi przez AI, środowisko testowe staje się bardziej zaawansowane, niezawodne i gotowe na przyszłe wyzwania technologiczne.

Wdrożenie omawianego rozwiązania w środowisku produkcyjnym ma istotny wpływ na jakość, efektywność i skalowalność procesu testowania. Automatyzacja testów, oparta na wzorcach projektowych takich jak Page Object Model oraz Singleton, pozwala na stworzenie stabilnego, modularnego i łatwego w utrzymaniu środowiska testowego, które spełnia wymagania współczesnych aplikacji webowych. Integracja z narzędziami do raportowania, dynamicznego zarządzania danymi oraz pipeline'ami CI/CD wspiera realizację projektów o wysokiej złożoności, jednocześnie umożliwiając szybkie dostosowanie do zmieniających się wymagań.

Rozwiązanie to przynosi wiele korzyści. Automatyzacja znacząco skraca czas wykonania regresyjnych i funkcjonalnych testów, które mogą być uruchamiane równolegle na różnych środowiskach i konfiguracjach. Dzięki temu możliwe jest częstsze wydawanie nowych wersji oprogramowania, co zwiększa konkurencyjność projektu. Modularność architektury pozwala na łatwe dodawanie nowych scenariuszy testowych i przypadków brzegowych, co poprawia wykrywalność błędów i minimalizuje ryzyko ich pojawienia się w środowisku produkcyjnym. Centralizacja konfiguracji WebDrivera eliminuje problemy związane z wielokrotnym inicjowaniem instancji przeglądarki, co zwiększa stabilność testów i poprawia wykorzystanie zasobów. Dodatkowo, zaawansowane raporty testowe generowane przez narzędzia takie jak ExtentReports dostarczają szczegółowych informacji o wynikach testów, w tym o błędach i zrzutach ekranu, co wspiera analizę oraz szybką identyfikację problemów.

Jednak wdrożenie automatyzacji niesie również wyzwania. Początkowe koszty są wysokie, obejmując inwestycje w narzędzia, szkolenia zespołu i opracowanie infrastruktury testowej. Wymaga to szczególnie dużych nakładów pracy w przypadku projektów o ograniczonym zakresie lub krótkim cyklu życia, gdzie zwrot z inwestycji może być niewielki. Dodatkowo, dynamiczne zmiany w aplikacji mogą wymagać częstego dostosowywania skryptów testowych, co zwiększa koszty utrzymania. Automatyzacja nie zastępuje również całkowicie testów manualnych, szczególnie w obszarach wymagających oceny interfejsu użytkownika czy doświadczenia końcowego użytkownika.

Z perspektywy stopy zwrotu inwestycji (ROI), wdrożenie automatyzacji testów przynosi wysokie korzyści w projektach długoterminowych i środowiskach CI/CD. W projektach o długim cyklu życia koszty początkowe rozkładają się na wiele iteracji, a redukcja błędów w środowisku produkcyjnym obniża koszty naprawy i potencjalne straty finansowe. W dynamicznych środowiskach CI/CD automatyzacja przyspiesza wydawanie nowych wersji, minimalizując ryzyko wystąpienia błędów dzięki ciągłej kontroli jakości. Jednak w projektach o małym zakresie ROI może być niższe, ponieważ koszty wdrożenia automatyzacji mogą przewyższać oszczędności wynikające z jej zastosowania.

Podziękowanie
Serdeczne podziękowania kieruję do dr. inż. Sławomira Hermy za nieocenione wsparcie merytoryczne i cenne wskazówki dotyczące inżynieryjnego podejścia do zagadnień, które znacząco przyczyniły się do realizacji niniejszego projektu. Jego wiedza i doświadczenie stanowiły istotną inspirację podczas przygotowywania oraz doskonalenia struktur testów automatycznych, co przełożyło się na jakość i skuteczność opracowanych rozwiązań.

Wyrazy wdzięczności składam również firmie Quality Island za udostępnienie zaawansowanych platform testowych, które stały się podstawą badań nad opracowaniem optymalnych struktur testów automatycznych. Dzięki wsparciu firmy możliwe było przygotowanie modułowych i uniwersalnych rozwiązań, które znajdują skuteczne zastosowanie zarówno w środowiskach produkcyjnych u licznych klientów, jak i w procesach szkoleniowych. Współpraca ta stanowiła kluczowy element osiągnięcia rezultatów, które z powodzeniem wspierają rozwój automatyzacji testów w różnych kontekstach biznesowych.

Dziękuję za współpracę i zaufanie, które pozwoliły na realizację tak innowacyjnego i wartościowego projektu.

Literatura

1. Fowler, M., Highsmith, J., The Agile Manifesto. IEEE Computer Society, 2001.
2. Gamba, A., et al., Selenium Framework Design in Data-Driven Testing. Springer, 2019.
3. IEEE Software, AI in Software Testing: Current Applications and Future Trends, 2023.
4. Cucumber Team, Behavior-Driven Development with Cucumber. O'Reilly Media, 2020.
5. Selenium Documentation: https://www.selenium.dev/documentation/
6. TestNG Documentation: https://testng.org/doc/
7. "Effective UI Testing with Selenium and TestNG," IEEE Software, 2023.