Narzędzie wspierające tworzenie testów dla aplikacji webowych

Łukasz Kowalewski
Promotor: dr inż. Marcin Adamski

Spis treści

Rozdział 1

Wstęp

W niniejszym wstępie zostanie uzasadniona istotność tematu pracy inżynierskiej. Tytuł pracy – *Narzędzie wspierające tworzenie testów dla aplikacji webowych* – wskazuje główną funkcję tworzonego w jej ramach programu. Jego rolą jest dostarczenie użytkownikom narzędzia pozwalającego przygotowywać oprogramowanie i testy w krótszym czasie przy zachowaniu wysokiej jakości. W kolejnych rozdziałach omówione zostaną motywacje, problematyka zagadnienia oraz możliwe rozwiązania. Aby zrozumieć powody podjęcia takiego tematu, należy wykazać potrzebę tworzenia oprogramowania wspierającego proces testowania.

Testy stanowią kluczowy element zaawansowanych i dojrzałych systemów informatycznych oraz aplikacji. W dużych korporacjach istnieją specjalne działy zajmujące się wyłącznie tworzeniem i utrzymywaniem testów, a sam rynek pracy sygnalizuje wysoki popyt na programistów wyspecjalizowanych w tym obszarze. W takich organizacjach powstają zaawansowane narzędzia do automatyzacji testów, co jest zrozumiałe w kontekście ogromnych strat finansowych, jakie mogą wynikać z błędów w krytycznym oprogramowaniu.

Odmiennie sytuacja bywa postrzegana w małych i średnich projektach, gdzie często brakuje dostatecznej motywacji (np. w postaci wysokich strat finansowych), by inwestować czas w rozbudowaną automatyzację testów. Celem tworzonej aplikacji jest ułatwienie generowania prostych testów w krótkim czasie, tak aby również mniejsze przedsięwzięcia mogły czerpać korzyści z automatyzacji, osiągając przy tym wyższe pokrycie testowe niewielkim nakładem pracy.

1.1 Cel i zakres pracy

Celem niniejszej pracy inżynierskiej jest opracowanie narzędzia, które znacząco usprawni proces tworzenia testów automatycznych w projektach webowych. Praca skupia się nie tylko na samej implementacji aplikacji, lecz także na analizie metod i dobrych praktyk w zakresie testowania.

Tworzone oprogramowanie ma rozwiązać przede wszystkim następujące problemy:

- **Wysoki próg wejścia** w automatyzację testów dla początkujących zespołów – narzędzie powinno dostarczyć przyjazne mechanizmy generowania przykładowych skryptów i scenariuszy testowych.
- **Długi czas przygotowywania testów** w małych i średnich projektach – planuje się zapewnić funkcje przyspieszające proces konfiguracji i pisania testów (np.

wstępne generowanie kodu testowego).

- **Integracja z istniejącymi technologiami** – narzędzie będzie wykorzystywać znane biblioteki (takie jak Selenium czy Playwright) oraz zapewni łatwą integrację z popularnymi pipeline'ami CI/CD.

Oczekiwanym rezultatem jest działające oprogramowanie, którego zastosowanie umożliwi efektywne tworzenie testów automatycznych oraz ich integrację z cyklem wytwarzania oprogramowania. Dokumentacja pracy przybliży zarówno aspekty teoretyczne (przegląd frameworków, strategii testowania), jak i praktyczne (omówienie kluczowych fragmentów kodu, przykłady uruchomienia testów oraz wdrożenia w środowisku CI/CD).

Zakres opracowania obejmuje:

- Analizę dostępnych narzędzi i bibliotek testowych dla aplikacji webowych,
- Zaprojektowanie i implementację modularnego narzędzia generującego testy,
- Przedstawienie możliwych sposobów dalszego rozwoju aplikacji, w tym plan rozbudowy funkcjonalności oraz integracji z innymi platformami.

1.2 Struktura pracy

W kolejnych rozdziałach przedstawiono najważniejsze aspekty projektowania i tworzenia narzędzia do wspierania testów automatycznych:

- **Rozdział 2** omawia przykłady zastosowań testów automatycznych, przedstawiając ich różnorodne formy (testy jednostkowe, integracyjne i end-to-end).
- **Rozdział 3** zawiera przegląd popularnych technologii i bibliotek wspierających testowanie aplikacji webowych, w tym frameworki testowe, narzędzia typu record-and-play czy rozwiązania CI/CD.
- **Rozdział 4** dotyczy strategii generowania testów, począwszy od podejść opartych na rejestrowaniu akcji, aż po wykorzystanie modeli aplikacji.
- **Rozdział 5** opisuje architekturę i interfejs projektowanej aplikacji, omawiając poszczególne moduły i sposób ich wzajemnej komunikacji.
- **Rozdział 6** poświęcony jest kluczowym elementom kodu źródłowego i zastosowanym bibliotekom.
- **Rozdział 7** prezentuje praktyczne przykłady użycia aplikacji w trybie interaktywnym oraz zintegrowanym w potoku CI/CD.
- **Rozdział 8** stanowi podsumowanie pracy, omawia wnioski końcowe i możliwe kierunki rozwoju narzędzia.

Rozdział 2

Przykłady zastosowań testów

Współczesne aplikacje webowe – niezależnie od skali – wymagają odpowiedniego poziomu kontroli jakości. Testy automatyczne są jednym z najbardziej efektywnych sposobów osiągnięcia tego celu. W niniejszym rozdziale omówiono podstawowe rodzaje testów stosowanych przy rozwoju oprogramowania, w szczególności aplikacji internetowych. Zwrócono przy tym uwagę zarówno na testy jednostkowe, integracyjne, jak i end-to-end (E2E). W bardziej złożonych projektach popularne jest podejście wielopoziomowe, pozwalające uniknąć błędów na różnych etapach tworzenia oprogramowania.

Testy automatyczne pełnią kluczową rolę w procesie ciągłej integracji i dostarczania (CI/CD). Ich cykliczne uruchamianie przed wdrożeniem minimalizuje ryzyko wprowadzenia wadliwych zmian. W dalszej części rozdziału zaprezentowano najistotniejsze cechy trzech podstawowych poziomów testów oraz omówiono korzyści i wyzwania wynikające z ich stosowania.

2.1 Testy jednostkowe

Testy jednostkowe (*unit tests*) dotyczą najmniejszych elementów oprogramowania, zazwyczaj pojedynczych funkcji czy metod. Ich celem jest weryfikacja poprawności działania konkretnego fragmentu kodu w oderwaniu od reszty systemu. Dzięki temu programiści mogą szybko wykrywać regresje w przypadku wprowadzania nowych funkcjonalności bądź zmian.

Charakterystyka i zalety testów jednostkowych

- **Wczesne wykrywanie błędów:** mały zakres testowanego kodu umożliwia łatwą diagnozę przyczyn problemów.
- **Szybkie uruchamianie:** testy jednostkowe są przeważnie mało zasobożerne, można je więc wykonywać nawet przy każdej kompilacji.
- **Wspieranie refaktoryzacji:** dobrze napisane testy jednostkowe pełnią rolę siatki bezpieczeństwa przy wprowadzaniu modyfikacji w kodzie.

Miejsce w cyklu życia aplikacji

Testy jednostkowe powstają zazwyczaj wraz z implementacją nowych funkcji, często w podejściu *Test-Driven Development* (TDD). Nawet w projektach niestosujących formalnie

TDD, testy jednostkowe są pisane równolegle bądź krótko po wprowadzeniu kluczowych metod. Dzięki temu deweloperzy na bieżąco weryfikują jakość kodu.

Znaczenie dla jakości oprogramowania

Choć testy jednostkowe nie wykrywają wszystkich możliwych błędów (w szczególności tych związanych z integracją czy kompleksową logiką biznesową), to stanowią podstawę solidnego procesu testowania. Ułatwiają utrzymanie wysokiej jakości kodu przez cały okres rozwoju aplikacji, zmniejszając liczbę nieoczekiwanych problemów w krytycznych częściach systemu.

2.2 Testy integracyjne

Testy integracyjne (*integration tests*) weryfikują poprawność współpracy pomiędzy różnymi komponentami systemu. W przeciwieństwie do testów jednostkowych, koncentrujących się na pojedynczych funkcjach, testy integracyjne sprawdzają, czy moduły wchodzące w skład aplikacji działają razem w sposób spójny i przewidywalny.

Główny cel i zakres

Podstawowym zadaniem testów integracyjnych jest upewnienie się, że wszystkie elementy systemu (np. warstwa serwerowa, baza danych, usługi zewnętrzne) współpracują zgodnie z oczekiwaniami. W aplikacjach webowych mogą obejmować m.in. testowanie komunikacji serwera z bazą, przepływ danych między mikrousługami czy integrację z API firm trzecich.

Przykłady zastosowań w aplikacjach webowych

- **Weryfikacja API i bazy danych:** testy integracyjne sprawdzają, czy żądania HTTP wysyłane przez front-end są prawidłowo obsługiwane w warstwie serwerowej oraz czy zwracane przez bazę dane są poprawne.
- **Integracje zewnętrzne:** jeśli aplikacja korzysta z usług płatności bądź map, testy integracyjne pozwalają zweryfikować, czy te usługi działają w oczekiwany sposób.
- **Reguły biznesowe po stronie serwera:** weryfikują sekwencje operacji wykonywanych przy współpracy wielu komponentów.

Korzyści i wyzwania

Zaletą testów integracyjnych jest możliwość szybkiego wykrywania błędów tam, gdzie różne moduły muszą ze sobą współdziałać. **Wyzwanie** stanowi zaś konieczność skonfigurowania środowisk testowych (bazy danych, serwerów, stubów dla usług zewnętrznych). Przy dużych projektach może to być czasochłonne i wymagające w utrzymaniu.

2.3 Testy end-to-end (E2E)

Testy end-to-end (*E2E*) to najbardziej rozbudowane testy funkcjonalne, w których symuluje się rzeczywiste zachowanie użytkownika końcowego (lub komunikację między sys-

temami) w całym przepływie aplikacji. Obejmują one wszystkie warstwy: od interfejsu użytkownika, przez warstwę serwera, aż po bazę danych i usługi zewnętrzne.

Na czym polega koncepcja testów E2E w aplikacjach webowych

Ideą testów E2E jest sprawdzenie działania całej aplikacji jako jednego spójnego rozwiązania. Taki test może obejmować:

1. Uruchomienie przeglądarki i przejście na stronę logowania.
2. Wprowadzenie danych logowania i przejście do kolejnej podstrony.
3. Wykonanie akcji biznesowych (np. zamówienie produktu).
4. Weryfikację wyników w bazie danych czy w komunikatach interfejsu.

Wszystko po to, by sprawdzić, czy aplikacja rzeczywiście działa zgodnie z wymaganiami i oczekiwaniami użytkownika.

Kiedy i dlaczego warto je stosować

- **Sprawdzenie kluczowych scenariuszy:** testy E2E pokrywają najważniejsze ścieżki biznesowe, zapewniając, że są one wolne od błędów krytycznych.
- **Realne warunki:** testy odzwierciedlają działania użytkownika końcowego, wykrywając problemy mogące pojawić się dopiero przy faktycznej interakcji z aplikacją.
- **Wysoka wiarygodność:** potwierdzenie poprawnego działania całości systemu.

Testy end-to-end są jednocześnie najwolniejsze i najbardziej wymagające w utrzymaniu, ponieważ trzeba uruchamiać wszystkie usługi składające się na system. Dlatego używa się ich głównie do kluczowych scenariuszy aplikacji i ostatecznej weryfikacji przed produkcyjnym wdrożeniem.

Przykładowe narzędzia

- **Selenium WebDriver** – klasyczne narzędzie automatyzujące przeglądarkę, dostępne w wielu językach programowania.
- **Cypress** – nowoczesny framework do testowania front-endu z szybkim sprzężeniem zwrotnym.
- **Playwright** – rozwijany przez Microsoft, obsługuje testy E2E dla Chromium, Firefox oraz WebKit, a także wiele języków programowania.
- **TestCafe** – rozwiązanie pozwalające pisać testy E2E w JavaScripcie/TypeScripcie, niewymagające instalowania dodatkowych sterowników przeglądarek.

Rozdział 3

Dostępne technologie

Rozdział ten omawia najpopularniejsze narzędzia i biblioteki wspierające proces testowania aplikacji webowych. Wybór właściwych frameworków i rozwiązań znacząco wpływa na wygodę tworzenia i utrzymywania testów, a także na łatwość integracji z otoczeniem projektowym.

3.1 Frameworki testowe

Framework testowy to zbiór narzędzi i bibliotek pozwalających pisać, organizować oraz uruchamiać testy w sposób zautomatyzowany. W przypadku testów aplikacji webowych frameworki te oferują wsparcie dla różnych języków programowania (Java, Python, JavaScript, C#) i rodzajów testów (jednostkowe, integracyjne, end-to-end).

Przegląd wybranych rozwiązań

- **JUnit / TestNG** (Java) – powszechnie używane w projektach Java. JUnit świetnie sprawdza się w testach jednostkowych, natomiast TestNG posiada bardziej rozbudowane funkcje konfiguracyjne i umożliwia równoległe uruchamianie testów.
- **Pytest** (Python) – cechuje się prostą składnią i bogatym ekosystemem wtyczek. Integruje się z wieloma innymi narzędziami, np. do raportowania.
- **Mocha / Jest** (JavaScript) – Mocha to elastyczny framework do testów asynchronicznych i synchronicznych, a Jest (tworzony przez Facebook) jest popularny w aplikacjach React.
- **Cucumber** – pozwala pisać testy w formie zrozumiałej dla nietechnicznych interesariuszy (składnia Gherkin), popularny w metodyce *Behavior-Driven Development*.
- **Playwright** – oprócz sterowania przeglądarką oferuje własny test runner, raportowanie i możliwość pisania testów w JavaScripcie, Pythonie czy C#.

Kryteria wyboru

- **Język programowania** – warto wybrać framework naturalnie pasujący do języka wiodącego w projekcie.

- **Zakres testów** – dla testów front-endu React często wybiera się Jest, a do testów back-endu w Javie może lepiej sprawdzić się JUnit lub TestNG.
- **Integracja z CI/CD** – popularne narzędzia do ciągłej integracji (GitLab CI, GitHub Actions, Jenkins) posiadają często wtyczki czy gotowe przykłady integracji z określonym frameworkiem.
- **Społeczność i dokumentacja** – dojrzałe rozwiązania z dużą społecznością ułatwiają rozwiązywanie problemów i zapewniają bogate zasoby przykładów.

3.2 Biblioteki wspomagające generowanie testów

Nowoczesne narzędzia do automatyzacji testów aplikacji webowych coraz częściej oferują opcje częściowego lub pełnego *generowania* skryptów testowych. Może to przybierać formę rejestrowania czynności wykonywanych w przeglądarce (tzw. podejście *record-and-play*) lub analizy kodu źródłowego w celu wygenerowania przykładowych testów. Takie rozwiązania mogą szczególnie przyspieszyć prace zespołów rozpoczynających przygodę z automatyzacją lub działających w mniejszych projektach.

Podejście *record-and-play*

Najłatwiejszą metodą generowania testów bywa *nagrywanie* czynności wykonywanych przez użytkownika w przeglądarce:

- **Selenium IDE** – wtyczka do przeglądarek Chrome i Firefox, nagrywająca akcje i eksportująca je do kodu w różnych językach (Java, Python, C#).
- **Playwright Codegen** – wbudowana w Playwright funkcja uruchamiająca przeglądarkę w trybie interaktywnym i generująca na tej podstawie gotowy kod testu (TypeScript, Python, C#, Java).
- **TestCafe Recorder** – dostępne w postaci wtyczek rozszerzenie do Chrome, nagrywające kliknięcia i wpisywane dane, a następnie tworzące skrypty w stylu TestCafe.

Podstawową zaletą *record-and-play* jest niski próg wejścia (łatwość generowania wstępnych testów E2E), natomiast wada to dość *sztywne* skrypty, wrażliwe na zmiany w interfejsie użytkownika.

Analiza kodu i generowanie szkieletów testów

Innym rozwiązaniem jest narzędziowe wspomaganie pisania testów poprzez analizę kodu aplikacji:

- **Narzędzia do analizy statycznej** – mogą wykrywać ścieżki wykonania w kodzie, wskazywać podejrzane fragmenty i generować szkielety testów.
- **Boty testujące interfejs** – niektóre projekty open-source (*heuristics-based bots*) potrafią *klikać* w różne elementy według heurystyk, a następnie na tej podstawie generować wstępne skrypty E2E.

Podejścia te mogą wykryć scenariusze, o których człowiek mógłby nie pomyśleć, choć zazwyczaj wymagają dalszej konfiguracji czy refaktoryzacji kodu.

Zalety i wyzwania stosowania generatorów testów

Korzyści:

- *Oszczędność czasu* – szybkie stworzenie bazowej wersji skryptów testowych,
- *Niższy próg wejścia* – idealne dla nowych członków zespołu bądź mniej doświadczonych w automatyzacji,
- *Standaryzacja* – generatory często stosują uniwersalne wzorce, co ułatwia utrzymanie.

Wyzwania:

- *Brak elastyczności* – kod może być mocno zależny od konkretnego układu strony,
- *Konieczność refaktoryzacji* – wygenerowany kod bywa zbyt rozbudowany i wymaga uporządkowania,
- *Ciągłe aktualizacje* – przy zmianach w aplikacji część testów może przestać działać bez korekty generatora.

3.3 Narzędzia CI/CD

W większych projektach nie wystarcza jedynie lokalne uruchamianie testów. Testy automatyczne stają się integralną częścią cyklu ciągłej integracji i dostarczania (CI/CD), dzięki czemu każda nowa zmiana w kodzie jest automatycznie sprawdzana pod kątem jakości i stabilności. Poniżej omówiono kluczowe narzędzia CI/CD oraz sposoby włączania do nich testów.

GitLab CI

GitLab oferuje wbudowany mechanizm CI/CD oparty na pliku *.gitlab-ci.yml*. Aby uwzględnić w nim testy aplikacji webowych (np. napisane w Playwright czy Selenium), definiuje się:

- Etap testów (job) instalujący zależności,
- Polecenie uruchamiające testy (np. `npx playwright test`),
- Artefakty (raporty, zrzuty ekranu) pozwalające na łatwą inspekcję nieudanych uruchomień.

```
stages:
  - build
  - test

e2e_tests:
  stage: test
  image: mcr.microsoft.com/playwright:focal
  script:
    - npm ci
```

```
    - npx playwright test
  artifacts:
    when: on_failure
    paths:
      - playwright-report
```

GitHub Actions

GitHub Actions to popularne narzędzie do automatyzacji pracy w repozytorium GitHub, w tym do uruchamiania testów. Plik *.github/workflows/test.yml* konfiguruje *jobs*, definiując:

- Obraz bazowy (np. *ubuntu-latest*),
- Instalację wymagań,
- Uruchomienie testów,
- Publikację wyników (np. logów i raportów) jako *artifacts*.

Jenkins

Jenkins to jedno z bardziej konfigurowalnych narzędzi CI/CD typu open source. Służy do tworzenia *pipeline'ów*, w których:

- Pobiera się kod z repozytorium,
- Instaluje zależności i uruchamia testy,
- Generuje raporty (HTML, JUnit czy Allure),
- Prezentuje wyniki w interfejsie Jenkins po zakończeniu procesu.

Dostępnych jest wiele wtyczek, m.in. pozwalających na integrację z Selenium Grid czy Dockerem.

Kluczowe korzyści integracji testów z CI/CD

- **Stała kontrola jakości** – testy są wywoływane przy każdym *commit*,
- **Automatyczne raportowanie** – zespół ma bieżący wgląd w stan aplikacji,
- **Ciągłe dostarczanie** – w razie powodzenia testów, kod może zostać od razu wdrożony,
- **Elastyczność** – możliwość równoległego uruchamiania testów w różnych konfiguracjach.

Wyzwania i rozwój

- **Skonfigurowanie środowisk** – konieczne może być uruchamianie baz danych, mikroserwisów czy mocków,
- **Czas wykonania** – duże zestawy testów E2E wydłużają pipeline, co można łagodzić poprzez równoległe uruchamianie i skalowanie,
- **Utrzymanie** – wraz z rozwojem projektu rośnie liczba testów i ich złożoność, wymagając regularnych aktualizacji.

Rozdział 4

Strategie generacji testów

W poprzednich rozdziałach przedstawiono ogólne zasady automatycznego testowania aplikacji webowych i zaprezentowano wybrane technologie. Kolejnym krokiem jest omówienie różnych strategii generowania testów – od metod całkowicie automatycznych po podejścia półautomatyczne. Istnieje bowiem wiele sposobów tworzenia scenariuszy testowych, różniących się choćby poziomem ingerencji człowieka czy sposobem odwzorowania zachowań użytkownika.

W niniejszym rozdziale skoncentrujemy się na najczęściej stosowanych metodach generowania testów, w szczególności na podejściu *record-and-play* oraz rozwiązaniach opartych na modelu aplikacji. Dla mniejszych projektów liczy się często błyskawiczne stworzenie zestawu testowego, co motywuje do korzystania z narzędzi rejestrujących realne akcje w przeglądarce. W większych, bardziej zorganizowanych zespołach spotyka się narzędzia bazujące na formalnym opisie zachowania systemu, co pozwala generować obszerne zestawy testów pokrywające wiele nietypowych ścieżek.

4.1 Generowanie testów na podstawie rejestrowania akcji

Jednym z najbardziej intuicyjnych sposobów przyspieszających tworzenie testów automatycznych jest rejestrowanie czynności użytkownika w przeglądarce, często określane mianem podejścia *record-and-play*. Polega ono na „nagrywaniu" akcji wykonywanych przez testera (lub dewelopera) na stronie internetowej: kliknięć, wprowadzania danych w pola tekstowe czy przechodzenia między podstronami. Następnie narzędzie, które uczestniczy w tym procesie, generuje skrypt testowy w wybranym języku programowania.

Zalety podejścia *record-and-play*

- **Niski próg wejścia:** do przygotowania wstępnych testów nie trzeba znać szczegółowo frameworków testowych – wystarczy poprawnie wykonać scenariusze w przeglądarce.
- **Szybkie prototypowanie:** w ciągu kilku minut można uzyskać podstawowy plik testowy, który potem można udoskonalić.
- **Naturalne odwzorowanie zachowań użytkownika:** test powstaje na bazie realnego korzystania z aplikacji.

Wyzwania i ograniczenia

- **Nadmierna szczegółowość skryptu:** automatycznie wygenerowane testy są zwykle wrażliwe na nawet drobne zmiany w interfejsie.
- **Brak abstrakcji i trudniejsza konserwacja:** w *record-and-play* często nie ma wzorców typu Page Object, przez co kod może być trudniejszy w utrzymaniu.
- **Ograniczone pokrycie przypadków brzegowych:** narzędzia nagrywające skupiają się na głównych ścieżkach użytkownika.

Przykładowe narzędzia

- **Selenium IDE** – rozszerzenie do Chrome/Firefox, które umożliwia nagrywanie akcji i eksport skryptów (Java, Python, C#).
- **Cypress Recorder** – wtyczki do Chrome do rejestrowania czynności i generowania testów w stylu Cypress.
- **Playwright Codegen** – narzędzie wbudowane w Playwright, pozwala uruchomić przeglądarkę w trybie interaktywnym i generować testy w TypeScripcie, Pythonie czy .NET.

Podsumowując, podejście *record-and-play* świetnie nadaje się do szybkiego uzyskania bazowego zestawu testów, choć w dłuższej perspektywie wymaga zwykle wprowadzenia wzorców ułatwiających utrzymanie (np. Page Object Model).

4.2 Generowanie testów w oparciu o model systemu

Kolejnym sposobem automatycznego tworzenia skryptów testowych jest *Model-Based Testing* (MBT). W tym podejściu scenariusze testowe są generowane na bazie formalnego modelu systemu, opisującego możliwe stany i przejścia między nimi (np. w formie diagramów stanów lub sieci Petriego).

Zasada działania

1. **Budowa modelu:** przygotowanie schematu zachowania systemu (np. diagram stanów, opis przejść).
2. **Definicja danych testowych:** określenie danych wejściowych i oczekiwanych wyników dla poszczególnych akcji.
3. **Generowanie ścieżek:** narzędzie MBT automatycznie wyznacza ścieżki przejść w modelu, starając się osiągnąć ustalone kryterium pokrycia (np. wszystkie stany lub wszystkie przejścia).
4. **Konwersja na skrypty testowe:** każda ścieżka to osobny test. Narzędzie tłumaczy go na kod w wybranym frameworku.

Zalety i przykładowe zastosowania

- **Szersze pokrycie ścieżek:** automatyczna eksploracja modelu wychwytuje błędy w mniej typowych scenariuszach.
- **Synchronizacja z dokumentacją:** aktualizacja modelu w razie zmiany wymagań przekłada się na natychmiastowe odświeżenie testów.
- **Refaktoryzacja i konserwacja:** gdy model jest utrzymany w sposób czytelny, łatwiej zachować wysoką jakość testów w dłuższym horyzoncie.

Wyzwania i ograniczenia

- **Konieczność posiadania modelu:** przygotowanie i aktualizacja formalnego modelu systemu bywa pracochłonne.
- **Specjalistyczne narzędzia:** MBT wymaga często dedykowanego oprogramowania oraz znajomości specyficznych notacji.
- **Integracja z testami E2E:** wygenerowane testy trzeba nierzadko dostosować do konkretnych frameworków (Selenium, Playwright) i środowisk CI/CD.

Przykładowy proces

1. Przygotowanie diagramu stanów (np. logowanie, wyszukiwanie, wylogowanie).
2. Zdefiniowanie warunków przejść i danych (np. nazwy użytkowników, hasła).
3. Wygenerowanie ścieżek i tłumaczenie ich na skrypty testowe.
4. Uruchomienie testów w ramach istniejącego procesu CI/CD.

Podejście MBT jest szczególnie atrakcyjne w większych, długoterminowych projektach, gdzie sprawne utrzymanie pełnej dokumentacji i testów ma kluczowe znaczenie.

Rozdział 5

Strategie generowania testów

Wstęp do rozdziału 4

Testy automatyczne w większych projektach często wymagają elastycznego i wydajnego podejścia do przygotowywania scenariuszy. Pojawia się zatem potrzeba narzędzi oraz strategii, które umożliwią **generowanie** skryptów testowych w zależności od specyfiki projektu, rodzaju testowanej aplikacji oraz preferowanego frameworka testowego.

Celem niniejszego rozdziału jest przedstawienie koncepcji i architektury systemu, który wspomaga proces tworzenia testów w sposób zautomatyzowany lub półautomatyczny. Omówione zostaną kluczowe elementy projektu, a także zaprezentowane diagramy klas i przypadków użycia. W dalszych sekcjach zostaną wskazane możliwości rozbudowy rozwiązania o dodatkowe funkcje, takie jak integracja z pipeline'ami CI/CD czy analiza błędów w generowanych scenariuszach.

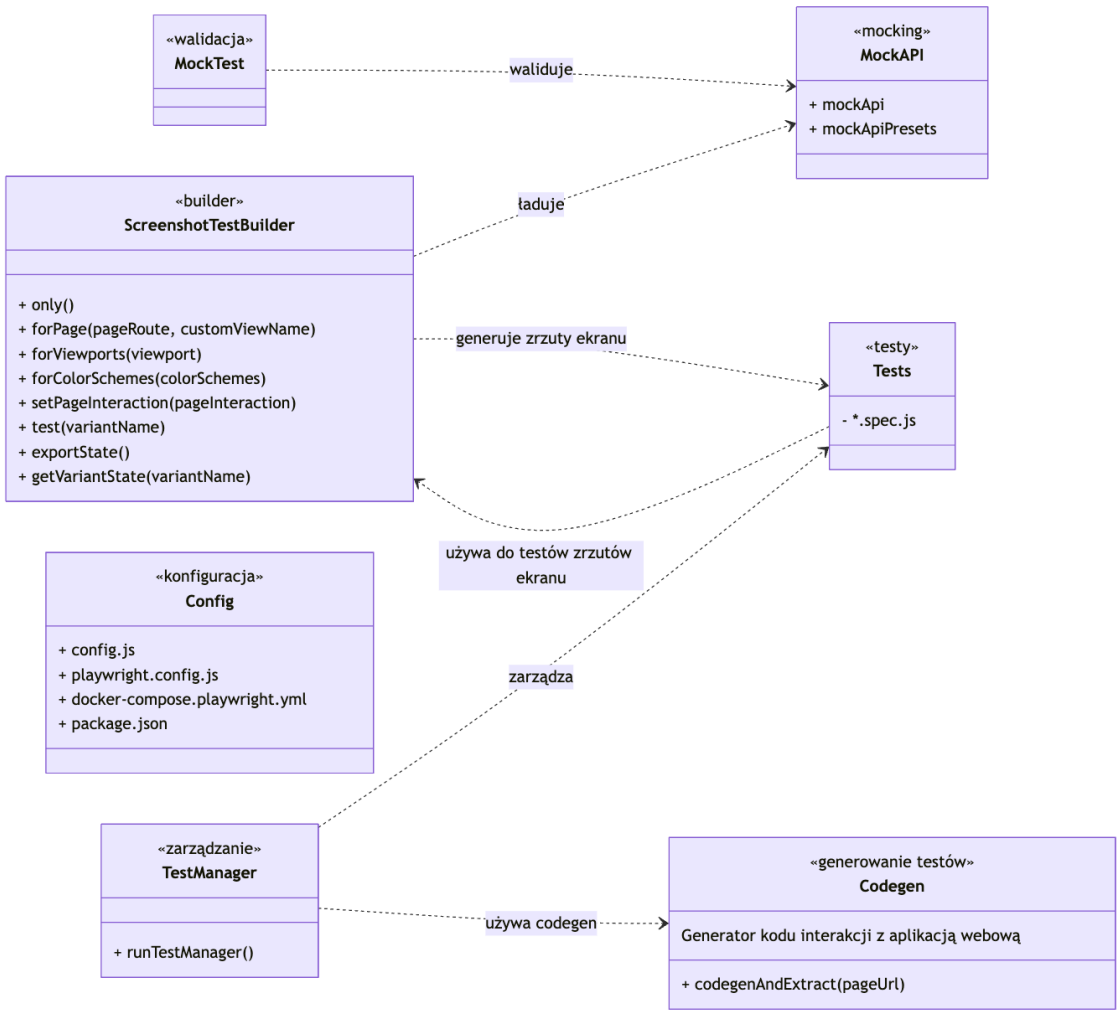
5.1 Analiza diagramów i architektura

W celu zrozumienia podstaw działania aplikacji oraz jej modułów, przygotowano dwa główne diagramy. Pierwszy z nich (Figure 5.1) przedstawia najważniejsze **klasy i relacje** pomiędzy nimi w systemie. Drugi (Figure 5.2) obrazuje **przypadki użycia**, czyli scenariusze interakcji użytkownika (bądź systemu CI/CD) z tworzonym narzędziem.

5.1.1 Diagram klas

Na Figure 5.1 widoczny jest podział na kilka zasadniczych modułów:

- *Generator kodu testów (Codegen)* – odpowiada za przekształcenie zadanych parametrów (np. akcji użytkownika z przeglądarki) w kod testowy wybranego frameworka.
- *Zarządzanie testami (TestManager)* – stanowi warstwę orchestracji, w której operator może m.in. wprowadzać modyfikacje w interakcjach testowych czy usuwać niepotrzebne kroki.
- *Moduły wspomagające (Helpers, MockAPI)* – ułatwiają generowanie przykładowych danych, weryfikację poprawności oraz walidację odpowiedzi (np. przeciwko fałszywym usługom).



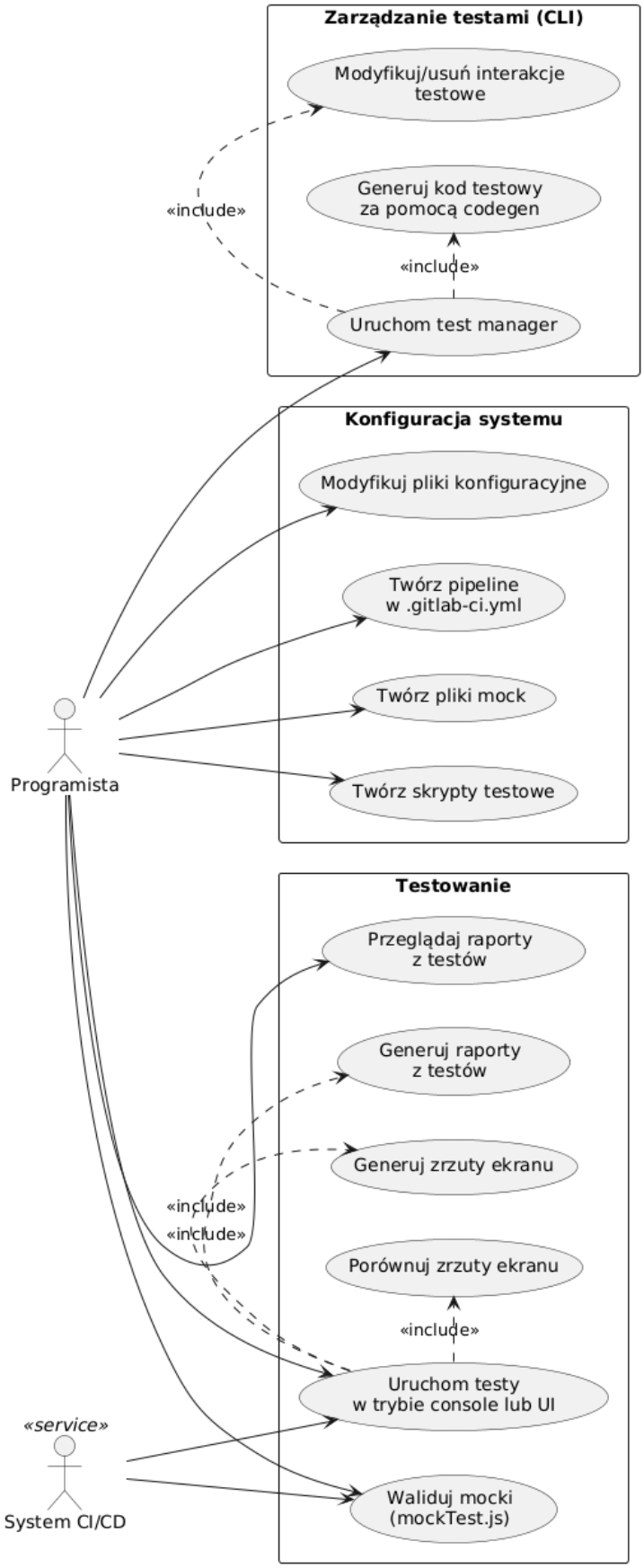
Rysunek 5.1: Diagram klas wygenerowany w Mermaid

Na powyższym diagramie zaprezentowano również interakcje klas, takie jak:

- `ScreenshotTestBuilder` współpracuje z `MockAPI` w celu ładowania danych i tworzenia zrzutów ekranu.
- `TestManager` używa `Codegen` do generowania kodu testów.
- `MockTest` służy do walidacji spójności między danymi testowymi a zasobami API.

5.1.2 Diagram przypadków użycia

Kolejnym istotnym elementem projektu jest **diagram przypadków użycia** (Figure 5.2), wskazujący na ogólny przepływ interakcji pomiędzy użytkownikiem (np. programistą lub testerem) a systemem. Przykładowo, „System CI/CD" może automatycznie uruchamiać testy przy każdym *pushu* do repozytorium i wykorzystywać w tym celu gotowe skrypty wygenerowane przez nasze narzędzie.



Rysunek 5.2: Diagram przypadków użycia wygenerowany w PlantUML

Diagram prezentuje także rozszerzone akcje w `TestManager`, takie jak:

- *Generowanie kodu* (Codegen) w oparciu o zarejestrowane kroki,
- *Modyfikacje i usuwanie interakcji*, np. gdy któryś z kroków staje się nieaktualny.

W następnych rozdziałach opisano w szczegółach implementację poszczególnych modułów (zarówno w zakresie kodu źródłowego, jak i logiki generującej scenariusze testowe), a także zaprezentowano sposoby integracji z zewnętrznymi narzędziami, np. GitLab CI czy Jenkinsem.

5.2 Interfejs użytkownika

Zaprezentuj, jak wygląda interfejs (graficzny lub konsolowy) Twojej aplikacji:

- główne ekrany/formularze,
- sposób nawigacji,
- kluczowe funkcje (np. generowanie pliku testowego, zapis ustawień).

Możesz dołączyć zrzuty ekranu i omówić je.

Rozdział 6

Omówienie kodu źródłowego aplikacji

Przedstaw strukturę plików oraz folderów oraz wyjaśnij najważniejsze fragmenty implementacji. Wskaż, które biblioteki z rozdziału o „Dostępnych technologiach" zostały wykorzystane i w jaki sposób.

6.1 Struktura projektu

Opisz, w jaki sposób podzieliłeś/łaś projekt na moduły, pakiety, foldery:

- logika generowania testów,
- definicja modelu danych (np. dane o testach, konfiguracje),
- klasy pomagające w tworzeniu raportów lub integracji z CI.

6.2 Kluczowe fragmenty kodu

Zademonstruj przykłady najważniejszych funkcji/metod, np.:

- kod odpowiedzialny za rejestrowanie interakcji,
- generowanie plików testowych,
- konwersję danych do konkretnych frameworków.

Przedstaw je w formie listingów z krótkim komentarzem.

Rozdział 7

Przykłady użyć aplikacji

W tym rozdziale możesz zaprezentować, jak Twoje narzędzie działa w praktyce.

7.1 Uruchamianie narzędzia w trybie interaktywnym

Pokaż krok po kroku, jak użytkownik wypełnia formularz czy wybiera opcje w CLI, aby wygenerować testy. Zademonstruj końcowe pliki testowe.

7.2 Integracja z procesem CI/CD

Wyjaśnij, jak można włączyć wygenerowane testy do automatycznego pipeline'u, np. w GitLab CI, Jenkinsie, itp. Przedstaw przykładowy fragment konfiguracji.

Rozdział 8

Podsumowanie

Na koniec podsumuj swoją pracę, oceń jej efekty i zaproponuj kierunki rozwoju.

8.1 Wnioski końcowe

Opisz, w jakim stopniu udało się osiągnąć założone cele, co było największym wyzwaniem, co dało najwięcej satysfakcji/rezultatów.

8.2 Możliwości dalszego rozwoju

Opisz, jakie jeszcze funkcjonalności można by dodać do narzędzia, jakie usprawnienia byłyby przydatne, jakie biblioteki lub technologie mogłyby zostać wykorzystane w przyszłości.

Bibliografia

[1] Roman, A., & Zmitrow, K. (2024). *Testowanie oprogramowania w praktyce: studium przypadków 2.0*.

[2] Osherove, R. (2024). *Testy jednostkowe: świat niezawodnych aplikacji*.

[3] Roman, A., & Zmitrow, K. (2024). *Testowanie oprogramowania w praktyce: studium przypadków*.

[4] Roman, A. (2024). *Testowanie i jakość oprogramowania: modele, techniki, narzędzia*.

[5] CircleCI. (n.d.). What is End-to-End Testing? Pozyskano z `https://circleci.com/blog/what-is-end-to-end-testing/`

[6] Microsoft Playwright. (n.d.). Introduction to Playwright. Pozyskano z `https://playwright.dev/docs/intro`