Wytwarzanie oprogramowania to niekończące się próby dostarczania i ulepszania produktów sprecyzowanych przez klienta. Jest to ewolucja pomysłu, na którego urzeczywistnianie często pracuje ogromna liczba zespołów. Cały przebieg tworzenia oprogramowania składa się z wielu aspektów, zarówno biznesowych jak i technicznych. Umiejętne ich połączenie i dopracowanie każdego z nich, przekłada się na powodzenie całego procesu. Przykładowy proces składa się między innymi na analizę wymagań klienta, następnie przełożenie ich na kod źródłowy, połączenie modułów kodu w działające ze sobą funkcjonalności i prezentację produktu końcowego. Aby zapewnić jak najwyższą jakość wytwarzanego oprogramowania, w każdej z tych składowych powinno być przeprowadzane rzetelne testowanie tworzonego produktu. W celu prawidłowego testowania powinno się wykorzystywać środowiska testowe. Ze względu na zwykle dużą ilość pracy oraz trudności w jego utrzymywaniu, mają one dość niski priorytet w zespołowych planach pracy. Nie mniej jednak prawidłowe utworzenie i zarządzanie takiego środowiska, może znacznie ułatwić i poprawić jakoś testowania w projekcie. Co więcej w niektórych przypadkach, stworzenie odpowiedniego środowiska testowego nie musi być procesem trudnym i dużo mniej czasochłonnym niż mogłoby się wydawać, szczególnie biorąc pod uwagę testowanie manualne aplikacji internetowych, które bywa procesem bardzo czasochłonnym i jeśli powtarzalnym - zazwyczaj dość nudnym dla testera. Odpowiednio skonfigurowane i wykorzystujące odpowiednie narzędzia środowisko, można zautomatyzować do wykonywania powtarzających się testów, a tym samym ułatwić pracę testerów i przyspieszyć proces testowania. Celem poniższej pracy jest stworzenie projektu środowiska testów automatycznych stron internetowych. Jest to środowisko uniwersalne, którego założenia mogą być użyte przy testowaniu każdej aplikacji internetowej, posiadającej interfejs użytkownika. W pracy zostały zastosowane takie metody badawcze jak krytyczna analiza literatury przedmiotu oraz własne doświadczenia w pracy testera. Literatura oparta jest o tytuły zarówno polskie, spolszczone jak i angielskojęzyczne, znajdują się w nich książki o tematyce informatycznej a także słowniki pojęć używanych w testowaniu.

Praca została podzielona na kilka kluczowych rozdziałów. Pierwszy z nich jest rozdziałem wprowadzającym, opisuje podstawową teorię testowania, jego metody i rodzaje. Drugi rozdział również ma charakter teoretyczny, stanowi on opis zagadnień związanych ze środowiskiem testów i automatyzacją. W trzecim rozdziale zostało przedstawione narzędzie Selenium, jego metody wykorzystywane w prezentowanym środowisku oraz wzorce stosowane w

**1.TEORIA TESTOWANIA**

**1.1.Istota testowania – dlaczego testujemy?**

1.1.1. Elementarne pojęcie testowania

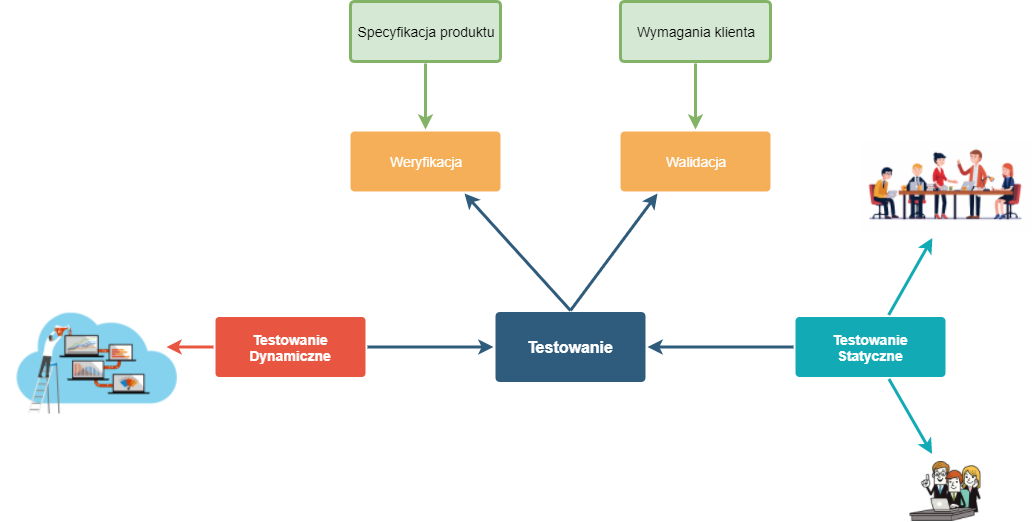
„*Testowanie jest niekończącym się procesem polegającym na porównywaniu tego, co niewidzialne, do tego, co wieloznaczne, a wykonuje się je po to, aby coś, co jest niemożliwe do przewidzenia, nie dotknęło tych, których nie znamy z imienia*”[[1]](#footnote-1).

Głównym rezultatem poprawnie przeprowadzonego testowania jest znajdowanie nieprawidłowego działania w wytwarzanym oprogramowaniu. Znalezienie takich nieprawidłowości, umożliwia ich zlikwidowanie, bądź poprawienie, a co za tym idzie – zapobieganie wystąpienia takich niechcianych defektów w produkcji, gdzie usuwanie błędów jest najbardziej kosztowne w całym procesie wytwarzania oprogramowania. Poprzez testowanie określa się również poziom ryzyka związanego z testowanym produktem oraz precyzuje jego jakość. Odpowiednio przeprowadzone, udokumentowane i przeanalizowane testowanie może nawet zapobiec powstawaniu błędów, i pomóc zwiększaniu jakości produktu, która jest kluczowym aspektem powodzenia projektu informatycznego. Efekt końcowy uzyskany na podstawie wniosków przeprowadzanego procesu, ukazuje prawdziwą jakość produktu, którą całe zespoły informatyczne pragną dostarczyć w jak najwyższej postaci. Testerzy przed którymi stoi zadanie wywierania na oprogramowaniu działań, które mają je przetestować pod odpowiednim kątem, w odpowiedni sposób, wybranymi narzędziami, przeprowadzają takie badania w oparciu o podstawowe podziały tego terminu. Owe pojęcie jest jednym z najobszerniejszych zagadnień w całej teorii wytwarzania oprogramowania. Istnieje ogromna ilość czynników dzielących testowanie w przeróżny sposób w zależności od obranego punktu widzenia. Poniżej zostały przedstawione najbardziej znane i najpowszechniej używane rozłamy tej gałęzi informatyki.

1.1.2. Podział testowania na statyczne i dynamiczne

Testowanie to proces wykorzystujący zarówno mechanizmy statyczne jak i dynamiczne, w celu weryfikacji czy dany produkt jest zgodny z jego specyfikacją i walidacji, czy produkt jest poprawny względem potrzeb i wymagań użytkownika.

Statyczny rodzaj skupia się na testowaniu produktu przed jego wykonaniem(kompilacją, uruchomieniem kodu). Takie testy przeprowadzane są przy użyciu przeglądów(procesów, lub spotkań, gdzie produkt przestawiany jest zainteresowanym osobom – przykładowo zespołowi projektowemu, w celu uzyskania opinii na temat rozwiązania) lub analizy statycznej(przeglądaniu kodu). Testowanie dynamiczne natomiast oparte są na uruchomieniu kodu i porównywaniu jego rezultatów z oczekiwaniami[[2]](#footnote-2).



Rysunek 1. Podstawowy podział testowania i jego cele. Źródło: Opracowanie własne.

1.1.3. Fazy testowania

W przedstawionej powyżej definicji testowania słowo proces zawiera w sobie kilka głównych faz. Wiadomo, iż stosuje się różne podejścia do testowania w zależności między innymi od wytwarzanego produktu, zespołu oraz filozofii, według której taki produkt jest wytwarzany. Nie mniej jednak, możliwe jest wyodrębnienie głównych etapów testowania – zazwyczaj występujących w podstawowym procesie testowym:

* Planowanie i nadzór

Samo planowanie jest kluczowym elementem procesu, przede wszystkim polega na określeniu celów testowania oraz wyboru odpowiednich czynności testowych, potrzebnych do zrealizowania postawionych celów.

Nadzór natomiast wymaga ciągłej pracy polegającej na monitorowaniu przebiegu zaplanowanych czynności testowych oraz raportowaniu ich rezultatów.

* Analiza i projektowanie

Oba etapy procesu są bardzo obszerne, ich głównymi zadaniami są:

* przegląd i ocena podstawy testów- czyli przebadanie dokumentacji projektu mającej znaczenie z testerskiego punktu widzenia(wymagań, poziomu ryzyka, architektury projektu, poziomu integralności oprogramowania itp.)
* ocena stopnia testowalności produktu
* określenie i priorytetyzacja warunków testowych i przypadków testowych
* projektowanie środowiska testowego wraz z używanymi narzędziami w jego ramach
* tworzenie dwukierunkowego śledzenia pomiędzy podstawą testów oraz przypadkami testowymi
* Implementacja i wykonanie

Tworzenie oraz wykonywanie testów na podstawie wcześniejszej analizy i zaprojektowanego środowiska, raportowanie rezultatów przeprowadzanych testów

* Ocena stopnia spełnienia warunków zakończenia

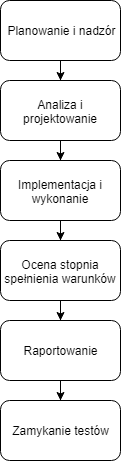
Polega na porównaniu wyników testowych z oczekiwanymi rezultatami oraz na ocenie, czy kryteria zakończenia testów są odpowiednie dla testowanego przypadku

* Raportowanie

Tworzenie raportu podsumowującego osiągnięte rezultaty testowe

* Zamykanie testów

Zbieranie ostatecznych danych z zakończonych czynności testowych, w celu ich analizy[[3]](#footnote-3)

****

Rysunek 2. Fazy testowania. Źródło: Opracowanie własne.

1.1.4. Siedem podstawowych zasad testowania

Testowanie w każdym projekcie może wyglądać inaczej. Procesy testowe, szczególnie w dużych projektach mogą rządzić się własnymi zasadami określonymi przez wyznaczone osoby i specyfikacje. Jednak testowanie, nie zależnie od projektu i przyjętego podejścia posiada siedem podstawowych zasad:

1. Testowanie ujawnia usterki

Niemożliwe jest napisane idealnego systemu, czy aplikacji. Każde wytwarzane oprogramowanie zawiera niedoskonałości a jednym z celów testowania jest je ujawniać.

1. Testowanie gruntowe jest niewykonalne

Przetestowanie każdego przypadku, jaki może dotyczyć danej funkcjonalności lub systemu nie jest możliwe. Testerzy powinni kierować się objętą strategią, a nie starać się przetestować wszystko.

1. Wczesne testowanie

Testowanie produktu powinno rozpocząć się jak najwcześniej. Zyskuje się przy tym więcej czasu do obrania strategii naprawy błędu i wprowadzenia jej w życie, a także sama naprawa niedoskonałości jest mniej kosztowna, niż w dalszych cyklach życia projektu.

1. Kumulowanie się błędów

Natura usterek ma to do siebie, iż usterki bardzo często występują obok siebie, często w tym samym module projektu, również tam gdzie testowanie jest przeprowadzane najczęściej, a nawet w tych, gdzie ryzyko jest najwyższe

1. Paradoks pestycydów

Testy powinny być co jakiś czas odnawiane i podawane modyfikacjom. Jeżeli monotonnie wykonywane będą te same scenariusz testowe – liczba błędów aplikacji może zmaleć, jednak tylko dlatego, iż testy nie będą w stanie znaleźć żadnych nowych błędów.

1. Testowanie zależy od kontekstu

W zależności od poziomu ryzyka, przeznaczenia projektu czy danej funkcjonalności dobiera się różne mechanizmy, scenariusze i narzędzia testowe.

1. Mylne przekonanie o braku błędów

Nawet jeśli produkt zostanie dogłębnie przetestowany oraz przygotowany do wypuszczenia na rynek, nie oznacza to, że nie zawiera błędów. Przede wszystkim proces testowania powinien być oparty o weryfikację czy produkt, zawiera wymagania i oczekiwania klientów.[[4]](#footnote-4)

1.1.5. Poziomy testów

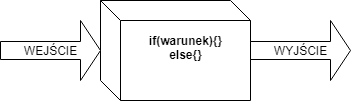
* Testy modułowe (jednostkowe) – polegają na weryfikacji pojedyncze funkcjonalności oprogramowania, przykładowo modułu, klasy i tym podobne moduły programistyczne.
* Testy integracyjne – weryfikują integrację pomiędzy modułami systemu, sprawdzają poprawność interfejsów i interakcji pomiędzy nimi.
* Testy systemowe – testy sprawdzające działanie funkcjonalności z punktu widzenia użytkownika, czy spełnia one specyfikowane wymagania.
* Testy akceptacyjne – ich celem jest nabranie zaufania do systemu oraz weryfikacja czy produkt jest gotowy do wdrożenia

1.1.6. Typy testów

* Testy funkcjonalne – sprawdza funkcjonalności oprogramowania, „co robi dany moduł?”
* Testy niefunkcjonalne – sprawdza działanie oprogramowania „jak działa oprogramowanie?”
* Testy regresywne – wykonywane po zmianach kodu, w celu upewnienia się, iż po wprowadzonych zmianach nie pojawiły się nowe błędy
* Re-testy – testy potwierdzające, polegające na ponownym uruchomieniu testu, po tym jak defekty znalezione przez niego, zostały naprawione

1.1.7. Podział testów na biało i czarno-skrzynkowe

W dużym uproszczeniu biało skrzynkowe testowanie można określić jako programistyczne. Dotyczą one kodu źródłowego wytwarzanego produktu. Umożliwiają one przetestowanie pojedynczych linii i części funkcjonalności kodu. W związku z tym głównym atutem białych skrzynek jest możliwość podejrzenia takich niskopoziomowych etapów jak struktura danych aplikacji czy algorytmy wewnętrzne. Jednak testy samego kodu są jedynie jedną z metod testowania biało skrzynkowego. Wyróżnia się także między innymi przeglądy kodu, dzięki którym możliwe jest zweryfikowanie jego poprawności bez kompilowania(testowanie statyczne). Jako testy „białe” definiuje się także bardzo popularne w testach jednostkowych, tak zwane „wstrzykiwanie defektów” poprzez dane wejściowe.



Rysunek 3. Testowanie biało-skrzynkowe. Źródło: Opracowanie własne.

Testy czarno skrzynkowe, inaczej zwane systemowymi, bazują na interfejsach poza   
„białą skrzynką” - polegają na tym co kod aplikacji wystawia na zewnątrz. Do ich metod między innymi należą „klikane” testy manualne, czy testy automatyzujące „przeklikiwanie” testów manualnych.



Rysunek 4.Testowanie czarno-skrzynkowe. Źródło: Opracowanie własne.

1.1.8. Historia testowania

Testowanie oprogramowania jest nieodłącznym procesem w cyklu jego wytwarzania i utrzymywania. Odgrywa znaczącą rolę w każdym projekcie oraz w historii informatyki. Testowanie samo w sobie, towarzyszy człowiekowi od dawien dawna. W każdym poprawnie wytwarzanym produkcie, stawiano na jego jakość i zadowolenie klienta. Nie jest inaczej w przypadku testowania oprogramowania. Aby osiągnąć cele wyznaczone przez klienta koniecznym było odpowiednie zaplanowanie, kontrola i poprawianie jakości produktu. Dwa ostatnie aspekty, opierają się na testowaniu zaplanowanego i wytworzonego dzieła. Do 1956 roku testowanie postrzegane było jako debugowanie. Od 1957 do 1978 następowało metodyczne rozróżnienie tych dwóch czynności i czas potwierdzania, że oprogramowanie jest zgodne z wymaganiami.

1979-1982 to "czas destrukcji", kiedy to celem było znajdywanie defektów. Od 1983 do 1987 następowała zmiana oceniania oprogramowania w kategoriach mierzenia jego jakości. Od 1988 zaczęliśmy mówić o testowaniu jako formie zapobiegania powstawania defektów.[].

Pierwszą publiczną wzmianką o testowaniu oprogramowania zawarł w swojej publikacji Charles L. Baker w 1957 roku o nazwie „Mathematical Tables and Other Aids to Computation” gdzie rozróżnił pojęcie testowania programów i debagowania. Rok później Gerald M. Weinberg stworzył pierwszy zespół testerski, pracujący w projekcie „*Mercury*”, dotyczący lotów załogowych w kosmos. W 1961 roku w książce „*Computer Programming Fundamentals*” pojawił się pierwszy rozdział w całości dedykowany testowaniu. Pierwsze pomysły o sformalizowaniu procesu testowania pojawiły się w 1967 roku w publikacji „*Evaluation of the Functional Testing of Control Programs*”. Sama teoria testowania oprogramowania kształtowała się na przestrzeni lat, a w 1969 roku została wypowiedziana jedna z najważniejszych zasad testowania iż, „Testowanie ujawnia obecność defektów a nie ich brak” przez Edsgerda W. Dijkstra. [[5]](#footnote-5)

W miarę rozwoju technologii, zaczęły pojawiać się standardy testowania, rodzaje oraz zaczęto coraz bardziej doceniać owy proces w cyklu wytwarzania oprogramowania. Powstały metodologie tworzenia projektów, takie jak Agile, czy model kaskadowy– w których testowanie stało się jednym z ważniejszych etapów.

1.1.9. Testowanie w metodologiach zwinnych – przykład Agile oraz rola testera

Filozofia Agile opiera się głównie na elastyczności, współpracy i adaptacji do narzucanych wymagań i zmian. Model został zbudowany w oparciu o ciągłe zmiany w wytwarzanych produktach. Główną zaletą takiego podejścia jest minimalizacja ryzyka związanego z ewentualną zmianą jakiejś części projektu. Agile zachęca zespoły wykorzystujące tę metodologię do systematycznej dokumentacji i częstej prezentacji swoich postępów oraz do zbierania informacji zwrotnych – wszystko po to, aby jak najszybciej przystosować się do zmian.[[6]](#footnote-6) Najlepszą definicją procesu testowania w metodologii zwinnej, jest opis pracy testera:

„W przypadku pracy testera w Agile wykreowały się pewne kluczowe wymagania zwinnej metodyki:

* skupienie się i uznanie za priorytetowe traktowanie wymagań opartych na ryzyku, ponieważ nie jest możliwe przetestowanie wszystkiego,
* automatyzacja testów w celu zwiększenia wydajności
* częstsze wykorzystywanie testów eksploracyjnych, by zmniejszyć czas pomiędzy dostarczaniem kodu a wykonywaniem testów, jednocześnie podkreślając potrzebę tworzenia działającego kodu
* przystosowanie do zmian od sprintu do sprintu”[[7]](#footnote-7)

Tester jak członek „zwinnego” zespołu powinien być w stałym kontakcie z programistami, aby pracować nad testowaniem przez cały cykl życia projektu oraz zapewniać dobrą komunikację w projekcie, na której w dużej mierze opiera się jakość wytwarzanego produktu. Często zdarza się, iż praca testera z pracą programisty łączą się w pewien sposób, w zależności od wykonywanego zadania. Stanowi to odejście od wąskich umiejętności testerskich, ważna jest bowiem adaptacyjność, wymagająca wykazywaniem się szerszymi, interdyscyplinarnymi umiejętnościami. Co ważne, prace testera i programisty mogą być wykonywane w tym samym czasie, są bardzo często niezależne od siebie. Testowanie nie polega wyłącznie na tym, co napisał programista, ale na szerszym spojrzeniu na cały projekt.

Testowanie w metodologiach zwinnych, jest jednym z czynników napędzających takie podejście do tworzenia projektów informatycznych. W dzisiejszych czasach zwinność jest już stosowana powszechnie. Ogromne znaczenie w takim podejściu mają całe środowiska testowe – ułatwiające przeprowadzanie testowania oraz zapewnianie jakości, nawet w obliczu ciągłej zmiany i dostosowywania się do niej.

**2. ŚRODOWISKA TESTOWE I AUTOMATYZACJA**

**2.Środowisko testowe**

2.1.1. Pojęcie środowiska testowego

Wytwarzanie oprogramowania, jest procesem, przechodzącym przez kilka środowisk, oczywiście w zależności od projektu. Najczęstszym podejściem jest rozdzielenie projektu na środowisko deweloperskie, produkcyjne i w optymalny sposób oparte na podstawie produkcyjnego – środowisko testowe. Jak sama nazwa wskazuje, jest to zespół modułów wytwarzanego oprogramowania, oprawiony w strukturę infrastruktury technicznej – zależnie od specyfikacji projektu i odwzorowania środowiska produkcyjnego. Ogółem takie środowisko pozwala na uruchomienie i przetestowanie danej aplikacji według określonych reguł i założeń. Kluczowym jest takie zaprojektowanie środowiska, aby możliwe było przetestowanie wszystkich funkcjonalności. Ważnym aspektem takiej części projektu jest również jego izolacja i niezależność od deweloperów – głównie ze względu na konieczność zaufania do gromadzonych w takim środowisku danych testowych, oraz znajdującego się w nim kodu. Stabilność jest także bardzo ważnym wymaganiem środowiska, Powinno ono pracować bez niechcianych przerw i błędów. Prawidłowo utrzymywany moduł testowy projektu pozwoli na gromadzenie danych niezbędnych do powtarzania testów a co za tym idzie – zapewniania jakości wytwarzanego oprogramowania. Jako środowisko testowe możemy rozumieć nie tylko stworzoną część projektu przez testerów, lub deweloperów po stronie wykonawcy. Po stronie klienckiej również bardzo często korzysta się z takich środowisk. Wykorzystuje się je do testów odbiorczych otrzymywanego produktu, czy też przykładowo w celu przetestowania błędów występujących w środowisku produkcyjnym.

2.1.2. Administracja środowiska testowego

Stworzenie wyizolowanego i niezależnego środowiska testowego jest dużym wyzwaniem, nawet dla całego zespołu testerskiego czy programistycznego. Jednak utrzymywanie i pielęgnowanie takiej części projektu, jest procesem zwykle jeszcze trudniejszym. Wytwarzane oprogramowanie, szczególnie w metodologiach zwinnych musi być podatne na zmiany. W związku z tym identycznego zachowania oczekuje się od środowiska testowego. Bardzo ważne jest zapewnianie ciągle zmieniającej się konfiguracji(szczególnie w dużych, komercyjnych projektach) w celu uzyskania jak najlepszego odwzorowania produktu końcowego i zweryfikowania jego jakości. Konieczna jest nieustanna praca i opieka nad środowiskiem testowym przez administratora lub wyznaczony do tego zespół. Wszystko to aby możliwe było rozwijanie zasobów w odpowiedzi na potrzeby i zmieniające się specyfiki systemu. Między innymi dlatego, kluczową rolę w dzisiejszych projektach informatycznych odgrywa automatyzowanie zarządzania takim środowiskiem, a także procesami testerskimi, przebiegającymi wewnątrz.

**2.2. Automatyzacja w testowaniu**

2.2.1. Pojęcie automatyzacji testów

W dużym uproszczeniu, można powiedzieć, że jest to wykorzystanie programowania do przeprowadzania powtarzających się testów. Zastępowanie manualnej pracy testera, poprzez skrypty automatyczne, tworzone, utrzymywane, wykonywane i rozwijane wedle określonych reguł. Wykorzystywanie automatyzacji w projekcie z reguły jest bardzo opłacalne – prawidłowo wdrożona przejmuje zadania nawet kilku osób, które mogą zająć się innymi zadaniami. Jednak należy pamiętać iż utrzymywanie takiego procesu wymaga przynajmniej jednej osoby, stale sprawującej pieczę nad automatycznymi skryptami i czynnościami z nimi związanymi. Jej zasadniczym celem jest minimalizacja kosztów związana z wykonywaniem powtarzalnych manualnych testów, poprzez zastępowanie ich skryptami. Automatyzacja może odciągnąć testerów od żmudnego wykonywania „klikanych” testów i umożliwić zagospodarowanie ich czasu na inne zadania.

2.2.2. Wymagania odpowiedniej automatyzacji

Na podstawie wykazanych poniżej wymagań, można zauważyć, iż sam proces automatyzacji wskazuje na dobrą praktykę zastosowania środowiska testowego.

Pomijając wcześniej wspomnianego opiekuna testów, automatyzacja wymaga czterech kluczowych czynników. Pierwszym z nich jest odpowiednia infrastruktura techniczna, w ramach której możliwe będzie odpowiednie utrzymywanie automatyzacji i odwzorowanie wymagań do przeprowadzenia testów. Drugim istotnym aspektem jest zapewnianie danych niezbędnych do prawidłowego wykonywania skryptów. Cały przebieg dobrze przeprowadzonej automatyzacji, dane wejściowe oraz wyjściowe muszą być odpowiednio dokumentowane i archiwizowane. Ostatnim wymaganiem jest odpowiedni dobór narzędzi wspierających automatyczne procesy testowe. Dostępnych technologii i narzędzi istnieje ogromna ilość, dlatego powinno się wyraźnie sprecyzować co w danym projekcie mogłoby być przetestowane przez „automaty”.

2,2.3. Co powinno się automatyzować?

Automaty najlepiej sprawdzają się w dwóch rodzajach testów: w testach wydajnościowych- (polegających na obciążaniu zasobów systemu, w celu zbadania jego wydajności przy pracy na zapewnionym obciążeniu) oraz regresywnych. Przyczyna automatyzacji tych drugich wynika z samej przedstawionej definicji automatyzacji – dotyczy ona przypadków powtarzalnych.

Interfejsy użytkownika to część projektu z którą tester manualny zazwyczaj ma dużo pracy, wykonując kolejne kroki scenariuszy testowych. Automatyzacja w takich przypadkach testowych sprawdza się znakomicie. Bazując na tym co aplikacja internetowa oferuje użytkownikowi – czyli najczęściej elementy html, możemy utworzyć skrypty automatyczne „przeklikujące” wizualną część aplikacji według narzuconych reguł i scenariuszy testowych.

2.2.4. Testowanie manualne aplikacji internetowych

Projektowanie stron internetowych stawia przed zespołami tworzącymi takie projekty wiele dylematów. Również związanych z odpowiednim testowaniem ich produktów. Przede wszystkim oprogramowanie, które tworzą, powinno być dostępne na różnych przeglądarkach, platformach i do tego działać jak najwydajniej. Testowanie takich aplikacji staje się dużym wyzwaniem, szczególnie przy dużych komercyjnych projektach. Bardzo często oprócz oczywistych aspektów jakościowych strony internetowej, takich jak jej wydajność czy szybkość weryfikowane jest również samo “działanie” konkretnych funkcjonalności. Najbardziej popularnym podejściem do sprawdzania prawidłowego, bądź nieprawidłowego rezultatu takich funkcjonalności, są testy manualne, polegające najczęściej na zwykłym i żmudnym “przeklikiwaniu” aplikacji. Oczywiście takie testy również powinny być w odpowiedni sposób utrzymywane, tworzone i przeprowadzane. Tester, bądź zespół testerski zajmuje się przygotowaniem przypadków testowych- tworzonych zazwyczaj przy świeżo zaimplementowanej funkcjonalności, lub w celu uzupełnienia braków w testach. *Przypadek testowy* jest to zbiór danych związanych z przeprowadzeniem testu, zawiera dane wejściowe, wstępne warunki, kroki do wykonania i oczekiwane rezultaty końcowe. Przykładowo kilka przypadków testowych weryfikuje tak zwany *warunek testowy(*funkcjonalność systemu). Kluczowym zadaniem testera manualnego jest odpowiednie tworzenie *scenariuszy testowych*- zbioru przypadków testowych(kroków) potrzebnych do sprawdzenia warunku testowego. Znanych także jako skrypty testowe. Aby prawidłowo i sprawnie zarządzać testami manualnymi, niezbędna jest ich prawidłowa identyfikacja - najczęściej poprzez nazwę i identyfikator scenariusza. Przyjęta konwencja w nazewnictwie testu zależy głównie od osoby zajmującej się nimi, ale także od przyjętych reguł w ramach projektu czy organizacji. Scenariusz powinien zawierać również spis czynności przygotowawczych - zbiór konfiguracji i danych wejściowych a także czynności weryfikacyjnych, które tester powinien zapewnić i przeprowadzić przed wykonaniem scenariusza. Oprócz zbioru kroków do przeprowadzenia, w celu zweryfikowania czy funkcjonalność działa poprawnie czy niepoprawnie, dobrze napisany scenariusz posiada także zdefiniowane czynności końcowe. Odpowiednie grupowanie testów ma także ogromne znaczenie. W rozrastającym się projekcie zazwyczaj chodzi o zachowanie maksymalnej prostoty w zarządzanych zasobach, w tym również w testach manualnych. Podział takich testów może być uzależniony przykładowo od iteracji w metodologii zwinnej, jak również od weryfikowanej funkcjonalności. Zależy to przede wszystkim od osoby zajmującej się zarządzaniem takich zasobów, przyjętych wzorców i ustalonych reguł. Egzekwowanie testów musi również być prawidłowo zaplanowane. Zazwyczaj testy są wykonywane na podstawie priorytetyzacji ustalanej w fazie planowania. Przy nadawaniu priorytetów tester musi mieć na uwadze kilka kluczowych aspektów. Niedoskonałości w projektach występują najczęściej tam, gdzie istnieją najbardziej złożone funkcjonalności, dlatego zazwyczaj w takich działaniach, takie moduły obejmowane są jako pierwsze. Z punktu widzenia biznesowego planowanie nie tylko całego projektu, ale także przeprowadzanych testów ma również duże znaczenie, często o priorytetyzacji decyduje widoczność funkcjonalności dla klienta- przykładowo przy prezentacjach, czy chociażby wpływ działania modułu na finanse. Warto także priorytetyzować testy zważając na jej wcześniejsze modyfikacje, tworzenie ich pod presją czasu czy występowania problemów. Pomimo wykorzystywania wielu narzędzi wspierających i dużej wartości dla projektu, testy manualne bywają żmudnym i monotonnym zadaniem. Bardzo często zdarza się, iż trzeba je regularnie powtarzać, co komplikuje pracę testera. Z biznesowego punktu widzenia prawidłowe działanie aplikacji jest jej najważniejszym aspektem, co sprawia, że takie testowanie staje się ważnym etapem jej rozwoju i zapewniania jakości klientowi.[[8]](#footnote-8) Aby “zastąpić” pracę testera manualnego i oddelegować go do innych czynności, które mogą być wykonywane w tym samym czasie, powstały narzędzia wspierające testowanie manualne. W gruncie rzeczy automatyzacja testów manualnych stron internetowych, opiera się na zastąpieniu klikania elementów dostępnych na stronie przez testera, na automatyczne klikanie poprzez wykonywanie instrukcji zdefiniowanych w kodzie skryptu automatycznego. Na przestrzeni ostatnich lat powstały narzędzia, wspierające testera do przeprowadzania automatyzacji.

**3. NARZĘDZIE SELENIUM I INNE NARZĘDZIA AUTOMATYZACJI TESTÓW**

**3.1. Selenium**

3.1.1. Komponenty Frameworka Selenium

**Selenium WebDriver** jest narzędziem służącym do automatyzacji przeglądarek internetowych. Używany przede wszystkim do testowania aplikacji, jednak posiada dużo szerszy zakres możliwości. Między innymi stosuje się go również do techniki screem scraping oraz do automatyzacji zadań w przeglądarkach internetowych. Selenium obsługuje najbardziej znane przeglądarki, takie jak: Chrome, Opera, Mozilla Firefox, Safari i Internet Explorer. Jest to darmowe narzędzie, stworzone jako open-source project. Posiada wsparcie różnych systemów operacyjnych i obsługuje kilka języków programowania, takich jak: C#. Java, Python czy Ruby. Oprócz WebDriver, Selenium posiada kilka innych dodatkowych elementów:[[9]](#footnote-9)

**Selenium Grid** jest to narzędzie umożliwiające organizację testów na kilku środowiskach – kilku komputerach podłączonych do siatki. Pozwala tym samym na przetestowanie skryptów na kilku różnych konfiguracjach jednocześnie. [[10]](#footnote-10)

**Selenium IDE -** Ten moduł występuje jako dodatek do przeglądarki Mozilla Firefox. Umożliwia nagrywanie oraz odtwarzanie akcji użytkownika. Nie jest jednak już wspierana od wersji 55.00 przeglądarki.

**Selenium RC** - Narzędzie znane jako Selenium 1.0, umożliwiający odtwarzanie testów na zewnętrznym serwerze, ostatecznie wyparty przez nowe Selenium 2.0 – Selenium WebDriver

3.1.2. Działanie Selenium WebDriver

Nowoczesne aplikacje internetowe są pisane za pomocą przeróżnych frameworków, wzorców, języków i narzędzi. Jednak to, co przeglądarka prezentuje użytkownikowi, ma zawsze podobny format, wykorzystujący templatkę HTML, style strony CSS oraz najczęściej kod Java Script. Użytkownik wybierając konkretne opcje na stronie, przechodząc przykładowo pod wskazany adres URL wysyła zapytanie do serwera WWW, który dane żądanie przetwarza i zwraca odpowiedz prezentowaną użytkownikowi właśnie jako połączenie HTML, CSS i JS. Cała filozofia automatyzacji działania przeglądarki za pomocą Selenium Webdriver, polega głównie na poinformowaniu Frameworka o tym, jak znaleźć element na stronie wykorzystując jej kod źródłowy, skrypty stworzone przez testera oraz jak zasymulować działania użytkownika. WebDriver działa jako pośrednik pomiędzy przeglądarką a wykonywanym skryptem, a komunikację między tymi modułami, umożliwia SeleniumAPI.[[11]](#footnote-11)  
  
3,1,3. Framework w kontekście testowania

Gdzie testowanie z wykorzystywaniem Framework Selenium znajduje się w klasyfikacji testów? Często są wykorzystywane jako testy regresywne, sprawdzające działanie oprogramowania po zmianach. Mogą też być traktowane jako testy potwierdzające, po naprawieniu defektu przez programistów. Są to oczywiście testy czarno skrzynkowe – oparte na interfejsie użytkownika w przeglądarce – nie wnika również w kod aplikacji a jedynie w templatkę html i zasoby z nią związane, udostępnione przez aplikację.

3.1.4. Plan działania skryptów Selenium

Przykładowy skrypt oparty na Selenium z wykorzystaniem jednego z dostępnych języków programowania, zawiera kilka podstawowych czynności. Pierwszą z nich jest wyszukiwanie elementów na stronie – Selenium WebDriver oferuje wiele opcji znalezienia takiego elementu, jednak w zależności od specyfikacji testu i jego wymagań, używa się różnych mianowników wyszukiwania. Następnie za pomocą metod udostępnionych przez narzędzie, oraz tych stworzonych przez programistę lub testera, w zależności od potrzeby, przeprowadzane są akcje na wybranych elementach – działania użytkownika zakodowanego przez kodera. Konkretne czynności wykonywane przez skrypt wykonywane są do czasu oczekiwanego rezultatu.

3.1.5. Wyszukiwanie elementów przez Selenium

Przeglądarka internetowa udostępnia użytkownikowi drzewko modelu DOM(Document Object Model). W takim dokumencie znajduje się wszystko, czego programista czy tester automatyzujący potrzebuje do znalezienia danego elementu na stronie. Osoba kodująca może dostać się do całego modelu DOM przykładowo przy użyciu narzędzi developerskich w przeglądarce, lub za pomocą zewnętrznych narzędzi, takich jak Firebug – dla przeglądarki Mozilla Firefox. Selenium WebDriver API – narzędzie umożliwiające odwoływanie się do wybranych elementów z poziomu kodu wyszukuje je za pomocą metody „fndElements()” lub „findElement()”. Jako argumenty funkcja ta przyjmuje właściwości dokumentu DOM jako egzemplarz klasy „By” – dostarczanej przez WebDriver, dzięki którym możliwe jest odnalezienie elementu poprzez przekazany lokalizator – metodę klasy By, między innymi: „id, class, xpath, cssSelector, linkText, partialLinkText,”. Metody „find()” zwracają obiekt, lub obiekty typu WebElement.[[12]](#footnote-12)

|  |  |
| --- | --- |
| Język | Składnia |
| Java | driver.findElement(wyrażenie) |
| C# | driver.findElement(wyrażenie) |
| Python | driver.find\_element\_wyrażenie |
| Ruby | driver.find\_element(wyrażenie) |
| PHP | $driver->findElement(wyrażenie) |

Tabela 1. Metoda FindElement() w poszczególnych językach programowania.

**3.2.Wyszukiwanie elementów za pomocą kodu źródłowego**

3.2.1. Metoda find() i przyjmowane przez nią agrumenty

**Atrybuty Id** - nadawane przez programistów do znaczników html są to najszybsze lokalizatory elementów, jednak labelowanie każdego elementu za pomocą identyfiakorów jest niepraktyczne z punktu programowania frontendowego – wiele elementów generowanych jest dynamicznie. Często wykorzystuje się je przy niepowtarzających się elementach, mających duże znaczenie w aplikacji internetowej. W parametrze metody findElement, przekazuje się metodę „Id” klasy „By” – w ten sposób Selenium wie, że chodzi właśnie o szukany element na podstawie ID.[[13]](#footnote-13)

driver.FindElement(By.Id("<Id elementu>"));

**Atrybut nazwy znacznika – TagName** **-**  Mogłoby się wydawać, że wyszukiwanie elementów tylko przy pomocy nazwy znacznika jest bez senesu. Nic bardziej mylnego. Mimo iż powtarzanie się znaczników html w aplikacjach internetowych, jest techniką powszechnie stosowaną i nie da się wyszukać jednego jedynego elementu o unikalnym znaczniku. Atrybuty nazwy znacznika znakomicie sprawdzają się przy „wyciąganiu” tabel i właśnie powtarzających się elementów, nie dokładnie przy wywoływaniu metody findElement(), ale findElements().[[14]](#footnote-14)

driver.FindElement(By.tagName ("<TagName elementu>"));

**Atrybut wyrażenia Xpath -** Język Xpath, to język bazujący na ścieżkach XML, w przypadku stron webowych wykorzystuje on formę drzewa. Możliwe jest używanie ścieżek bezwzględnych – operujących na całej hierarchii DOM.Innym ciekawym podejściem jest używanie ścieżki względnej, za pomocą zależności w elementach drzewa. Na uwagę również zasługuje wykorzystywanie predykatów – elementów w nawiasach klamrowych, które w Xpath, służą do wyszukiwania określonego węzła, lub węzłów o podanej wartości. Podane powyżej przykłady „wyciągania” elementów ze strony, zostały przedstawione na najbliższej tabeli. Wykorzystywanie XPath’ów jest bardzo przydatnym zabiegiem w skryptach testowych, istnieje bardzo dużo możliwości dopasowania określonej wartości elementu, właśnie poprzez kombinację w Xpath. Programista aby odwołać się do ścieżki Xpath danego elementu, z kodu używa znów klasy By i metody Xpath. [[15]](#footnote-15)

driver.FindElement(By.XPath ("<Xpath elementu>"));

**Atrybut selektora CSS (CssSelector) -** Arkusze CSS zostały wprowadzone w celu oddzielenia szkieletu i wartości prezentacji witryny internetowej od jej wyglądu. Przy pomocy składni CSS definiuje się, jak dany element ma wyglądać na stronie. Znajdywanie wybranych elementów strony, poprzez selektor CSS, może być oparte na prostych właściwościach jak i w przypadku Xpath na bardziej złożonych wyrażeniach. Framework Selenium WebDriver używa klasy „By” i metody „cssSelector” i również operuje na drzewie DOM, kontekst w ramach którego ma zostać wyszukany dany element, zależy tylko od osoby kodującej i przyjętego wzorca. Poniżej przedstawiono instrukcje kodu wyszukujące poszczególne elementy, za pomocą wyrażenia XPath i selektora CSS. [[16]](#footnote-16)

driver.FindElement(By.CssSelector ("<CssSelector elementu>"));

**Atrybut tekstu elementu (LinkText, PartialLinkText) -** Element na stronie jest wyszukiwany przez WebDriver przy pomocy zawieranego wewnątrz tekstu.W przypadku Partial Link Text, Selenium wyszukuje element na podstawie przyjętej części tekstu danego elementu.[[17]](#footnote-17)

driver.FindElement(By.LinkText ("<Text elementu>"));

driver.FindElement(By.PartialLinkText("<PartialLinkText elementu>"));

**Atrybut nazwy elementu (Name)** - Często zdarza się, iż elementu w drzewie DOM posiadają atrybut „name”, za jego pomocą również możliwe jest odpowiednie wyszukanie wybranego elementu. [[18]](#footnote-18)

driver.FindElement(By.Name ("<Name elementu>"));

3.2.2 Dobre praktyki wyszukiwania elementów

Przede wszystkim wyszukiwanie elementów w skryptach testowych musi być odpowiednio prosto i przejrzyście zdefiniowane. W ramach przyjętych wzorców, nazewnictwa w skryptach i tym podobnych, przyjętych przez programistę zależności. Po drugie, powinno się unikać szukania elementów poprzez ścieżki bezwzględne, takie wyszukiwanie jest dużo bardziej czasochłonne, szczególnie w przypadku wyrażeń Xpath.

**3.3.Metody na wyszukanych elementach**

3.3.1. Obsługa elementów za pomocą Selenium Web Driver API

Framework Selenium dostarcza bardzo bogate API, umożliwiające szereg operacji obsługujących wybrane elementy HTML. Zawiera rozwiązania dla różnego rodzaju kontrolek, menu, tabel, elementów interaktywnych na stronach internetowych, ponadto wykonywania kodu JavaScript. Poniżej zaprezentowano operacje na poszczególnych elementach HTML, z wykorzystaniem metod automatyzacji Selenium.

**Klikanie elementu -** Podstawową operacją na stronie wykonywaną przez użytkownika jest klikanie elementu, Selenium udostępnia metodę „click()” dzięki której, możliwe jest automatyzowanie takiego zachowania. Klikany element musi być widoczny na stronie i mieć szerokość większą niż 0.

**Automatyzacja pól tekstowych** - Pola tekstowe są jednymi z najczęściej powtarzających się elementów witryn internetowych. Selenium WebDriver posiada kilka kluczowych metod służących do pracy z obiektami tego typu, między innymi:

* Usuwanie tekstu przy pomocy metody **clear()** – metoda działa na elementach typu „<input>” reprezentującej obszary tekstowe, poprzez jej wywołanie pole jest czyszczone
* Wprowadzanie tekstu metodą **sendKeys()** – możliwe jest użycie podanej metody na każdym elemencie, który akceptuje wpisywanie wartości, jako argument przekazuje się tekst przekazywany do elementu
* Symulowanie naciśnięcia klawiszy dzięki **sendKeys()** – dodatkowo funkcja posiada możliwość automatyzowania zachowania klawiatury,
* Pobieranie tekstu za pomocą **getText()** – zwraca zawartość właściwości innerText danego elementu[[19]](#footnote-19)

**Sprawdzanie wartości atrybutów elementu i wartości CSS** Zarządzanie właściwościami danego elementu jest bardzo szeroko używane w aplikacjach internetowych, głównie z perspektywy kodu, dany element może zmieniać swoje atrybuty i wartości w ramach innych działań użytkownika. Toteż w Selenium, nie może zabraknąć metod związanych z pobieraniem i weryfikowaniem danych atrybutów.

* Metoda **getAttribute()** pobiera atrybuty danego elementu, gdzie w argumencie funkcji przekazywany jest typ atrybutu
* Natomiast **getCSSValue()** pobiera wartości CSS wybranego fragmentu strony, również na podstawie przekazywanego parametru, określającego, jaka wartość powinna być wyciągnięta [[20]](#footnote-20)

**Automatyzacja rozwijanych menu oraz list -** Rozwijane menu oraz listy, stworzone za pomocą znaczników „<select>”, jako iż są nieodzowną częścią wielu aplikacji internetowych – znalazły w Selenium swoją własną klasę „Select” reprezentującą takie obiekty, oraz dostarczającą metody do interakcji. Innymi słowy klasa Select w tym wypadku zastępuje wyszukany obiekt na stronie, zamiast typu WebElement, używa się właśnie Select:   
Select specialMenu = new Select(driver.FindElement(By.Id(„SpecialMenu”)));

* Za pomocą metody **isMutiple()**, możliwa jest weryfikacja czy element, rzeczywiście ma kilka opcji do wyboru
* Metody takie jak **selectByValue()**, czy **selectByIndex()** ułatwiają automatyzację wyboru konkretnych wartości
* Możliwe jest skorzystanie również z takich funkcji jak **getAllSelectedOptions()** – wyciągającej ze strony wszystkie wybrane opjce itp.

**Automatyzowanie pól wyboru -** Selenium WebDriver Framework obsługuje również pola wyboru, zaznaczanie i odznaczanie takich pól możliwe jest za pomocą metody click(), oprócz której możliwe jest także zweryfikowanie czy danych element jest zaznaczony, przy pomocy funkcji isSelected().

**Sprawdzanie istnienia i stanu elementu -** Selenium oferuje testerowi trzy główne metody do wykorzystania w skryptach testowych, sprawdzające stan danego elementu na stronie. Poniższe funkcje są szeroko używane w testowym kodzie, szczególnie w instrukcjach warunkowych:

* **isEnabled()** – metoda sprawdzająca czy fragment strony jest aktywny, w pozytywnym przypadku zwraca true, w negatywnym false
* **isSelected()** – funkcja weryfikująca zaznaczenie danego elementu, jeśli element jest zaznaczony - zwraca true, jeśli nie – fasle
* **isDisplayed()** – metoda sprawdzająca czy podany element jest wyświetlony

3.3.2. Automatyzacja zewnętrznych elementów

**Akcje urządzeń wejścia– wyjścia** - Moduł „Advanced User Interactions” frameworka Selenium pozwala na wykonywanie zarówno prostych, jak i bardziej złożonych akcji klawiatury oraz myszki. Przy pomocy klasy Actions możliwe jest zbudowanie łańcuchów zdarzeń imitujących działania użytkownika a następnie wykorzystanie ich w skrypcie testowym.

* Przy pomocy takich funkcji jak **keyDown()** – możliwe jest zainicjowanie kliknięcia przekazywanego w argumencie klawisza, a następnie zwolnienie go metodą **keyUp()**
* Intuicyjnie Selenium udostępnia również metodę podwójnego kliknięcia **doubleClick()**
* Istnieje także odwzorowanie zachowania przeciągnij-upuść(ang. drag and drop), inicjalizowane poprzez metodę **dragAndDrop(),** gdzie w argumentach przekazywany jest element początkowy i końcowy

**Obsługa menu kontekstowych** - Selenium oferuje użytkownikowi również inicjalizację akcji otwierania menu kontekstowego i wykonywania akcji w jego obrębie. W skryptach testowych istnieje możliwość wykorzystania metody **contextClick()**, służącej do obsługi właśnie tego elementu.

**Egzekwowanie kodu JavaScript -** Ddzisiejsze aplikacje internetowe, bynajmniej ich frontendowe części, są przepełnione kodem Java Script, wykonującym się w przeglądarkach po stronie klienta. WebDriver również umożliwia insynuowanie działania takiego kodu w przypadkach testowych. Poprzez interfejs „JavascriptExecutor”, możliwe jest wykonywanie „opakowanego” w nim kodu w kontekście przeglądarki.

**Zrzuty ekranu za pomocą Selenium** - Aby usprawnić wykonywanie testu i rozwiewać wszystkie domyślenia testera, co mogło być przyczyną niepowodzenia testu – Selenium dostarcza interfejs TakesScreenShot, przechwytujący zrzuty ekranu Wityn internetowych. Dzięki wykonywaniu takich zrzutów, dużo łatwiej zlokalizować przyczynę wystąpienia błędu. Przykładowo aby zwrócić screenshot w postaci pliku na dysku, możliwe jest wykorzystanie metody **getScreenshotAs().**

3.3.3. Automatyzacja działań przeglądarki

Działania użytkownika mogą również sprowadzać się do manipulacji oknem przeglądarki, czasami również niektóre funkcje na stronach internetowych automatycznie kierują użytkownika do nowo utworzonej karty itd. Selenium posiada szereg przydatnych funkcji do inicjalizacji takich działań w skryptach testowych.

* Metoda **maximize()** pozwala na maksymalizację okna przeglądarki
* Przy pomocy funkcji **back()** czy **forward()** możliwe jest przechodzenie pomiędzy ostatnimi stronami w historii
* Selenium udostępnia także metodę odświeżania – **refresh()**
* Oraz funcję **to()** – przyjmującą jako agrument adres url, pod który przeglądarka zostanie przekserowana [[21]](#footnote-21)

**Obsługa plików cookie sesji** -Ciasteczka są szeroko używanymi plikami w aplikacjach internetowych i służą do przechowywania preferencji użytkownika, szczegółów użytkowania przeglądarki, danych logowania i tym podobnych. Selenium WebDriver API udostępnia możliwości zarządzania cookie’sami z poziomu skryptu testowego:

* **addCookie()** – dodaje pliki cookie
* **getCookieNamed()** – zwraca plik cookie o podanej nazwie w argumencie funkcji
* **getCookies()** – zwraca wszystkie pliki cookie dla bieżącej domeny
* **deleteCookieNamed()** – usuwa plik cookie o podanej nazwie w parametrze
* **deleteCookie()** – usuwa plik cookie
* **deleteAllCookies()** – usuwa wszystkie pliki cookie dla bierzącej domeny[[22]](#footnote-22)

Powyżej zostały zaprezentowane przykładowe funkcje Selenium, których tester automatyzujący mógłby użyć w skryptach testowych. Jednak same metody udostępnione przez framework nie są w stanie w pełni odwzorować działań użytkownika na stronie, a co za tym idzie pokryć przypadków testowych. W związku z tym tester automatyzujący powinien łączyć owe funkcje z zależnościami zakodowanymi w ramach wymagań testowych. Bardzo ważną cechą osoby kodującej takie skrypty jest umiejętność wykorzystania zasobów programistycznych Selenium z własnym wytworzonym kodem, które w połączeniu mogą reprezentować działania realnego użytkownika na stronie. Wystarczy odpowiednio je przetestować przez co dostarczyć wartość biznesową do tworzonego projektu, oraz utrzymywać.

Bardzo często to ostatnie zagadnienie w projekcie przynosi najwięcej komplikacji – utrzymywanie. W przypadku testów Selenium, możemy ustrzec się usterek i niedoskonałości wynikających z błędnego utrzymywania już przy tworzeniu skryptów testowych. Chodzi tutaj głównie o tak zwaną „adaptywność” kodu – łatwość dostosowywania się do zmiennych wartości wokół niego, ale też jego prostotę. Doskonały wzorzec do tworzenia przejrzystych, prostych i łatwych w rozwoju skryptów testowych, również zapewnia Selenium- poprzez wykorzystywanie to tak zwanego modelu „Page Object”, który opisano w poniższym podrozdziale.

**3.4.Wzorzec Page Object**

3.4.1. Dlaczego Page Object?

W projektach informatycznych bardzo ważną częścią jest prawidłowe pisanie kodu – trzymanie się przyjętych konwencji, nazewnictwa, przyjętych wzorców projektowych, metodologii itp. Również w przypadku wytwarzania testów automatycznych, tworzenie łatwego kodu w utrzymaniu to jeden z głównych czynników powodzenia tej partii projektu i dostarczania odpowiedniej wartości biznesowej. Jako iż środowisko automatyzacji powinno reprezentować środowisko produkcyjne, bądź inne tego typu środowisko objęte w wymaganiach, kod używany w skryptach powinien również być zbliżony do produkcyjnego. Podczas tworzenia takiego kodu, powinny być używane standardy i wzorce wspierające w wytwarzaniu kodu tego typu oraz jego utrzymywaniu. Przykładem takie wzorca jest właśnie Page Object. Stosowanie tej techniki pomaga w zarządzaniu testami: zwiększa łatwość ich utrzymywania, zmniejsza ryzyko powtarzalności kodu, buduje warstwę abstrakcji oraz ukrywa wewnętrzne szczegóły implementacji. [[23]](#footnote-23)

3.4.2. Jak działa Page Object?

Wzorzec Page Object działa na podstawie dobrych praktyk programowania obiektowego. Podejście wykorzystuje tworzenie klas, jako abstrakcję dla pojedynczych stron, oraz reprezentacji jej właściwości, poprzez właściwości klasy, a zachowania użytkownika poprzez jej metody. Dzięki takiemu podejściu, skrypty testowe mogą być podzielone na 2 warstwy, warstwę reprezentacji strony i warstwę wykonawczą na obiektach warstwy reprezentacji, która zawierać będzie kroki testowe, przełożone na kod skryptu testowego. Dużym plusem tego podejścia jest ukrywanie implementacji całej abstrakcji strony. Gdyby skrypt testowy zawierałby zarówno instrukcje odpowiedzialne za znalezienie danego elementu, następne jego przetworzenie i weryfikację, czy przetworzenie przebiegło w pozytywny sposób, byłoby to bardzo czasochłonne, obfite w powtarzalne ilości kodu a także trudne do utrzymania. Za pomocą Page Object buduje się tylko interfejs danej strony a do jego części odwołuje się w ramach potrzeby testu. [[24]](#footnote-24)



Rysunek 5. Odwzorowanie strony logowania za pomocą Page Object. Źródło: Opracowanie własne na podstawie: Gundecha U., Selenium i testowanie aplikacji. Receptury, Gliwice 2017, s. 182.

3.4.3. Implementacja Page Object

Aby wprowadzić wzorzec Page Object w życie, potrzeba implementacji klasy reprezentującej rzeczywistą stronę internetową. Przykładowo przy automatyzacji testów strony logowania „Login Page”, tworzymy klasę „LoginPage” gdzie jako właściwości klasy definiujemy „loginButton” oraz metodę „ClickLoginButton()”, w której używa się metody udostępnionej przez Selenium – Click(), jako argument tej metody przekazując element html – button, obsługujący funkcję logowania.

**3.5.Inne narzędzia automatyzacji UI stron internetowych**

3.5.1.Katalon Studio

Katalon studio – darmowe narzędzie służące do automatyzacji testowania, między innymi aplikacji internetowych. Cały program jest bardziej systemem automatyzacji, w który należy opakować dany projekt. Umożliwia między innymi nagrywanie akcji użytkownika a następnie wykorzystanie nagrania do odtworzenia kroków testowych – tworzonych jako skrypt w języku Groovy lub Java. Jeśli chodzi o dane wejściowe, konieczne do przeprowadzenia testów środowisku, istnieje możliwość importu takich danych poprzez plik csv, lub przez plik bazodanowy. Program posiada możliwość uruchomienia równolegle od jednego do czterech testów. W przypadku inicjalizowania testów na zewnętrznym serwerze potrzebna jest instalacja dodatkowych elementów i ich konfiguracja. Katalon Studio umoóżliwia również raportowanie o przebiegu testów. Dużym plusem oprogramowania jest to, iż jest on narzędziem darmowym, w pełni rozwijanym i może być również bardzo opłacalnym wyborem w przeprowadzaniu automatyzacji. Katalon ogranicza niestety możliwość pisania skryptów testowych do dwóch języków programowania. Nie posiada również tak dużej społeczności, jak w przypadku Selenium.[[25]](#footnote-25)

3.5.2.Ranorex

Ranorex – jest to komercyjne(płatne) narzędzie do automatyzacji testów manualnych, w ramach aplikacji webowych, desktopowych oraz mobilnych. Przy rozpoznawaniu obiektów interfejsu użytkownika, bazuje na zmodyfikowanej pod własne oprogramowanie ścieżce Xpath. Między innymi posiada opcję tworzenia testów typu „Record and play”. Oprogramowanie posiada zaawansowany system raportowania wyników testów oraz możliwość korzystania z zewnętrznych danych testowych. Umożliwia także równoległe przeprowadzanie testów, przykładowo przypadki testowe mogą być przeprowadzane na urządzeniu mobilnym a weryfikowane w aplikacji webowej. Głównym aspektem przeciw wykorzystywaniu Ranorexa w automatyzacji jest jego wysoka cena oraz brak Multi-platformowej oferty – to narzędzie można wykorzystywać tylko w przypadku automatyzacji aplikacji desktopowych.

3.5.3. Dlaczego Selenium – krytyczna analiza

Jako uniwersalne narzędzie, które ma dość niski próg wejścia do automatyzacji testów webowych, Selenium wydaje się najrozsądniejszym wyborem. Przede wszystkim dlatego, iż jest to darmowe oprogramowanie i koszty związane z jego wdrożeniem, związane są tylko z wyszkoleniem odpowiedniej osoby oraz zasobami zewnętrznymi środowiska – przykładowo z serwerem oraz oprogramowaniem zarządzającym scenariuszami. Nie mniej jednak, przyjęte koszta zależą tylko od przyjętych reguł i założeń. Kolejnym atutem Selenium jest jego społeczność, która kształtowała się na przestrzeni ostatnich lat. Automatyzacja za pomocą tego narzędzia jest możliwa w kilku głównych językach programowania, wobec tego programista czy tester związany z konkretnym językiem, prawdopodobnie nie będzie musiał się „przesiadać” na inny. Co do samego kodu źródłowego aplikacji, Selenium to tak naprawdę zbiór dostępnych usług, które dowolnie można oprogramować. Dzięki temu, środowisko testowe, może być stworzone w pełni pod wymagania projektowe. Nie chodzi bowiem o to, aby testerzy byli ograniczani tylko do jednego narzędzia i automatyzacji w jego w obrębie, lecz o dostosowywanie narzędzi pod swoje potrzeby. Oczywiście Selenium posiada również wady. W źródłach internetowych możliwe jest znalezienie wielu opinii o ciężkim utrzymywaniu automatyzacji przy jego pomocy, jednak są one wynikiem najczęściej nieprzemyślanych reguł, niejasno sprecyzowanych i źle lub nawet wcale nie udokumentowanych założeń procesu automatyzacji. Za pomocą prezentowanego Selenium, możliwe jest niestety tylko testowanie aplikacji webowych. Akurat w tym aspekcie konkurencja ma dużą przewagę, pozwalającą na testy desktopowe i mobilne. Pomimo kilku istotnych wad, prezentowa narzędzie jest jednak najbardziej optymalnym wyborem do wprowadzenia automatyzacji do projektu informatycznego.

**4.KONCEPCJA WŁASNEGO ŚRODOWISKA TESTOWEGO**

**4.1.Główne założenia środowiska testowego**

4.1.1. Założenia testowania

Przedstawiane środowisko testowe jest projektem, przede wszystkim uniwersalnym. Ze względu na duże koszta przeprowadzania, a w szczególności wdrażania automatyzacji w projektach informatycznych, środowisko zaprojektowane zostało w ten sposób, iż powinno dopasowywać się do realiów danego projektu. Innymi słowy ma wpasowywać się we wzorce, podejścia i przyjęte metodologie prowadzenia i wytwarzania takiego oprogramowania webowego. Prezentowa wersja środowiska nie jest utożsamiona z żadną metodologią projektową. Takie środowisko, powinno być punktem startowym we wdrażaniu automatyzacji. Bazuje ono tylko na elementach HTML, udostępnianych w automatyzowanej przeglądarce, odpowiednio odwzorowanych w kodzie.

**Testowanie w prezentowanym środowisku nie jest nastawione na znajdowanie awarii**  - Wykonywanie skryptów testowych służy potwierdzaniu, iż w projekcie nie nastąpił regres jakości. Testy mogą znajdywać awarie w projekcie, jednak nie jest to w ogóle ich celem. Środowisko może być wykorzystywane do testów regresyjnych, Re-testów – głównie do powtarzających się procesów testowania. [[26]](#footnote-26)

**Nie jest pokrywane 100% interfejsu użytkownika –** Pokrywanie w tak dużym stopniu, jest bardzo czasochłonne i nieopłacalne. Testy automatyczne UI, powinny stanowić najmniejszy odsetek wszystkich testów aplikacji. Kod źródłowy powinien być przetestowany przede wszystkim jednostkowo- w tym aspekcie pokrycie testowe powinno być największe. Sprawdzane ścieżki aplikacji w prezentowanym środowisku poprzez testy przy pomocy Selenium powinny być sprawdzonymi i ustabilizowanymi częściami projektu.[[27]](#footnote-27)

4.1.2. Założenia kodu źródłowego

**Rozszerzalność kodu-** środowisko testowe zostało zaprogramowane z myślą o jego rozszerzaniu w kolejnych etapach wytwarzania lub utrzymywania projektu. Kod środowiska obejmuje konieczne mechanizmy do inicjalizacji przeglądarki, wykonywania kroków testowych w jej obrębie, oraz uzupełniania warstwy prezentacji na podstawie rezultatów testów. Każdy moduł, zawierający kod źródłowy jest napisany w ten sposób, jakby miał być przejmowany i rozwijany przez innego programistę – użyto dobrych praktyk adaptywnego kodu.

**Zastosowanie wzorca Page Object**– w obrębie każdej testowanej strony html, jaką zawiera testowa aplikacja, została stworzona jej reprezentacja w postaci klasy w języku c# wykorzystującej Page Object Pattern. Pola takiej klasy odwzorowują elementy html potrzebne do przeprowadzenia danego testu. Mechanizmy wyszukiwania elementów zaimplementowane są w obrębie właśnie tych klas. Jeśli chodzi o przetwarzanie powtarzających się wyszukanych obiektów, metody służące do tego zostały utworzone w klasie bazowej po której dziedziczą klasy stron. Jeśli jakaś testowana witryna posiada unikatowe elementy, występujące tylko w jej obrębie, metody je przetwarzające zaimplementowane są w klasie odwzorowującej stronę.

**Unikatowe podejście do skryptów testowych –** Nazewnictwo metod testowych oparte jest na podejściu „Given-when- then”. Używana specyfikacja nazywania skryptów testowych ma kilka solidnych reguł, które zostały zachowane przy ich tworzeniu. jako wartość „given” podaje się początkowe warunki(założenia) przeprowadzanego testu.. Pomiędzy „when” a „then” została specyfikowana czynność testowa, która zostanie wykonana w ramach tego przypadku. Natomiast na końcu nazwy testu, zawarto oczekiwany jego rezultat. Kod źródłowy w środku metody testującej, opiera się na inicjalizacji przeglądarki, korzystania z zasobów klas stron – metod i pól, oraz aktualizacji wyników testu.

4.1.3. Założenia scenariuszy testowych i prezentacji

**Scenariusze testowe** są modułem wejściowym prezentowanego środowiska. Skrypty testowe powstają na ich podstawie i za ich pomocą definiowane są wszystkie kroki testowe oraz potrzebne dane do wykonania testu. W scenariuszach również używane jest podejście nazewnictwa z skryptów testowych środowiska - „Given-when-then”. Ta część projektu również została stworzona z myślą o jej rozszerzalności. W ramach rozwoju testowanej aplikacji, może zajść potrzeba automatyzacji większej ilości testów, w związku z tym, większej ilości scenariuszy, dlatego w prezentowanym module zachowano prostotę w jego wyglądzie i zarządzaniu.

**Warstwa prezentacji** to zintegrowany moduł ze scenariuszami testowymi. Ma za zadanie jak sama nazwa wskazuje- prezentować rezultaty przeprowadzanych testów automatycznych w obrębie środowiska. Znajdują się w nim wykresy ilustrujące skutki testowania i wszelkie informacje o przeprowadzanej automatyzacji oraz konfiguracji środowiska testowego.

4.1.4. Założenia wykorzystywanych zewnętrznych komponentów

Egzekucja testów może być przeprowadzana na wszystkich przeglądarkach, jakie można wykorzystać w obrębie narzędzia Selenium. Testowa aplikacja będzie projektem webowym, który udostępnia środowisku tylko to, co widoczne w przeglądarce. Ze względu na różne ścieżki poszukiwania elementów przez środowisko, w aplikacji użyto szeregu różnych elementów, których zlokalizowanie i przetworzenie wymaga nie tylko użycia metod Selenium, ale również własnego kodu obsługującego znalezione obiekty. Przeprowadzane testy mają charakter testów czarno skrzynkowych – bazują tylko na elementach HTML, CSS i JS z przeglądarki, które udostępnia testowa aplikacja. Środowisko testowe stosuje podejście Page Object do reprezentacji zewnętrznych elementów aplikacji a następnie przetwarza je w skrypcie testowym. Testy wykonywane są na specjalnie wyznaczonym serwerze, w prezentowanej wersji środowiska, użyto Selenium Server, jako narzędzia do egzekucji testów, inicjalizowanego na lokalnej maszynie.



Rysunek 6. Środowisko testowe i jego składowe. Źródło: Opracowanie własne.

**4.2. Role poszczególnych modułów środowiska**

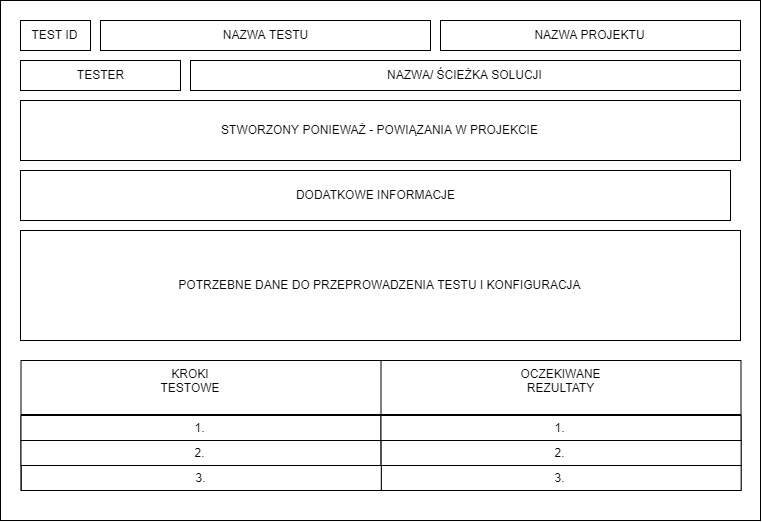
4.2.1. Moduł dokumentacji

Pierwszy z nich – moduł dokumentacji, jest to część projektu odpowiedzialna za szczegółowy opis specyfikacji całego środowiska i połączenia go z projektem. Jest to bardzo ważny element wdrażania tego typu części projektu, jaką jest automatyzacja. Powinna ona jasno i rzetelnie opisywać wszystkie technologie, narzędzia i części środowiska, używane w ramach automatyzowania. Przede wszystkim istotną rolę odgrywa tutaj sposób powiązania środowiska automatycznego z projektem testowanym, mimo tego iż, środowisko testowe jest w pewnym sensie oddzielną partią projektu i posiada osobną dokumentację. Moduł opisowy zawiera opis specyfikacji komunikacji aplikacji utworzonych w ramach środowiska testowego, a także współpracy pomiędzy tworzonymi aplikacjami oraz narzędziami. Między innymi opisuje działania zewnętrznego serwera lub serwerów, na których przeprowadzane będą testy. Definiuje architekturę całego systemu- interakcję pomiędzy skryptami a serwerem, modułem raportowania, modułem prezentacji oraz modułem zapewniania danych testowych. Zawiera również informacje na temat nazewnictwa w scenariuszach testowych i skryptach testowych – całej konwencji przyjętej przez programistów i testerów w ramach tworzenia tej części projektu. Jeśli chodzi o zależności pomiędzy testowanymi aplikacjami a samym środowiskiem, powinny one również być odpowiednio udokumentowane. Środowisko testowe wymaga podobieństwa do środowiska produkcyjnego(docelowego). W związku z tym, konieczne może być kopiowanie wersji aplikacji i ich zasobów ze środowiska produkcyjnego, co przede wszystkim powinno być kompatybilnie udokumentowane. Jak wspominano wcześniej środowisko testowe powinno być w pełni niezależną częścią projektu, a jego zasoby wykorzystywane tylko w jego ramach. Dokumentacja automatyzacji powinna być przeglądana i rzetelnie aktualizowana przez osoby za nią odpowiedzialne. Bardzo ważnym aspektem jest skuteczne utrzymywanie procesów automatyzacji w projekcie i nie utrudnianie tym samym pracy testerów i programistów, w tym znakomicie przydaje się dokumentacja.

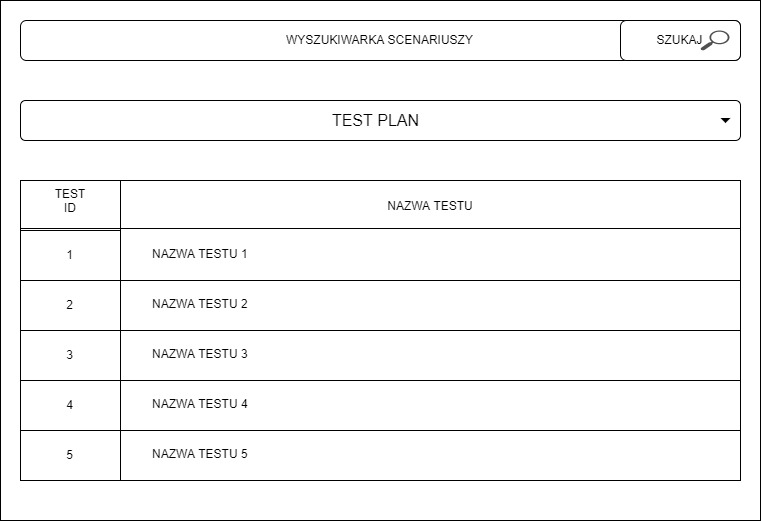
4.2.2. Moduł scenariuszy testowych

Kolejną częścią środowiska są scenariusze testowe ta część środowiska jest w pewnym sensie jego punktem wejściowym. W celu dokonania automatyzacji testowania manualnego, scenariusze testowe są jedną z istotniejszych kwestii powodzenia tego procesu. W prezentowanym środowisku takie harmonogramy testowania są opisową reprezentacją skryptu testowego, na ich podstawie wytwarzany jest kod testujący. Moduł scenariuszy, jest intuicyjnie połączony z warstwą prezentacji, gdzie prezentowane są rezultaty danego skryptu, a co za tym idzie - również scenariusza testowego. Takie podejście znacznie ułatwia poruszanie się w obrębie kolejnych warstw reprezentacji testów.

Scenariusz testowy dzieli się na kilka części. Pierwszą z nich jest częścią organizacyjną, gdzie między innymi została sprecyzowana nazwa projektu oraz solucji, której funkcjonalność jest weryfikowana a także znajduje się tam informacja jaka osoba napisała scenariusz. Test posiada również unikatowy numer ID, co pomaga w jego zarządzaniu. Jeśli dana słowna reprezentacja testu, została stworzona w celu przetestowania jakiegoś naprawionego baga lub w przypadku weryfikacji działania nowej funkcjonalności, również takie informacje zostaną zawarte w części organizacyjnej scenariuszu. Między innymi z tego powodu prowadzenie rzetelnego wzorca nazewnictwa oraz oznaczanie elementów projektu elementami id jest bardzo ważnym elementem zachowania dobrze zorganizowanej struktury środowiska. W przypadku prezentowanego środowiska testowego, jako wzorzec nazewnictwa scenariuszy testowych, a potem ich automatyzowanej reprezentacji wybrano podejście ”Given-when- then”. Przykładowy test weryfikujący logowanie na stronie internetowej, zdefiniowany za pomocą takiego podejścia, posiadałby następującą nazwę: „Given home page opened, when login using valid credentials, then login is successful”. Kolejną częścią scenariusza jest sekcja zapewnianych danych, potrzebnych do przeprowadzania testu. Bardzo często w projektach informatycznych, dopiero w krokach testowych, często jako pierwsze z nich, zdefiniowane są dane, na podstawie których możliwe jest przeprowadzenie testu, czy też jego wymagana konfiguracja. W prezentowanym podejściu, konfiguracja potrzebna do zapewnienia wraz ze spisem wszystkich danych potrzebnych do pozytywnego rezultatu przypadku testowego, znajduje się w specjalnej, wydzielonej sekcji. Poprawia to przejrzystość scenariusza i jest łatwiej edytowalne, jako osobna część.



Rysunek 7. Model strony scenariusza testowego. Źródło: Opracowanie własne.



Rysunek 8. Model listy scenariuszy testowych. Źródło: Opracowanie własne.

4.2.3. Moduł aplikacji zarządzającej

**Połączenie Visual Studio i Selenium -** Główną częścią prezentowanego środowiska, jest aplikacja konsolowa nim zarządzająca, napisana w języku C#. Do projektu aplikacji, zostały dodane potrzebne pakiety NuGet, które udostępnia Selenium, między innymi „Selenium WebDriver”, „Selenium WebDriver Support Classes”, „WebDriver-backend Selenium”, „Selenium Remote Control”. Takie pakiety są rozszerzeniem do Visual Studio, pozwalającym na łatwiejsze zarządzanie referencjami do bibliotek. Dzięki takiemu podejściu, wystarczy że do kodu źródłowego zostaną dodane odpowiednie przestrzenie nazw, poprzez ich deklaracje, za pomocą słowa kluczowego „using”[[28]](#footnote-28). Udostępnianych przez Selenium zasobów poprzez pakiety NuGet, używa się w obrębie całej aplikacji sterującej środowiskiem. W skrócie ta część projektu odpowiedzialna jest za cale „flow” wykonywania testów. Poniżej zaprezentowano, pozostałe kluczowe akcje występujące w obrębie projektu, którymi zajmuje się aplikacja zarządzająca.

**Inicjalizacja serwera wykonawczego** – moduł wykonawczy projektu jest sterowany i ustawiany poprzez aplikację zarządzającą, mimo tego, iż jest niezależnym miejscem do egzekucji testów. Taka koncepcja pozwala na łatwiejsze utrzymywanie zewnętrznych zasobów - takich jak serwer wykonawczy.

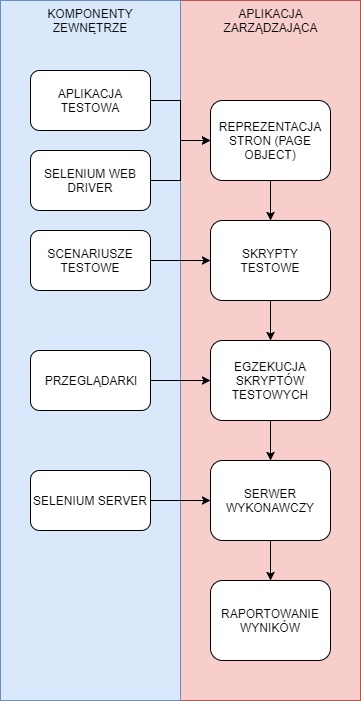
**Inicjalizacja przeglądarek –** przeglądarki, wykorzystywane w skryptach testowych, sterowane w nich przez Selenium WebDriver potrzebują również dodatkowych pakietów NuGet oraz ich obsługi przez klasy kodu źródłowego. Wszystkie klasy, zajmujące się takimi akcjami, również są częścią aplikacji zarządzającej.

**Reprezentacja stron Page Object** – W tej części, zostały zawarte klasy i wszystkie metody potrzebne do obsługi i przedstawienia elementów testowanych stron internetowych. Jest to moduł ściśle powiązany ze skryptami testowymi środowiska.

**Skrypty testowe** – Reprezentacja kroków testowych scenariuszy, przerobiona na kod źródłowy, wywołujący metody zdefiniowane poziom niżej - w klasach Page Object. Wywoływane funkcje, używane są chronologicznie – tak samo jak w krokach scenariuszy. Natomiast cały skrypt testowy, zawarty jest w testowej metodzie.

**Egzekwowanie metod testowych** – Cześć aplikacji odpowiedzialna za przeprowadzanie (wywoływanie) definiowanych skryptów, z użyciem danej konfiguracji, przykładowo – na przeglądarce Google Chrome.

**Zbieranie rezultatów z przeprowadzanych testów** – W tym module, przetwarzane są wszystkie wyniki z egzekucji testów i wysyłane do modułu prezentacji, gdzie zostają odpowiednio odwzorowane i zaprezentowane na wykresach.



Rysunek 9. Zestawienie zewnętrznych i wewnętrznych części środowiska. Źródło: Opracowanie własne.

4.2.4. Moduł prezentacji

Aby proces automatyzacji był procesem kompletnym, należy zadbać o jego odpowiednią prezentację. Często w wielkich korporacjach automatyzacja obejmuje wszystkie projekty i bardzo ważnym aspektem jest jej „widoczność” – czyli to, jak przebiega ona w danym projekcie. Odpowiednie odwzorowanie wyników i umiejętne ich przedstawienie ułatwia także zarządzanie procesem automatyzacji oraz wnioskowanie jej rezultatów. Głównym celem prezentacji takich wyników w odpowiedniej formie, jest potwierdzanie, iż automatyzacja testów w projekcie wygląda i działa bardzo dobrze, lub kiedy wykonywana jest w niepoprawny sposób – wymaga poprawek.

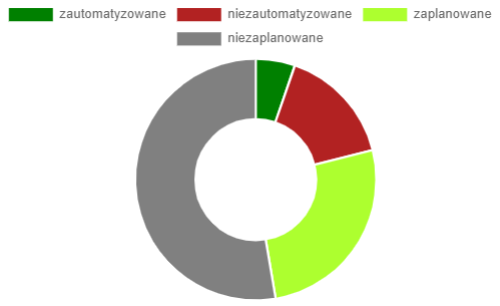
Warstwa prezentacji prezentowanego środowiska ma za zadanie odpowiednio odwzorowywać oraz przedstawiać rezultaty automatyzacji aplikacji testowej. Jest to część projektu zintegrowana z modułem scenariuszy testowych – ułatwia to przeglądanie konkretnych testów, pod warunkiem ich pozytywnego lub negatywnego rezultatu. Moduł prezentacji zawiera kilka wykresów oraz informacji dotyczących konfiguracji przeprowadzanych przypadków testowych. W prezentacji zawarto takie wykresy jak:

* Wykres rozwoju automatyzacji testów w projekcie, w przeciągu wybranych sprintów,



Rysunek 10. Automatyzacja w ostatnich sprintach. Źródło: Opracowanie własne.

* Wykres różnicy pomiędzy zautomatyzowaną i niezautomatyzowaną częścią testów,



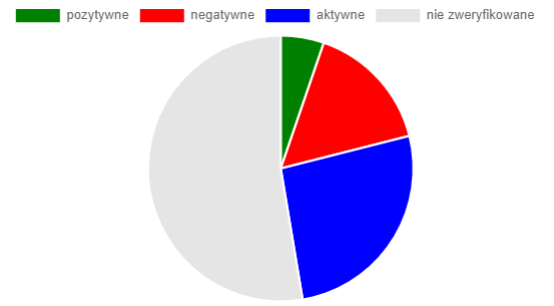
Rysunek 11. Różnica pomiędzy zautomatyzowaną a niezautomatyzowaną częścią testów. Źródło: Opracowanie własne.

* Wykres dziennego raportu automatyzacji testów,



Rysunek 12. Dzienny raport automatyzacji testów. Źródło: Opracowanie własne.

* Wykresy zautomatyzowanych testów z podziałem na części projektu.



Rysunek 13. Automatyzacja testów dla poszczególnej części projektu. Źródło: Opracowanie własne.

Prezentacja wyników została zaimplementowana z użyciem frameworka Java Script – Angular, wraz z użyciem biblioteki do tworzenia wykresów – Chart.JS. Kod tej części aplikacji jest również połączony z wynikami testów. Dzięki takiemu podejściu wyniki są automatycznie aktualizowane po przeprowadzanych testach.

4.2.5. Aplikacja testowa

Przy użyciu prezentowanego środowiska testowany jest projekt aplikacji webowej – strona internetowa, której tematyka porusza podstawy testów. Testowy projekt ma charakter osobistego bloga, w którym dowolny użytkownik może przykładowo dodawać komentarze pod jednym z wpisów, dodawanych przez autora. Aplikacja została stworzona przy pomocy popularnego narzędzia CMS – WordPress. Ponad 30 procent stron internetowych na świecie jest utworzone i skonfigurowane za pomocą właśnie tej technologii. Przy jej użyciu strona jest tworzona na podstawie dodatków i wzorów wykorzystujących rozmaite układy HTML. W związku z tym możliwe jest maksymalne zastosowanie zasobów Selenium, chociażby do znajdywania i przetwarzania konkretnych elementów – reprezentowanych poprzez wachlarz różnych znaczników.

W celu prezentacji działania środowiska według jego założeń, poniżej przedstawiono automatyzację testu sprawdzającego właśnie dodawanie komentarzy pod postem z wykorzystaniem narzędzi omawianego środowiska i jego założeń.

**4.3 Prezentacja wykorzystania środowiska na przykładzie testu dodającego komentarz**

4.3.1.Stworzenie scenariusza testowego

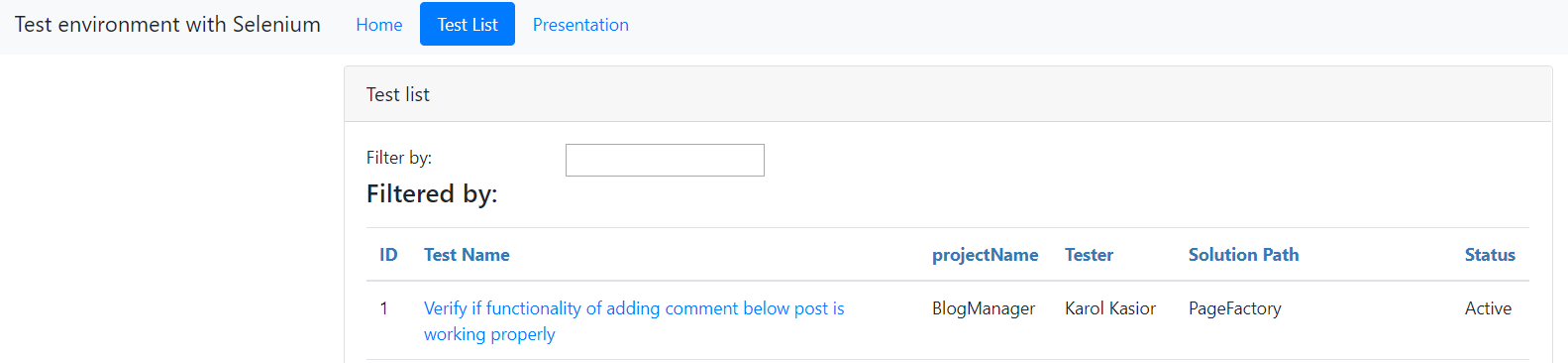
W podanym przykładzie założono, iż w projekcie nie istnieje scenariusz testu manualnego, weryfikujący funkcjonalność dodawania komentarzy. W związku z tym w pierwszej kolejności, należy zdefiniować niezbędne dane do udokumentowania testu oraz jego kroki testowe. W prezentowanym podejściu użyto języka Angielskiego, ze względu na obecne standardy panujące na rynku informatycznym.

Dane testu:

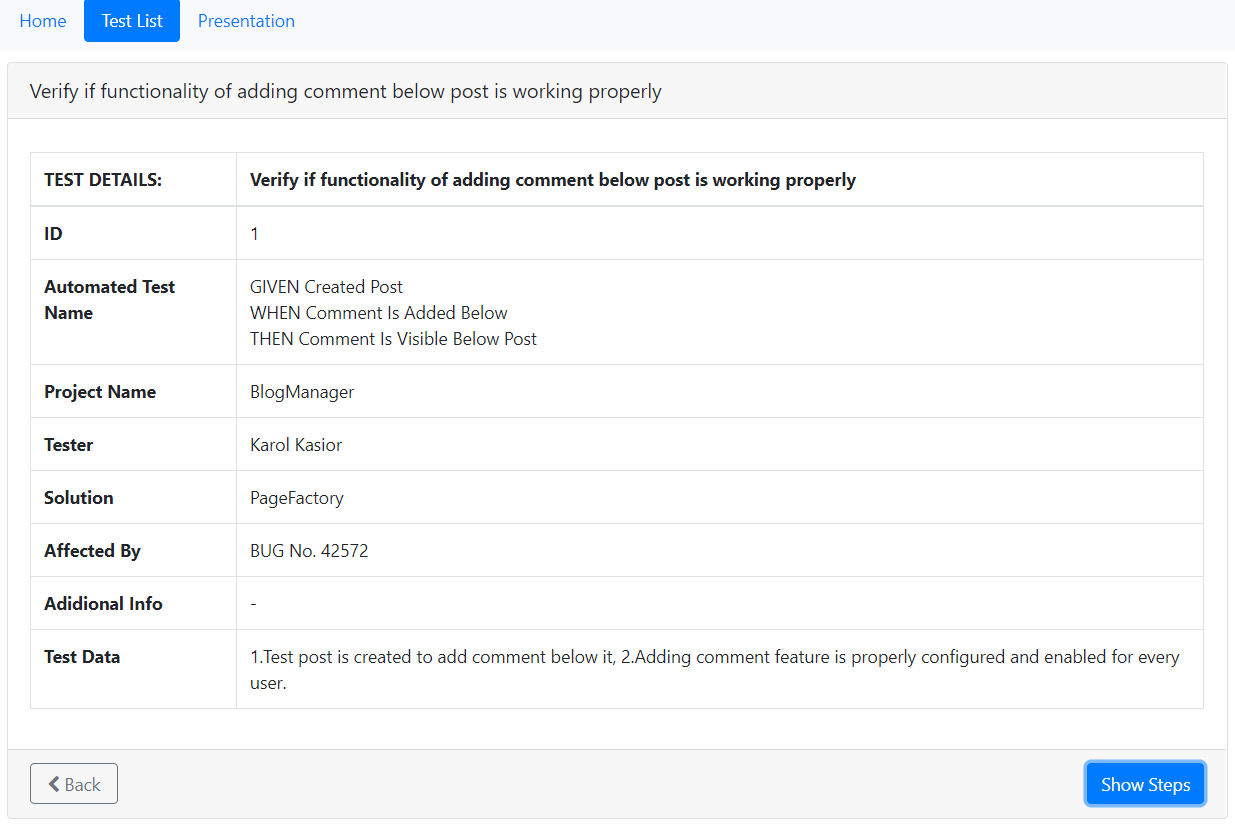
* **ID :** 1
* **Nazwa testu (Test name):** Sprawdź czy funkcjonalność dodawania komentarza pod postem działa poprawnie
* **Nazwa automatyzowanego testu (Automated test name):** GIVEN Created Post WHEN Comment Is Added Below THEN Comment Is Visible Below Post,
* **Nazwa projektu (Project name):** BlogManager,
* **Tester:** Karol Kasior,
* **Solucja (Solution):** PageFactory (link do repozytorium),
* **Spowodowane przez (Affected by):** BUG nr 42572 – Dodawanie komentarzy pod postem nie działa
* **Dodatkowe informacje (Additional info):** brak,
* **Dane do przeprowadzenia testu (Test data):**

1.Testowy post jest dodany,

2.Funkcjonalność dodawania komentarza jest prawidłowo skonfigurowana i dozwolona dla każdego użytkownika.



Rysunek 14. Scenariusz testowy na liście scenariuszy. Źródło: Opracowanie własne.



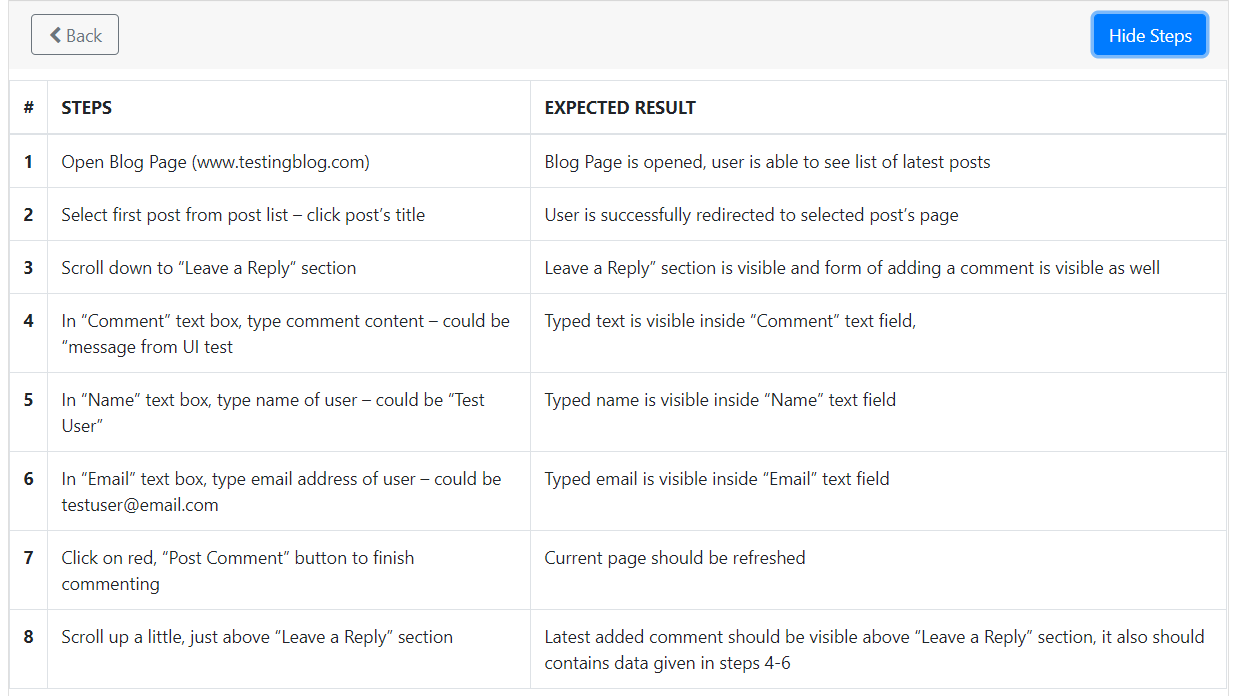
Rysunek 15. Detale przypadku testowego. Źródło: Opracowanie własne.

Kroki testowe:

1. Otwórz stronę główną („www.testingblog.com”),
2. Wybierz pierwszy wpis z listy (kliknij w jego tytuł),
3. Przewiń stronę w dół, do sekcji “Leave A Reply”,
4. W polu “Comment” wpisz treść komentarza,
5. W polu “Name” wpisz nazwę użytkownika,
6. W polu “E-mail” wpis adres e-mail użytkownika [testuser@email.com](mailto:testuser@email.com),
7. Kliknij czerwony przycisk “Post Comment”,
8. Przewiń stronę w górę, tuż nad sekcję „Leave A Reply”.

Oczekiwane rezultaty:

1. Strona główna blogu jest otwarta, użytkownik może zobaczyć listę ostatnich postów,
2. Użytkownik został prawidłowo przekierowany do wybranej strony,
3. Sekcja “Leave A Reply” jest widoczna wraz z formularzem dodawnia komentarza,
4. Wpisany teskst jest widoczny w polu “Comment”,
5. Wpisany teskst jest widoczny w polu “Name”,
6. Wpisany teskst jest widozny w polu “E-mail”,
7. Strona powinna zostać odświerzona,
8. Ostatnio dodany komentarz powinien być wyświetlony wraz ze wszystkimi wartościami dodanymi w krokach 4-6.



Rysunek 16. Kroki przypadku testowego i oczekiwane rezultaty. Źródło: Opracowanie własne.

4.2.2.Utworzenie klas Page Object

W przedstawianym przykładzie założono, iż przypadek testowy zaprezentowany w poprzednim podrozdziale będzie pierwszym testem automatycznym tworzonym w ramach środowiska. Można przyjąć jego kroki testowe, jako plan implementacji– kolejne zaprogramowane metody będą wykonywane w tej samej kolejności co w scenariuszu. W pierwszej kolejności, należy odwzorować klasy Page Object na podstawie stron występujących w teście. Są to kolejno:

- strona startowa zawierająca listę ostatnich postów – klasa „PostList.cs”,

- uniwersalna strona pojedynczego wpisu – klasa „SinglePostPage.cs”.

W obu klasach jako właściwości zostały zdefiniowane potrzebne elementy do przeprowadzenia testu. Przykładowo tytuł pierwszego postu odwzorowano pod nazwą „PostTitle”:

[FindsBy(How = How.ClassName, Using = "site-title")]

public IWebElement PageTitle { get; set; }

Dodatkowo została utworzona klasa „BaseBlogPage.cs”. Stanowi ona klasę bazową, po której dziedziczą wszystkie klasy Page Object, dzięki takiemu podejściu zminimalizowane zostało ryzyko powtarzalności kodu w obrębie reprezentacji stron.

4.3.3.Zaprogramowanie testu automatycznego na podstawie utworzonego scenariusza testowego

Kroki testowe zostały odwzorowane jako metody poszczególnych klas Page Object, natomiast wywoływane są one w klasie testowej „SinglePostPageUITest.cs”, która również posiada swoją klasę bazową „BaseUITest.cs”. Do implementacji i egzekucji testów użyto frameworka NUnit, zaimportowanego poprzez odpowiednie pakiety NuGet do projektu oraz do kodu – poprzez przestrzenie nazw.

Każdy test zostaje rozpoczęty poprzez inicjalizację przeglądarki na zdalnym serwerze Selenium. W początkowej sekcji „SetUp” do pola „driver” przypisywany jest rezultat metody „InitializeRemoteChromeWebDriver()” klasy „BaseUITest”.

[SetUp]

public void TestInitialize()

{

driver = InitializeRemoteChromeWebDriver();

}

Następnie w sekcji „Test” wywoływane już są kolejne kroki testu. W pierwszej kolejności zostaje utworzony obiekt klasy „PostList” – „postListPage”, na którym przeprowadzone zostaną wszystkie akcje użytkownika ze strony startowej.

PostList postListPage = new PostList(driver);

W konstruktorze „PostList” ustawiana jest przeglądarka, przechowywana w polu driver, dodatkowo inicjalizowane są obiekty typu „IWebElement” poprzez metodę „InitElements()” klasy „PageFactory”(jednego z dodatków Selenium) oraz ustawiany jest adres url strony startowej witryny – następuje przekierowanie na stronę startową aplikacji testowej.

public PostList(IWebDriver \_driver)

{

this.driver = \_driver;

PageFactory.InitElements(driver, this);

driver.Url = postListPageUrl; }

Po inicjalizacji strony startowej, należy sprawdzić czy przekierowanie na nią nastąpiło poprawnie. Za pomocą metody „AreEqual” klasy „Assert” weryfikowany jest tytuł znajdujący się w górnej części strony - „Testing Blog” - czy jest on wyświetlany na stronie. Jeśli weryfikacja jest poprawna, przypisujemy do zmiennej typu „string” tytuł pierwszego postu znajdującego się na stronie głównej oraz przechodzimy na stronę tego wpisu, poprzez wywołanie metody „OpenFirstPostPage”.

Assert.IsTrue(postListPage.PageTitle.Displayed);

string firstPostTitle = postListPage.FirstPostTitle.Text;

SinglePostPage firstPostPage = postListPage.OpenFirstPostPage();

Implementację metody otwierającej pierwszy wpis zaprezentowano poniżej:

public SinglePostPage OpenFirstPostPage()

{

try

{

FirstPostTitle.Click();

}

catch(Exception e)

{

if(FirstPostTitle.Displayed && FirstPostTitle.Enabled)

FirstPostTitle.Click();

}

SinglePostPage singlePostPage = new SinglePostPage(driver);

return singlePostPage;

}

Aby uniknąć niepotrzebnego oczekiwania na przeglądarkę i spowolnienia testu – użyto szkieletu „try-catch”. Na samym początku w „try” następuje próba otwarcia strony wpisu, jednak element na stronie może nie zostać jeszcze wygenerowany. W takim wypadku, zwracany jest błąd do sekcji „catch”, w której sprawdzamy istnienie elementu i jeśli rzeczywiście on istnieje – zostaje kliknięty, a użytkownik zostaje przekierowany na stronę wpisu. Na końcu metody zostaje zwrócony obiekt klasy „SinglePostPage”, przypisany do „firstPostPage” w kodzie testu. Po przekierowaniu na stronę pojedynczego posta – zostaje przeprowadzona weryfikacja, czy strona jest prawidłowa. Porównywany jest tytuł wpisu z tytułem znajdującym się w zmiennej „firstPostTitle”.

Assert.AreEqual(firstPostTitle.ToString(), firstPostPage.PostTitle.Text);

Kolejnym krokiem scenariusza testowego jest przejście do sekcji „Leave A Reply”. W przedstawianym przypadku nie ma potrzeby przewijania strony za pomocą kodu aby zlokalizować i przetworzyć dany element, w związku z tym nie została użyta tutaj żadna metoda przewijająca. Jednakże została ona zaimplementowana w klasie „BaseUITest”, na wypadek wystąpienia takiej potrzeby w którymś z przypadków testowych w przyszłości oraz w celu prezentacji działania środowiska.

public void ScrollToSelectedElement(IWebElement element)

{

var js = (IJavaScriptExecutor)driver;

const string script = @"arguments[0].scrollIntoView(true);";

js.ExecuteScript(script, element)}

Następnie według scenariusza testowego wykonywana jest weryfikacja czy istnieje element „LeaveAReplySection”, jeśli przebiega ona poprawnie inicjalizowane są trzy zmienne typu „string”, do których przypisywane są kolejno: treść komentarza, nazwa użytkownika oraz jego adres e-mail:

Assert.IsTrue(firstPostPage.LeaveAReplySection.Displayed && firstPostPage.LeaveAReplySection.Enabled);

string comment = "this is test comment";

string username = "testeruser";

string email = "supertester@user.com";

Zmienne są przekazywane do funkcji „AddTextToComment”, która ustawia ich wartości w polach tekstowych komentarza:

public void AddTextToComment(string commentContent, string username, string email)

{

CommentTextInput.SendKeys(commentContent);

NameTextInput.SendKeys(username);

EmailTextInput.SendKeys(email);

}

Wypełniony komentarz dodawany jest poprzez kliknięcie elementu „PostCommentButton”:

firstPostPage.AddTextToComment(comment, username, email);

firstPostPage.PostCommentButton.Click();

Ostatnim krokiem testowym jest porównanie dodanego komentarza, z ustawianymi wcześniej wartościami, wykonywane jest to poprzez metodę „CompareLastCommentWithGivenData”:

firstPostPage.CompareLastCommentWithGivenData(username, comment);

Implementacja tej metody opiera się na sprawdzeniu, czy ostatni komentarz jest prawidłowo wyświetlony, jeśli tak – dopiero wtedy znajdywane i sprawdzane są przekazywane wartości w obrębie komentarza.

public void CompareLastCommentWithGivenData(string username, string commentContent)

{

try

{

if (LastComment.Displayed)

{

IWebElement usernameFromComment = LastComment.FindElement(By.ClassName("fn"));

Assert.AreEqual(usernameFromComment.Text, username);

IWebElement commentContentFromComment = LastComment.FindElement(By.CssSelector(".comment-content > p"));

Assert.AreEqual(commentContent, commentContentFromComment.Text);

}

}

catch (AssertionException e)

{

System.Console.WriteLine(e);

}

}

1. James Bach (cyt za: Smilgin R., *Zawód Tester,* Wydawnictwo Naukowe PWN SA, Warszawa 2016*,* s. 31) [↑](#footnote-ref-1)
2. *Słownik wyrażeń związanych z testowaniem,* praca zbiorowa, Stowarzyszenie Jakości Systemów Informatycznych, s. 71, 80, wersja 2.3(2014) [↑](#footnote-ref-2)
3. *Słownik wyrażeń związanych z testowaniem,* praca zbiorowa, Stowarzyszenie Jakości Systemów Informatycznych, s. 52 wersja 2.3(2014) [↑](#footnote-ref-3)
4. Siedem ogólnych zasad testowania, (http://testerka.pl/7-ogolnych-zasad-testowania). Data pobrania: 12.09.2018 r. [↑](#footnote-ref-4)
5. Historia testowania oprogramowania, (<http://testerzy.pl/baza-wiedzy/historia-testowania-oprogramowania>). Data pobrania 15.09.2018 [↑](#footnote-ref-5)
6. McLean Hall G., *Adaptywny kod*, Wydawnictwo HELION, Gliwice, s. 22 [↑](#footnote-ref-6)
7. Testowanie w Agile, Część 1, (http://browserspot.com/2018/03/09/testowanie-w-agile-czesc-i). Data pobrania 16.09.2018 [↑](#footnote-ref-7)
8. Przypadki testowe. Planowanie przebiegu testów, (<https://rst.software/pl/blog/2016/04/15/przypadki-testowe-planowanie-przebiegu-testow/>). Data pobrania: 21.09.2018 r. [↑](#footnote-ref-8)
9. Selenium WebDriver, (<https://www.seleniumhq.org/projects/webdriver/>). Data pobrania 21.09.2018 r. [↑](#footnote-ref-9)
10. Selenium Grid, (<http://toolsqa.com/selenium-webdriver/selenium-grid/>). Data pobrania 21.09.2018 r. [↑](#footnote-ref-10)
11. Pierwsze kroki z Selenium WebDriver. (<http://itcraftsman.pl/pierwsze-kroki-z-selenium-web-driver/>). Data pobrania 21.09.2018 r. [↑](#footnote-ref-11)
12. Gundecha U., Selenium i testowanie aplikacji. Receptury, Gliwice 2017, s. 48. [↑](#footnote-ref-12)
13. Tamże, s. 56. [↑](#footnote-ref-13)
14. Tamże, s. 61. [↑](#footnote-ref-14)
15. Gundecha U., Selenium i testowanie aplikacji. Receptury, Gliwice 2017, s. 62. [↑](#footnote-ref-15)
16. Tamże, s. 86. [↑](#footnote-ref-16)
17. Tamże, s 55. [↑](#footnote-ref-17)
18. Tamże, s. 55. [↑](#footnote-ref-18)
19. Gundecha U., Selenium i testowanie aplikacji. Receptury, Gliwice 2017, s. 82 [↑](#footnote-ref-19)
20. Gundecha U., Selenium i testowanie aplikacji. Receptury, Gliwice 2017, s. 86 [↑](#footnote-ref-20)
21. Gundecha U., Selenium i testowanie aplikacji. Receptury, Gliwice 2017, s. 117 [↑](#footnote-ref-21)
22. Tamże, s.115 [↑](#footnote-ref-22)
23. Getting started with Page Object Pattern for your Selenium tests, (<https://www.pluralsight.com/guides/getting-started-with-page-object-pattern-for-your-selenium-tests>). Data pobrania: 25.09.2018 [↑](#footnote-ref-23)
24. Page Object Design Pattern, (<https://www.seleniumhq.org/docs/06_test_design_considerations.jsp#page-object-design-pattern>). Data pobrania: 25.09.2018 [↑](#footnote-ref-24)
25. Katalon Studio, (https://artoftesting.com/automationTesting/katalon-studio-features-advantages-and-disadvantages.html). Data pobrania: 22.10.2018 [↑](#footnote-ref-25)
26. Błędy popełniane przy automatyzacji testów [online]. Testerzy.pl, 2018 [Dostęp z 24 października 2018]. Dostępny w Internecie: < http://testerzy.pl/baza-wiedzy/bledy-popelniane-przy-automatyzacji-testow>. [↑](#footnote-ref-26)
27. Błędy popełniane przy automatyzacji testów [online]. Testerzy.pl, 2018 [Dostęp z 24 października 2018]. Dostępny w Internecie: < http://testerzy.pl/baza-wiedzy/bledy-popelniane-przy-automatyzacji-testow>. [↑](#footnote-ref-27)
28. An introduction to NuGet, (https://docs.microsoft.com/en-us/nuget/what-is-nuget). Data pobrania: 27.10.2018 [↑](#footnote-ref-28)