OEX DIVANTE
BUSINESS SERVICES

PORADNIK O TESTOWANIU OPROGRAMOWANIA

Czyli co każdy Tester Oprogramowania powinien wiedzieć o swojej pracy?

Jakub Pakuła

Software Tester @ Divante

Testowanie oprogramowania niewątpliwie jest pracą, ale testowanie jest również pewnego rodzaju „sztuką”. Testując oprogramowanie (każde) należy o tym pamiętać – nie ma znaczenia, czy testujesz kalkulator, czy wykonujesz testy bezpieczeństwa najbardziej skomplikowanych systemów teleinformatycznych. Zapewne oba systemy będziesz testować inaczej, ale pamiętaj, że cel Twojej pracy jest zawsze ten sam, zawsze powinieneś wykonywać ją z pełnym zaangażowaniem i zgodnie ze sztuką.

Jeśli:

- interesują Cię zagadnienia związane z testowaniem oprogramowania…
- chcesz rozpocząć swoją przygodę z testowaniem oprogramowania…
- chcesz w prosty i przystępny sposób posiąść wiedzę o testowaniu oprogramowania…
- chcesz dowiedzieć się podstawowych rzeczy o automatyzacji testów…
- masz ochotę przyswoić kilka podstawowych pojęć związanych z testowaniem oprogramowania…
- szykujesz się na rozmowę kwalifikacyjną, do ważnego egzaminu…
- lub po prostu chcesz odkurzyć swoją wiedzę…

...ten poradnik jest jak najbardziej dla Ciebie! Znajdziesz tutaj wszystkie podstawowe informacje na temat **sztuki testowania oprogramowania.** Wiedza zawarta w tym poradniku jest niezbędna każdemu testerowi oprogramowania, dla którego testowanie nie jest zabawą, lecz pracą.

DIVANTE
BUSINESS SERVICES

Spis treści

1. Podstawy testowania – po co testujemy, co to jest błąd? .... - 4 -
2. Jak wygląda proces testowy? .... - 6 -
3. Dobry przypadek testowy .... - 8 -
4. Jak poprawnie zgłaszać znaleziony błąd? .... - 10 -
5. Techniki projektowania testów .... - 11 -
6. Poziomy testów .... - 14 -
7. Typy testów .... - 16 -
8. Narzędzia wspomagające testowanie .... - 19 -
9. Dwa słowa o testach automatycznych .... - 20 -
10. Słownik pojęć podstawowych .... - 29 -

Bibliografia .... - 35 -

1. Podstawy testowania – po co testujemy, co to jest błąd?

Dlaczego testowanie oprogramowania jest ważne?

Pytanie niby banalne, ale wcale nie tak prosto znaleźć na nie odpowiedź. Z jednej bardzo ważnej rzeczy powinniśmy zdać sobie sprawę już na początku: samo testowanie nie podnosi jakości oprogramowania i dokumentacji. Źle działający system może doprowadzić do utraty pieniędzy, czasu, reputacji biznesowej, może również opóźnić podanie wyników samorządowych, a nawet spowodować utratę zdrowia lub życia.

Kiedy testowanie wykrywa usterki, jakość systemu podnosi się wraz z ich naprawieniem. Testowanie pozwala zmierzyć jakość oprogramowania wyrażoną przez ilość znalezionych usterek zarówno dla funkcjonalnych, jak i niefunkcjonalnych wymagań i atrybutów oprogramowania. Dzięki testom możemy budować zaufanie do jakości tworzonego przez cały zespół ludzi oprogramowania (jeśli osoby testujące system znajdują mało usterek lub nie znajdują ich w ogóle).

Najważniejsze cele testowania:

- znajdowanie usterek,
- nabieranie zaufania do poziomu jakości,
- dostarczanie informacji potrzebnych do podejmowania decyzji,
- zapobieganie defektom.

Testowanie oprogramowania polega na jego wykonywaniu
z intencją wykrywania tkwiących w nim błędów.

Co to jest testowanie i jak dużo testów potrzeba?

Testowanie oprogramowania to wykonywanie kodu dla kombinacji danych wejściowych i stanów w celu wykrycia błędów. Jest to proces (szereg procesów) zaprojektowany w celu zapewnienia, że skutek wykonania określonego kodu dokładnie zgadza się z założeniami projektowymi oraz że wykonanie to nie powoduje skutków niezgodnych z tymi założeniami.

Kolejną rzeczą, o której musimy pamiętać jest to, że nie jesteśmy w stanie przetestować wszystkich możliwych danych wejściowych / wyjściowych programu. W świecie idealnym chcielibyśmy przetestować wszelkie możliwe rodzaje danych wejściowych, ale liczba potencjalnych przypadków testowych może sięgać setek, tysięcy, a w dużych systemach nawet milionów – co przekracza granice ludzkich możliwości. Liczba możliwych ścieżek prowadzących przez program jest ogromna, czasem wręcz niepoliczalna.

Jednoznacznie nie da się określić, jak dużo czasu należy przeznaczyć na testy. Jest to sprawa bardzo indywidualna i zależy przede wszystkim od czasu i przewidzianego budżetu. Testowanie powinno

dostarczyć informacji wystarczających do podjęcia świadomych decyzji o dopuszczeniu testowanego oprogramowania lub systemu do następnej fazy rozwoju lub przekazaniu go klientowi.

Co to jest błąd?

- Oprogramowanie nie wykonuje czegoś, co według specyfikacji powinno wykonywać.
- Oprogramowanie robi coś, czego według specyfikacji nie powinno robić.
- Oprogramowanie wykonuje coś, o czym specyfikacja w ogóle nie wspomina.
- Oprogramowanie nie wykonuje czegoś, o czym specyfikacja wprawdzie nie wspomina, ale powinna.
- Oprogramowanie jest trudne do zrozumienia, trudne do użycia, powolne albo – zdaniem testera – będzie w oczach użytkownika po prostu nieprawidłowe.

Podstawowe reguły testowania programów.

- **Testowanie ujawnia istnienie błędów**

Testowanie może pokazać, że istnieją usterki, ale nie może dowieść, że oprogramowanie nie posiada defektów.

- **Testowanie wyczerpujące nie jest możliwe**

Jest to podejście do testów, w którym zestaw testowy obejmuje wszystkie kombinacje wartości wejściowych i warunków wstępnych – jest to wykonalne jedynie w trywialnych przypadkach.

- **Testowanie należy rozpocząć jak najwcześniej**

Aktywności testowe powinny rozpoczynać się w cyklu życia oprogramowania tak szybko, jak to tylko możliwe. Możemy testować zanim jeszcze powstanie oprogramowanie. Co więcej, możemy rozpocząć testowanie zanim powstanie dokumentacja oprogramowania. Możemy testować już samą koncepcję i stopień możliwości jej realizacji.

- **Kumulacja błędów**

Pracochłonność testowania jest dzielona proporcjonalnie do spodziewanej oraz zaobserwowanej gęstości błędów w modułach. Niewielka liczba modułów zwykle zawiera większość usterek znalezionych przez wydaniem lub jest odpowiedzialna za większość awarii na produkcji.

- **Paradoks pestycydów**

Mamy z nim do czynienia, kiedy obecny zestaw testów nie jest w stanie znaleźć defektów. Rozwiązaniem takiego problemu jest odpowiednie zarządzanie przypadkami testowymi. Przypadki testowe nieustanie powinny być przeglądane, modyfikowane i uaktualniane.

- **Testowanie jest zależne od kontekstu**

Tak jak zostało wspomniane wcześniej, sposób testowania programu zależy od jego przeznaczenia i złożoności. Inaczej będziemy testować oprogramowanie krytyczne ze względu na bezpieczeństwo, inaczej testuje się sklep internetowy.

- **Błędne przeświadczenie o braku błędów**

Znalezienie i eliminacja błędów nie pomoże, jeżeli system jest nieużyteczny oraz nie spełnia potrzeb i oczekiwań Klienta, użytkowników.

2. Jak wygląda proces testowy?

Według syllabusa „Certified Tester – Foundation Level Syllabus" (ISTQB®) podstawowy proces testowy składa się z następujących czynności:

1. **Planowanie i nadzór nad testami**

Planowanie testów:
- weryfikacja misji testowania,
- zdefiniowanie strategii testowania,
- zdefiniowanie celów testowania,
- określenie aktywności testowych mających spełnić cele i misję testowania.

Kontrola testów:
- porównywanie aktualnego postępu w stosunku do założonego planu,
- raportowanie statusu (szczególnie odchyłek od założonego planu),
- podejmowanie kroków niezbędnych do spełnienia założonej misji i celów testów,
- aktywność ciągła w projekcie,
- kontrola możliwa tylko dzięki ciągłemu monitorowaniu testów.

2. **Analiza i projektowanie testów**

Głównymi zadaniami analizy i projektowania testów są:
- przeglądanie podstawy testów (wymagań, poziomu integralności oprogramowania, raportów analizy ryzyka, architektury, projektu, specyfikacji interfejsów),
- ocena testowalności podstawy testowania i przedmiotu testów,
- identyfikacja i priorytetyzacja warunków testowych na podstawie analizy elementów testowych, specyfikacji, zachowania i struktury oprogramowania,
- projektowanie i priorytetyzacja przypadków testowych wysokiego poziomu,
- ustalenie, jakie dane testowe są potrzebne dla warunków testowych oraz przypadków testowych,
- projektowanie środowiska testowego oraz identyfikacja potrzebnej infrastruktury i narzędzi,
- tworzenie dwukierunkowego śledzenia pomiędzy podstawą testów oraz przypadkami testowymi.

3. Implementacja i wykonanie testów

Głównymi zadaniami implementacji i wykonania testów są:
- dokończenie, implementacja i priorytetyzacja przypadków testowych (włącznie z identyfikacją danych testowych),
- przygotowanie i priorytetyzacja procedur testowych, tworzenie danych testowych oraz, opcjonalnie, przygotowywanie jarzm testowych i pisanie automatycznych skryptów testowych,
- tworzenie zestawów testów z procedur testowych w celu efektywniejszego wykonania testów,
- sprawdzenie, czy środowisko testowe zostało poprawnie skonfigurowane,
- weryfikacja oraz uaktualnienie dwukierunkowego śledzenia pomiędzy podstawą testów oraz przypadkami testowymi,
- wykonanie procedur testowych w zaplanowanej kolejności, ręcznie lub przy pomocy narzędzi do wykonywania testów,
- zapisywanie wyników wykonania testów oraz zapisywanie identyfikatorów i wersji testowanego oprogramowania, narzędzi testowych oraz testaliów,
- porównywanie wyników rzeczywistych z wynikami oczekiwanymi,
- raportowanie rozbieżności jako incydentów oraz ich analiza w celu ustalenia ich przyczyny (usterki w kodzie, w danych testowych, w dokumencie testowym albo pomyłka w trakcie wykonywania testów),
- powtarzanie czynności testowych jako rezultat akcji podjętych po stwierdzeniu rozbieżności.

4. Ocena kryteriów zakończenia i raportowanie

Głównymi zadaniami oceny spełnienia kryteriów zakończenia są:
- sprawdzanie w logach (dziennikach) testów, czy zostały spełnione kryteria zakończenia testów określone podczas planowania,
- ocenienie, czy potrzeba więcej testów lub czy nie powinny zostać zmienione kryteria zakończenia,
- napisanie raportu podsumowującego testy dla interesariuszy.

5. Czynności zamykające test

Głównymi zadaniami wykonywanymi w ramach czynności zamykających testy są:
- sprawdzenie, które planowane produkty zostały dostarczone,
- zamknięcie raportów incydentów lub utworzenie zgłoszeń zmian dla tych, które pozostały otwarte,
- udokumentowanie akceptacji systemu,
- dokończenie i zarchiwizowanie testaliów, środowiska testowego i infrastruktury testowej do ponownego użycia w późniejszym terminie,
- przekazanie testaliów do zespołu serwisowego,
- przeanalizowanie doświadczeń by ustalić, jakie zmiany są potrzebne w przyszłych wydaniach i projektach,
- wykorzystanie zebranych informacji do podniesienia dojrzałości testowania.

3. Dobry przypadek testowy

Przypadek testowy to zbiór danych wejściowych, wstępnych warunków wykonania, oczekiwanych rezultatów i końcowych warunków wykonania opracowany w określonym celu lub dla warunku testowego, jak wykonanie pewnej ścieżki programu lub zweryfikowanie zgodności ze specyfikacją.

Dla każdego przypadku testowego powinny zostać określone wyniki oczekiwane. Powinny one zawierać opis wyjść, zmian w danych i stanie oprogramowania oraz inne skutki testu. Gdy wyniki oczekiwane nie są zdefiniowane, może się zdarzyć, że wiarygodne, ale błędne wyniki zostaną uznane za poprawne. Najlepiej jest, gdy wyniki oczekiwane zostaną zdefiniowane przed wykonaniem testów.

Testowanie oprogramowania nie może zagwarantować, że program jest całkowicie wolny od błędów. Zatem każde testowania z założenia jest niekompletne, w związku z tym trzeba tak konstruować przypadki testowe, aby możliwie jak najbardziej zminimalizować skutki niekompletności testów. Podsumowując: zestaw przypadków testowych użytych w testowaniu danego programu powinien być w jak największym stopniu kompletny.

Każdy przypadek testowy powinien zawierać:

- Identyfikator – ma jasno określać przynależność danej grupy przypadków testowych,
- Nazwa testu – powinna określać, co ogólnie chcemy otrzymać po przeprowadzeniu danego przypadku,
- Dotyczy – konkretna funkcjonalność, klasa, metoda itp.,
- Utworzył – dane osoby, która utworzyła dany przypadek testowy, może być również login tej osoby,
- Data utworzenia – data, kiedy powstał dany przypadek, ułatwia tworzenie historii testów,
- Typ – czy test należy przeprowadzić manualnie, czy ma to zrobić automat,
- Opis – szczegółowy opis celu przypadku testowego, powinien zawierać informacje:
  1) Wymagania – które wymagania dotyczą konkretnego przypadku testowego,
  2) Cel – słowny opis celu przeprowadzenia testu wraz ze szczegółami, czego test dotyczy,
  3) Warunki początkowe – Jakie warunki muszą zostać spełnione, by można przystąpić do wykonywania testu – np. jakie są potrzebne uprawnienia lub wcześniej przygotowane dane,
- Status – Informuje, czy test został już przeprowadzony, czy nie,
- Etapy – Instrukcja krok po kroku, jak należy wykonać test. Składa się z:
  1) Krok – numer wykonywanego kroku,
  2) Opis – szczegółowy opis czynności w danym punkcie,
  3) Oczekiwania – co powinno być wynikiem czynności przeprowadzonych w danym kroku,
- Rezultat – miejsce na wpisanie wyników przeprowadzonego testu oraz dodanie ewentualnych uwag.

Tabela 1. Podstawowy szablon przypadku testowego

<table>
<tr><td>Identyfikator</td><td colspan="2">ID (numer przypadku testowego)</td></tr>
<tr><td>Nazwa testu</td><td colspan="2">Zdefiniowanie czynności wykonywanej podczas testu</td></tr>
<tr><td>Utworzył</td><td colspan="2">Osoba odpowiedzialna za stworzenie przypadku testowego</td></tr>
<tr><td>Data utworzenia</td><td colspan="2">Data, godzina</td></tr>
<tr><td>Typ testu</td><td colspan="2">Manualny / Automatyczny</td></tr>
<tr><td rowspan="3">Opis przypadku</td><td colspan="2">Wymagania</td></tr>
<tr><td colspan="2">Cel testu</td></tr>
<tr><td colspan="2">Warunki początkowe</td></tr>
<tr><td>Status</td><td colspan="2">Wykonany: Tak / Nie</td></tr>
<tr><td colspan="3">Etapy</td></tr>
<tr><td>Nr kroku</td><td>Opis operacji</td><td>Spodziewane rezultaty</td></tr>
<tr><td>#1</td><td>Opis wykonywanej czynności</td><td>Spodziewana reakcja systemu</td></tr>
<tr><td>#2</td><td>Opis wykonywanej czynności</td><td>Spodziewana reakcja systemu</td></tr>
<tr><td>#n</td><td>Opis wykonywanej czynności</td><td>Spodziewana reakcja systemu</td></tr>
<tr><td>Rezultat</td><td colspan="2">Wynik testu: Poprawny / Niepoprawny (jeśli wynik niepoprawny należy podać numer kroku i krótki opis błędu, uwagi dotyczące przypadku)</td></tr>
</table>

Ważne jest, żeby przypadki testowe były napisane jasno, by nie trzeba było zbyt fachowej wiedzy do ich przeprowadzenia. Trzeba liczyć się z tym, że nie każdy przyszły użytkownik systemu będzie ekspertem w danej dziedzinie. Przypadki powinny tez być odpowiednio skomentowane, tak aby po jakimś czasie można było dokonać poprawek bez dłuższego zastanawiania się, co dana część kodu robi. Podczas implementacji testów przypadki testowe są rozwijane, implementowane, priorytetyzowane i układane w specyfikację procedur testowych.

4. Jak poprawnie zgłaszać znaleziony błąd?

Jednymi z najczęściej popełnianych błędów przez testerów (zarówno tych początkujących, jak i doświadczonych) jest to, że zgłaszane błędy są bez wystarczającego opisu. Zgłaszając znaleziony przez Ciebie defekt musisz pamiętać o tym, że stworzony raport błędu jest nośnikiem informacji, w którym informujesz programistę o różnicach pomiędzy oczekiwanym rezultatem i aktualnym stanem. Głównym celem tworzenia raportu o błędzie jest umożliwienie programiście zobaczenie i zrozumienie problemu. Raport powinien również zawierać inną niezbędną informację, jak wywołać błąd ponownie. Znaleziony błąd powinien być opisany w sposób jednoznaczny, zwięzły, precyzyjny i dokładny, aby programista czytający zgłoszenie nie musiał zastanawiać się „co autor miał na myśli". Programista powinien możliwie szybko wywołać wskazany defekt bez przeszukiwania całego serwisu.

Co zrobić, kiedy pojawi się błąd?

Po pierwsze: nie panikować. To najgorsze, co można zrobić. Nigdy nie powinniśmy tracić głowy i odruchowo zamykać okna informujące o błędzie – tak tester oprogramowania nigdy nie powinien robić. Pierwsze co powinieneś zrobić, to przeczytać komunikat o błędzie i zrobić zrzut ekranu (na pewno się przyda). Nawet jeśli dla nas komunikat jest niezrozumiały, zawiera ciąg dziwnych, z pozoru losowych znaków, to dla programisty może być sporym ułatwieniem w znalezieniu defektu w kodzie.

Drugim nawykiem, który powinien wyrobić w sobie każdy tester oprogramowania to sprawdzanie, czy dany błąd jest powtarzalny. Dla programisty jest to bardzo istotna informacja, czy błąd jest jednorazowy, czy można go bez problemu wywołać wielokrotnie. Znajdując błąd powinniśmy zawsze sprawdzać, czy podanie np. innych danych wejściowych, spowoduje pojawienie się błędu (być może problem pojawia się tylko po wpisaniu długiego tekstu, być może długość aktywności w sesji ma znaczenie?).

Znajdując defekt, najczęściej nie będziesz znał jego faktycznego powodu występowania. Zapewne nie podasz programiście gotowego sposobu na rozwiązanie problemu, ale możesz pomóc programiście pisząc mu o swoich przypuszczeniach. Ważne jest to, abyś nie bał się używać takich sformułowań, jak: „Wydaje mi się, że…", „Problem może być spowodowany przez…", „Możliwe, że…".

Jak zgłaszać?

Tytuł
Dobry tytuł jest zwarty i w kilku słowach oddaje istotę problemu. Unikajmy jednak zbyt ogólnych sformułowań, takich jak „Menu nie działa". O wiele lepszym byłoby „Elementy menu są nieklikalne".

Miejsce wystąpienia
W tym miejscu podajemy adres URL lub ciąg dostępu do miejsca, w którym pojawił się błąd, a także usytuowanie na stronie.

Środowisko
W przypadku aplikacji i serwisów internetowych niezbędne jest podanie informacji o wersji przeglądarki oraz systemu operacyjnego. Podobnie w razie zgłoszenia błędów występujących na urządzeniach mobilnych – należy podać urządzenie, wersję systemu oraz przeglądarkę internetową.

Ścieżka odtworzenia
Programiści lubią konkrety, a ścieżka odtworzenia błędu nieraz jest niezbędna do jego naprawy. Niestety musimy liczyć się z tym, że developer odrzuci zgłoszenie, jeśli nie będzie wiedział, jak uzyskać opisywany bug. Sposób zapisu odtworzenia błędu jest dowolny. Pamiętajmy jednak o zasadzie: im prościej, tym lepiej. Z pewnością należy unikać długich, wielokrotnie złożonych zdań.

Informacje dodatkowe
Jest to miejsce na zawarcie swoich przypuszczeń oraz spostrzeżeń. Zawsze warto dodać, że błąd jest powtarzalny lub jednorazowy. W dobrym zwyczaju jest także podanie swoich oczekiwań co do spodziewanego efektu. Jeśli czujemy się na siłach, możemy dołączyć także nasze przypuszczenia co do powodu występowania problemu.

Bardzo ważne również jest to, jak zgłaszasz swoje uwagi i spostrzeżenia. Jakikolwiek znalazłeś błąd pamiętaj, że **opis dotyczy zaistniałej sytuacji nie zaś osoby, która kodowała daną funkcjonalność.** Zgłaszając błąd nigdy nie krytykuj programisty, raport z błędem nie jest odpowiednim do tego miejscem. Można uniknąć spięć pomiędzy testerami a analitykami, projektantami i programistami przez komunikowanie błędów, usterek i awarii w sposób **konstruktywny.**

5. Techniki projektowania testów

Celem technik projektowania testów jest zdefiniowanie warunków testowych, przypadków testowych i danych testowych.

Najmniej skutecznym sposobem projektowania przypadków testowych jest ich losowy wybór spośród wszystkich możliwych w danej sytuacji. Taki sposób jest najgorszy pod względem prawdopodobieństwa wykrywania błędów. Losowy zestaw przypadków testowych ma nikłą szansę na to, by być zestawem optymalnym lub bliskim optymalnemu.

Klasyczny podział wyróżnia techniki **czarnoskrzynkowe** oraz **białoskrzynkowe.**

Techniki czarnoskrzynkowe

Testowanie metodą „czarnej skrzynki” (testowanie oparte na specyfikacji) opiera się na traktowaniu testowanego programu jak swoistej „skrzynki”, której wewnętrzna struktura pozostaje nieznana. Wykonujące testy opracowane metodą czarnoskrzynkową ważny jest jedynie aspekt funkcjonalny, czyli otrzymywane wyniki na określone dane wejściowe. Celem testowania jest wykrycie i zdefiniowanie warunków, w których wynik nie jest zgodny ze specyfikacją (w czasie testowania nie wykorzystujemy żadnych informacji o strukturze wewnętrznej testowanego modułu lub systemu).

Techniki testowania oprogramowania oparte na specyfikacji:

- Podział na klasy równoważności,
- Analiza wartości brzegowych,
- Testowanie w oparciu o tablicę decyzyjną,
- Testowanie przejść między stanami,
- Testowanie w oparciu o przypadki testowe.

Zalety testowania metodą czarnej skrzynki:
- Testy są powtarzalne,
- Testowane jest środowisko, w którym przeprowadzane są testy,
- Zainwestowany wysiłek może być użyty wielokrotnie.

Wady testowania metodą czarnej skrzynki:
- Wyniki testów mogą być szacowane nazbyt optymistycznie,
- Nie wszystkie właściwości systemu mogą zostać przetestowane,
- Przyczyna błędu nie jest znana.

Techniki białoskrzynkowe

Testowanie za pomocą technik białoskrzynkowych (techniki oparte na strukturze) zakłada wgląd w wewnętrzną strukturę programu i uczynienie tej struktury podstawą do projektowania przypadków testowych. Celem testów strukturalnych jest zbadanie pokrycia, czyli wskaźnika przetestowania struktury (wybranego typu struktury) przez zestaw przygotowanych testów.

Techniki testowania oprogramowania oparte na strukturze:

- Testowanie i pokrycie instrukcji,
- Testowanie i pokrycie decyzji,
- Testowanie pokrycia warunków,
- Testowanie wielokrotnego pokrycia decyzji.

Zalety testowania metodą białej skrzynki:
- ponieważ wymagana jest znajomość struktury kodu, łatwo jest określić, jaki typ danych wejściowych/wyjściowych jest potrzebny, aby efektywnie przetestować aplikację
- oprócz głównego zastosowania testów, pomaga zoptymalizować kod aplikacji,
- pozwala dokładnie określić przyczynę i miejsce, w którym znajduje się błąd.

Wady testowania metodą białej skrzynki:
- ponieważ wymagana jest znajomość struktury kodu, do przeprowadzenia testów potrzebny jest tester ze znajomością programowania, co podnosi koszty,
- jest prawie niemożliwym przejrzenie każdej linii kodu w poszukiwaniu ukrytych błędów, co może powodować błędy po fazie testów.

Testowanie oparte na doświadczeniu

Testy oparte na doświadczeniu wykorzystują umiejętności i intuicję testera wraz z jego doświadczeniem w testowaniu podobnych aplikacji lub technologii. Techniki te są skuteczne w znajdowaniu błędów, ale nie tak odpowiednie do badania pokrycia testowego i tworzenia reużywalnych procedur testowych, jak inne techniki. Testerzy mają skłonność do testowania opartego na doświadczeniu. Taki sposób testowania bardziej reaguje na zdarzenia podczas testów niż podejścia planowane.

Techniki testowania oprogramowania opartego na doświadczeniu:
- Zgadywanie błędów,
- Testowanie eksploracyjne,
- Atak usterkowy.

W technikach opartych na usterkach i doświadczeniu stosuje się wiedzę na temat usterek oraz inne doświadczenia do zwiększenia skuteczności znajdowania usterek. Obejmują one szeroki zakres działań, od "szybkich testów", gdzie działania testera nie zostają wcześniej zaplanowane, przez planowane sesje do sesji ze skryptami. Są one prawie zawsze użyteczne, ale szczególnie przydają się w następujących okolicznościach:
- brak jest specyfikacji,
- testowany system jest słabo udokumentowany,
- brak jest wystarczającego czasu na zaprojektowanie i utworzenie procedur testowych,
- testerzy mają doświadczenie w testowanej dziedzinie lub technologii,
- szuka się urozmaicenia w stosunku do testowania ze skryptami,
- podczas analizy awarii na produkcji.

6. Poziomy testów

Testowanie oprogramowania jest procesem podzielonym na poziomy. Każdy poziom testów wpasowuje się w kolejne fazy procesu tworzenia oprogramowania. Dla każdego poziomu testowania można zdefiniować: ogólne cele, produkty, na podstawie których tworzy się przypadki testowe (podstawę testów), przedmiot testów (to co jest testowane), typowe defekty i awarie do wykrycia, wymagania na jarzmo testowe oraz wsparcie narzędziowe, środowisko testowe, specyficzne podejście, odpowiedzialność.

1. Testy modułowe (komponentów)

Typowe obiekty testów:

- Moduły,
- Programy,
- Programy do konwersji lub migracji danych,
- Moduły bazodanowe,
- Obiekty.

Główne cechy testów:

- Testy modułowe polegają na wyszukiwaniu błędów i weryfikacji funkcjonalności oprogramowania, które można testować oddzielnie,
- Testowanie komponentów może zawierać testowanie funkcjonalności oraz pewnych niefunkcjonalnych parametrów, takich jak zachowanie zasobów (np. wycieki pamięci) lub testowanie odporności na atak,
- Przypadki testowe tworzone są na podstawie specyfikacji komponentu, projektu oprogramowania lub modelu danych,
- Usterki są usuwane jak tylko zostaną wykryte, bez formalnego zarządzania nimi.

2. Testy integracyjne

Typowe obiekty testów:

- Implementacja baz danych podsystemów,
- Infrastruktura,
- Interfejsy,
- Konfiguracja systemu i dane konfiguracyjne,
- Podsystemy.

Główne cechy testów:

- Testy integracyjne sprawdzają interfejsy pomiędzy modułami, interakcje z innymi częściami systemu oraz interfejsy pomiędzy systemami.

- Im większy jest zakres integracji, tym trudniejsze może być określenie, który moduł lub system zawiera defekt, co powoduje zwiększone ryzyko i dłuższy czas rozwiązywania problemów.
- Podczas testów integracyjnych można wykonać testy niektórych atrybutów niefunkcjonalnych (np. wydajności) na równi z testami funkcjonalnymi.
- Na każdym etapie integracji testerzy koncentrują się wyłącznie na samej integracji. Na przykład, gdy integrują moduł A z modułem B, interesują się tylko testowaniem komunikacji pomiędzy modułami, a nie funkcjonalnością poszczególnych modułów, gdyż ta była sprawdzona wcześniej w testach modułowych.
- W idealnym przypadku tester powinien rozumieć architekturę i mieć wpływ na planowanie integracji.

3. Testy systemowe

Typowe obiekty testów:

- Podręczniki systemowe, użytkownika i operacyjne,
- Konfiguracje systemu i dane konfiguracyjne.

Główne cechy testów:

- Testy systemowe zajmują się zachowaniem systemu/produktu.
- Zakres testów powinien być jasno określony w głównym planie testów oraz w planach testów poszczególnych poziomów.
- Testowanie systemowe powinno uwzględniać zarówno funkcjonalne, jak i niefunkcjonalne wymagania systemu.
- Środowisko testowe, podczas testów systemowych, powinno być zgodne ze środowiskiem docelowym/produkcyjnym w jak najwyższym możliwym stopniu, żeby zminimalizować ryzyko wystąpienia awarii spowodowanych przez środowisko.

4. Testy akceptacyjne

Typowe obiekty testów:

- Proces biznesowy na systemie w pełni zintegrowanym,
- Procesy utrzymania i obsługi,
- Procedury pracy użytkowników,
- Formularze,
- Raporty,
- Dane konfiguracyjne.

Główne cechy testów:

- Odpowiedzialność za testy akceptacyjne leży często po stronie klientów lub użytkowników systemu.

- Celem testów akceptacyjnych jest nabranie zaufania do systemu, jego części lub pewnych atrybutów niefunkcjonalnych.
- Testy akceptacyjne oceniają gotowość systemu do wdrożenia i użycia, chociaż nie muszą być ostatnim poziomem testowania.
- Testy akceptacyjne mogą pojawić się w wielu momentach cyklu życia oprogramowania.

Testy akceptacyjne zazwyczaj dzielą się na testy:

- Użytkownika - weryfikuje dopasowanie systemu do potrzeb użytkowników,
- Operacyjne - akceptacja systemu przez administratorów systemu zawierająca: sprawdzenie kopii zapasowej i zdolności do przywrócenia funkcjonalności po wystąpieniu problemów itp.,
- Kontraktowe i regulacyjne - testowanie kryteriów wytworzenia oprogramowania specyfikowanego dla klienta. Kryteria akceptacyjne powinny być zdefiniowane po uzgodnieniu kontraktu,
- Testy alfa i beta - ludzie tworzący oprogramowanie dla klienta kupującego produkt z półki w sklepie oczekują od niego informacji zwrotnej, zanim produkt pojawi się na rynku. Testowanie alfa odbywa się w organizacji tworzącej oprogramowanie. Testowanie beta dokonuje się po stronie odbiorcy oprogramowania. Obydwa typy testowania przeprowadzane są przez potencjalnych odbiorców produktu.

7. Typy testów

Testowanie funkcjonalne

Cel testów funkcjonalnych:
Testowanie funkcjonalne stanowi zazwyczaj próbę wykrycia rozbieżności pomiędzy programem a jego zewnętrzną specyfikacją, precyzyjnie opisującą jego zachowanie z punktu widzenia użytkownika lub bazują na wiedzy i doświadczeniu zespołu testującego.
Celem testowania funkcjonalnego nie jest udowodnienie zgodności programu z jego specyfikacją zewnętrzną, lecz wykrycie jak największej liczby niezgodności.

Główne cechy testów:

- Funkcje są tym "co" system robi.
- Do tworzenia warunków testowych oraz przypadków testowych dla funkcjonalności systemu mogą zostać użyte techniki oparte na specyfikacji.
- Testowanie funkcjonalne ma zazwyczaj charakter czarnoskrzynkowy.

Testowanie niefunkcjonalne

Cel testów niefunkcjonalnych:
Celem testów niefunkcjonalnych jest uzyskanie informacji, pomiaru o właściwościach systemu/aplikacji/modułu.

Główne cechy testów:

- Testowanie niefunkcjonalne polega na sprawdzeniu "jak" system działa.
- Testowanie niefunkcjonalne może być wykonywane na wszystkich poziomach testów.
- Testy niefunkcjonalne zajmują się zewnętrznym zachowaniem oprogramowania i w większości wypadków wykorzystują techniki czarnoskrzynkowe.
- Są to testy wymagane do zmierzenia właściwości systemów i oprogramowania, które mogą zostać określone ilościowo na zmiennej skali (np. czasy odpowiedzi w testach wydajnościowych).

Klasyfikacja testów niefunkcjonalnych:

- testowanie wydajnościowe
- testowanie obciążeniowe
- testowanie przeciążeniowe
- testowanie użyteczności
- testowanie pielęgnowalności
- testowanie niezawodności
- testowanie przenaszalności
- testowanie bezpieczeństwa

Testowanie strukturalne

Cel testów strukturalnych:
Celem testów strukturalnych jest zbadanie pokrycia, czyli wskaźnika przetestowania struktury (wybranego typu struktury) przez zestaw przygotowanych testów, wyrażony jako odsetek pokrytych elementów.

Główne cechy testów:

- Testy strukturalne są to testy białoskrzynkowe.
- Testy strukturalne można wykonywać na każdym poziomie testowania.
- Technik strukturalnych najlepiej użyć po technikach opartych na specyfikacji, by zmierzyć dokładność testowania przez ocenę stopnia pokrycia wybranego typu struktury.
- Do pomiaru pokrycia na wszystkich poziomach, ale szczególnie na poziomie testów modułowych i poziomie testów integracji modułów, mogą zostać użyte narzędzia.
- Testowanie strukturalne może zostać oparte na architekturze systemu.

Testowanie potwierdzające oraz regresywne

Bardzo wiele osób (także profesjonalnych testerów) często myli oba pojęcia używając zamiennie obu tych nazw, tak jakby oznaczały one to samo. Jest to oczywiście spory błąd i każdy tester powinien posiadać podstawową wiedzą o tych dwóch typach testów.

Cel testów regresywnych:
Testy regresywne wykonywane są w celu sprawdzenia oprogramowania po wykonanych modyfikacjach i potwierdzeniu gotowości zarówno oprogramowania, jak i środowiska testowego do dalszych czynności testowych, przejścia do kolejnych iteracji projektu testowego, itp.

Cel testów potwierdzających:
Testy potwierdzające wykonywane są w celu potwierdzenia, że znaleziony defekt został naprawiony przez programistów.

Główne cechy testów:

- Testy, które mają być stosowane w testowaniu potwierdzającym i regresywnym muszą być powtarzalne.
- Testy regresywne wykonuje się po zmianach w oprogramowaniu, a także po zmianach w jego środowisku.
- Zakres testów regresywnych wynika z ryzyka nieznalezienia usterek w oprogramowaniu, które poprzednio działało poprawnie.
- Testy regresywne można wykonywać na wszystkich poziomach testów i dla wszystkich typów testów: funkcjonalnych, niefunkcjonalnych i strukturalnych.

Testowanie pielęgnacyjne

Cel testów pielęgnacyjnych:
Testowanie pielęgnacyjne wykonuje się na działającym systemie na skutek modyfikacji, migracji lub złomowania oprogramowania lub systemu. Testy pielęgnacyjne, oprócz przetestowania tego co zostało zmienione, zawierają testy regresywne, tych części systemu, które się nie zmieniły.

Główne cechy testów:

- Modyfikacjami mogą być planowane ulepszenia, poprawki lub naprawy awaryjne oraz zmiany środowiska, takie jak planowane podniesienia wersji systemu operacyjnego, bazy danych lub oprogramowania z półki albo łaty zabezpieczeń systemu operacyjnego.
- Testy pielęgnacyjne migracji oprogramowania (np. z jednej platformy na inną) powinny, oprócz testów zmian w oprogramowaniu, uwzględniać również testy produkcyjne nowego środowiska.
- Testy pielęgnacyjne związane z wycofywaniem oprogramowania mogą zawierać testy migracji lub archiwizacji danych, jeżeli wymagany jest długi okres ich przechowywania.
- Testy pielęgnacyjne, oprócz przetestowania tego co zostało zmienione, zawierają testy regresywne tych części systemu, które się nie zmieniły.

8. Narzędzia wspomagające testowanie

Klasyfikacja narzędzi testowych

Istnieje wiele narzędzi wspierających różne aspekty testowania. Niektóre narzędzia służą do wsparcia tylko jednej czynności, inne mogą być wykorzystywane do różnych czynności, ale zaklasyfikowane zostały zgodnie z tą czynnością, do której najczęściej są stosowane. Niektóre narzędzia komercyjne wspierają tylko jeden rodzaj czynności, ale niektórzy dostawcy komercyjnych narzędzi oferują zestawy albo rodziny narzędzi, wspierających wiele różnych czynności testowych.

Najważniejsze typy narzędzi, które ułatwiają testowanie

- Narzędzia do zarządzania testami i raportami

Narzędzia te dostarczają interfejsów do wykonywania zadań, śledzenia defektów oraz zarządzania wymaganiami, jak również wspierają analizę ilościową i raportowanie na temat obiektów testów. Wspierają również śledzenie powiązań pomiędzy obiektami testów i mogą posiadać własne mechanizmy zarządzania wersjami lub interfejs do zewnętrznych narzędzi zarządzających wersjami.

- Narzędzia do automatyzacji

Narzędzia te pozwalają na automatyzację czynności, które wymagałyby wielkich nakładów, gdyby były wykonywane ręcznie. Narzędzia do testów automatycznych pozwalają na zwiększenie efektywności czynności testowych przez zautomatyzowanie powtarzających się zadań lub wsparcie dla czynności testowych wykonywanych ręcznie, takich jak planowanie testów, projektowanie testów, raportowanie i monitorowanie testów.
Automatyzacja jest szczególnie dobrym rozwiązaniem dla długoterminowych projektów. Dzięki zautomatyzowaniu większości testów regresji zyskujesz czas. Testy trwają krócej, a w dłuższym okresie koszty poświęcone na automatyzację zwracają się.

- Narzędzia do zarządzania incydentami

Narzędzia ułatwiające rejestrację incydentów i śledzenie ich statusów. Często oferują funkcje śledzenia i kontroli przepływu pracy związanego z przydziałem, naprawą i retestami. Zapewniają również możliwość raportowania.

- Narzędzia do wykonywania testów

Narzędzia do wykonywania testów uruchamiają skrypty zaprojektowane tak, aby zaimplementować testy przechowywane elektronicznie.

Nagrywanie akcji wykonywanych przez testy ręczne wydaje się być atrakcyjne, ale nie skaluje się do dużej liczby zautomatyzowanych skryptów testowych. Nagrany skrypt jest liniową reprezentacją z konkretnymi danymi i akcjami będącymi częścią każdego skryptu. Taki typ skryptu może okazać się niestabilny, gdy wystąpią niespodziewane zdarzenia.

System zarządzania incydentami powinien zapewniać:

a) elastyczne narzędzia redukujące natłok zdarzeń prezentowanych administratorowi,
b) skalowalny interfejs graficzny prezentujący selektywną informację odpowiednim osobom,
c) metody powiadamiania (pager, poczta elektroniczna, telefon) określonych grup obsługi technicznej zarządzania, a także użytkowników o pojawiających się problemach,
d) narzędzia do korelacji zdarzeń, zmniejszające czas lokalizowania i izolacji uszkodzeń, uwalniające od konieczności angażowania ekspertów,
e) narzędzia wspomagające szybką ocenę wpływu uszkodzenia na biznes (usługi, klientów).

- Narzędzia do testów wydajnościowych

Narzędzie wspierające testowanie wydajnościowe, zazwyczaj mające dwie funkcjonalności: generacja obciążenia i pomiar transakcji. Generowane obciążenie może symulować zarówno wielu użytkowników, jak i dużą ilość wprowadzanych danych. W czasie wykonywania testów pomiary są logowane tylko z wybranych transakcji. Narzędzia do testów wydajnościowych zazwyczaj dostarczają raporty bazujące na logowanych transakcjach oraz wykresy obciążenia w zależności od czasów odpowiedzi.

- Narzędzia do przygotowywania danych testowych

Jest to narzędzie, które zezwala na wybranie danych z istniejącej bazy danych lub ich stworzenie, wygenerowanie, przetworzenie i edycję dla użycia w testowaniu.

9. Dwa słowa o testach automatycznych

<u>Możemy wyróżnić następujące rodzaje testów automatycznych:</u>

- Testy modułowe (tworzone przez programistów dla programistów) – testowanie pojedynczych modułów programu,
- Testy obciążeniowe (tworzone i przeprowadzane przez testerów) – testowanie przeprowadzane w celu określenia wydajności oprogramowania,
- Testy funkcjonalne (tworzone i przeprowadzane przez testerów) – testowanie wykonywane w oparciu o analizę specyfikacji funkcjonalności systemu lub jego modułu.

Co nadaje się do automatyzacji, a co raczej nie?

Aplikacja, która jest stabilna, jest bardzo dobrym kandydatem do automatyzacji (koszty utrzymania testów automatycznych w takim przypadku są niewielkie). Każda aplikacja, która ma zostać zautomatyzowana powinna posiadać dokumentację, wg której testy zostaną napisane. Jeśli dokumentacja będzie stworzona z błędami lub będzie niekompletna, to testy automatyczne również nie będą dobrze zaprojektowane.

Automatyzowane powinny szczególnie te obszary aplikacji, które są często wykorzystywane. Dzięki testom automatycznym skróci się czas testowania tych obszarów, ale również taki test automatyczny wyręczy testera przed n-tym przechodzeniem tej samej ścieżki. Testy automatyczne są również przydatne ze względu na ilość danych testowych niezbędnych do testów. W prosty sposób możemy przygotować zestaw danych testowych niezbędnych do testów aplikacji.

Testy automatyczne mogą być sporym problemem, jeśli automatyzacji chcemy poddać dość złożoną aplikację. Ze względu na dużą ilość możliwych przypadków trudno jest przewidzieć i zautomatyzować wszystkie przypadki w taki sposób, aby nasz test był stabilny.

Lepiej wstrzymać się z automatyzacją takich aplikacji, które są często modyfikowane. Ma to bezpośredni związek z kosztownym utrzymywaniem takich testów, które będą wymagały ciągłej pracy, aby działały one w sposób zgodny z założeniami.

Zalety i wady testów automatycznych

Zalety testów automatycznych:

- Koszty wykrycia błędu są mniejsze – testy modułowe,
- Możliwość podania dużej liczby danych testowych – testy wydajnościowe,
- Analiza testów – program po zakończeniu testowania dostarcza informacji o jego przebiegu,
- Możliwość odtworzenia identycznego testu – jest to szczególnie przydatne, kiedy chcemy sprawdzić, czy wprowadzone poprawki rozwiązały problem,
- Znaczny wzrost prędkości testowania – programy testujące przeprowadzają testy znacznie szybciej niż testerzy.

Wady testów automatycznych:

- Koszty związane z zakupem narzędzi do automatyzacji (dostępne są również darmowe narzędzia),
- Koszty związane ze szkoleniami,
- Koszty związane z wykonywaniem, rozwojem i utrzymywaniem skryptów,
- Test automatycznie nie jest w stanie zastąpić człowieka. Nie dokona analizy logów, klasyfikacji nieprawidłowości i błędów, nie zgłosi i nie opisze błędów do poprawy.

W dalszej części w naszych rozważań skupimy się na darmowym narzędziu do automatyzacji testów – SELENIUM.

Selenium jest narzędziem służącym przede wszystkim do automatyzacji aplikacji internetowych. Narzędzie to pomaga testerom w przeprowadzaniu testów funkcjonalnych oraz w testach regresji. Narzędzie to wspiera wszystkie popularne platformy: Windows, Linux oraz OS. Framework składa się z następujących modułów: IDE, RC, WebDriver, GRID.

Selenium IDE

Selenium IDE jest pluginem do Firefoxa pozwalającym na nagrywanie i odtwarzanie testów. Istnieje możliwość pisania w nim prostych skryptów w wewnętrznym skryptowym języku pluginu. Narzędzie to nadaje się do małych projektów. Do obsługi Selenium IDE nie jest wymagana znajomość języka programowania, dzięki czemu jest to idealne narzędzie dla początkujących testerów, którzy stawiają dopiero pierwsze kroki w automatyzacji testów.

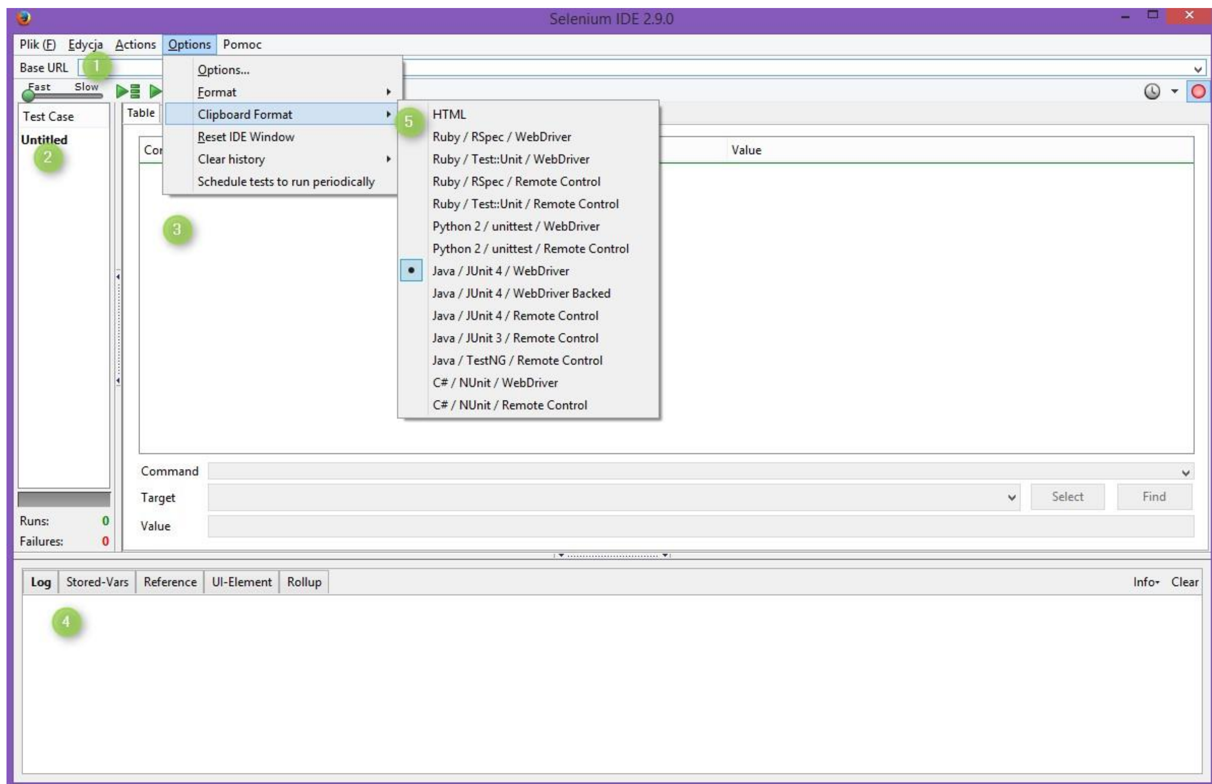
Zalety:

- Prosta instalacja,
- Nie jest wymagana znajomość języka programowania,
- Możliwość eksportowania testów do konkretnego języka programowania,
- Prostota obsługi,
- Duża ilość dodatkowych, przydatnych dodatków.

Wady:

- Przeznaczenie do dość prostych testów,
- Dostępność tylko w postaci wtyczki dla przeglądarki Firefox,
- Dość wolne wykonywanie testów.

Opis Interfejsu:



Rys. 1. Opis interfejsu użytkownika

1 – Base URL jest to miejsce na wpisanie adresu bazowego strony, którą chcemy testować. Może to być dowolna strona internetowa, np. www.divante.pl

Poniżej widoczny jest pasek, za pomocą którego możemy ustawiać szybkość wykonywanych przez nas testów. Fast – szybko, slow – wolno. Jest to przydatna opcja, kiedy chcemy sprawdzić czy po nagraniu naszego testu wykonuje się on poprawnie.

Obok paska do ustawiania prędkości wykonywanych testów mamy przyciski kontroli, przy użyciu których uruchamiamy nagrane testy bądź wybraną grupę testów, zatrzymujemy wykonywanie testów w dowolnym momencie. Ikonka w kształcie „Enter-a" pozwala nam na krokowe wykonywanie testu (debbuger). Fioletowa ikona w kształcie ślimaka służy do grupowania dowolnej liczby poleceń w jedną całość, podobnie jak w przypadku metod, które w swoim ciele zawierają różnego rodzaju sekwencje kodu, które możemy w każdej chwili wykorzystać przy użyciu wywołania jednej nazwy.

W prawym górnym rogu, pod Base URL znajduje się czerwona kropka, służąca do rozpoczęcia i zakończenia nagrywania poszczególnych kroków testu.

OEX DIVANTE
BUSINESS SERVICES

2 – sekcja Test Case zawiera nagrane przez nas przypadki testowe. Pozwala to na dokładną dokumentację i kontrolę prowadzonych zmian. Każdy Case może mieć dowolną nazwę.
Na dole widzimy licznik testów, które udało nam się utworzyć przy użyciu Selenium IDE wraz z liczbą testów, które nie powiodły się.

3 – pole Table podczas nagrywania testu, Selenium samo wygeneruje kod testu oraz sporządzi listę potrzebnych komend wymaganych do zrealizowania testu. W tym miejscu będziemy mogli również ręcznie deklarować poszczególne komendy, które mają zostać wykonywane automatycznie przez IDE.

4 – sekcja logów. W tej sekcji znajduje się kilka zakładek, które służą głównie do zwracania błędów, podpowiadania składni tworzonych komend itp.

5 – wybór języka. W tej sekcji możemy sobie wybrać, w jakim języku programowania chcemy sobie wygenerować nasz test. Aby było to możliwe, konieczne jest zaznaczenie „Enable experimental features" w zakładce Opcje.
W zakładce Format możliwy jest wybór języka, w którym nasz test zostanie wygenerowany. Domyślnie nasze IDE generuje testy w języku HTML.

Podstawowe komendy:

Tabela 2. Lista najczęściej używanych komend w Selenium IDE:

NAZWA	Target	Value	Opis
open	adres strony: względny - dodawany do base URL bezwzględny - w formacie http://....		Otwiera stronę wpisaną jako parametr.
click	Uchwyt do elementu na stronie w postaci: xpath=... css=... id=... itd.		Klika na element na stronie nie czekając na przeładowanie się strony. Dobre do otwierania różnego rodzaju layoutów.
clickAndWait	Uchwyt do elementu na stronie w postaci: xpath=... css=... id=... itd.		Klika na element i czeka, aż strona się przeładuje.
type	Uchwyt do elementu (jw)		Wysyła zadany ciąg znaków do pola. Wartość może też być zmienną, podaje

DIVANTE BUSINESS SERVICES

			się ją w formacie ${nazwa_zmiennej}.
check	Uchwyt do elementu		Sprawia, że checkbox jest zawsze zaznaczony, poprzedni stan nie ma na to wpływu.
store	Wartość do zapamiętania. Nie ma rozróżnienia czy to jest liczba czy ciąg znaków	Nazwa zmiennej	Zapamiętuje zmienną pod zadaną nazwą.
storeText	Uchwyt do elementu tekstowego. W Firebugu zaznaczone na szaro.	Nazwa zmiennej	Zapamiętuje tekst pobrany ze strony pod zadaną nazwą.
storeAtributte	Uchwyt do elementu @ nazwa atrybutu. Np. id=email@class	Nazwa zmiennej	Zapamiętuje atrybut elementu pod zadaną zmienną.
storeEval	Skrypt w języku javascript	Nazwa zmiennej	Wykonuje skrypt JS, a wynik zapisuje do zmiennej.
storeXpathCount	Uchwyt do elementu wielokrotnego	Nazwa zmiennej	Zlicza ilość liści/rozgałęzień drzewa bez zagłębiania się.
getEval	Skrypt w języku javascript		Wykonuje skrypt JS.
waitForElementPresent	Uchwyt do elementu		Czeka na załadowanie się elementu kryjącego się pod zadanym uchwytem. Nie musi być on widoczny na stronie.
waitForText	Uchwyt do elementu	Wzorzec (na jaki tekst czekać)	Czeka na wczytanie się tekstu w zadanym miejscu pasującego do wzorca.
waitForVisible	Uchwyt do elementu (jw)	Wzorzec	Czeka na uwidocznienie się

			elementu. Podobnie jak by czekał użytkownik na pojawienie się np. flayouta).

if
..
else
...
endIf
	Wyrażenie logiczne. Zmienne przekazywane w formacie storedVars.nazwa_zmiennej		Klasyczne wyrażenie warunkowe. Wymagana wtyczka SelBlocks.

for ... endFor	Zmienne przekazywane w formacie storedVars.nazwa_zmiennej i=1;i<storedVars.nazwa_zmiennej;i++		Klasyczna pętla for.
while ... endWhile	Wyrażenie logiczne. Zmienne przekazywane w formacie storedVars.nazwa_zmiennej		Klasyczna pętla while.
pause	Czas oczekiwania w ms		Pauza/sleep nic nie robienie przez X ms. Nie używać, chyba że nie ma innego wyjścia.

Selenium RC

Selenium Remote Control (tzw. Selenium 1) pozwala na pisanie oraz uruchamianie testów na dowolnej przeglądarce. Selenium RC składa się z dwóch komponentów. Pierwszym z nich jest Selenium Server, na którym uruchamiane są wszystkie wcześniej przygotowane testy. Drugim z komponentów są biblioteki klienckie oferujące interfejs pomiędzy dowolnym językiem programowania a usługami Selenium RC Server. Selenium RC przez bardzo długi czas stanowiło główny framework w rodzinie Selenium. Wspiera następujące języki programowania: Java, C#, PHP, Python, Perl, Ruby.

Zalety:

- Wsparcie dla różnych przeglądarek,
- Możliwość przeprowadzenia testów typu data-driven,
- Zdecydowanie szybsze wykonywanie testów niż w Selenium IDE.

Wady:

- Bardziej skomplikowana instalacja niż Selenium IDE,
- Wymaga wiedzy z zakresu programowania,
- Do jego obsługi potrzebny jest również serwer RC,

- Wolniejszy niż jego następca WebDriver.

W chwili obecnej Selenium RC zostało oficjalnie zastąpione przez Selenium WebDriver.

Selenium WebDriver

Selenium Wedriver powstał w wyniku połączenia Selenium z bibliotekami WebDrivera. Narzędzie to jest zewnętrzną biblioteką *.jar, która łączy w sobie sterownik do danej przeglądarki oraz serwer Selenium. WebDriver dostarcza użytkownikowi gotowe API pozwalające na interakcję z przeglądarką. Selenium WebDriver wspiera następujące języku programowania: Java, C#, PHP, Python, Perl, Ruby.

Zalety:

- Komunikacja bezpośrednio z przeglądarką,
- Duża szybkość wykonywanych testów,
- Większa stabilność od Selenium RC,
- Wsparcie dla różnych przeglądarek.

Wady:

- Wymaga znajomości jednego z języków programowania,
- Trudniejsza instalacja od Selenium IDE,
- Problem z obsługą najnowszych wersji przeglądarek.

Page Object Pattern

POP jest wzorcem, w którym każda strona to osobna klasa. Posiadanie wspólnego modelu strony dla programistów i testerów jest ideologicznie nie do zaakceptowania. Testerzy powinni sami stworzyć swój model strony, niezależnie od developerów. Dzięki temu unikniemy świadomego lub nieświadomego ukrywania błędów przez programistów. Cechą wspólną większości wersji POP jest to, że klasa-strona przyjmuje w konstruktorze drivera. Każda klas daje możliwość interakcji z nią poprzez przejrzysty zestaw metod. Jest to nie tylko zaleta w postaci czytelnego kodu, ale także możliwość wykorzystywania tak przygotowanych fragmentów w wielu scenariuszach testowych.

Ponieważ testując aplikację webową mamy zwykle do czynienia ze stronami, bardzo wygodne jest zastosowanie wzorca Page Object, aby w jednym miejscu mieć kod który odwołuje się do danej strony. Wówczas jeśli coś się zmieni na danej stronie, nie musimy zmieniać scenariuszy testowych, uaktualniamy tylko obiekt, który reprezentuje tę stronę. Page Object nie musi reprezentować całej strony, może dotyczyć tylko jej części , np. nawigacji strony.

W jaki sposób możemy wyszukiwać elementów dostępnych na stronie:

- Na podstawie unikalnego identyfikatora (id),
- Określonej klasy CSS,
- Nazwie elementu HTML,

- Nazwie,
- Selektorów CSS3,
- Tekstu bądź jego fragmentu znajdującego się w elemencie.

Podstawowe metody WebDrivera

- Metoda driver.get (String url); przenosi nas na stronę podaną jako argument. Metoda ta czeka na załadowanie całej strony.
- Jeżeli testujemy stronę, która wczytuje się w niestandardowy sposób (wykonanie js onLoad() nie jest faktycznym momentem końca ładowania strony).
- Metoda driver.quit(); zamyka wszystkie otwarte okna przeglądarki. Warto wspomnieć jeszcze o metodzie driver.close(); która zamyka obecnie aktywne okno przeglądarki (przydatne, kiedy test operuje na paru oknach/kartach).
- Aby pobrać tytuł strony, użyjemy String title = driver.getTitle();.

Oprócz WebDrivera kolejną ważną klasą jest WebElement. Obiekt WebElementu jest tożsamy z pewnym elementem strony - jakimś divem, polem tekstowym lub innym dowolnym obiektem.

Selenium GRID

Selenium Grid pozwala na równoległe wykonywanie testów na różnych komputerach oraz pod różnymi przeglądarkami w tym samym czasie. Wykorzystuje koncepcję huba i węzłów. Centralny punkt zwany hubem posiada informacje o różnych platformach testowych, czyli węzłach i przypisuje je, kiedy skrypt testowy tego wymaga.

Zalety:

- Równoległe wykonywanie testów na różnych platformach programowych oraz sprzętowych.

Wady:

- Bardziej skomplikowana konfiguracja,
- Wymagana odpowiednia infrastruktura.

10. Słownik pojęć podstawowych

Pojęcia znajdujące się w tej części zostały zaczerpnięte z dokumentu **„Słownik wyrażeń związanych z testowaniem"** wydanym przez Stowarzyszenie Jakości Systemów Informatycznych w roku 2013 (Wersja 2.2.2). We wspomnianym słowniku znajdziesz też dużo więcej pojęć związanych z testowaniem oprogramowania.

Analiza pokrycia - pomiar pokrycia osiągnięty podczas wykonywania testów wg z góry określonych kryteriów, przeprowadzany w celu określenia czy potrzebne są dodatkowe testy; jeśli odpowiedź brzmi tak, to podejmowana jest decyzja, które przypadki testowe należy wykonać.

Analiza ryzyka – proces oceny zidentyfikowanych ryzyk mający na celu oszacowanie ich wpływu i prawdopodobieństwa ziszczenia się.

Atak na oprogramowanie – ukierunkowane działanie mające na celu ocenę jakości, w szczególności niezawodności obiektu testów, poprzez wymuszanie wystąpienia określonej awarii.

Bezpieczeństwo – zdolność oprogramowania do osiągania akceptowalnych poziomów ryzyka wystąpienia szkody w stosunku do ludzi, biznesu, oprogramowania, majątku lub środowiska w określonym kontekście użycia.

Błąd – działanie człowieka powodujące powstanie nieprawidłowego rezultatu.

Cykl testowy – wykonanie procesu testowego w stosunku do pojedynczego, możliwego do zidentyfikowania wydania testowanego obiektu.

Cykl życia oprogramowania – okres rozpoczynający się, kiedy pojawi się pomysł na oprogramowanie i kończący się, gdy oprogramowanie jest już dostępne do użytku. Zazwyczaj cykl życia oprogramowania zawiera fazę koncepcji, fazę wymagań, fazę projektowania, fazę implementacji, fazę testów, fazę instalacji i zastępowania, fazę wykorzystania produkcyjnego i pielęgnowania oraz – czasami – fazę wycofania.

Dane testowe – dane, które istnieją (przykładowo w bazie danych) przed wykonaniem testu i które mają wpływ na testowany moduł lub system lub na które wywiera wpływ testowany moduł lub system.

Debugowanie – proces wyszukiwania, analizowania i usuwania przyczyn awarii oprogramowania. Debugowanie jest czynnością wykonywaną przez programistów.

Defekt – wada modułu lub systemu, która może spowodować, że moduł lub system nie wykona zakładanej czynności, np. niepoprawne wyrażenie lub definicja danych. Defekt, który wystąpi podczas uruchomienia programu, może spowodować awarię modułu lub systemu.

Faza testów – wyróżniony zbiór aktywności testowych zebrany w podlegającą zarządzaniu fazę projektu, np. wykonanie testów na jakimś poziomie testów.

Gęstość usterek – liczba usterek znalezionych w module lub systemie przypadająca na jednostkę wielkości modułu lub systemu (wyrażoną za pomocą standardowej miary oprogramowania, np. w liniach kodu, liczbie klas lub punktach funkcyjnych).

Harmonogram testów – lista aktywności, zadań lub zdarzeń z procesu testowego, określająca ich zamierzoną datę rozpoczęcia i zakończenia i/lub czas realizacji oraz ich współzależności.

Incydent – każde zdarzenie wymagające zbadania.

Infrastruktura testu – organizacyjne artefakty niezbędne do przeprowadzenia testu, składające się ze środowisk testowych, narzędzi testowych, wyposażenia biurowego i procedur.

Integracja – proces łączenia modułów lub systemów w większe zespoły.

Jakość oprogramowania – ogół funkcjonalności i cech oprogramowania, które charakteryzują zdolność zaspokajania stwierdzonych lub przewidywanych potrzeb.

Kryterium wejścia – zbiór ogólnych i specyficznych warunków, których spełnienie jest wymagane do kontynuacji procesu dla określonego zadania, np. fazy testów. Celem kryterium wejścia jest ochrona przed rozpoczęciem zadania, w sytuacji, gdy pociąga to za sobą więcej (zmarnowanych) nakładów pracy w porównaniu z nakładem pracy potrzebnym do osiągnięcia stanu spełnienia kryterium wejścia.

Kryterium wyjścia – zbiór ogólnych i specyficznych warunków, uzgodnionych z udziałowcami, których spełnienie jest wymagane do oficjalnego zakończenia procesu. Celem kryterium wyjścia jest ochrona przed uznaniem zadania za ukończone w przypadku, gdy jakieś jego elementy nie są jeszcze w pełni wykonane. Kryteria wyjścia są stosowane jako argument przeciwko zakończeniu testów oraz do planowania, kiedy można to zrobić.

Małpie testowanie – metoda testowania polegająca na losowym wyborze z szerokiego zakresu wejść i losowym naciskaniu przycisków, ignorując sposób, w jaki produkt powinien być używany.

Maskowanie defektów – sytuacja, w której występowanie jednego defektu uniemożliwia wykrycie innego.

Podejście do testu – implementacja strategii testów dla konkretnego projektu. Zwykle zawiera decyzje podjęte na podstawie celów i analizy ryzyka projektu (testowego), punkty startowe procesu testowego, techniki projektowania testu do wykorzystania, kryteria wyjścia i typy testu do wykonania.

Podstawowy zestaw testów – zestaw przypadków testowych powstały na podstawie wewnętrznej struktury modułu lub specyfikacji, który zapewnia osiągnięcie 100% określonego kryterium pokrycia.

Pokrycie – proces przypisania liczby bądź kategorii do obiektu mający na celu opisanie danej właściwości obiektu.

Pokrycie kodu – metoda analityczna, określająca które części programu zostały wykonane (pokryte) przez zestaw testowy oraz które części nie zostały wykonane, np. pokrycie instrukcji kodu, pokrycie decyzji, pokrycie warunków.

Pokrycie ścieżek – odsetek ścieżek w module wykonanych przez zestaw testowy. Realizacja 100% pokrycia ścieżek oznacza 100% pokrycie LSKiS.

Proces testowy – podstawowy proces testowy składa się z następujących faz: planowanie testów i kontrola, analiza i projektowanie testów, implementacja i wykonanie, ocena kryteriów wyjścia i raportowanie oraz czynności związane z zakończeniem testów.

Przypadek testowy – zbiór danych wejściowych, wstępnych warunków wykonania, oczekiwanych rezultatów i końcowych warunków wykonania opracowany w określonym celu lub dla warunku testowego, jak wykonanie pewnej ścieżki programu lub zweryfikowanie zgodności z konkretnym wymaganiem.

Przypadek użycia – ciąg transakcji w dialogu pomiędzy użytkownikiem a systemem z namacalnym rezultatem.

Retestowanie – testowanie polegające na uruchomieniu przypadków testowych, które podczas ostatniego uruchomienia wykryły błędy, w celu sprawdzenia poprawności naprawy.

Rezultat fałszywie niezaliczony – test, w którym defekt został zaraportowany, chociaż defekt ten wcale nie występuje.

Rezultat fałszywie zaliczony – test, w którym nie zidentyfikowano obecności występującej w testowanym obiekcie usterki.

Ryzyko – czynnik, który w przyszłości może skutkować negatywnymi konsekwencjami; zazwyczaj opisywany jako wpływ oraz prawdopodobieństwo.

Rzeczywisty rezultat – wytworzone zaobserwowane zachowanie się modułu lub systemu podczas testowania tego modułu lub systemu.

Skrypt testowy – powszechnie używana nazwa specyfikacji procedury testowej, zwłaszcza automatycznej.

Specyfikacja testów – dokument zawierający specyfikację projektu testów, specyfikacje przypadków testowych i/lub specyfikację procedury testowej.

Spotkanie retrospektywne – spotkaniemna końcu projektu, podczas którego członkowie zespołu projektowego oceniają projekt i wyciągają wnioski, które mogą być wykorzystane w następnym projekcie.

Stabilność – zdolność produktu oprogramowania do unikania niespodziewanych zachowań z modyfikacji w oprogramowaniu.

Test dymny – podzbiór wszystkich zdefiniowanych/zaplanowanych przypadków testowych, które pokrywają główne funkcjonalności modułu lub systemu, mający na celu potwierdzenie, że kluczowe funkcjonalności programu działają, bez zagłębiania się w szczegóły. Codzienne budowanie i testy dymne stanowią dobre praktyki wytwarzania oprogramowania.

Test wstępny – szczególny rodzaj testu dymnego mający na celu podjęcie decyzji, czy moduł lub system jest gotowy do dalszego szczegółowego testowania. Najczęściej jest wykonywany na początku fazy wykonywania testów.

Testalia – wszystkie dokumenty i narzędzia (artefakty) wytworzone i używane podczas procesu testowania niezbędne do planowania, projektowania i wykonywania testów, takie jak dokumentacja, skrypty, wejścia, oczekiwane rezultaty, procedury, pliki, bazy danych, środowiska oraz każde dodatkowe oprogramowanie i narzędzia użyte podczas testowania.

Testowanie dokumentacji – kontrola jakości (dokładności, prawidłowości, kompletności itp.) dokumentacji, np. podręcznika użytkownika lub opisu instalacji.

Testowanie dopasowania – proces testowania mający zapewnić dopasowanie oprogramowania do potrzeb.

Testowanie interfejsu – testowanie wykonywane w celu wykrycia błędów w interfejsach pomiędzy modułami.

Testowanie kombinatoryjne – sposób na identyfikację odpowiedniego podzbioru kombinacji testów dla osiągnięcia uprzednio zdefiniowanego poziomu pokrycia, gdy testujemy obiekt wieloparametrowy i gdy każdy z tych parametrów ma kilka wartości, co powoduje, że mamy więcej kombinacji, niż można przetestować w zadanym czasie.

Testowanie konsultacyjne – testowanie prowadzone przy współpracy i pod nadzorem odpowiednich ekspertów biznesowych spoza zespołu testowego (ekspertów technologicznych i/lub ekspertów biznesowych).

Testowanie konwersji – Testowanie programów używanych do przenoszenia danych z istniejących systemów do systemów je zastępujących.

Testowanie negatywne – testowanie, którego celem jest pokazanie, że oprogramowanie nie działa. Testowanie negatywne jest bardziej związane z postawą testerów niż ze specyficznym podejściem czy techniką projektowania testów, np. testowanie z błędnymi wartościami wejściowymi lub wyjątkami.

Testowanie parami – testowanie, w którym dwie osoby, np. dwóch testerów, programista i tester lub użytkownik końcowy i tester, pracują wspólnie w celu znalezienia błędów. Zwykle podczas testowania osoby te współdzielą jeden komputer.

Testy ad-hoc (zwane też testami eksploracyjnymi) – jest to rodzaj testów, które są wykonywane bez formalnego planu testów, dokumentacji oraz przypadków użycia. Jest to najmniej formalna metoda testowania. Wykonywanie takich testów pomaga testerom nauczyć się aplikacji przed rozpoczęciem zaplanowanych rodzajów testów.

Testy end-to-end – jest to rodzaj testów sprawdzający, czy cały „flow" tworzonej aplikacji jest działający zgodnie z założeniami od początku do końca. Testy te potwierdzają, że aplikacja spełnia oczekiwania użytkownika końcowego

Użyteczność – zdolność oprogramowania do bycia używanym, zrozumiałym, łatwym w nauce i atrakcyjnym dla użytkownika, gdy oprogramowanie to jest używane w określonych warunkach.

Walidacja – sprawdzanie poprawności i dostarczenie obiektywnego dowodu, że produkt procesu wytwarzania oprogramowania spełnienia potrzeby i wymagania użytkownika.

Współczynnik awarii – stosunek liczby awarii w danej kategorii do określonej jednostki miary, np. awarie na jednostkę czasu, awarie na liczbę transakcji, awarie na liczbę uruchomień komputera.

Wstrzykiwanie defektów – proces zamierzonego dodawania defektów do systemu w celu wykrycia, czy system może wykryć defekt i pracować mimo jego występowania. Wstrzykiwanie błędów stara się imitować błędy, które mogą wystąpić w produkcji.

Wydajność – stopień, w jaki system lub moduł, realizuje swoje wyznaczone funkcje w założonych ramach czasu przetwarzania i przepustowości.

Zablokowany przypadek testowy – przypadek testowy, który nie może zostać wykonany, ponieważ jego warunki wstępne nie mogą zostać osiągnięte.

Zarządzanie defektami – proces składający się z rozpoznania, analizy, prowadzenia działań i likwidacji usterek. Polega on na rejestracji usterek, ich klasyfikacji oraz określaniu wpływu defektów.

Zarządzanie incydentami – Proces składający się z rozpoznania, analizy, podejmowania działań i rozwiązywania incydentów. Polega on na rejestracji klasyfikacji oraz określaniu wpływu incydentów.

Zarządzenia jakością – ogół skoordynowanych czynności mających na celu kierowanie organizacją i kontrolowanie jej pod kątem jakości. Zwykle obejmuje czynności takie jak: zdefiniowanie polityki jakościowej i celów jakościowych, planowanie jakości, kontrolowanie jakości, zapewnienie jakości i poprawa jakości.

Zarządzanie ryzykiem – systematyczne wdrażanie procedur i praktyk dla zadań identyfikacji, analizowania, ustalania priorytetów i kontrolowania ryzyka.

DIVANTE
BUSINESS SERVICES

Bibliografia

1. http://www.testowanie.net/automatyzacja/rodzaje-testow-wydajnosciowych/
2. https://testerprof.wordpress.com/tag/testy-wydajnosciowe/
3. https://www.kainos.pl/blog/wprowadzenie-do-testow-automatycznych-czesc-2/
4. http://wazniak.mimuw.edu.pl/images/7/70/Io-11-wyk.pdf
5. http://przemek.yum.pl/testy-automatyczne-selenium-cz1-selenium-ide/
6. http://www.droptica.pl/blog/selenium-ide-krok-w-strone-automatyzacji
7. http://wrotqa.org/wp-content/uploads/2014/03/selenium-podstawy-natialia-krawczyk-wrotqa.pdf
8. http://www.wstt.edu.pl/pliki/materialy/mta/metodologia_testowania_aplikacji_c2.pdf
9. http://autentika.pl/blog/automatyzacja-testow-serwisu-internetowego
10. http://itcraftsman.pl/testy-automatyczne-interfejsu-uzytkownika-przy-uzyciu-selenium-ide/
11. http://rst.com.pl/rst-blog/automatyzacja-testow-selenium/
12. http://qa-24.pl/poczatki-selenium-ide/
13. http://www.inflectra.com/Partners/Articles/SoftwareDeveloperJournal_SpiraTeamALM_Softlab_PL.pdf
14. http://software-testing-tutorials-automation.blogspot.in/2013/07/steps-of-running-selenium-ide-test.html
15. http://www.codeproject.com/Questions/307417/To-install-Selenium-IDE-in-Internet-Explorer
16. http://docs.seleniumhq.org/projects/ide/
17. http://docs.seleniumhq.org/projects/webdriver/
18. książka: Sztuka testowania oprogramowania, autorzy: Glenford J. Myers, Corey Sandler, Tom Badgett, Todd M. Thomas, Wydawnictwo Helion, 2005r.
19. Słownik testerski dla Poziomu Podstawowego i Zaawansowanego + tester zwinny (v.2.3) http://sjsi.org/wp-content/uploads/2014/10/slownik-termin%C3%B3w-testowych-ver-2.3-PL.pdf
20. Sylabus ISTQB, poziom podstawowy http://sjsi.org/wp-content/uploads/2013/08/sylabus-poziomu-podstawowego-wersja-2011.1.1_20120925.pdf