Quale

Quale 4/2014

Wydanie pokonferencyjne

TESTWAREZ

Spis treści

TESTOWANIE OPROGRAMOWANIA

4. Relacja z Testwarez
Krzysztof Chytła

7. Automatyczne testy w Agile to nie wszystko
Remigiusz Dudek

15. Behaviour Driven Development. Moja podróż
Bartosz Szulc

29. Testy akceptacyjne w administracji publicznej – pro publico bono?
Michał Kruszewski

39. Pokaz możliwości narzędzia Microsoft Test Manager
Marta Firlej

49. Zautomatyzuj swoje testy automatyczne oparte na Selenium
Michał Sierzputowski

REDAKCJA

Redaktor naczelny
Karolina Zmitrowicz
karolina.zmitrowicz@quale.pl

Zastępca redaktora naczelnego:
Krzysztof Chytła
krzysztof.chytla@quale.pl

Redaktorzy:
Bartłomiej Prędki
bartlomiej.predki@quale.pl
Michał Figarski
michal.figarski@quale.pl

Współpraca:
Tomasz Olszewski
tomasz.olszewski@quale.pl
Piotr Poznański
piotr.poznanski@quale.pl
Zuzanna Gościcka-Miotk
z.goscicka@gmail.com

Website:
www.qualemagazine.com (ENG)
www.quale.pl (PL)

Facebook:
http://www.facebook.com/qualemagazine

Spis treści

TESTOWANIE OPROGRAMOWANIA

59. Optymalizacja automatycznych testów regresywnych w organizacji transformującej do Agile
Adam Marciszewski

71. Słowa kluczowe jak góry lodowe – czyli rzecz o bibliotekach testowych
Marcin Kowalczyk

81. Testowanie Mutacyjne? Proste!
Michał Chudy

95. Gimbaza testerów
Bogdan Bereza

101. Co wiemy, a czego jeszcze nie o TestingCup 2015?
Joanna Moćko

104. Inspired by Quality

Krzysztof Chytła

Relacja z Testwarez

TestWarez 2014 za nami – dwa dni konferencji o programie wypakowanym atrakcyjnymi wykładami, prelekcjami, warsztatami i panelami dyskusyjnymi strzeliło jak z bicza. Aż żal było wyjeżdżać z Gdyni z perspektywą niemal 6 godzin za kółkiem na trasie do Wrocławia. Na szczęście w doborowym towarzystwie redaktora Prędkiego wśród kawałków Queen. Niemal czułem wiatr we włosach pędząc na złamanie karku autostradą – Headlong!

Już sama różnorodność formy zasługuje na uwagę i pochwałę – każdy, w zależności od poziomu zaawansowania, mógł znaleźć coś dla siebie, w czym z pewnością pomagał zmyślnie zaprojektowany identyfikator uczestnika z rozkładem jazdy na odwrocie. Tutaj wkradł się mały chochlik – oba dni były opisane, jako poniedziałki, a dwa wystąpienia zmieniły termin, ale nie było to żadne wyzwanie dla zawodowych testerów, których przybyło ok. 300. Organizatorzy popisali się czujnością oraz refleksem i szybko rozwiali wątpliwości.

Skoro jesteśmy przy organizatorach, to należą im się gratulacje nie tylko za refleks, ale przede wszystkim za całokształt. Sprostali wyzwaniu organizacji unikalnej konferencji na lądzie (PPNT) i morzu (prom Gdynia-Karlskrona-Gdynia)! Ciekawi prelegenci, dobry catering, dopięty transport, wyposażone sale oraz świetna wieczorna integracja w rytmach muzyki szanty. „Mobilis in mobili" w pełnej krasie. Kto nie był, niech żałuje!

Poniżej chciałbym podzielić się wrażeniami z uczestnictwa w poszczególnych gigach. Warsztaty Karoliny Zmitrowicz i Radosława Smilgina pod tytułem „Testowanie wymagań" przyciągnęły więcej chętnych niż mogła pomieścić sala (i wcześniejsze zapisy). Szczęśliwie miejsca przy stołach było dość, a dodatkowe krzesła udało się załatwić. Warto było powalczyć o uczestnictwo, bo zabawa w dobrego i złego policjanta, a właściwie policjantkę, pomogła w bardzo przystępny sposób przybliżyć wyzwania, przed jakimi stają testerzy i analitycy w każdym projekcie. Rozważany na warsztatach „problem biznesowy do rozwiązania" - sterownik bramy garażowej – pokazał ważność ustalenia słownika i zakresu projektu już na samym początku. Sterownik okazał się dla jednych (w tym prowadzących) pilotem, a dla drugich całym syste-

mem z siłownikami i radioodbiornikiem. Znajomy dysonans?

Na warsztatach można było dowiedzieć się jak wymagania dzielą się na poziomy i co z tego wynika, jaki jest ich cykl życia w projekcie oraz jak nie zakopać się w szczegółach implementacyjnych, a skupić się na tym „co" i „jak" ma robić system.
Lekcja: piszmy testowalne wymagania i nie bójmy rozmawiać się o niejasnościach. Jakość systemu może na tym tylko zyskać.

Seretta Gamba i jej „Automation through the back door" okazało się rozczarowaniem. Spodziewałem się spektakularnego success story przyprawionego know-how ekspertki domeny automatyzacji z imponującym doświadczeniem, a dostałem… prezentację na poziomie podstawowym, z materiałami częściowo po niemiecku (zrzuty z narzędzia), ale za to z ukrytymi kotami (podobno było ich przeszło 30, których a słuchacze mieli je wyszukiwać podczas prelekcji). To nie tak miało być, zupełnie nie tak [Budka Suflera]. Szkoda, że Dorothy Graham nie dotarła. Cała nadzieja w Tilo Linz.

Lekcja: Automatyzować każdy może – jeden lepiej, drugi trochę gorzej [Jerzy Stuhr]. Zabrakło konkretów.

„Nietypowe podejście do automatyzacji w systemie rozproszonym" Daniela Deca, to zupełnie co innego. Framework do weryfikacji stanów systemu i ich zmian żywo zainteresował uczestników, w tym mnie. Prowadzący wyszedł spod gradu pytań obronną ręką. Podejście do automatyzacji zaprezentował zaprawdę nietypowe – przygotowane na życzenie klienta dla systemu transakcyjnego. Dobry kawałek inżynierskiej roboty.

Lekcja: Czasem to zmiana stanu i jej okoliczności jest bardziej interesująca z punktu widzenia testowania, niż sam stan. Interesujące podejście.

„Eksploracja nieznanego" z Remigiuszem Dudkiem to – merytorycznie – najlepsze, co mnie spotkało na ten konferencji, mimo że uczestniczyłem dosłownie z progu sali konferencyjnej. Doświadczenie prowadzącego i jego zdolności przekazywania wiedzy biły na kilometr, także odbiór był wyśmienity. Materiał pokrywał tematy od wytłumaczenia idei, systematycznego przygotowania, przez techniki eksploracji, po psychologię procesu poznawczego oraz osadzenie całego rozwiązania w procesie wytwarzania i testowania oprogramowania. Rewelacja.

Lekcja: Czas okrasić swoje testy przemyślaną eksploracją. Teraz wiem jak wyposażyć się na taką wyprawę.

Kolejny punkt programu. Tomasz Osojca wie jak zachęcić uczestników do żywej dyskusji. Dał temu dowód prowadząc panel o inteligencji w testowaniu; umiejętnie grając tony odbijające się echem na dumie zawodowej każdego testera. Zwinne podejście, lekka forma, ciągła interakcja z uczestnikami i sprawna szermierka markerem na szklanej tablicy zapewniły wartką wymianę zdań i opinii.

Lekcja: Wkładajmy głowę w testowanie, powtarzalną resztą zajmą się automaty.
Agnieszkę Skokowską i Justynę Rybarczyk mieliśmy już okazję gościć na łamach magazynu Quale. „Jak złapać testera?" było ciekawie zapowiadającym się panelem poruszającym bolączki nękające niemal każdego kierownika budującego zespół lub kierownika projektu alokującego zasoby

ludzkie w projektach realizowanych przez firmę. Choć początkowo szło niemrawo (same podbramkowe sytuacje z wielką ilością ograniczeń), to dyskusja ruszyła w dobrym kierunku zahaczając o kwestie rozwoju zawodowego testera, przywiązanie do firmy, staże i outsourcing. Żeby zejść głębiej zabrakło na sali doświadczonych kierowników projektów czy liniowych, ale to nie dla nich był TestWarez.

Lekcja: Captain Obvious powiada, że na bazie wiedzy domenowej i ekspertyzy testowej da się zbudować zespół, jednak to kosztuje. Szanuj testera swego i inwestuj w rozwój jego.

Na koniec, zgodnie z porządkiem chronologicznym, zostały wrażenia z prelekcji Tilo. Było futurystycznie, ale lekko sennie. Dynamika wypowiedzi przypominała mi Jeffa Bezosa prezentującego nową zabawkę Amazona - nie porwała tłumów. (Notabene, chyba dawno już nie widziałem tylu umęczonych nieustanną wymianą wiedzy i poglądów twarzy. Przypisywanie całej zasługi prowadzącemu byłoby jednak niesprawiedliwe. Chwała organizatorom za wodę mineralną!) Zaciekawiło mnie ujęcie testowania na Gartnerowskim hype cycle oraz przedstawienie scenek rodzajowych z AI SI czy bioniką w tle. A Ty, co byś sobie dał/-a wszczepić dla testowania? Ja pewnie wszystko, żeby tylko przetestować.

Raczej niewielu podzielało mój entuzjazm, co zostało zmierzone przy pomocy szybkiego głosowania. Powiedziałbym – łagodny finisz.
Lekcja: Kolejny model do skolekcjonowania. Być sprzedawcą idei jak Steve Jobs – bezcenne, za wszystko inne (w tym polecane przez wielu książki Tilo) zapłacisz kartą znanej firmy.

Dla mnie największą wartością dodaną były rozmowy w kuluarach z prelegentami i nie tylko. Ciekawi ludzie, ciekawe pomysły, ciekawa konferencja. Warto było. Do zobaczenia za rok!

AUTOR

Krzysztof Chytła

Kierownik, projektant i automatyk testów z wieloletnim doświadczeniem w domenie systemów wbudowanych. Zdobywał doświadczenie w dużych projektach o międzynarodowym zasięgu, zapewniając najwyższą jakość produktu. Tester z krwi i kości z ciekawością i wnikliwością analizujący prężnie rozwijający się świat nowych technologii. Absolwent Wydziału Elektroniki Politechniki Wrocławskiej. Trener i coach, pasjonat zdobywania i dzielenia się wiedzą.

W życiu prywatnym fan literatury fantastycznej, fantastyczno-naukowej i gier planszowych przy szklaneczce zacnej whisky - najlepszej przyjaciółce redaktora.

Remigiusz Dudek

Automatyczne testy w Agile to nie wszystko

Wprowadzenie

Przez ostatnie lata zwinne testowanie i testowanie automatyczne stały się swego rodzaju synonimami. Podczas gdy w lekkich metodykach faktycznie nie sposób obyć się bez testów automatycznych, odnoszę wrażenie, że zbyt mały nacisk kładziemy na pozaautomatyczne aspekty testowania, w szczególności eksploracyjne testy manualne.

Celem zarówno testów unitowych, jak i automatycznych testów akceptacyjnych jest zapewnienie zespołowi inżynierów oprogramowania „siatki bezpieczeństwa", niezbędnej przy pracy w wysoce zmiennym środowisku, wymagającym ciągłego refaktoryzowania już istniejącego kodu – testy te mają zatem rolę testów regresyjnych.

Pozostają jednak pytania:

- W jaki sposób przetestować rzetelnie nową funkcjonalność (mając w świadomości skomplikowane zależności pomiędzy różnymi modułami naszej aplikacji)?
- W jaki sposób zapewnić, że wpasowuje się ona w już istniejący model biznesowy?
- W jaki sposób na nowo wyćwiczyć w sobie krytyczne spojrzenie na oprogramowanie stworzone własnymi rękami?

Celem tego artykułu jest przekonanie czytelników, że testy eksploracyjne nadają się wyśmienicie do powyżej zdefiniowanych celów, jednocześnie nie poddając się zupełnie automatyzacji.

Continuum testów

Testy każdego typu możemy umieścić na swego rodzaju continuum. Na jego jednym końcu znajdą się przypadki testowe, dokładnie opisane za pomocą skryptu – mają one bardzo precyzyjnie zdefiniowane kroki oraz punkty weryfikacji, dzięki którym możemy zdecydować, czy test zakończył się sukcesem, czy też nie. Na drugim końcu tego continuum znajdują się testy bez dokładnie zdefiniowanych kroków. Testy, w których kluczową rolę odgrywa pytanie: „a co się stanie, gdy...?". Testy, w których kolejny krok zależy bezpośrednio od wyniku, jaki otrzymaliśmy w kroku poprzednim. Wchodząc na teren testów eksploracyjnych, musimy posłużyć się przeczuciem, doświadczeniem, heurystykami, ale przede wszystkim znajomością domeny biznesowej.

Przygotowania do drogi – określenie celu

Doskonałą analogią do testów eksploracyjnych jest wyprawa poprzez nieodkryte dotychczas tereny. Na takiej wyprawie bardzo łatwo się zgubić, dlatego ważne jest, aby wytyczyć sobie jasny cel i w ciągły sposób kontrolować, czy aby za bardzo nie zbaczamy. Jest w tym wszystkim pewna sprzeczność: skoro jest to nieznany teren, to w jaki sposób możemy zdefiniować cel takich testów? Przecież nie wiemy, co możemy tam znaleźć.

Wszystkie techniki określania celów testów eksploracyjnych opierają się na słuchaniu. Uważnie słuchaj pytań, które co jakiś czas na pewno pojawiają się w Twoim zespole albo... w Twojej głowie:

- „ciekawe, co się stanie, gdy....?"
- „wiecie, jak się zachowa system, gdy...?"

„Wyobraźcie sobie, że budzicie się rano, jecie śniadanie, popijając kawą. Wychodząc do pracy, jak codziennie przechodzicie koło kiosku z gazetami, lecz tym razem waszą uwagę przykuwa wielki nagłówek w jednej z gazet. Na pierwszy plan wybija się czerwoną czcionką napisana nazwa waszej firmy oraz produktu. Co dokładnie mówi nagłówek?"

- „trzeba by sprawdzić, czy przypadkiem...”

Można by przytoczyć wiele innych, podobnych pytań. Ważne, by je spisać i przekuć w zadania na tablicy scrumowej, w przeciwnym wypadku zostaną zapomniane tak szybko, jak się pojawiły.

Innym ciekawym sposobem na zdefiniowanie tematów do eksploracji jest tzw. gra w „koszmarne nagłówki”. Zgromadź w jednym pokoju ludzi zaangażowanych w tworzenie danej funkcjonalności, a następnie nakreśl następującą historię:

„Wyobraźcie sobie, że budzicie się rano, jecie śniadanie, popijając kawą. Wychodząc do pracy, jak codziennie przechodzicie koło kiosku z gazetami, lecz tym razem waszą uwagę przykuwa wielki nagłówek w jednej z gazet. Na pierwszy plan wybija się czerwoną czcionką napisana nazwa waszej firmy oraz produktu. Co dokładnie mówi nagłówek?”.

Wyniki tej gry potrafią być bardzo zaskakujące.

Przygotowania do drogi – każda podróż zaczyna się od pierwszego kroku

Załóżmy zatem, że cel mamy już określony, teraz musimy wykonać pierwszy krok w jego kierunku. Wybór odpowiedniego punku początkowego nie zawsze należy do prostych. W dzisiejszych czasach, gdy architektura oparta jest często na serwisach, nasze oprogramowania pozwala na wykonanie dowolnej czynności na co najmniej kilka sposobów. Zatem kluczowe jest zrozumienie wszystkich interfejsów, jakie udostępnia nasza aplikacji, jest również zdefiniowanie tzw. granic zaufania. Umożliwi nam to skupienie się na wartościowych wariacjach parametrów wejściowych i nieskupianie się na wartościach w 100% kontrolowanych przez aplikację (oczywiście musimy się upewnić, że istnieją testy automatyczne, które zapewniają pokrycie takich wariacji).

Nieocenioną pomocą w zrozumieniu skomplikowanych programów (zbiorów serwisów) są ich modele w postaci diagramów przepływu informacji. Ważne jest, by zdać sobie sprawę, że testowanie rozpoczyna się już na etapie tworzenia diagramu. Jest to pierwszy moment, w którym tworzymy sobie model odpowiedzialności poszczególnych modułów i jednocześnie badamy, czy odpowiedzialności te są spójnie i jasno określone – to na tym właśnie etapie zaczyna się eksploracyjne testowanie architektury systemu. Aktualnie w znaczącej większości aplikacje powstają w sposób przyrostowy - czy nam się to podoba, czy też nie, architektura naszych aplikacji również powstaje w ten sam sposób (wielu nie zgodziłoby się z określeniem „architektura”). Testowanie eksploracyjne ma między innymi zapewnić, że:

- granice odpowiedzialności poszczególnych modułów/serwisów nie zostały naruszone
- w aplikacji nie tworzą się wąskie gardła
- wprowadzone konwencje są respektowane
- komunikacja pomiędzy poszczególnymi modułami przebiega w sposób spójny (np. komunikacja z bazą danych zawsze następuje za pośrednictwem DAO, jeśli jakiś moduł utrzymuje bezpośrednio połączenie z bazą jest to wbrew konwencji

- pamiętajmy, że brak konwencji w bardzo krótkim czasie prowadzi do nieutrzymywalnego kodu)
- publiczne interfejsy klas/modułów opisują rzetelnie ich odpowiedzialności
- testy jednostkowe dokładnie opisują kontrakt zawierany z daną klasą
- model biznesowy używany w aplikacji jest spójny i przejrzysty
- odpowiedzialności biznesowe zgromadzone w encjach są spójne i faktycznie należą do danej encji
- value-objects nie posiadają logiki biznesowej
- analitycy biznesowi są w stanie zrozumieć diagram klas i potwierdzić, że zależnościom pomiędzy klasami odpowiadają faktyczne zależności pomiędzy danymi koncepcjami biznesowymi

Aby osiągnąć wszystkie powyższe cele niezbędna jest umiejętność tworzenia modeli działania naszej aplikacji. Powiedzmy zatem kilka słów o tworzeniu samych modeli:

- Nie od razu Rzym zbudowano, toteż nie starajmy się modelować całości aplikacji. Wybraliśmy sobie już cel naszych testów eksploracyjnych, a zatem skupmy się tylko na aspektach powiązanych bezpośrednio z celem.
- Wiedza w obecnych projektach rzadko spoczywa w jednej głowie, jak najszybciej pokazujmy/poddawajmy ocenie nasz model.
- Nade wszystko pamiętajmy: nie ma modeli w 100% poprawnych (też nie jest to celem), niektóre natomiast są użyteczne. Nie starajmy się szlifować modelu i dodawać do niego coraz to drobniejsze szczegóły implementacyjne – jest to zazwyczaj strata czasu.

W drodze – sidła i pułapki

Z mojego doświadczenia wynika, że jest kilka rodzajów sideł, w które możemy wpaść eksplorując nasze oprogramowanie:

- Oczywistość – wykorzystywanie testów eksploracyjnych w sytuacjach, w których powinniśmy użyć klasycznych testów skryptowych - są to sytuacje, w których zachowanie systemu jest jednoznacznie określone. Jeśli w trakcie testów eksploracyjnych zadajesz pytania analitykowi biznesowemu i otrzymujesz natychmiastowe odpowiedzi, oznacza to, że najprawdopodobniej używasz testów eksploracyjnych w obszarze, który powinien być pokryty testami skryptowymi – najlepiej automatycznymi.
- Quasi-oczywistość – innymi słowy, coś wydaje się oczywiste, ale tak naprawdę takie nie jest. Te sidła najczęściej zastawiane są w dużych projektach (ponad 20-30 osób robiących jednocześnie zmiany w aplikacji). Aby uzmysłowić sobie zasadę działania tej pułapki, przeprowadźmy prostą symulację rzeczywistości. Dostarczenie funkcjonalności trwa tydzień, w trakcie tego tygodnia 30 osób zmienia nasze oprogramowanie. A zatem po tygodniu w naszej aplikacji jest osobotydzień zmian zrobionych przez nas oraz 30 osobotygodni innych zmian. Teraz, by umożliwić łatwiejsze wyobrażenie sobie tej sytuacji, przenieśmy te wielkości na dwuosobowy projekt – to tak, jakbyśmy stworzyli przez tydzień naszą funkcjonalność, a następnie udali się na 7-8 miesięcy urlopu, podczas których nasz kolega dzielnie i wytrwale zmienia nasz produkt. Odpowiedzmy sobie na pyta-

nie: co będzie dla nas wciąż oczywiste po 7-8 miesiącach nieobecności w projekcie? Niestety o tym, że wpadliśmy w te sidła dowiadujemy się zazwyczaj poniewczasie – kiedy określony defekt ujawnia się w dalszych fazach życia oprogramowania, często niestety na produkcji.

- Pokuszenie – kiedy już nauczymy się odróżniać rzeczy oczywiste od quasi-oczywistych, w trakcie eksploracji zaczniemy dostrzegać mnogość ścieżek, które kuszą, by nimi podążyć. Niektóre mogą tylko delikatnie, chwilowo zwieść nas z początkowo obranego celu. Niestety, inne mogą wieść w niezbadane otchłanie naszego oprogramowania – po tym, jak obudzi się w nas dusza badacza, bardzo trudno się oprzeć pokusie podążenia do takiej „króliczej nory". Jeśli po całym dniu eksploracji nie jesteśmy w stanie stwierdzić, że udało nam się dojść do celu, to znaczy, że albo wytyczyliśmy sobie zbyt odległy/szeroki cel, albo wpadliśmy w sidła pokuszenia.
- Potwierdzenie – te sidła zastawiamy sami na siebie. Nasz mózg robi wiele uproszczeń i raczej nie jest chętny do wykonywania pracy w swoim mniemaniu raz już wykonanej. Innymi słowy- jeśli jesteśmy przekonani, że coś jest prawdą (wykonaliśmy pracę związaną z zbudowaniem tego przekonania), to mózg niechętnie będzie chciał zmieniać to przekonanie. Jego działanie będzie nawet bardziej podstępne – otóż będzie miał tendencję do selektywnego postrzegania tylko tych faktów, które potwierdzają raz powzięte założenie i pomijania tych, które mogą temu założeniu przeczyć. Innymi słowy- jeśli wydaje Ci się że jakaś funkcjonalność działa, nawet nie zaczynaj eksplorować i tak nic nie znajdziesz (i to nie dlatego że nie ma nic do znalezienia). W psychologii nazywa się to efektem potwierdzenia.

Software'owe rozstaje – którędy do celu?

Testowanie eksploracyjne nie jest łatwe, jeśli jednak nie mamy dostępu do kodu, jest ono jeszcze trudniejsze i musimy się zdać na naszą intuicję oraz różnego rodzaju heurystyki. Posłużmy się raz jeszcze analogią do wyprawy w nieznane. Na wyprawie to bezpośrednio my podejmujemy decyzję odnośnie do ścieżki, którą będziemy podążać. Testowanie oprogramowania ma subtelniejszą naturę. Tutaj wyboru również dokonujemy my, lecz za pośrednictwem zmiennych zdefiniowanych w oprogramowaniu. Umiejętność rozpoznawania zmiennych w testach eksploracyjnych jest zatem równie ważna jak umiejętność rozpoznawania rozstajów czy skrzyżowań w świecie rzeczywistym. Zmienne możemy podzielić na trzy typy (klasyfikacja zaczerpnięta z książki Elisabeth Hendrickson „Explore It!"):

- Oczywiste – parametry wywołania linii poleceń, wartości przekazywane w formularzu, fakt zaznaczenia poszczególnych check-boxów czy też radio buttonów;
- Subtelne – parametry przekazywane poprzez link (GET), parametry przekazywane w plikach konfiguracyjnych, pliki cookie, ustawienia językowe systemu;

TABELA 1

Jak rozpoznać rozstaje	Jakie są możliwe kierunki
Cokolwiek możemy policzyć	- Zero/Jeden/Wiele elementów - Ujemne wartości - Wielkie wartości (zarówno dodatnie jak i ujemne) - Poziom precyzji w przypadku wartości ułamkowych
Wartość numeryczna podawana przez użytkownika	
Rozmiar czegokolwiek, co dostaje się do naszego oprogramowania	
Położenie elementów względem siebie	- Przy brzegach/w środku kontenera - Obok siebie, przekrywające się
Interakcje z zewnętrznymi systemami (DB, WebService, System plików)	- Brak dostępu zasobu/przywrócenie dostępu - Brak odpowiednich praw dostępu do zasobu/nadanie praw - Zmiana zawartości (jeśli dany zasób mieści się poza zdefiniowanymi granicami bezpieczeństwa)
Położenie geograficzne	(tylko w przypadku, jeśli nasze oprogramowanie powinno obsługiwać wiele lokalizacji geograficznych) - Używanie innych formatów daty/kwoty/ etc. - Długość słów w innych językach - Obsługa innego kodowania znaków - Sortowanie w innych językach
Głębokość (w przypadku danych hierarchicznych)	- Brak zagnieżdżenia - Jeden/wiele poziomów - Bardzo wiele poziomów
Czas	- Długość - Częstotliwość
Interakcja z użytkownikiem	- Sposób nawigacji (używając klawiszy klawiatury 'TAB', 'TAB+SHIFT', używając przycisku 'Wstecz' w przeglądarce, etc.) - Definicja różnych ról użytkowników – symulacja poziomu zaawansowania danego użytkownika

- Ukryte – do tych zmiennych nie mamy bezpośrednio dostępu, np.: liczba dodanych maszyn klienckich, typ zmiennej reprezentującej dane w zależności od ich formatu.

Rodzaje rozstajów, jakie możemy napotkać, eksplorując oprogramowanie, przedstawia Tabela 1.

U kresu drogi (sił)

Ze względu na rozliczne pułapki, w które możemy wpaść w trakcie eksploracji naszego oprogramowania, dobrze, jeśli ustalimy sobie ramy czasowe – eksploracja, jeśli nieujęta w ramy czasowe, może ciągnąć się praktycznie bez końca (por. „Pokuszenie" w sekcji „W drodze – sidła i pułapki). Monitoruj na bieżąco, ile czasu zostało z początkowo przeznaczonego w porównaniu z tym, jak daleko od celu wciąż jesteś. Pomoże to zdać sobie sprawę z tego, czy przypadkiem nasz software nie wodzi nas na pokuszenie.

Zakończenie

Na zakończenie kilka ogólnych wskazówek odnośnie wpasowania testów eksploracyjnych w istniejący proces.

- Preferujmy mniejsze sesje odbywane częściej – im dłużej zwlekamy z przeprowadzeniem testów eksploracyjnych naszej aplikacji, tym bardziej skomplikowane się one stają. Zwłaszcza w początkowej fazie naszej przygody z tym rodzajem testów dobrze jest wykonywać je często, by nie brać zbyt dużych funkcjonalności na jedną sesję.
- Testowanie w parach – w świecie programistów praktyka tzw. „programowania w parach" zadomowiła się już na dobre. Świat testowy jakby wciąż się wzbrania przed uznaniem testowania w parach jako jednej z technik testowania. Jeśli chodzi o testowanie eksploracyjne, to testowanie w parach zdecydowanie zdaje egzamin – pozwala nam uniknąć błądzenia po oprogramowaniu oraz znacząco zmniejsza prawdopodobieństwo ulegnięcia pokusom zboczenia z dążenia do wytyczonego celu.

Bazując na własnym doświadczeniu, pokusiłbym się o stwierdzenie, że w projekcie o skomplikowanej domenie biznesowej każda nowa funkcjonalność powinna zostać przetestowana zarówno za pomocą testów skryptowych (preferencyjnie zautomatyzowanych), jak również za pomocą testów eksploracyjnych, zwłaszcza że żadne inne testy nie są wycelowane w testowanie utrzymywalności modelu biznesowego zastosowanego w naszej aplikacji.

AUTOR

Remigiusz Dudek

Quality Engineer, Agile'owiec, szkoleniowiec specjalizujący się w Behaviour Driven Development. Swoją wiedzą dzieli się na różnego rodzaju eventach poświęconych zagadnieniom jakości oprogramowania oraz zwinnym metodykom zarządzania.

Karierę rozpoczynał jako programista Javy, w następnej kolejności przeszedł do zagadnień jakości. Definiując metryki wdrożone w jednej z większych firm IT w Krakowie, miał okazję zaobserwować, jak potrafią one zmieniać rzeczywistość – nie zawsze w pożądaną stronę – oraz jak je właściwie interpretować. W swej karierze miał również okazję pracować jako Project Manager – mnogość perspektyw pozwala mu w chwili obecnej spojrzeć na każdy problem z wielu stron i wybierać optymalne rozwiązania.

Pracę rozpoczął w zespołach zarządzanych klasycznymi metodami, by następnie wziąć udział w transformacji ich w stronę zespołów zwinnych. Rozumie, iż metodyki zwinne nie są lepsze od klasycznych – są to po prostu inne narzędzia służące do rozwiązywania innych problemów.

Obecnie pracuje w dojrzale zwinnym Kanbanowym środowisku jako inżynier jakości. Z pasją wdraża Behaviour Driven Development starając się wyrazić zagadnienia biznesowe kodem w możliwie najbardziej optymalny i najczytelniejszy sposób – jednocześnie zachęcając ludzi biznesu do czytania dokumentacji testującej kod.

REFERENCJE

[1] „Explore It!: Reduce Risk and Increase Confidence with Exploratory Testing", autorka: Elisabeth Hendrickson

[2] „Agile Testing: A Practical Guide for Testers and Agile Teams", autorki: Lisa Crispin, Janet Gregory

[3] Kurs „Psychologii Społecznej" na uniwersytecie Wesleyan,

Bartosz Szulc

Behaviour Driven Development

Moja podróż

Wprowadzenie

Behaviour Driven Development (w skrócie BDD) od dłuższego czasu zyskuje na popularności wśród organizacji wytwarzających oprogramowanie w sposób zwinny.

Coraz więcej firm adaptuje BDD jako metodę poszerzania wiedzy dotyczącej wymagań, szczególnie przydatną wśród programistów implementujących nowe cechy produktów.

BDD staje się sformalizowanym sposobem komunikowania skomplikowanych zachowań ukrytych pod lakoniczną nazwą cechy czy też opisującej ją historyjki użytkownika.

W ciągu swojej pracy zawodowej pracowałem w zespołach, które starały się wprowadzić BDD - oczywiście z różnym skutkiem. W niniejszym artykule będę starał się przedstawić moją podróż z BDD. Nie będę skupiał się na idealnym podejściu i korzyściach z niego płynących, ile na wariacji nieudanych podejść, z czego one wynikały, czym skutkowały, jakie były zyski i straty, abyście w swojej pracy potrafili szybciej zdiagnozować miejsce, w jakim się znajdujecie, jeżeli chodzi o BDD i jakie potencjalne konsekwencje mogą z tego wypłynąć.

Czym jest BDD?

Więc czym jest BDD? Dan North mówi o BDD jako dopełnieniu TDD, więc jawi nam się to jako metoda bliska kodu. Nie jest to do końca prawdziwe spostrzeżenie. Gdy zastanowimy się nad definicją jednostki (unit) w TDD, zaczniemy pytać, czym jest owa jednostka. Czy jednostka jest jedynie

klasą, którą będziemy tworzyć w TDD – zaczynając od pisania testów – i wraz z nimi kompletować jej funkcjonalność oraz zmieniać wygląd poprzez refaktoryzację? Czy jest to może coś bardziej ogólnego Niezależna encja.

W tym znaczeniu również cecha produktu może być rozpatrywana jako jednostka.

Behaviour Driven Development, jako dopełnienie Test Driven Development, uzupełnia ją o brakujący element dokumentacji zachowania jednostki, czy to w zrozumieniu klasy kodu produkcyjnego, czy cechy produktu.

Chociaż w TDD dokumentacja istnieje, to jest to dokumentacja niejawna, tworzona za pomocą zastosowania opisowych nazw metod, zmiennych, odpowiedniej kompozycji. Taka forma dokumentacji nie sprawdzi się niestety przy definiowaniu cech tworzonego produktu, gdzie zainteresowane grupy mogą nie być w stanie przedstawić oczekiwanego wyniku w postaci kodu.

Dlatego BDD proponuje tworzenie dokumentacji w formie scenariuszy zapisanych językiem naturalnym z zastosowaniem odpowiedniego formatu, który charakteryzują trzy słowa kluczowe: *Given, When, Then.*

Proces

Przed przystąpieniem do tworzenia nowej cechy produktu w BDD wszyscy interesariusze, lub reprezentanci grup interesariuszy spotykają się, by wspólnie przedyskutować niedopowiedziane kwestie odnośnie zachowania (behaviour) implementowanej cechy.

W skład interesariuszy uczestniczących w spotkaniu wchodzą eksperci domenowi, reprezentanci biznesu oraz przedstawiciele zespołu implementującego cechę.

Zadaniem grupy jest wspólnie wypracować, zrozumiałe dla wszystkich uczestników, scenariusze opisujące zachowanie

„Coraz więcej firm adaptuje BDD jako metodę poszerzania wiedzy dotyczącej wymagań, szczególnie przydatną wśród programistów implementujących nowe cechy produktów”

nowej cechy produktu z wykorzystaniem słów kluczowych *Given, When, Then*, gdzie *Given* ustawia stan, w którym aktor, np. użytkownik, musi się znaleźć przed możliwością skorzystania z cechy produktu, When opisuje sposób użycia cechy, wykonania akcji, udostępnionej aktorowi przez cechę, natomiast *Then* pozwala przedstawić oczekiwany wynik akcji z punktu widzenia aktora i szerzej: systemu. Każdy z tych trzech podstawowych kroków scenariusza może zostać rozszerzony o dodatkowe kroki z wykorzystaniem And i Or, co pozwala rozbudować scenariusz w celu osiągniecia wymaganego poziomu szczegółowości.

Następnie tak stworzone scenariusze napędzają implementację.

Sama metoda nie narzuca języka, w którym należy scenariusze zapisywać. Dostępne są implementacje słów kluczowych dla wszystkich naturalnych języków świata, więc np. siadając do implementacji nowej cechy dla produktu tworzonego dla ZUS, prawdopodobnie lepszy wynik osiągniemy tworząc scenariusze opisujące zachowania w języku polskim, korzystając ze *Scenariusza* zamiast *Scenario*, *Zakładając* zamiast *Given, Kiedy* zamiast *When* i *Wtedy* zamiast *Then*.

Cel

Celem BDD jest dostarczenie produktu końcowego, który będzie implementował cechę spełniającą założenia biznesu spisane za pomocą ustandaryzowanych scenariuszy.

... a testowanie

Tworzone scenariusze powinny służyć nam, programistom, jako narzędzie, które rozwieje nasze wątpliwości odnośnie do zachowania cechy, dzięki czemu będziemy w stanie zaimplementować ją poprawnie, zgodnie z życzeniem, wizją biznesu. Jest to więc poniekąd forma wczesnego testowania naszych założeń dotyczących implementacji. Z racji tego, że testowanie to następuje bardzo wcześnie w procesie tworzenia, bo już na etapie analizy, korzyści z wykrycia błędu, błędnego założenia, są wysokie.

Patrząc z punktu widzenia testera, BDD, ze swoimi scenariuszami, wpisuje się w drugą ćwiartkę kwadrantu testowania w Agile zaproponowanego przez Lisę Crispin, czyli w ćwiartkę wspierającą zespół developerski i bardziej wychodzącą naprzeciw biznesowi niż technologii.

BDD wspiera zespół w dostarczeniu oprogramowania zgodnego z oczekiwaniami, wykorzystuje wiedzę biznesową zebraną w formie scenariuszy zachowania, by dostarczyć odpowiednie oprogramowanie, oczekiwane przez biznes.

Jak widać, korzyść związana z testowaniem, a wychodząca z BDD jest tylko częścią wszystkich aktywności testera w Agile. Pracując z BDD, udało mi się jednak spotykać z zespołami, gdzie testowanie odbywało się jedynie wokół scenariuszy zachowań i przykładów (examples). Cała energia skupiona była na jednej ćwiartce, co nie do końca jest najlepszym podejściem. Czynności wynikające z pozostałych ćwiartek są również ważne. Pozwalają spojrzeć na tworzony produkt z różnych

perspektyw, zauważyć różnego typu problemy, ryzyka, które mogą zmaterializować się na różnych etapach życia produktu.

BDD nie jest złotym cielcem i skupienie uwagi jedynie na scenariuszach nie zapewni wysokiej jakości produktu. Oczywiście tworzenie oprogramowania zgodnego z oczekiwaniami biznesu jest bardzo ważne i powinniśmy jako testerzy poświęcić czas, by potwierdzić, że tak jest, ale nie powinniśmy zapominać o spojrzeniu na produkt z innej perspektywy, takiej jak jakość kodu, wymagania niefunkcjonalne czy potencjalny sposób, w jaki końcowy użytkownik będzie korzystał z naszej aplikacji.

Moja podróż

Wiedząc, jaki cel przyświeca BDD, jakie są potencjalne oczekiwane korzyści, możemy przystąpić do implementacji techniki w naszym projekcie.

Więc od czego rozpocząłem swoją przygodę z BDD… ano od automatyzacji.

Automatyzacja

Patrząc teraz z perspektywy czasu, było to bardzo niemądre podejście, ale jak wiadomo, człowiek popełnia błędy i na nich się uczy. Najlepiej oczywiście na własnych. Na swoją obronę mogę dodać tylko tyle, że nie byłbym w owym czasie zdolny napisać podobnego artykułu, w którym rozpoczynam od zastanowienia się nad problemem, który staram się rozwiązać, a następnie proponuję rozwiązanie i narzędzia.

Tak się składa, że nadal, spotykając się z testerami i pytając ich o to, czy używają BDD w swoim projekcie, dostaję odpowiedź, że tak, mamy Cucumbera, tudzież mamy Behave. Co jest poniekąd równoważne z pytaniem do developera, czy stosuje TDD? I uzyskaniem odpowiedzi, że tak, piszę testy z wykorzystaniem JUnit.

Tak niestety już jest, niezwykłą popularnością cieszą się narzędzia wspierające proces… a nie sam proces czy też problem, jaki stara się rozwiązać.

Osobiście zawsze byłem zorientowany na technologię, narzędzia. Jako prawilny inżynier, absolwent szkoły technicznej, wydziału informatyki, praca z narzędziami, blisko technologii, była w pracy testera zawsze dla mnie czymś oczywistym. Uwielbiam programować. Programowanie skryptów testowych wiąże mnie mocno z zespołem, gdzie wszyscy jesteśmy programistami. Oni implementują cechy, ja implementuję testy.

Przygodę z BDD rozpocząłem więc od użycia Cucumbera jako kolejnej warstwy abstrakcji dla mojego projektu ze skryptami testowymi. Miałem już warstwę abstrakcji w postaci Page Object, która przesłaniała akcje wywoływane z przeglądarki na testowanej aplikacji. Miałem również warstwę abstrakcji dla konfiguracji, realizowanej przez Maven i serwer budowania, Jenkins. Brakowało jeszcze jednej warstwy. Warstwy abstrakcji opisu czynności wykonywanych przez test, które byłyby zrozumiałe dla osób spoza zespołu, osób, które posługują się biegle jedynie językiem naturalnym.

Padło na Cucumbera, który doskonale spisuje się jako narzędzie DSL (Domain Specific Language), transformując oczekiwania wyrażane za pomocą języka naturalnego w akcje wykonywane przez maszynę ro-

zumiejącą język techniczny, niskiego poziomu.

Na ten czas takie było moje zrozumienie BDD - jako sposób sterowania testami z wykorzystaniem języka angielskiego.

Oczywiście byłem z siebie niezmiernie zadowolony, w końcu miałem w projekcie BDD, bo przecież używałem Cucumbera. Dodatkowo rozbudowałem swoje środowisko testowe o dodatkową warstwę abstrakcji, dzięki czemu stało się jeszcze bardziej kompletne, więc miałem bazę pod kolejne prezentacje, szkolenia, warsztaty.

Rezultat

Być może czytając to, widzisz siebie realizującego "BDD" w swoim projekcie. Czy twoim zdaniem osiągnąłeś cel, któremu przyświeca Behaviour Driven Development? Ja na pewno tego celu na tamten moment nie osiągnąłem. Produkt, który tworzyliśmy, nie zaczął nagle magicznie spełniać oczekiwań biznesu tylko dlatego, że moje testy end-to-end były spisane w języku angielskim i umożliwiały interakcję z kodem. Jakość wręcz spadła.

Więc co poszło nie tak? Dlaczego jakość spadła i produkt nadal nie był zgodny z oczekiwaniami?

Po pierwsze, jakość spadła, gdyż miałem mniej czasu na to, by poświęcić się testowaniu, czynnościom opisanym przez pozostałe ćwiartki kwadrantu testera. Więcej czasu zacząłem poświęcać kodowaniu, polepszaniu swoich umiejętności programistycznych.

Dodatkowa warstwa abstrakcji skomplikowała kod testowy, czyniąc go trudniejszym w utrzymaniu, bardziej podatnym na błędy, co skutkowało wieloma błędami typu false-negative.

Po drugie, sam produkt nadal nie spełniał założeń, gdyż sposób zbierania wymagań wcale się nie zmienił. Zmienił się jedynie sposób, w jaki były one składowane, jako pliki z roszerzeniem feature zapisane z wykorzystaniem formatu BDD (Gherkin). Źródło wymagań, czyli założenia zespołu developerskiego odnośnie do implementowanej cechy, pozostało niezmienione.

Po trzecie, mając mniej czasu na interakcje z biznesem, założenia rzadziej były podważane.

I tak wpadłem w błędne koło.

Poświęcając więcej czasu na rozbudowę i utrzymywanie testów automatycznych, które dawały niewiarygodne rezultaty, miałem mniej czasu, by poddawać produkt testowaniu z różnych płaszczyzn i kontaktować się z biznesem w celu potwierdzania założeń dotyczących cech implementowanego produktu.

Rola testera i struktura testów

Dzięki temu dobitnie zrozumiałem, że moją główną rolą nie jest programowanie, tworzenie skomplikowanych systemów testujących. Nie jestem developerem – więc aby osiągnąć poziom zrozumienia niektórych zagadnień, dobrych praktyk, technik, które pozwolą mi zbudować coraz większe systemy, muszę przyswoić ogromną wiedzę, którą developerzy zbierają przez lata podczas swojej pracy. A tworząc słabej jakości testy automatyczne, powiększam dług techniczny kodu produkcyjnego.

Zrozumiałem również, że nie każdy test musi być testem end-to-end spisanym w języku naturalnym.

Testy powinny mieć odpowiednią strukturę, strukturę piramidy, na dole której znajdują się testy jednostkowe, sprawdzające poprawność implementacji jednostki kodu, klasy, metody – są to testy, których powinno być jak najwięcej, są one najbardziej atomowe, przez co najbardziej stabilne, wykonują się najszybciej, dając szybką informację zwrotną odnośnie do implementacji.

W środku piramidy powinny znaleźć się testy integracyjne, serwisów, API, testy spajające wiele jednostek pracujących wspólnie nad dostarczeniem wyniku. Powinno ich być zdecydowanie mniej niż testów jednostkowych, z racji tego, że są bardziej skomplikowane, zakładają poprawne działanie wielu komponentów, ich czas wykonania jest dłuższy, a rezultaty bardziej obarczone ryzykiem błędnej konfiguracji, środowiska czy innych czynników zewnętrznych, niezwiązanych bezpośrednio z naturą testu.

Na szczycie piramidy powinny się znaleźć testy end-to-end, testy sprawdzające aplikację z perspektywy korzystającego z niej użytkownika, który, w przypadku aplikacji internetowych, będzie z niej korzystał poprzez przeglądarkę internetową. To ten typ testów, który był najbardziej interesujący dla mnie. Testy, które cechują się największym poziomem skomplikowania spośród dotychczas wymienionych, które uzależnione są od dużej ilości komponentów, których czas trwania jest najdłuższy, a wyniki najmniej wiarygodne, obarczone najwyższym ryzykiem wprowadzenia false -negative, czyli negatywnego rezultatu,

który nie został spowodowany niepoprawnym działaniem testowanej aplikacji, ale czynnikami zewnętrznymi, jak np. opóźnienia po stronie przeglądarki, czy ryzykiem wyników false-positive, gdzie otrzymany wynik pozytywny nie oddaje rzeczywistości, gdyż test przez swoje skomplikowanie niepoprawnie sprawdza system.

Do samej standardowej piramidy testów dodałbym jeszcze wieżę na szczycie, czyli testy akceptacyjne stworzone z użyciem BDD w moim zrozumieniu. W większości przypadków byłyby to testy end-to-end opisane scenariuszami zachowania.

W moim przypadku piramida testów uległa kompletnemu odwróceniu. Zaczęła przypominać rożek, na szczycie którego jak lody w rożku lodowym znajdowały się testy end-to-end spisane w języku naturalnym, spajane z kodem znajdującym się pod spodem z wykorzystaniem Cucumbera.

Skutkowało to tym, że testy były niestabilne, rezultaty traciły na wiarygodności, a czas wykonania testów był zbyt zauważalny.

Wątpliwości

Wtedy w mojej poirytowanej głowie zaczęły kotłować się pytania:

- Czy każdy scenariusz testowy to automatycznie scenariusz akceptacyjny, który muszę opisać Given, When, Then i zautomatyzować, patrząc na aplikację z perspektywy użytkownika?
- Jaki powinien być poziom skomplikowania tworzonych scenariuszy, bo same narzędzia dają ogrom możliwości - możemy tworzyć wiele kroków i dodawać przykłady, wiążąc kroki w kodzie. Czy z punktu widzenia inżyniera narzędzia dają nam moc stworzenia tak skomplikowanych struktur, że można by było opisać cechę wręcz jednym scenariuszem?
- Czy jako tester powinienem dbać jedynie o scenariusze zachowania? Czy one, jako jedyne artefakty wykorzystywane w mojej pracy, pomogą mi zapewnić oprogramowanie spełniające oczekiwania biznesu, użytkownika klienta, wysoką jakość?
- Czy BDD to framework testowy?
- Czy BDD to jedynie automatyzacja?

Korzyści

Oczywiście nic w świecie nie jest biało-czarne. Również w przypadku zastosowania BDD jedynie jako warstwy abstrakcji opisu przypadków testowych udało się osiągnąć korzyści, takie jak:

- żywa dokumentacja - funkcjonalność aplikacji opisana w języku naturalnym, zrozumiałym dla każdego, nie tylko developera, ale również człowieka np. z pierwszej linii wsparcia, która zawsze przechowuje rzeczywistą informację o stanie aplikacji, z racji tego, że spisane zachowania powiązane są z skryptami automatycznymi regularnie sprawdzającymi poprawność implementacji;
- narzędzie do automatyzacji testów dla osób nietechnicznych - wykorzystując dostępne w scenariuszach kroki, osoby nietechniczne mogą tworzyć nowe przypadki testowe poprzez kopiowanie lub mogą rozbudowywać scenariusze testowe, dodając nowe przykłady do już dostępnych scenariuszy, przez co zwiększać również ich pokrycie;
- czytelniejsze wyniki testów - zamiast

przesyłać plik xml z niewiele mówiącymi nazwami metod testowych, możemy wysłać plik html z czytelnymi nazwami cech i opisującymi je scenariuszami, wraz z informacją o aktualnym statusie implementacji tych zachowań w systemie.

Nie ma wśród tych korzyści jednak głównego powodu wprowadzenia BDD w projekcie, czyli dostarczenia produktu zgodnego z oczekiwaniami. Należało więc kontynuować podróż.

Komunikacja z biznesem

Nauczony porażką, postanowiłem zmienić swoje podejście do BDD.

Jeżeli głównym problemem jest nadal problem dostarczania cech niezgodnych z oczekiwaniami, użyjmy BDD do tego, by te wymagania zebrać - wyciągnąć scenariusze zachowań od biznesu.

Tak więc rozpocząłem regularne spotkania, na których ja, reprezentant zespołu developerskiego, oraz przedstawiciele biznesu, spotykaliśmy się, by omówić oczekiwane cechy produktu i spisać je jako scenariusze i zachowania z wykorzystaniem odpowiedniego formatowania narzuconego przez BDD.

I tak zaczęły powstawać scenariusze, przykłady. Zacząłem coraz bardziej rozumieć domenę i oczekiwania względem tworzonego produktu. Pracując z przedstawicielami biznesu, ucząc ich BDD, poprawnego formatowania scenariuszy, poprawnego formułowania oczekiwań, widziałem siebie jako most łączący świat biznesu ze światem technologicznym.

Z biegiem czasu spotkania przechodziły coraz sprawniej, odpowiednie scenariusze powstawały szybciej, pojawiało się mniej kwestii spornych co do sposobu tworzenia scenariuszy, tak by były na tyle zrozumiałe, aby umożliwiały zaimplementowanie poprawnie cechy, ale nie aż tak skomplikowane, by proces tworzenia nużył biznes. Przynajmniej tak mi się wydawało. Jednak rezultaty wcale nie były takie, jakich wszyscy oczekiwali.

Rezultat

Developerzy, którzy po spotkaniu otrzymywali paczkę ze scenariuszami, wcale nagle nie zaczęli implementować produktu zgodnie z oczekiwaniami. Nadal implementacja odbiegała od tego, co biznes uważał, że znajdowało się w scenariuszu.

Co więc poszło nie tak? Przecież scenariusze były tworzone przez ludzi najbardziej kompetentnych do ich tworzenia.

Okazuje się, że o ile scenariusze były zrozumiałe przeze mnie, nie były one zrozumiałe przez resztę zespołu oraz że poziom szczegółowości, który ja uznawałem za wystarczający, nie był wystarczający dla reszty zespołu.

W scenariuszach pojawiała się niezrozumiała terminologia, pojęcia domenowe, skróty, które były zrozumiałe dla mnie, gdyż uczestniczyłem w spotkaniach i w locie wyciągałem informacje od biznesu odnośnie do niezrozumiałych stwierdzeń. Nie były one jednak zrozumiałe przez resztę zespołu, która w tych spotkaniach nie uczestniczyła. Chociaż starałem się przybliżyć zespołowi terminologię domeny biznesowej, wyjaśnić niejasności, rozmawiając i spisując odpowiedzi jako komentarze

do scenariuszy czy dodatkowe opisy do cechy, to nie byłem w stanie cofnąć się w czasie i rozwiać wątpliwości developerów wynikających z niedostatecznego poziomu szczegółowości stworzonych scenariuszy.

Dla mnie wszystko było zrozumiałe, jednak dla developerów nie.

Cała sytuacja zaczęła mocno przypominać mini-waterfall czy też iteracyjny waterfall, gdzie raz na jakiś czas ktoś tworzy specyfikację, która później wręczana jest zespołowi developerskiemu, od którego oczekuje się dostarczenia produktu zgodnie z dokumentacją, której zespół nie potrafi do końca zrozumieć i do której ma masę pytań.

Kolejne podejście do BDD zaczęło skutkować kolejnymi problemami, nowymi problemami.

Dopasowanie przekazu pod odbiorcę i efekt głuchego telefonu

Wiadomo, że każdy człowiek rozumie problem, czy ogólnie przekaz, inaczej, na różnym poziomie. Może mieć różnego typu wątpliwości i pytania do postawionej tezy, by ją dostatecznie zrozumieć. Założenie, że mój udział przy definiowaniu scenariuszy będzie wystarczający, by rozwiązać wątpliwości wszystkich członków zespołu, było bardzo optymistyczne. To, co dla mnie stawało się czymś zrozumiałym, oczywistym, nadal nie było zrozumiałe dla zespołu. Z biegiem czasu skutkowało to tym, że scenariusze były coraz bardziej ogólne. Mój poziom zrozumienia wzrastał, więc nie widziałem w tym nic złego, ja widziałem w tym optymalizację. Zrozumienie po stronie developerów jednak nie wzrastało i "oczywiste oczywistości" generowały masę pytań. Pytań, na które starałem się odpowiadać, generując przy tym efekt głuchego telefonu, wprowadzając niedokładne interpretacje odpowiedzi zasłyszanych od przedstawicieli biznesu.

Korzyści

Kolejne podejście do BDD nie przyniosło oczekiwanego rezultatu, ale również, podobnie jak w poprzednim przypadku, pojawiły się ciekawe, poboczne, korzyści, takie jak:

- głębsza analiza potrzeby - łatwo jest wyrazić oczekiwanie, trudniej jest je uszczegółowić, zawęzić, wymaga to dodatkowego zastanowienia się, próby odpowiedzenia sobie na trudne pytania, na które w przypadku implementowanej cechy i jej zachowań do tej pory starali się odpowiadać developerzy ją implementujący. Teraz ciężar spadł na biznes, co spowodowało, że niektóre cechy straciły na znaczeniu, nie były tak bardzo potrzebne, inne stały się wieloma cechami, niektóre okazały się całkowicie niepotrzebne, zbędne, gdyż były realizowane przez inne cechy lub sposób, w jaki użytkownik końcowy miałby z nich korzystać, nie miał sensu, był zbyt skomplikowany;
- odpowiednie testy akceptacyjne - struktura piramidy testów nabrała kształtów, scenariusze akceptacyjne były tworzone we współpracy z reprezentantami biznesu i tylko one trafiały na szczyt piramidy, były automatyzowane z wykorzystaniem Cucumbera, reszta testów znajdowała odzwierciedlenie w testach pozostałych warstw piramidy;
- zrozumienie zagadnień - wraz z kolejnymi spotkaniami zanikała potrzeba tłumaczenia poszczególnych elementów

tworzących scenariusz w BDD, łatwiej udawało się znaleźć złoty środek, jeżeli chodzi o poziom skomplikowania scenariusza, poziom jego opisowości;biznes nauczył się Gita - bo przecież nadal musimy tworzyć pliki feature w edytorze i trzymać je lokalnie… (swoją drogą, dlaczego Cucumber nigdy nie dostarczał edytora bliskiego nietechnicznemu użytkownikowi? Przynajmniej teraz pojawił się Cucumber.pro).

Poziom szczegółowości scenariuszy

Oprócz powyższych korzyści po części udało mi się odpowiedzieć na dręczące mnie pytania. Wiedziałem już, co stanowi test akceptacyjny i na jakim poziomie skomplikowania tworzyć scenariusze.
Kwestia tworzenia scenariusza o odpowiednim poziomie skomplikowania, poziomie opisowości jest chyba najbardziej dyskutowanym tematem wśród ludzi pracujących z BDD. Nie wiem, czy istnieje jeden przepis. Wszystko zależy od kontekstu. Scenariusze mogą mieć inną strukturę przy systemach bankowych czy serwisach internetowych udostępniających funkcjonalność do skonsumowania zależnym aplikacjom lub przy bibliotekach open-source dostarczających ciekawe API. Moim zdaniem poziom szczegółowości powinien wynikać z obserwacji współpracy reprezentantów różnych grup nad tworzonym scenariuszem.

„Scenariusze testowe mogą mieć inną strukturę przy systemach bankowych czy serwisach internetowych udostępniających funkcjonalność do skonsumowania zależnym aplikacjom lub przy bibliotekach open-source dostarczających ciekawe API. Moim zdaniem poziom szczegółowości powinien wynikać z obserwacji współpracy reprezentantów różnych grup nad tworzonym scenariuszem.”

Czy wszystkie stwierdzenia są jasne? Czy ktoś zasypia, bo zaczynamy schodzić zbyt nisko w poziom szczegółowości? Czy zaczynamy zaszywać informacje związane z technicznym rozwiązaniem problemu, implementacją, a nie skupiamy się na opisie zachowania? To są prawdopodobnie pytania, na które doświadczony moderator musi ciągle odpowiadać podczas obserwacji uczestników sesji BDD, aby pomóc wypracować grupie odpowiedni poziom szczegółowości, dopasowany do ich potrzeb.

Zirytowany moimi nieudanymi podejściami do BDD, postanowiłem dać metodzie ostatnią szansę, ale zanim ją opiszę, muszę przedstawić wariację poprzedniego podejścia, moim zdaniem mogącą mieć większe negatywne przełożenie na pracę zespołu i sposób dostarczenia produktu. Jest to mianowicie BDD używane jako metoda dostarczania cech zgodnie z oczekiwaniami jedynie przez zespół developerski, z wykluczeniem biznesu.

Zamknięcie się na biznes

Musimy uświadomić sobie, że stosując BDD jedynie wewnątrz zespołu developerskiego, nie rozwiążemy problemu niedostarczania produktu zgodnego z oczekiwaniem klienta, biznesu. Wprowadzimy jedynie standard zarządzania naszymi założeniami co do tych oczekiwań. Standard, który może się okazać trudny we wdrożeniu, wymagający od wszystkich członków zespołu zaangażowania, nauki nowych technik, również trochę gry aktorskiej, gdyż musimy starać się wcielać w rolę klienta i mówić jego niezrozumiałym, skomplikowanym językiem domenowym.

Rezultat

Chociaż potencjalne problemy będą bardzo przypominały problemy zawarte w pierwszej części, gdy to tester zaczyna automatyzować testy z wykorzystaniem technik i narzędzi BDD, zyskają one na sile, gdyż dotyczyć będą całego zespołu, a nie tylko jednostki.

Osoby, które miały wątpliwą przyjemność rozwiązywać takie konflikty, wiedzą, że łatwiej jest uświadomić o błędzie jednostkę niż grupę. W przypadku BDD stosowanym wyłącznie przez zespół developerski reprezentant biznesu może spotkać się ze skumulowanym niezadowoleniem wynikającym z niezaakceptowania implementacji produktu. Przecież tyle pracy zostało włożone przez zespól developerski w to, by wszystko się udało, a tu taka wiadomość od człowieka, który nie musiał ponosić żadnych kosztów wprowadzenia BDD.

Chociaż niezadowolenie to prawdopodobnie nie zmaterializuje się w postaci szeroko wyrażanego oburzenia, co nawet byłoby lepsze dla produktu, bo pozwoliłoby zauważyć i przez to rozwiązać problem szybciej, przyjmie ono formę utajnionego konfliktu, podziału zespołu na my i oni. My, developerzy mocno pracujący, starający się zastosować najnowsze techniki i metody w naszym procesie wytwarzania, oni, reprezentanci biznesu, wiecznie niezadowoleni z dostarczonego produktu, niezainteresowani usprawnieniami, torpedujący nasze działania.

Bardzo ważna odpowiedzialność spoczywa na testerze, który powinien starać się jedną i drugą stronę zaangażować do wspólnej pracy nad produktem. Uniknąć konfliktu ***my - oni***.

Otwarcie i współpraca

Na tym też skupiło się moje trzecie ostatnie podejście do BDD. Nad zaangażowaniem wszystkich zainteresowanych grup, biznesu, developerów, testerów w pracę nad scenariuszami opisującymi zachowania poszczególnych cech produktu.

Bo czym tak naprawdę jest BDD? Behaviour Driven Develoment. Co kryje się pod tymi słowami?

Przez *Behaviour* rozumiemy tworzone scenariusze.

Kto pomoże stworzyć nam odpowiednie scenariusze? Oczywiście biznes, osoby wychodzące do nas, developerów z potrzebą, dla których sprawa jest oczywista. My za pomocą scenariuszy spisanych w odpowiedni sposób musimy tę oczywistość wyciągnąć i pojąć.

Do czego nam są potrzebne scenariusze?

Tutaj z pomocą przychodzi drugi człon nazwy. Potrzebne są do tego, by ciągnąć development. Aby developerzy skupili się na tym, by dostarczyć dokładnie taką funkcjonalność, jaka została zawarta w scenariuszach, jakiej oczekuje od nich klient.

Aby więc wszystko się udało, potrzebujemy developerów i przedstawicieli biznesu pracujących razem. Ale gdzie w tym procesie jest tester?

Tester jest spoiwem wiążącym dwa różne światy. Mostem pomiędzy światem biznesu i technologii, gdzie jego umiejętność translacji języka domenowego na język techniczny staje się niezastąpiona, by pociągnąć projekt do przodu. Również po zakończeniu sesji BDD, gdy developerzy pracują nad cechą, by w przypadku widocznych problemów stanąć na wysokości zadania i rozbudować bazę scenariuszy we współpracy z wszystkimi zainteresowanymi stronami.

Dlaczego w nazwie nie ma nic o testowaniu? Bo BDD nie jest metodą testowania. Co najwyżej jest to metoda dostarczająca artefakty, które można następnie wykorzystać do testowania czy przetestować, sprawdzić poprawność, spójność, jasność scenariuszy bądź poprawność implementacji, jej zgodność z uzgodnionymi scenariuszami.

Rezultat

W końcu, gdy zrozumiemy, czym jest BDD, prawdopodobnie dostarczymy produkt, którego oczekuje od nas klient. Dlaczego prawdopodobnie? Bo tworzenie oprogramowania to skomplikowany proces, w którym nigdy nie ma pewności, że coś się nie uda, że metody czy standar-

dy, które sprawdzają się dla jakiegoś projektu, sprawdzą się też w naszym. Więc prawdopodobnie reprezentuje ten poziom niepewności, której musimy być świadomi, adaptując BDD czy inne potencjalne usprawnienie.

Dodatkowe korzyści

Moim zdaniem BDD dla testera przynosi wiele dodatkowych korzyści - w końcu przenosimy część naszej odpowiedzialności na zespół.

Od teraz wszyscy jesteśmy odpowiedzialni za drugą ćwiartkę kwadrantu testera - definiowanie scenariuszy akceptacyjnych, przykładów. Od teraz tester ma więcej czasu, by poświęcić się pozostałym obszarom.

Prawdopodobnie developerzy pomogą też z implementacją kodu automatyzującego scenariusze. Dlaczego? Bo nauczeni TDD i zyskiem płynącym z analizy założeń dotyczących kompozycji przed implementacją, mogą stwierdzić, że wyjście na wyższym poziom przy analizie może dać podobne rezultaty i pozwolić uniknąć wielu problemów związanych ze złą kompozycją kodu produkcyjnego.

Dodatkowo przez udział w automatyzacji prawdopodobnie napiszą kod, który potwierdzi zgodność ze scenariuszem, używając niższych warstw dostępu do logiki aplikacji, a nie poprzez przeglądarkę, przez co rezultaty będą bardziej wiarygodne i redukcji ulegnie czas potrzebny na otrzymanie wyniku.

Zyskamy też wszystkie pośrednie benefity płynące z BDD, o których wcześniej wspomniałem, jak np. żywa dokumentacja.

Jest to też doskonała okazja, uniknąć potencjalnych konfliktów.

Więcej czasu dla testera to więcej czasu na przeanalizowanie aplikacji okiem użytkownika, od strony wymagań niefunkcjonalnych, analizy rynku i zmian w popularności urządzeń, systemów operacyjnych, z których nasz przyszły użytkownik będzie korzystał pracując z naszą aplikacją. To w końcu więcej czasu dla testera, by skupić się na jakości kodu, skomplikowaniem kodu, procesami związanymi z ciągłym budowaniem, integracją, dostarczaniem.

Na tym etapie możemy stwierdzić, że testy akceptacyjne to nie jedyne testy, którymi powinniśmy być zainteresowani jako testerzy.

Wiemy, że BDD nie powinno być ślepo używane do każdego typu problemów, nie każde testy powinny mieć odpowiadający scenariusz spięty przez Cucumbera z kodem.

Powinniśmy szukać optymalnych rozwiązań, mieć na uwadze stabilność testów i szybkość ich wykonania. Przecież automatyzacja nie wychodzi od BDD, ale jest to reakcja na skrócenie cyklu dostarczania oprogramowania, przez co musimy szukać usprawnień w czasie, jakiego potrzebujemy na uzyskanie potwierdzania, że nasza aplikacja działa poprawnie po zastosowanej zmianie.

Podróże uczą

Jak dzisiaj nauczony doświadczeniem podszedłbym do Behaviour Driven Development?
Zacząłbym od przedyskutowania idei. Od

AUTOR

Bartosz Szulc

Właśnie rozpocząłem 6 rok swojej przygody z testowaniem. Przez ostatnie 5 lat nosiłem różne kapelusze: testera, senior testera, test managera, scrum mastera, aktualnie architekta testów.

Odkąd pamiętam, zawsze byłem zainteresowany automatyzacją, zarówno testów, jak i innych procesów związanych z wytwarzaniem programowania. Próbowaniem nowych bibliotek, narzędzi, języków, łączeniem poszczególnych elementów w większe całości. Pozbywaniem się czynnika ludzkiego, manualnej pracy. Wraz z czasem coraz bardziej interesujące stawały się dla mnie biznesowe aspekty testowania, analiza ryzyk, kosztów, rynków. Dzisiaj mogę powiedzieć, że są one dla mnie tak ważne, jak do tej pory była dla mnie automatyzacja. Prawdopodobnie, jak wielu inżynierów informatyki, zacząłem od zrozumienia jak, by z czasem zrozumieć, kiedy i dlaczego. Mając świadomość wagi tych pytań, w końcu mogę zacząć sprawnie eksperymentować… to znaczy testować.

tego, jaki problem staramy się rozwiązać, dlaczego i w jaki sposób. Starałbym się znaleźć, wypracować zrozumienie co do problemu i metody jego rozwiązania wśród wszystkich zainteresowanych grup, developerów, testerów, reprezentantów biznesu.

Następnie przeprowadziłbym przykładową iterację, gdzie nacisk byłby wywarty na sesję zbierania scenariuszy. Na otwarcie wszystkich zainteresowanych do dyskusji odnośnie do tworzonych scenariuszy, wypracowanie optimum odnośnie do poziomu skomplikowania, poziomu szczegółowości scenariuszy, zdefiniowanie słownika domenowego, dogłębne zrozumienie struktury cechy, scenariusza, trzech słów kluczowych: Given, When, Then.

Zadowolony z rezultatów spotkań, widząc, że scenariusze i kierowany nim development, przynoszą poprawę - oprogramowanie zaczyna spełniać oczekiwania, zacząłbym szukać optymalizacji, narzędzi wspierających proces. Rozpocząłbym pracę z developerami nad automatyzacją.

Podsumowując, wywróciłbym moją przygodę z BDD do góry nogami. Rozpocząłbym od problemu, a nie od narzędzi.

Mam nadzieję, że Wy czytając ten artykuł, rozumiejąc jakie problemy czyhają przy wybiórczym czy niepełnym wprowadzeniu tej metody tworzenia oprogramowania, będziecie w stanie ich uniknąć w swoich projektach. Bo tak naprawdę BDD warto wprowadzać, ale jak ze wszystkim, najpierw trzeba zrozumieć dlaczego.

Michał Kruszewski

Testy akceptacyjne w administracji publicznej – pro publico bono?

Wprowadzenie

Dlaczego testy akceptacyjne „wystawiane są na zewnątrz"? Jakie są największe wyzwania podczas ich realizacji? Czy wynika to ze świadomych braków kompetencji u zamawiającego, który oczekuje pomocy? Czy może jest to potrzeba przeniesienia odpowiedzialności na podmiot zewnętrzny (wykonawcę testów) za jakość systemu?

W ciągu ostatniego 1,5 roku znacząco wzrosło zainteresowanie podmiotów publicznych wsparciem podczas realizacji testów oprogramowania, w tym testów akceptacyjnych. Przyczyn tego zjawiska jest kilka. Jedną z głównych jest zakończenie ostatniego okresu programowania (Wieloletni okres planowania budżetu Unii Europejskiej. Okres obowiązywania dokumentów programowych) z lat 2007 – 2013 (co oznacza zamykanie lub kończenie się projektów w latach 2014 – 2015) i konieczność zagwarantowania odpowiednio wysokiej jakości oprogramowania przy odbiorze szerokiej gamy systemów ITC o wartości od kilkuset tysięcy do kilkudziesięciu milionów złotych. Co więcej, w nadchodzącej perspektywie finansowej 2014 – 2020 należy się spodziewać jeszcze szerszego rozwoju e-usług, a co za tym idzie, dalszego wzrostu zapotrzebowania na wsparcie ze strony ekspertów, m.in. w dziedzinie testowania.

Najważniejsze wyzwania

„Nasze systemy są niezawodne" – czyli co zrobić, gdy nie ma błędów

Jednym z największych wyzwań, z jakim zderzamy się podczas testów akceptacyjnych w instytucjach publicznych, są kryte-

Rysunek 1. Budowa systemu z perspektywy zamawiającego

ria odbioru systemu. Oczywiste stwierdzenie, że nie możemy mówić o całkowitym braku błędów (defektów), staje często w sprzeczności z ogólnymi zapisami SIWZ (Specyfikacja Istotnych Warunków Zamówienia), w której jednym ze stosowanych standardów jest zapis „odbiór systemu bez zastrzeżeń". Jak więc będąc między młotem a kowadłem - zamawiającym a wykonawcą - doprowadzić do szczęśliwego końca? Jak wybrnąć ze słowno-znaczeniowych pułapek prawnych? Doświadczenie pokazuje, że można. W jaki sposób? Niektóre z elementów drogi do sukcesu zostaną przedstawione w dalszej części artykułu.

Wymagania niefunkcjonalne – testowanie tego, czego nie ma?

Drugą największą bolączką testowanych systemów są wymagania niefunkcjonalne. Właściwie często należałoby mówić o ich braku lub nietestowalności. Rozpoczynając od zapisów mówiących o zgodności systemu z prawem, polegających na wylistowaniu kilkudziesięciu aktów prawnych, a kończąc na wymaganiach wydajnościowych typu „system umożliwi wprowadzenie 1000 dokumentów rocznie" (autentyczne zapisy z SIWZ). Jak to przetestować? Jak naprawdę sprawdzić wydajność systemu?

Brak wiedzy zamawiającego, czyli... za wszystko odpowiada wykonawca

Choć receptą na brak wiedzy mamy być My – profesjonaliści – to problem braku wiedzy u zamawiającego pozostaje. Dokumenty, na podstawie których realizujemy pracę, już istnieją – zostały przygotowane wcześniej, a to właśnie na ich podstawie mamy realizować dalsze prace. Cytowane często stwierdzenie „Wykonawca przyjdzie, zbuduje system, zaprojektuje testy,

a potem przetestuje… a my… my wtedy zaczniemy na nim pracować" nie jest niedorzeczną historią, ale często realnym oczekiwaniem wyrażanym przez zamawiającego. Jak zmienić to przekonanie? Jak wpłynąć na postawę zamawiającego?

Jak wygląda budowa systemu z punktu widzenia zamawiającego i kim jesteśmy „My"?

My, czyli kto?

Zamawiający jest często świadomy braku swoich kompetencji i potrzeby pomocy. Z tego przeświadczenia wyrasta chęć pozyskania specjalisty-sojusznika, który pomoże osiągnąć cel projektu. Nasza rola wybiega jednak dalej, w praktyce często jesteśmy narzędziem równowagi i rozwiązywania sporów w projektach. Dlaczego?

Są dwa powody:

- Wykonawca systemu przestaje „w naszej obecności" wprowadzać „nowe powszechnie obowiązujące standardy", które nie są standardami, a które inaczej zostałyby wprowadzone do projektu, a w razie protestów tłumaczone byłyby niewiedzą zamawiającego… „my budujemy systemy, my się znamy a wy nie, więc róbcie jak mówimy",
- Zamawiający potrafi zrezygnować z nierealnych lub błędnych wymagań, bo mówimy to My, jego sojusznik, a nam ufa.

Perspektywa czasowa – kiedy pojawia się wykonawca, a kiedy My

Zrozumienie przedstawionej problematyki jest ściśle powiązane ze sposobem prac

nad systemami w instytucjach publicznych. W dużych ośrodkach projekty budowy systemu trwają często kilka lat. Z tej perspektywy myśl o odbiorze systemu, jego wdrożeniu, a więc i testowaniu odsuwana jest na później.

Dochodzi do stworzenia koncepcji systemu, napisany zostaje SIWZ, wybrany wykonawca systemu, a potem – czasami dopiero przed samymi testami – pojawiamy się My.

Niestety, jak wspomniano już wcześniej, opis realizacji prac i stawianych zadań już istnieje. Jaką ścieżką wybrać, którędy zmierzać do celu? Co zrobić lub zmienić?

Droga do sukcesu

Zrozumieć zamawiającego, czyli co to jest „PZP" (Prawo Zamówień Publicznych) i „Ustawa o odpowiedzialności za naruszenie dyscypliny finansów publicznych"

„Nawet ludzie, którzy się nie lubią, pomagają sobie w kłopotach, gdy płyną na tej samej łodzi"

Sun Tzu „Sztuka Wojny"

Prowadzenie projektów (nie tylko testowych) w administracji publicznej wymaga zrozumienia kluczowych aktów prawnych. Ich znajomość pozwala ominąć wirtualne ściany, których nie rozumiemy i nie potrafimy przejść. Dlaczego?

Dyscyplina finansów publicznych nakłada odpowiedzialność na urzędników za ich działania. W naszym przypadku odpowiedzialność ta związana jest z odbiorem systemu. Fakt odbioru systemu jest potwierdzeniem, że pieniądze publiczne zostały spożytkowane prawidłowo, że nie ma zastrzeżeń, że wszystko jest tak jak należy! Pod protokołem trzeba złożyć podpis i za odebrany produkt (system) przyjąć tym samym odpowiedzialność. Tu pojawiamy się My – i jest to jeden z podstawowych czynników sukcesu. Jesteśmy ekspertami – profesjonalistami, którzy majestatem wiedzy i doświadczenia pozwolą na spokojny sen zamawiającego. To My mamy zapewnić, że odbierany system został przetestowany, nie ma błędów (teoria oczywiście w tym miejscu mija się z oczekiwaniami), a poniekąd mamy przejąć część odpowiedzialności.

„W ciągu ostatniego 1,5 roku znacząco wzrosło zainteresowanie podmiotów publicznych wsparciem podczas realizacji testów oprogramowania, w tym testów akceptacyjnych."

Tabela 1. Definicja „błędu krytycznego”

Definicja z Umowy	Definicja zmodyfikowana w Planie Testów
Błąd skutkujący stałą niedostępnością systemu lub powodujący: • Brak możliwości kontynuowania w systemie procesu biznesowego zgodnie z tym, jak zostało to zaprojektowane. • Brak spełnienia przez system obowiązkowych wymagań zgodnie z tym, jak ich wykonanie zostało zaprojektowane. • Brak spełnienia przez system preferowanych wymagań zgodnie z tym, jak ich wykonanie zostało zaprojektowane lub brak możliwości zastosowania tymczasowego rozwiązania, pochłaniającego co najwyżej 3 razy tyle czasu, co rozwiązanie standardowe. Tymczasowe rozwiązanie nie może powodować niepoprawnych operacji systemowych.	Błąd skutkujący stałą niedostępnością systemu lub powodujący: • Brak możliwości kontynuowania w systemie procesu biznesowego zgodnie z tym, jak zostało to zaprojektowane. • Brak spełnienia przez system obowiązkowych lub preferowanych wymagań zgodnie z tym, jak ich wykonanie zostało zaprojektowane.
Komentarz Zmodyfikowana definicja nie jest dobra, była jednak jedyną akceptowalną dla obu stron (zamawiający/wykonawca). Pozwoliła również wyeliminować najważniejsze problemy i jej wewnętrzne sprzeczności. Dzięki tak elastycznemu podejściu projekt mógł być kontynuowany.	

Nowe–stare definiowanie błędów i wymagań

„You must unlearn, what you have learned”
Yoda „The Empire Strikes Back”

Jak wspomniałem wcześniej, pojawiamy się z reguły po wytworzeniu najważniejszych dokumentów uzgodnieniowych (m.in. umowy pomiędzy zamawiającym a wykonawcą, ale często po uzgodnieniu planu testów), których zmiana ze względu na podpisane umowy i zobowiązania często nie jest możliwa. Jednym z istotnych elementów związanych z karami finansowymi, odbiorami itd. są błędy i ich klasyfikacja.

Każdy SIWZ posiada własną klasyfikację błędów. Są one różne: od zero-jedynkowych, przez dwu- do pięciostopniowych. Choć podjęto próby pewnej standaryzacji podejścia i stan ten ulega stopniowej poprawie (istnieje np. Rozporządzenie Ministra Nauki i Informatyzacji z dnia 19 października 2005 r. w sprawie testów akceptacyjnych oraz badania oprogramowa-

nia interfejsowego i weryfikacji tego badania, rozporządzenie to nie wprowadza wzorca klasyfikacji błędów), to istniejące wzorce często nie są znane, nie nadążają za rzeczywistością i odbiegają od stosowanych standardów.

Praktyka często wymusza na nas zbudowanie nowej definicji czy klasyfikacji błędów, które będą zgodne z wymogami formalnymi podpisanych umów, pozwolą na sprawną naprawę defektów i co najważniejsze, doprowadzą do ostatecznego odbioru systemu – sukcesu. Jak do tego dojść? Musimy wziąć pod uwagę następujące aspekty:

- Wymogi formalne umowy,
- Faktyczne kryteria odbioru systemu,
- Definicję „zmiany" w SIWZ,
- Zakres wsparcia świadczonego przez wykonawcę na etapie eksploatacji systemu.

Niestety, biorąc uwagę powyższe, poprawność metodyczna jest ostatnim elementem, jaki bierzemy pod uwagę.

A jak to wygląda w praktyce? W tabelach 1 i 2 przedstawiono dwa przykłady rozwiązań zrealizowanych w kilkumilionowych projektach związanych z „zastanymi" definicjami błędów.

Krystalizacja wymagań niefunkcjonalnych

Tak jak klasyfikacja błędów niesie ze sobą konsekwencje formalno-prawne, tak wymagania niefunkcjonalne związane są z reguły z zadowoleniem przyszłych użytkowników systemu.

Takich wymagań często nie ma w dokumentacji formalnej (SIWZ), a przecież nie można przetestować tego, co nie jest

Tabela 2. Definicja „awarii"

Definicja z SIWZ	**Definicja zmodyfikowana w Planie Testów**
Błąd oprogramowania lub konfiguracji uniemożliwiający prawidłowe działanie lub uniemożliwiający korzystanie z dowolnej funkcjonalności Systemu. **Uwaga!** W dalszej części umowy kary zdefiniowane były w stosunku do pojęcia awarii.	Błąd Krytyczny: wada w oprogramowaniu, która uniemożliwia użytkownikowi korzystanie z systemu zgodnie z jego przeznaczeniem lub uniemożliwia mu korzystanie z krytycznej funkcjonalności. Jeżeli możliwe jest funkcjonalne obejście problemu, wada oprogramowania nie będzie uważana za krytyczną.
Komentarz Wprowadzona modyfikacja pojęć pozwoliła „obejść" zapisy SIWZ i stworzyć dość poprawną klasyfikację błędów. Plan testów określił kryteria odbioru systemu w kontekście nowej klasyfikacji, co pozwala na odbiór i wdrożenie systemu z błędami np. drobnymi.	

określone, zapisane. Często też istniejące wymagania są nietestowalne. Wyjścia z tej sytuacji są dwa: możemy pominąć milczeniem testy niefunkcjonalne (i często tak się robi, odkładając np. kwestię problemów wydajnościowych na etap utrzymania/serwisu oprogramowania) lub na nowo je zdefiniować.

Droga do celu jest „prosta", ale niełatwa, należy wydobyć istniejące oczekiwania i zapisać je w formie wymagań. Oczywiście jest to praca analityczna, ale w praktyce zwykle realizowana przez testerów lub kierowników testów i zapisywana jedynie w formie przypadków testowych. Mamy oczywiście świadomość, że nie potrafimy powiązać scenariuszy z wymaganiami, że działamy niemetodycznie… może i tak, ale działamy na pewno praktycznie. Równie częstym posunięciem jest „uszczegóławianie" przypadkami testowymi wymagań niefunkcjonalnych.

Trudność realizacji powyższych prac polega głównie na potrzebie wzięcia pod uwagę uwarunkowań prawnych realizowanego projektu i kryteriów z SIWZ. Sukces ich zdefiniowania lub przedefiniowania istniejących wymagań związany jest z:

- ukierunkowaniem wymagań na odbiór systemu,
- uwzględnieniem formalnych zapisów umowy,
- nieznacznymi zmianami w zakresie sposobu realizacji testów.

Tabela 3. „Uszczegóławianie" wymagań niefunkcjonalnych

Wymaganie z Umowy	**Zrealizowany scenariusz testowy**
System musi być w stanie obsługiwać przynajmniej 1000 użytkowników jednocześnie.	• I scenariusz testowy: Przeprowadzono symulację logowania jednoczesnego 1000 użytkowników systemu z pełnym pomiarem parametrów wydajnościowych. • II scenariusz testowy: Przeprowadzono symulację jednoczesnej pracy 1000 użytkowników realizujących kilkanaście procedur przewidzianych skryptami (z różnych obszarów biznesowych systemu) z pełnym pomiarem parametrów wydajnościowych.
Komentarz Wykonawca zgodził się na realizację scenariuszy testowych, ale bez określenia parametrów wydajnościowych (czasów odpowiedzi). Osiągnięty kompromis pozwolił na przetestowanie systemu w warunkach zbliżonych do obciążenia produkcyjnego, pozwalając zamawiającemu na realną ocenę wydajności systemu, jednocześnie pozwalając wykonawcy pozostać bezpiecznym w przypadku niemożności osiągnięcia porozumienia w zakresie satysfakcji z otrzymanych rezultatów.	
Wydajność Systemu musi wynosić minimum 98,5%, w stosunku pomiarowym miesięcznym, w trybie dostępności 11h/5dni.	Brak
Komentarz Dla powyższego wymagania nie udało się wypracować w porozumieniu z wykonawcą systemu miernika (wprowadzenie miernika mogłoby pociągnąć za sobą skutki finansowe w postaci kar w przyszłości). Brak mierzalnych wymagań wydajnościowych w Umowie spowodował, że zapis pozostał martwy.	

Michał Kruszewski

AUTOR

Od 2010 roku współpracuje z największymi jednostkami Administracji Publicznej na szczeblu Rządowym i Samorządowym w Polsce, jako niezależny ekspert w zakresie inżynierii oprogramowania, audytu i zarządzania projektami (wiedzę i doświadczenie potwierdzają takie certyfikaty, jak m. in.: PRINCE2, TOGAF, ISTQB, CISA).

Jest współautorem strategii i koncepcji w zakresie budowy ładu korporacyjnego i architektury systemów w dużych jednostkach Administracji Publicznej.

W zakresie testów brał udział jako Kierownik Testów w wielomilionowych wdrożeniach systemów m.in. w takich instytucjach, jak UZP, KRUS, CPI.

W tabeli 3 przedstawiono kilka przykładów wymagań niefunkcjonalnych i wypracowaną później formę ich realizacji.

Nauka, nauka, nauka

Ostatnim, ale równie ważnym elementem ostatecznego sukcesu jest przekazywanie wiedzy, dzielenie się nią z zamawiającym. Istniejące przeświadczenie, że „jeżeli nauczymy, to przestaniemy być potrzebni", nie znajduje odzwierciedlania w rzeczywistości. Przekazując wiedzę, nie tylko stajemy się wiarygodni w oczach zamawiającego, ale również budujemy zaufanie i postrzeganie nas jak faktycznych - a nie tylko „papierowych" - ekspertów. Dodatkowo nasze działanie, które stają się dzięki temu zrozumiałe, uzyskują łatwiejszą akceptację, nawet jeżeli wcześniej pozornie wydawać się mogły sprzeczne z interesem zamawiającego.

Pro publico bono?

Podjęta polemika z celowością i problematyką udziału i angażowania się w testowanie w szeroko rozumianej administracji publicznej miała na celu przybliżyć tym z nas, dla których to środowisko jest nowe, najważniejsze problemy i uwarunkowania.

Udział Nas, ekspertów, w testach w administracji publicznej jest generowaniem powszechnego dobra publicznego. Dzięki niemu mamy lepiej przygotowane i działające systemy w coraz szerzej rozwijającej się e-Administracji. Niosąc kaganek oświaty, wprowadzamy najlepsze praktyki i standardy w te miejsca, gdzie trafiają nasze wspólne podatki.

test:fest

Wrocław

21:lutego:2015

Festiwal Jakości

Budynek D20
Politechniki Wrocławskiej
ul. Janiszewskiego 8

t:f

http://testfest.wroclaw.pl

facebook.com/testfestwroclaw

Marta Firlej

Pokaz możliwości narzędzia Microsoft Test Manager

Wprowadzenie

Na rynku jest wiele narzędzi służących do zarządzania testami. Jeszcze więcej jest tych wspomagających zarządzanie projektem. Do tej pory korzystałam z wielu narzędzi: od Excela przez JIRA i HP Quality Center. Żadne z nich nie ułatwiało mi pracy tak bardzo jak Microsoft Test Manager. Narzędzie to wspiera nie tylko w wykonywaniu i zarządzaniu testami skryptowymi, ale także znacząco ułatwia wykonywanie i przechowywanie wyników testów eksploracyjnych. Wszystkie testalia mogą być łatwo połączone z wymaganiami, przez co znacząco ułatwione jest traceability. Microsoft Test Manager to przede wszystkim bardzo intuicyjne i łatwe w użyciu narzędzie, które pozwala nawet początkującym testerom na bardzo wiele aktywności testerskich, takich jak podstawy automatyzacji, wymuszanie i ułatwianie tworzenia test case'ów czy defektów.

Na prezentacji Testwarez 2014 chciałam przedstawić możliwości Microsoft Test Manager ze wskazaniem jego mocnych i słabych stron. Uczestnicy dowiedzieli się od czego zacząć, jak skonfigurować pierwszy test plan, jak dodać, zarządzać i wykonywać testy, jak dodawać i zarządzać defektami oraz konfiguracją. Zaprezentowane zostały także dodatkowe funkcje, takie jak wspieranie wykonywania, nagrywania i odtwarzania testów oraz zbierania ich wyników. Jak każde narzędzie, także to ma swoje ograniczenia.

Microsoft Test Manager to samodzielna aplikacja, która nie jest częścią Visual Studio, jednak wymaga połączenia z Team Foundation Server (TFS), który jest dla niego bazą danych, a także narzędziem służącym do zarządzania projektem, wymaganiami i kodem.

Praca z Testing Center

Przed rozpoczęciem pracy należy stworzyć oraz skonfigurować plan testów (Test Plan). Następnie można skorzystać z jednej z dwóch części Testing Center lub Lab Center.

Testing Center został zaprojektowany po to, żeby wspierać główne obszary i aktywności testerskie, takie jak:

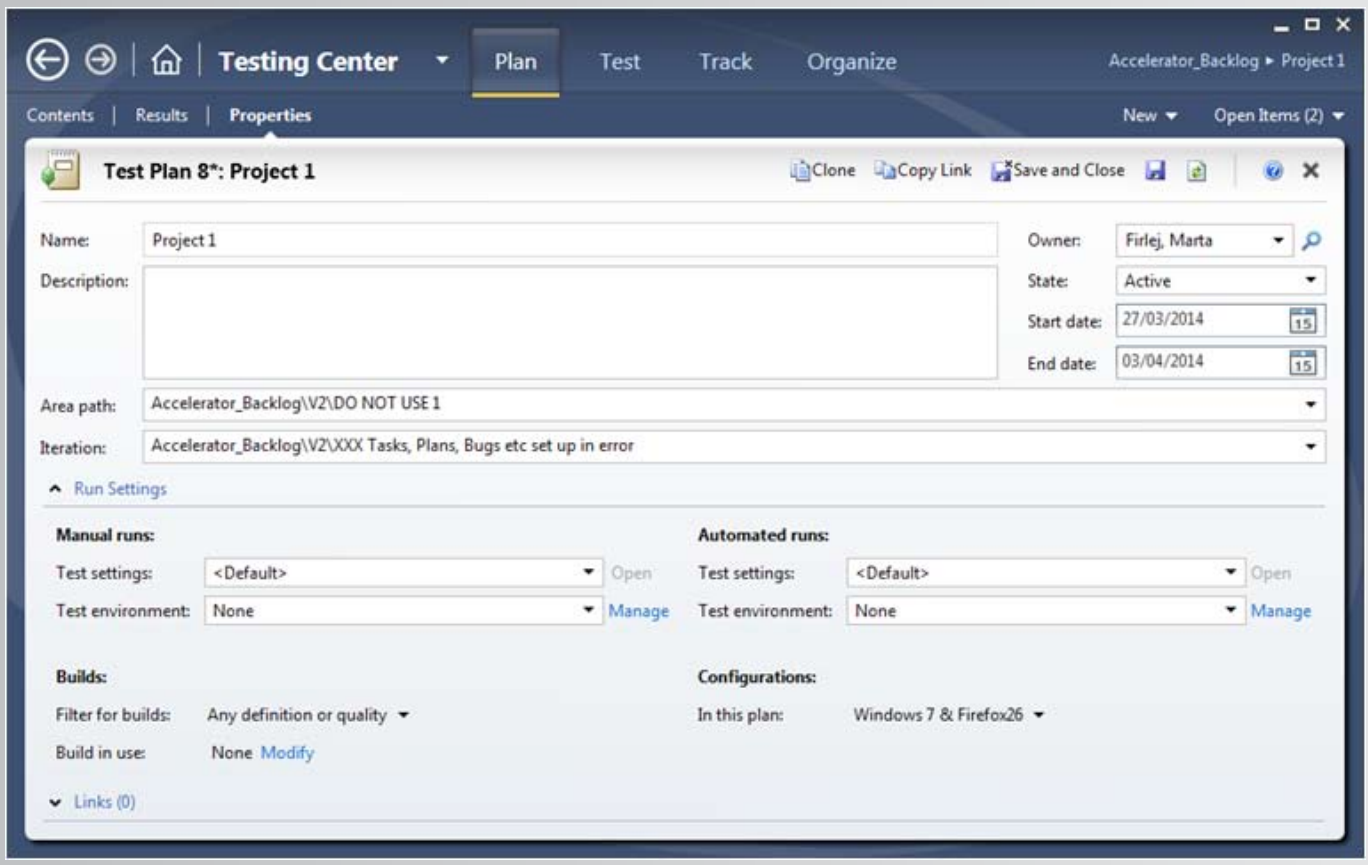
- Planowanie (Plan)
- Testowanie (Test)
- Śledzenie (Track)
- Organizowanie (Organize)

Testing Center to część Microsoft Test Manager, z której korzystam na co dzień w mojej pracy. Tak jak już wspomniałam - przed rozpoczęciem pracy wymagane jest utworzenie planu testów - **Test Plan** (Ekran 1).

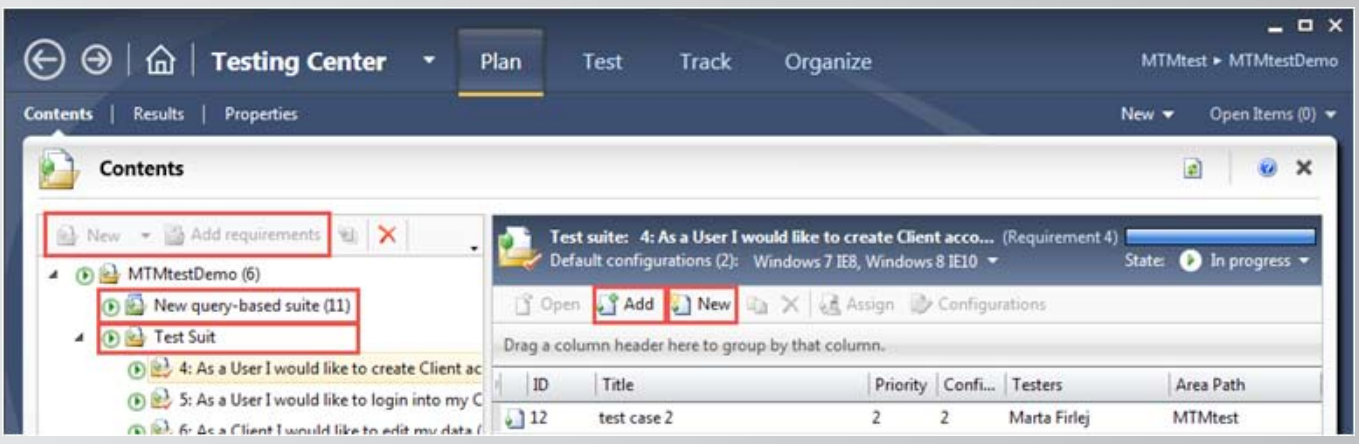
Podczas jego edycji można opisać wykonywane testy, określić miejsce, w którym poprzez TFS zapisywane będą przypadki testowe i wyniki testów, określić konfigurację oraz środowiska dla testów manualnych i automatycznych, a także zbudować definicję testowanej wersji zbudowanego oprogramowania.

W zakładce **Plan** dla używanego planu te-

Ekran 1. Konfiguracja Test Planu



Ekran 2. Tworzenie Test Suites



stów możemy utworzyć zestawy testowe (**Test Suites**), które służą do organizowania przypadków testowych (Ekran 2). Możemy traktować je jak foldery, których są dwa typy: standardowy oraz opierający się na zapytaniu do bazy danych. Do zestawów testowych możemy także przyporządkowywać wymagania z projektowego *backlogu*.

Do tak zorganizowanych folderów możemy „wkładać" nasze przypadki testowe.

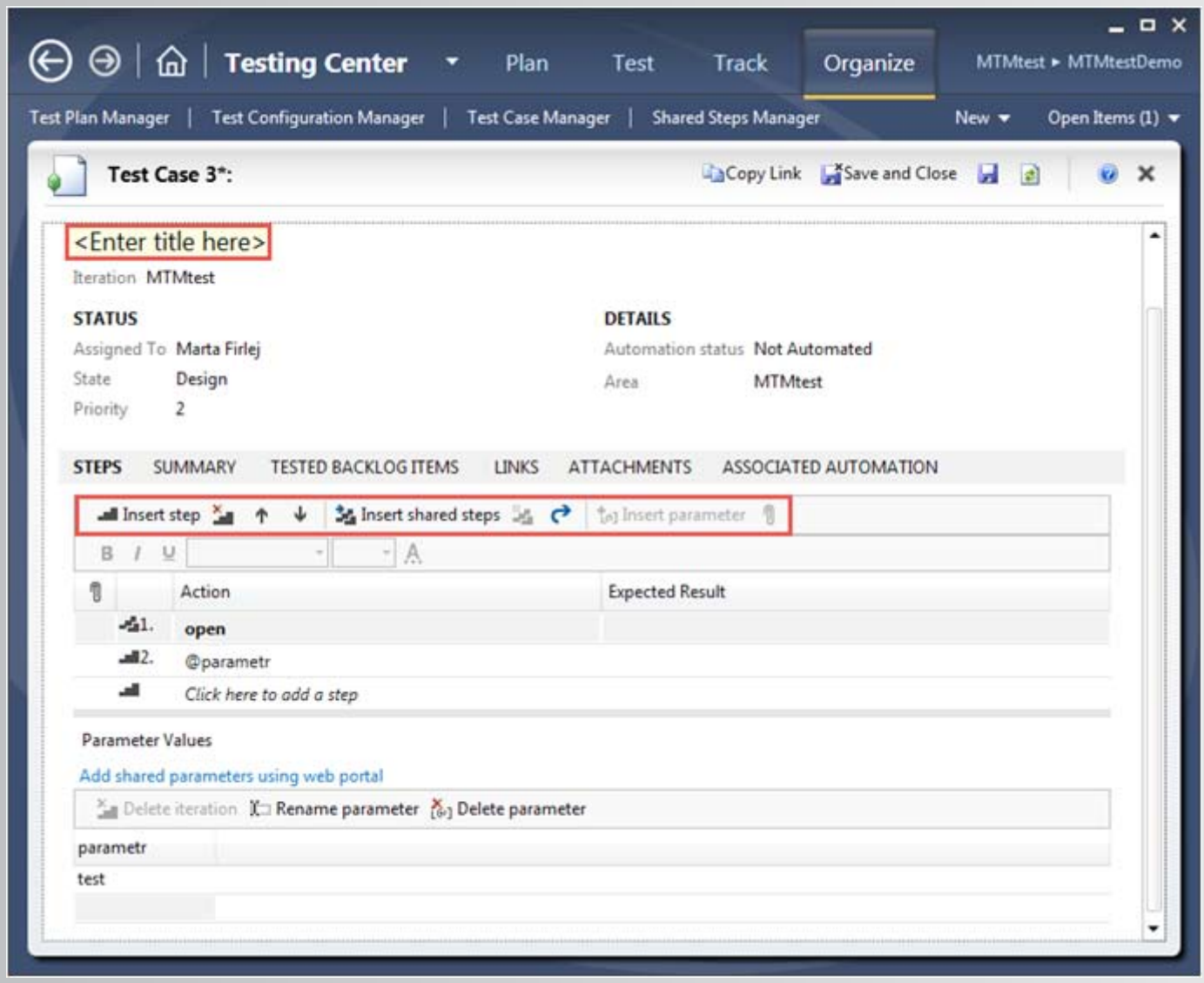
Przypadek testowy (**Test case**) wygląda podobnie jak wszystkie taski przechowywane za pomocą Team Foundation Server.

Ma on sekcje (Ekran 3):

- tytuł;
- status - gdzie określona jest osoba odpowiedzialna, priorytet, status i informacje o planowanej bądź wykonanej automatyzacji;
- klasyfikacja - gdzie znajdują się ścieżki do iteracji oraz obszaru testów;
- kroki testowe - gdzie wpisujemy scenariusz testowy wraz z oczekiwanymi rezultatami;
- podsumowanie - jedyne miejsce, gdzie możemy dodawać opis słowny bądź też uwagi do naszego przypadku testowego;
- testowane user story - linki do konkretnych user stories, których ten przypadek testowy dotyczy;
- linki - wszelkie wewnętrzne i zewnętrzne linki;
- załączniki;
- przypisana automatyzacja - w tej sekcji możliwe jest podpięcie stworzonej przy pomocy Coded UI automatyzacji.

Podczas tworzenia scenariusza testowego możliwe jest użycie parametrów. Są one określane przez dodanie znaku @, np. *@parametr*. Użycie parametru umożliwia wykonanie tego samego przypadku testowego wielokrotnie dla wielu różnych wartości, np. testowanie logowania dla 3 różnych użytkowników. Możliwe jest także wcześniejsze stworzenie współdzielonych kroków testowych (Shared steps), które mogą być wykorzystane w tej samej formie w wielu przypadkach testowych, a ich zmiana spowoduje zmianę we wszystkich miejscach ich użycia jednocześnie.

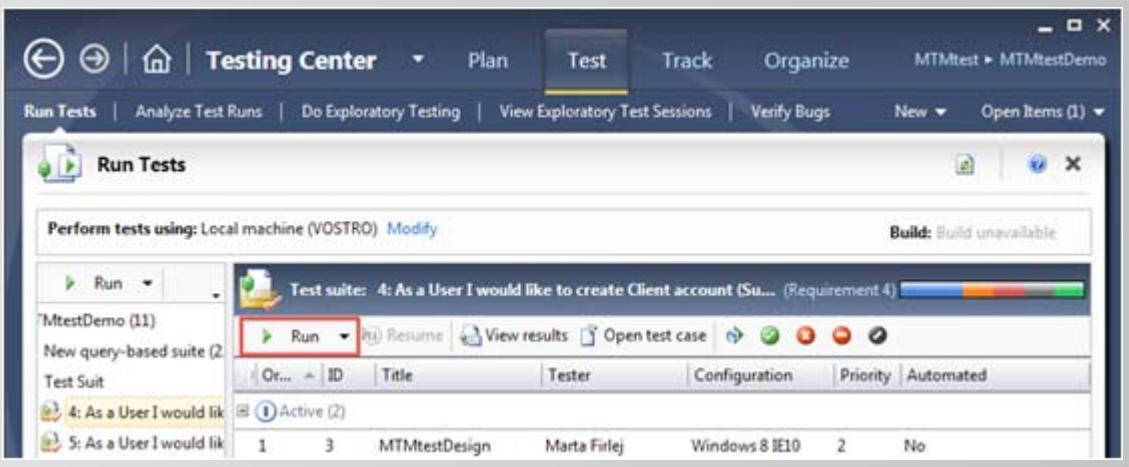
Ekran 3. Tworzenie Test Case



Poszczególne przypadki testowe mogą mieć przypisaną różną, stworzoną wcześniej konfigurację testową, taką jak system operacyjny i przeglądarka, czy też osoby odpowiedzialne za testy. Zmieniając zakładkę główną na **Test,** przechodzimy do sekcji stworzonej z myślą o wykonywaniu testów.

Wybierając zakładkę **Run Tests,** widzimy listę wszystkich aktywnych przypadków testowych (Ekran 4). Przypadki testowe z przypisaną więcej niż jedna konfiguracją będą wyświetlane wielokrotnie. Wybierając poszczególne testy i używając funkcji *Run,* możemy nagrywać wykonywanie testu manualnego, jednocześnie wspierając się stworzonym wcześniej scenariuszem testowym. Stworzone podczas testowania nagrania możemy wykorzystać do automatycznego odtwarzania testów (Fast-forward). Z tego miejsca możliwe jest także uruchamianie testów automatycznych przypisanych do danego przypadku testowego.

Ekran 4. Zakładka Run



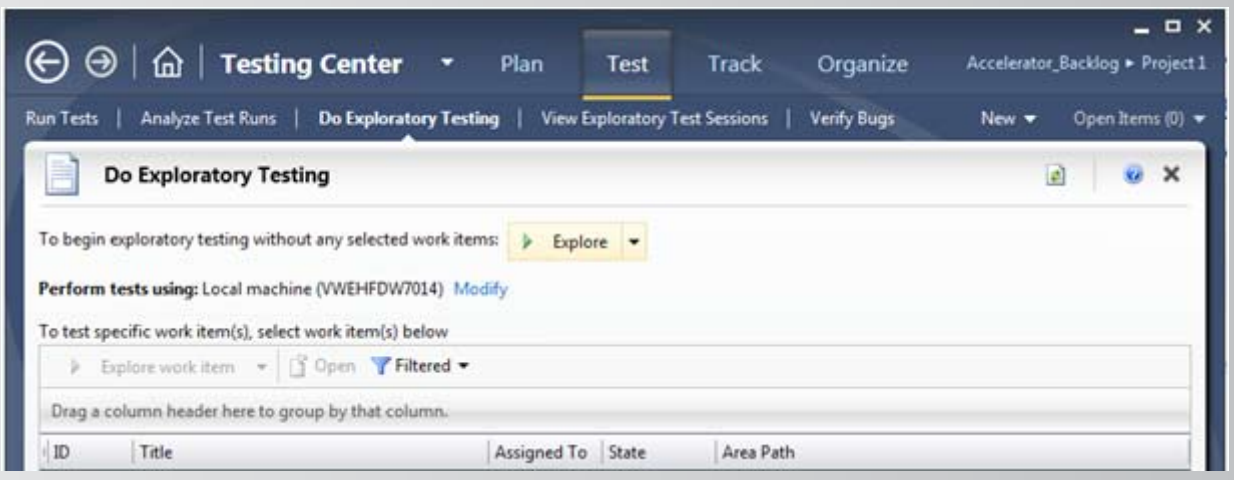
Zmieniając zakładkę na **Do Exploratory Testing,** przechodzimy do sekcji, z której możemy rozpocząć wykonywanie i nagrywanie testów eksploracyjnych, przypisywanie ich do user story czy wybranie środowiska albo wersję oprogramowania, na którym wykonane zostaną testy (Ekran 5).

Tutaj w zakładce **View Exploratory - Test Sessions** możemy wrócić do zapisów z wykonanych wcześniej sesji testów eksploracyjnych.

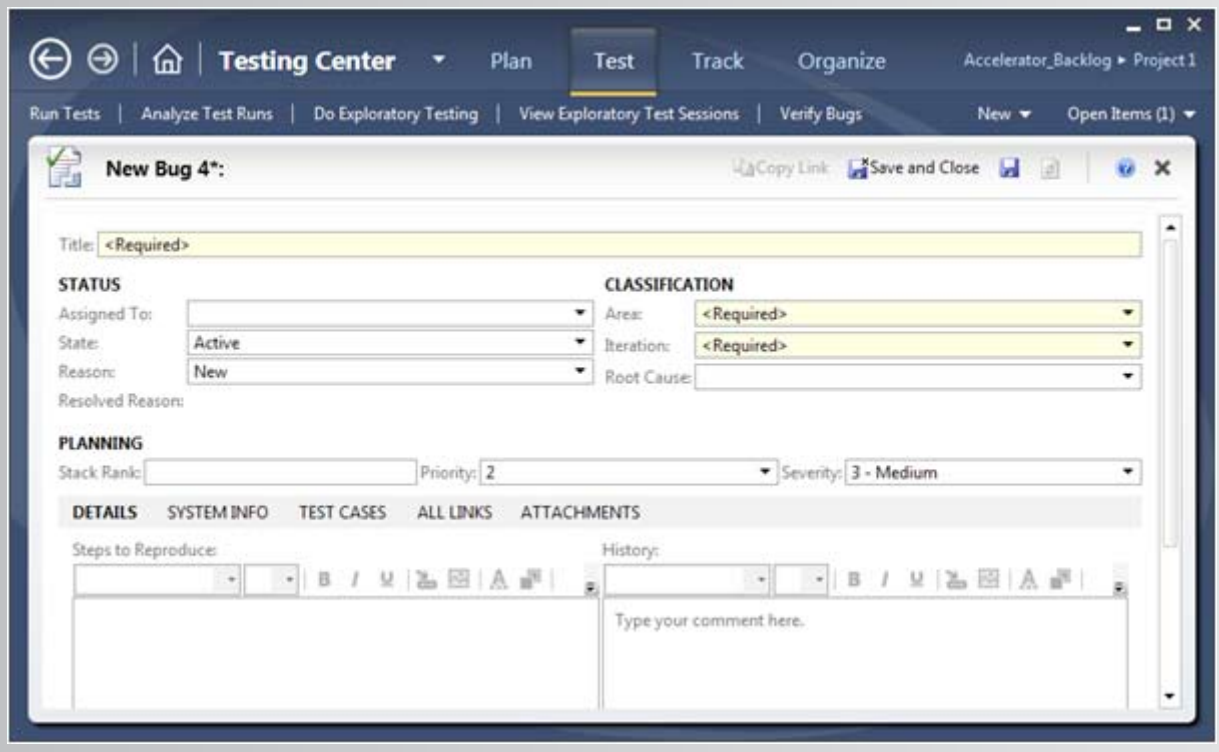
Poprzez zakładkę **Verify Bugs** możliwe jest tworzenie oraz zarządzanie defektami stworzonymi wcześniej podczas wykonywania testów manualnych, automatycznych czy eksploracyjnych (Ekran 6).

W zakładce głównej **Track** możliwa jest analiza wpływu wykonanych testów na poszczególne wersje oprogramowania oraz wyłonienie rekomendowanych testów regresji czy defektów do retestów (Ekran 7).

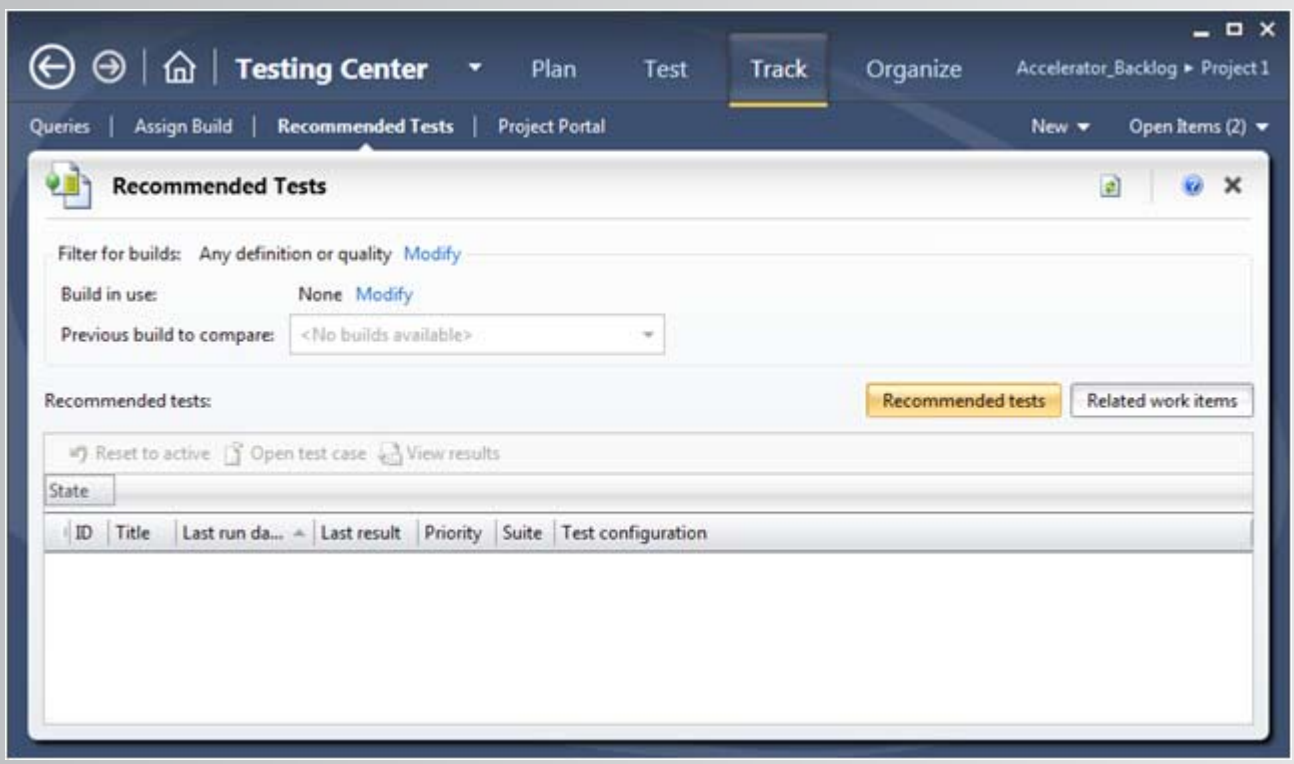
Ekran 5. Zakładka do Exploratory Testing



Ekran 6. Zakładka New Bug



Ekran 7. Zakładka Track



Testy te wyłaniane są na podstawie wprowadzanych przez programistów zmian w metodach, które zostały wykorzystane do testów w poprzednich wersjach oprogramowania. Testy te rekomendowane są za pomocą porównywania wersji oprogramowania używanego obecnie do wybranych wersji poprzednich.

W zakładce głównej **Organize** możemy zarządzać przypadkami testowymi, współdzielonymi krokami testowymi, konfiguracją planu testów oraz samym narzędziem.

Podsumowując, używając narzędzia Microsoft Test Manager, Testing center, możliwe jest:

- tworzenie i zarządzanie planami testów (Test Plan);
- tworzenie i zarządzanie zestawami testowymi (Test Suites);
- tworzenie, zarządzanie i przypisywanie współdzielonych kroków testowych (Shared steps) do przypadków testowych;
- tworzenie i zarządzanie przypadkami testowymi dla danego planu testów;
- tworzenie, zarządzanie i przypisywania konfiguracji testowych do danego przypadku testowego, takich jak system operacyjny, przeglądarki;
- przypisywanie testerów do poszczególnych przypadków testowych;
- przyporządkowywanie każdego elementu Test Planu do wymagań z Team Foundation Server;
- przypisywanie używanej wersji oprogramowania (Bulid) do wykonywanych przypadków testowych;
- wykonywanie testów manualnych;
- wykonywanie testów eksploracyjnych;
- tworzenie i zarządzanie defektami;
- analizę wpływu wykonanych testów i wyłonienia rekomendowanych testów regresji;
- nagrywanie i odtwarzanie testów (Fast-forward);
- uruchamianie testów automatycznych przypisanych do danego przypadku testowego.

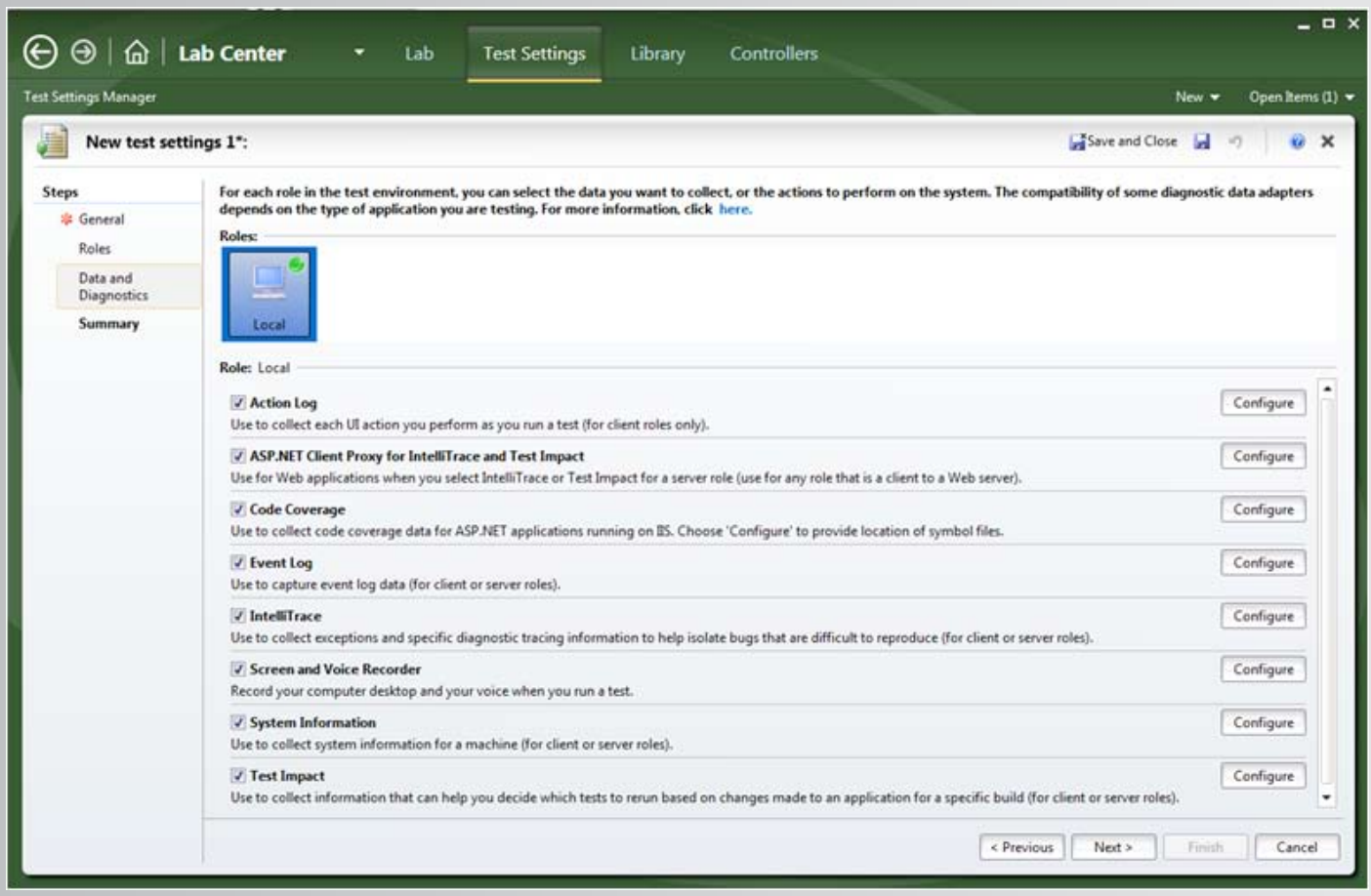
Największym wyzwaniem podczas wykonywania testów jest zbieranie danych diagnostycznych potrzebnych do identyfikacji problemów. Opisywane narzędzie umożliwia zbieranie wielu danych diagnostycznych, takich jak:

- zrzuty ekranu (screen shots)
- nagrania wideo (video recordings)
- informacje systemowe (system information)
- wpływ testu (test impact)
- dziennik akcji (action log)
- dziennik zdarzeń (event log)
- pokrycie kodu testami (code coverage)
- ASP.NET profiler
- IntelliTrace
- network emulation

Lab Center

Druga część Microsoft Test Manager, **Lab Center**, jest używana do tworzenia oraz ustawiania środowisk potrzebnych do wykonywania testów (Ekran 8).

Ekran 8. Lab Center



Uwagi krytyczne

Jak każde narzędzie Microsoft Test Manager nie jest pozbawiony wad, a są to m.in.:

- CENA! - oficjalnie na stronie Microsoft cena pakietu Visual Studio Test Professional 2013 z usługą MSDN zaczyna się od 5143PLN za odnowienie do 12409PLN za nową licencję, jednak często jest on dodawany w pakiecie przy kupowaniu licencji na Visual Studio,
- Poprzez Team Foundation Server możemy wybrać tylko 3 rodzaje metodyk zarządzania projektem i testami: Scrum, Agile i CMMI,
- Wykorzystanie większości funkcji możliwe jest tylko wtedy, gdy korzystamy z poprawnie skonfigurowanego projektu w Team Foundation Server,
- Narzędzie jest pomocne tylko wtedy, kiedy cały projekt zarządzany jest za pomocą Team Foundation Server,
- Ciężka jest samodzielna konfiguracja, wymaga zmian w definicji Team Foundation Server,
- Wykorzystanie większości funkcji wymagania doświadczenia i wiedzy dotyczącej używanego narzędzia,
- Jeżeli wiele osób pracuje jednocześnie przy jednym wymaganiu, należy często odświeżać ekran, niestety, samoczynnie nie zawsze dzieje się to dostatecznie szybko,
- Pamięciożerność - wraz z ilością zbie-

AUTORKA

Marta Firlej

Quality Engineer i Team Lead w zespole agile'owym. Zaangażowana w aspekty związane z zapewnianiem jakości od 2006 roku.

W trakcie pracy skupiona na produkcie i procesie, zarówno testowym, jak i projektowym. Zawodowo zaangażowana w tworzenie dobrej atmosfery w zespole oraz zapewnienie jakości, ze szczególnym uwzględnieniem testów eksploracyjnych, użyteczności i narzędzi wspomagających całokształt procesu testowego. Wykazuje się proaktywną postawą zarówno w firmach, w których pracuje, jak i poza nimi.

Pomagała przy organizacji oraz była członkiem komisji sędziowskiej podczas TestingCup 2013 i 2014. Posiadaczka certyfikatu ISTQB Advanced Level Test Manager.

ranych danych diagnostycznych oraz długością trwania testu Microsoft Test Manager zajmuje coraz więcej pamięci, przez co czas wykonywania testów zostaje znacząco wydłużony.

Podsumowanie

Microsoft Test Manager to potężne narzędzie stworzone specjalnie dla testerów w celu ułatwienia i zwiększenia transparentności ich codziennej pracy. Wspiera ono nie tylko organizowanie pracy testerów i zarządzanie testami poprzez ich planowanie, ale także wykonywanie testów manualnych i automatycznych. Narzędzie to umożliwia śledzenie wyników testów i znalezionych defektów oraz zarządzanie środowiskami testowymi.

W powyższym artykule przedstawione zostało krótkie podsumowanie najważniejszych funkcji desktopowej wersji aplikacji. W celu bliższego poznania Microsoft Test Managera warto pobrać bezpłatną 90-dniową wersję Visual Studio Test Professional oraz skorzystać z bazy wiedzy dostępnej pod adresem: http://msdn.microsoft.com/pl-pl/library/jj635157.aspx.

WarszawQA

Spotkania z jakością
i testowaniem oprogramowania
w Warszawie:

cykliczne,
bezpłatne,
ciekawe,
zabawne,
wciągające.

My i Wy.
Wspólnie. Dla Nas.

www.warszawqa.pl

Michał Sierzputowski

Zautomatyzuj swoje testy automatyczne oparte na Selenium

Wprowadzenie

Testy w projekcie są bardzo ważną częścią tworzenia oprogramowania, a testy automatyczne pomagają w wykrywaniu błędów na wczesnych etapach procesu wytwarzania. Zwiększają one również pewność, że w czasie zmian czy refaktoringu naszej aplikacji niczego nie popsuliśmy. Jednak należy pamiętać, że automatyzacja to proces ciągły i wymaga ciągłej uwagi i udoskonaleń – tak samo jak wytwarzanie nowych funkcjonalności w aplikacji. Jeśli nie podejdziemy odpowiednio do tego problemu, to możemy skończyć z bardzo dużym zestawem testów trudnych w zarządzaniu i utrzymaniu.

Pracując nad automatyzacją testów w kilku projektach miałem okazję poznać dobre praktyki, któ,re pomagają w utrzymaniu kodu testowego, zwiększają jego jakość czy przyspieszają pracę nad samymi przypadkami testowymi. Przykłady, które postaram się tu omówić, będą oparte na moim doświadczeniu z aplikacjami webowymi i frameworkiem Selenium.

Podstawy dobrych testów automatycznych

Zacznijmy od tego, że testy automatyczne to, szczególnie w dużych projektach, tak naprawdę oddzielny projekt informatyczny i należy podejść do nich jak do każdego innego projektu. Należy wziąć pod uwagę to, że kod testów będzie zmieniał się wraz ze zmianami testowanej aplikacji. Dlatego powinniśmy przede wszystkim zacząć od ustalenia podobnych zasad, jakie stosuje się przy rozwijaniu aplikacji.

Musimy ustalić standardy kodowania, jakich będziemy się trzymać. W kilkuosobowych zespołach tak prozaiczna, wydawać by się mogło sprawa, jak konwencje nazw zmiennych czy metod, w znaczący sposób zmniejsza późniejsze nieporozumienia czy ogranicza duplikacje.

Popularne określenie „czystego kodu" (ang. clean code) musi dotyczyć również kodowania testów. Jest to zagadnienie dość obszerne. Każda osoba pisząca testy automatyczne powinna sobie przyswoić jego zakres i stosować się do jego reguł. Możemy być pozytywnie zaskoczeni, jak nawet kilka prostych reguł zaczerpniętych z zasad „czystego kodu" może znacząco zwiększyć jego jakość. Wdrażanie odpowiednich wzorców projektowych pozwala na wykorzystanie sprawdzonych sposobów na rozwiązanie typowych problemów, skupienie się bardziej na logice, jaką chcemy zaimplementować, jak również oszczędza czas poprzez redukcję duplikacji kodu.

„Testy automatyczne to, szczególnie w dużych projektach, tak naprawdę oddzielny projekt informatyczny i należy podejść do nich jak do każdego innego projektu"

Kolejnymi sprawami, dość oczywistymi w czasie tworzenia aplikacji, są inspekcje kodu (ang. code review) i jego wersjonowanie za pomocą odpowiednich narzędzi. Kod testów powinien być również nadzorowany przez członków zespołu, co oprócz kwestii kontroli daje bezsprzeczną zaletę transferu wiedzy i dzielenia się dobrymi praktykami.

Selenium

Kolejne zagadnienia chciałbym przedstawić, opierając się już na konkretnym narzędziu. Jest nim Selenium – czyli darmowy framework, który pozwala na zautomatyzowanie interakcji użytkownika z aplikacją webową. Swoją popularność zyskał m.in. dzięki otwartemu kodowi źródłowemu (ang. open source) i wsparciu większości nowoczesnych języków programowania, najpopularniejszych przeglądarek i środowisk. Wiele innych produktów wspiera go czy też integruje się z nim.

Selenium dostarcza wersję uproszczoną swojego rozwiązania w postaci Selenium IDE, dzięki któremu możemy nagrać nasze

akcje wykonywane na stronie i potem odtworzyć je automatycznie wraz z dołączonymi asercjami już jako konkretne testy. Rozwiązanie wydaje się idealne dla osób zaczynających swoją przygodę z automatyzacją. W prosty sposób można stworzyć dość obszerną bazę testów. Należy jednak zauważyć, że w wypadku zmian testowanej aplikacji trzeba oczywiście wdrożyć poprawki w przygotowanych wcześniej testach. Okazuje się, że w wypadku „nagranego testu" szybciej będzie nagrać nowy, niż naprawiać ten, który przestał działać. Nie jest to większym problemem, gdy mamy zestaw kilku testów, jednak gdy baza testów jest już większa, koszty utrzymanie znacząco wzrastają, zwłaszcza że tego typu testy nie współdzielą nawet części swojego kodu. Jest to więc rozwiązanie niepolecane w praktyce dla większych projektów.

Drugą opcją, na której też dalej będę bazował w swoich przykładach, jest pisanie skryptów testowych w wybranym języku programowania i wywoływanie tzw. WebDrivera odpowiedzialnego za komunikację z wybraną przeglądarką.

Oddzielenie logiki biznesowej od części technicznej

Przykład jednej z dość podstawowych kwestii przy pisaniu testów automatycznych chciałbym oprzeć na prostym skrypcie napisanym w języku Java. Jego zadaniem będzie otwarcie strony z aukcjami, kliknięcie guzika „zaloguj", wypełnienie pól z nazwą i hasłem użytkownika, kliknięcie guzika login i ostatecznie zweryfikowanie, czy dane użytkownika są wyświetlane na stronie (Listing 1.).

Przedstawiony test, a w zasadzie jego kod, ma wiele wad. Jedną z głównych jest fakt, że aby go zrozumieć, należy posiadać wiedzę na temat samego frameworku Sele-

Listing 1.

```
WebDriver driver = new FirefoxDriver();
driver.get(„http://www.aukcje.pl");
WebElement login_link = driver.findElement(By.
linkText(„zaloguj"));
login_link.click();
WebElement user_name = driver.findElement(By.id(„userForm_lo-
gin"));
user_name.sendKeys(„Jan_Kowalski");
WebElement password = driver.findElement(By.id(„userForm_pas-
sword"));
password.sendKeys(„TajneHaslo1");
WebElement login_button = driver.findElement(By.id(„login"));
login_button.click();
WebElement user_link = driver.findElement(By.linkText(„Jan
Kowalski"));
assertThat(user_link.isDisplayed(),is(true));
driver.quit();
```

nium. Na samym początku zauważamy zainicjalizowanie obiektu WebDrivera. Jest on odpowiedzialny za komunikację z przeglądarką. Dzięki niemu możemy też przekazywać polecenia w dalszej części. Metoda *findElement()* zwraca nam określony obiekt, na którym możemy wykonywać takie interakcje, jak kliknięcie - *click()* czy wpisanie tekstu - *sendKeys()*. Dodatkowo należy przekazywać tzw. lokatory, które wiążą obiekt w teście z fizycznym obiektem na stronie. To wszystko sprawia, że ten prosty przypadek testowy jest prawie niezrozumiały na pierwszy rzut oka i nawet osoba na co dzień pracująca z Selenium musiałaby spędzić chwilę, by zrozumieć logikę testu.

Innym problemem, jaki możemy zauważyć, jest fakt ścisłego powiązania logiki testu z techniczną częścią odpowiedzialną za jego wykonywanie. W przypadku zmian w aplikacji musimy wprowadzać również zmiany w testach. Z racji braku warstwy pośredniczącej miedzy testem a wykorzystywanymi elementami Selenium modyfikację dotykają bezpośrednio klas testowych. Kod do obsługi konkretnego obiektu na stronie jest duplikowany wszędzie tam, gdzie jest on wywoływany w teście; ewentualne zmiany będą wymagały modyfikacji w każdym z tych miejsc.

Listing 2.

```
By USER_NAME = By.id(”username”);
By PASSWORD = By.id(”password”);
By LOGIN_BUTTON= By.id(”login”);

public WebDriver driver;

public LoginPage(WebDriver driver){
     this.driver = driver;
}


public void clickLogin(){
driver.findElement(LOGIN_BUTTON).click();

public void typeUsername(String userName){
driver.findElement(USER_NAME).sendKeys(userName);

}

public void typePssword(String password){
driver.findElement(PASSWORD).sendKeys(password);

}
```

Wzorzec Page Object

Podstawowym rozwiązaniem powyższych problemów jest wprowadzenie do naszych testów wzorca Page Objectów. Jego koncepcja polega na mapowaniu strony internetowej lub jej fragmentu na klasę w kodzie. Klasa ta zawiera pola odpowiadające elementom na stronie a metody odwzorowują interakcje, jakie możemy wykonać na tych elementach. Na przykład dla prostego formularza logowania z polami do wypełnienia nazwy i hasła użytkownika oraz guzikiem login, klasa „Page Objectu" może wyglądać tak, jak na listingu 2.

Jak widać, zapewniamy tu metody, które mogą być zrozumiałe z biznesowego punktu widzenia. Nasz test z wykorzystaniem Page Objectów może wtedy wyglądać tak jak na listingu 3, przy czym przedstawiony jest już tu tylko fragment odpowiedzialny za samą logikę testu. Dodatkowo zastosowano mechanizm łańcuchowania metod (ang. chaining) dla poprawienia czytelności kodu. Wymaga on tylko zwracania referencji do danego obiektu w metodach Page Objectów.

W tej formie nasz test nie zawiera już „niepotrzebnych" szczegółów technicznych frameworku. Dzięki przyjaznym metodom zrozumienie logiki staje się prostsze. Te same metody Page Object'ów mogą być używane w wielu innych testach. Teraz zmiany w aplikacji muszą być odwzorowywane w pojedynczych miejscach poszczególnych Page Object'ów a testy mogą być w znaczącym stopniu nienaruszone.

Delegaty

Dość naturalnym kolejnym krokiem, jaki możemy wykonać, jest dodanie kolejnej warstwy abstrakcji nad naszymi Page Objectami. Tutaj naszą intencją jest dalsze agregowanie metod w kroki testowe. W projektach, w których pracowałem, nazywane były one delegatami. Tę koncepcję może zobrazować prosty przykład logowania. W naszym teście często jesteśmy zainteresowani wykonaniem określonego kroku, np. zalogowania się jako konkretny użytkownik z danym hasłem. Możemy przy tym nie chcieć martwić się o poszczególne „drobne" kroki, jak na przykład użycie określonych pól i kliknięcie guzika. Ważny jest sam fakt wykonania logowania. Możemy więc zgrupować metody odpowiadające za logowanie w jedną metodę, tak jak na listingu 4.

Listing 3.

```
homePage.clickLoginLink();
loginPage.typeUsername(„Jan_Kowalski");
    .typePassword(„TajneHaslo1");
    .clickLogin();
assertThat(homePage.getUser(),is(„JanKowalski"));
```

Listing 4.

```
public void loginAsUserWithPassword(String user, String password)
      {
      loginPage.typeUsername(user)
          .typePassword(password)
          .clickLogin();
      }
```

Zauważamy tu ważną z punktu widzenia utrzymania testów kwestię. W wypadku zmiany formatki do logowania czy nawet całego sposobu uwierzytelniania sama zmiana może być odwzorowana w tej pojedynczej metodzie i Page Objectach a reszta testów, która z niej korzysta, pozostaje nienaruszona. Dodatkowo jest to kolejny krok ku poprawie czytelności testów, co jest szczególnie ważne wtedy, gdy zawierają wiele złożonych kroków.

Takie funkcje agregujące można by oczywiście umieszczać w samych Page Objectach i o ile w tym trywialnym wypadku logowania nie mielibyśmy z tym problemów, to w rzeczywistych rozwiązaniach z wieloma klasami Page Objectów należy umieszczać je w oddzielnych klasach delegat odpowiedzialnych za określone funkcjonalności biznesowe. Same funkcje w delegatach mogą zawierać metody z wielu Page Objectów, jak i innych delegat. Mogą być odpowiedzialne za nawigowanie zarówno pomiędzy kilkoma punktami w aplikacji, jak i za wszelkie interakcje z elementami na stronie. Warto też dodać, że „asercje" nie są umieszczanie ani w Page Objectach, ani delegatach, tylko w samym teście.

Listing 5.

```
@Inject
LoginPage loginPage
```

Zarządzanie zależnościami

W czasie tworzenia naszych testów posługujemy się obiektami delegat i Page Objectów a te potrzebują do działania przekazanego im obiektu WebDrivera. Dodatkowo możemy potrzebować innych obiektów, które mogą być częścią naszego frameworka. Wszystkie te wspomniane obiekty trzeba inicjalizować wraz z zadbaniem o przekazywanie im niezbędnych zależności. Wprowadza to oczywiście narzut czasowy potrzebny na manualne ustawienie wszystkich tych elementów, jak również dodatkowy kod, który nie wydaje się związany z naszym testem. Co ważniejsze, możemy tu zauważyć fakt, że nasza klasa testowa powinna być odpowiedzialna wyłącznie za wykonanie testów, a nie za inicjalizowanie określonych obiektów. Te obiekty powinny być dla niej niejako dostarczane od agenta, który „wie", jak określony obiekt powinien być zainicjalizowany i sam zapewni mu potrzebne zależności.

Powyższe problemy możemy rozwiązać dzięki wprowadzeniu tzw. odwróconego sterowania, realizowanego przez wzorzec projektowy wstrzykiwania zależności (ang. Dependency Injection). Dzięki niemu możemy zrealizować budowanie całego drzewa zależności oddzielnie od klasy testowej, a ona sama dostaje już odpowiednio zainicjalizowane obiekty, których

potrzebuje. Implementacja całego mechanizmu jest zależna od używanego języka programowania i frameworku. Wymaga też pewnego nakładu pracy na jego wdrożenie, który jednak zwraca się w postaci późniejszej łatwości i elastyczności użycia. Przykładowo w Javie możemy posłużyć się frameworkiem Spring czy Guice (autorstwa firmy Google). Żądanie określonego obiektu Page Objectu może sprowadzić się jedynie do zdefiniowania jego nazwy wraz z anotacją @Inject, tak jak na listingu 5.

Generator kodu

Wracając do Page Objectów możemy zauważyć, że po zapoznaniu się z ich koncepcją samo tworzenie nowych klas nie stanowi większego problemu. Często też sprowadza się to do manualnego wyszukiwania lokatorów na stronie i tworzenia dla nich odpowiednich zmiennych, a następnie elementarnych metod. Proces ten wymaga czasu, lecz w rzeczywistości nie jest skomplikowany.

W tworzeniu Page Objectów pomocne są narzędzia do łatwego lokalizowania elementów na stronie, takie jak np. Firebug. Możemy również posłużyć się dostępnymi generatorami całych klas Page Object'ów. Jednym z bardziej zaawansowanych narzędzi jest SWD (http://swd-tools.com), który znacząco może wspomóc nas w tworzeniu naszych klas.

W jednym z projektów, w którym pracowałem, mieliśmy własne rozwiązanie stworzone przez członka zespołu. Opierało się na skrypcie napisanym w języku JavaScript, który generował całe szkielety gotowych Page Objectów wraz z odpowiednimi konstruktorami czy nawet komentarzami. Rozwiązanie sprawdzało się znakomicie również dlatego, że elementy w aplikacji miały ustandaryzowane nazewnictwu, dzięki czemu mogliśmy generować większość metod, jakie były nam potrzebne.

Zarządzanie testami

Jedną z podstawowych kwestii przy tworzeniu frameworku testowego jest decyzja o sposobie uruchamiania testów. Operując na często pokaźnym zbiorze stworzonych skryptów testowych, musimy posiadać wygodny i elastyczny mechanizm do uruchamiania konkretnego testu, określonej grupy czy wszystkich testów. Doskonałym rozwiązaniem jest tutaj użycie biblioteki, która w zamyśle mogła być stworzona dla testów jednostkowych, jednak równie dobrze sprawdza się przy testach Selenium. Każdy z języków wspieranych przez Selenium zapewnia własne implementacje takiej biblioteki.

Przy wyborze konkretnego rozwiązania warto zwrócić uwagę na to, czy są wspierane następujące funkcjonalności:

- Definiowanie metod uruchamianych przed lub po klasie i metodach testowych w celu przygotowywania środowiska testowego i „sprzątania" po teście.
- Definiowanie własnych grup testowych np. "Smoke tests", "Read only", "Disabled on IE8" itd.
- Ustawianie kolejności i zależności między testami.
- Parametryzacja danych testowych dla Data-driven testing (DDT) – czyli uruchamianie jednego testu wielokrotnie z różnymi danymi wejściowymi.
- Dodatkowe opisy testów pomocne w raportach.
- Wsparcie dla równoległego uruchamiania testów.

Przykładem z języka Javy dla biblioteki spełniającej wyżej wymienione postulaty jest TestNG.

Zarządzanie środowiskiem testowym

W poprzednim punkcie wspomniałem o zagadnieniu przygotowania środowiska przed uruchomieniem testu oraz o „posprzątaniu" po skończonym teście. To jedna z podstawowych zasad testów automatycznych – aby każde testy były niezależne. Jest to wymagane do tego, aby uznać wynik testu za wiarygodny. Dodatkowo mamy pewność, że testy uruchamiane równolegle również nie będą wpływały wzajemnie na swoje wyniki.

Z tego powodu przed wykonaniem testu należy ustawić niezbędne dane testowe czy środowiskowe. W wielu wypadkach bardzo ważne jest też to, by zestaw danych był generowany unikalnie dla konkretnego przypadku. Wszystko to może być usprawnione i wręcz zautomatyzowane dzięki odpowiednio przygotowanym funkcjom. Mogą one przykładowo wykorzystywać dostępne serwisy naszej aplikacji, komunikując się przez polecenia SOAP czy REST. W wypadku braku takich serwisów możemy sprawdzić, czy mamy możliwość bezpośredniego dostępu do bazy danych lub plików aplikacji, w których przechowywane są dane potrzebne dla naszych testów.

Listing 6.

```
@BeforeClass
public void prepareEnvironment(){
     createTestOrganisation();
     createTestWorker();
     assignPermissionsToWorker();
}
```

Warto też przemyśleć formę czyszczenia przygotowanych danych testowych. Ważna jest możliwość zweryfikowania konkretnych danych po wykonanym teście. Jeśli nie możemy ich zostawić w systemie, to warto sprawdzić, czy zamiast całkowitego usuwania można je w pewien sposób deaktywować, dzięki czemu z poziomu aplikacji będą niewidoczne, a my nadal będziemy mieli do nich dostęp.

Przykładowy zestaw funkcji ustawiających dane testowe przed klasą testową i deaktywujące je po jej zakończeniu jest przedstawiony na listingu 6.

Logowanie

Ostatnią sprawą, jaką chciałbym poruszyć, jest śledzenie kroków wykonywania naszych testów. Musimy wiedzieć, co dokładnie test wykonał i w razie problemów móc jak najszybciej określić tego przyczynę. Najprostszym sposobem jest dodanie funkcji wypisujących informacje logujące do naszych testów. Naturalnym usprawnieniem jest użycie gotowych bibliotek odpowiednio formatujących informacje wyjściowe, dodających również znaczniki czasowe czy pozwalające na ustawienie stopnia szczegółowości zwracanych informacji.

Szybko jednak zauważymy, że jest to dość pracochłonna czynność. Pewnym usprawnieniem jest przeniesienie jak największej liczby tych poleceń wewnątrz klas Page Objectów czy delegat, aby móc je dzielić pomiędzy wieloma testami.

To też jednak nie jest rozwiązaniem idealnym, ponieważ nadal pozostawia nas z manualnym zadaniem wpisywania tych komend. Drugą istotną kwestią jest to, że dodając polecenia logujące czy to do klas testowych, czy Page Objectów i delegat, łamiemy zasadę pojedynczej odpo-

AUTOR

Michał Sierzputowski

Absolwent wydziałów Elektroniki, Telekomunikacji i Informatyki oraz Zarządzania i Ekonomii na Politechnice Gdańskiej. Od 2008 roku związany z testowaniem oprogramowania. Zwolennik projektów Agile i wdrażania automatyzacji w testach. Zdobywał doświadczenie w kilku międzynarodowych korporacjach. Obecnie pracuje jako starszy inżynier testów w gdańskim oddziale firmy Kainos, gdzie miał okazję wdrażać testy automatyczne oparte na Selenium w kilku projektach prowadzonych w Scrumie.

Prywatnie pasjonat jazdy na rowerze i zimowych kąpieli w Bałtyku.

wiedzialności klasy. W tym wypadku klasa zostaje obarczona dodatkowym zadaniem, które nie jest związane bezpośrednio z jej przeznaczeniem.

Z tym problemem możemy sobie poradzić za pomocą programowania aspektowego (ang. Aspect-oriented programming, AOP). Dzięki niemu możemy oddzielić od siebie niezwiązane funkcjonalnie części naszego rozwiązania. Dodatkowo cała funkcjonalność logowania jest zarządzana z jednego miejsca. Należy tylko zdefiniować określone filtry i logikę, jaką chcemy zaimplementować. Przykładowo możemy przechwytywać nasze metody z testów i przed każdym ich wywołaniem, jak i zaraz po skończeniu ich zadania, dodać odpowiednią informację do logów. Za mechanizm przechwytywania jest odpowiedzialny tzw. Tracing Aspect. Implementacja całego rozwiązania jest ściśle zależna od języka programowania i użytego frameworka.

Podsumowanie

Kilka omówionych powyżej punktów to jedynie zaznaczenie pewnych problemów, z jakimi musimy się zmierzyć w trakcie tworzenia frameworka dla testów automatycznych. Najważniejszą kwestią jest przyłożenie odpowiedniej wagi do projektu testów, gdyż w przeciwnym wypadku zamiast zwiększenia jakości testów możemy skończyć z problemem mnóstwa trudnych i kosztownych w utrzymaniu testów. Ze swojego doświadczenia mogę polecić ścisłą współpracę testerów i programistów w pracy nad takim rozwiązaniem. Daje to możliwość wprowadzania odpowiednich zmian technicznych do testów oraz – co nie mniej istotne - możliwość dzielenia się wiedzą między członkami zespołu.

Adam Marciszewski

Optymalizacja automatycznych testów regresywnych w organizacji transformującej do Agile

Wprowadzenie

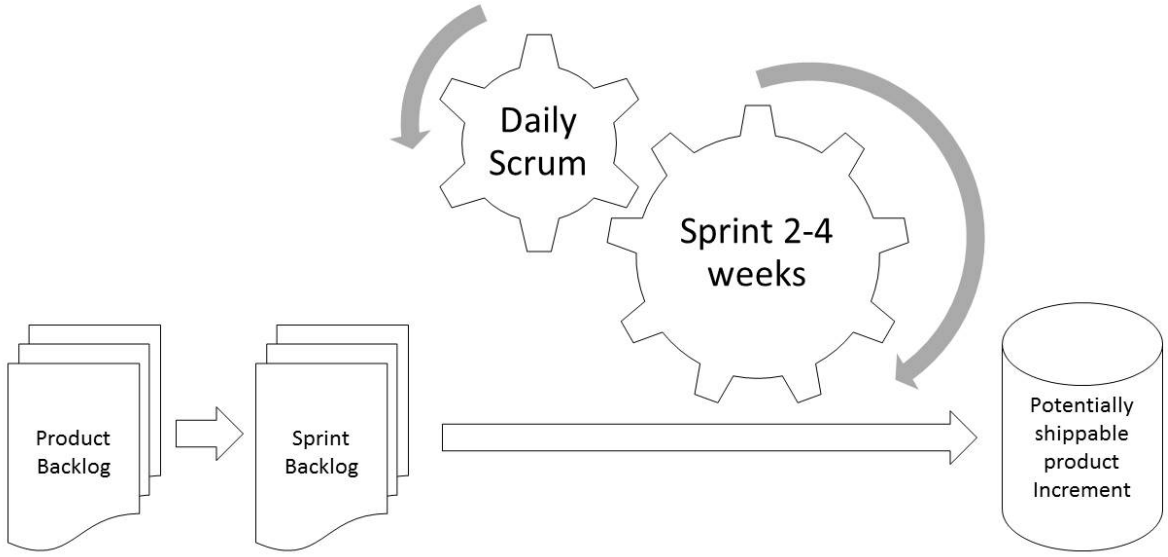
Z pewnością każdy z nas, wykonując swoją pracę, zetknął się z problemem nadmiernie rozrastających się testów regresywnych. Początkowo zgrabny zestaw w miarę rozwoju testowanego systemu zaczyna niepokojąco przybierać na wadze. W pewnym momencie nasze testy, jeszcze nie tak dawno skuteczne i efektywne, stają się uciążliwym balastem, który skutecznie spowalnia projekt. W takiej sytuacji z pomocą przychodzą różne metody testowania i optymalizacji testów. Często pojawia się pokusa, by odchudzić zestaw testów regresywnych, pozbywając się niektórych przypadków testowych. Kolejnym wyjściem może być wydłużenie czasu pomiędzy kolejnymi cyklami regresji. Ale jak sobie poradzić, kiedy nie znamy wymagań, nie znamy stopnia pokrycia systemu testami oraz skuteczności testów? Dodatkowo sytuację utrudnia fakt, że klient właśnie podjął decyzję, by przejść do metodyk zwinnych.

Kontekst projektu

Chcąc mówić o optymalizacji testów regresywnych, należy rozpocząć od przedstawienia kontekstu projektu, gdyż to właśnie kontekst determinuje i uzasadnia rozwiązania wybrane do rozwiązania danego problemu. Kontekst realizowanego przez nas projektu rysował się następująco: rozproszony geograficznie projekt, realizowany na rzecz dużego międzynarodowego klienta działającego w sektorze telekomunikacji. Zarówno organizacja projektu, jak i architektura produktu charakteryzowały się dużą złożonością. Projekt miał charakter częściowo utrzymaniowy, częściowo rozwojowy (podział prac: 60%/40%). Rozwijany produkt był bazą, na której opierały się inne produkty, stąd wysokie ryzyko „zainfekowania" produktów wyższego poziomu oraz przedostania się defektów do środowiska klienta. Krótko po przejęciu projektu od innego podwykonawcy klient podjął decyzję, by transformować do SCRUM.

Typowe podejście

Pełni zapału i entuzjazmu rozpoczęliśmy pracę nad nowym projektem. Oczywistym było, że odziedziczone testy regresywne wymagają adaptacji, by sprostać specyfice projektu realizowanego zwinnie. Dobór odpowiedniego zestawu testów oraz optymalizacja czasu wykonania wydają się być standardowym działaniem, które może być czasochłonne, ale nie powinno przysporzyć zbyt wielu trudności. Standardowym podejściem przy realizacji takiego zadania jest analiza strategii testów, planu testów oraz specyfikacji testowej. Specyfikacja wymagań jest konfrontowana z dokumentacją testową celem określenia pokrycia systemu testami. W dalszym kroku bazując na analizie ryzyka, dobralibyśmy odpowiedni zestaw testów regresywnych. Niestety, niedługo po rozpoczęciu pracy okazało się, że w tym przypadku standardowe podejście nie miało szansy powodzenia.



Wyzwania

W chwili rozpoczęcia pracy nad optymalizacją testów wyszło na jaw, że brakuje podstawowej dokumentacji, która pozwoliłaby sprawnie zrealizować zadanie. Specyfikacja testowa była niekompletna lub nieaktualna. Brakowało planu oraz strategii testów. Wymagania były zdefiniowane tylko dla nowych cech. Brakowało wymagań definiujących dotychczas dostarczony produkt. Brakowało wiedzy na temat aktualnego pokrycia systemu testami. Czas wykonania pojedynczej kampanii automatycznych testów regresywnych zajmował ponad 30 godzin. Z tego powodu przeprowadzenie dziennej regresji było niemożliwe. Praca wymagała przeanalizowania ponad tysiąca przypadków testowych. Brakowało szczegółowej wiedzy na temat skuteczności przejętych testów. Wiedzieliśmy tylko, że skuteczność testowania jest niska. Sporo błędów wykrywano na dalszych etapach produkcji lub u klienta. Ze względu na brak możliwości określenia pokrycia wymagań systemowych testami należało opracować nowe podejście do optymalizacji testów w tak wymagających warunkach.

Cel

Opracowując plan optymalizacji automatycznych testów regresywnych, postawiono następujące cele:

- Czas wykonania pełnej dziennej regresji musi być mniejszy niż 12 godzin.
- Skrócenie czasu wykonania regresji nie może obniżyć skuteczności wykrywania błędów.
- Skrócenie czasu regresji nie może zredukować pokrycia systemu testami.
- Obszary systemu obarczone większym ryzykiem będą testowane intensywniej.
- Obszary kluczowe, krytyczne dla klienta będę testowane intensywniej.
- Granulacja przypadków testowych (ilość przypadków/wymaganie) będzie zmienna i zależna od poziomu ryzyka oraz ważności cechy dla klienta.

Założenia

Przedmiotem testów była platforma, na której budowano produkty wyższego poziomu. Testowany system nie posiadał interfejsu graficznego. Był za to wyposażony w liczne interfejsy oparte na różnych technologiach i służące do komunikacji z systemami wyższego poziomu. Wiedząc, że zdobycie specyfikacji wymagań jest niemożliwe, musieliśmy w inny sposób oszacować pokrycie systemu testami. Zdecydowaliśmy się przyjąć jedno odważne założenie. Mianowicie system od dłuższego czasu jest używany przez klienta. Klient nie ma zastrzeżeń do zakresu funkcjonalnego. Zatem testowany system posiada wszystkie zamówione/wymagane przez klienta funkcjonalności. Na tej podstawie założyliśmy, że produkt spełnia wszystkie wymagania funkcjonalne. Zatem walidacja produktu przebiegła poprawnie, a obserwowalne problemy dotyczą weryfikacji. Na tej podstawie przyjęto następujące założenia:

- Klient otrzymał wszystkie wymagane cechy/funkcjonalności.
- Dostarczone cechy spełniają wymagania funkcjonalne.
- Walidacja była przeprowadzona poprawnie.

W kwestii nowych funkcjonalności nie było problemu z oszacowaniem pokrycia wymagań testami, gdyż te były wyczerpująco udokumentowane.

Opis metody krok po kroku

Zastosowane podejście do optymalizacji automatycznych testów regresywnych zostało podzielone na cztery fazy:

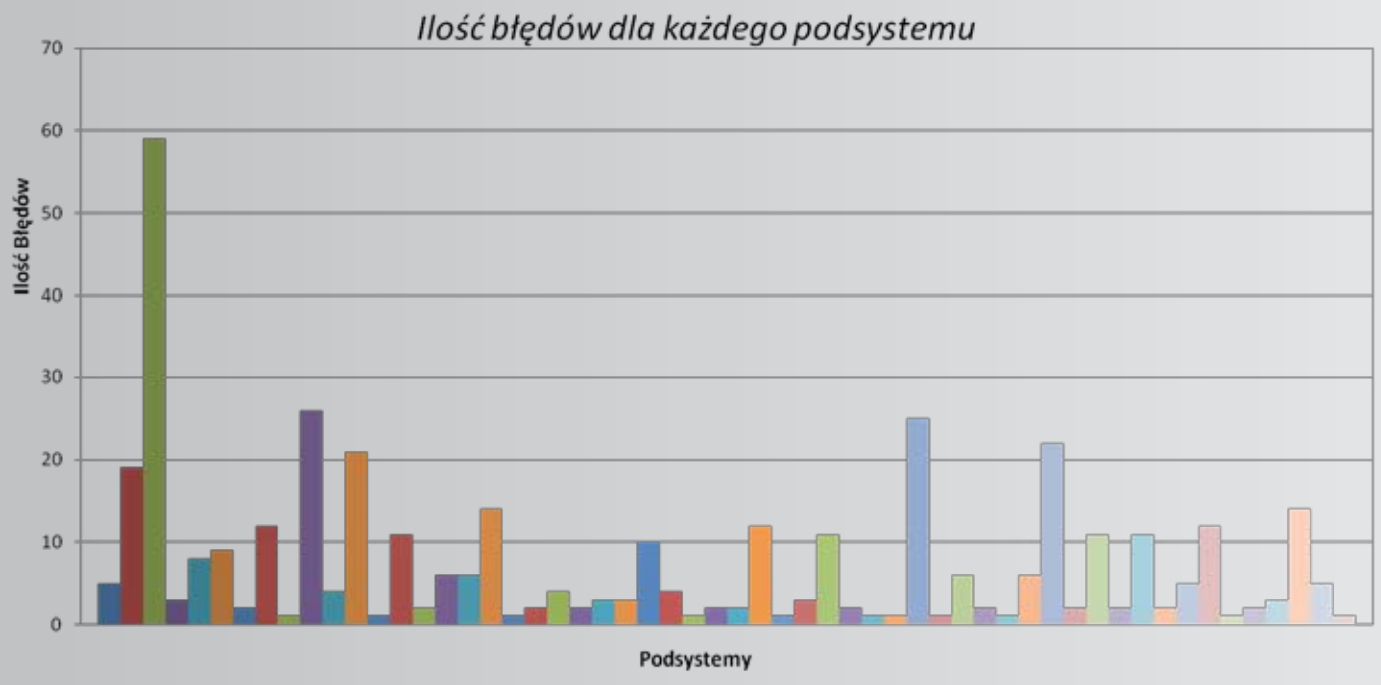
Faza 1. – Analiza i optymalizacja zbioru testów regresywnych
Faza 2. – Optymalizacja techniczna
Faza 3. – Monitorowanie i dostrajanie (minimalizacja ryzyka)
Faza 4. – Start!

Faza 1. – Analiza i optymalizacja zbioru testów regresywnych

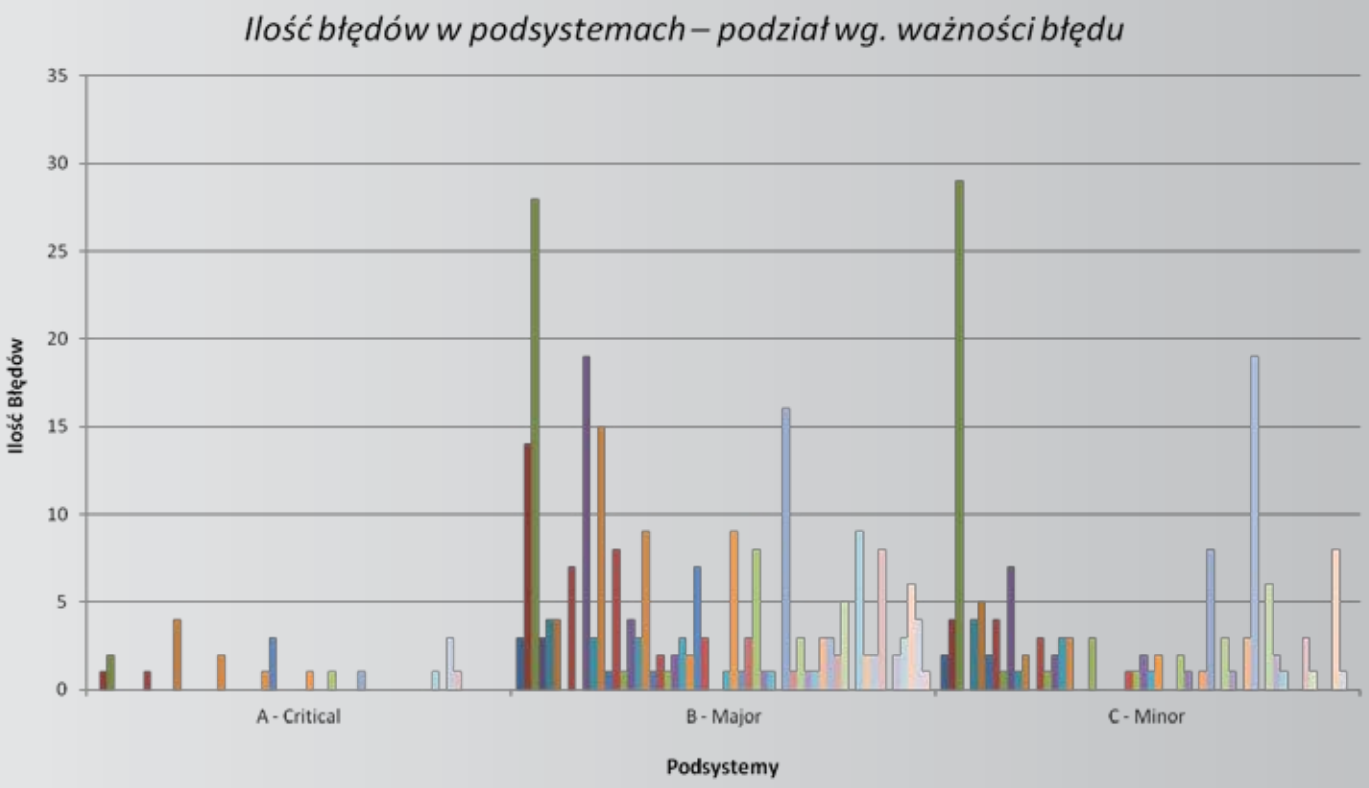
Celami fazy 1 były: identyfikacja wymagań, oszacowanie pokrycia systemu testami, ocena skuteczności testowania, identyfikacja ryzyka poprzez zbadanie rozkładu błędów w testowanym systemie, identyfikacja obszarów kluczowych dla klienta, a w końcu dobór nowego, zoptymalizowanego zbioru testów regresywnych.

Na podstawie przyjętych założeń zakres funkcjonalny testowanego systemu został zidentyfikowany poprzez testy biało-skrzynkowe i statyczną analizę kodu. W szczególności przeanalizowano deklaracje i definicje interfejsów testowanego systemu. Tak zewidencjonowane funkcjonalności posłużyły za wymagania funkcjonalne, które następnie skonfrontowano z istniejącymi przypadkami testowymi. Na tej pod-

Rysunek 1. Rozkład błędów z uwzględnieniem każdego podsystemu



Rysunek 2. Rozkład błędów dla każdego podsystemu z uwzględnieniem ważności błędu



stawie określono pokrycie funkcjonalności systemu testami.

Równolegle do analizy kodu rozpoczęto gromadzenie i analizę wyników automatycznych testów regresywnych na przestrzeni dwóch ostatnich lat. Z systemu zarządzania defektami wydobyto dane dotyczące zgłoszonych błędów, uwzględniając wersję testowanego systemu, podsystemu oraz źródła zgłoszenia błędu. Po przeanalizowaniu i przetworzeniu wszystkich zebranych danych otrzymano informację na temat rozkładu błędów w testowanym systemie. Rozkład błędów w testowanym systemie z uwzględnieniem każdego podsystemu został przedstawiony na rysunku 1.

Rozkład błędów w testowanym systemie jest źródłem cennej wiedzy na temat obszarów systemu, które ze względu na dużą liczbę defektów obarczone są wysokim ryzykiem, zatem zasługują na szczególną uwagę podczas optymalizacji i doboru testów regresywnych.

Pomimo zalet rozkład błędów nie dostarcza informacji na temat ważności błędu. Brakującą wiedzę można uzupełnić, klasyfikując rozkład błędów według ważności błędu. Rysunek 2 przedstawia rozkład błędów sklasyfikowany według ważności błędu. Z rysunku nr 2 wynika, że najwięcej zgłoszono błędów o ważności średniej i niskiej. Wykryte błędy dotyczyły głównie podsystemu zielonego, niebieskiego, fioletowego i brązowego.

Badając rozkład błędów, a następnie klasyfikując go według ważności błędu, uzyskano cenną informację na temat obszarów o podwyższonym poziomie ryzyka. Niestety, nie jest to informacja kompletna, gdyż nie wiadomo, ile z obserwowanych

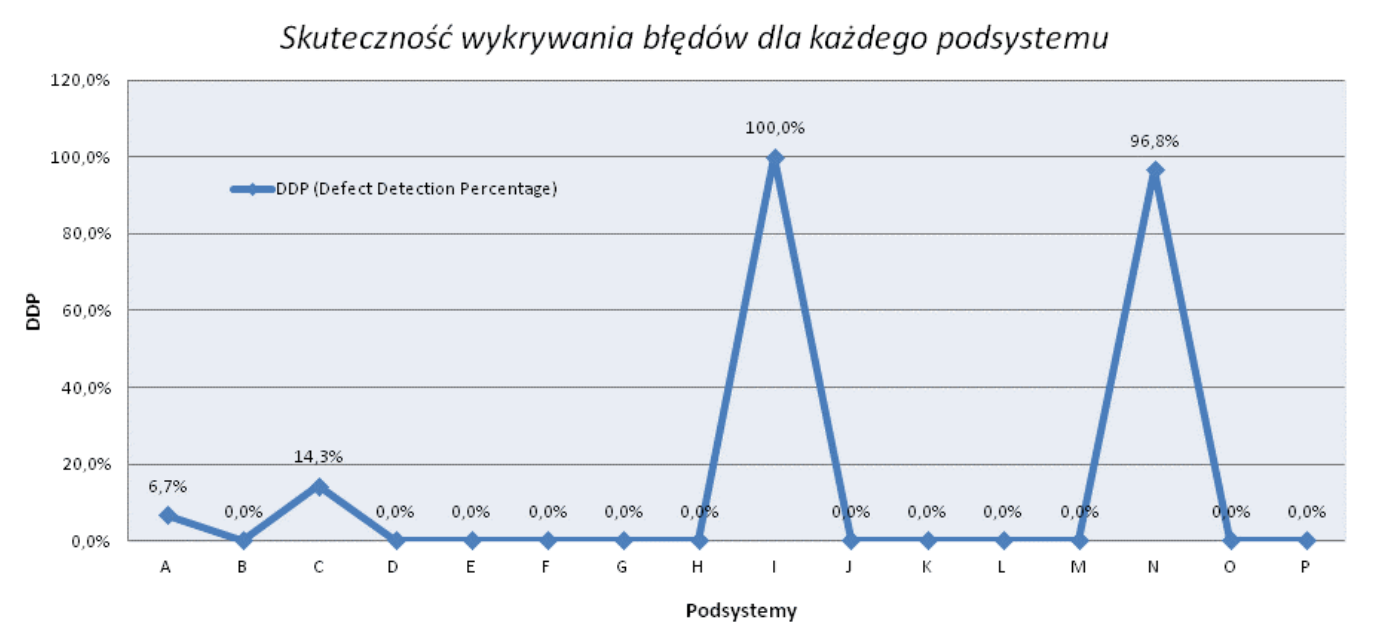
błędów zostało wykrytych przez nasze testy regresywne, a ile przez pozostałe testy. Taką informację można zdobyć, mierząc efektywność testowania (ang. DDP – Defect Detection Percentage). Efektywność Testowania określona jako skuteczność wykrywania błędów przedstawia się następującym wzorem:

DDP =
*(Liczba błędów wykrytych przez nasze testy/Liczba wszystkich wykrytych błędów) * 100%*

Z podstaw ISTQB Foundation pamiętamy, że na skuteczność testów do wykrywania błędów znaczący wpływ ma efekt pestycydów. Efekt pestycydów mówi o tym, że z biegiem czasu testowany system uodparnia się na testy. Pociąga to za sobą konieczność ciągłego utrzymywania, modyfikowania i usprawniania testów. Podobnie rzecz się ma z testami automatycznymi. Również one wymagają ciągłego utrzymywania i udoskonalania, by utrzymać eEfektywność testowania na wysokim poziomie. Rysunek nr 3 przedstawia efektywność testowania dla każdego podsystemu w testowanym produkcie.

Wykres ilustrujący efektywność testowania w testowanym systemie ujawnia, że dla podsystemów „A" i „C" skuteczność testowania jest bardzo niska. Testy regresywne znalazły odpowiednio tylko 6,7% i 14,3% błędów. Oznacza to, że pozostałe błędy były wykryte przez inne testy lub niestety przez klienta. Z kolei dla podsystemów „I" oraz „N" skuteczność testowania jest bardzo wysoka: wynosi odpowiednio 100% i 96,8%. W tym przypadku rozważane testy regresywne wykryły wszystkie lub prawie wszystkie z odnotowanych defektów. Punkty, w których skuteczność testowa-

Rysunek 3. Efektywność testowania (ang. DDP – Defect Detection Percentage)

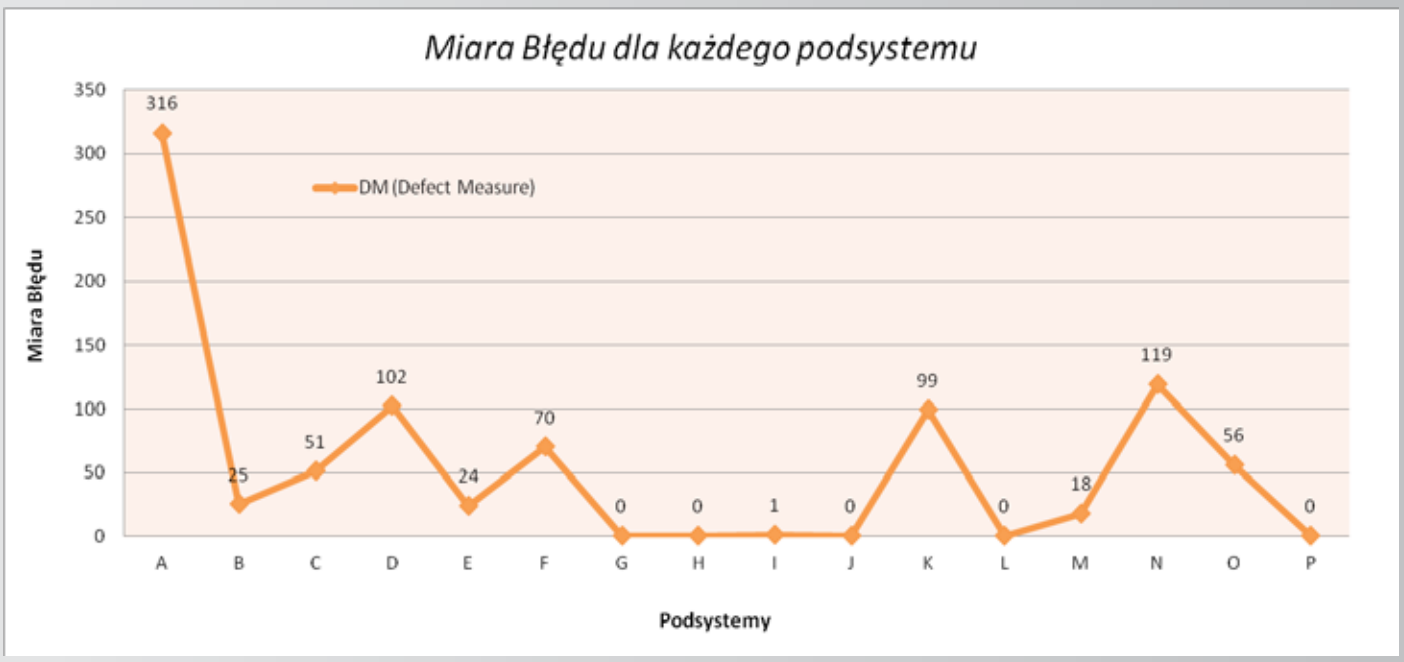


nia wynosi 0%, dostarczają dwuznacznej informacji. DDP na poziomie 0% może oznaczać zerową skuteczność detekcji błędów lub całkowite nieodnotowanie błędów w danym obszarze. Dwuznaczność informacji można rozwiązać, weryfikując dane wydobyte z systemu zarządzania defektami lub porównując ***efektywność testowania*** z ***miarą błędu.***

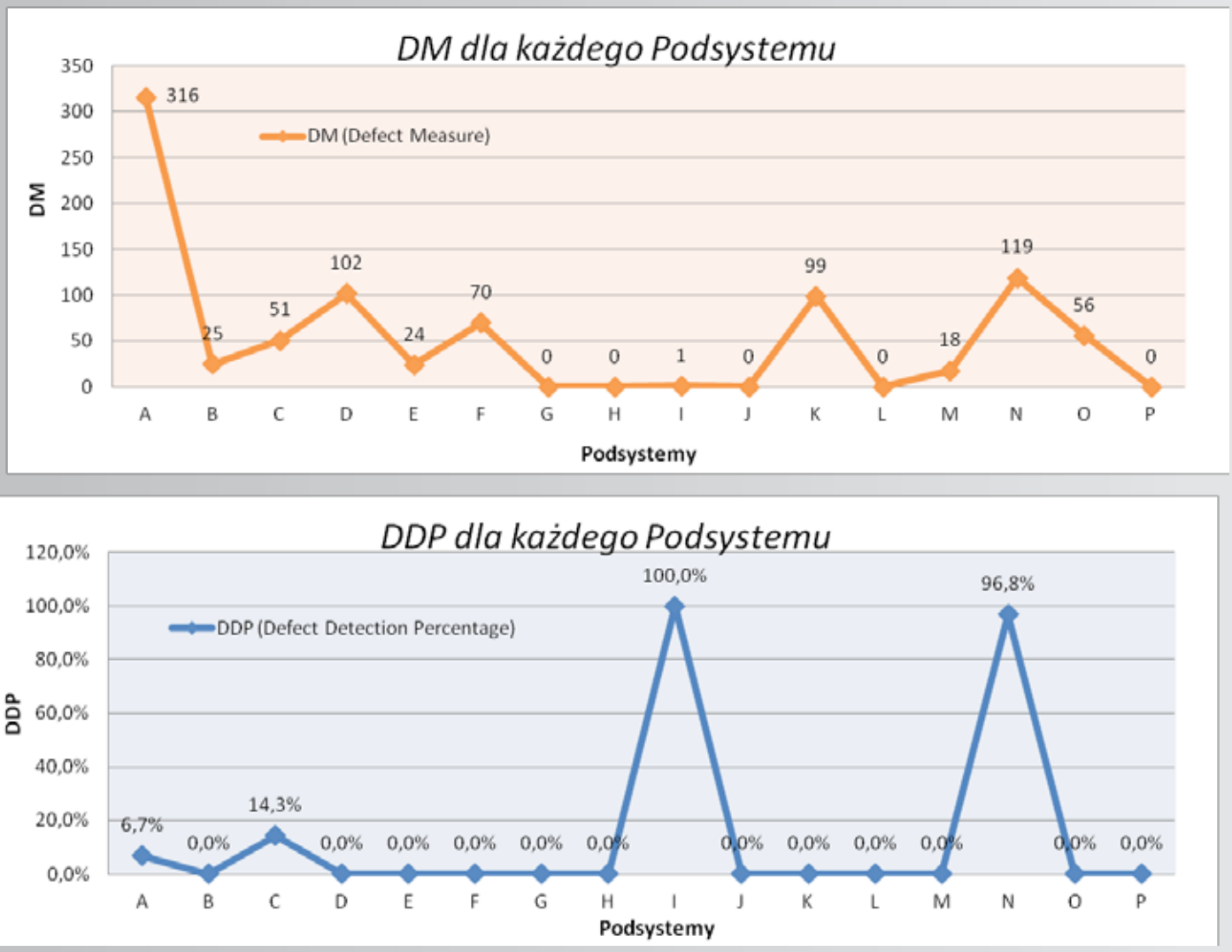
Znamy już dokładny rozkład błędów, uwzględniający ich ważności oraz usytuowanie w testowanym systemie. Wiemy, jaka jest skuteczność detekcji błędów w każdym z testowanych podsystemów. W tym momencie jest to naprawdę cenna wiedza, którą można się posiłkować, planując optymalizację testów. Niestety, nadal jest to informacja niekompletna i niemiarodajna, gdyż nie uwzględnia potencjalnego kosztu związanego z wystąpieniem błędu. Mówiąc koszt, można to interpretować jako szkodę wyrządzoną klientowi. Może to być również czas lub koszt naprawienia błędu. Informacji dotyczącej kosztu naprawy lub kosztu wystąpienia błędu dostarcza ***miara błędu (ang. DM - Defect Measure).*** Miarę błędu liczy się w prosty sposób. Mianowicie każdemu stopniowi ważności błędu przypisuje się odpowiednią wagę. Wagi relatywnie oddają koszt, czas lub szkodę związaną z wystąpieniem błędu o danej ważności.

W prezentowanym przykładzie wagi odzwierciedlają koszt naprawy błędu o danej ważności. Informację o kosztach lub czasie naprawy błędów można uzyskać od kierownika projektu/produktu lub osoby odpowiedzialnej za jakość i procesy w organizacji.

Rysunek 4. Miara błędu dla każdego podsystemu w testowanym produkcie



Rysunek 5. Porównanie sutecznośći testowania (DDP) I miary błędu (DM)



W prezentowanym przykładzie przyjęto następujące wagi:

Błąd krytyczny = 10
Błąd średni = 5
Błąd niski =1

Oznacza to, że koszt naprawy błędu krytycznego jest dwukrotnie wyższy niż koszt naprawy błędu o średniej ważności i dziesięciokrotnie wyższy od naprawy błędu o niskiej ważności.

W dalszej kolejności miarę błędu oblicza się, korzystając z poniższego wzoru:

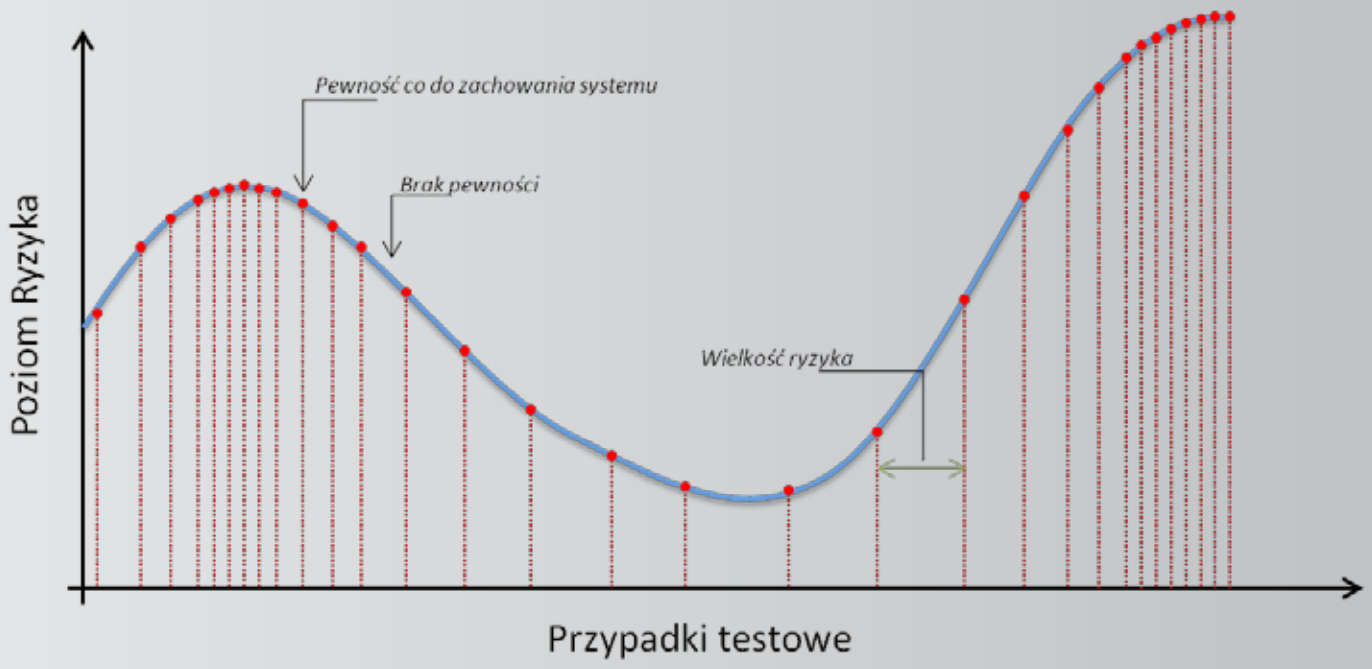
$$DM=10*j+5*k+l$$

Gdzie j, k, l oznaczają odpowiednio:

j – ilość błędów krytycznych
k – ilość błędów o średniej ważności
l – ilość błędów o niskiej ważności

Miara błędu dla testowanego systemu z uwzględnieniem każdego podsystemu została przedstawiona na rysunku nr 4.

Rysunek 6. Zmienna granulacja przypadków testowych w zależności od poziomu ryzyka



Najbardziej interesujących wniosków dostarcza prównanie obu metryk – miary błędu i skuteczności testowania. Przyjrzyjmy się bliżej, co można zaobserwować, zestawiając oba wykresy. Pórównanie miary błędu (DM) i skuteczności testowania (DDP) przedstawia rysunek 5.

Oto wnioski, jakie można wysunąć analizując oba wykresy. Dla podsystemów „A", „D", „F", „K" miara błędów posiada wysoką wartość (jest to jednostka relatywna, bezwymiarowa). Równocześnie skuteczność testowania dla wymienionych podsystemów jest bardzo niska lub zerowa. Co za tym idzie, koszt naprawy niewykrytych błędów jest dla organizacji bardzo wysoki i może zaważyć na dochodowości projektu. Co więcej, niewykrycie błędów w tych podsystemach może znacząco zaszkodzić dobru klienta. Dlatego optymalizując testy regresywne, szczególną uwagę i staranność należy poświęcić tym obszarom (podsystemom).

Ciekawie przedstawia się sytuacja dla podsystemu „I", gdzie skuteczność testowania wynosi 100%, a miara błędu jest bardzo niska. W tym wypadku może to sugerować, iż podsystem „I" jest testowany zbyt intensywnie. W porównaniu do niskiego ryzyka skojarzonego z podsystemem „I", nakład środków na testy może być zbyt wysoki.

W przypadku podsystemu „N" wysoki poziom związanego z nim ryzyka jest skompensowany wysoką skutecznością wykrywania błędów. W związku z tym, optymalizując testy dla podsystemu „N", należy wprowadzić jedynie niewielkie poprawki kosmetyczne.

W tym momencie posiadamy już naprawdę bezcenną wiedzę. Wiemy, jakie są koszty związane z niewykryciem błędów oraz jak kształtuje się rozkład błędu. W dalszym kroku z pomocą analityków systemowych i biznesowych mogliśmy określić obszary systemu, funkcjonalności, komponenty,

które z jednej strony obarczone są wysokim ryzykiem, a z drugiej strony przynoszą największe korzyści dla klienta lub są najintensywniej użytkowane. Rezultatem Etapu 1. był nowy, zoptymalizowany zestaw testów regresywnych, dobrany na podstawie zmiennej granulacji przypadków testowych, zależną od ryzyka (ang. Risk-Based Testing) oraz korzyści klienta (ang. Benefit-Driven Testing). Testowanie na podstawie zmiennej granulacji przypadków testów przedstawiono na rysunku 6.

Wyobraźmy sobie, że krzywa reprezentuje testowany system. W miejscach, gdzie amplituda jest wyższa, istnieje wyższe ryzyko skojarzone z testowanym systemem. W obszarach, gdzie amplituda jest niższa, poziom ryzyka jest niski. W obszarach (funkcjonalności, podsystemy) o wysokim ryzyku system testowany jest intensywniej – większa granulacja przypadków testowych. W obszarach o niskim ryzyku granulacja przypadków testowych jest niższa, a system testowany jest z mniejszą intensywnością. Granulacja przypadków testowych może być wyrażona za pomocą liczby przypadków testowych przypadających na pojedyncze wymaganie, liczby przypadków testowych na podsystem, funkcjonalność lub innej dogodnej miary oddającej pokrycie systemu testami.

Odkrycia

Poza nowym zbiorem testów regresywnych wynikiem analizy przeprowadzonej w ramach Etapu 1. były następujące odkrycia:

- Skuteczność wykrywania błędów wynosiła zaledwie 25%.
- Te 25% defektów było wykryte przez zaledwie 6% z 1000 przypadków testowych.
- Pozostałe 94% z 1000 przypadków testowych nigdy nie wykryło żadnego błędu, co nie oznacza, że tych błędów

tam nie było.
- Czas potrzebny na wykonanie pojedynczej kampanii regresji wynosił ponad 31 h.
- 40% tego czasu zajmowały testy tylko jednego podsystemu.
- W tym czasie znaleziono tylko 7% błędów dotyczącego tego podsystemu.
- Regresja wykrywała błędy tylko w 4 z kilkunastu podsystemów.
- Brak kontroli wersji testów regresywnych (brak powtarzalności procesu).
- Część funkcjonalności w ogóle nie była testowana.
- Część funkcjonalności interfejsu posiadała fałszywą implementację lub w ogóle jej nie miała.

Faza 2. – Optymalizacja techniczna

Etap 1. polegał głównie na wykonaniu pracy analitycznej, której rezultatem był nowy, zoptymalizowany zestaw testów regresywnych oraz dopisanie brakujących przypadków testowych. Z kolei Etap 2. skupiał się na optymalizacji technicznej zestawu automatycznych testów regresywnym. W ramach etapu drugiego zautomatyzowano brakujące przypadki testowe. Zoptymalizowano istniejące skrypty testowe celem skrócenia czasu wykonania i podniesienia niezawodności skryptów (automatycznych testów). Dodatkowo wprowadzono usprawnienia narzędzi testowych. Usprawnienia dotyczyły głównie silnika uruchamiającego testy. W końcowej fazie etapu drugiego wprowadzono zrównoleglenie wykonania testów automatycznych. Jeden zbiór testów był uruchamiany równolegle na trzech lustrzanych środowiskach testowych. Dzięki temu zabiegowi uzyskano dalsze skrócenie czasu wykonania kampanii testów regresywnych.

Faza 3. – Monitorowanie i dostrajanie (minimalizacja ryzyka)

W tym momencie posiadaliśmy dwa zestawy testów regresywnych. Stary, który ze względu na długi czas wykonania był uruchamiany podczas weekendu. Nowy, który był uruchamiany codziennie między godziną 18:00 a 6:00 rano. W trakcie optymalizacji testów regresywnych istniało ryzyko, że coś mogło zostać przeoczone bądź w wyniku błędnych wniosków z fazy analizy pozbyto się zbyt wielu przypadków testowych. Celem zapobieżenia ryzyku niewykrycia błędów przez nowe, zoptymalizowane testy zapobiegawczo porównywano wyniki starych i nowych testów regresywnych. Dodatkowo, aby zbadać skuteczność wykrywania błędów, zastosowano technikę posiewu błędów (and. Defect Seeding).

Faza 4. – Start!

Upewniwszy się, że nowy zestaw testów regresywnych działa tak, jak planowano, mogliśmy ze spokojnym sumieniem wyłączyć stare testy regresywne i wdrożyć nowe, zoptymalizowane.

„Rozkład błędów nie dostarcza informacji na temat ważności błędów”

AUTOR

Adam Marciszewski

Inżynierią jakości oprogramowania zajmuje się od blisko 10 lat. Umiejętności dotyczące testowania oprogramowania, automatyzacji i zarządzania testami budował, uczestnicząc w licznych międzynarodowych projektach z sektora telekomunikacji. Budowanie jakości produktu oraz usprawnianie testowania są jego pasją.

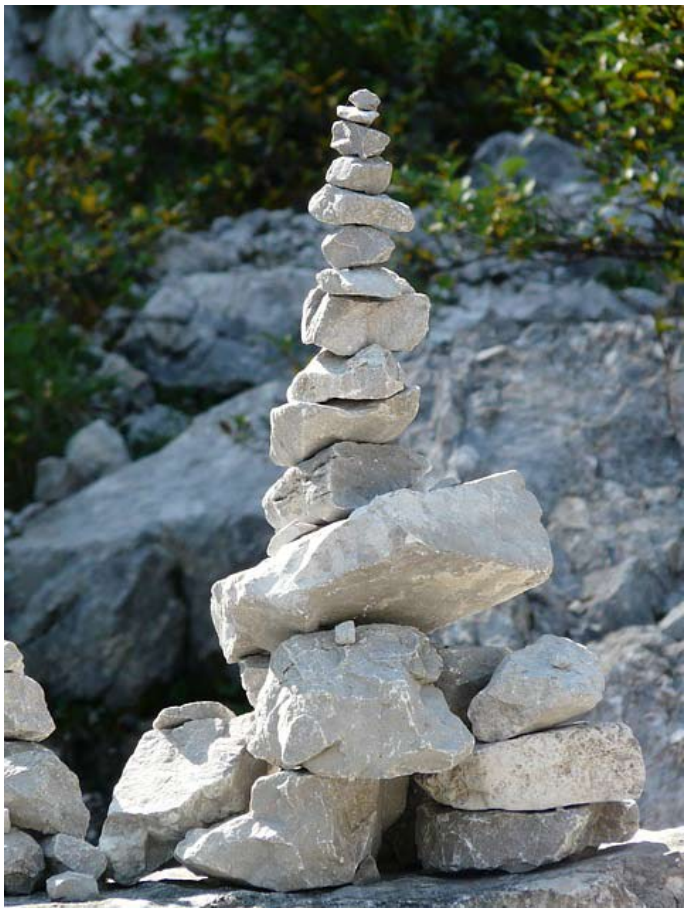
W trakcie pracy zawodowej pełnił funkcję inżyniera testów, kierownika testów, scrum mastera, a w ostatnim czasie product ownera dla frameworku do automatyzacji testów. Od niedawna interesuje się także testami użyteczności oraz testowaniem wymagań.

Po raz drugi postanowił wcielić się w rolę prelegenta na konferencji Testwarez.

Wynik optymalizacji

W wyniku przeprowadzenia optymalizacji automatycznych testów regresywnych uzyskano następujące rezultaty.

- Czas regresji po optymalizacji skrócił się do 8 h.
- Zrównoleglenie wykonania testów skróciło ten czas do 3 h.
- Wzrost skuteczności detekcji błędów przez testy automatyczne z 25% do 70%.
- Wzrost wydajności testowania – wyższe pokrycie przy krótszym czasie wykonania.
- Współczynnik pokrycia funkcjonalności testami wzrósł do 100%
- Krótszy czas dostarczania informacji zwrotnej.
- Więcej funkcjonalności testowana w krótszym czasie.
- Mniej przypadków testowych przy wyższym pokryciu systemu testami.



Marcin Kowalczyk

Słowa kluczowe jak góry lodowe – czyli rzecz o bibliotekach testowych

Wprowadzenie

Testowanie w oparciu o słowa kluczowe jest bardzo popularną techniką testowania, najczęściej wykorzystywaną przy automatyzacji. Słowo kluczowe reprezentuje akcję testową i jest interpretowane przez framework testowy. Jednym z najbardziej popularnych narzędzi na licencji open source korzystających z tej techniki jest Robot Framework.

Dlaczego słowa kluczowe są jak góry lodowe, po co tworzyć własne biblioteki

Każde słowo kluczowe ukrywa pewną funkcjonalność testową. Skojarzenie z górami lodowymi wynika z tego, że są one zanurzone w większości w wodzie, natomiast z lądu widoczny jest tylko ich wierzchołek. Analogicznie słowa kluczowe posiadają wierzchołek w postaci wywołania w platformie testowej, natomiast funkcjonalność jest niewidoczna. Rzecz w tym, że w bardzo prosty sposób możemy stworzyć własny „keyword" i samemu zdecydować, gdzie jest granica między wywołaniem a funkcjonalnością. Innymi słowy, możemy przenieść kroki testowe do biblioteki i ukryć je za wywołaniem, dzięki czemu składnia samego przypadku testowego jest bardziej przejrzysta, a funkcjonalność testowa gotowa do wykorzystania w innych testach.

Decydując się na tworzenie własnych bibliotek testowych, warto zastanowić się, co dostarcza framework testowy. Cztery główne cechy będące zaletami frameworków testowych opierających się o „keywordy" to:

- składnia testów opisana słowami kluczowymi;
- logowanie wyników i przebiegu testów;
- obsługa wyjątków rzucanych w bibliotekach testowych;
- możliwość łatwego dodawania własnych bibliotek.

Uważam, że poza niepodważalnymi zaletami frameworki mają również pewne wady. Można do nich zaliczyć:

- Mało przejrzysta i niezbyt efektywna składnia testów zbudowanych ze słów kluczowych, gdy mamy do czynienia ze złożonym przypadkiem testowym.
- „Programowanie w tabelkach" często zniechęca inżynierów testów (zwłaszcza tych z doświadczeniem programistycznym) do korzystania z narzędzia.

Korzystając z wybranego narzędzia, naszym celem powinno być wykorzystanie zalet i zminimalizowanie wad. Pomóc w tym może zaprojektowanie i stworzenie własnej biblioteki testowej z nowymi słowami kluczowymi, w której ciężar logiki testu będzie we właściwym dla naszego projektu miejscu.

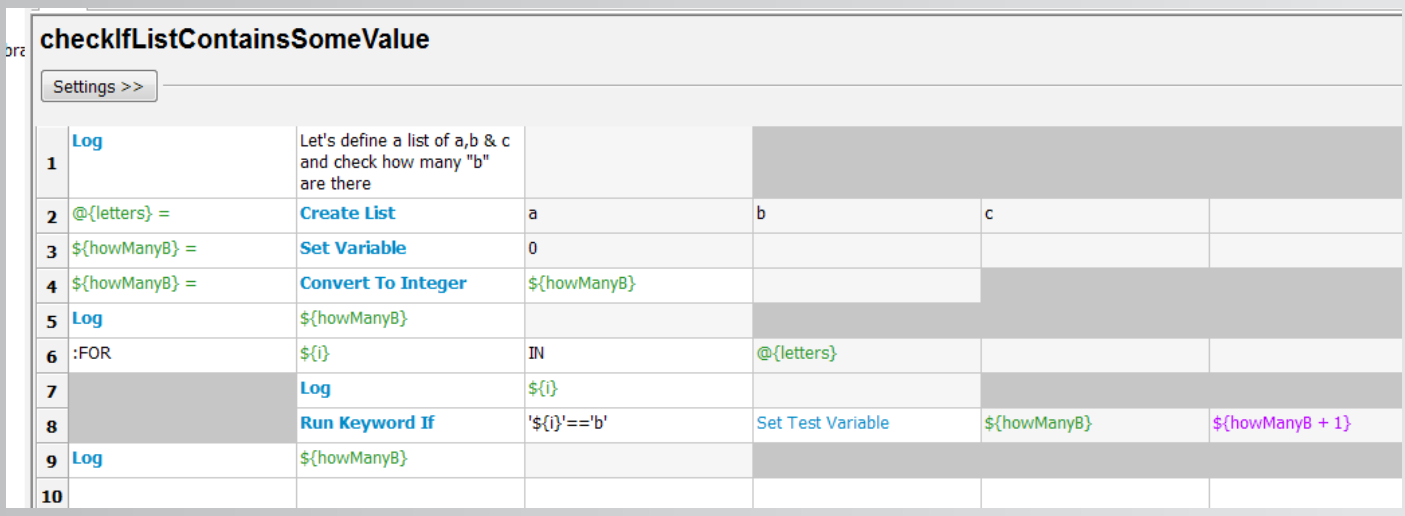
Rysunek 1. Przykład słów kluczowych – widok z RIDE (Robot Framework IDE)

1	${start} =	Set Variable	Hello world TESTWAREZ2014 !
2	Log	${start}	

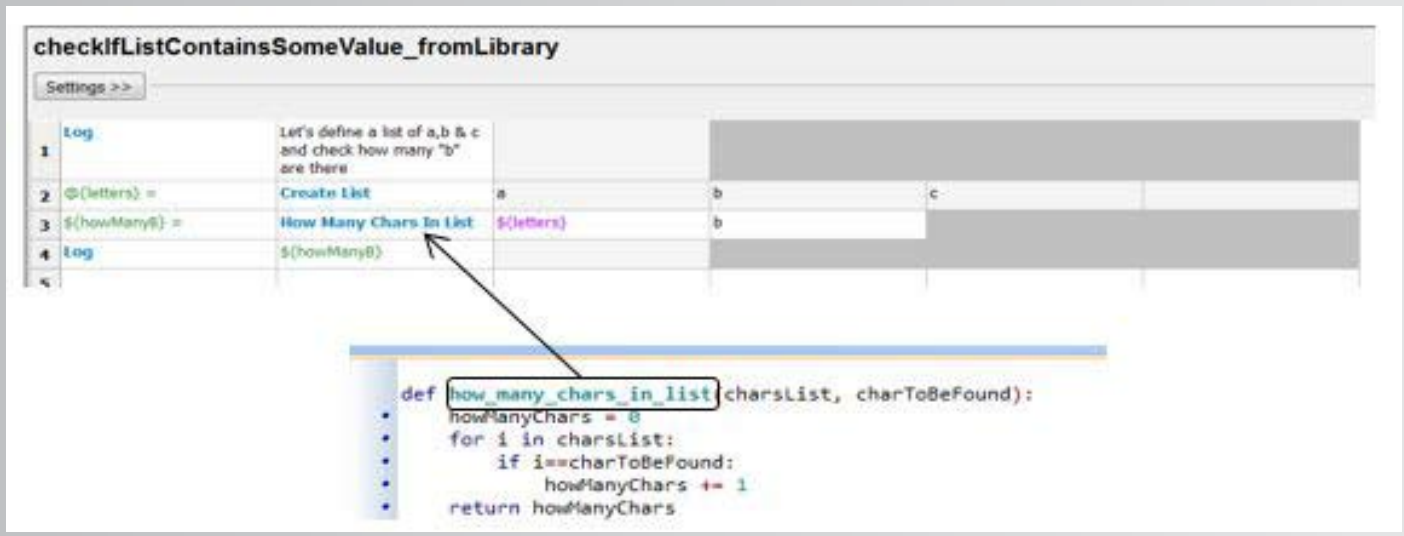
Rysunek 2. Gdzie ma być ciężar logiki testu? Są różne podejścia

Większość wbudowanych generycznych bibliotek testowych często dostarcza funkcjonalność, w której ciężar logiki testu leży po stronie słów kluczowych. W związku z tym, aby usprawnić programowanie testu, możemy napisać własne słowo kluczowe, w którym więcej funkcjonalności testowej będzie ukryte za słowem kluczowym. Poniższe rysunki przedstawiają abstrakcyjny przykład, w którym szukamy liczby wystąpień danej litery w zdaniu. W łatwy sposób możemy stworzyć własne słowo kluczowe, dzięki czemu 5 słów kluczowych zostanie przeniesionych do biblioteki, dzięki czemu zyskamy na przejrzystości, a słowo kluczowe będzie mogło być ponownie wykorzystane.

Rysunek 3. Przykład klasycznego podejścia: szukanie liczby wystąpień litery „b” w liście zawierającej litery – 9 linii



Rysunek 4. Przykład: po stworzeniu własnego słowa kluczowego 5 linii zostało przeniesionych do biblioteki

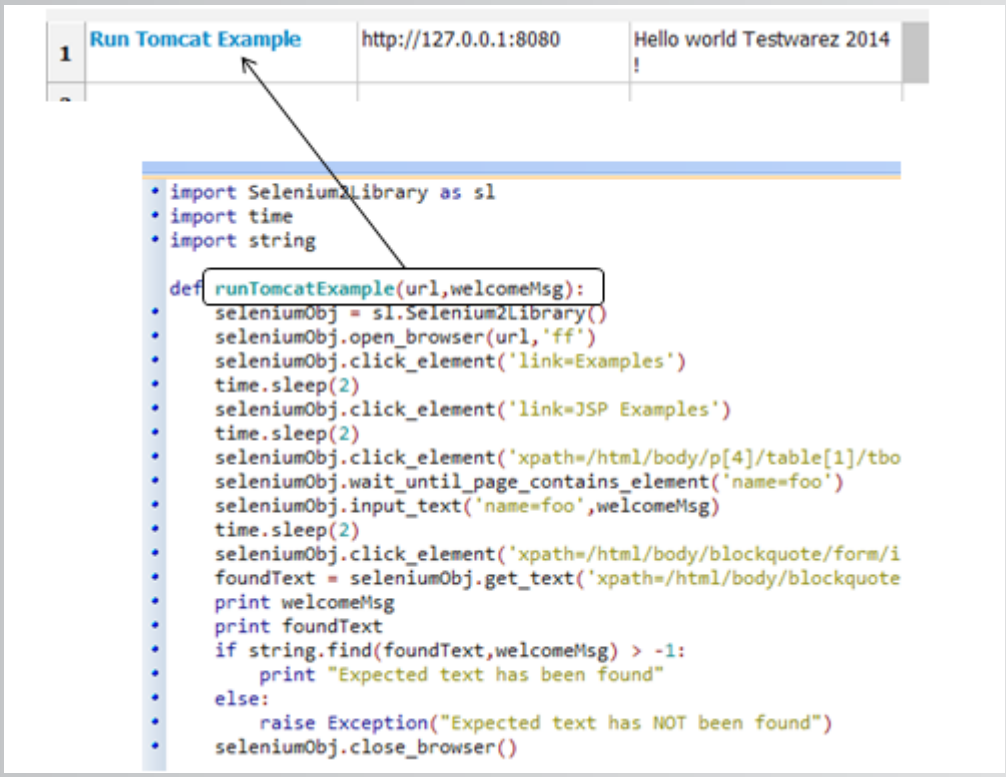


Kolejny przykład, tym razem z życia wzięty, to tworzenie grupy wywołań Selenium wykorzystywanych w czasie testów stron www. Uruchomienie przykładowych skryptów tomcatowych to 16 linijek, używając Robotowej biblioteki Selenium2Library. Całość można ukryć za jednym „keywordem", tworząc bibliotekę korzystającą z Selenium2Library bądź bezpośrednio Selenium WebDriver.

Rysunek 5. Przykład: „klikanie" po domyślnej stronie Tomcata w celu uruchomienia jednego z przykładowych skryptów – 16 linii

1	${welcomeMsg} =	Set Variable	Hello world Testwarez 2014 !
2	Open Browser	http://127.0.0.1:8080	browser=ff
3	Sleep	2	
4	Click Element	link=Examples	
5	Sleep	2	
6	Click Element	link=JSP Examples	
7	Sleep	2	
8	Click Element	xpath=/html/body/p[4]/table	
9	Sleep	2	
10	Wait Until Page Contains Element	name=foo	timeout=10
11	Input Text	name=foo	text=${welcomeMsg}
12	Sleep	2	
13	Click Element	xpath=/html/body/blockquote	
14	${dispMsg}=	Get Text	xpath=/html/body/blockquote
15	Should Contain	${dispMsg}	${welcomeMsg}
16	Close Browser		

Rysunek 6. Przykład: „klikanie" po domyślnej stronie Tomcata w celu uruchomienia jednego z przykładowych skryptów – nowe słowo kluczowe wykorzystujące Selenium2Library



Poza optymalizacją automatycznych testów istnieją również bardziej oczywiste powody, dla których moglibyśmy chcieć napisać własne biblioteki testowe. Po pierwsze, potrzebna nam funkcjonalność testowa nie została jeszcze przez nikogo napisana, więc musimy stworzyć ją samemu. Po drugie, potrzebna nam funkcjonalność może być specyficzna dla naszego środowiska, w związku z czym trudno oczekiwać, że jest ogólnodostępna.

Praktyczne aspekty tworzenia własnych bibliotek testowych

Czym właściwie jest biblioteka testowa w kontekście „keyword driven testing"? Jest to po prostu zbiór funkcji w wybranym języku używanych przez framework testowy. W czasie pisania własnych „keywordów" najczęściej używanym wzorcem projektowym jest adapter (ang. wrapper). Przykład: Robotowa biblioteka Selenium2Library stanowi Robotowy adapter do Selenium WebDriver.

Biblioteki do Robot Frameworka można tworzyć w Pythonie lub Javie. Python jest najbardziej naturalną formą tworzenia pluginów, ponieważ samo narzędzie też jest

Rysunek 7. Przykład: „klikanie" po domyślnej stronie Tomcata w celu uruchomienia jednego z przykładowych skryptów – nowe słowo kluczowe wykorzystujące bezpośrednio Selenium WebDriver

1	Run Tomcat Example2	http://127.0.0.1:8080	Hello world Testwarez 2014 !

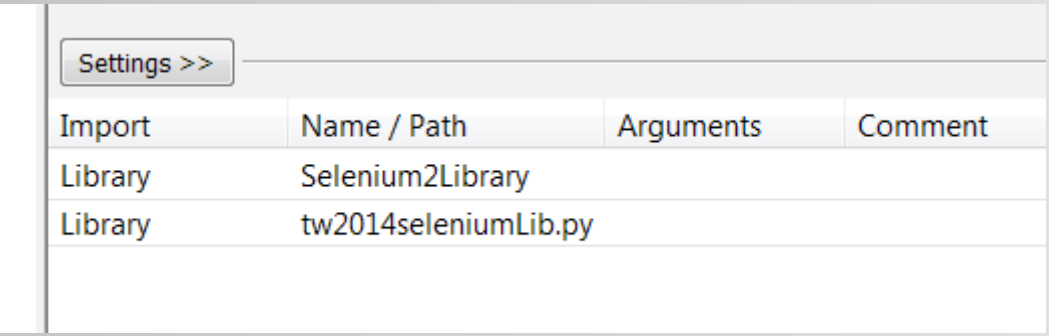
```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.common.keys import Keys
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC

def runTomcatExample2(url,welcomeMsg):
    browser = webdriver.Firefox()
    browser.get(url)
    time.sleep(2)
    browser.find_element_by_link_text('Examples').click()
    time.sleep(2)
    browser.find_element_by_link_text('JSP Examples').click()
    time.sleep(2)
    browser.find_element_by_xpath('/html/body/p[4]/table[1]/tbody
    element = WebDriverWait(browser, 10).until(EC.presence_of_ele
    time.sleep(2)
    for i in range(10):
        element.send_keys(Keys.BACKSPACE)
    element.send_keys(welcomeMsg)
    time.sleep(2)
    browser.find_element_by_xpath('/html/body/blockquote/form/inp
    foundText = browser.find_element_by_xpath('/html/body/blockqu
    time.sleep(2)
    print welcomeMsg
    print foundText
    if string.find(foundText,welcomeMsg) > -1:
        print "Expected text has been found"
    else:
        raise Exception("Expected text has NOT been found")
    browser.quit()
```

w tym języku napisane. Takie biblioteki mogą być uruchamiane zarówno z pyBota jak i jyBota. Tworzenie w Javie jest trochę bardziej skomplikowane i powinno być wykorzystywane wówczas, gdy jest wyraźny powód do użycia Javy. Takie biblioteki mogą być uruchamiane wyłącznie przez jyBota i wymagają instalacji Jythona.

Biblioteka to po prostu moduł Pythonowy (czyli plik .py). Aby ją uruchomić, musi się ona znajdować w tej samej lokalizacji, co skrypt testowy albo w dowolnej lokalizacji wskazanej przez PYTHONPATH (zmienna systemowa wskazująca, gdzie znajdują się moduły Pythonowe).

Rysunek 8. Przykład importu bibliotek w Robot Framework



Silnik narzędzia Robot Framework mapuje nazwy funkcji bibliotecznych na słowa kluczowe. „Keywordy" mogą przyjmować argumenty i zwracać wartości jak zwykłe funkcje. Poniższa tabela prezentuje sposób mapowania funkcji na słowa kluczowe:

Nazwa metody pythonowej	**Zamapowane słowo kluczowe**
jakas_nazwa_funkcji	Jakas Nazwa Funkcji
jakasNazwaFunkcji	Jakas Nazwa Funkcji
click_element	Click Element

Drukowanie do loga jest również niezwykle proste – framework przechwytuje standardowe wyjście. Aby ustawić test w stanie negatywnym, wystarczy rzucić dowolny wyjątek - zostanie obsłużony przez framework, a rezultat testu ustawiony na „fail". Poniższa tabela podsumowuje drukowanie i ustawianie negatywnego rezultatu:

Akcja	**Jak**	**Przykład**
Drukowanie do logu	Drukowanie na standardowe wyjście	print „cos drukujemy"
Ustawianie negatywnego rezultatu	Rzucenie dowolnego wyjątku	Raise Exception(„jakis komunikat")

Aby udokumentować bibliotekę testową, wystarczy użyć prostego parsera o nazwie Libdoc, dołączonego do Robot Framework. Skrypt dokumentuje ogólne informacje o bibliotece, nazwy słów kluczowych wraz z przyjmowanymi argumentami i opisem (rysunek 9).

Podsumowanie

Tworzenie własnych bibliotek testowych i słów kluczowych może niewątpliwie poprawić efektywność testowania techniką „keyword driven testing". Czasami rozszerzenie frameworka jest koniecznością, gdyż nie zawsze potrzebna funkcjonalność została opublikowana. Uważam, że warto testować w ten sposób i rozwijać swoje kompetencje w tym zakresie z powodów

Rysunek 9. Przykład dokumentacji słów kluczowych za pomocą narzędzia Libdoc

tw2014seleniumLib

Library scope: global
Named arguments: supported

Introduction

Przykladowa biblioteka testowa na potrzeby konferencji Testwarez 2014 !

Shortcuts

Run Tomcat Example · Run Tomcat Example 2

Keywords

Keyword	Arguments	Documentation
Run Tomcat Example	*url, welcomeMsg*	Uruchamia przyklady z Tomcata uzywajac robotowej biblioteki Selenium2Library. Przyklad uzycia: Run Tomcat Example | http://127.0.0.1:8080 | Hello world Testwarez !
Run Tomcat Example 2	*url, welcomeMsg*	Uruchamia przyklady z Tomcata uzywajac biblioteki Selenium WebDriver. Przyklad uzycia: Run Tomcat Example2 | http://127.0.0.1:8080 | Hello world Testwarez again !

Altogether 2 keywords.
Generated by libdoc on 2014-08-22 12:24:08.

AUTOR

Marcin Kowalczyk

Inżynier jakości oprogramowania od ponad 9 lat. Specjalizuje się w projektowaniu, rozwoju i utrzymaniu automatycznych testów. Największe doświadczenie ma w dziedzinie oprogramowania telekomunikacyjnego.

Programista i entuzjasta języka Python. Od czasu do czasu publikuje funkcjonalności testowe na licencji open source. Jest autorem jednej z oficjalnych bibliotek do narzędzia Robot Framework.

Posiada certyfikat ISTQB Advanced TTA. Po 3 latach ponownie występuje jako prelegent na konferencji Testwarez.

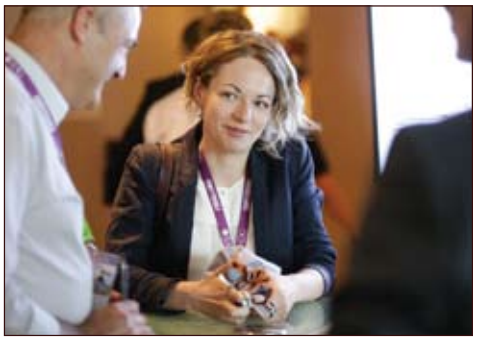
KONFERENCJA **COMPUTERWORLD**

V FORUM
JAKOŚCI SYSTEMÓW IT

już 18-19 czerwca 2015, Warszawa








ZAREZERWUJ TERMIN!

W przypadku zainteresowania prosimy o kontakt:

Agata Goździk
Współpraca z Partnerami
Agata_Gozdzik@idg.com.pl
Tel. +48 22 321 78 30

Aleksandra Zygarska
Program konferencji
Aleksandra_Zygarska@idg.com.pl
Tel. +48 22 321 78 72

jakosc.computerworld.pl

Michał Chudy

Testowanie Mutacyjne? Proste!

Wprowadzenie

Kiedy po raz pierwszy usłyszałem o testowaniu mutacyjnym (przyznam, że przez przypadek), to zainteresowała mnie ta dość egzotyczna jak na IT nazwa. Sama nazwa kojarzy się bardziej z genetyką i biologią aniżeli z informatyką, ale jak się okazuje – jest bardzo trafna.

Zadaniem testowania mutacyjnego jest sprawdzenie jakości testów jednostkowych. Znajdując potencjalne błędy, zwiększamy jakość testów jednostkowych, a pośrednio także i jakość naszego oprogramowania. Myślę, że największą zaletą testowania mutacyjnego jest eliminacja potencjalnych błędów, które najczęściej atakują nas w czasie regresji. Regresja najczęściej pojawia się wtedy, kiedy modyfikujemy istniejący już kod lub dodajemy coś nowego. Testy jednostkowe chronią nas przed stratą funkcjonalności lub zmienionym zachowaniem aplikacji, ale jak się okazuje – nie zawsze sobie z tym radzą. Nikt za regresją nie przepada, a szczególnie chyba nasi klienci, no bo jak wytłumaczyć komuś, że coś działało i nagle przestało…

Hej, ale ja nie jestem developerem! Testy jednostkowe, mutacyjne? To nie moja działka!

Nic bardziej mylnego. Developerzy, oczywiście, powinni dbać o jakość oprogramowania i jakość testów jednostkowych. Ale to właśnie tester jest adwokatem jakości w projekcie. I o ile odpowiedzialność za ja-

kość jest wspólna, to właśnie on podnosi czerwoną flagę i informuje zespół o tym, że istnieje problem. To on zajmuje się monitorowaniem jakości projektu – kiedy ta spada – kieruje na nią uwagę zespołu. Bo o jakości trzeba pamiętać, a z tym jest różnie, prawda? Przekonani? Zapraszam dalej, postaram się wyjaśnić, jak moim zdaniem powinny wyglądać role developera i testera w testowaniu mutacyjnym.

Dla przypomnienia: testy jednostkowe

Testy jednostkowe mają za zadanie przetestować działanie samego kodu. Każdy test testuje jakieś określone działanie pojedynczych fragmentów kodu, najczęściej po prostu metod. Wynik lub zachowanie testowanej metody porównywane jest z oczekiwanym wynikiem lub zachowaniem. Sam kod testów nie jest częścią oprogramowania i najczęściej leży równolegle do niego. Przyjrzyjmy się zatem przykładowi metody z biblioteki Apache Commons Lang (listing 1).

Opisana metoda ma za zadanie tylko sprawdzić, czy zmienna podana na wejściu ma wartość true. Taka swoista konwersja logiki trójwartościowej na binarną.

A oto test jednostkowy, który sprawdza wszystkie trzy możliwe wejścia dla omawianej metody i porównuje otrzymane wyniki z wartością oczekiwaną (lListing 2). W przypadku, kiedy wartość oczekiwana różni się od otrzymanej – asercja rzuca wyjątek i test kończy się niepowodzeniem. Niewątpliwą zaletą testów jednostkowych jest to, że można je uruchamiać podczas

Listing 1. Fragment kodu z klasy BooleanUtils z biblioteki Apache Commons Lang

```
public static boolean isTrue(final Boolean bool) {
     return Boolean.TRUE.equals(bool);
 }
```

Listing 2. Test jednostkowy z klasy BooleanUtilsTest z biblioteki Apache Commons Lang

```
@Test
public void test_isTrue_Boolean() {
    assertTrue(BooleanUtils.isTrue(Boolean.TRUE));
    assertFalse(BooleanUtils.isTrue(Boolean.FALSE));
    assertFalse(BooleanUtils.isTrue((Boolean) null));
}
```

każdego budowania projektu, są szybkie i automatyczne. Zwiększają odporność kodu na regresję, a jednocześnie są całkiem niezłą jego dokumentacją. Ale skąd mamy pewność, że nasz kod jest dobrze pokryty testami jednostkowymi?

Dla przypomnienia: pokrycie kodu

Do sprawdzenia, czy nasz kod jest dobrze pokryty testami, używamy metryki zwanej pokryciem kodu.

Metrykę tę wyraża się w procentach i wyróżniamy kilka jej odmian, m.in.:

- pokrycie linii kodu (Statement Coverage) – sprawdzamy, które linie kodu zostały odwiedzone/wykonane podczas wykonywania wszystkich testów jednostkowych;
- pokrycie metod (Function Coverage) – jak powyżej, z tą różnicą, że zmniejszamy nieco granularność.

Przyjrzyjmy się następującej klasie i jej testom jednostkowym (listing 3).

Listing 3. Przykładowa testowana klasa

```
public class Example {

    private int number;

    public Example(int number) {
        this.number = number;
    }

    public boolean isItEven() {
      return number % 2 == 0;
    }

    public int makeItTwice() {
        return number * 2;
    }
}
```

Listing 4. Testy jednostkowe dla klasy z listingu 3

```
public class ExampleTest {

    @Test
    public void testIsItEven() throws Exception {
        Example example = new Example(0);
        assertTrue(example.isItEven());
    }

    @Test
    public void testIsNotEven() throws Exception {
        Example example = new Example(3);
        assertFalse(example.isItEven());
    }
}
```

Linie oznaczone zielonym kolorem to linie odwiedzone podczas wykonywania testów. Linie czerwone, analogicznie, to linie, które nigdy nie zostały wykonane podczas uruchamiania testów. Łatwo możemy policzyć, że pokrycie linii kodu dla powyższej klasy wynosi 80%, a pokrycie metod 66%. To całkiem dobre wyniki.

Spróbujmy zatem dopiąć do pełnych 100%, ale tym razem nie wysilając się zbytnio, dodamy następujący test (listing 5).

Jak widać, nasz test nie robi nic poza wywołaniem niepokrytej wcześniej metody. Nawet gdyby usunąć asercję z tego testu, to nadal nasze nowe pokrycie pięknie się zazieleni. Dobiliśmy do 100% pokrycia zarówno linii, jak i metod. Jaki z tego wniosek? Code Coverage da się oszukać i to wcale nie jest takie trudne. Tylko po co oszukiwać? W niektórych projektach bardzo trudno jest osiągnąć 100% pokrycia. Zdarza się, na szczęście już rzadko, że programiści są nagradzani za utrzymy-

Listing 5. Dodatkowy test, który nie testuje klasy z listingu 3, ale uruchamia jej metodę

```
    @Test
    public void phonyTest() throws Exception {
        new Example(1).makeItTwice();
        assertTrue(true);
    }
```

Listing 6. Po dopisaniu nowego testu, który nic nie sprawdza, mam 100% pokrycia

```
public class Example {

    private int number;

    public Example(int number) {
        this.number = number;
    }

     public boolean isItEven() {
       return number % 2 == 0;
     }

     public int makeItTwice() {
         return number * 2;
     }
}
```

wanie wysokiego pokrycia. Czasem nasz klient od nas tego wymaga. Co zrobić, jak żyć?

Wreszcie, testy mutacyjne

Lekiem na powyższe bolączki mogą być właśnie testy mutacyjne! Na czym to polega? Zasada jest prosta. Do kodu naszej aplikacji wprowadzane są tzw. mutacje/mutanty. Mutacje mogą być różne i zależą od tego, co w danej linii kodu się znajduje. Jeśli jest to operacja matematyczna, to np. operator może zostać zmieniony z – na +. Jeśli jest to warunek *if,* to operator porównania może zostać zmieniony z > na >= lub nawet całe wyrażenie może zostać zastąpione wartością *true.* Jeśli jest to pętla, to zmienione zostaną jej warunki brzegowe itd. Pojedyncza linia może być mutowana kolejno wiele razy, ale zarówno liczba, jak i typ mutacji są zawsze deterministyczne.

Co ważne, po każdorazowym wprowadzeniu pojedynczej mutacji uruchamiane są tylko te testy jednostkowe, które podczas wykonywania odwiedzają zmutowaną linię kodu.

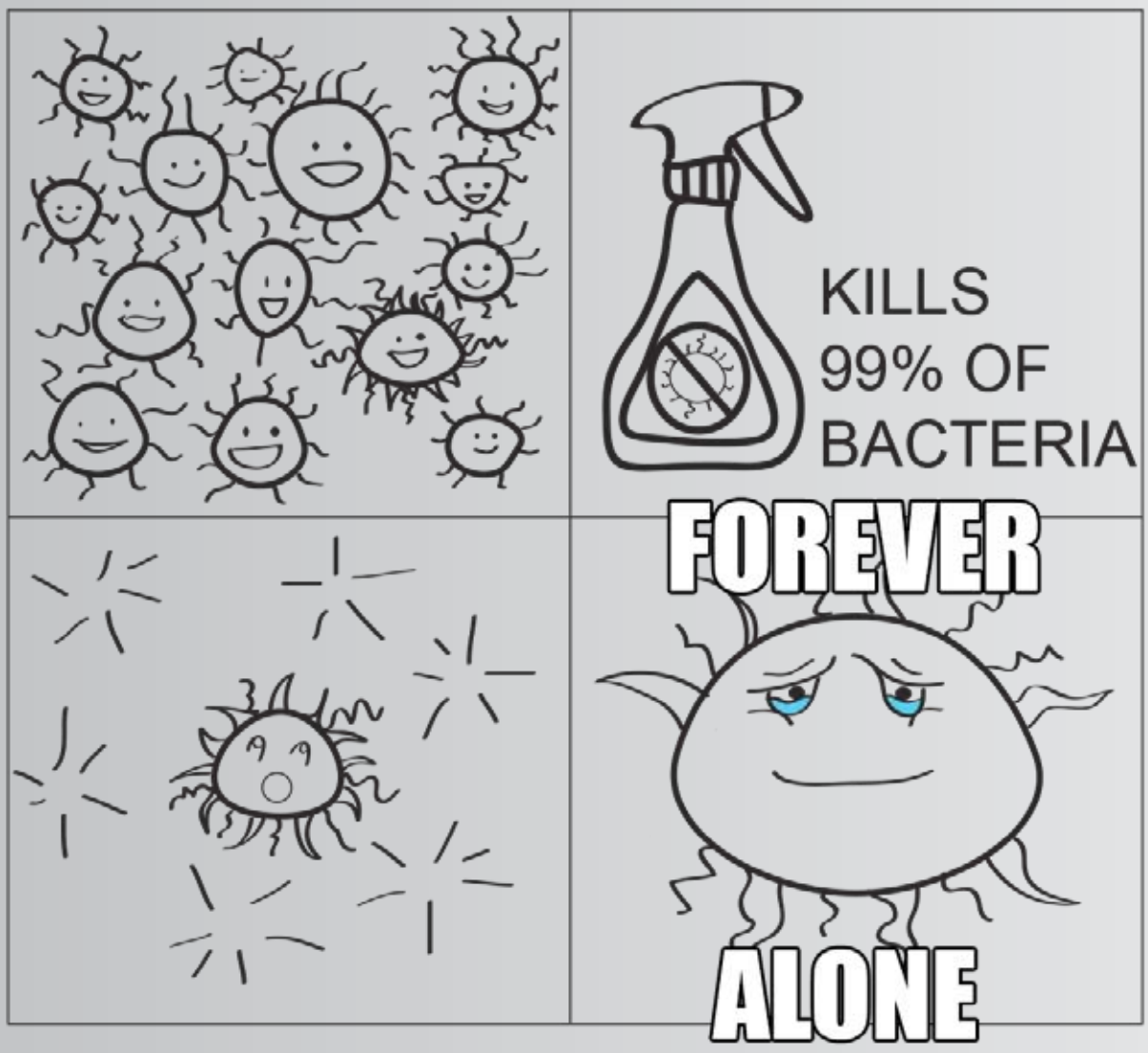
Jeśli po wprowadzeniu mutanta co najmniej jeden z naszych testów jednostkowych kończy się niepowodzeniem, to znaczy, że mutacja została wykryta!

Jeśli po wprowadzeniu mutanta wszystkie testy jednostkowe nadal przechodzą poprawnie, to znaczy, że mutacja nie została wykryta.

Innymi słowy, nasze testy jednostkowe uruchamiane są na automatycznie modyfi-

Rysunek 1. Mutant został zabity!

Rysunek 2. Mutant przeżył....



kowanym kodzie naszej aplikacji. Jeśli kod naszej aplikacji się zmienił, rezultat wykonania również powinien się zmienić, a nasze testy jednostkowe powinny to wykryć. Jeśli tak się nie dzieje, może to świadczyć o tym, że nasze testy wymagają nieco więcej pracy.

Dlaczego to działa? Przecież są to tylko niewielkie pomyłki, które nijak się mają do często złożonych błędów regresyjnych, przypadków brzegowych, o których nikt by nie pomyślał. Otóż zauważono, że złożone błędy są powiązane z tymi prostymi – najczęściej się z nich składają. I to w taki sposób, że zestaw testów wykrywających wszystkie proste błędy poradzi sobie również z tymi najbardziej skomplikowanymi. Skuteczność tej metody zwiększa fakt, że wygenerowane mutanty są podobne do prawdziwych popełnianych programistycznych pomyłek.

Wróćmy zatem do naszego przykładu. Wyobraźmy sobie, że uruchomiliśmy testy mutacyjne na naszej klasie Example. Dodatkowo w każdej linii zostały wprowadzone mutacje. Pamiętajmy, że mutacje wprowadzane są pojedynczo, a następnie dla każdego mutanta uruchamiane są testy, ale na potrzeby artykułu wprowadziłem do listingu wszystkie 3 mutacje naraz.

Mutacja 1. polega na usunięciu linii kodu. Mutant bez problemu zostanie wykryty przez test testIsNotEven. Oczekujemy re-

Listing 7. Przykłady mutacji w naszej klasie z listingu 3

```
public class Example {

    private int number;

    public Example(int number) {
        // Mutation 1 - remove this line this.number
= number;
    }

    public boolean isItEven() {
      return number * 2 == 0; // Mutation 2 change %
to /
    }

    public int makeItTwice() {
        return null; // Mutation 3 always return
null
    }
}
```

zultatu dla liczby nieparzystej 3, a zostanie zwrócony rezultat dla parzystego 0, ponieważ usunięta została linia przypisania. Test zakończy się niepowodzeniem - mutant zabity!

Mutacja 2. nie zostanie wykryta, ponieważ zmiana modulo na znak mnożenia dla naszych danych testowych nie zmieni rezultatu funkcji. 3 * 2 nadal będzie różne od 0, a 0 * 2 nadal równe 0. Mutant przeżył!

Mutacja. 3 również nie zostanie wykryta, ponieważ nasza metoda testująca jest naszym małym oszustwem. Kolejny mutant przeżył.

Z nawet tak prostego przykładu widać, że testy mutacyjne wykrywają nie tylko jawne oszustwa! Nawet jeśli wydaje nam się, że nasze testy dobrze pokrywają testowane metody, to może się okazać, że jednak w niektórych przypadkach nie są w stanie wykryć tak znaczącej modyfikacji w kodzie jak zmiana znaku arytmetycznego. Powyższy kod został oczywiście spreparowany dla przykładu. Ale to nic, szybko po uruchomieniu testów na własnym kodzie można się przekonać, że to dość powszechny scenariusz. W naszym przypadku wykryliśmy tylko 1 mutanta z 3. Kill ratio = 33%.

Jakość testów jednostkowych może być zatem wyrażona odsetkiem zabitych mutacji! I ten odsetek nazywamy metryką Mutation Coverage. Jest to nic innego jak stosunek ilości mutacji, na które nasze testy są odporne, do wszystkich możliwych mutacji w kodzie. Oczywiście z zamkniętego skończonego zbioru możliwych mutatorów.

Ok, czemu to tak mało popularne? To musi być coś nowego… nic bardziej mylnego…

Odrobina historii

Idea testowania mutacyjnego została zaproponowana już w 1971 roku przez ówczesnego studenta, Richarda Liptona. Pierwszą implementacją było narzędzie służące do testowania testów napisanych w języku Fortran IV. Nazywało się ono PIMS i zostało napisane w 1978 roku. Kolejny przełom pojawił się pod koniec lat 80. – narzędzie nazywało się Mothra i nie było zbyt przyjazne w obsłudze. Wymagało wielu manualnych kroków, zanim można było wygenerować raport. Nadal można je znaleźć w sieci! Przez wiele lat narzędzia do testowania mutacyjnego nie wychodziły poza środowisko akademickie – były tematem badań i rozważań. Były niedoskonałe i powolne. Dopiero pojawienie się języków wysokopoziomowych, rozwój sprzętu i drastyczna zmiana mocy obliczeniowej komputerów pozwoliły na powstanie następnej generacji narzędzi.

Narzędzia

Narzędzi do testowania mutacyjnego jest całkiem sporo, natomiast niewiele z nich jest aktywnie rozwijanych i wspieranych. Mutacyjnie potestujemy kod napisany w Javie, C#, Ruby, Pythonie i wielu innych. Znajdziemy wśród nich narzędzia zarówno open source, jak i takie, do których kodu źródłowego dostęp mają tylko uczelniani badacze (μJava). Znajdą się też produkty komercyjne (Insure++). Sytuacja przypomina trochę tę, jaka miała miejsce 10 lat

temu wśród frameworków do testów jednostkowych. Wszystko jest przed nami!

Skupimy się na narzędziach przeznaczonych dla Javy – na tym podwórku konkurencja wydaje się największa. Tutaj również, jak się okaże, mamy wiele akcentów rodzimych – np. Judy – narzędzie do testowania mutacyjnego w Javie napisane na Uniwersytecie Wrocławskim. Ostatnia wersja Judy (2.1.0) została wydana na początku 2014 roku, co jest nie lada nowością w świecie testowania mutacyjnego!

Zajmiemy się jednak narzędziem PIT, którego wersja 1.0.0 ujrzała światło dzienne 18 maja 2014 roku. Jak sam autor przekonuje, PIT jest narzędziem z prawdziwego zdarzenia. Jest szybki, łatwy w użyciu, rozwijany i aktywnie wspierany. Alternatywy są zazwyczaj wolne, trudne w użyciu i pisane raczej pod potrzeby akademickie aniżeli pod potrzeby prawdziwych zespołów developerskich.

Czym się zatem charakteryzuje? PIT wprowadza mutacje na poziomie byte code – oznacza to, że nie potrzebuje po każdej zmianie kompilować kodu, co znacznie przyspiesza proces testowania. Oznacza to również, że nasze pliki źródłowe nigdy nie są zmieniane, więc nie ma obaw, aby jakiekolwiek zmiany w nich pozostały.

Najbardziej istotną moim zdaniem zaletą PIT-a jest cała masa wsparcia, która wytworzyła się wokół niego na rynku. Wśród narzędzi automatyzujących budowę projektów PIT doczekał się już swoich wtyczek do Anta, Mavena oraz Gradle'a. Ten ostatni został napisany przez Marcina Zajączkowskiego – to kolejny, ale nie ostatni, polski akcent w świecie testowania mutacyjnego. Bo PIT-a możemy równie dobrze uruchomić z poziomu naszego ulubionego IDE. Plugin do IntelliJ IDEA zawdzięczamy Michałowi Jedynakowi. Pozwala on z poziomu IDE uruchomić testy i obejrzeć raport jeszcze przed wysłaniem kodu do repozytorium. Bez obaw, plugin do Eclipse'a również istnieje!

Raport 1. Fragment raportu PIT-a pokazujący kod mutowanej klasy. Dwie mutacje przeżyły testy

```
46          public String featureId(String name) {
47    1         return featureId = id(name);
48          }
49
50          public String featureElementId(String name) {
51    1         return featureElementId = featureId + ":" + id(name);
52          }
```

Raport 2. Przykładowy raport z podziałem na pakiety dla Apache Commons Lang

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage	
107	94%	11959/12690	86%	9783/11369

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage	
org.apache.commons.lang3	26	97%	4995/5140	90%	4859/5377
org.apache.commons.lang3.builder	12	88%	1514/1714	83%	1131/1357
org.apache.commons.lang3.concurrent	10	99%	303/307	97%	161/166
org.apache.commons.lang3.event	2	97%	73/75	96%	22/23
org.apache.commons.lang3.exception	4	94%	238/253	81%	123/151
org.apache.commons.lang3.math	3	98%	701/712	85%	646/757
org.apache.commons.lang3.mutable	8	100%	289/289	99%	162/163
org.apache.commons.lang3.reflect	7	78%	696/889	68%	487/716
org.apache.commons.lang3.text	9	96%	1600/1673	81%	1192/1469
org.apache.commons.lang3.text.translate	12	96%	217/225	85%	200/234
org.apache.commons.lang3.time	8	94%	1258/1337	83%	755/905
org.apache.commons.lang3.tuple	6	99%	75/76	88%	45/51

Report generated by PIT 1.0.0

Uruchomienie PIT-a domyślnie generuje raport w postaci HTML. W raporcie znajdziemy metryki, takie jak tradycyjne Code Coverage oraz Mutation Coverage – dla całego projektu, w rozbiciu na poszczególne pakiety, a nawet poszczególne klasy. Nawigując po raporcie, możemy obejrzeć wyniki dla każdej klasy z osobna. Oprócz metryki wyświetlany jest kod klasy, a każda znacząca linia oznaczona jest dwoma z 4 kolorów. Jasny zielony – linia odwiedzona. Jasny czerwony – linia nieodwiedzona. Oraz drugim kolorem, który implikuje, jak nasze testy zareagowały na mutację. Ciemny zielony – mutant zabity. Ciemny czerwony – mutant przeżył.

Ile czasu zajmuje testowanie mutacyjne? Jest szybkie, przynajmniej w stosunku do rozwiązań sprzed kilku lat. Na warsztat wziąłem wspomnianą wcześniej bibliotekę Apache Commons Lang 3. Składa się ona ze 132 klas oraz 2559 testów jednostkowych, także jest to całkiem dobrze przetestowana biblioteka. Projekt buduje się około 30 sekund wraz z uruchomieniem

wszystkich testów jednostkowych. Uruchomienie analizy testów mutacyjnych wygenerowało 11369 mutacji, dla których zostało uruchomione 19650 testów jednostkowych (1,73 testu/mutację, dla każdej mutacji tylko testy, które powodują odwiedzenie mutowanej linii). Cała analiza, uruchomiona na 4 wątkach, trwała ok. 7 minut. Moim zdaniem to naprawdę niewiele. Przy okazji, kill ratio wyszło na poziomie 86%. Nieźle.

Konfiguracja – Maven

Zacznijmy od najpopularniejszej i chyba najprostszej zarazem konfiguracji PIT-a. Sprowadza się ona do prostego „kopiuj i wklej" do mavenowego poma i modyfikacji dwóch ważnych parametrów: targetClasses oraz targetTests. Są to pakiety, w których znajduje się odpowiednio nasz kod oraz nasze testy jednostkowe. Należy przypomnieć, że testy mutacyjne nie nadają się do sprawdzania testów funkcjonalnych, dlatego pamiętajmy o tym podczas konfiguracji. Bo jeśli w projekcie znajdują się testy funkcjonalne operujące np. na bazach danych, to nasze mutacje mogą przy tym trochę narozrabiać...

Listing 8. Plugin basic configuration for maven project

```
<build>
    <plugins>
        <plugin>
            <groupId>org.pitest</groupId>
            <artifactId>pitest-maven</artifactId>
            <version>1.0.0</version>
            <configuration>
                <targetClasses>
                    <param>cucumber*</param>
                </targetClasses>
                <targetTests>
                    <param>cucumber*</param>
                </targetTests>
             </configuration>
        </plugin>
        (...)
    </plugins>
    (...)
</build>
```

Powyższa konfiguracja w większości wypadków jest wystarczająca. Natomiast lista możliwości i przełączników jest długa. Do zapoznania się z nimi zapraszam do oficjalnej dokumentacji.

Mając tak skonfigurowany projekt, możemy go skompilować i jednocześnie uruchomić testy za pomocą linii poleceń:

```
mvn clean install org.pitest:pitest
-maven:mutationCoverage
```

Wystarczy chwilę poczekać i voila – raport HTML z uruchomienia znajdziemy w target/pit-reports/YYYYMMDDHHMI. Teraz pozostaje tylko zapoznać się z wynikami i załatać nasze testy jednostkowe. W wynikach od czasu do czasu znajdziemy tzw. false positive, czyli błędy, które błędami nie są. Sam PIT stara się unikać takich sytuacji choćby przez wykluczenie z mutowania metod logujących. Nie zawsze jednak się to udaje. Pomóc może dodatkowa konfiguracja narzędzia lub pogodzenie się z faktem, że wszystkich mutantów nie zabijemy…

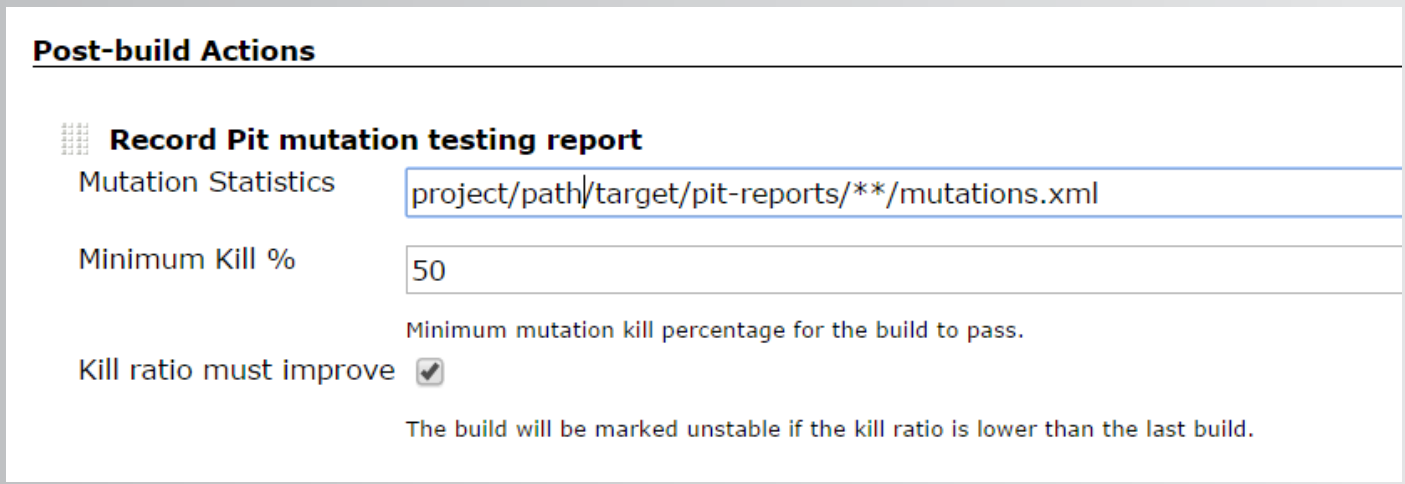
Konfiguracja – Jenkins

Czemu nie zautomatyzować naszych mutacji? Dobrym powodem jest gotowy plugin do integracji z Jenkinsem. Należy najpierw skonfigurować Jenkinsowego builda tak, aby budował projekt i uruchamiał PIT-a, a następnie dodać plugin jako akcję wykonywaną po zbudowaniu projektu, czyli po uruchomieniu testów mutacyjnych. Konfiguracja sprowadza się tylko do wskazania miejsca, gdzie generują się raporty. Dzięki takiemu ustawieniu dostajemy dodatkowy widok, który pozwala nam na przeglądanie ostatnich rezultatów i raportów PIT-a wprost z Jenkins'a.

To nie wszystko. Plugin pozwala nam na ustawienie progu zabitych mutacji, poniżej którego build zostanie uznany za nieudany. Drugą opcją kontroli jest wymuszenie, aby każdy następny build miał co najmniej tak dobre kill ratio jak ten poprzedni.

Nic nie stoi na przeszkodzie, aby skonfigurować build w taki sposób, aby uruchamiał się codziennie lub nawet po każdej zmianie

Rysunek 3. Przykładowa konfiguracja wtyczki do Jenkinsa



w repozytorium. Informowałby wtedy zainteresowanych o niepowodzeniu, umożliwiając stałą kontrolę nad jakością testów jednostkowych. Piękne.

Podsumowanie

Szczerze zachęcam wszystkich do spróbowania, to bardzo proste. Nawet jeśli w naszej bazie kodu nie mamy zbyt wiele testów jednostkowych – to nic, lepiej, żeby były dobrej jakości, skoro już tam są. Samo oglądanie wyników sprawia wiele frajdy i często można się wiele nauczyć, szczególnie pisania dobrych testów jednostkowych.

Pamiętajmy też, że metryka Code Coverage trochę nas oszukuje, usypia naszą czujność. Wysoka wartość tej metryki nie jest wcale równoważna z dobrze przetestowaną aplikacją, bo Code Coverage nie mówi nam o tym, co zostało przetestowane, ale tylko o tym, co definitywnie nie zostało przetestowane! Nie przywiązujmy zatem dużej wagi do metryki pokrycia kodu, pamiętajmy, że to metryki mają nam służyć, a nie my im. Jakie powinny być optymalne wartości pokrycia kodu, kill ratio? Sam chciałbym znać jedną odpowiedź, ale jej nie ma. To w dużej mierze zależy od specyfiki projektu. Na pewno utrzymywanie Mutation Coverage na wysokim poziomie powoduje, że metryka Code Coverage ma większą wartość. Jednak nie ma się co zabijać i dążyć do wartości bliskich 100%. Jest wiele innych metod testowania, które zapewniają jakość i poprawność oprogramowania na wielu różnych poziomach i płaszczyznach – powinniśmy używać rozsądnie wielu z nich.

Jaka powinna być rola testera i developera przy testowaniu mutacyjnym? Myślę, że każdy developer, kiedy dodaje nowe testy lub funkcjonalności albo modyfikuje już istniejący kod, powinien uruchomić mutacje u siebie lokalnie, aby sprawdzić, że testy są dobrze napisane. Wtedy może ze spokojem ducha wysłać swój kod do repozytorium.

W miejscu, gdzie aktualnie pracuję, przyjęło się, że tester nie tylko testuje oprogramowanie pod różnym kątem, ale też dba o to, aby jego jakość była na wysokim poziomie. Dlatego też, między innymi, co jakiś czas uruchamia Sonara na projektowym codebase – czyli oprogramowanie wykrywające potencjalne błędy programistyczne czy brzydkie zapachy w kodzie. (Przy okazji, PIT posiada również plugin do Sonara!) Po uruchomieniu sprawdza, jak wygląda dług techniczny i w przypadku zaniedbań informuje o tym zespół. Podnosi czerwoną flagę, jeśli w kodzie są błędy o statusie critical lub major. Dokładnie taką samą rolę powinien przyjąć w przypadku testów mutacyjnych. Powinien dbać, aby Mutation Coverage był na wysokim poziomie. Powinien zajrzeć, czy krytyczny dla aplikacji kod jest pokryty testami, którym można ufać. Bo mimo tego, że nie jest odpowiedzialny za pisanie testów jednostkowych, to może być już odpowiedzialny za stanie na straży ich jakości.

Testujcie! Zabijajcie mutanty!

AUTOR

Michał Chudy

Software Engineer z poznańskiej firmy Cognifide, absolwent Wydziału Informatyki Politechniki Poznańskiej. Od 3 lat pracuje z Adobe CQ/AEM w projektach dla globalnych marek.

Programista z przeznaczenia, zamiłowania i wyboru. Niepoprawny entuzjasta Java & OSGI i wielki fan czystego kodu. Kiedy musi poświęcić jakość kosztem czasu lub ceny, umiera w nim mała część.

Otwarty na nowe technologie, najchętniej zaszyłby się w dziale R&D i ciągle rozwijał. Uzależniony od IntelliJ, do którego jest nastawiony wprost ewangelicznie. Autor Mobiletu (w wersji na Androida) – najpopularniejszej aplikacji, umożliwiającej kupno biletów transportu miejskiego oraz biletów parkingowych w wielu polskich miastach.

Bogdan Bereza

Gimbaza testerów

Trochę historii

Kiedy w latach 2002-2003 prowadziłem pierwsze w Polsce szkolenia przygotowujące do egzaminów na certyfikat ISEB, poprzednika ISTQB, zainteresowanie było ogromne, a profil uczestników – nadzwyczaj zróżnicowany, mimo że wiedza wymagana do tego zupełnie podstawowego certyfikatu była zbyt płytka, zbyt powierzchowna dla wielu słuchaczy, mających w testowaniu spore czasem doświadczenie, jak i zbyt wąska dla osób pełniących funkcje kierowników projektów, albo analityków biznesowych; ograniczona tylko do małego w końcu zakresu wiedzy dotyczącej projektowania, wykonywania i automatyzacji testów. Mimo tego ci ludzie, zainteresowani zupełną nowością, jaką były wówczas te certyfikaty i szkolenia, chętnie brali w nich udział.

Od tego czasu na powierzchni nastąpiły ogromne zmiany. Dziesiątki tysięcy pracowników sektora IT definiuje swoją rolę jako „testerzy", certyfikaty ISTQB stoją na drugim miejscu w rankingu popularności po kierowniczej falandze certyfikacji PRINCE 2, PMI, IPMA oraz ITIL. We Wrocławiu, w Krakowie, Gdańsku, Łodzi, Poznaniu, Bydgoszczy i Warszawie powstały lokalne organizacje osób zajmujących się testowaniem, konferencja „Testwarez" bez trudu gromadzi co roku kilkaset uczestników, a turniej testerów „TestingCup" stał się prawdziwym przebojem w ciągu ledwo dwóch lat swego istnienia. Niemal każda firma szkoleniowa ma w swojej ofercie kursy testowania. Można śmiało powiedzieć, że nie tylko przestaliśmy być białą plamą na testowej mapie Europy, ale staliśmy się pod wieloma względami ważnym aktorem w tej branży. Ale…

W testerskich okopach

Wprawdzie testowanie lubi się nazywać „QA", Quality Assurance, ale z *zapewnieniem* jakości w swojej obecnej formie ma bardzo niewiele wspólnego. Owszem, testowanie, czyli kontrola jakości, może pośrednio spowodować wzrost jakości, ale często jest to najmniej skuteczna metoda. Warto podkreślić, że opisywane tutaj przeze mnie zjawiska nie są w żadnym razie polską specjalnością, nawet moda na mylące nazywanie testowania „QA" przyszła do nas z USA.

Większość uczestników szkoleń, konferencji i rozmaitych innych wydarzeń testowych to ludzie młodzi, na niezbyt wysokich stanowiskach. Popularne wśród nich tematyki to przede wszystkim narzędzia i automatyzacja, testy urządzeń mobilnych, sposoby zdobywania popularnych certyfikatów, trochę kwestii organizacyjnych, zwłaszcza w kontekście Agile, czyli… same sprawy niskopoziomowe, takie swoiste testerskie hakerstwo. Nic w tym złego, oczywiście, to sprawy przydatne, tylko mam pytanie, gdzie wobec tego dyskutowane są zagadnienia strategii testowej, jej roli w inżynierii jakości, w zarządzaniu, w udoskonalaniu procesów produkcji i, szerzej, procesów biznesowych? Czyżby w kręgach MBA, PMI, IIBA, PRINCE 2? Niestety, nie, tam mówi się o testowaniu/weryfikacji/walidacji same teflonowe banały: teflonowe, czyli niepodważalne, ale mało konkretne i mało przydatne.

Wprawdzie testowanie lubi się nazywać „QA" Quality Assurance, ale z zapewnieniem jakości w swojej obecnej formie ma bardzo niewiele wspólnego.

Tester do wszystkiego

Na swoje usprawiedliwienie za nieuzasadnione i mylące stosowanie skrótu QA testerska gimbaza ma to, że w praktyce często rzeczywiście jest jedyną instancją dbałości o jakość. Nie, MBA ani PMI, ani IIBA niczego godnego uwagi o testowaniu nie nauczają. Słowa Hansa Schefera, napisane przez dwudziestu laty, „tester nie jest odkurzaczem" (Hans miał na myśli, że tester nie powinien być odkurzaczem, choć nim zwykle bywa), nic niestety nie straciły na aktualności.

Obecność licznej rzeszy mądrych i technicznie sprawnych testerów, marzących o tym, aby wypróbować w praktyce swoje umiejętności i kompetencje w stosowaniu „Selenium" czy „JMeter", stwarza dodatkową pokusę dla kierownictwa, aby problemy likwidować zamiast u źródła to tylnymi drzwiami, od strony skutków.

Sam osobiście doświadczyłem tego niejeden raz, na przykład trzy lata temu pracowicie uruchamiałem „FoneMonkey" oraz „Squish", narzędzia do automatycznego wykonywania testów na iPhone, w firmie, gdzie daremnie usiłowałem obronić i wdrożyć znacznie prostsze i tańsze usprawnienie, w formie przenoszenia wymagań opisanych artystycznie w… PowerPoint do arkuszy kalkulacyjnych.

Ogłoszenia „dam pracę testerowi" wśród listy wymagań stawianych kandydatom, oprócz znajomości – oczywiście! – „Selenium", „JMeter" oraz „HP Quality Center", zwykle zawierają też coś na temat elastyczności, komunikatywności oraz umiejętności znajdowania, gromadzenia i porządkowania chaotycznie zabałaganionych wymagań dla oprogramowania. Pracodaw-

ca kalkuluje więc sobie, że kosztownego inżyniera wymagań/analityka biznesowego zastąpi tańszym testerem, którego w dodatku można będzie wykorzystać jako kozła ofiarnego, jeśli coś w projekcie się opóźni. Gimbaza!

Co prawnicy wiedzą o testowaniu?

Prawnicy dbają o to, aby do umów „wdrożeniowych" (czytaj – umów na zbudowanie, akurat w odniesieniu do produktów IT zwanych dziwacznie „wdrożeniowymi") wpisać tak zwane kryteria odbioru, czyli te warunki, które produkt musi spełniać, aby móc uznać, że wykonawca zrealizował swoją część umowy i ma prawo wystawić fakturę oraz żądać zapłaty. Jak sprawdzenie spełnienia kryteriów odbioru ma być dokonane, to już – słusznie – prawnika nie interesuje. Dominującą formą kontroli są „testy", z reguły „testy odbiorcze", na samym końcu projektu. Umowa może zawierać załącznik określający precyzyjniej te testy i sposób ich wykonania, ale nie oczekujmy, ze znajdziemy tam dojrzałe i zaawansowane rozwiązania. Forma, którą zakłada wiele umów, to testy przy użyciu scenariuszy napisanych… przez samego dostawcę!

Nie jest rolą prawników, ani tym bardziej prawa, dbać o merytoryczną stronę umów; mają zadbać o stronę formalno-prawną. Dobrze napisana umowa musi zawierać kryteria odbioru, ale nie jest rzeczą prawnika pomóc stronom określić, na ile sposób ich weryfikacji jest dobrze opisany! Tym bardziej że choć umowa na budowę programu i umowa na budowę domku letniego mają bardzo podobną strukturę, o którą dba prawnik, to techniki testowania są zupełnie różne dla domów i dla aplikacji, więc powinny je znać i określać strony umowy, nie prawnicy!

Dyskusja o PKW, czyli festiwal ignorancji

Niestety, strony umowy, szefowie i dyrektorzy i zamawiającego, i wykonawcy zwykle nie znają ani testowania, ani inżynierii wymagań. Wiedza na temat testowania pozwoliłaby im wszak unikać takich porażających wpadek jak ta z systemem do wyborów samorządowych, a wiedza na temat inżynierii wymagań spowodowałaby, że załączniki do umów „wdrożeniowych", zwane notorycznie „specyfikacjami funkcjonalnymi" (czemu „funkcjonalnymi"???), byłyby odrobinę lepsze, niż zwykle są.

Szczególnie niezasłużenie złą sławą cieszy się, niesłusznie, prawo zamówień publicznych, oskarżane o to, że jakoby powoduje historie w stylu ostatniej afery PKW. Powtarzany bezmyślnie i do znużenia zarzut wobec tego prawa, że premiuje kryterium ceny, nie wytrzymuje krytyki. Po pierwsze, ostatnia aktualizacja tego prawa sugeruje stosowanie również kryterium jakości, a po drugie, jeśliby specyfikacja wymagań – SIWZ – była dostatecznie precyzyjna, określając dokładnie i przedmiot zamówienia, i warunki jego odbioru, w tym jakość, lub metody jej zapewnienia, i pomiaru, wówczas kryterium ceny byłoby zupełnie wystarczające! Oczywiście, to stawia wymagania wobec zleceniodawcy, którym zwykle nie jest on w stanie podołać, bo ani na testowaniu, ani na innych obszarach inżynierii jakości nie zna się ani w ząb.

Znając krzywą Boehma, której znaczenie dla jakości musi znać każdy, kto ukończył

choćby kurs podstawowy ISTQB, umiano by powszechnie wykorzystać - wybacz, Czytelniku, prawniczy język – artykuł 31a tzw. tak zwanego dialogu technicznego: „zwracając się o doradztwo lub udzielenie informacji w zakresie niezbędnym do przygotowania opisu przedmiotu zamówienia, specyfikacji istotnych warunków zamówienia (SIWZ) lub określenia warunków umowy".

Dyskusja, która wybuchła wokół informatycznych aspektów afery PKW, ujawniła szokujący ogrom powszechnej niewiedzy, czym jest testowanie oprogramowania i ani co można za jego pomocą osiągnąć. Wypowiadali się na jego temat rzekomi specjaliści od „cyberbezpieczeństwa" (?), od jakości kodu, od audytów. Zamiast „testowanie" pisano „audyt", „audyt bezpieczeństwa", „testy bezpieczeństwa", dziwiono się, czemu nie przetestowano wszystkiego („to oczywiste, że trzeba było sprawdzić wszystko", napisał jakiś młody z wypiekami). Czyżby należało w szkolnej podstawie programowej wymienić jedną z wielu tam zalegających, bezużytecznych bzdur, na trochę nauki podstawowych zasad inżynierii jakości, nie tylko dla oprogramowania?

Suchar

Gimbaza testerska nie ma ograniczenia wieku. Wielu szacownych, znanych na świecie specjalistów jakości do późnych lat wciąż kręci się wokół tych samych zagadnień, jakby nigdy nie dorośleli. Przykładów takich siwowłosych gimbusów znam wiele… Na przykład po raz pierwszy zetknąłem się z koncepcją automatyzacji testów przy użyciu słów-kluczy dwadzieścia lat temu. Znakomity pomysł, pomyślałem, pomyślało tak wielu, metodę zaczęto stosować szerzej. Wydawałoby się, że stanie się częścią powszechnej wiedzy, nie trzeba jej będzie już szerzyć i propagować jako nowości, ale gimbaza działa! Po pierwsze, dwie popularne metodyki automatyzacji stosowane w świecie Agile, skupione wokół narzędzi „FitNesse" oraz „Cucumber", 100% metody słów-kluczy, nawet tej nazwy nie wspominają; cóż, każde kolejne pokolenie widać musi się bawić na nowo tymi samymi zabawkami, udając, że są wielką nowością. A jednocześnie tej jesieni w Gdyni, niedługo przed odpłynięciem promu do Karlskrony, z zakłopotaniem słuchałem wykładu siwowłosej osoby omawiającej tę metodę… na nowo. Nie wypada mieć tyle lat i wciąż bawić tymi samymi, starymi zabawkami!

Beka

Beka musi być, inaczej trudno coś nawet sprzedać. Mania nowości i rozrywki, czyli kompulsja tego, co czterdzieści lat temu nazywano „byciem równym", a dziesięć lat temu, „byciem cool", ogromnie utrudnia samodzielne myślenie, odbiera odwagę wyjścia poza dozwolone i narzucone przez rówieśnicze środowisko ramy. To są społeczne zjawiska, nie nowe, ale dziś łatwiej niż dawniej stwarzające pozory myślenia, no bo przecież trudno jakoś spodziewać się, że na arcywysokotechnologicznej platformie ktoś będzie głosił zupełne głupoty.

Myślę, że nie musi ciągle być beka. Myślę, że testowanie powinno porzucić gimbazę i zacząć wdzierać się w obszary wyższe, niesłusznie w tej chwili zarezerwowane dla absolwentów MBA, IIBA i PMI, niemających do niego kompetencji. Wydorośleć z fascynacji narzędziami i pseudonowinkami. Aby to się powiodło jednak, testowanie

AUTOR

Bogdan Bereza

Posiada wieloletnie doświadczenie w inżynierii jakości.

Jest skutecznym konsultantem, trenerem i prelegentem, autorem kilku książek i wielu artykułów z dziedziny inżynierii oprogramowania. Jest ceniony za umiejętność szerokiego, kompleksowego traktowania jakości oraz łączenia w całość różnorodnych obszarów, takich jak programowanie, projektowanie systemów, inżynieria wymagań, zarządzanie projektami oraz psychologiczne aspektów projektów IT.

Jest inicjatorem i wiceprezesem Stowarzyszenia Inżynierii Wymagań, a także inicjatorem i organizatorem 1. Ogólnopolskiego Turnieju Inżynierii Wymagań.

musi być dopuszczone na salony dorosłości. W tej chwili, aby być uważnie słuchanym na wyżynach, o testowaniu lepiej w ogóle nie wspominać! Tam mądrale mówią o architekturze korporacyjnej, o zgodności, o devopsie, o ładzie korporacyjnym, o roli CIO, o zarządzaniu kompetencjami, a sprawy podstawowe, testy oraz wymagania, brzmią tam jakoś niestosownie, dziecinnie. Dopóki nie zawali się system bankowy, rezerwacji biletów PKP albo PKW; wtedy na chwilę wracają do łask, by wkrótce zniknąć ponownie.

Trollowanie

Boję się, że ten artykuł może mnie narazić na zarzut trollowania. Pozwólcie mi się bronić na zapas. Ironiczny styl nie ma służyć obrażaniu – obrażałbym przecież także sam siebie – lecz ożywieniu dialogu, zbyt często oziębłego i suchego w branży technicznej. Nie wymyślam, jak rasowy troll, kontrowersji na siłę, żeby mieć okazję zabłysnąć, tylko odnoszę się do sytuacji, która, choć chronicznie utrwalona, jest kuriozalna i zła. Wiadomo, jak dobrze realizować przedsięwzięcia IT, ta wiedza jest w licznych opracowaniach, w książkach i w głowach wielu mądrych ludzi, specjalistów w dziedzinie inżynierii oprogramowania, w tym testerów. Niestety, często nie ma jej tam, gdzie zapadają decyzje, ani tam, gdzie realizuje się projekty. Bywa, że nie ma jej ani w głowach biznesowych zleceniodawców, ani w głowach wykonawców projektów. Zróbmy coś, żeby była!

Trzeba nauczyć się działać dobrze. Porzuciwszy na zakończenie poetykę gimbusową, powiem: potrzeba więcej praktycznej prakseologii – umiejętności skutecznego działania.


TESTING
CUP 2015
III EDYCJA MISTRZOSTW POLSKI
W TESTOWANIU OPROGRAMOWANIA
11-13/05/15
STADION WROCŁAW
11/05 x WARSZTATY
GOJKO ADZIC "SPECIFICATION BY EXAMPLE"
12/05 x ZAWODY
13/05 x KONFERENCJA
testingcup
@testingcup
testingcup.pl

Joanna Moćko

Co wiemy, a czego jeszcze nie o TestingCup 2015?

TestingCup 2015

Po raz trzeci i w ciągu trzech dni (11-13 maja 2015 roku) odbędzie się TestingCup czyli zawody, konferencja i warsztaty dedykowane testerom oprogramowania. Wydarzenie, które za sprawą skali i pomysłowości prawdopodobnie nie ma żadnego odpowiednika w Polsce i na świecie.

Wiemy już o zmianie lokalizacji imprezy, która po dwóch latach na Stadionie Narodowym przenosi się na Stadion Wrocław. Wszystko to by tchnąć nowego ducha w imprezę, ale też by podziękować wrocławianom, stanowiącym najliczniejszą reprezentację na dotychczasowych mistrzostwach.

12 maja odbędą się zawody, podczas których stanie w szranki 200 testerów z całej Polski. Ze względu na stale rosnące zainteresowanie uczestnictwem w mistrzostwach, organizatorzy zakładają wcześniejsze eliminacje. Podobnie jak podczas poprzednich dwóch edycji, i tym razem współzawodnictwo odbywać się będzie w dwóch kategoriach – zespołowej i indywidualnej. Wyciągając wnioski z poprzednich edycji organizatorzy wprowadzą modyfikacje w zawodach i choć są dość tajemniczy, to mówią o daleko idących zmianach.

Na szczegóły odnośnie przebiegu samych zawodów musimy jeszcze chwilę poczekać. Zostaną opublikowane po Nowym Roku. Zaplanowano sporo niespodzianek oraz nowych atrakcji dla uczestników – panele dyskusyjne, konkursy, a sama formuła Mistrzostw ma być jeszcze bardziej emocjonująca.

Po emocjach związanych z rywalizacją, na uczestników wydarzenia czeka możliwość uczestnictwa w konferencji 13 maja. Konferencja składać się będzie z kilku ścieżek tematycznych. Wśród prelegentów znajdą się zarówno specjaliści z kraju, jak i goście z zagranicy. A na koniec zostaną ogłoszone wyniki.

Jednym z zaproszonych gości ma być Gojko Adzic. Gojko jest znanym na świecie specjalistą zajmującym się tworzeniem

oprogramowania, który pracuje z zespołami nad poprawę jakości ich produktów i procesów. Specjalizuje się w zakresie agile i lean, a w szczególności metodzie specification by example i behaviour driven development.

Jego kultowa już książka „Specification by example" znalazła się na II miejscu listy 100 najlepszych książek agile w 2012 i zdobyła nagrodę Jolt Award dla najlepszej książki roku 2012. W 2011 roku został wybrany najbardziej wpływowym profesjonalistą testowania Agile, a jego blog http://gojko.net/ wygrał UK Agile Award dla najlepszej publikacji online w 2010 roku.

Gojko Adzic ma także poprowadzić towarzyszące TestingCup warsztaty Specification by Example: From User Stories to Acceptance Tests, zaplanowane są na 11 maja, dzień przed zawodami. Liczba miejsc na warsztatach jest ograniczona, warto już teraz zarezerwować sobie miejsce. Więcej informacji na ich temat znajdziecie na stronie organizatora: http://testingcup.pl/gojko-adzic-we-wroclawiu-warsztaty-autora-kultowej-ksiazki-specification-by-example/.

I wy możecie stanąć obok Gojko Adzica i wystąpić w roli Mówcy TestingCup - do 15 lutego można zgłaszać propozycje prezentacji.

Podczas III edycji TestingCup organizatorzy chcą się skoncentrować na następujących tematach i mówcy z tych obszarów będą preferowani:

- podstawy testowania oprogramowania
- testowanie bezpieczeństwa
- automatyzacja wykonania testów
- języki skryptowe w automatyzacji testów
- testowanie w Kanban, DevOps, Lean Startup i innych współczesnych metodach rozwoju oprogramowania
- nowoczesne podejście do testowania oprogramowania.

Inne ciekawe tematy nie zostaną jednak wykluczone. Szczegółowe informacje na temat sposobu zgłaszania propozycji oraz wymagań organizatorów znajdziecie pod adresem: http://testingcup.pl/zostan-mowca-testingcup-2015/.

Podsumowując, 11-13 maja 2015 to data do zapisania w kalendarzu każdego testera. Tam po prostu trzeba być!

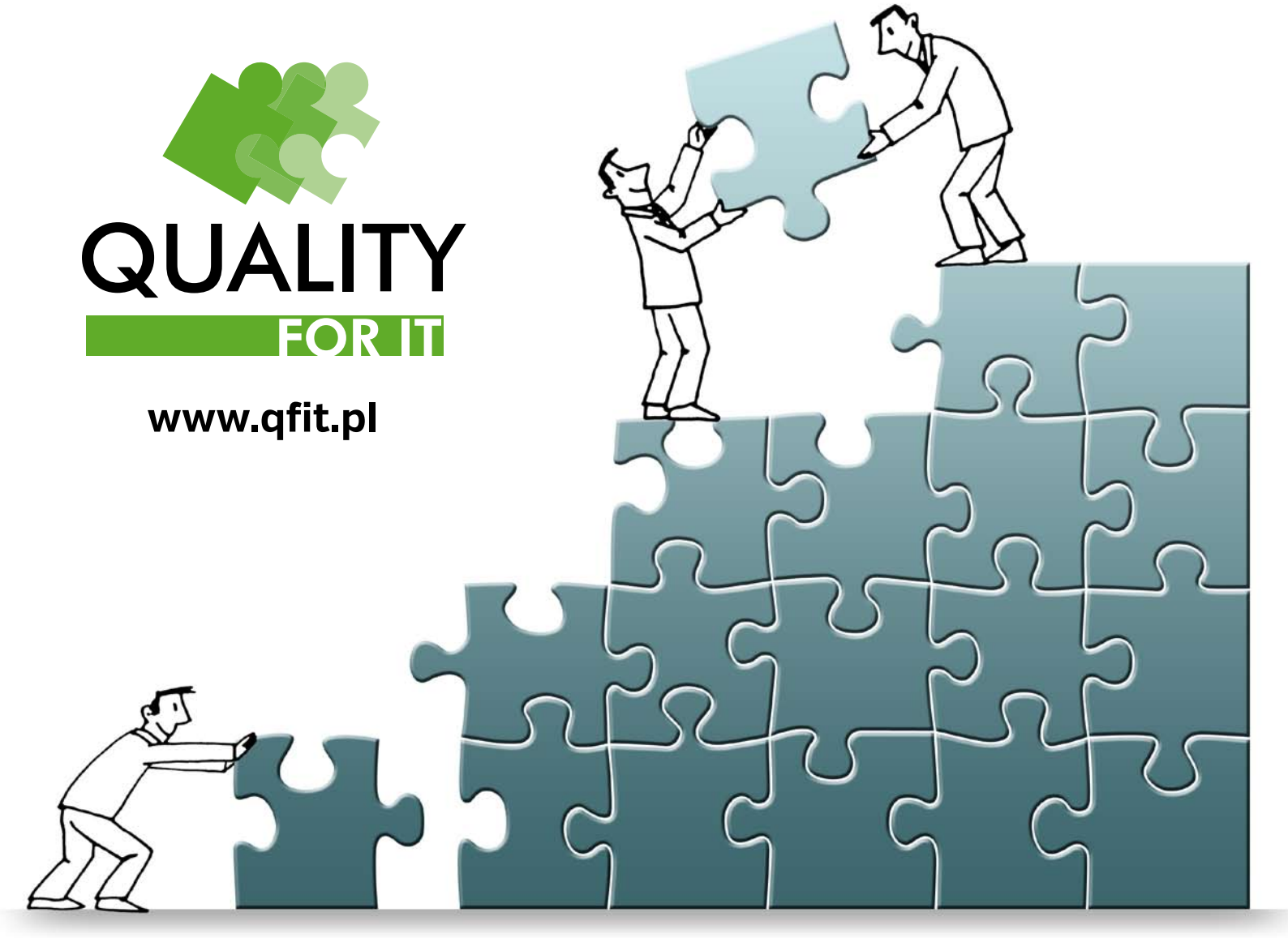
Jeśli chcecie na bieżąco śledzić przygotowania do III edycji TestingCup, zachęcamy do polubienia profilu wydarzenia na facebooku, warto też subskrybować newsletter Mistrzostw poprzez stronę testingcup.pl.

Jak co roku opiekę merytoryczną nad zawodami i konferencją TestingCup sprawuje portal testerzy.pl.

Magazyn Quale, jako patron medialny wydarzenia także będzie Wam na bieżąco przekazywał wszystkie aktualności dotyczące Mistrzostw.

QUALITY FOR IT

www.qfit.pl



KONFERENCJA

QUALITY FOR IT

✓ CELE BIZNESOWE – WSPARTE!

✓ WARTOŚĆ DLA KLIENTA – DOSTARCZONA!

✓ CZAS I BUDŻET – DOTRZYMANE!

5-6 marca 2015, hotel Warszawianka, Jachranka k. Warszawy

Organizatorzy

EVENTION

SJSI STOWARZYSZENIE JAKOŚCI SYSTEMÓW INFORMATYCZNYCH

Inspired by QUALITY

W branży IT coraz więcej uwagi przywiązuje się do jakości wytwarzanego oprogramowania. To właśnie jakość była inspiracją do stworzenia konferencji Quality Excites, która w roku 2015 odbędzie się już po raz czwarty. Jest ona platformą wymiany wiedzy w dziedzinie jakości oraz okazją do poznania nowych branżowych trendów i wymiany doświadczeń. Edycja 2014 zgromadziła 160 uczestników i 35 prelegentów.

Na Quality Excites nie tylko mówi się o jakości, ale stawia się na nią w praktyce, dlatego w programie znajdują się zarówno wykłady, jak i warsztaty. Organizatorzy dają możliwość wystąpienia z własną prelekcją. Zgłoszenia są starannie weryfikowane, by zapewnić najwyższy poziom merytoryczny konferencji.

„Jest to jedyna w Polsce konferencja na temat jakości oprogramowania, gromadząca taką liczbę młodych profesjonalistów, którzy rozumieją i wykorzystują najnowsze technologie i metody pracy. Dotyczy to nie tylko prelegentów, ale również publiczności. Duża wartość merytoryczna, pełne pasji dyskusje, poszerzające horyzonty, zabawa, atmosfera współpracy i profesjonalizmu, dobra lokalizacja, pyszny catering to wszystko czego potrzeba, żeby zorganizować świetną konferencję (...)" – tak edycję Quality Excites 2014 podsumował Krystian Kaczor (Agile Coach, QAgile).

Konferencja odbywa się raz w roku w okolicach maja. Jako platforma networkingowa, tworzy ciągle rosnącą społeczność ludzi zainteresowanych tematem jakości w oprogramowaniu. Pomiędzy kolejnymi edycjami konferencji, organizowane są cykliczne, nieformalne spotkania Quality Meetup, na których również odbywają się krótkie prelekcje.

Organizatorem konferencji jest firma Future Processing, zajmująca się wytwarzaniem zaawansowanych systemów informatycznych. Firma znajduje się w czołówce zestawienia Deloitte dla najszybciej rozwijających się spółek technologicznych Europy Środkowej „Technology Fast 50”.

Więcej informacji można znaleźć na stronie konferencji oraz na profilu na Facebooku.

Wydawca
VWT Polska Michał Kruszewski
Przy Lasku 8 lok. 52, 01-424 Warszawa
Numer NIP 5272137158
Numer REGON 142455963

Redaktor naczelny
Karolina Zmitrowicz
karolina.zmitrowicz@quale.pl

Zastępca redaktora naczelnego
Krzysztof Chytła
krzysztof.chytla@quale.pl

Redaktorzy
Bartłomiej Prędki
bartlomiej.predki@quale.pl
Michał Figarski
michal.figarski@quale.pl

Korekta
Zuzanna Gościcka-Miotk
z.goscicka@gmail.com

WWW
www.qualemagazine.com
www.quale.pl

Facebook
http://www.facebook.com/qualemagazine

Reklama
info@quale.pl
Współpraca
Osoby zainteresowane współpracą w zakresie publikacji prosimy o kontakt:
info@quale.pl

Grafiki użyte w magazynie

Okładka

http://pixabay.com/pl/aplikacja-telefon-kom%C3%B3rkowy-iphone-141046/

Treść

http://pixabay.com/pl/brzegowe-morze-morze-ba%C5%82tyckie-339251/
http://pixabay.com/pl/nowym-jorku-most-brookli%C5%84ski-noc-336475/
http://pixabay.com/pl/chaos-rozporz%C4%85dzenia-tarcza-p%C5%82yta-391652/
http://pixabay.com/pl/fala-koncentryczne-kr%C4%99gi-fal-wody-64170/
http://pixabay.com/pl/schody-%C5%9Blimak-latarnia-morska-274614/
http://pixabay.com/pl/opryskania-powitalny-aqua-wody-164964/
http://pixabay.com/static/uploads/photo/2012/10/06/03/46/steinmann-60007