QUALE

Quale 3/2015

TESTWAREZ

Testwarez 2015

Materiały pokonferencyjne

Edycja specjalna

Spis treści

TESTWAREZ 2015

5. Dlaczego nie powinniśmy zapominać o metodologii Waterfall?
Marta Firlej, Tomasz Olszewski

9. Złoty środek na wszelkie zło – czyli jak zabrać się do pisania testów automatycznych
Natalia Krawczyk

14. Wzorce projektowe w automatyzacji testów aplikacji webowych
Natalia Krawczyk

20. JMeter – narzędzie testera
Adrian Bala

26. Teoria kolejek – matematyka i efektywne wytwarzanie oprogramowania
Wiktor Żołnowski

32. Cardinal sin
Remigiusz Dudek

Spis treści

INŻYNIERIA OPROGRAMOWANIA

44. Certyfikacja ISTQB – czy warto?
Michał Węgrzyn

49. Zapobieganie defektom za pomocą technik statycznych
Patrycja Jakubowska

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

Korekta:
Zuzanna Gościcka-Miotk
zuzanna.goscicka-miotk@quale.pl

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

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

TESTWAREZ

3 dni

300 uczestników

44 prelegentów

8 warsztatów

Ponad 30 prelekcji

Panele dyskusyjne

Egzaminy ISTQB

Konkursy

Integracja

Marta Firlej
Tomasz Olszewski

Dlaczego nie powinniśmy zapominać o metodologii Waterfall?

Ilu z Was pracuje w projektach prowadzonych w Agile? A ilu z Was potrafi o sobie powiedzieć, że są Agile? I uwaga: „Agile" niekoniecznie oznacza, że musicie godzić się na wszystko i minutę przed terminem zmieniać pół aplikacji ;-)

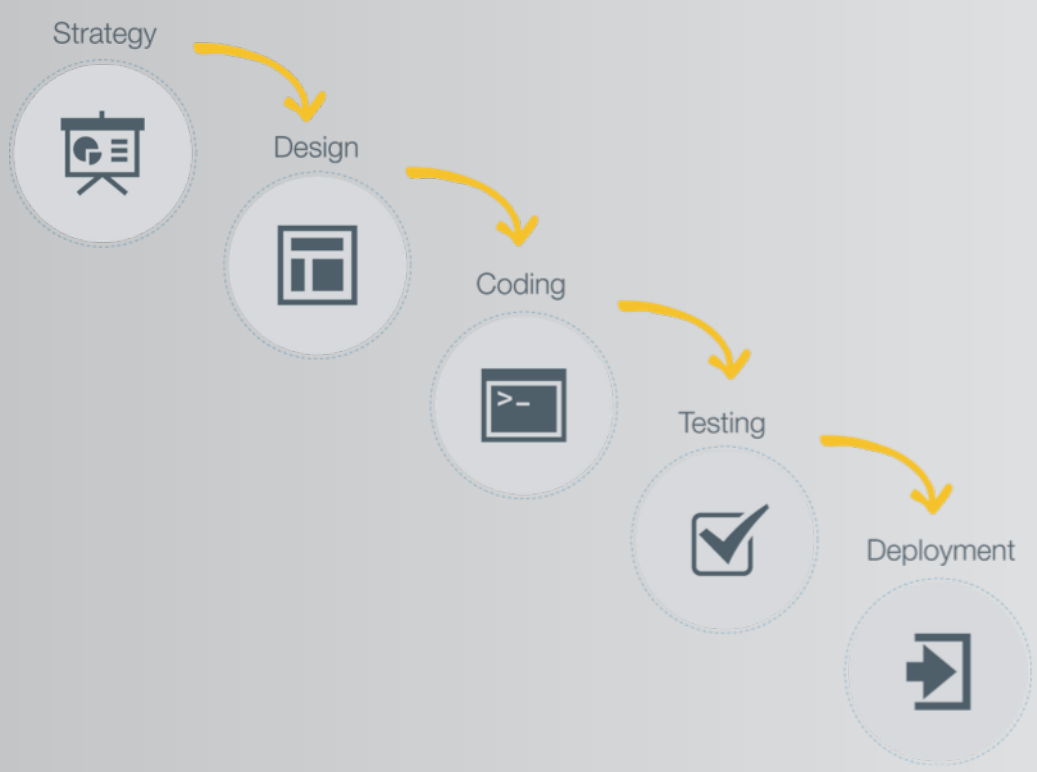
Wprowadzenie

Agile ma lepszy PR niż Waterfall – uchodzi za bardziej „cool", ale to nie znaczy, że jest lepszy. W końcu trzeba teraz wymyślić coś nowego, sprzedać dużo książek, bo czy tak naprawdę jesteśmy jednocześnie testerami manualnymi, automatycznymi, developerami, scrum masterami, asystentkami i project managerami? Większość z Was mówi, że pracuje w Agile. A kiedy tak naprawdę testujecie? Pierwszego dnia sprintu czy może ostatniego? Kiedy rozpoczynacie cały proces testowania? Na etapie planowania czy w momencie, gdy dokumentacja jest kompletna?

Praca w Agile wymaga zaangażowania klienta. Klienta, który w pełni poświęci się realizacji danego projektu. Nie będzie oczekiwał konkretnej estymacji w przypadku nie do końca sprecyzowanych wymagań.

Istnieje mnóstwo projektów, które zwyczajnie muszą być prowadzone w „stary" sposób. I tutaj zwykle wracamy do Waterfalla. No bo kto wsiadłby do samolotu, którego hamulce byłyby wytwarzane w metodologiach zwinnych – w ostatnim sprintcie dodamy user story: „As a owner of this Boeing 747 I want it to break before end of the airstip so that nobody from passengers die"? A może: „As a NASA we want this mars rover to land so we don't waste milions of dolars"?

Rysunek 1. Model Waterfall



Chcielibyśmy pokazać Wam, że niekiedy zamiast na siłę wdrażać w projektach zwinne podejście, lepiej skupić się na eliminacji strat, stałej poprawie i np. zastosować LEAN Software Development.

Model Waterfall

Zacznijmy od początku. Czym jest Waterfall? To sekwencyjna metodologia wytwarzania oprogramowania, w którym poszczególne procesy przebiegają etapami (rys. 1).

Ze względu na powtarzalność możliwa jest pełna przewidywalność wszystkiego, co będzie się działo w trakcie realizacji projektu. Klient, a może raczej wykonawca oprogramowania, musi mieć jasną wizję tego, co ma powstać. Dzięki temu możemy opracować dokładny plan, harmonogram dostarczania, a co za tym idzie – określić konkretną datę zakończenia. W efekcie koszt projektu jest łatwy do oszacowania. Czy wszystkie te cechy pozwalają na stworzenie czytelnego planu działania? Na pewno…

Zastosowanie

Przejdźmy do sedna.

W przypadku jakich projektów Waterfall sprawdza się lepiej niż Agile? Oto przykłady:

- Safety Critical Systems. Firmy, których produkty określamy jako Safety Critical, należą m.in. do branży bankowej, automotive i healthcare. W tego typu organizacjach cały proces wytwarzania oprogramowania musi być pod stałą kontrolą. Każdy komponent projektu z założenia jest w pełni audytowalny. Ponadto wszystkie elementy specyfi-

kacji powinny mieć pokrycie w testach. Waterfall sprawdza się w przypadku systemów, których najmniejsza awaria może nieść ze sobą poważne konsekwencje dla użytkownika końcowego (np. śmierć klienta) i samej firmy. Takie organizacje zwyczajnie nie mogą sobie pozwolić na wytwarzanie oprogramowania niskiej jakości. Ponadto każde wdrożenie obarczone jest ogromnym ryzykiem.

- Duże, rozbudowane i skomplikowane projekty. Nie twierdzimy, że Waterfalla i Agile'a nie wolno łączyć. W przypadku dużych projektów możliwe jest zastosowanie metodologii zwinnych do testów poszczególnych komponentów, testów integracyjnych czy unit testów. Całość przedsięwzięcia wymaga natomiast dużo bardziej ustrukturyzowanej pracy. Dobrym przykładem jest projekt integracyjny, w którym biorę udział. Niby wytwarzamy w Agile, ale dostawca oprogramowania, do którego mamy się integrować, pracuje w Waterfallu z dostawami raz w miesiącu. Wraz z nową wersją systemu dostarczane są także poprawione defekty. Sam klient wymaga określonego procesu testowania, z tworzeniem dokumentacji testowej i zarządzaniem konkretnymi, z góry zdefiniowanymi zadaniami. W efekcie Agile zamienia się w Waterfall.

Podsumowanie

Podsumowując: zachęcamy Was do czerpania tego, co najlepsze, z różnych metodyk. Nie starajcie się być na siłę Agile. Nie zawsze podążanie za modą jest najlepszym wyjściem. W moim zespole wprowadziliśmy pewne elementy Agile, takie jak: daily meetingi, aby szybciej i lepiej się komunikować; retrospektywę po wdrożeniach; ograniczoną liczbę zadań rozpoczynanych przez poszczególnych QA. Czy to spowodowało, że projekt jest Agile? Nie, ale jakość naszej pracy zdecydowanie się poprawiła.

ZALETY I WADY MODELU WATERFALL

Wady	Zalety

• Klient późno widzi efekty pracy nad projektem.
• Wprowadzanie zmian jest kłopotliwe.
	• Pełna kontrola.
	• Znane są:
	  • budżet,
	  • zakres projektu,
	  • harmonogram.
	• Projekt bez problemu przejdzie audyt.

AUTORZY

Marta Firlej

Przygodę z testowaniem oprogramowania rozpoczęła w 2006 roku. Podczas swojej pracy jest skupiona na produkcie, procesie i zespole. Angażuje się w zapewnianie jakości; szczególnie upodobała sobie testy użyteczności, testy funkcjonalne i zarządzanie testami. Miała okazję nie tylko budować zespoły testerskie, ale także pracować z klientem przy wymyślaniu, planowaniu i tworzeniu oprogramowania.

Członek komisji na zawodach TestingCup 2013 i 2014. Pomysłodawca i organizator konferencji TestFest we Wrocławiu. Posiada certyfikat ISTQB Advance Test Manager.

Po pracy podróżuje, gotuje i jeździ na rowerze.

Tomasz Olszewski

W projektach związanych z zapewnianiem jakości oprogramowania pracuje od 2010 roku. Bogate doświadczenie zdobyte w innych branżach pozwala mu na wniesienie powiewu świeżości i innego punktu widzenia do testowania oprogramowania. W pracy skupiony na budowaniu użyteczności i testach eksploracyjnych. Zaangażowany w poprawę i usprawnianie procesu testowania w projektach, w których uczestniczy. Obecnie związany z testami bazodanowymi dla banku inwestycyjnego.

W życiu prywatnym biegacz długodystansowy, tropiciel, wielbiciel wycieczek górskich i fan *Gwiezdnych Wojen*.

Natalia Krawczyk

Złoty środek na wszelkie zło – czyli jak zabrać się do pisania testów automatycznych

Wielu z nas przeważnie cieszy się na nowy projekt, w którym będziemy mogli pisać testy automatyczne. Czasem startujemy od początku, innym razem dostajemy utworzone już testy do tzw. utrzymania. Często ten entuzjazm szybko przemija. Gdy zaglądamy do kodu, który dostaliśmy „w spadku", łapiemy się za głowę i pytamy samych siebie: „dlaczego ja?". Czasem gdy zaczynamy nowy projekt, jesteśmy nim tak zafascynowani, że zapominamy usiąść na chwilę, zastanowić się nad jego strukturą, nad tym, co chcemy osiągnąć. Zamiast tego zaczynamy od razu pisać. Po pewnym czasie zdajemy sobie sprawę z tego, że popełniliśmy błąd. Testy nie testują tego co powinny, nie są wiarygodne, a ich utrzymanie jest koszmarem.

Geneza

Podczas jednego z projektów, w którym opiekowałam się mało doświadczonymi testerami automatycznymi, zauważyłam, że praca przebiega właśnie w tym złym kierunku. Zmieniłam więc przebieg typowego spotkania statusowego. Wiedziałam, że musimy zmienić sposób tworzenia testów – i to jak najszybciej.

Zaczęliśmy od wymagań. Najpierw zapytałam zespół, jak wyobraża sobie swoje testy w idealnym świecie. Jakie by były? Padło wiele odpowiedzi: stabilne, szybkie, niezależne, czytelne i proste. Następnie poprosiłam zespół o szczerą ocenę obecnych testów z uwzględnieniem tych kryteriów. Utworzyliśmy trzystopniową skalę:

1. (:( ) – testy nie spełniają tego wymagania;
2. (:/) – testy po części spełniają to wymaganie;
3. (:)) – testy w 100% spełniają dane wymaganie.

Jeśli chociaż jedna osoba wahała się nad oceną pozytywną, uznawaliśmy, że testy nie zasługują na taką notę.

Później wybraliśmy najważniejsze wg nas cele, które chcielibyśmy osiągnąć. Następnie opisaliśmy sposoby, na jakie można by to zrobić. Zastanowiliśmy się wspólnie, co ma wpływ na dane wymaganie. Omówiliśmy też realia – czyli co tak naprawdę w naszym przypadku dałoby się zrobić, aby chociaż trochę przybliżyć się do celu. I tak właśnie powstał nasz złoty środek – jako zbiór wskazówek, jak pisać testy, aby zrealizować postawione im wymagania.

Jak stworzyć swój złoty środek?

Nie ma jednego uniwersalnego złotego środka. W zależności od projektu zastosujemy inną strategię, inny poziom testów, który będziemy automatyzować. Cele i wymagania postawione testom mogą się znacznie różnić w przypadku dwóch podobnych projektów. Wpływ na to ma w dużej mierze klient, który zleca nam napisanie testów.

Wymagania

Tworzenie złotego środka zaczynamy od określenia wymagań klienta oraz swoich. Klient przeważnie nie dostarcza nam długiej listy „życzeń”, jednak nie oznacza to, że nie zażąda od nas 90% pokrycia przypadków testowych na poziomie testów akceptacyjnych, czasu trwania zestawu testów poniżej 2 godzin czy szczegółowego raportu z testów.

Czego my oczekujemy? Niezależnie od poziomu testów czy testowanej aplikacji możemy wymienić kilka pożądanych cech.

Testy idealne są:
- szybkie,
- stabilne,
- łatwe w utrzymaniu,
- czytelne,
- wiarygodne,
- a ponadto przekazują dobrą odpowiedź zwrotną.

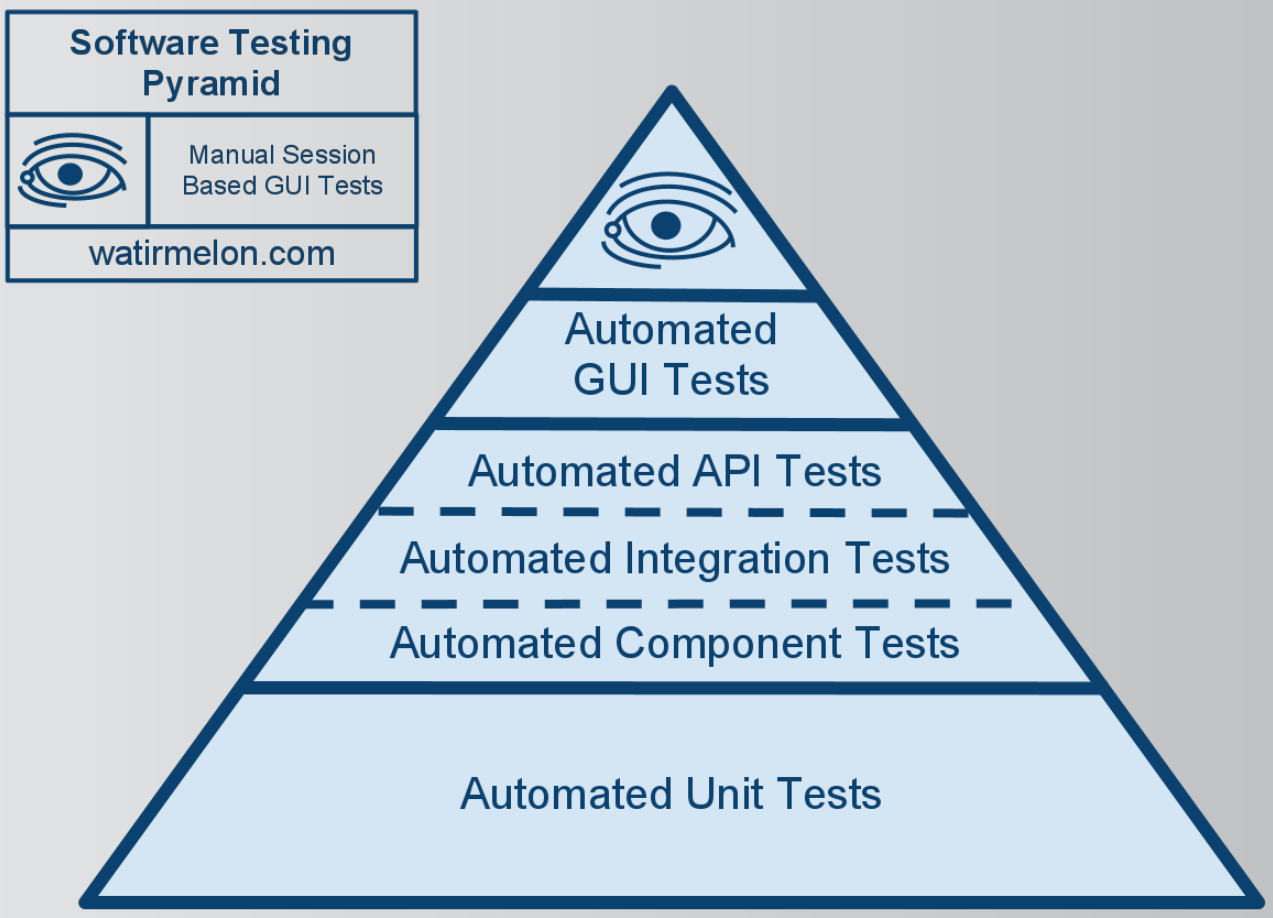
Strategia

Kolejne rzeczy, jakie należy wybrać czy też określić na podstawie wymagań klienta, to poziom testów oraz strategia, wg której będziemy tworzyć procedury testowe i decydować o pokryciu.

W zależności od wybranego poziomu testów inaczej zdefiniujemy procedury testowe. Za przykład niech posłużą nam koszyk w sklepie internetowym oraz funkcjonalność dodawania produktu do koszyka. W przypadku testu jednostkowego będziemy sprawdzać wynik metody odpowiadającej za dodawanie produktu do koszyka. Na poziomie testów integracyjnych możemy kontrolować np. to, czy stan magazynu w innym systemie się zmienił. Podczas testów systemowych sprawdzimy zaś, czy koszyk zawiera właściwy produkt w odpowiedniej ilości oraz cenę. Na poziomie testów akceptacyjnych skupimy się również na tym, czy będziemy mogli kontynuować zakupy lub zakończyć je, zapłacić za zamówienie i otrzymać potwierdzenie. To oczywiście przykłady, które mogą się różnić w zależności od systemu.

Jeśli zamierzamy tworzyć testy na wielu poziomach, warto pamiętać o Piramidzie Testowania Oprogramowania zaprezentowanej przez Alistera Scotta (rys. 1). Zgodnie z nią najwięcej testów automatycznych powinniśmy wykonać na poziomie testów jednostkowych, a najmniej na poziomie interfejsu użytkownika. Testy automatyczne na poziomie jednostkowym są tanie, szybkie i łatwe w utrzymaniu. Odwrotnie jest w przypadku testów akceptacyjnych

Rysunek 1. Piramida testowania oprogramowania wg Alistera Scotta



wykonywanych na interfejsie użytkownika. Napisanie ich i utrzymanie pochłaniają duże koszty, a czas ich trwania okazuje się bardzo długi.

Istnieje kilka podstawowych strategii testowych stosowanych w automatyzacji:

1. **Smoke tests** – testy takie uruchamiane są przeważnie po każdej zmianie kodu wgranej do systemu kontroli wersji. Sprawdzają np., czy aplikacja się uruchamia, czy możemy nawiązać połączenie z bazą danych, czy jesteśmy w stanie zalogować się do aplikacji.
2. **Money path** – strategia ta polega na automatyzacji testów znajdujących się na tzw. ścieżce money path, czyli w ciągu operacji zapewniających klientowi zysk. W przypadku sklepu internetowego będzie to na pewno proces zakupu produktów.
3. **Happy path** – w tym podejściu stosujemy testy pozytywne, sprawdzamy proste ścieżki poprawnego użycia aplikacji.
4. **End to end** – testy sprawdzające cały przepływ aplikacji od początku do końca. Zwykle przenikają przez wiele warstw aplikacji.
5. **Regresja** – strategia opiera się na pokryciu jak największej liczby testów regresyjnych.

Konstrukcja złotego środka

Posiadając wiedzę o zebranych wymaganiach, wybranych poziomach testów do automatyzacji oraz ich poszczególnym pokryciu, możemy przejść do kolejnego etapu – konstrukcji własnego złotego środka. Jeśli decydujemy się na automatyzację kilku poziomów, polecam wykonanie tego kroku dla każdego poziomu testów osobno, ze względu na inne możliwości, ograniczenia oraz cele.

Ten etap to tak naprawdę burza mózgów – wspólnie z zespołem zastanawiamy się, jak możemy osiągnąć zamierzone cele, a następnie spisujemy pomysły. Załóżmy, że chcemy mieć wiarygodne testy. W pierwszej kolejności musimy odpowiedzieć sobie na pytanie, czy na pewno rozumiemy, co to znaczy. W naszym przykładzie testy wiarygodne to testy bez fałszywie negatywnych i fałszywie pozytywnych wyników. Następnie zapisujemy, jak możemy osiągnąć takie efekty. Jeśli ciężko nam znaleźć rozwiązanie, warto spróbować podejść do zadania od drugiej strony. Możemy zastanowić się, dlaczego otrzymujemy takie, a nie inne wyniki (być może problemem są nieodpowiednie asercje czy niestabilne środowisko), a następnie odwrócić te problemy, aby uzyskać odpowiedzi – napisać asercje sprawdzające tylko to, co ma skontrolować dany test, oraz uruchamiać testy na stabilnym, wyizolowanym środowisku testowym. I tym sposobem powinniśmy opracować zasady, które pozwolą nam na osiągniecie zamierzonych celów.

Na takie spotkanie warto zaprosić osobę z dużym doświadczeniem w automatyzacji testów. Wspomoże ona cennymi radami zespół poszukujący rozwiązań.

Spisane zasady powinny być umieszczone w miejscu łatwo dostępnym dla każdego członka teamu. W momencie dołączenia nowej osoby do projektu należy przedstawić jej te reguły oraz wytłumaczyć, dlaczego zostały spisane i jak się je wykorzystuje.

Jak wdrożyć złoty środek w życie?

Opisane wcześniej spotkanie warto zorganizować jak najwcześniej, jeszcze zanim zaczniemy pisać testy automatyczne. Wówczas należy przeprowadzić sesję narad, podczas których ustalone zostaną cele oraz zasady, wskazówki pomagające je osiągnąć. Kolejny krok to planowanie. Na tym spotkaniu określimy zadania na najbliższy okres. Wspólnie zastanowimy się nad konkretnymi rozwiązaniami (zgodnymi z regułami spisanymi w złotym środku). Po zakończeniu wszystkich zadań należy przeprowadzić retrospekcję. Jest to spotkanie, na którym podsumujemy, co udało nam się osiągnąć. Ocenimy, czy wykonane zadania spełniają założenia spisane w złotym środku. Podczas tego spotkania warto również ustalić, czy wymagania się nie zmieniły. Jeśli tak, należy nanieść poprawki na złoty środek i w kolejnym planowaniu uwzględnić nowe zasady.

W sytuacji, gdy projekt już trwa, trzeba zacząć od spisania wymagań. Następny krok to ocena swoich testów w celu uświadomienia sobie, co robimy źle. Później należy postępować podobnie jak w przypadku początku nowego projektu: zaprojektować własny złoty środek, często planować nowe zadania – z uwzględnieniem spisanych zasad, przeprowadzać regularne przeglądy wykonywanych zadań oraz złotego środka. Trzeba również zmodyfikować testy zgodnie z wprowadzonymi zmianami. Czasem najlepszym rozwiązaniem okazuje się przepisanie ich na nowo, z uwzględnieniem złotego środka. Jeśli możemy pozwolić sobie na zmianę w kodzie istniejących testów, należy robić to powoli, krok po kroku. Najpierw trzeba wybrać jedno wymaganie, na którym najbardziej nam zależy, i zacząć wprowadzać zmiany w kodzie pomagające je osiągnąć.

Podsumowanie

Złoty środek powstał, aby dostarczyć odpowiedzi na pytanie, w jaki sposób definiować testy, jak implementować kod, żeby osiągnąć pożądany efekt. Należy pamiętać o nim podczas planowania nowych testów oraz w trakcie retrospekcji – warto ocenić swoje testy pod jego kątem. Złoty środek powinno się również raz na jakiś czas przeglądać i oceniać. Wymagania wobec testów mogą się bowiem zmienić w trakcie trwania projektu.

Natalia Krawczyk

Wzorce projektowe w automatyzacji testów aplikacji webowych

Wzorce projektowe są odpowiedzią na częste problemy, jakie napotykamy w projektach. Zawierają sprawdzone rozwiązania, dzięki którym możemy tych problemów uniknąć. Dlaczego potrzebujemy ich w testach automatycznych?

Wprowadzenie

Napisanie testów automatycznych do aplikacji webowej wiąże się z dużym wydatkiem. Klienci, którzy za nie płacą, oczekują szybkiego zwrotu inwestycji. Jednak bardzo często testy automatyczne nie spełniają postawionych im zadań, takich jak:

- wykrywanie błędów na stronie;
- zastąpienie części pracy testera manualnego poprzez automatyzacje niektórych jego zadań.

Problemy te są powszechne i dobrze znane, dlatego powstało wiele wzorców projektowych w automatyzacji testów, które w założeniu pozwalają uniknąć wspomnianych trudności.

Problemy projektowe w automatyzacji testów aplikacji webowych

Wielu z nas spotkało się z problemami w trakcie tworzenia lub utrzymania testów automatycznych. Często ich utrzymanie okazuje się droższe od przeprowadzania testów manualnych. Dlaczego tak się dzieje? Z wielu powodów. Jednym z nich jest problem stabilności testów. Okazuje się on poważny, ponieważ wiąże się z utratą czasu na poszukiwanie błędu (czy leży on po stronie testowanej aplikacji, czy testu?), jak również z obniżeniem zaufania do testów.

Stabilność

Kiedy test można nazwać stabilnym? Kiedy uruchomimy go 10 razy i za każdym razem uzyskamy taki sam wynik? A co będzie za 11. razem? Mianem stabilnego można określić test, który zawsze daje taki sam wynik.

Action Wrapper Pattern

Idealnym rozwiązaniem problemu ze stabilnością aplikacji webowych jest Action Wrapper Pattern. Wzorzec ten polega na implementacji własnych metod odpowiadającym akcjom (np. kliknięcie elementu), które dodatkowo zawierają oczekiwania dotyczące różnych warunków. Przykład: wyświetlenie się danego elementu na stronie. Metody te powinny uwzględniać też inne warunki, tj. czekanie na zakończenie wszystkich zapytań Ajax, aż zniknie symbol ładowania się strony albo element będzie dostępny.

Testy, które wykorzystują tylko takie metody, są znacznie stabilniejsze od testów opierających się na standardowych metodach dostarczonych przez frameworki do automatyzacji testów. Czy opisane podejście ma jakieś minusy? Tak, lecz niewielkie. Jednym z nich jest konieczność zmiany przyzwyczajeń. Trzeba pamiętać o tym, aby zawsze używać własnych metod odpowiadającym akcjom na stronie.

Stosując ten wzorzec, możemy popełnić kilka błędów. Szczególnie należy uważać przy implementacji własnych metod czekających na spełnienie warunku. Najbardziej popularnym błędem jest stosowanie nieskończonej pętli while – czekającej, aż element nie będzie wyświetlony na stronie. W przypadku poprawnie działającej aplikacji nie zauważymy tego problemu. Jeżeli jednak w danym momencie pojawiłby się błąd polegający na tym, że dany element nie jest ukrywany lub usuwany – pętla nie zostałaby nigdy przerwana. Implementując testy automatyczne, zawsze należy pamiętać o tym, jak nasz kod zachowa się w przypadku błędu na stronie.

Hermetic test pattern

Na stabilność testu może wpływać również stan aplikacji w momencie uruchomienia się testu. Jeśli test oczekiwałby aplikacji w pewnym stanie (np. z dodanym użytkownikiem X), a zastałby ją w innym, to zakończyłby się bez osiągnięcia swojego celu.

Hermetic test pattern polega na pisaniu testów w taki sposób, aby nie dbały o stan aplikacji, nie oczekiwały na dane znajdujące się w niej. W tym wzorcu każdy test sam zapewnia sobie dane testowe, nie korzysta z tego, co już jest dostępne i utworzone. Często takie testy uruchamianie są w bardzo czystym środowisku testowym, bez nadmiaru danych, a nawet w stanie początkowym aplikacji, gdy baza danych zawiera tylko dane początkowe.

Zastosowanie tego wzorca pozwala zyskać przede wszystkim niezależność testów, a dzięki niej możliwość równoległego ich uruchamiania. Jest to bardzo duża zaleta, ponieważ stosowanie tego wzorca wiąże się z koniecznością skompletowania własnych danych testowych. Czas na ich przygotowanie będzie długi, jeśli zdecydujemy się utworzyć je poprzez interfejs graficzny aplikacji. Należy pamiętać również o tym, że testy uruchamiane w hermetycznym środowisku mogą nie wykryć pewnych

błędów, które występują tylko w sytuacji, gdy aplikacja posiada wiele danych.

Inne przyczyny problemów ze stabilnością

Wpływ na stabilność testu często mają dodatkowe, poboczne usługi w aplikacji webowej, takie jak reklamy. Istnieje również wzorzec, wg którego powinniśmy ucinać wszystkie takie zapytania – jest to Black Hole Proxy Pattern. Zastosowanie go nie wiąże się z wydłużeniem czasu trwania testu, tak jak w przypadku poprzednich wzorców. Wręcz przeciwnie – może się on skrócić. Minusem tego podejścia jest środowisko niezbliżone do środowiska produkcyjnego.

Utrzymanie testów automatycznych

Drugim najbardziej popularnym problemem projektowym w automatyzacji testów jest ich utrzymanie. Koszt wprowadzenia zmian w kodzie testów niekiedy okazuje się bardzo wysoki. Czy da się uniknąć tak dużych wydatków? Na szczęście jest parę wzorców, których zastosowanie może pomóc.

Page Object Pattern

Najczęściej zmiany, jakie musimy wprowadzić w testach, dotyczą modyfikacji interfejsu użytkownika. W tym przypadku najlepszym rozwiązaniem jest Page Object Pattern. Wzorzec ten polega na zdefiniowaniu elementów na stronie oraz metod z akcjami do nich tylko raz – w klasie dedykowanej konkretnej stronie. Przykładowo w klasie LoginPage powinny znaleźć się definicje elementów, tj. pole tekstowe do wpisania hasła, loginu czy przycisk Zaloguj, a także metoda odpowiadająca za wpisanie nazwy użytkownika.
Zastosowanie tego wzorca ma wiele zalet. Możemy do nich zaliczyć choćby porządek w kodzie. Wiemy, że metod i lokatorów odpowiadających za zmianę nazwy użytkownika będziemy szukać w klasie UserDetailsPage, a nie w np. LoginPage. Dzięki temu również modyfikowanie testów jest łatwiejsze. Każdą zmianę będziemy musieli wprowadzić tylko raz – w klasie odpowiedzialnej za daną stronę. Czy takie rozwiązanie ma jakieś wady? Owszem. Można do nich zaliczyć dodatkowy koszt, który poniesiemy na implementację poszczególnych stron przed rozpoczęciem pisania testów. Jest to jednorazowy wydatek, ponadto płynące z niego korzyści będą znacznie większe.

Default Values Pattern

Testy opierają się na danych testowych, które mogą się różnić między środowiskami testowymi. Jeśli zostaną wpisane na sztywno, koszt ich zmian będzie wysoki. Wzorzec Default Values Pattern polega na wykorzystywaniu w teście wartości domyślnych. Oparty na nim test odnosi się do zmiennej lub parametru konfiguracyjnego, a nie do sztywno wprowadzonej wartości. Podczas logowania się do aplikacji test nie dba o to, czy hasło użytkownika to „Test123”, czy „qwerty”. Ważne jest, aby w polu Hasło wpisać ciąg znaków odpowiadający hasłu użytkownika, na którego konto chcemy się zalogować.

Stosując ten wzorzec, trzeba jednak uważać, aby wykorzystywane dane nie były zbyt jednorodne. W tym celu należy posługiwać się generatorami danych. Ich użycie wiąże się zaś z trudnością określenia real-

nego pokrycia. Jeśli wygenerowane dane nie będą zapisane w dzienniku testów, nie da się sprawdzić, co tak naprawdę przetestował skrypt.

Behaviour Driven Development (BDD)

Procedury testowe podlegają również utrzymaniu. W przypadku, gdy zmieniają się wymagania, musimy pamiętać o wprowadzeniu zmian w dwóch miejscach: w dokumentacji testowej oraz w testach automatycznych. W tym celu możemy zastosować Behaviour Driven Development, w którym testy automatyczne tworzą dokumentację testową.

BDD polega na opisywaniu działania aplikacji na konkretnych przykładach – scenariuszach, następnie automatyzowanych. Każdy krok w scenariuszu testowym ma swoją definicję w kodzie. Kroki te są reużywalne. Oznacza to, że wystarczy raz zaimplementować krok odpowiedzialny np. za logowanie się do aplikacji, aby móc go użyć w każdym kolejnym scenariuszu.

Scenariusze pisane są w języku Gherkin, opierającym się na słowach kluczowych. Słowa te ułatwiają czytanie scenariusza, który powinien brzmieć jak zdanie.

Oto przykładowy scenariusz napisany w takim języku:

Given I am on login page
When I login using incorrect credentials
Then I should see proper validation errors
And I should not be logged in

Niestety, wzorzec ten nierzadko jest źle stosowany. Scenariusze powinny być dokumentacją, a nie szczegółową instrukcją dla testu. Często dodajemy kroki, które jedynie ułatwiają implementację testu. Początkowo wydaje się to nieszkodliwe, ale z czasem tracimy przejrzystość i czytelność testu. Scenariusze przestają być dokumentacją testową, a stają się słownym opisem wszystkich operacji wykonywanych przez skrypt automatyczny.

Inne dobre praktyki w inżynierii oprogramowania

Wielu z nas zapomina, że testy automatyczne to tak naprawdę kolejna aplikacja. Dlatego warto poznać i stosować dobre praktyki w inżynierii oprogramowania, takie jak DRY, KISS czy YAGNI.

DRY (*Don't Repeat Yourself*, czyli Nie powtarzaj się) – zasada ta polega na unikaniu wszelkich powtórzeń w kodzie. Jeśli zamierzamy użyć pewnego fragmentu 2 razy, powinien on zostać odseparowany i umieszczony w osobnej metodzie. Takie podejście pozwala zaoszczędzić sporo czasu, ponieważ zmiany w kodzie musimy wprowadzać tylko w jednym miejscu.

KISS (*Keep it simple, stupid!*, czyli Nie komplikuj, głuptasku!) – odnosi się do wielu dziedzin inżynierii, ale ma swoje zastosowanie również w inżynierii oprogramowania. Zasada ta sprowadza się do stosowania możliwie najprostszej architektury oraz rozwiązań programistycznych.

YAGNI (*You ain't gonna need it*, czyli Nie będziesz tego potrzebować) – praktyka polegająca na implementacji tylko potrzebnego kodu. Zgodnie z tą zasadą nie powinniśmy pisać kodu „na zapas", co często się zdarza przy implementacji stron w Page Object Pattern. Według YAGNI należy

tworzyć tylko kod, który zostanie użyty.

Stosowanie tych praktyk podczas tworzenia testów automatycznych znacznie ułatwia ich utrzymanie.

Najczęściej popełniane błędy

Często w pośpiechu popełniamy błędy, wpadamy na pomysły, które początkowo wydają się dobre, jednak z czasem okazują się antywzorcem. Wiele z nich wynika z braku specjalistycznej wiedzy programistycznej i nieznajomości dobrych praktyk kodowania. Zdarza się również, że test od strony developerskiej jest idealny, ale nie spełnia swojej najważniejszej funkcji – nie testuje.

Chain linked pattern

Bardzo popularnym problemem jest uzależnianie testu od stanu aplikacji, jaki pozostawia po sobie poprzedni test. Przykładowo test A tworzy użytkownika X, test B ma za zadanie przetestować edycję danych użytkownika – wg Chain Linked Pattern test B powinien użyć użytkownika X utworzonego w teście A. Początkowo brzmi to rozsądnie, bo dlaczego nie wykorzystać tego, co już istnieje? Dzięki temu skrócimy przecież czas trwania testu. Jednak podejście to kryje w sobie wiele pułapek i jest uważane za antywzorzec.

Problematyczna może być m.in. ustalona kolejność uruchamiania testów – uzależniamy jeden test od kolejnego. Gdy mamy łańcuch składający się z 10 testów, to

AUTOR

Natalia Krawczyk

Posiada kilkuletnie doświadczenie w testowaniu oprogramowania w różnych technologiach. Testowaniem interesuje się bardziej od strony technicznej, stąd jej zamiłowanie do automatyzacji. Posiada certyfikat ISTQB Advanced Level Test Manager. Prelegentka WrotQA oraz uczestniczka TestingCup.

Lubi zadawać filizoficzne pytania dotyczące testów i szukać na nie odpowiedzi.

W wolnym czasie lchętnie jeździ na festiwale muzyczne i ogląda *Przyjaciół*.

aby sprawdzić 10. test, musimy najpierw uruchomić 9 testów przed nim. Znacznie utrudnia to debugowanie oraz wprowadzanie zmian. Musimy pamiętać, że zmiana w teście 5. powinna być uwzględniona w kolejnym. Dużą wadą takiego podejścia jest brak możliwości uruchamiania testów równolegle.

Spaghetti code

Jeśli nie przykładamy wagi do tego, w jaki sposób implementujemy testy i programujemy, nasz kod nierzadko zaczyna przypominać spaghetti. Wszystko jest ze sobą powiązane. Zmiana w jednym miejscu w kodzie może spowodować wiele nieoczekiwanych błędów i problemów.

Mianem spaghetti code określa się kod z dużą ilością instrukcji warunkowych oraz zagnieżdżeń, które znacznie utrudniają debugowanie oraz utrzymanie.

Podsumowanie

Niestety, nie ma jednego wzorca, który dostarczyłby uniwersalne rozwiązanie wszystkich problemów związanych z automatyzacją aplikacji webowych. Często powinniśmy stosować wiele wzorców naraz, np. Action Wrappern Pattern z Page Object Pattern oraz Hermetic Test Pattern, aby zapewnić stabilne, łatwe w utrzymaniu testy. Należy zadbać o to, by test właściwie się zachował, gdy na stronie będzie błąd. W takiej sytuacji test powinien się zakończyć i odpowiednio zaraportować napotkany problem.

Adrian Bala

JMeter – narzędzie testera

Czym jest JMeter?

Apache JMeter to otwarte oprogramowanie, napisane w Javie i przeznaczone do tworzenia, wykonywania oraz monitorowania testów obciążeniowych, wydajnościowych i funkcjonalnych. Oryginalnie było projektowane i rozwijane przez Stefano Mazzocchiego z Apache Software Foundation, który utworzył je w celu testowania wydajności Apache JServ (projektu zastąpionego przez Apache Tomcat). Następnie JMeter przeprojektowano i wyposażono w GUI, aby ułatwić jego zastosowanie do testów funkcjonalnych. W listopadzie 2011 roku oprogramowanie stało się projektem Apache najwyższego poziomu (ang. *top level*), co oznacza, że zyskał społeczność odpowiedzialną za jego rozwój (ang. *Project Management Commitee*) oraz dedykowany serwis.

Do czego służy JMeter?

Apache JMeter jest używany do testowania wydajności statycznych oraz dynamicznych zasobów, takich jak pliki, dynamiczne języki programowania serwisów internetowych, choćby PHP, Java, ASP.NET, obiekty javowe, bazy danych i kwerendy, serwery FTP itp. Z powodzeniem wykorzystuje się go do symulowania wzmożonego ruchu na serwerze, grupie serwerów, w sieci lub na „hartowanym" obiekcie. Służy również do analizowania całkowitej wydajności pod obciążeniem różnego typu, np. do graficznej analizy całkowitej wydajności lub do testowania zachowania się serwera/skryptu/obiektu przy wzmożonym i zrównoleglonym obciążeniu.

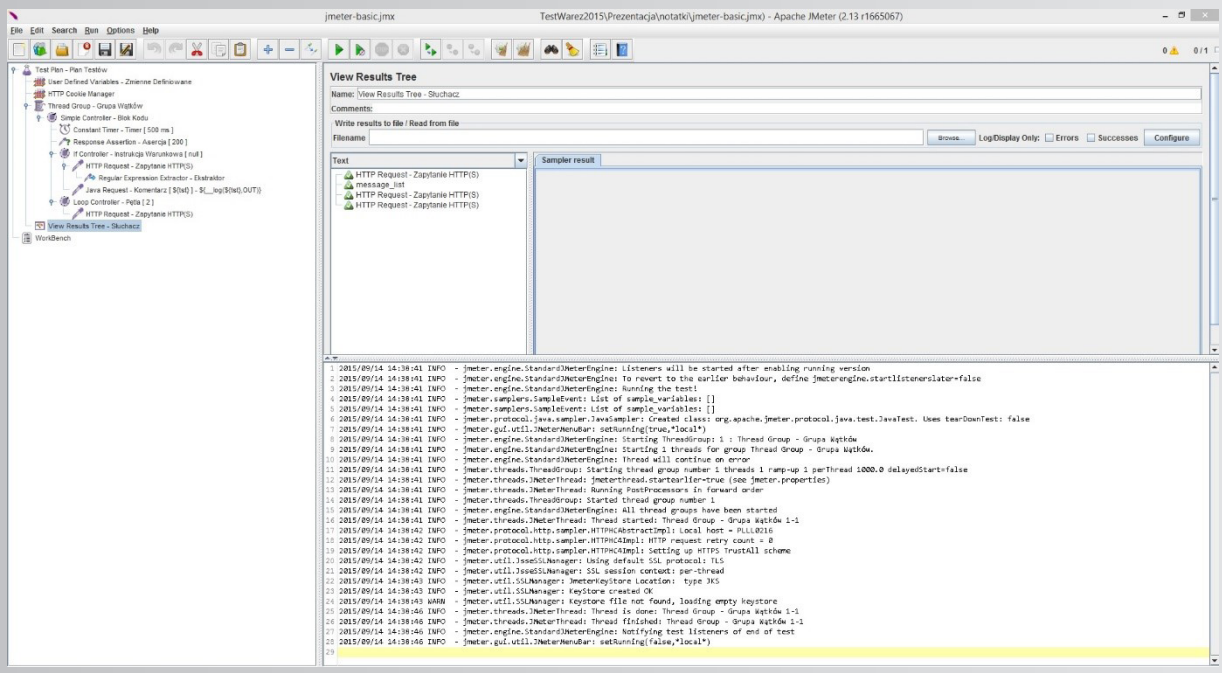
Podstawowe zalety Apache JMeter

- Pozwala na automatyzację testowania (oszczędność).
- Działa na platformach wspierających Javę (przenośność).
- Umożliwia działanie wielowątkowe (zrównoleglenie).
- Jest narzędziem „wielopoziomowym" (użyteczność).
- Wspiera różne protokoły (uniwersalność).
- Dostarcza wysoki poziom abstrakcji (łatwość obsługi).
- Ma rozbudowaną dokumentację (łatwość nauki).
- Jest wspierany i rozwijany (aktualność).
- Może być modyfikowany (dostosowywanie).
- Jest darmowy (ograniczenie kosztów).

Do czego można użyć JMeter?

- Pozwala na wykonywanie zapytań http(s) – GET, POST oraz przetwarzanie zwróconych kodów [1] i wyników odpowiedzi.
- Umożliwia ustanowienie bezpośredniego połączenia z bazą MySQL i wykonywanie zapytań przy wykorzystaniu sterownika JDBC [2].
- Potrafi działać w trybie konsolowym bez GUI, dzięki czemu zainstalujemy go na maszynach bez wyświetlaczy, np. serwerach.
- Może być użyty w celu monitorowania pracy serwera/grupy serwerów poprzez okresowe sprawdzanie statusu.
- Pozwala na symulowanie powiększonego ruchu sieciowego za pomocą jednoczesnego wykonywania zapytań http(s) – GET, POST z użyciem wielu wątków.
- Ułatwia zbieranie statystyk dotyczących wydajności pracy serwera/grupy serwerów poprzez pomiar czasów przetwarzania zapytań, czasu uzyskiwania odpowiedzi itp.
- Pomaga porównywać wybrane konfiguracje sprzętowo-programowe przy podejmowaniu decyzji o zakupie lub wyborze konkretnego rozwiązania.
- Doskonale nadaje się do testowania RESTful API po http(s) – GET, POST, PUT, PATCH, DELETE.
- Dostarcza możliwość nagrywania i odtwarzania scenariuszy testowych na zasadzie zbliżonej do Selenium IDE [3].
- Daje się włączać w proces CI, tj. integracji ciągłej (ang. *continuous integration*) dzięki współpracy z różnymi systemami, takimi jak Jenkins [4] lub Hudson [5].

Rysunek 1. JMeter – podstawowe elementy



Podstawowe elementy JMeter

Apache JMeter ma budowę hierarchiczną i – co warte podkreślenia – „drzewiastą". Podstawowym elementem jest **TestPlan**, który zawiera co najmniej jedną **GrupęWątków**.

Inne główne elementy to:

- zmienne – wbudowane, definiowane;
- instrukcje warunkowe – IF, WHILE itp.;
- pętle – LOOP, FOREACH itp.;
- zapytania – HTTP(S), JDBC itp.;
- asercje – odpowiedzi, rozmiaru itp.;
- timery – stałe, losowe itp.;
- słuchacze – drzewiaste, sumaryczne itp.;
- ekstraktory – wyrażeń regularnych, XPath itp.;
- menedżery – cookie, header itp.;
- dzienniki – log, console;
- funkcje – wbudowane i definiowane w BeanShell [6].

Wymienione wyżej elementy pozwalają w opinii autora spojrzeć na JMeter jak na uniwersalny język lub framework wysokiego poziomu do tworzenia, wykonywania oraz monitorowania testów automatycznych różnego rodzaju: od niskopoziomowych, wykonywanych bezpośrednio na bazie MySQL, poprzez zapytania http(s) i testowanie RESTful API, aż po nagrywanie i odtwarzanie scenariuszy zorientowanych na działania użytkownika z wykorzystaniem przeglądarki.

Czym NIE jest JMeter?

Należy jasno i wyraźnie podkreślić, że Apache JMeter nie jest przeglądarką. Mimo że wygląda i zachowuje się jak przeglądarka, a właściwie to wiele różnych przeglądarek, oprogramowanie to nie obsługuje wszystkich dostępnych w nich akcji. W szczególności JMeter nie wykonuje JavaScriptów

wstawionych w tagach HTML strony. Co więcej, nie renderuje widoku stron WWW tak jak przeglądarki internetowe. Nie brakuje również ograniczeń dotyczących wspierania WebSocket.

Wsparcie i rozwój

Apache JMeter to otwarte oprogramowanie, które w listopadzie 2011 roku stało się projektem najwyższego poziomu i zyskało społeczność odpowiedzialną za jego rozwój. Dzięki temu powstał serwis *http://jmeter.apache.org/*.

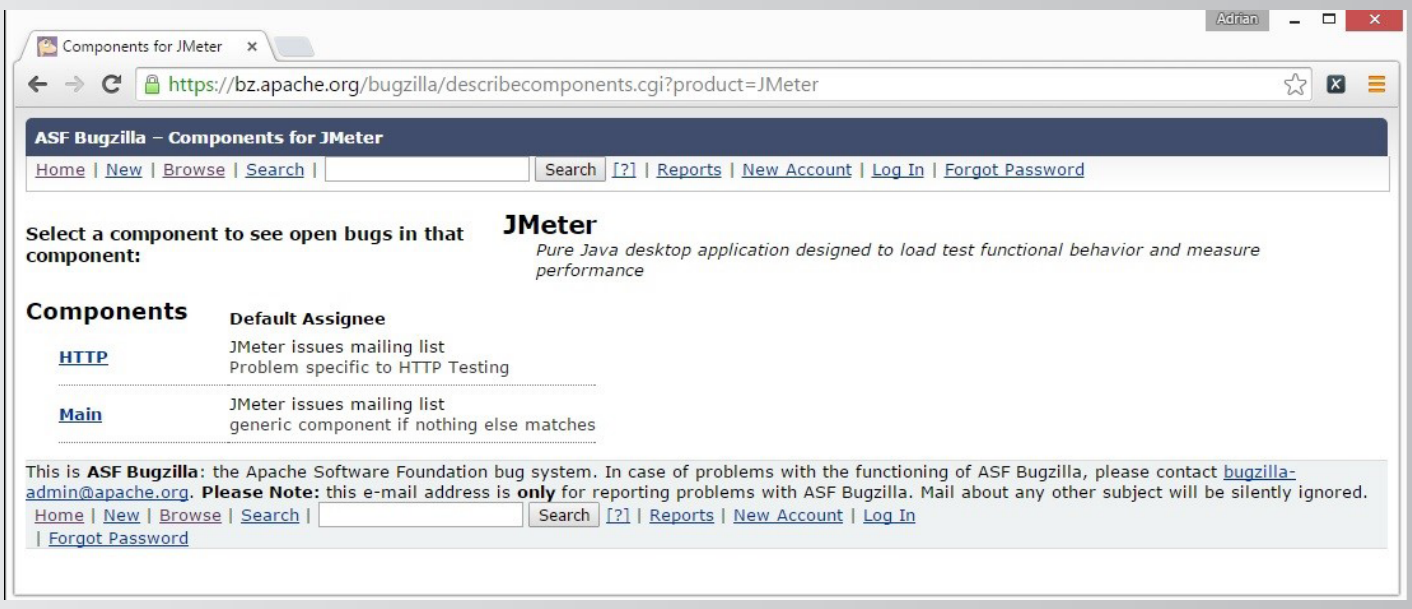
Opracowano też szczegółową dokumentację – *http://jmeter.apache.org/usermanual/index.html* oraz zestaw pomocnych tutoriali. Co więcej, uruchomiony został serwis do zgłaszania defektów (ang. *bug*) i propozycji zmian (ang. *feature request*) do kolejnego wydania (ang. *release*). Przykładem może być m.in. *https://bz.apache.org/bugzilla/show_bug.cgi?id=51817*.

Podsumowanie

JMeter powszechnie jest używany jako narzędzie do tworzenia, wykonywania i monitorowania testów obciążeniowych, wydajnościowych i funkcjonalnych. Bywa również wykorzystywany przy testowaniu rozproszonym (ang. *distributed testing*), testach jednostkowych z JUnit oraz do nagrywania i odtwarzania scenariuszy testowych.

Osobiście autor woli postrzegać i traktować JMeter jak język programowania lub framework do tworzenia testów automatycznych. Wynika to z faktu, że narzędzie to dostarcza wiele funkcjonalności, cech i zalet wysokopoziomowego języka programowania wizualnego, potrafiącego działać wielowątkowo na praktycznie dowolnej architekturze.

Rysunek 2. JMeter – ASF Bugzilla



Wskazówki i podpowiedzi

Ctr ^ SER – jako przydatny skrót od Save, Erase i Run.

Ctr ^ - oraz Ctr + Shift ^ - – zwija i rozwija drzewo Test Planu.

Ctr ^ . – kończy wykonanie wątku/pętli itp.

Nazwy zmiennych są tekstowe, a odwołania do nich wykonujemy poprzez ${ }.

JMeter pozwala na KONFIGURACJĘ (kod, dane, konfiguracja).

Przykładowe wywołanie z linii komend: jmeter -n -t testplan.jmx -l logfile.log.

Dobrze jest, gdy na jeden rdzeń procesora przypadają co najwyżej 2 instancje JMetera.

Zaleca się tworzyć do 1000 wątków na jedną instancję JMetera.

Proces JMetera powinien zajmować zasoby procesora co najwyżej w 75%.

Przydatne są setUp Thread Group oraz tearDown Thread Group.

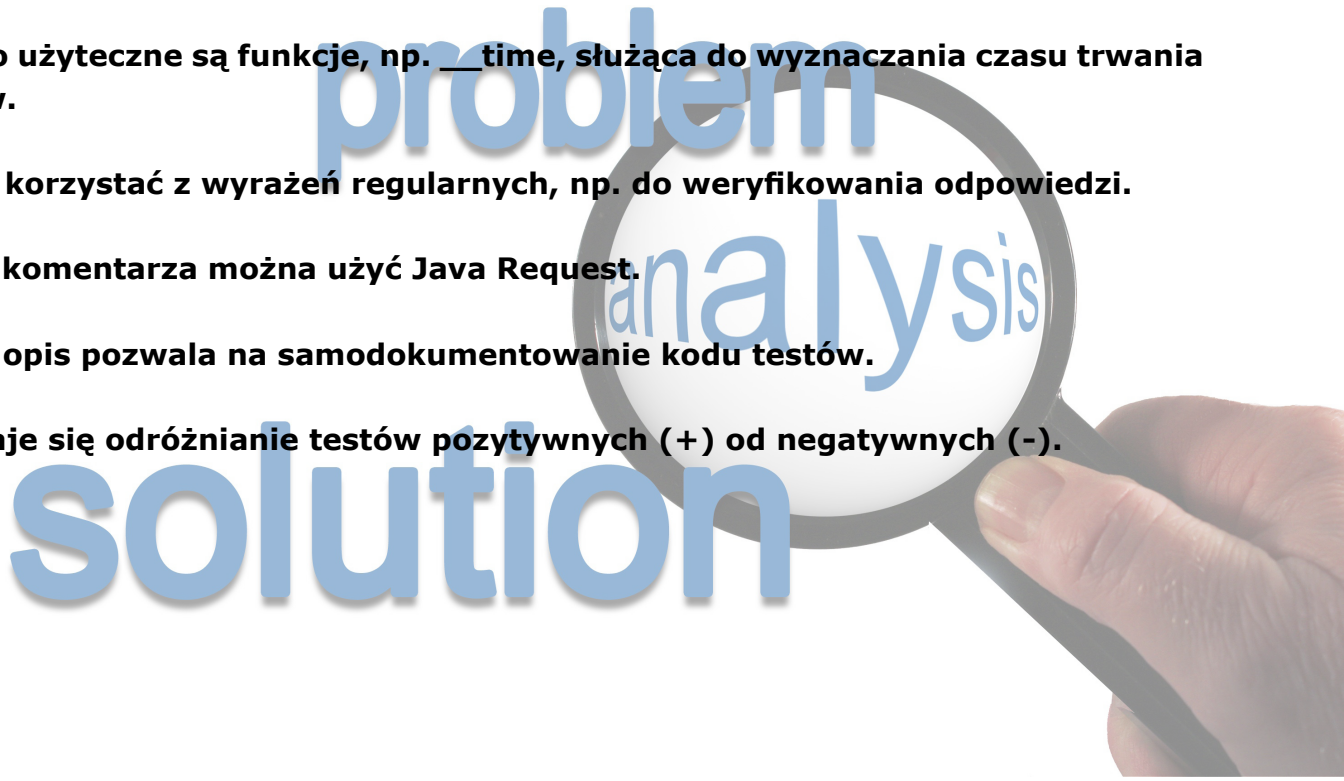
Bardzo użyteczne są funkcje, np. __time, służąca do wyznaczania czasu trwania testów.

Warto korzystać z wyrażeń regularnych, np. do weryfikowania odpowiedzi.

W roli komentarza można użyć Java Request.

Dobry opis pozwala na samodokumentowanie kodu testów.

Przydaje się odróżnianie testów pozytywnych (+) od negatywnych (-).



AUTOR

Adrian Bala

Absolwent informatyki na PJWSTK i UWr. Od blisko 7 lat pracujący jako tester oprogramowania, m.in. w Nasza Klasa, Samsung R&D oraz GFT.

Posiadacz certyfikatu ISTQB. Przeszkolony w zakresie zarządzania testami i zarządzania projektami programistycznymi. Uczestnik kilku konferencji poświęconych testowaniu oraz bezpieczeństwu aplikacji webowych. Gościnnie występował w roli prelegenta, m.in. w Instytucie Informatyki UWr. Zaangażowany w testowanie najbardziej strategicznych produktów NK, np. wersji mobilnych, API i gier, a także firmy Samsung: EPG i Orchestra. Miłośnik otwartego oprogramowania, usability & quality.

Odnośniki w tekście

1. Hypertext Transfer Protocol – HTTP/1.1: https://www.ietf.org/rfc/rfc2616.txt.
2. JDBC driver for MySQL: http://dev.mysql.com/downloads/connector/j/.
3. Selenium IDE: http://www.seleniumhq.org/projects/ide/.
4. Jenkins: https://jenkins-ci.org/.
5. Hudson: http://hudson-ci.org/.
6. BeanShell – Simple Java Scripting: http://www.beanshell.org/manual/bshmanual.html.

Literatura

[1]. Apache JMeter – http://jmeter.apache.org/index.html.
[2]. Apache JMeter Manual – http://jmeter.apache.org/usermanual/index.html.
[3]. ASF Bugzilla – https://bz.apache.org/bugzilla/describecomponents.cgi?product=JMeter.
[4]. Apache JMeter Wiki – http://wiki.apache.org/jmeter/.
[5]. Apache JMeter Group – https://www.linkedin.com/groups?gid=2017104.
[6]. Firebase RESTful API – https://www.firebase.com/docs/rest/api/.
[7]. Apache JMeter workshop by Piotr Sobieraj – materiały szkoleniowe GFT.
[8]. Apache JMeter – automatyzacja testowania – Adrian Bala (opracowanie własne).

Wiktor Żołnowski

Teoria kolejek – matematyka i efektywne wytwarzanie oprogramowania

Wprowadzenie

W ciągu ostatnich kilku lat pomagałem różnym organizacjom jako Agile Coach i niezależny konsultant. Zauważyłem, że często ta sama metoda wytwarzania oprogramowania w niektórych organizacjach i zespołach działa, a w innych nie. Podobnie, zbierając materiały do książki *Mity i Problemy w Agile* podczas organizowanych przez nas spotkań z cyklu Bring Your Own Problem, wielokrotnie słyszałem wypowiedzi uczestników, którzy stosowali podobne podejścia, jednak osiągali zaskakująco różne efekty. Dlaczego tak się działo? Aby poznać odpowiedź na to pytanie, m.in. próbowałem przypomnieć sobie, jak to było, gdy sam pracowałem jako QA i developer w zespołach, które, przynajmniej według mojej subiektywnej oceny, funkcjonowały bardzo sprawnie i efektywnie. Poza oczywistymi różnicami, takimi jak np. to, że w każdym teamie byli inni ludzie oraz działali oni w kontekście różnych organizacji, ciężko było mi wskazać ogólne czynniki mogące realnie decydować o tym, że dana ekipa wykazywała się większą efektywnością i przewidywalnością oraz rozwijała się znacznie szybciej od pozostałych. Odpowiedź pojawiła się niespodziewanie. Otóż szukając wskazówek dotyczących tego, jak policzyć maksymalną wydajność jednego z budowanych przez nas systemów, musiałem odświeżyć sobie wiedzę na temat teorii kolejek i prawa Little'a…

Podstawy teorii kolejek i prawo Little'a

Dlaczego akurat teoria kolejek? Coraz częściej mamy do czynienia z systemami przetwarzającymi duże ilości danych, komunikującymi się z innymi systemami czy też po prostu działającymi asynchronicznie. Stosuje się w nich różnego typu kolejki (mniej lub bardziej bezpośrednio – w końcu funkcjonowanie komputerów, procesorów, RAM-u czy dysków twardych też opiera się na kolejkach). Bazując na danych, które zazwyczaj jesteśmy w stanie zebrać, można – na podstawie wzorów zaczerpniętych z teorii kolejek – oszacować maksymalne obciążenie systemu czy też policzyć czasy przetwarzania dużej ilości danych w zależności od liczby procesów lub wątków. Temat jest bardzo szeroki i poświęcono mu wiele książek oraz publikacji naukowych.

W moim przypadku teoria ta znalazła inne zastosowanie. Takim (nie)typowym systemem kolejkowym jest przecież zespół developerski, do którego spływają wymagania. Te zaś, czy to w postaci karteczek na tablicy, stories w Jirze, czy tasków w Trello, układają się w kolejki.

Pomocne w zrozumieniu tego, dlaczego dany proces/system działa, a inny nie, okazało się również prawo Little'a, mówiące o tym, że średnia liczba rzeczy/klientów w systemie jest równa iloczynowi średniego czasu przebywania w systemie oraz średniego tempa ich przybywania.

Dlaczego Scrum i Kanban (czy jakikolwiek inny proces wytwarzania oprogramowania) działają?

Jako Agile Coach miałem już na koncie kilka udanych wdrożeń Scruma i Kanbana w różnych zespołach. Oczywiście wiedziałem, jak istotne w Scrumie są Sprint Planning oraz Backlog Refinement. Zdawałem sobie również sprawę, że przestrzeganie limitu Work in Process (WIP) to podstawowy warunek efektywnego stosowania Kanbana. Świadomość, co stanowi klucz do działania danego procesu, nie jest jednoznaczna z wiedzą na temat tego, dlaczego akurat te, a nie inne czynniki są tak ważne.

Czy próbowaliście kiedyś przekonać kogoś do stosowania danej metody, stwierdzając jedynie: „Powinniśmy się na nią zdecydować, bo z mojego doświadczenia wynika, że zadziała"? Albo: „Wszyscy tak robią i u nich przynosi to efekt, więc u nas też będzie. Spróbujmy"? Czasami się udaje, ale częściej napotykamy zdrowy sceptycyzm. Ludzka natura często sprawia, że opieramy się temu, czego nie pojmujemy. Rzeczy niezrozumiałe wręcz wzbudzają w nas poczucie strachu przed nieznanym. Wystarczy spojrzeć na naszą mitologię i historię – jeśli nie wiemy, jak i dlaczego coś działa, bardzo szybko przypisujemy temu nadprzyrodzone moce; potrzebujemy wytłumaczenia.

Prawo Little'a jest dobrym wyjaśnieniem tego, dlaczego limit WIP i dobre planowanie w Sprincie są tak ważne. Ale o tym za chwilę.

Artykuł pierwotnie ukazał się w materiałach pokonferencyjnych PyCon 2015.

Zastosowanie teorii kolejek – kilka bardziej przyziemnych przykładów

Zanim przejdziemy do rzeczy, chciałbym, abyście zastanowili się przez chwilę nad odpowiedzią na poniższe pytania dotyczące utylizacji zasobów.

- Jak wygląda autostrada zutylizowana w 100%?
- Jak działa komputer, w którym procesor jest wykorzystany w 100%?
- Jak efektywny jest zespół, którego zasoby czasowe są wykorzystane w 100%?

Pracując ostatnio z pewnym zespołem, zauważyłem, że ten team ma problem z „dowożeniem" tego, co zaplanował w iteracjach. Po kilku Sprintach stało się dla mnie jasne, gdzie tkwi problem. Otóż zespół ten przeważnie zbyt optymistycznie podchodził do Planowania Sprintu. Dodatkowo praktycznie zawsze do Sprintu wpadało coś z poprzedniej iteracji, co było „w połowie skończone". Ten nadmierny optymizm wynikał również z faktu, że zespół praktycznie nie przeprowadzał Backlog Refinementu (Groomingu), więc planowanie polegało na niekończących się dyskusjach o tym, co należy zrobić w danym zadaniu. Zazwyczaj kończyło się to tym, że co najmniej połowa wymagań/zadań polegała na „Researchu". Jak dobrze wiecie, tego typu zadania zazwyczaj nie wnoszą zbyt dużej wartości do produktu. Liczy się to, co jest zaimplementowane i działa. Na pytania, dlaczego zespół nie robi tego „Researchu" podczas Backlog Refinementu (i dlaczego w ogóle pomija Refinement, na który miał przeznaczać do 10% czasu w Sprincie), odpowiedzią było oczywiście: „nie mamy na to czasu – przecież naszym problemem jest dowożenie tego, co zaplanowaliśmy, więc dodatkowe spotkania w Sprincie mogą go jedynie pogłębić". Jak zapewne zauważacie, błędne koło się zamyka.

Udowadnianie bezsensowności takich działań zabrało mi sporo czasu, ale w końcu udało się doprowadzić do sytuacji, w której zespół po raz pierwszy w swojej historii dowiózł wszystko, co zaplanował. W następnym Sprincie przewidziano już czas na Refinement i od tej pory wszystko zaczęło się (oczywiście z kilkoma potknięciami) układać tak jak powinno.

Teoria kolejek a straty w wytwarzaniu oprogramowania

Analiza zastosowania wzorów teorii kolejek i prawa Little'a pozwala szybko wyciągnąć wniosek, że w każdym systemie kolejkowym można wskazać pewien stan (określony przez liczbę zadań wykonywanych w danej chwili), w którym system ten działa najbardziej efektywnie. Podobnie dla każdego zespołu developerskiego da się oszacować (lub raczej podejść do tego eksperymentalnie) optymalną liczbę równolegle wykonywanych zadań.

Praktyka pokazuje, że liczba ta jest zazwyczaj mniejsza od liczby osób w teamie. Może się to wydawać nieintuicyjne. Bo przecież jak to: ktoś nie miałby przypisanego taska? I co będzie robić – przysypiać ze znużenia, oglądać filmiki na Joe Monster? Po pierwsze – inteligentni ludzie nigdy się nie nudzą. Po drugie – jeśli firma zatrudnia kogoś, kto na każdym kroku szuka okazji do oszukiwania pracodawcy, to raczej ma większy problem niż to, że zespół jest mało efektywny.

Mały eksperyment myślowy – czy czekałeś kiedyś, aż ktoś zajmie się Code Review, które mu zleciłeś? Co wtedy zrobiłeś? Pewnie zacząłeś pracę nad kolejnym zadaniem… Gdy tamta osoba akurat znalazła czas na Code Review, byłeś już zajęty, więc cały proces odbył się asynchronicznie poprzez komentarze w kodzie lub, co gorsza, w mailu. Mało tego – wracając do uwag po takim review, musiałeś ponownie zmienić kontekst. Jak bardzo to jest efektywne? Przecież wszyscy wiemy, że Code Review przynosi najlepsze efekty, gdy rozmawiamy o zmianach w kodzie i – jeśli to możliwe – od razu wprowadzamy poprawki, wspólnie poprawiamy jakość tego, co zrobiliśmy. Ale jak to osiągnąć, gdy wszyscy są skupieni na swoich zadaniach i trzeba czekać, aż znajdą wolną chwilę? Albo Ty będziesz czekać, albo oni będą musieli zmienić kontekst. W obu przypadkach któreś z wykonywanych zadań utknie w kolejce. W danym momencie nie będzie wykonywane. Takie zjawisko nazywamy w Leanie przestojem produkcyjnym. To jeden z głównych czynników generujących straty w procesie wytwórczym.

Czy nie byłoby łatwiej, gdybyś zawsze miał kogoś, kto nie pracuje w danym momencie nad niczym konkretnym i może bardzo szybko przejrzeć zmiany, które wprowadziłeś? Cóż, nie wiem, jak będzie w przypadku Twojego zespołu, ale z pewnością warto spróbować i zmierzyć wyniki takiego eksperymentu.

Trochę matematyki

Prawo Little'a opisuje się zazwyczaj wzorem:

*Średni czas przetwarzania = Liczba elementów w systemie * częstotliwość napływania nowych elementów do systemu.*

W kontekście Kanbana i Lean Software Development (a także w Lean Manufacturing) przyjmuje się często (nie do końca poprawne – o tym za chwilę) założenie, że w stabilnym systemie częstotliwość napływania do niego nowych elementów jest równa częstotliwości, z jaką elementy go opuszczają.

Czyli możemy przyjąć, że:

*Średni czas przetwarzania = WIP Limit * średnia częstotliwość, z jaką zadania opuszczają system.*

Gdy przekształcimy to równanie, okaże się, że:

Średnia częstotliwość, z jaką zadania opuszczają system = Średni czas przetwarzania / WIP Limit

i

WIP Limit = Średni czas przetwarzania / średnia częstotliwość, z jaką zadania opuszczają system.

Jak widzicie, za pomocą tych wzorów możemy określić, jaki powinien być limit WIP. Musimy jedynie znać średni czas przetwarzania i średnią częstotliwość, z jaką zadania opuszczają system.

Wydaje się to proste. Dlaczego więc na co dzień nie mamy do czynienia z superefektywnymi zespołami? Dlaczego tak wiele fabryk jest nieefektywnych? Dlaczego system zutylizowany w 100% okazuje się nieefektywny?

Odpowiedź jest prosta i wynika z błędnego założenia, że:

- wszystkie zadania mają dokładnie taką samą wielkość, więc nie tworzą się kolejki;
- nie występują czynniki losowe, które wytrącają nasz system ze stabilnego stanu.

Czy możemy zatem rozwiązać ten problem?

„Proste" rozwiązanie

Wielu managerów i dyrektorów oczekuje od swoich podwładnych tego, że będą wykazywali inicjatywę, że będą innowacyjni, że będą się rozwijać i usprawniać całą organizację – czasami pytam tylko po cichu, kiedy mają to robić, skoro ich czas jest zaalokowany w 100%?

Jak zatem zoptymalizować proces wytwarzania oprogramowania w każdej organizacji? Kilkukrotnie już udało mi się z powodzeniem wykorzystać pewną prostą metodę, składającą się z kilku kroków:

- Mierzymy przez określony okres czasu (np. Sprint), ile mniej więcej zespół jest w stanie skończyć zadań, Story Pointów albo czegoś innego mierzalnego (czyli Velocity w Scrumie – tzw. pojemność zespołu). Co ważne, interesuje nas tylko to, co jest zrealizowane, a nie „rozgrzebane".
- W następnej iteracji planujemy 80% tego, co udało się skończyć w poprzedniej. Jeśli zdołamy zrealizować wszystkie zaplanowane zadania przed końcem Sprintu, dobieramy kolejne, aż skończy się iteracja.
- W kolejnej iteracji powtarzamy poprzednie kroki. Ponownie bierzemy 80% tego, co udało się skończyć w ostatnim Sprincie.

Logika podpowiada Wam zapewne, że w ten sposób z iteracji na iterację zespół będzie kończył coraz mniej zadań. Praktyka pokazuje jednak, że zazwyczaj te 80% rzeczy udaje się zrealizować przed końcem Sprintu i zespół dobiera kolejne zadania. W efekcie często skutkuje to tym, że w kolejnych iteracjach team kończy mniej więcej tyle samo, co w poprzednich. Czasem zdarza się, że uda się zrealizować więcej zadań i średnie Velocity stopniowo rośnie.

Podsumowanie

Powiecie zapewne: „U mnie to nie zadziała", „W naszej organizacji tak się nie da", „Nikt tego nie zrozumie", „Ktoś cały czas każe nam brać więcej" etc.

Jeśli ktoś każe Wam brać więcej do Sprintu, zadajcie mu proste pytanie: „Czy chcesz, żebyśmy więcej brali, czy może jednak wolisz, żebyśmy więcej dowozili?".

Na zakończenie trochę przemyśleń. Nieefektywność zespołu czy firmy stanowi moim zdaniem problem kulturowy, wynikający z braku podstawowej wiedzy na temat zarządzania, teorii kolejek i teorii ograniczeń (Theory Of Constraints) na wysokich poziomach w organizacji (często

AUTOR

Wiktor Żołnowski

Trener, coach, konsultant zajmujący się wdrażaniem praktyk jakościowych oraz Zwinnych Metodyk Zarządzania Projektami (Agile) w firmach z branży IT. Po pracy – bloger, twórca pierwszego w Polsce bloga o jakości oprogramowania – *blog.testowka.pl*.

Od kilku lat Wiktor jest trenerem i doradcą w dziedzinie zarządzania projektami i wdrażania procesów jakościowych oraz automatyzacji testów. W pracy tej wykorzystuje wiedzę praktyczną i nawiązuje do realnych przykładów i problemów, jakie napotkał w trakcie swojej kariery.

tych najwyższych). Nie jest to nic nowego – dokładnie te same błędy popełniano od czasów rewolucji przemysłowej. Mam nadzieję, że teoria kolejek pomoże Wam, drodzy Czytelnicy, szerzyć wiedzę i racjonalne podejście do wytwarzania oprogramowania i zarządzania organizacją. I pamiętajcie – nawet jeśli jesteście „tylko" developerami, wyłącznie od Was zależy, jak będziecie pracować, jakich narzędzi używać i na co pozwolicie swojemu biznesowi. Pamiętajcie, że organizację zawsze można zmienić – albo można zmienić organizację!

Literatura

1. Eliyahu M. Goldratt, Jeff Cox, *The Goal: A Process of Ongoing Improvement.*
2. Wprowadzenie do teorii kolejek: https://www.youtube.com/watch?v=rSZWig173xM.
3. WIP limit i prawo Little'a: https://www.youtube.com/watch?v=VU8TUSnQ-vw.
4. Lean i teoria kolejek: https://dzone.com/articles/lean-tools-queuing-theory.
5. Tablice wzorów w teorii kolejek: http://irh.inf.unideb.hu/user/jsztrik/education/09/english/3f.html.
6. Kalkulator przydatny w eksperymentach nad teorią kolejek: http://www.supositorio.com/rcalc/rcalclite.htm.
7. Wstęp do teorii kolejek: http://www.referenceforbusiness.com/encyclopedia/Pro-Res/Queuing-Theory.html.
8. Zarządzanie kolejkami: http://less.works/less/principles/queueing_theory.html.

Remigiusz Dudek

Cardinal sin

Wiele zostało już napisane na temat antywzorców w świecie testów automatycznych. James Carr i jego *TDD Anti-Patterns* to lektura obowiązkową dla każdego, kto chce uniknąć podstawowych błędów. W tym artykule skupię się na mniej oczywistych błędach, decyzjach, które w momencie podejmowania wydają się rozsądne lub też zupełnie niegroźne, a w rzeczywistości prowadzą do nieoczekiwanych rezultatów. Całość została ujęta w ramy siedmiu grzechów głównych, choć – jak się przekonacie – błędów (podobnie jak grzechów) jest znacznie więcej.

Obżarstwo

Już papież Grzegorz I wyszczególnił 5 dróg, które wiodą do popełnienia grzechu obżarstwa… Podążę tym samym tropem.

a) Praepropare – zbyt wczesne testowanie. Wyobraźmy sobie system, którego rolą jest wysyłanie raportów na podstawie wiadomości wejściowych. Raport zawiera identyfikator wiadomości wejściowej, która stanowi jego katalizator. Operacja wysyłania raportów jest oczywiście asynchroniczna względem wykonania testu, toteż musimy poczekać na odpowiedź. Test możemy skonstruować zatem na dwa sposoby (kod 1).

Pierwsze podejście jest nieco bardziej czytelne od drugiego, zatem zadajmy sobie

Kod 1.

```
@Test
public void shouldGenerateReportCorrespondingToInboundMessage() {
    InboundMessage message = new InboundMessage(id, legalEntity);

    sender.send(message);

    await().atMost(TEN_SECONDS).until(appropriateReportIsReceived(message));
}

private Runnable appropriateReportIsReceived(InboundMessage message) {
    return () -> reportListener.listenFor(aReport()
                                    .thatHasId(message.getId())
                                    .thatIsAgainstLegalEntity(message.getLegalEntity())
   );
}

@Test
public void shouldGenerateReportCorrespondingToInboundMessage_2() {
    InboundMessage message = new InboundMessage(id, legalEntity);

    sender.send(message);

    await().atMost(TEN_SECONDS).until(correspondingReportIsReceived(message));
    Report report = reportListener.getReport(id);

    assertThat(report, is(aReport().thatIsAgainstLegalEntity(legalEntity)));
}

private Runnable correspondingReportIsReceived(InboundMessage message) {
    return () -> reportListener.listenFor(aReport().thatHasId(message.getId()));}
```

pytanie, czy istnieje powód, dla którego poświęcilibyśmy czytelność testu. Jednym z głównych powodów inwestowania w automatyzację testów jest szybkość uzyskiwania informacji zwrotnej. Rozważmy raport wygenerowany dla danej wiadomości wejściowej w sytuacji, gdy ma on złe *legalEntity*:

1. W przypadku pierwszego podejścia o tym fakcie dowiemy się dopiero po upływie zadeklarowanego czasu oczekiwania (w naszym przypadku 10 s). Dodatkowo informacja o błędzie będzie bardzo ogólna, dowiemy się jedynie, iż raport o zadanym id oraz legalEntity nie pojawił się w oczekiwanym czasie – zwróćmy uwagę na to, iż jest to in-

formacja nie tylko zbyt ogólna, ale co gorsza – wprowadzająca w błąd naszą intuicję.
2. W podejściu drugim o błędzie dowiemy się zaraz po otrzymaniu raportu i będzie on bardzo dokładnie zidentyfikowany.

Dodatkowo czytelność drugiego rozwiązania możemy nieco poprawić poprzez enkapsulację oczekiwania na raport w klasie ReportListener.

b) Laute – zbyt drogie testowanie. Ogromna bolączka zwinnych testów, choroba, którą zwykliśmy diagnozować, dopiero gdy jest już w bardzo zaawansowanym stadium. Dobrą regułą, pozwalającą szybko stwierdzić, iż choroba właśnie się zaczęła, a także umożliwiającą wyciągnięcie maksimum korzyści z inwestycji w testy automatyczne, jest uruchamianie kompletnego zbioru testów po każdej zmianie kodu źródłowego. Musimy sobie zdawać sprawę z tego, iż często optymalizacja szybkości wykonania testu może nieść ze sobą zmianę kodu produkcyjnego.

1. Pierwszą, najważniejszą regułą optymalizacji kosztów testowania jest umiejscowienie testu na odpowiednim poziomie automatycznej piramidy testów, przy czym złota reguła piramidy brzmi: im niżej tym taniej. Oczywiście nie każdą funkcjonalność będziemy w stanie przetestować na poziomie testów jednostkowych.
2. Wyobraźmy sobie, iż nasza aplikacja wykonuje jakąś czynność periodycznie. Zaimplementowaliśmy scheduler, który odpala się zgodnie z cronowym harmonogramem. Aby sensownie integracyjnie przetestować taką funkcjonalność, możemy zbudować część kontekstu dookoła klasy wyzwalanej przez scheduler (bez samej funkcjonalności automatycznego odpalania) i uruchomić jej odpowiednią metodę publiczną. Co możemy jednak zrobić, gdy chcemy przetestować integrację tej funkcjonalności z pozostałymi modułami naszego systemu? Zmienić kod produkcyjny w dwojaki sposób:
   - dodać interfejs (jmx, rest, rmi, etc.) pozwalający na uruchomienie owej funkcjonalności w trybie „na żądanie";
   - dodać możliwość wyłączenia schedulerów.
3. Wyobraźmy sobie, iż nasz system wysyła raporty dla określonych wiadomości wejściowych, acz są też takie wiadomości, dla których nasz system nie wysyła żadnych raportów… Jak automatycznie przetestować fakt, iż raport nie został wysłany? Najprostszy, brutalny sposób to oczywiście zaimplementować timeout w naszym teście. Domyślnie stwierdzamy: „jeśli do tej pory nic się nie zdarzyło, to znaczy, że się już nie zdarzy". Taki test szkodzi całemu zbiorowi testów w dwojaki sposób:
   - Znacząco wydłuża czas trwania testów (za każdym razem musimy czekać cały timeout).
   - Potencjalnie wprowadza „migotanie testów" (test czasem przejdzie, czasem nie). Wyobraźmy sobie prostą sytuację, gdy nasz serwer ciągłej integracji jest bardzo obciążony i nagle system zaczyna zwalniać, w efekcie dla zadanej wiadomości wejściowej wyśle błędnie raport, ale zazwyczaj dopiero po timeoucie i tylko czasem uda mu się zrobić go szybciej. W ten sposób ludzie przyzwyczają się, że jest jeden test, który „migocze",

więc… trzeba go uruchomić jeszcze raz, bo przecież zmiany z czerwonym testem nikt nam nie pozwoli włączyć do mastera/trunka.

Odpowiedzią na przedstawiony problem jest kolejna złota reguła testowania automatycznego: **nie można udowodnić/przetestować że coś się nie stało** (musielibyśmy czekać *nieskończenie* długo… a to trochę za długo). Jedyny sensowny sposób na wybrnięcie z tego impasu stanowi przetestowanie, że coś się stało ;-). To może wymagać sporych zmian w samym systemie (celem jest poprawienie testowalności). Jakkolwiek skomplikowane technicznie by to nie było, koncepcyjnie sprowadza się do bardzo prostej rzeczy: test musi „wiedzieć", iż procesowanie danej wiadomości się skończyło i istnieje jakiś marker, na który musi czekać.

c) Nimis /Ardenter – testowanie w zbyt dużej ilości, zbyt dokładne. Intuicyjnie może się nam wydawać, że im więcej przetestujemy, tym lepiej… Nic bardziej mylnego. Zrobimy sobie potencjalnie krzywdę na trzy sposoby:

1. Tracimy czytelność kontraktu, jaki nasza klasa zawiera ze swoimi kolaboratorami.
2. Tracimy czas na utrzymywanie większej ilości kodu.
3. Wydłużamy czas trwania naszego zbioru testów. Jeśli dopuścimy się takiej nierozwagi na poziomie jednostkowym, owo wydłużenie będzie zazwyczaj marginalne, lecz wystarczy wejść po stopniach piramidy testów nieco wyżej, by czas oczekiwania na testy znacząco się wydłużył.

Co to zatem znaczy „zbyt dużo" i jak się przed tym uchronić? Stare, dobre praktyki wypracowane w czasach intensywnych testów manualnych przychodzą nam z pomocą: klasy równoważności, wartości brzegowe, pokrycie diagramu przepływów, tabele decyzyjne. Opis tych metod wykracza poza ramy tego artykułu, ale też jest dużo dokumentacji na ten temat w sieci. Chciałbym się skupić na czymś, czego w internecie jeszcze nie znalazłem, czyli na sposobie poradzenia sobie z testami automatycznymi skomplikowanych warunków logicznych. Z jednej strony rozwiązanie wydaje się dziecinnie proste: wystarczy użyć testów parametrycznych i przetestować cały zbiór kombinacji… Z drugiej zaś, będąc tak dokładni, całkowicie zaciemniamy obraz odpowiedzialności naszej klasy. Musimy zatem wypracować sensowny kompromis. Rozważmy następującą, w sumie dość prostą implementację, która bazuje decyzję na podstawie 4 parametrów (kod 2).

By dokładnie przetestować powyższy kod, musimy pokryć 16 (2^4) przypadków (kod 3).

Kod 2.

```
public boolean isEligible(InboundMessage message) {
    LegalEntity legalEntity = message.getLegalEntity();
    Counterparty counterparty = message.getCounterparty();
    PartiesRelation partiesRelation = relationService.determineRelation(legalEntity, counterparty);
    Product product = message.getProduct();

    return legalEntity.isEligible() &&
            counterparty.isEligible() &&
            partiesRelationship.isEligible() &&
            product.isEligible(
}
```

Kod 3.

```
public Object[] eligibleParameters() {
    return $(
        $(eligibleLegalEntity, eligibleCounterparty, eligibleProduct, eligibleRelationship, true),
        $(eligibleLegalEntity, eligibleCounterparty, eligibleProduct, notEligibleRelationship, false),
        $(eligibleLegalEntity, eligibleCounterparty, notEligibleProduct, eligibleRelationship, false),
        $(eligibleLegalEntity, eligibleCounterparty, notEligibleProduct, notEligibleRelationship, false),
        $(eligibleLegalEntity, notEligibleCounterparty, eligibleProduct, eligibleRelationship, false),
        $(eligibleLegalEntity, notEligibleCounterparty, eligibleProduct, notEligibleRelationship, false),
        $(eligibleLegalEntity, notEligibleCounterparty, notEligibleProduct, eligibleRelationship, false),
        $(eligibleLegalEntity, notEligibleCounterparty, notEligibleProduct, notEligibleRelationship, false),
        $(notEligibleLegalEntity, eligibleCounterparty, eligibleProduct, eligibleRelationship, false),
        $(notEligibleLegalEntity, eligibleCounterparty, eligibleProduct, notEligibleRelationship, false),
        $(notEligibleLegalEntity, eligibleCounterparty, notEligibleProduct, eligibleRelationship, false),
        $(notEligibleLegalEntity, eligibleCounterparty, notEligibleProduct, notEligibleRelationship, false),
        $(notEligibleLegalEntity, notEligibleCounterparty, eligibleProduct, eligibleRelationship, false),
        $(notEligibleLegalEntity, notEligibleCounterparty, eligibleProduct, notEligibleRelationship, false),
        $(notEligibleLegalEntity, notEligibleCounterparty, notEligibleProduct, eligibleRelationship, false),
        $(notEligibleLegalEntity, notEligibleCounterparty, notEligibleProduct, notEligibleRelationship, false)
    );
}

@Test
@Parameters(method = „eligibleParameters”)
public void eligibilityShouldBeAsExpected(LegalEntity entity, Counterparty counterparty,
                    Product product, PartiesRelationship partiesRelationship,
                    boolean expectedResult) {
    (...)
}
```

Bardzo trudno z testu przedstawionego w *kodzie 3.* odczytać, jaki „kontrakt" zawiera klasa testowana ze swoimi kolaboratorami, choć jednocześnie możemy się bronić tym, iż jest to bardzo dokładny kontrakt, w którym wszystko zostało zawarte. Tchnijmy nieco pragmatyzmu do powyższego podejścia (kod 4).

W ten sposób zamieniliśmy 16 testów na 5. Zwróćmy ponadto uwagę na to, iż przyrost przypadków testowych w funkcji liczby parametrów wejściowych w pierwszym podejściu jest geometryczny (z każdym nowym parametrem podwajamy liczby przypadków testowych) natomiast w podejściu drugim – liniowy (z każdym nowym parametrem dodajemy jeden nowy test). Poniżej zestawienie liczby przypadków testowych odpowiednio w podejściu „dokładnym" oraz „pragmatycznym" dla określonej liczby parametrów wejściowych:

Liczba parametrów wejściowych	**Liczba przypadków testowych (podejście dokładne)**	**Liczba przypadków testowych (podejście pragmatyczne)**
1	2	2
2	4	3
3	8	4
4	16	5
5	32	6
10	1024	11

Kod 4.

```
@Test
public void shouldBeEligibleWhenAllCompoundsAreEligible() { (…) }

private Object[] atLestOneCompoundNotEligible() {
    return $(
            $(eligibleLegalEntity, eligibleCounterparty, eligibleProduct, notEligibleRelationship),
            $(eligibleLegalEntity, eligibleCounterparty, notEligibleProduct, eligibleRelationship),
            $(eligibleLegalEntity, notEligibleCounterparty, eligibleProduct, eligibleRelationship),
            $(notEligibleLegalEntity, eligibleCounterparty, eligibleProduct, eligibleRelationship)
    );
}

@Test
@Parameters(method = „atLestOneCompoundNotEligible")
public void shouldNotBeEligibleWhenAtLeastOneCompoundIsNotEligible(
                LegalEntity legalEntity, Counterparty counterparty,
                Product product, PartiesRelationship partiesRelationship) { (…) }
```

Zmiana charakterystyki z geometrycznej na liniową wiąże się oczywiście z ceną – tracimy dokładność. Zadajmy sobie jednak pytanie, czy jest to autentyczny problem. Poniżej trzy najważniejsze wymagania, jakie stawiamy testom automatycznym:

1. Służą jako testy regresji, mają być naszą siatką bezpieczeństwa przy refaktoryzowaniu kodu.
2. Dają szybką informację zwrotną.
3. Powinny jasno oddawać odpowiedzialności klasy i w efekcie ułatwiać refaktoryzowanie kodu.

Ważną rzeczą jest uświadomienie sobie, iż w sytuacji, gdy mamy dość wysokie pokrycie kodu/wymagań testami, często błędy, które wydają się błędami regresji, tak naprawdę są związane z niewłaściwym zrozumieniem nowych wymagań. Inżynier, implementując nowe wymaganie, zmienia również test – to normalna praktyka. Przed tego typu błędem nie uchroni nas ani podejście „dokładne", ani „pragmatyczne". Pozostają zatem modyfikacje dokonane przypadkowo. Zobaczmy zatem, jaką zmianę trzeba wprowadzić, aby rozwiązanie „pragmatyczne" okazało się zbyt liberalne:

Tabelkę podzieliłem na trzy sekcje (pierwszy rząd służy tylko za punkt odniesienia, który pokazuje oryginalny warunek logiczny).

Pierwsza (żółta) część tyczy się błędów, które nie zostaną wykryte w żadnym z podejść:

1. Zmiana operatora AND z logicznego na bitowy. W przypadku wartości boolowskich wynik jest taki sam, więc trudno tu mówić o autentycznym błędzie – jest to natomiast na pewno mylące, narzędzia do statycznej analizy kodu powinny to wychwycić.
2. Dodanie nowego warunku (f) – z testowego punku widzenia to dodanie nowej funkcjonalności i powinno wiązać się z dodaniem/zmianą testu, tak by pokrywał tę nową funkcjonalność. Jak już wspominałem, testy automatyczne są testami regresji, więc to niemożliwe, aby znalazły tego typu błąd, jeśli tylko zmiana jest kompatybilna wstecz.

Druga część (zielona) to faktycznie przypadkowe błędy, które możemy popełnić, jeśli szybko edytujemy kod (np. usunięcie jednego z parametrów, przypadkowa

Warunek logiczny	**Rezultat testów pragmatycznych**	**Rezultat testów dokładnych**
a && b && c && d && e	Ok	Ok
A & b && c && d && e	Błąd niewykryty	Błąd niewykryty
A && b && c && d && e && f	Błąd niewykryty	Błąd niewykryty
A && c && d && e	Błąd wykryty	Błąd wykryty
A | b && c && d && e	Błąd wykryty	Błąd wykryty
A && b && c && d && e || (a && !b)	Bład niewykryty	Błąd wykryty

zmiana operatora AND na OR etc.) – tego typu usterki są wykrywane przez oba testy.

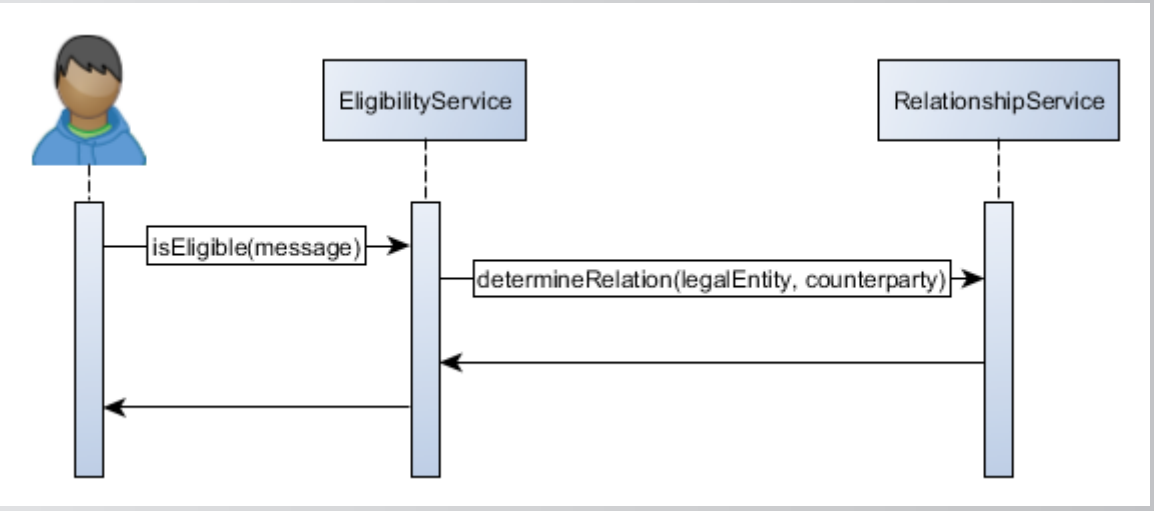
Trzecia część (czerwono-zielona) faktycznie obrazuje, jak skomplikowana (wybrałem możliwie najprostszą) musi być (przypadkowa) zmiana w warunku logicznym, aby test pragmatyczny zawiódł. Teraz musimy sobie tylko odpowiedzieć na pytanie, jakie jest prawdopodobieństwo wprowadzenia takiej zmiany przypadkowo… i czy chcemy ponosić cenę całkowitego pokrycia, aby się przed nią ustrzec.

d) Studiose – zbyt wystawne testowanie. Dzieje się to w sytuacji, gdy do przetestowania naszej klasy wymagamy szeregu mocków, stubów etc. Najprostsze rozwiązanie problemu zbyt dużej liczby mocków to… niestosowanie ich. Podejście funktorowe przychodzi nam z odsieczą. Jego implementacja będzie jednak wymagać refaktoryzowania kodu. Wykorzystajmy ponownie algorytm do wyznaczania eligibility. Wygląda on tak (rys. 1).

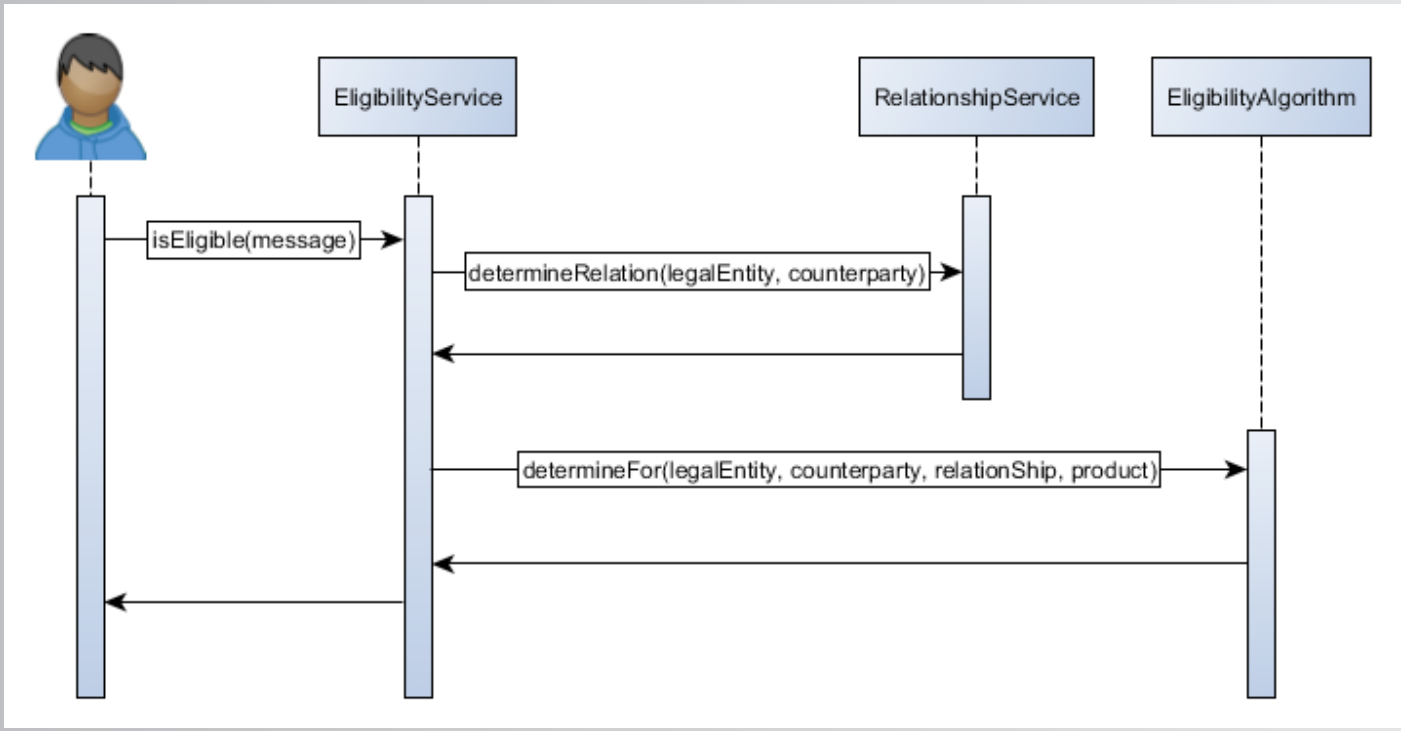
Pozostaje pytanie, czy potrafimy dokonać takich zmian, aby mockowanie nie było potrzebne w powyższej sytuacji. W swojej aktualnej formie EligibilityService łamie zasadę pojedynczej odpowiedzialności. Wie, skąd uzyskać potrzebne mu informacje (RelationshipService) oraz jak wyznaczyć eligibility na podstawie posiadanych danych. Proponuję rozdzielić te dwie odpowiedzialności na część serwisową oraz algorytmiczną. Serwis zbierze dane (bardzo prosta logika, w zasadzie to jej brak – nie bardzo jest co testować) oraz algorytm (tu tkwi serce, które musimy należycie przetestować), który te dane przetworzy i wyda odpowiedni werdykt, a następnie serwis zwróci ten werdykt na zewnątrz (rys. 2).

Kolejny bardzo prosty sposób na ustrzeżenie się mocków to wprowadzenie tzw. biblioteki obiektów testowych – obiektów biznesowych będących w odpowiednich stanach. Istnieje wiele różnych typów zamienników, które stosujemy w testach, jednak ich dokładny opis wykracza poza

Rysunek 1.



Rysunek 2.



ramy tego artykułu (zainteresowanego czytelnika odsyłam do sieci: *Mocks aren't Stubs by Martin Fowler, Mocks, Fakes, Stubs and Dummies from xUnitPatterns*). Tomasz z Akwinu dodał jeszcze jedną ścieżkę do zatracenia w kontekście grzechu obżarstwa.

e) Forente – testowanie niepohamowane, zdziczałe. Na wstępie tego krótkiego punktu zaznaczę dwie rzeczy:

- Testy powinny opisywać biznesowy kontrakt, jaki klasa zawiera ze swoimi kolaboratorami, a także sprawdzać z możliwie biznesowego punktu widzenia, czy zachowanie klasy jest poprawne.
- Testy nie powinny być splątane z kodem (coupled) ponieważ dramatycznie utrudnia to refaktoryzowanie, które stanowi nieodłączny aspekt utrzymywania czystości kodu.

Łamiemy obie powyższe reguły, gdy wykorzystujemy pewne funkcjonalności frameworków do mockowania, takie jak np. sprawdzanie kolejności wywołania poszczególnych metod, kontrolowanie, czy dana metoda została wywołana, sprawdzana, czy parametry wywołania były odpowiednie. Zwróćmy uwagę, że taki test ani nie odpowiada na pytanie o odpowiedzialności biznesowe klasy, ani nie pomoże w przypadku refaktoringu, bo i tak będzie musiał zostać zmieniony. Zatem po co go wykorzystywać?

Chciwość

Chciwość w świecie testów zazwyczaj manifestuje się w testach systemowych, pożądających dostępu do zasobów, do których dostępu mieć nie powinny. Zgodnie z ideą BDD test systemowy, poza tym, że sprawdza, czy nasz system zachowuje się w odpowiedni sposób, jest również swego rodzaju biznesową dokumentacją. Aby to stwierdzenie było prawdziwe, taki test musi wykorzystywać tylko interfejsy, które są rozumiane przez ludzi biznesu. Jeśli nagle, by sprawdzić, czy system po wykonaniu testu jest w odpowiednim stanie, test sięga do bazy danych – oto chciwość, o jakiej mówię. Przyczyna? Być może system jest nietestowalny i to jedyny sposób na przetestowanie go, ale z drugiej strony trudno wymagać od użytkowników (z definicji akceptujących każdą dostarczaną im historyjkę), że będą w pełni zrozumieli test akceptacyjny, który w połowie wykonania sięga do bazy danych lub kończy się stwierdzeniem: „wszystko działa poprawnie, w tabelach Users oraz Reports są odpowiednie wpisy"…

Lenistwo

Testy potrafią być leniwe w dwojaki sposób.

Po pierwsze, same zostawiają po sobie bałagan, system w zabrudzonym stanie. Z czasem może się okazać, iż nowo dodany test, który przechodzi w izolacji, zawsze gdy zostaje uruchomiony na serwerze ciągłej integracji, zaczyna nam migać (czasem przechodzi, czasem nie). Otóż określona kombinacja brudów pozostawiona przez poprzednie testy interferuje z tym nowo dodanym. To jeden z najtrudniej identyfikowalnych błędów, ponieważ może pojawiać się tylko czasem, w zależności od kolejności wywołania testów poprzedzających nasz test (kolejności, na którą nie mamy wpływu, zwłaszcza gdy uruchamiamy testy wielowątkowo).

Druga manifestacja lenistwa jest jeszcze gorsza niż poprzednia – celowe wykorzystanie zanieczyszczeń zostawionych przez poprzedni test. Jeśli zdecydujemy się iść tą drogą, musimy zdawać sobie sprawę z tego, że zakładamy sobie na nogi łańcuchy komplikacji. Od tego momentu nie będziemy już w stanie jasno wywnioskować startowego stanu systemu, czytając jeden test, a być może, co gorsza, powstanie efekt domina: jeśli jeden test się nie powiedzie, pociągnie za sobą inne testy, a te pociągną kolejne. To znacząco utrudnia analizę. Kajdany zaczynają również dźwięczeć, gdy przyjdzie nam zmienić funkcjonalność (a zatem również test), który „przygotowywał" stan dla jakiegoś innego testu…

Zazdrość

Zazdrość manifestuje się zazwyczaj w testach wyższego poziomu. Ogólnie rzecz biorąc, sprowadza się to do zazdroszczenia innym testom, że mogą coś sprawdzać. Wyobraźmy sobie test systemowy, którego celem jest skontrolowanie, czy dla określonej wiadomości wejściowej został wygenerowany raport. W takim teście poza sprawdzeniem, czy faktycznie raport został wygenerowany, weryfikujemy jeszcze dokładnie jego zawartość… Czy to dobrze?

Dlaczego miałoby to być złe? Otóż zaciemniamy obraz odpowiedzialności tego konkretnego testu (problem możemy obejść poprzez ukrywanie tego, co faktycznie jest kontrolowane, na którymś z niższych poziomów abstrakcji naszego frameworku testowego). Tego rodzaju sprawdzenia zachęcają do testowania wariacji prowadzących do wygenerowania raportu z innymi wartościami na poziomie testów systemowych. To już jest poważne naruszenie, ponieważ znacząco wydłuży czas trwania testów systemowych.

Kenodoxia

W IV wieku naszej ery grecki mnich Evagrius Ponticus zdefiniował listę „8 złych myśli", które to następnie niespełna dwa wieki później papież Grzegorz I przekuł w „7 śmiertelnych grzechów". Na liście „siedmiu" Kenodoxia (czy też pycha) nie widnieje (znajduje się tam jedynie duma, choć, co ciekawe, w katolickim katechizmie jest odwrotnie: pycha zastąpiła dumę). Niemniej jednak ze względu na to, iż objawiła się ona ostatnio w testach napisanych przez mój zespół, postanowiłem nie pomijać jej w tym artykule. Jedyne miejsce, w którym test może chwalić się swoimi poczynaniami, to logi testowe. Zarówno logi testowe, jak i raport z wykonania testów powinny być łatwe do odczytu i analizy. Jeśli jest inaczej, musimy popracować nad poziomem logowania.

Dwa ostatnie punkty nie tyczą się testów, lecz raczej inżyniera, który je pisze.

Duma

Dobrze jest znać różne języki programowania, ale to nie znaczy, że należy ich wszystkich używać w swoich testach czy też ogólnie w projekcie. W moim projekcie posługujemy się 3 językami (Java, Groovy, XQuery, brakuje tylko Scali i Clojure – acz i takie zapędy były ;-)) i bardzo się staramy, aby istniały jasne granice stosowania każdego z nich. W przypadku testów udało nam się sprowadzić tę liczbę do dwóch (Java, Groovy). Każdy język to nowe praktyki kodowania, nowa definicja statycznej analizy kodu, nowe rodzaje błędów do popełnienia. Poważnie się zastanówcie, zanim zrobicie sobie krzywdę wprowadzeniem drugiego języka do Waszego projektu. Określcie konkretne zyski, jakie planujecie uzyskać, bo cena jest wysoka.

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.

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 – to po prostu inne narzędzia służące do rozwiązywania odmiennych 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 zachęcają ludzi biznesu do czytania dokumentacji testującej kod.

W wolnych chwilach dzieli się swoimi przemyśleniami na blogu.

Gniew/pożądanie

Zdarza się, że już po prostu chcemy, żeby ktoś wreszcie zmerge'ował nasze zmiany, bo przebyliśmy dwukrotnie „rebasing hell", a ktoś się czepia: a to pokrycia (że tylko 80%), a to nazewnictwa (że niejasne), a to długości metod (że za długie i nadające się do zmiany)… I to „coś" zaczyna w nas rosnąć, rozsadzając zdrowy rozsądek, pozwalając kiełkować sarkazmowi i ironii. W naszym środowisku w takich wypadkach bardzo dobrze działa programowanie w parach. Po prostu prosisz osobę, która robi review, żeby napisała ten kawałek z Tobą i jednocześnie robiła re-review. Wymaga to wyjścia spomiędzy krzewów sarkazmu i ironii, ale opłaca się, ostatecznie wszyscy jedziemy na tym samym wózku i chcemy, żeby prowadził się on jak najłatwiej i przynajmniej był niebrzydki ;-).

Michał Węgrzyn

Certyfikacja ISTQB – czy warto?

Co musisz wiedzieć:

- Środowisko testerskie na całym świecie nie jest homogeniczne. Mamy frakcje, obozy, zwolenników i przeciwników technik i metodologii. Z tego też powodu stworzenie jedynego słusznego programu jest etycznie wątpliwe.
- Zdecydowana większość ogłoszeń na stanowiska związane z jakością oprogramowania zawiera słowo „ISTQB" w sekcji „wymagane" lub „mile widziane".
- Certyfikacja ISTQB jest dowodem na zainteresowanie tematem. Pytanie tylko, czy tematem posiadania certyfikatu, czy też tematem testowania oprogramowania…

Część 1.: Przebudzenie

Na sam początek dwa zdania wstępu. Posiadam certyfikat ISTQB Foundation Level oraz odbyłem 5-dniowe szkolenie ISTQB Test Manager. Przebrnąłem też przez znaczną cześć książki Rexa Blacka przygotowującej do egzaminu z tego zakresu. Pomysł na ten artykuł zrodził się w czasie bolesnych godzin czytania wyżej wymienionej publikacji.

Biorąc pod uwagę to, jak wygląda proces certyfikacji, fakt, że posiadanie tego certyfikatu jest wymagane lub znajduje się co najmniej w sekcji „nice to have" w zasadzie w każdym ogłoszeniu o pracę dla testera czy też test managera, stanowi moim zdaniem fenomen.

Posiadanie certyfikatu potwierdza przynależność do pewnej społeczności. Nie ma w tym nic złego. No, może poza tym, że ciało certyfikujące wcale nie reprezentuje całego community testerskiego, lecz jego fragment, ale przede wszystkim siebie i swoje interesy.

Żeby wszystko było jasne, karty na stół. Sam jestem wielkim zwolennikiem szkoły context driven testing. Wiem jednak, mając już doświadczenie w dużych firmach, iż wdrożenie tego podejścia do zapewnienia jakości nie jest proste w dużych organizacjach. Wynika to z wielu czynników. Głównym z nich jest moim zdaniem mała wiedza naszej kadry managerskiej na tematy związane z testowaniem oraz potrzeba raportowania i dokumentowania wszystkiego, niezależnie, czy ma to sens, czy nie.

Powyższe przemyślenia sprawiły, iż napisałem do mojego „guru" – Jamesa Bacha. Korespondencja jest po angielsku. Według mnie przetłumaczenie jej na polski sprawiłaby, że zniknęłyby pewne smaczki. Poniżej całość naszej korespondencji:

Michał Węgrzyn: First of all: you are doing an amazing job with promoting real software testing. Just to make it clear. The world however is a strange place. Place where paper matters. No one cares if you are an amazing exploratory testing machine with various skills and great understanding of cause-effect. People who pay look for proof. This is the reason ISTQB is such a worldwide success.

James Bach: That isn't proof. Some „people who pay" are remarkably gullible. That's all.

MW: I have to admit I am ISTQB FL certified and ISTQB Advanced trained and I am preparing for the exam. This is the reality I live in. Maybe it is different in Silicon Valley, but I live in Poland and here, where software development in mostly done as support for large corporations, certification matters. Why? Because this is how world runs. Who cares that I try to apply all the right ways during test management over ISTQB crap. If I will try to find a better, more interesting and higher paying job I need to be (or at least this will help) ISTQB Advanced Level certified. I hate giving money away paying for the exam, but sometimes you need to bite the bullet.

JB: You work in the fake testing industry. Of course, in that industry they value fake testing. I work in the real testing industry. It is a smaller industry.

MW: Have you ever considered creating your own certification path? I know, how to measure what cannot be measured, but still, if I can adapt so can you. If you'll adapt this might be the end of ISTQB….

JB: I only tell the truth. I only do what helps. Yes, I have considered it. It is not economically viable. Just as in medicine, real cures are difficult and expensive, but fake cures are cheap.

Fake testing industry – to określenie zostało w mojej głowie na długo.

Czy naprawdę tak wygląda nasza rzeczywistość? Czy cały dorobek ISTQB, wraz z sylabusami, egzaminami to jedna wielka

ściema, mająca na celu dać zarobić organizacji, trenerom i firmom przeprowadzającym kursy, ułatwić rekruterom selekcję kandydatów, a managerom nieznającym tematu – podjęcie decyzji o zatrudnieniu? Czy też jest to program mający przedstawić pewien sposób myślenia, powszechnie uznany za „best practices", dać globalnie rozpoznawalną jednostkę miary w postaci certyfikatu, pomóc w usystematyzowaniu ogromu wiedzy?

Moim zdaniem najważniejsze pytanie, jakie należałoby sobie zadać, brzmi: **Czy posiadanie certyfikatu czyni kogokolwiek lepszym testerem?** Moim zdaniem nie. Czyni go lepszym kandydatem do roli testera w oczach HR.

Część 2.: Argumenty przeciw ISTQB

Uważam, że proces certyfikacji testera powinien w jakikolwiek sposób sprawdzać umiejętności, nie tylko wiedzę. Sama certyfikacja jest procesem testowania, co testuje egzamin ISTQB?

Co interesuje mnie u testerów? Umiejętność testowania, posiadanie wymaganych cech charakteru, takich jak profesjonalna ciekawość, dokładność, umiejętność kojarzenia pozornie niezwiązanych ze sobą faktów. Nie zależy mi natomiast na zdolności zapamiętywania definicji na temat bardzo uproszczonych sytuacji.

Podczas egzaminu Foundation Level weryfikowana jest teoretyczna znajomość podstawowych pojęć. Z kolei test certyfikacyjny poziomu Advanced skupia się na specyficznej kolejności słów. Podstępnie ułożone pytania mają za zadanie wprowadzić zdającego w błąd. Po raz kolejny pytam, co testuje taki egzamin?

Jeden z moich znajomych postanowił zmienić profesję na testerską. W tym celu zdał certyfikat ISTQB FL. Osoba ta nie przepracowała ani jednego dnia jako tester. Zero. Jednak otrzymała certyfikat potwierdzający w oczach wielu osób profesjonalizm i wiedzę z zakresu testowania. O czym to świadczy?

Warto także wspomnieć o tym, że efektem ubocznym ścieżki certyfikacyjnej jest jej stagnacja. Te same pomysły, niezmienne od lat. Brak innowacji. Trenerzy nie są zainteresowani tworzeniem nowych szkoleń, wolą status quo, czyli odcinanie kuponów poprzez używanie tych samych materiałów, odpowiadanie na te same pytania.

W moim przekonaniu sednem doskonalenia się w roli testera nie jest rozwijanie umiejętności nauki na pamięć, lecz wykształcenie u siebie zdolności myślenia systemowego, zrozumienie filozofii i metodologii potrzebnych w danej sytuacji.

ISTQB oferuje tylko drogę na skróty. Zarówno dla nowych osób zainteresowanych pracą w branży IT, jak i dla managerów odpowiedzialnych za rekrutację, jednak pełniących funkcje niezwiązane bezpośrednio z procesem testowania. Dla kadry kierowniczej to skrót w procesie rekrutacyjnych. Po co dokładnie sprawdzać kandydata? Można przecież wpisać wymaganie „certyfikat ISTQB" i po sprawie. Czyżby?

Kolejną sprawą jest ważność tego certyfikatu. Otóż FL i Advanced są dożywotnie.

Jakimi kompetencjami trzeba się wykazać, aby utrzymać certyfikat? Żadnymi.

Jeśli chodzi o sam proces testowania, to wniosek płynący z nauk ISTQB wydaje prosty: testowanie tylko wówczas można uznać za godne zaufania i skuteczne, kiedy jest dokładnie zaplanowane, przygotowanie, sformalizowane i raportowane. Czy na pewno?

Moim zdaniem problem ISTQB stanowi to, że plan nauczania koncentruje się tylko na jednej z kilku dostępnych metodologii testowania. Nie uwzględnia natomiast żadnych umiejętności testerskich.

Niska świadomość tego, o co naprawdę chodzi w testowaniu, wynika z wieloletniego upośledzenia tej dziedziny jako gałęzi IT. Sprawia to, iż:

- Niewiele osób odróżnia umiejętności od metodyki.
- Łatwiej sprawdzić znajomość metodyki niż umiejętności, stąd opisana wcześniej droga na skróty.
- Osoby uczestniczące w tym biznesie chyba zdają sobie z tego sprawę, jednak wolą kontynuować proceder, dopóki sieje grozę wśród nowych testerów i daje światełko nadziei managementowi.

Z wyżej wymienionych powodów, szczególnie z uwagi na fakt, że egzamin testuje nie to, co ważne w pracy testera, uważam, że nie pozwala nawet podstawowym zakresie ocenić, czy dana osoba nadaje się do tego zawodu.

Część 3.: Argumenty za ISTQB

Jedno trzeba przyznać. Posiadanie certyfikatu jest dowodem na zainteresowanie testowaniem. Świadczy o tym, że dana osoba poświęciła czas na przygotowanie się i zdanie egzaminu. Jest to niewątpliwy plus, o ile mówimy o szczerym zainteresowaniem testowaniem, a nie jedynie posiadaniem certyfikatu. Tego już egzamin nie weryfikuje.

Posiadanie certyfikatu na pewno ułatwia szukanie pracy w branży. Tego faktu również nie można negować. Analiza ogłoszeń daje jednoznaczny wynik. Mając certyfikat, znacząco zwiększamy swoje szanse na etapie selekcji CV kandydatów.

Dlaczego więc ktokolwiek decyduje się na kurs z zakresu ISTQB? Bo wszyscy tak robią! W Holandii 90% testerów ma certyfikat, czy to ISTQB, czy też Tmap. Reszta to testerzy agile, niezainteresowani tematem certyfikacji [Źródło: *Challenges of implementing CDT in a large organization* by Jean-Paul Varwijk, *https://www.youtube.com/watch?v=3-RDFIXjbDM*].

Posiadanie certyfikatu jest też globalnie rozpoznawalnych symbolem. A co ten symbol naprawdę znaczy – to inna sprawa. Jednak nie można w żaden sposób odmówić ISTQB sukcesu, który ta organizacja odniosła poprzez promowanie swoich produktów.

Ostatnia sprawa to komunikacja. Znacznie łatwiej jest porozumiewać się w sprawie jakości, jeśli mówimy tym samym językiem. ISTQB ujednolica słownictwo i nazewnictwo. Ułatwia to dialog z testerami

AUTOR

Michał Węgrzyn

Doświadczony test engineer, quality assurance specialist, technical writer, trener i team leader. Pełni wiele funkcji w projektach informatycznych, przy czym zawsze kieruje się jednym celem: zadowoleniem użytkownika końcowego.

Pracuje w ABB Polska, gdzie na co dzień zajmuje się testami aplikacji zarówno webowych, jak i desktopowych oraz sprawowaniem pieczy nad procesem kontroli jakości wytwarzanego oprogramowania.

Cele zawodowe autora to ciągłe dążenie do zdobywania nowych umiejętności: technicznych i miękkich oraz coaching młodszych współpracowników.

Michał Węgrzyn jest zapalonym miłośnikiem kulturystyki, literatury fantasy oraz swojego buldoga angielskiego Lebrona.

z innych części świata, jak i komunikację z managementem. Jeśli chodzi o decyzje kadry zarządzającej – łatwą przekonać ją do podejścia procesowego, tj. ISTQB.

Część 4.: Zdecyduj sam

Mimo iż starałem się pełnić funkcję adwokata diabła, moja opinia jest chyba oczywista. Pytanie, które powinien zadać sobie każdy z nas, testerów, brzmi: czy kontynuować ścieżkę certyfikacyjną ISTQB poprzez zdawanie egzaminów Advanced czy też Expert? A może należy się zastanowić, czy w ogóle warto podejść do części Foundation?

Ważne jest też, jakie wnioski wyciągnie nasze środowisko. Czy pisząc ogłoszenia, nadal będziemy wymagać certyfikatów? Czy te mają jakąkolwiek wartość?

Najważniejszy moim zdaniem argument przeciwko wszelkim certyfikacjom brzmi: bardzo często wykorzystują one niewiedzę ludzi. *Zdaj ISTQB, zostań testerem!* – znacie to hasło? A co ma jedno do drugiego?

Patrycja Jakubowska

Zapobieganie defektom za pomocą technik statycznych

Wprowadzenie

Jak mierzyć jakość systemu informatycznego, którego jeszcze nie ma? Jak zaprojektować i zaimplementować oprogramowanie, gdy nie posiadamy konkretnej wiedzy dotyczącej tego, jakie ma ono być i jakie funkcje powinno oferować? Samo zadawanie tego rodzaju pytań potwierdza, że pożądane i możliwe jest wpływanie na jakość systemu informatycznego od samego początku jego wytwarzania – już na etapie pomysłu na wykonanie aplikacji.

Źródłem wysokiej jakości systemu informatycznego jest przygotowanie wysokiej jakości wymagań. Dlatego należy odpowiedzieć sobie na pytanie, w jaki sposób jako analitycy/testerzy możemy wpłynąć na to, żeby rzeczywiście były one dobrej jakości. W takich sytuacjach pomocne okazują się techniki statyczne – przeglądy i inspekcje.

Niniejszy artykuł prezentuje korzyści płynące ze stosowania technik statycznych oraz trudności związane z ich używaniem w odniesieniu do dokumentów analitycznych, będących podstawą wytworzenia wysokiej jakości produktu. Ponadto autorka zaprezentuje projekty list kontrolnych niezbędnych przy przeprowadzaniu przeglądów i inspekcji dokumentów analitycznych oraz przedstawi sposób analizy i weryfikacji wyników.

Korzyści płynące ze stosowania technik statycznych – przeglądów i inspekcji

Przeglądy i inspekcje pozwalają znajdować defekty jeszcze przed wykonaniem testów dynamicznych oraz identyfikować usterki, które prawdopodobnie mogłyby zostać pominięte na dalszym etapie przeprowadzanie testów. Dzięki technikom statycznym, w szczególności przeglądom i inspekcjom, można znaleźć słabe punkty w materiałach będących podstawą do wytwarzania oprogramowania, czyli w dokumentacji i specyfikacjach wymagań.

Znajdowanie błędów na początku procesu budowania systemu pozwala na osiąganie wymiernych korzyści. Na szczególną uwagę zasługuje zmniejszenie liczby błędów na późniejszych etapach procesu wytwarzania oprogramowania. Opieranie się na nieprawidłowych wymaganiach i specyfikacjach wymagań prowadzi do powielania istniejących usterek w następnych fazach produkcji, a niekiedy również do generowania nowych defektów. Stosowanie technik statycznych pozwala znacznie ograniczyć liczbę usterek „przechodzących" do kolejnych etapów wytwarzania systemu, ponieważ kontrola jakości wymagań zapobiega tworzeniu oprogramowania na bazie nieprawidłowych założeń. Znalezienie i poprawienie defektów na początku procesu budowania aplikacji pozwala uzyskać wysokiej jakości podstawy (wymagania), a co za tym idzie, umożliwia ograniczenie liczby usterek, które mogłyby zostać zaimplementowane. W efekcie oszczędzamy czas i koszty związane z wytwarzaniem oprogramowania.

Jeśli wymagania zawierające defekty (nieznalezione we wczesnym etapie budowania aplikacji) są implementowane, usterki muszą następnie zostać naprawione. W efekcie proces produkcji wydłuża się. Znajdowanie błędów na etapie wymagań pozwala zredukować łączny czas wytwarzania oprogramowania i zmniejszyć ryzyko dostarczenia systemu niskiej jakości.

Warto też zwrócić uwagę na to, że nakład pracy i koszty związane z usuwaniem usterek rosną wraz z upływem czasu. Problemy zidentyfikowane na późniejszych etapach produkcji generują wyższe koszty naprawy. Dodatkowo wymagają większej ilości czasu na korektę w porównaniu z błędami usuwanymi wcześniej.

Odpowiednio przeprowadzane przeglądy i inspekcje przyczyniają się również do lepszej komunikacji w zespole biorącym udział w tego typu przedsięwzięciach. Ponadto umożliwiają poprawę relacji na linii analityk – deweloper – tester. Wymagania, jakie system powinien spełniać, są właściwie określone i opisane, dzięki czemu programiści mają świadomość, jaki produkt mają przygotować, a testerzy wiedzą, jak przetestować taką aplikację, ponieważ mają pewien punkt odniesienia.

Reasumując, defekty znalezione i naprawione w początkowej fazie wytwarzania oprogramowania zdecydowanie zwiększają produktywność procesu i przyczyniają się do podniesienia jakości produktu.

Rodzaje błędów identyfikowanych dzięki stosowaniu przeglądów i inspekcji

Stosowanie technik statycznych pozwala na identyfikację usterek trudnych do znalezienia przy wykorzystaniu technik dynamicznych, np. błędów takich jak: odchylenia od standardów, niskiej jakości wymagania (np. niekompletne, niespójne, niejednoznaczne), wady projektowe, niedostateczna utrzymywalność czy nieprawidłowa specyfikacja interfejsów.

Rodzaje defektów znajdowanych dzięki stosowaniu technik statycznych podzielono ze względu na rodzaje dokumentów, które podlegają ocenie.

Rodzaje identyfikowanych błędów w wymaganiach

Błędy wymagań, które można zidentyfikować dzięki stosowaniu przeglądów i inspekcji, to:

- niekompletność wymagania, tzn. brak szczegółowych informacji niezbędnych do zrozumienia go, określenia autora, źródła itp.;
- niejednoznaczność wymagania, przejawiająca się w odmiennym rozumieniu wymagania przez różne osoby;
- brak możliwości technologicznej implementacji rozwiązania;
- niepoprawność wymagania, tj. brak dokładnego opisu cechy, jasnego zdefiniowania wymagania;
- nietestowalność wymagania;
- niespójność z dokumentami wyższego rzędu;
- brak określenia priorytetu.

Rodzaje identyfikowanych błędów w specyfikacji wymagań

Błędy dotyczące zbioru wymagań

Błędy specyfikacji wymagań, które można zidentyfikować dzięki stosowaniu przeglądów i inspekcji, to:

- niekompletność wymagań;
- niepoprawność wymagań, tj. brak dokładnego opisu cechy, jasnego zdefiniowania wymagań;
- niejednoznaczność wymagań, przejawiająca się w odmiennym rozumieniu wymagań przez różne osoby;
- niespójność wymagań – brak powiązania między wymaganiami, sprzeczność między poszczególnymi wymaganiami, wykluczające się wymagania;
- nietestowalność wymagań.

Błędy dotyczące zawartości i struktury specyfikacji wymagań

Błędy dotyczące zawartości i struktury dokumentu specyfikacji wymagań to:

- brak określonego celu;
- brak opisu rozwiązania;
- nieprzejrzysta struktura dokumentu;
- niespójność dokumentu;
- nieczytelność dokumentu;
- brak możliwości śledzenia zmian;
- niewskazanie wersji dokumentu;
- stosowanie niepoprawnych lub nieaktualnych odnośników;
- niewskazanie autorów;
- brak opisu użytkowników końcowych systemu;
- brak aktualnego i poprawnego słownika terminów i skrótów stosowanych w dokumencie.

Trudności i wyzwania związane z realizacją przeglądów

Rozpoczęcie testów w momencie, gdy produkt jeszcze nie istnieje, a więc według opinii wielu osób „nie ma czego testować", jest bardzo trudne. Problemy dotyczące realizacji przeglądów często wynikają z poziomu dojrzałości organizacji oraz umiejętności stosowania takich technik przez firmę i jej personel. Wpisanie stosowania przeglądów i inspekcji w kulturę organizacji i jej procesy wytwórcze wymaga czasu. Wdrażanie i stosowanie takich technik w firmie okazuje się wyzwaniem m.in. z powodu obszernych dokumentów wymagań, dużych zespołów inspekcyjnych, rozproszenia geograficznego recenzentów oraz ich nieprzygotowania do przeglądów. Brak znajomości potencjalnych problemów przez organizację może mieć wpływ na niepowodzenie wykorzystywania statycznych technik testowania.

Większość problemów związanych z przeglądami i inspekcjami wynika z charakterystyki uczestników procesu i ich podejścia.

W sytuacji, gdy kierownictwo nie zachęca podwładnych do stosowania technik statycznych, autorzy oraz recenzenci, którym zależy na tego rodzaju testach, mogą nie mieć wystarczającej siły przebicia. W efekcie niewykluczone, że nie zdołają przekonać pozostałych uczestników do celowości organizowania przeglądów i inspekcji.

Ponadto niezwykle ważnym czynnikiem powodzenia testów statycznych jest zaproszenie do udziału w nim osób, które rozumieją cel takich przedsięwzięć i korzyści płynące ze stosowania ich. Zaangażowanie w przeglądy i inspekcje pracowników z niewłaściwym nastawieniem lub po prostu nieodpowiednich osób może skutkować brakiem należytego przygotowania do przeglądu i udziału w nim.

Inny rodzaj trudności związanych z wykorzystywanie technik statycznych to problemy wynikające z błędów procesu przeglądu. Proces przeglądu formalnego składa się z następujących faz: [2]

1. ocena wstępna;
2. przygotowanie;
3. planowanie przeglądu;
4. przeglądanie procedur dotyczących przeglądów;
5. przygotowanie indywidualne;
6. badanie grupowe;
7. czynności naprawcze;
8. ocena końcowa

Defekty znalezione podczas początkowych faz przeglądu i zgłoszone najpóźniej w fazie badania – spotkania grupowego, powinny zostać usunięte. Jeśli czynności naprawcze nie będą wykonane oraz/lub brakuje ostatniej fazy przeglądu – oceny końcowej, cały proces przeglądu lub inspekcji nie spełnia swojej funkcji. Samo zebranie uwag przez autora, bez podjęcia działań niezbędnych do usunięcia defektów oraz bez sprawdzenia kryteriów zakończenia (w przypadku inspekcji), nie przyczynia się do poprawy jakości dokumentów wymagań i specyfikacji wymagań. Jednocześnie takie postępowanie może negatywnie nastawiać potencjalnych uczestników procesu przeglądu i inspekcji do właściwego udziału w nim; takie osoby mają prawo uznać, że ich praca – np. przy wcześniejszych przeglą-

dach – została w pewnym sensie zmarnowana i nie przyczyniła się do stworzenia lepszych produktów.

Rodzaje dokumentacji, która może podlegać przeglądom

Przeglądom i inspekcjom może podlegać każdy produkt procesu wytwarzania oprogramowania. Bazując na standardzie IEEE 1028:2008, wskazano następujące przedmioty przeglądów (rysunek 1). Dokumenty, których dotyczy niniejszy artykuł, wyróżniono kolorem.

Projekty list kontrolnych i sposoby analizy ich wyników

W tej części opracowania zaprezentowano projekty list kontrolnych dla wymagań oraz specyfikacji wymagań. Zakładają one, że dla przeglądów i inspekcji z użyciem list kontrolnych zasadne jest wykonanie testów dymnych, które są swego rodzaju testami wstępnymi. Ich cel to potwierdzenie, że kluczowe aspekty czy cechy testowanego obiektu działają, a na samym obiekcie można przeprowadzić szczegółowe testy. Testy dymne w przypadku list kontrolnych dla przeglądów pozwalają na identyfikację rażących błędów, niespójności i luk uniemożliwiających dalszą ocenę

Rysunek 1. Przykładowe przedmioty przeglądów i inspekcji. *Źródło: IEEE 1028:2008*

danego produktu. Z punktu widzenia powodzenia stosowania technik statycznych listy kontrolne są bardzo istotne. Pomagają one zweryfikować cechy wymagań i specyfikacji, które powinny być bezwzględnie spełnione. Z tego powodu proponowane w dalszej części artykułu listy kontrolne zawierają pytania dotyczące niezbędnych atrybutów, tzn. takich, których spełnienie jest wymagane w 100%. Tylko udzielenie pozytywnych odpowiedzi umożliwia przejście do pozostałych pytań. W przeciwnym wypadku weryfikowane wymaganie lub specyfikacja muszą zostać zmodyfikowane, tak aby spełniały dane kryterium.

PROJEKT LISTY KONTROLNEJ DLA KONKRETNEGO WYMAGANIA

Identyfikator wymagania: ***Treść wymagania:***		***Wersja wymagania:***	
Data przeglądu:		***Recenzent:***	
L.p.	***Analiza cech niezbędnych***	***Spełnienie***	
		Tak	***Nie***
1.	**Kompletne** (zawiera wszystkie informacje niezbędne do zrozumienia go, w tym dane autora, źródło)		
2.	**Jednoznaczne** (osoby przeglądające rozumieją wymaganie w ten sam sposób)		
3.	**Wykonalne** (istnieje możliwość zaimplementowania)		
4.	**Poprawne** (opisuje daną cechę zgodnie z oczekiwaniami interesariuszy)		
Sposób wyznaczenia wyniku: • **„Pozytywny”**, gdy wszystkie odpowiedzi brzmią „Tak”. • **„Negatywny”**, gdy co najmniej jedna odpowiedź brzmi „Nie”.		....	
Interpretacja wyniku: • Gdy wynik „Pozytywny” – przejście do dalszej części listy kontrolnej. • Gdy wynik „Negatywny” – zakończenie listy kontrolnej- wymaganie należy poprawić.			

PROJEKT LISTY KONTROLNEJ DLA KONKRETNEGO WYMAGANIA cd.

L.p.	*Analiza cech*	*Stopień spełnienia [%]*		
		0	*50*	*100*
5.	**Konieczne** – niezbędne (opisuje cechę, którą system bezwzględnie musi mieć)			
6.	**Weryfikowalne** (może być przetestowane; istnieje sposób mierzalnej weryfikacji poprawności implementacji)			
7.	**Abstrakcyjne** (niezależne od sposobu implementacji)			
8.	**Spójne** (nie stoi w sprzeczności z innymi wymaganiami ani dokumentami wyższego rzędu)			
9.	**Ma priorytet** (znaczenie wymagania jest scharakteryzowane przy wykorzystaniu priorytetów)			
10.	**Nie duplikat** (nie jest powtórzeniem innego wymagania)			
Sposób wyznaczenia wyniku: (0,5 × Iiczba cech spełnionych w 50% + liczba cech spełnionych w 100%) / 6 (0,5 × ..... + ...) / 6 = ...				
Interpretacja wyniku: • Gdy wynik > 0,6 – wymaganie spełnia badane kryteria. • W przeciwnym wypadku wymaganie należy poprawić.				

Analiza i interpretacja wyników list kontrolnych

Wypełnienie list kontrolnych jest podstawą do obliczenia wyniku spełnienia kryteriów. Wynik końcowy spełnienia kryteriów obliczamy na podstawie wzoru:

(0,5 × liczba cech spełnionych w 50% + liczba cech spełnionych w 100%) / liczba pytań*.

** liczba pytań po odjęciu liczby pytań dotyczących cech niezbędnych*

W zależności do potrzeb i rodzaju wytwarzanego oprogramowania poziom oczekiwanego wyniku może być inny. Dla potrzeb opracowania przyjęto, że satysfakcjonującym rezultatem jest spełnienie przyjętych kryteriów jakościowych dla wymagań i specyfikacji wymagań na poziomie 60%. Można przyjąć, iż dla oprogramowania krytycznego powinien on być zbliżony do 100%.

PROJEKT LISTY KONTROLNEJ DLA SPECYFIKACJI WYMAGAŃ

Identyfikator specyfikacji wymagań: ***Autor specyfikacji:***		***Wersja specyfikacji wymagań:***	
Data przeglądu:		***Recenzent:***	
L.p.	***Analiza cech niezbędnych***	***Spełnienie***	
		Tak	***Nie***
Dotyczących wymagań			
1.	**Kompletne** (m.in. mają identyfikator i źródło)		
2.	**Poprawne** (dokładnie opisują cechy, jasno definiują funkcjonalności)		
3.	**Jednoznaczne** (osoby przeglądające rozumieją wymagania w ten sam sposób)		
4.	**Spójne** (wymagania nie stoją w sprzeczności ze sobą ani dokumentami wyższego rzędu)		
Sposób wyznaczenia wyniku: • **„Pozytywny”**, gdy wszystkie odpowiedzi brzmią „Tak”. • **„Negatywny”**, gdy co najmniej jedna odpowiedź brzmi „Nie”.		....	
Interpretacja wyniku: • Gdy wynik „Pozytywny” – przejście do dalszej części listy kontrolnej. • Gdy wynik „Negatywny” – zakończenie listy kontrolnej- wymaganie należy poprawić.			

Sposobem na analizę wyniku listy kontrolnej oraz jego ocenę i interpretację danego wymagania jest odniesienie rezultatu do:

- wyniku list kontrolnych poprzednich przeglądów innego projektu (przy oprogramowaniu tworzonym dla podobnych klientów, zbliżonych branż);
- wyniku list kontrolnych poprzednich przeglądów danego projektu;
- wyników innych wymagań, dla których sporządzono badanie jakości na bazie list kontrolnych w czasie danego przeglądu.

O wzroście jakości wymagań możemy mówić wówczas, gdy następuje ilościowy wzrost spełnienia kryteriów jakościowych w odniesieniu do wymienionych list kontrolnych.

Sposobem na weryfikację wyniku listy kontrolnej specyfikacji wymagań jest od-

PROJEKT LISTY KONTROLNEJ DLA SPECYFIKACJI WYMAGAŃ cd.

L.p.	*Analiza cech*	*Stopień spełnienia [%]*		
		0	*50*	*100*
Dotyczących wymagań				
5.	**Modyfikowalne** (łatwość wprowadzania modyfikacji i przeformułowywania wymagań)			
6.	**Uporządkowane wg ważności** (wymagania mają określone priorytety)			
7.	**Testowalne** (możliwość opracowania testów w celu weryfikacji zaimplementowania wymagań)			
Dotyczących dokumentu				
8.	Określony cel dokumentu			
9.	Opis rozwiązania			
10.	Przejrzysta struktura dokumentu			
11.	Możliwość śledzenia zmian dokumentu			
12.	Określony numer wersji dokumentu			
13.	Stosowanie poprawnych, aktualnych odnośników			
14.	Opis aktorów systemu			
15.	Aktualny i poprawny słownik terminów i skrótów			
Sposób wyznaczenia wyniku: (0,5 × Iiczba cech spełnionych w 50% + Iiczba cech spełnionych w 100%) / 11 (0,5 × …….. + ……..) / 11 = ……				
Interpretacja wyniku: • Gdy wynik > 0,6 – wymaganie spełnia badane kryteria. • W przeciwnym wypadku wymaganie należy poprawić.				

niesienie rezultatu do:

- wyniku list kontrolnych poprzednich przeglądów innego projektu (przy oprogramowaniu tworzonym dla podobnych klientów, zbliżonych branż);
- wyniku list kontrolnych poprzednich przeglądów danego projektu.

Podsumowanie

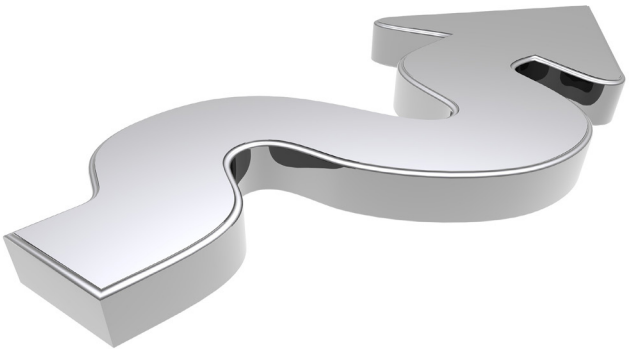
Przeglądy i inspekcje są technikami oceny jakości dokumentów będących podstawą do stworzenia systemu oprogramowania, które przynoszą satysfakcjonujące rezultaty. Szacuje się, że godzina zainwestowana w inspekcję dokumentów wymagań i specyfikacji wymagań pozwala zaoszczędzić 10 godzin dodatkowej pracy na późniejszych etapach budowania aplikacji, a więc można uzyskać 1000% zwrot z inwestycji poświęconego czasu [6].

W związku z tym, iż prowadzenie przeglądów i inspekcji bywa dość żmudne i czasochłonne, bardzo ważne jest odpowiednie zaplanowanie tych procesów. Optymalny czas trwania pojedynczego spotkania to 2 godziny. Z kolei liczba sprawdzanych stron na godzinę inspekcji wg IEEE1028:2008 wynosi 3-4. Aby zapewnić sprawniejszy i bardziej efektywny przebieg takich spotkań, warto posiłkować się listą kontrolną.

Ponadto koszty przeprowadzania przeglądów i inspekcji są zdecydowanie niższe niż nakłady ponoszone w celu naprawienia błędów odkrywanych na późniejszych etapach tworzenia oprogramowania – podczas implementacji, testów czy działania produkcyjnego. Według danych IBM względny koszt usunięcia usterki w fazie przeglądu projektu wynosi „1”. Na kolejnych etapach wzrasta i w czasie testów maszynowych osiąga poziom „82” (wartości symboliczne, służące pokazaniu przepaści między kosztami usunięcia defektu na poszczególnych etapach) [2]. Na podstawie przedstawionych danych można jednoznacznie stwierdzić, iż znajdowanie defektów w początkowej fazie wytwarzania systemu jest zdecydowanie bardziej opłacalne.

Poza przedstawionymi korzyściami związanymi z oszczędnością czasu i pieniędzy warto wymienić pozostałe zalety stosowania technik statycznych: lepszą komunikację w zespole wytwórczym oraz fakt, że osoby biorące udział w przeglądach i inspekcjach mają szansę poznać budowane oprogramowanie od najwcześniejszych etapów przygotowywania go. Procentuje to w przyszłości, np. podczas późniejszych testów.

Przedstawione w artykule korzyści płynące ze stosowania technik statycznych, takich jak przeglądy i inspekcje, pozwalają stwierdzić, że warto wdrażać je w organizacji.



AUTOR

Patrycja Jakubowska

Karierę w IT rozpoczęła 8 lat temu; łączyła wówczas pracę analityka biznesowego, testera i wdrożeniowca systemu ERP. Nieustannie podnosi swoje umiejętności i kwalifikacje, m.in. poprzez uzyskanie certyfikatu ISTQB oraz uczestnictwo w szkoleniach, warsztatach i konferencjach związanych z szeroko rozumianym IT.

Nabyte doświadczenie pozwoliło na sprecyzowanie ścieżki kariery zawodowej i wprowadzenie zmian w życiu zawodowym. Efektem zmian była decyzja o współtworzeniu (jako analityk i projektant systemu) autorskiego systemu informatycznego firmy SESCOM.

Zarówno prywatnie, jak i zawodowo uwielbia planować, analizować i osiągać założone cele.

Miłośniczka skandynawskich kryminałów (które również da się analizować). Odpoczywa na treningach zumby.

Literatura

1. BABOK: *A Guide to the Business Analysis Body of Knowledge (BABOK Guide)*, http://www.iiba.org/.
2. Chrabski Bartosz, Zmitrowicz Karolina: *Inżynieria wymagań w praktyce*, PWN, Warszawa 2015.
3. IEEE 1028:2008: IEEE Standard for Software Reviews and Audits.
4. ISTQB, *Certyfikowany Tester. Plan poziomu podstawowego,* wersja 2011.1.1, 2012, http://istqb.pl/.
5. REQB: *REQB. Certyfikowany Profesjonalista Inżynierii Wymagań. Poziom Podstawowy*, wersja 1.3.1, 2011, http://reqb.org/.
6. Wiegers Karl, Beaty Joy: *Specyfikacja oprogramowania. Inżynieria wymagań*. Wydanie III, Helion, Gliwice 2014.
7. Zmitrowicz Karolina, *Analityk Systemów*, PWN, Warszawa 2015.

Magazyn

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

Współpraca
Tomasz Olszewski
tomasz.olszewski@quale.pl
Piotr Poznański
piotr.poznanski@quale.pl
Zuzanna Gościcka-Miotk
zuzanna.goscicka-miotk@quale.pl

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

Wszystkie znaki firmowe zawarte w piśmie są własnością odpowiednich firm.

Prawa autorskie
Wszystkie opublikowane artykuły są objęte prawem autorskim autora lub przedsiębiorstwa. Wykorzystywanie w innych publikacjach, kopiowanie lub modyfikowanie zawartości artykułu bez pisemnego upoważnienia autora jest zabronione.

Grafiki użyte w magazynie
http://pixabay.com/
Grafiki na licencji CC
Free for commercial use / No attribution required

Okładka
Fractal spiral star light
http://pixabay.com/
Grafiki na licencji CC
Free for commercial use / No attribution required