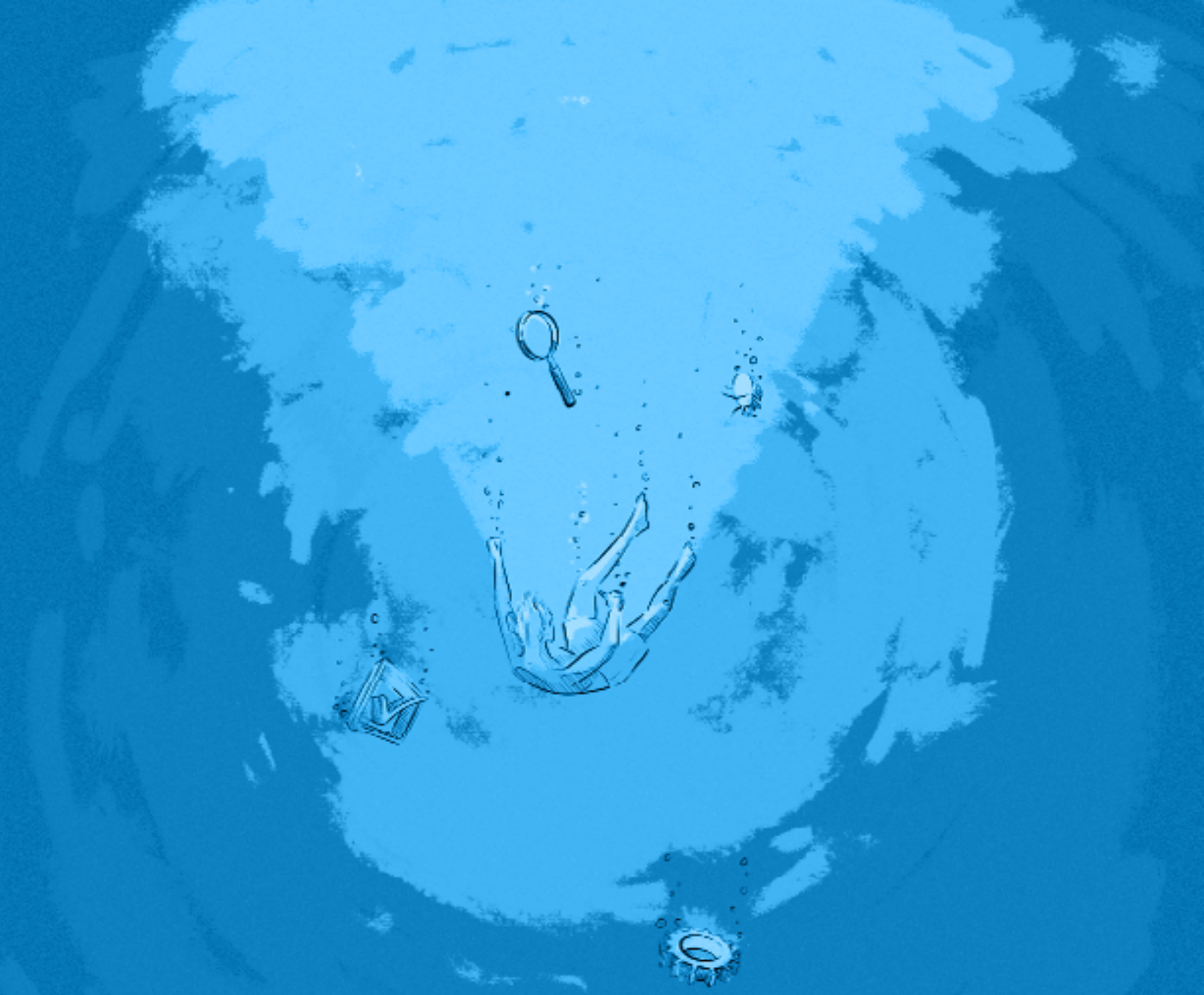


Przygoda z QA

Jak przeżyć w świecie pełnym testów?

Mateusz Stachurzewski

Wstęp

Trudno jest określić początek branży testerskiej. Przyjmijmy, że od momentu, gdy zaczęły powstawać programy, pierwszą próbę weryfikacji tych programów można określić jako początek testowania. Bez względu na to, czy robił to tester czy programista, można powiedzieć, że miało to miejsce dziesiątki lat temu.

Nie trudno się domyślić, że proces wytwarzania oprogramowania, jak i same programy to zupełnie inna bajka niż to, co działo się na samym początku. Co więcej, co 5-10 lat na rynku można zauważyć zmiany, które wpływają na to, jakie aplikacje tworzymy i w jaki sposób je wytwarzamy. To sprawia, że zarówno programowanie, jak i sposoby zapewniania jakości również się zmieniają.

Dla wielu osób obraz zapewniania jakości pochodzi sprzed kilku lub nawet kilkudziesięciu lat. To sprawia, że trudno jest zrozumieć ten obszar zarówno początkującym QA, jak i osobom, z którymi QA współpracują, tj. programiści, czy managerowie. Z tego powodu postanowiłem podzielić się swoją wiedzą i spostrzeżeniami z zakresu testowania i zapewniania jakości, aby zaktualizować postrzeganie QA wśród osób, z którymi pracuję na co dzień.

Każdy klient, projekt, zespół oraz finalny produkt, jaki wytworzymy, będzie inny. Dlatego w moich tekstach staram się podkreślić istotę procesu jakości szytego na miarę i przekazywać informacje, które pozwolą nie tyle powielać, co zrozumieć i dopasować praktyki zapewniania jakości.

Zapraszam do lektury.

Mateusz Stachurzewski

Spis treści

Cases & inne 162

Część I

Teoria Testowania

Wpis #1 Trzy fronty testowania

Z mojego punktu widzenia testowanie bardzo często postrzegane jest w sposób okrojony. W branży testerskiej, problem ten został przedstawiony jako testing vs checking.

Co jest zatem esencją testowania? Można to ująć na co najmniej trzy sposoby:

- Testujemy, bo chcemy sprawdzić, czy nasze oczekiwania spotykają się z rzeczywistością (sprawdzanie, weryfikacja, checking).
- Testujemy, bo chcemy sprawdzić, czy nasza rzeczywistość nie przestała spełniać naszych oczekiwań (testowanie regresywne).
- Testujemy, bo chcemy sprawdzić, czy są rzeczy, o których nie pomyśleliśmy, których nie byliśmy w stanie przewidzieć, a które mogą być istotne (testowanie eksploracyjne).

Jak widać checking jest sprawdzaniem naszych oczekiwań. Mogą one wynikać na przykład z udokumentowanych (bądź też nie) wymagań, ustaleń z klientem, czy z zespołem. Testowanie regresywne można rozumieć jako regularnie powtarzany checking po to, aby upewnić się, że system nie przestał spełniać naszych oczekiwań.

Problem w tym, że na tym etapie najczęściej kończy się postrzeganie testowania. Często jednak okazuje się, że sprawdzenie tylko tego, czego oczekujemy od systemu jest niewystarczające, aby produkt spełniał określone kryteria jakości.

I tutaj pojawia się testowanie eksploracyjne, w którym uczymy się systemu i odkrywamy rzeczy, które wychodzą poza nasze oczekiwania. Szukamy tego, o czym nie pomyśleliśmy i czego nie byliśmy w stanie przewidzieć, a co wciąż wpływa na postrzeganie jakości. Zawiedziony klient lub użytkownik, który jednak oczekiwał czegoś innego lub system, który zachowuje się

nie tak, jak powinien, choć nie określają tego żadne wymagania, czy kryteria akceptacji.

Sprawdzanie przynosi bardzo dużo wartości - zwykle są to kluczowe ścieżki systemu. Jednak pomijanie eksploracji może sprawić, że oddamy system z równie istotnymi błędami. A to, że o nich nie pomyśleliśmy nie robi żadnej różnicy klientowi, użytkownikom, czy samemu systemowi, który się wysypie.

Zatem krótka lekcja na dzisiaj - nie traktujmy testowania jako sprawdzania. Pamiętajmy, aby analizować, myśleć krytycznie, umiejętnie obserwować co się dzieje z naszym systemem i wychodzić poza ramy ustawione przez wymagania, czy acceptance criteria.

Wpis #2 Więcej o testowaniu eksploracyjnym

Zdaje sobie sprawę, że testowanie eksploracyjne może być początkowo trudnym tematem, dlatego postanowiłem go uzupełnić.

Zgodnie z tym co pisałem, testowanie eksploracyjne to szukanie tego, co nieoczekiwane, tego, o czym nie pomyśleliśmy. Czy to znaczy, że mogę klikać gdzie popadnie i patrzeć co się dzieje? Nie o to chodzi. Testowanie eksploracyjne nie ma być przypadkowe w takim sensie. Wręcz powinno być przemyślane, zaplanowane i ustrukturyzowane - nawet, jeśli nie mamy konkretnego planu dobry tester eksploracyjny zawsze gdzieś w głowie ma poukładane, co chciałby zrobić. Ale nie chodzi tu o przypadki testowe.

W pierwszej kolejności myślimy o tym, co robi nasz system (jeśli nie wiemy, to musimy się go nauczyć), jakie obszary są dla niego kluczowe, a jakie są ryzykowne, złożone, posiadające wiele zależności.

Następnie planujemy pracę i to ile czasu chcemy poświęcić na grzebanie w danym obszarze, czyli tworzymy priorytety i zarys naszych działań.

Gdy podchodzimy już do testowania, rozpoczynamy eksperyment. Wchodzimy w dany obszar - ten, który w naszej opinii na to zasługuje. Zaczynamy od rzeczy prostych i oczywistych. Zwykle robimy to, co standardowo zrobiłby użytkownik. W międzyczasie obserwujemy, szukamy tzw. bad smells, nowych ścieżek i sposobów na używanie systemu, zadajemy sobie pytania i kwestionujemy to, co widzimy.

W trakcie eksploracji to nie przypadek kieruje działaniami, a obserwacja i myślenie krytyczne. Pytania typu: “A co jeśli...?”, “A gdyby tak spróbować tutaj...?”, “Czy na pewno o to chodziło...?” są wartościowe. Pamiętajmy, że testowanie eksploracyjne dotyczy wszystkich warstw systemu, nie tylko GUI.

Jaką wartość przynosi eksploracja w porównaniu do standardowego sprawdzania? Sprawdzanie weryfikuje nasze oczekiwania - zwykle zawarte w wymaganiach lub acceptance criteria. Są to kluczowe ścieżki, które mają działać. Skoro są kluczowe, to ich weryfikacja wydawałaby się najważniejsza i najbardziej wartościowa, a eksploracja to tylko tak dla pewności. Niestety tak nie jest.

Okazuje się, że w dobie pracy zwinnej, DevOps, CI i generalnie dobrych procesów zapewniania jakości, zwykle nie mamy problemu w tych kluczowych rzeczach. W trakcie sprawdzania zazwyczaj wszystko działa (zwykle, nie zawsze). Ta obserwacja wychodzi z mojego doświadczenia i tego, o czym czytam w literaturze i słyszę od innych.

Miejsca, w których faktycznie można znaleźć problemy to właśnie te, o których nie pomyśleliśmy lub których nie przewidzieliśmy. A często są to równie istotne błędy, co te dotyczące wymagań ustalonych z góry. Stąd warto eksplorować, ponieważ ani sprawdzanie, ani testy automatyczne nigdy nie wykryją takich problemów. Testy automatyczne mają to do siebie, że pisze je ktoś zgodnie z jakimiś oczekiwaniami, a skoro mamy problemy z rzeczami nieoczekiwanymi to ... (choć czasem test automatyczny wykryje więcej niż nam się wydaje)

Eksploracja powinna być uzupełnieniem naszych działań testowych. Jednak nie musimy osobno sprawdzać, a osobno eksplorować. Najprościej zasiadając do testowania jest wyposażyć się w odpowiednie metodyki i narzędzia, i stosować je tam, gdzie widzimy taką potrzebę, naprzemiennie.

Wpis #3 Heurystyki testowe

Zostałeś postawiony w roli testera albo jesteś testerem Siedzisz, testujesz, głowa się pali, nie masz już pomysłów ale czujesz, że to nie koniec. Co robisz? Przypominasz sobie o heurystykach.

Czym są heurystyki? Heurystyki to wysokopoziomowe, generyczne, uproszczone wzorce, zasady, metody, które z jednej strony mówią nam wiele, z drugiej nie mówią nam nic - zależy jak z nich korzystamy.

“Unikaj overengineeringu” jest jedną z takich zasad. Istnieje powszechne przekonanie, że jest to zła praktyka, ponieważ, nie wchodząc w szczegóły, jej negatywne rezultaty przewyższają pozytywne. Kiedy popełniam overengineering? To już jest pytanie, na które każdy powinien sobie odpowiedzieć sam.

Kolejnym przykładem już stricte testerskim jest heurystyka wartości granicznych (również nazywana metodą projektowania testów poprzez analizę wartości granicznych). I znów, metoda ta nie podrzuca nam gotowego rozwiązania, czy przypadków testowych. Przypomina, że istnieje coś takiego jak wartości graniczne i swoje przypadki testowe należy projektować z uwzględnieniem tej metody. To znaczy, że w pierwszej kolejności musimy poznać założenia i intencje stojące za daną metodą, następnie zastanowić się, w jaki sposób możemy ją wykorzystać w naszym kontekście (i czy w ogóle warto), a w dalszej kolejności świadomie ją stosować.

W ten sposób możemy dodawać do swojego arsenału testerskiego kolejne narzędzia, które pomogą nam stać się testowym ninja, a może bardziej testowym komandosem.

Na pewno będę wspominał o różnych przydatnych heurystykach, ale na sam początek przytoczę już kilka:

Heurystyka wymaganiowa - pierwsza i podstawowa. Wymagania klienta muszą zostać zaimplementowane. Testujemy tak, aby wiedzieć, że są.

Heurystyka historyczna - nasze doświadczenia z problemami, na które natrafiliśmy do tej pory wzbogacają mapę naszych poszukiwań problemów. Jest duża szansa, że to, co do tej pory sprawiało problemy zwyczajnie się powtórzy. Testując przeanalizujmy, gdzie najczęściej odbywały się potknięcia i upewnijmy się, że tym razem ich tam nie ma (właściwie tę zasadę można zastosować nie tylko w pracy testera ale i w życiu!).

Heurystyka porównawcza - nasze doświadczenie w pracy, jak i poza nią wystawia nas na różnego rodzaju aplikacje i funkcjonalności. Przypomnijmy sobie miejsca podobne do tych, z którymi aktualnie mamy do czynienia i zastanówmy się, jakie problemy tam widzieliśmy lub co wyjątkowo sobie ceniliśmy. Warto sprawdzać, czy nasz system to zapewnia.

Istnieje wiele heurystyk, choć rzecz jasna nie ma sensu przechodzić przez nie wszystkie w trakcie testowania. Jednak od czasu do czasu można poznawać nowe i przypominać sobie o istnieniu tych, które już poznaliśmy, szczególnie, gdy czujemy się wypaleni z pomysłów.

Warto również tworzyć własne heurystyki - na przykład jeśli widzimy, że w zespole często mamy problemy, które wynikają z komunikacji, testujmy tak, aby wyjść temu naprzeciw. Zastanówmy się, jakie błędy mogły wyniknąć z powodu tego problemu w tym konkretnym przypadku. Na przykład, jeśli klient przekazuje wymagania w sposób mało zorganizowany i często jest sporo wersji, z czego tylko jedna jest właściwa, skupmy się na ustaleniu która i czy to, co mamy, na pewno jest tą właściwą.

Lektura uzupełniająca:
https://www.satisfice.com/download/heuristic-test-strategy-model

Wpis #4 7 Zasad Testowania

W świecie testowania istnieje coś takiego jak 7 zasad testowania, czyli podstawowe koncepcje, z którymi każdy początkujący tester spotyka się na początku swojej zawodowej drogi. Warto je znać, przypomnieć sobie o nich, bądź skonfrontować je z własnym doświadczeniem.

I Testowanie ujawnia problemy.

To znaczy, że nie gwarantuje ich braku. Testowanie zmniejsza prawdopodobieństwo, że w oprogramowaniu pozostają niewykryte defekty. Skoro nie jesteśmy w stanie znaleźć wszystkich błędów, to oddając system zawsze ponosimy jakieś ryzyko. Róbmy to więc świadomie. Z pomocą przychodzi nam wówczas analiza ryzyka. Zastanówmy się, z jakim ryzykiem mamy do czynienia tym razem. Co złego może się stać do czego nie możemy dopuścić. Przez co stracimy reputację, pieniądze i możemy mieć spore problemy. Testujemy w pierwszej kolejności, aby unikać ryzyka, którego nie możemy ponosić. W drugiej, (jeśli pozostaną nam zasoby) testujemy, aby unikać ryzyka, które możemy ponieść, ale wciąż wolimy je zminimalizować.

II Testowanie gruntowne jest niemożliwe.

Jeśli znajdziemy błąd w części systemu, który był już testowany, powinniśmy przynajmniej zastanowić się, dlaczego go pominęliśmy. Może zabrakło nam heurystyki, informacji? Uzupełnijmy braki na tyle, na ile jest to możliwe i wyciągnijmy lekcję na przyszłość.

III Wczesne testowanie oszczędza czas i pieniądze.

Istnieje przekonanie, że im później wykryjemy problemy, tym droższe stają się one w naprawie. Osobiście zgadzam się z tym tylko częściowo, ponieważ tyczy się to tylko pewnego rodzaju

problemów. Jeśli na przykład znajdziemy błąd związany z wymaganiami - dajmy na to, że wymagania były dwuznaczne - taki błąd może sprawić, że będziemy musieli przepisywać daną funkcjonalność i tym samym stracimy sporo czasu, o ile nie wykryjemy go, zanim rozpocznie się implementacja. Jeśli natomiast znajdziemy błąd, którego poprawka wymaga np. zmiany wartości zmiennej, taki błąd można szybko poprawić. Jeśli nie niesie on za sobą żadnych dodatkowych implikacji, wtedy nie ma znaczenia, czy wykryjemy go wcześnie, czy późno.

IV Zasada Pareto - 80 % błędów znajduje się w 20% systemu.

Osobiście tego nigdy nie odczułem. Ciekaw jestem, czy ktoś z was to zauważył. Niezaprzeczalnym jest, że często popełniamy błędy w podobnych okolicznościach lub w podobnych miejscach. Ja na przykład często spotykam się z problemami związanymi ze zmiennymi środowiskowymi. Ale nie określiłbym tego jako zasady Pareto, gdyż zasada Pareto odnosi się do konkretnego obszaru kodu źródłowego np. danego modułu.

V Paradoks pestycydów - ciągłe powtarzanie tych samych testów prowadzi do sytuacji, w której przestają one wykrywać nowe defekty.

Problemy z regresją zdarzają mi się bardzo rzadko.. Unit testy robią robotę nie przepuszczając kodu dalej. Dlatego, jeśli mam ograniczone zasoby i muszę dokonywać jakichś wyborów więcej czasu poświęcam na testowanie eksploracyjne, czyli testowanie nowych rzeczy. Jeśli zaś chodzi o regresję, traktuje ją jako polisę ubezpieczeniową i zabezpieczam tylko to, co przynosi wartość biznesową. Mówiąc krótko, nie sprawdzam za każdym razem wszystkich przypadków testowych, nie piszę też obszernych automatycznych testów systemowych. Upewniam się, czy podstawowa ścieżka potrzebna do realizowania wartości biznesowej nie przestała działać. Jeśli jakiś edge case sie wysypie, przeżyjemy to.

VI Testowanie zależy od kontekstu.

Gdybyśmy byli w stanie przetestować wszystko, czyli poświęcić testowaniu niewyobrażalną porcję czasu, wtedy to stwierdzenie nie miałoby sensu. Jednak odkąd musimy wybierać, co i jak testujemy, próbujemy wyciągać jak największą wartość przy ograniczonych zasobach. Priorytetyzujemy i targetujemy testowanie biorąc pod uwagę to, czego oczekuje klient, jak system będzie używany, co przynosi w nim najwięcej wartości lub gdzie znajduje się największe ryzyko.

VII Przekonanie o braku błędów jest błędem.

Ten punkt nawiązuje w dużym stopniu do punktu pierwszego. Jeśli decydujemy się na release nie róbmy tego w oparciu o przekonanie, że jeśli tester przetestował całość, to tam na pewno nie ma błędów. Jeśli mamy zaufanie do testera i procesów zapewniania jakości możemy założyć, że solidna część funkcjonalności została przetestowana, a tester oddając ją do wydania poprawnie ocenił ryzyko (jeśli nie mamy zaufania do testera albo procesu, porozmawiajmy z testerem o tym, co i jak testował). Niemniej zespół powinien zawsze być przygotowany na to, że na produkcji pojawią się błędy i trzeba będzie robić rollback, czy hotfix.

Wpis #5 Trzy metody testowania

Istnieje coś takiego, jak metody testowania lub inaczej poziomy, na których można testować system. Mam na myśli testowanie czarnoskrzynkowe, białoskrzynkowe i szaroskrzynkowe. W trzech feedach omówię każdy z osobna zaczynając od testowania czarnoskrzynkowego.

Testowanie czarnoskrzynkowe, z ang. black-box testing to metoda testowania, która polega na ograniczeniu swojej aktywności do graficznego interfejsu systemu. Tester skupia się na tym, co oferuje interfejs i swoje przypadki testowe projektuje tylko na podstawie tego interfejsu. Krótko mówiąc skupiamy się

na inpucie i outpucie systemu, a to, jak działa pod spodem nas nie interesuje.

Do technik wykorzystywanych przy tego typu testowaniu zalicza się:

- Analizę klas równoważności (equivalence partitioning)
- Analizę wartości brzegowych (boundary values analysis)
- Tablice decyzyjne (decision tables)
- Wykres przyczynowo-skutkowy (cause-effect graphing)

Te techniki są na tyle wymowne, że nie będę ich omawiać szczegółowo. Jeśli ktoś z was jest zainteresowany wystarczy wpisać w Google nazwę po angielsku i szybciutko wszystko stanie się jasne.

Po co testować czarnoskrzynkowo? Mówi się, że to najprostsza forma testowania, gdzie próg wejścia dla testera jest najniższy. Otóż i tak, i nie. Faktycznie próg wejścia wydaje się być niski, ale w przypadku bardziej złożonych interfejsów, aby testować skutecznie znów potrzeba doświadczenia i wiedzy na temat tego, jak szukać problemów. W przypadku prostych interfejsów próg wejścia rzeczywiście jest niski. Ale to równie dobrze może oznaczać, że większość logiki biznesowej kryje się pod spodem i w takiej sytuacji testowanie czarnoskrzynkowe może wiele nie wnieść.

Niemniej, jeśli chodzi o zalety:

Pierwsza i najważniejsza: **testujemy z punktu widzenia użytkownika**. Powstrzymujemy się od myślenia technicznego, a myślimy bardziej domenowo, wchodzimy w buty usera i zaczynamy korzystać z aplikacji. Prócz problemów funkcjonalnych pozwala nam to lepiej dostrzec problemy użyteczności.

Po drugie: **unikamy uprzedzenia technicznego**. Bez znajomości logiki, która realizuje kod źródłowy projektujemy przypadki testowe używając innego spojrzenia, niż gdy wiemy, co tam się dzieje pod spodem. Testując w ten sposób jest duża szansa,

że wpadniemy na rzeczy, na które nie wpadlibyśmy znając kod, który pokierował by naszą kreatywność w nieco inną stronę.

Po trzecie: mówi się, że **jedną z korzyści testowania czarnoskrzynkowego jest to, że nie trzeba znać się na programowaniu,** by do niego przystąpić. Mam mieszane uczucia, czy można uznać to za zaletę.

Z drugiej strony oczywiście są i wady takiego testowania.

Jeśli ograniczamy się tylko do testowania czarnoskrzynkowego, możemy pominąć sporą część problemów, które kryją się w jaskiniach backendu. Stąd, można rozpoczynać testowanie od metody czarnoskrzynkowej, aby zebrać te tzw. low hanging fruits, a następnie wziąć się za łopatę i zacząć kopać głębiej.

Wpis #6 Testy białoskrzynkowe

Z ang. white-box testing, czyli testy białoskrzynkowe, testy białej skrzynki, czy testy strukturalne. Są to testy, w których tester ma wgląd do kodu źródłowego systemu, jego architektury, konfiguracji czy środowiska.

Do testów białoskrzynkowych aktualnie można zaliczyć code review (zwane również analizą statyczną) oraz jednostkowe i integracyjne testy automatyczne wykonywane z wykorzystaniem wglądu do kodu źródłowego.

Code review wykonujemy zwykle, aby ocenić jakość jakość kodu, jego architekturę, znaleźć błędy logiczne, poprawić utrzymywalność i rozszerzalność. Standardowo code review wykonuje więcej niż jedna osoba. Każdy ma na to swój sposób i swoje spojrzenie, dzięki czemu stosunkowo tanio jesteśmy w stanie zidentyfikować problemy różnego sortu. Tanio, ponieważ przejrzenie kodu wymaga mniej czasu niż uruchomienie go, doprowadzenie do odpowiedniego stanu, załadowanie danych testowych i wywołanie odpowiednich linijek, aby sprawdzić ich działanie. Ja sam korzystam z możliwości zrobienia code review,

żeby spojrzeć na kod na swój własny sposób. Zwykle robię to w stylu end-to-end, czyli identyfikuję wejście, a następnie przechodzę całą ścieżkę aż do wyjścia. Dodatkowo sprawdzam pokrycie funkcjonalności testami, aby uniknąć powtarzania tych samych testów na wyższym poziomie. Samo code review pozwala mi też poznać system lepiej, dzięki czemu zyskuję więcej materiału do analizy i planowania testowania.

Testy automatyczne na poziomie unitowym mają niewątpliwie tę zaletę, że oferują sporo możliwości. Na tym poziomie przetestować można prawie wszystko. Prawie, ponieważ czasem jesteśmy blokowani przez brak narzędzi, brak ich znajomości lub przez limity kreatywności. Niemniej tego rodzaju testy są bardzo skuteczne, szybkie, adresują wiele use case'ów i myślę, że można śmiało powiedzieć, że tego typu testy to fundament każdego projektu. Wiele źródeł zaleca spychanie testów funkcjonalnych w dół, czyli właśnie do poziomu unit testów, jeśli tylko się da.

Jest wiele technik projektowania takich testów, z czego dla przykładu można wyróżnić:

- Pokrycie instrukcji
- Pokrycie rozgałęzień
- Pokrycie decyzji

Czy warto je stosować? Trudno powiedzieć. Każdy ma swoje zdanie. Jedni lubią mierzyć pokrycie kodu i uważają to za efektywne podejście. Innym zależy głównie na tym, aby mieć testy pokrywające funkcjonalności i nie testować implementacji. Pewnie można to połączyć i pisać takie testy, które pokrywają funkcjonalności poprzez pokrycie instrukcji, rozgałęzień, czy decyzji.

Czy w takim razie testy na tym poziomie to sama śmietanka? Niestety zawsze znajdzie się jakieś ale.

Przede wszystkim utrzymywalność. Jeśli zmieniam kod produktu zdarza się, że muszę też zmienić testy. Z jednej strony to dobrze. Z drugiej, jeśli refaktoruję, tzn. nie zmieniam funkcjonalności, tylko sposób implementacji, to czy testy powinny failować? Nie zawsze życie jest takie kolorowe i nie zawsze pisanie testów,

które tylko testują funkcjonalność i nie są oparte o implementacje, będzie proste. Zawsze jednak możemy do tego dążyć. Ponadto, jeśli testy już się wysypią, to ile czasu potrzebuję na ich poprawę? Mało? To dobrze. Dużo? Może to już nie najlepiej. Testy, o ile są bardzo pomocne, potrafią również być ogromnym ciężarem. Stąd naszym celem jest wypracować sobie sposób, aby te proporcje zmienić na korzyść tej pomocnej części. Trudne, ale prawdziwe!

Wpis #7 Testowanie Gruntowne (Exhaustive Testing)

Będzie po angielsku, bo oryginał częściej oddaje sedno, niż nieudolne tłumaczenie.

‘Exhaustive testing is when the tester is too exhausted to continue’.

Testowanie gruntowne jest niemożliwe. Jednak grzechem w tym wypadku jest nadmierny optymizm i zbyt silna wiara w swoje umiejętności projektowania testów.

Jeśli kończymy testowanie z myślą, że przetestowaliśmy już wszystko, to najprędzej jesteśmy w błędzie. Nie dopuszczamy wtedy możliwości istnienia innych błędów, których nie udało nam się znaleźć. Jednocześnie nie wracamy już do tych obszarów, nie myślimy o nich, mamy fałszywe poczucie pewności, że tam już żadnych błędów nie ma. Zdarza się, że po teoretycznym zakończeniu testowania danego obszaru, wciąż udaje się odnaleźć inne błędy, gdy od czasu do czasu wracamy do tego obszaru z nowymi pomysłami. Dlatego nie warto się tak oszukiwać.

Koniec testowania zwykle ma miejsce, gdy uznajemy, że przetestowaliśmy już istotne obszary, gdy udało nam się zminimalizować ryzyko lub, gdy na dany moment nie przychodzi nam już nic innego do głowy.

Wpis #8 Testy szaroskrzynkowe

Ostatni typ, czy poziom testów, to testy szaroskrzynkowe. Właściwie jest to fuzja poprzedników. Oba typy przynoszą na tyle dużo różnych korzyści, że naturalną drogą jest skorzystanie z obu, aby wyciągnąć jak najwięcej wartości z testowania.

Istnieją organizacje w których testy czarnoskrzynkowe wykonują testerzy, testy białoskrzynkowe wykonują programiści, a jedni niekoniecznie wiedzą, co robią drudzy. W sytuacji, gdzie granica między tymi dwoma poziomami została wyraźnie zaznaczona, dużo ciężej jest wykonywać testy obu typów, a wręcz testera wychodzącego z poziomu czarnej skrzynki można postrzegać niekorzystnie. Jednak zostawię skrajności za nami i skupię się na środowisku, w którym pracuje nasza firma i tu na szczęście mamy moc możliwości. Tak poważnie są systemy i metodyki, w których testerzy czarnoskrzynkowi sprawdzą się świetnie. Patologia powstaje wtedy, gdy dokładnie to samo podejście w ciemno stosuje się tam, gdzie być go nie powinno.

Mówiąc o testach szaroskrzynkowych wyobrażam sobie testera, który analizując ryzyko, planując, wykonując, a nawet raportując czynności testowe bierze pod uwagę nie tylko to, co udostępnia mu interfejs graficzny użytkownika ale robi to w oparciu o wiedzę o tym, jak wygląda kod źródłowy, architektura systemu, jego konfiguracja i środowisko. I dotyczy to zarówno weryfikacji jak i eksploracji.

Taki poziom, a w zasadzie oba poziomy, pozwalają wykrywać problemy na różnych poziomach systemu. Wzrasta nie tylko zakres poszukiwań ale też ich skuteczność. Tester, który wie, gdzie szukać, jest w stanie dokonywać trafnych wyborów priorytetyzując swoje działania.

Należy się jednak wystrzegać kilku rzeczy przy takim podejściu, jeśli jesteś testerem, że tak to nazwę dedykowanym, a nie osobą, która przyjęła rolę testera na czas robienia review.

Po pierwsze kod analizować można godzinami. Trzeba wiedzieć, kiedy powiedzieć stop i uznać, że tyle, ile wiem jest wystarczające i przejść dalej. Co więcej, robiąc code review robimy go w konkretnym (testerskim) celu. Inaczej code review zrobi programista, a inaczej tester.

Po drugie tester, który z reguły nie jest programistą, próbuje radzić sobie z mniej i bardziej technicznymi rzeczami systemu. Przez to może wpaść w pułapkę niekończącej się nauki technologii. Owszem, uczymy się przez całe życie, nowe technologie powstają i trzeba je znać, ale wystarczy, jeśli znamy je do pewnego stopnia. Wystarczy, że je rozumiemy i być może znamy problemy, jakie są z nimi związane. Z czasem będziemy poznawać nowsze albo pogłębiać wiedzę o tych istniejących. Nie ma jednak sensu fiksować się na nauce w celu dogonienia programistów.

Po trzecie przechodząc na niższe poziomy systemu widzimy nowe możliwości, tzn. nowe test case'y, które być może warto sprawdzić. Mając ograniczony czas na testowanie, trzeba więc mocniej priorytetyzować i wybierać tylko te case'y, które w naszym mniemaniu niosą ryzyko.

Wpis #9 Heurystyka otaczającego nas świata

Przybliżę kolejną heurystykę zwaną World Heuristics. Jeszcze taka przypominajka - prócz tego, że heurystyki pomagają nam testować pamiętajmy, że równie dobrze pozwalają nam zapewniać jakość wbudowaną. Planując nasze rozwiązanie, jeśli użyjemy heurystyk i przemyślimy sobie co chcemy zrobić, a czego chcemy uniknąć jest duża szansa, że zapobiegniemy błędom, a tester będzie mógł wypić mniej kawy.

Heurystyka Świata opiera się o założenie, że system, który projektujemy dla użytkownika z punktu widzenia UX powinien być jak najbardziej zbliżony do świata, w którym nasi użytkownicy żyją na co dzień. Odwrotnie patrząc, powinien być

pozbawiony wszelkiej ‘techniczności’. Z natury ludzie lubią to, co znane, a to co nieznane sprawia dyskomfort. Wychodzenie poza strefę komfortu często oznacza rozwój ale w przypadku naszych użytkowników nie zależy nam na tym, aby wychodzili poza swoją strefę komfortu. Wręcz przeciwnie, zależy nam na tym, aby czuli się jak najbardziej komfortowo, gdy korzystają z naszych rozwiązań. Im mniej nasz system będzie techniczny, a będzie bardziej naturalny dla naszego użytkownika, tym wygodniej ten użytkownik będzie się czuł korzystając z niego.

Aby osiągnąć naturalność naszego systemu, warto zwrócić uwagę na naturalny język. Jeśli nasz system próbuje komunikować się z użytkownikiem używając terminologii technicznej zamiast języka naturalnego, możemy liczyć na to, że użytkownik nie zrozumie komunikacji. Będzie miał problem z jej zrozumieniem, bądź zrozumie ale nie będzie to dla niego fajne. Nasza komunikacja z użytkownikiem powinna być super szybka, tzn. powinniśmy używać języka, który jest dla użytkownika oczywisty, a wszystkie trudniejsze koncepcje warto upraszczać tak, aby pomóc użytkownikowi je zrozumieć.

Warto też zwrócić uwagę na interakcje z życia wzięte. Żyjąc w naszym świecie jesteśmy przyzwyczajeni do tego, jak jest on skonstruowany. W oparciu o to, czego doświadczamy budujemy sobie przyzwyczajenia, czy tzw. modele mentalne, aby ułatwiać sobie życie - coś w rodzaju cache’owania rzeczywistości. Przechodząc ze świata realnego do świata wirtualnego nie przełączamy swoich modeli. Korzystamy dokładnie z tych samych, które wykształciliśmy sobie w stosunku do świata realnego. Stąd projektując nasze rozwiązania warto doszukiwać się podobieństw do realnych obiektów i interakcji, i w ten też sposób je implementować. Dla przykładu panel użytkownika, gdzie może on coś konfigurować (ustawiać), może przypominać rozwiązania, z którymi użytkownik spotka się korzystając z pralki czy zmywarki. Będzie to dla niego znajome i tym samym intuicyjne. Swoją drogą taki rodzaj projektowania nazywa się `skeuomorphic design` - dla zainteresowanych.

Za każdym razem gdy projektujemy, czy testujemy warto się zastanowić, czy rozwiązanie jest intuicyjne, wygodne w użyciu,

oraz czy wymaga przerabiania swoich modeli mentalnych. Jeśli tak, to może zwyczajnie nie być atrakcyjne.

Trzeba pamiętać, że ten rodzaj projektowania nie jest złotym środkiem. Specjaliści od UX najlepiej wiedzą, jak z tego rodzaju koncepcji korzystać. Nam testując pozostaje tylko branie pod uwagę komfortu użytkownika i każdego sygnału mówiącego o tym, że nasze rozwiązanie wydaje się być nienaturalne. Jest to miejsce, w którym warto się zatrzymać i przemyśleć je jeszcze raz.

Wpis #10 Praca z macierzą testową

Z ang. testing matrix bądź traceability matrix - to taka tabelka, w której rozpisujemy przypadki testowe.

Załóżmy, że nasza funkcjonalność składa się z dwóch ścieżek i dwóch różnych konfiguracji, które wpływają na to, jak dana ścieżka przebiega. W ten sposób przypadki testowe zaczynają się mnożyć. 2 ścieżki, z czego każda może być skonfigurowana na dwa sposoby, dają 4 przypadki testowe.

W ten sposób powstaje nam macierz:

	Konfiguracja A	Konfiguracja B
Ścieżka A	Test 1	Test 2
Ścieżka B	Test 3	Test 4

Nie wygląda to aż tak strasznie. Często jednak ścieżek lub konfiguracji jest więcej. Tabela nam puchnie i przechodzenie wszystkich możliwości staje się okropnie czasochłonne.

Co daje nam używanie macierzy? Przede wszystkim pozwala uświadomić sobie mnogość możliwości, tzn. jakie mamy ścieżki, jakie mamy konfiguracje i w jaki sposób jedno wpływa na drugie.

Dzięki temu powstaje przejrzysty obraz tego, z czym mamy do czynienia.

W dalszej kolejności utworzenie takiej tablicy pozwala nam zdać sobie sprawę już na samym początku, że testowanie zajmie sporo czasu. Możemy wtedy zadać sobie pytanie, czy damy radę przetestować wszystko i czy w ogóle warto.

W ten sposób dochodzimy do trzeciej korzyści jaką jest priorytetyzacja. Czy zawsze warto testować wszystko? Czasem nie. Szczególnie, jeśli jesteśmy w jakikolwiek sposób ograniczeni. Tak więc patrząc na taką tablicę łatwiej jest nam ocenić, które przypadki testowe można wykonać jako pierwsze, bo dadzą nam najwięcej informacji zwrotnej lub pokryją największą część systemu. Na drugi plan możemy odłożyć przypadki testowe, które mniej więcej angażują podobne części systemu ale nie jesteśmy ich pewni. W takiej sytuacji możemy np. wykonać tylko kilka przypadków zakładając, że pozostałe dadzą podobne rezultaty. Na trzeci plan odkładamy przypadki testowe, które są skrajnościami, są mało możliwe, aby miały miejsce lub zwyczajnie jesteśmy ich na tyle pewni po wykonaniu poprzednich, że już nie ma sensu dalej tracić czasu. Trzeci etap to właśnie kwestia przemyślenia, czy w ogóle warto do niego przechodzić.

Czy zawsze należy stosować tablicę? Czasem funkcjonalność jest tak zbudowana, że nie da się jej zbytnio rozpisać w ten sposób. Nawet jeśli będzie się dało, może to nie mieć sensu. Wtedy wiadomo, że nie należy na siłę robić tablicy. Możemy również natrafić na funkcjonalność, która aż się prosi o tablicę. W takiej sytuacji, zwykle nie porywam się na tworzenie tablicy dopóki nie uznam, że może ona być spora. Tworzenie tablicy także zajmuje czas i jeśli uznam, że mogę się bez niej obejść, to jej nie robię.

O tablicach wiedziałem od samego początku swojej przygody z testowaniem, bo mówi się o nich często. Nie stosowałem ich jednak od razu. Zdarzało się tak, że zaczynałem coś testować, w trakcie pracy dostrzegałem mnogość możliwości i zaczynałem je gdzieś spisywać, żeby ułożyć sobie to wszystko w głowie. Jako efekt uboczny powstawało coś na kształt takiej tablicy. Pod koniec testowania orientowałem się, że w sumie gdybym od razu

przemyślał sobie funkcjonalność i gdyby taka tablica powstała na samym początku, nie testował bym wielu rzeczy, które przetestowałem (ergo straciłem czas). No ale, człowiek uczy się na błędach.

Wpis #11 Heurystyka danych

Dane są paliwem każdego systemu. Bez przepływu danych system stoi w miejscu. Dane determinują też działanie systemu. To one decydują jaka ścieżka zostanie wywołana, jaka decyzja zostanie podjęta i ile kodu zostanie do tego wykorzystane.

Mając na uwadze dane podczas testowania robimy trzy rzeczy:

- Dywersyfikujemy - to znaczy używamy takich danych, aby sterować pokryciem uruchomionego kodu. Zastanówmy się, co robi nasz kod, które jego fragmenty zostaną uruchomione, gdy te dane będą różne, np. uppercase/ lowercase, string albo integer, jeden obiekt lub dwa obiekty itd.
- Myślimy o stanie nawet, gdy wydaje nam się, że nasze serwisy są bezstanowe. Wszystko, co przechowujemy w bazie danych buduje stan i może wpłynąć na to, jak dany serwis się zachowa. Tak więc mamy dane nowe dla systemu ale także te dane, które już w tym systemie istnieją. Nowe dane muszą się poprawnie zaaklimatyzować, to po pierwsze. Po drugie muszą też pokojowo koegzystować z tymi danymi, które już w systemie istnieją. Z kolei te istniejące już dane, muszą nam służyć niczym posłuszni obywatele. Stąd, dane które wpuszczamy do systemu, gdy baza świeci pustkami to dopiero początek testowania.
- Myślimy o ograniczeniach, bo często jakieś istnieją. Tak, jak każda droga, która gdzieś prowadzi ma swoją przepustowość, tak i społeczność, która się tworzy, może działać inaczej i mieć inne potrzeby w zależności od tego,

jak wielka ona jest. Danych może być zarówno mało jak i dużo, mogą trafiać do systemu rzadko lub często, wolno lub szybko, mogą być ładne, ale też brzydkie itd...

Jeśli testujemy - wykorzystujemy dane. Jeśli wykorzystujemy dane, to dywersyfikujemy je. Jeśli dywersyfikujemy dane, to róbmy to mając na uwadzę specyfikę systemu.

Wpis #12 Więcej o klasach równoważności

Klasy równoważności, z ang. equivalence partitioning, to w zasadzie podstawowa technika projektowania testów, według której identyfikujemy tzw. input do systemu. Staramy się go przeanalizować pod kątem jego różnorodności tj. rozmiaru, zakresu czy np. mnogości i po takiej analizie wychodzi nam cała lista różnych możliwych inputów. Najprościej, z zakresu od 1-10 wychodzi nam aż 10 różnych inputów. Dodając do tego 11-100 wychodzi nam już 100 różnych inputów.

Aby nie tracić czasu na testowanie wszystkiego jak leci, zakładamy, że przedział od 1-10 zachowa się inaczej niż przedział od 11-100. Z kolei cyfra/liczba z danego przedziału, zachowa się dokładnie tak samo, jak inna cyfra/liczba z tego samego przedziału. W ten sposób wciąż dywersyfikujemy nasz input ale ograniczamy czas poświęcony na testowanie.

Pozostaje tylko pytanie, co to znaczy, że zachowa się tak samo i skąd to wiemy?

Po pierwsze oceniamy, czy nasz input poddawany jest jakiejś walidacji i identyfikujemy input, który nie będzie poprawnie zwalidowany. Tutaj wyznaczenie klas w zakresie walidacji i poza jej zakresem będzie proste. Po drugie, analizujemy rzeczy podchwytliwe (tricky). Np. 0, które wiadomo, że generuje problemy, gdy chcemy przez nie dzielić. Wyznaczając klasę równoważności, jej granice wyznaczamy za pomocą takich podchwytliwych przypadków, które wiemy, że zwykle stanowią problemy. Po trzecie, nasze klasy równoważności

powinny aktywować różne ścieżki kodu. Jeśli mamy dwie klasy równoważności, które wyznaczyliśmy sobie ot tak, a aktywują one dokładnie te same ścieżki w kodzie, to być może to nie są najlepsze klasy równoważności. Jeśli możemy, to zaglądamy do kodu. Jeśli nie możemy lub nie potrafimy, to zawsze można się domyślić, jak mógłby zostać rozdzielony kod w zależności od inputu.

Podsumowując: walidacja, różnorodność, pułapki, ścieżki kodu to jest to, co nas interesuje przy wyznaczaniu klas równoważności.

Wpis #13 Czym jest BDD

Behaviour Driven Development - czyli scenariusze Given When Then. Jeśli do tej pory widząc lub słysząc o BDD myślałeś o scenariuszach, to świetnie. Masz okazję zmienić swoje podejście do BDD i wrócić do podstawowych założeń tej koncepcji.

W implementacji BDD bardzo często widzi się scenariusze Given, When, Then czyli tzw. Gherkin syntax. Jednak scenariusze to tylko narzędzie, które niestety z czasem stało się sednem tej koncepcji, przysłaniając całkowicie to, co faktycznie ma realizować.

Sprawa jest prosta. Jak sama nazwa wskazuje BDD to development, który jest sterowany poprzez funkcjonalne wymagania systemu. Na cały ten proces składa się pozyskiwanie tych wymagań, rozprzestrzenianie ich jednolitej wizji wśród członków zespołu, wykorzystywanie technik, które pozwolą upewnić się, że nasz development będzie realizował te wymagania i weryfikacja, że te wymagania są realizowane, gdy development jest już skończony.

Podstawowym problemem, który miał zostać rozwiązany był fakt, że założenia funkcjonalne gdzieś po w drodze całego procesu projektowego się rozmywają i zaczynają być przysłaniane założeniami technicznymi oraz sposobami implementacji. Wartości biznesowe gdzieś się zatracają i na sam koniec developmentu okazuje się, że rozjechaliśmy się względem

oczekiwań klienta. Jeśli zespół korzysta ze scenariuszy, które korzystają z tzw. Gherkin syntax, to w zupełności nie oznacza, że wdrożył do swojego procesu BDD. To tylko znaczy, że korzysta ze scenariuszy. Aby scenariusze miały jakikolwiek sens, muszą powstawać przed implementacją, a sama implementacja powinna być prowadzona w oparciu o te scenariusze.

Scenariusze to tylko jedna z form dla kierowania developmentem. Można używać jakiejkolwiek innej formy, która będzie prezentować wymagania funkcjonalne. Grunt, aby te wymagania pokazywały szczegóły realizowania funkcjonalności biznesowej oraz przykłady, zarówno kiedy ta wartość jest realizowana niepoprawnie, jak i kiedy nie jest realizowana wcale. Wymagania również powinny być spisywane w takiej formie, która pozwoli na łatwą i oczywistą ich weryfikację. To pozwala zbudować przejrzysty obraz tego, czego oczekuje klient i w ten sposób pokierować developmentem tak, aby na sam koniec klient dostał system realizujący wartości biznesowe, zamiast realizować założenia techniczne w oderwaniu od tych wartości.

Wpis #14 Przerwij, zatrzymaj, rozpocznij na nowo, powtórz

Dzisiaj o heurystyce skupionej przede wszystkim na użytkowniku i na tym, w jaki sposób zwykle korzysta on z naszej aplikacji.

Happy path to ścieżka, która nie ma ani jednej czynności wymienionej powyżej. Happy path to ścieżka, na której skupiamy się w pierwszej kolejności, gdy implementujemy, a potem, gdy rozpoczynamy testowanie. Test korzystający z happy path można by nawet określić mianem smoke test'u. A może nawet i trzeba...

Czasem nazewnictwo robi dobrą robotę. Zależy jak je rozumiemy, czy może jaką mamy wiedzę np. o testowaniu. Smoke test to taki teścik wstępny, podstawowy, żeby tylko sprawdzić, czy sprzęt się nie spali po podłączeniu do prądu. Tzn. jeśli powiem, że taki

happy path to (zaledwie) smoke test, oznacza, że co? Zaledwie podłączyliśmy sprzęt do prądu. No to gdzie reszta testów?! To mi się podoba, uznaliśmy, że happy path to za mało, żeby zakończyć testowanie, a wystarczyło tylko użyć odpowiedniej terminologii.

Tak więc, jak jak już dowiemy się, że happy path działa, to lecimy dalej i szukamy kolejnych możliwości przepływu danych w systemie.

“Przerwij”, czyli sytuacja, gdy użytkownik się rozmyśla i nie chce dalej kontynuować danego procesu. Bywa tak, że podczas jakiegoś procesu system zaczyna gromadzić dane i zaczyna osiągać pewien stan. Co się dzieje z tymi danymi, czy stanem gdy proces zostanie przerwany? Czy nie wpływają one negatywnie na dalsze działanie systemu?

“Zatrzymaj”, czyli sytuacja, gdy użytkownik wstrzymuje realizację procesu ale zamierza do niego wrócić, np. po kawie albo po dłuższej wizycie w toalecie. Czy istnieje szansa, że proces nie zostanie dokończony? Co jeśli użytkownik zapomni o danym procesie? Czy ten proces będzie wisiał już do końca ? Czy będzie miał wpływ na działanie systemu?

“Rozpocznij na nowo” , choć brzmi podobnie do ‘powtórz’ , oznacza sytuację, w której użytkownik przerywa realizację procesu, aby rozpocząć od nowa, np. z inną konfiguracją. Z kolei “powtórz” oznacza ponowną realizację zaraz po tym, jak już raz udało nam się zrealizować proces po raz pierwszy. I tu sytuacja jest podobna do powyższej. Po pierwsze jest duża szansa na to, że użytkownik będzie chciał wykonać takie akcje. Po drugie, system musi pozwolić użytkownikowi na powtórzenie procesu, czy wykonanie go na nowo. Po trzecie, jeśli jednak nie ma takiej możliwości, bo mamy bug’a, to tracimy wartość biznesową.

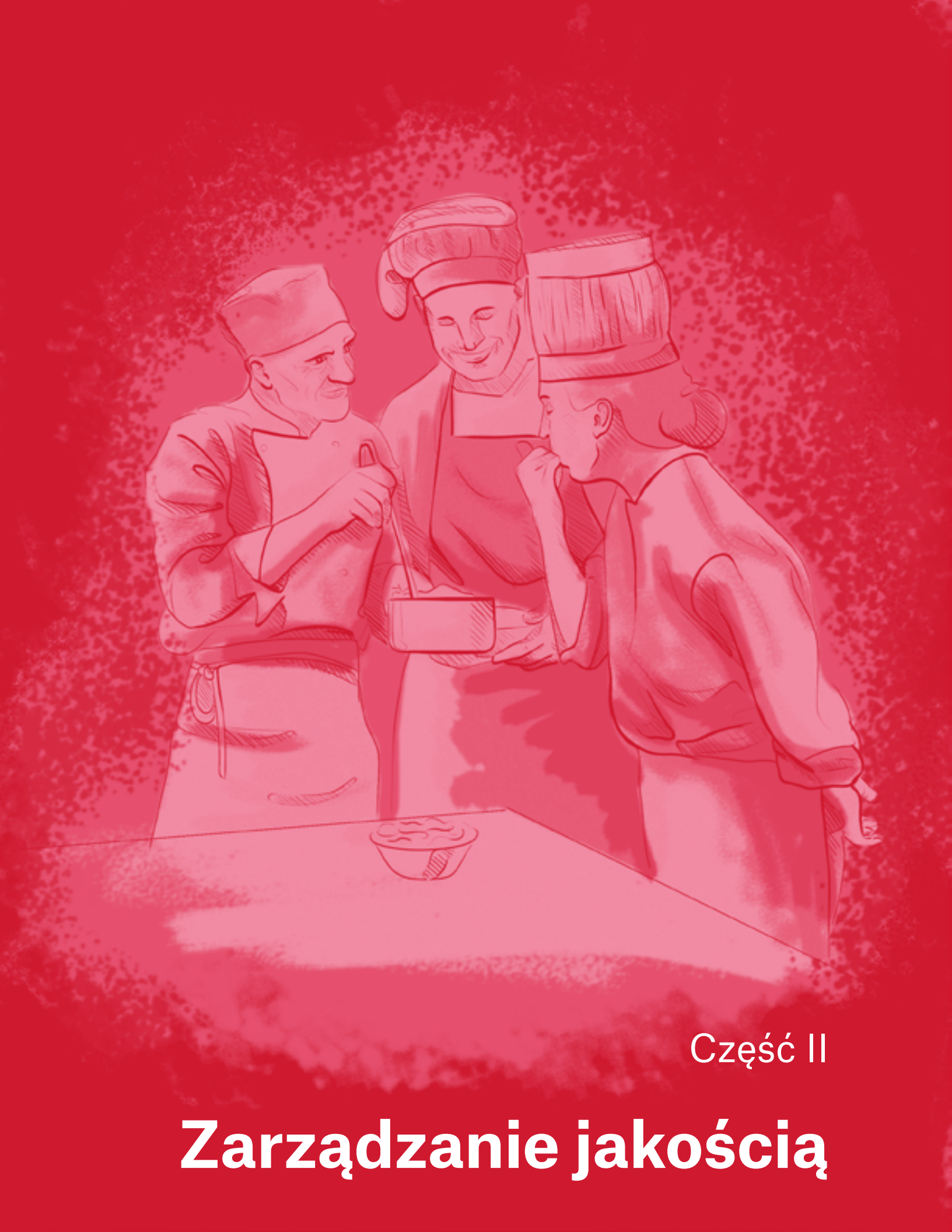
Na ten moment, opisałem głównie perspektywę systemu z którego korzysta użytkownik. Niemniej taką heurystykę można z powodzeniem stosować do procesów backendowych. Tutaj akurat nie mamy użytkownika ale wciąż pozostają nam procesy, a one mogą ulec zatrzymaniu (błąd), wstrzymaniu (chwilowy problem z połączeniem, długi czas oczekiwania), powtórzeniu - to akurat standard, rozpoczynaniu na nowo - powtarzamy proces

w przypadku błędu. To zależy od implementacji, ale wszystkie te sytuacje, to w większości obsługa błędów i wyjątków.

Na koniec - dlaczego happy path jest zwykle bardziej działający (choć nie zawsze) niż pozostałe ścieżki? Ja to widzę tak: zabierając się za implementację od czegoś trzeba zacząć. Zwykle zaczynamy od happy path i zostawiamy resztę case'ów na potem, jak już będziemy mieli coś, co działa. Następnie okazuje się, że implementacja jednak nie jest tak bezproblemowa, jak nam się wydawało i zaczynamy szarpać się z problemami. Cała nasza uwaga zaczyna przesuwać się na dopięcie podstawowego flow. Gdy już dochodzimy na koniec tej drogi cali brudni, spoceni, zmęczeni, pozbawieni jakiejkolwiek energii i entuzjazmu cieszymy się, że w końcu udało się domknąć tego taska i wypychamy go do code review. Nie chcemy już na niego patrzeć - olewamy pozostałe ścieżki, albo już nawet o nich nie pamiętamy.

To tylko moje wyobrażenie, sam czasem mam takie myśli, gdy pracuje nad automatyzacją testów. Ale ciekawy jestem czy wy widzicie to inaczej? Czy i dlaczego waszym zdaniem pomijamy różne case'y w trakcie implementacji?

Jako środek zaradczy poleca się rozpisywanie case'ów testowych przed implementacją, póki mamy jeszcze chęć, siły, niezmącone spojrzenie, i jasność umysłu.



Część II

Zarządzanie jakością

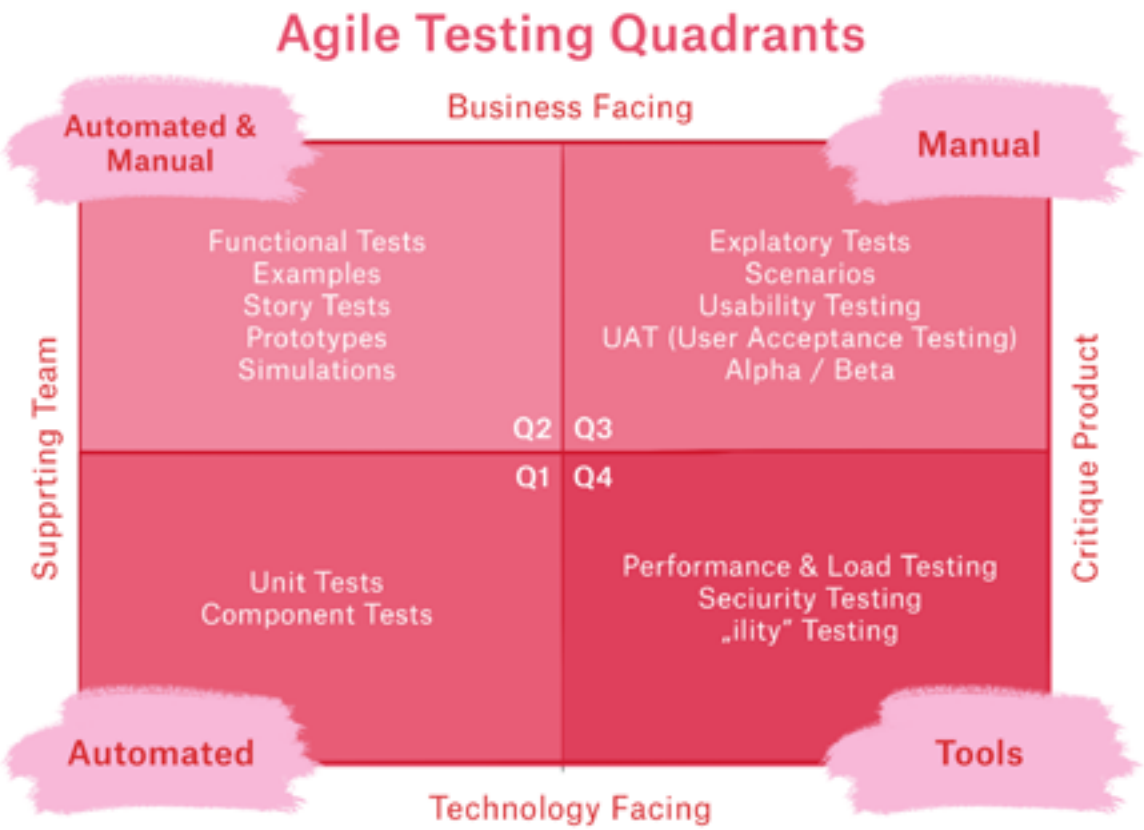
Wpis #15 Agile Testing Quadrants

W tym feedzie chciałbym zaprezentować tzw. thinking tool, które nazywa się agile testing quadrants. Narzędzie to wspiera proces analizy i planowania testów systemu (zdjęcie pod spodem); jak również dzieli się na cztery kategorie testów i cztery rodzaje potrzeb, które takie testy zapewniają.

W uproszczeniu są to następujące wymiary:

- Business Facing - testy weryfikujące, czy system spełnia oczekiwania biznesowe;
- Technology Facing - testy weryfikujące, czy system spełnia założenia funkcjonalne na poziomie technicznym;
- Supporting the Team - testy, które mają za zadanie wspierać zespół oraz;
- Critique the Product - testy, które mają za zadanie pokazać niedoskonałości systemu (tzw. orędownicy klienta).

Cztery kwadranty pokazują nam, że testowanie może zostać wykonane na różnych poziomach i w różnym celu. Poniżej przedstawiam obrazek do analizy, a w kolejnych feedach postaram się omówić każdy z nich troszkę szerzej.



Wpis #16 Kwadrant pierwszy

Na początku chciałbym zaznaczyć, że kolejność kwadrantów nie stanowi żadnego rodzaju mapy i może być planowana oraz wykonywana w dowolnej kolejności - w zależności od potrzeb i możliwości.

Kwadrant pierwszy, znajdujący się w lewym-dolnym rogu, jest opisany jako testy stawiające czoła aspektom technicznym i wspierające zespół. Składa się on z testów jednostkowych, modułowych etc. Mogą to być nawet testy integracyjne. Chodzi o te niskie poziomy testów, w których aby sprawdzić, czy funkcjonalność działa, używamy ciężkich technikaliów, tzn.: odpalamy customowo część kodu źródłowego aplikacji opakowanego we wszystkie potrzebne mocki, generatory danych, biblioteki klienckie itd. Chcemy w ten sposób sprawdzić,

jak aplikacja sobie radzi w sytuacjach, w których to my zdecydujemy się ją postawić. Takie testy charakteryzują się sporą elastycznością i prędkością.

Elastycznością, ponieważ stworzenie określonego stanu i danych wejściowych jest o niebo prostsze (czasem możliwe tylko z poziomu tych testów), niż w przypadku testów wykonywanych na uruchomionym całym serwisie.

- Prędkością, ponieważ ten sam zakres w przypadku testów e2e może zajmować kilka godzin, gdzie w przypadku testów z kwadrantu pierwszego będzie to raczej kwestia minut.

Są to testy techniczne, ponieważ tutaj "grzebiemy" bezpośrednio w kodzie źródłowym i prócz funkcjonalnych, testujemy także sporo technicznych aspektów systemu.

Dlaczego testy mają wspierać zespół? Ponieważ tego rodzaju testy to nasze pierwsze koło ratunkowe w momencie pracy z legacy code, refaktoryzacji, czy rozszerzania funkcjonalności (pewnie znalazło by się więcej use case'ów). I to programiści właśnie piszą je do swojego kodu. Czemu oni? Te testy to przede wszystkim pomoc dla programistów. Dają im ekstremalnie szybką informację zwrotną, o tym czy dokonywane zmiany nie modyfikują czegoś nieoczekiwanego. Dlatego jeśli czasem zastanawiamy się, czy nasze testy są dobre, możemy zadać sobie pytanie: Czy test który piszę da mi szybko znać gdy popsuje coś istotnego? Jeśli testy dają znać, gdy zmieniam implementacje a nie "psuję" niczego istotnego z funkcjonalnego punktu widzenia, to możemy się zastanowić, czy takie testy są mi faktycznie potrzebne (konsumują czas pisania, wykonywania, utrzymywania, debugowania i refaktoryzacji).

Do tego kwadrant sugeruje, że tego rodzaju testy są automatyzowane. Ciężko wyobrazić sobie pracę z systemem na tym poziomie bez automatyzacji, więc tutaj nie ma czego kwestionować.

Czasem słyszy się o tym, że testerzy piszą testy na tym poziomie, choć mnie ta praktyka wydaje się trochę bezcelowa. O ile tester zrobi dobrą robotę wykonując code review tych testów pod kątem wartości biznesowych, o tyle programista, który na co dzień pracuje z kodem źródłowym będzie wiedział lepiej z czym

zwykle ma problemy i jak sobie pomóc takimi właśnie testami. W dodatku, jeśli programista raz czy dwa będzie miał trudności w napisaniu testów do swojego kodu, istnieje duża szansa na to, że za trzecim razem postara się tak napisać kod, żeby był bardziej testowalny. W tym przypadku to temat- rzeka, ale między innymi taka jest właśnie moja opinia.

Wpis #17 Kwadrant drugi

Kwadrant drugi to testy ukierunkowane na wartość biznesową, które za zadanie mają wspierać zespół. To, w jaki sposób to robią wyjaśniłem w poprzednim feedzie. Tym razem testy wykonywane są jednak na innym poziomie.

Według obrazka z wpisu #15, tego typu testy mogą być zarówno zautomatyzowane jak i manualne. Jeśli jakikolwiek test może być zautomatyzowany, to tylko ten, dla którego mamy jakieś oczekiwania i wiemy czego się spodziewać w rezultacie, czyli testy z kategorii “checking”- weryfikujące nasze oczekiwania funkcjonalne względem systemu. Tego typu testy najczęściej wykonywane są na poziomie systemowym.

Tutaj zatrzymam się na chwile przy pytaniu: Czym są testy systemowe oraz czym różnią się od testów integracyjnych? Można powiedzieć, że testy systemowe są testami integracyjnymi, jednak testy integracyjne nie muszą być są testami systemowymi.

Testy integracyjne skupiają się na weryfikowaniu integracji, np. dwóch komponentów systemu: moduł - moduł / serwis - serwis.

Testy systemowe, choć pośrednio też sprawdzają integrację, przede wszystkim sprawdzają działanie systemu jako całości. Tak testując systemowo, np. system składający się z 4 serwisów, testujemy go w sposób end-to-end, tzn. wprowadzając dane na jednym końcu i oczekując konkretnych wyników na drugim.

Aby można było uzyskać wyniki, system korzysta z tylu serwisów, ile jest mu do tego potrzebne. Choć tak jak napisałem integracja tych serwisów jest koniecznością, tutaj mniej interesuje nas integracja sama w sobie, a bardziej dostarczenie konkretnej

wartości na sam koniec całego procesu przepływu danych. Z zasady, ten przepływ danych ma realizować wartości biznesowe, czyli pomagać naszym klientom rozwiązywać ich problemy efektywniej, niż gdyby robili to ręcznie. Stąd nazwa "Testy ukierunkowane na wartość biznesową". Czasem wykonuje się je manualnie. Najczęściej, gdy mamy do czynienia z daną funkcjonalnością, po raz pierwszy. Potem, albo je porzucamy, albo chcemy je powtarzać, by upewnić się, że następuje tylko pogress, nie regress. W tym momencie wykonywanie takich testów zaczyna być powtarzalne a my zaczynamy myśleć o automatyzacji, żeby zaoszczędzić sobie pracy (automatyzacja na tym poziomie to obszerny temat, główna zasada: przede wszystkim musi się opłacać). W ten sposób regularnie sprawdzamy, czy wartości biznesowe- czyli procesy odbywające się na poziomie całego systemu- nie przestały działać zgodnie z oczekiwaniami.

W ten właśnie sposób wspieramy siebie samych, rozwijając system (rozszerzanie, refaktoryzacja) tak samo, jak robimy to w przypadku testów z kwadrantu pierwszego. Z tą tylko różnicą, że robimy to na wyższym poziomie i informacja zwrotna przychodzi z lekkim opóźnieniem.

Czy takie testy są potrzebne? Myślę, że tak. Tak samo dają nam feedback o tym, czy nie popsuliśmy czegoś istotnego jak i testy na poziomie jednostkowym. Nie wszystko będziemy w stanie przetestować jednostkowo, szczególnie jeśli w grę wchodzi integracja, i w tej sytuacji na ratunek przychodzą testy właśnie na poziomie systemowym.

Testując manualnie na tym poziomie, osobiście robię to zwykle raz. W pierwszej kolejności czerpiąc z różnych źródeł, staram się zebrać do kupy wszelkie oczekiwania, jakie mogą się pojawić względem danej funkcjonalności, następnie je weryfikując. W dalszej kolejności staram się automatyzować te weryfikacje w realnym stopniu- biorąc pod uwagę ograniczony czas, automatyzuje tylko to, co uważam za krytyczne.

Wpis #18 Kwadrant trzeci

Powiedziałbym, ten najbardziej intrygujący. Ukierunkowany jednak tym razem na wartości biznesowe, stawiający dobro zespołu na drugim miejscu. Niekoniecznie niepomocny, ale czasem nie na rękę, chyba że ktoś potrafi go docenić.

Kwadrant trzeci to testy eksploracyjne, akceptacyjne, i/lub po prostu testy, które mają poddać nasz system krytyce. W tej krytyce nie chodzi o to, że coś, co miało działać, nie działa. Tym zajmują się testy z kwadrantu drugiego. Tutaj konfrontujemy to, co wyszło z naszej głowy (naszej albo klienta), z brutalną rzeczywistością. Odnajdujemy rzeczy, o których nie wiedzieliśmy, których nie przemyśleliśmy, lub które w naszej głowie wyglądały inaczej. Poddajemy system otwartej krytyce poprzez interakcję z funkcjonalnościami dostarczającymi wartości biznesowe. Krótko mówiąc, doszukujemy się dziury w całym.

Myślenia krytycznego (na ten moment?) nie da się zautomatyzować, a więc tego rodzaju testów również. Choć często słyszy się opinie o automatyzacji testowania; autorzy tych opinii zapominają dodać, że odnoszą się do automatyzacji sprawdzania (checking).

Testowanie akceptacyjne zwykle robi (powinien, co ma to dla mnie najwięcej sensu), klient lub osoba reprezentująca stronę klienta. Weryfikuje on pod różnym kątem, czy dostarczone rozwiązanie działa i czy jest zgodne z ich oczekiwaniami. Może się zdarzyć tak, że rozwiązanie będzie działać, ale nie będzie służyć wystarczająco dobrze. Może być nieatrakcyjne, nieefektywne mimo, że wykonaliśmy je zgodnie z wymaganiami klienta.

Testowanie eksploracyjne z kolei zwykle robi tester, osoba doświadczona (mniej czy więcej) w zawiłościach systemów, która wie jak eksplorować. Wcześniej wyjaśniałem już, czym jest testowanie eksploracyjne, ponieważ jednak ten rodzaj testowania przysparza najwięcej problemów myślę, że kolejna definicja będzie przydatna (i w jęz. angielskim).

“Testing is the process of evaluating a product by learning about it

through exploration and experimentation, which includes: questioning, study, modeling, observation and inference, output checking, etc."

Często to właśnie ten kwadrant dostarcza nam informacji o błędach, które potem "wychodzą" na produkcji. Im więcej czasu na niego poświęcimy, tym teoretycznie tych informacji powinniśmy zebrać więcej. Wiadomo, w pewnym momencie trzeba też powiedzieć stop i zaakceptować ryzyko. Ciężko nadawać priorytety i wartości tym kwadrantom, ponieważ każdy wnosi inną wartość do jakości systemu. Jeśli miałbym to zrobić, byłoby to z uwagi na częstotliwość występowania błędów, jakich do tej pory doświadczyłem. Na tej podstawie, oceniłbym kwadrant trzeci jako najbardziej godny uwagi.

Wpis #19 Kwadrant czwarty

Wygląda na to, że mamy już sporo testów za sobą. Podsumujmy więc. Dzięki poszczególnym testom:

- Testy jednostkowe - możemy bezpiecznie refaktorować;
- Testy sprawdzające - wiemy, że system działa;
- Testy eksploracyjne - upewniamy się, że nie wyskoczy nam niespodzianka na produkcji.

Cudownie! To teraz na koniec sprawdzimy, czy system jest użyteczny, bezpieczny, wydajny, niezawodny, rozszerzalny, czy utrzymywalny... Zaraz! Sporo tych testów jak na sam koniec. Mamy to wszystko upchnąć w ostatnim sprincie? A co jak trzeba będzie coś poprawiać?

Kwadrant czwarty obejmuje testy zwane "ilities" - od angielskich nazw tych testów: sacalability, usability etc. Po polsku również znane jako testy niefunkcjonalne. Czyli znów, konfrontacja z brutalną rzeczywistością, tylko tym razem od strony technicznej. To znaczy, że eksplorujemy system w poszukiwaniu niedociągnięć

technicznych, które równie mocno mogą wpłynąć na biznes, jak niedociągnięcia funkcjonalne.

O testach niefunkcjonalnych myśli się najmniej. Jednak często okazują się to być najtrudniejsze testy, jakie można wykonywać. Testy z zakresu wydajności, czy bezpieczeństwa to twardy orzech do zgryzienia. Dlaczego?

Dla przykładu testy wydajności. Zdefiniowanie endpointa, który może być mocno obciążony, nie jest aż tak trudne. “Strzelanie” do niego używając benchmarka to w zasadzie jedna linijka w terminalu. Obserwowanie czy system się nie wysypie też nie jest trudne. Tylko to- niestety- ułamek tego, czym jest testowanie wydajności. Środowisko produkcyjne często poddaje system takim obciążeniom, które ciężko jest przewidzieć i zasymulować. Stąd testowanie wydajności to ogromna praca analityczna, w której wykonanie testów i zebranie wyników to tylko uwieńczenie. Problem z tego rodzaju testami to problem góry lodowej. W przypadku dużego ryzyka i trudnych wymagań danego aspektu niefunkcjonalnego, potrzeba naprawdę dobrych specjalistów, którzy wiedzą co robią, żeby testy tego typu przyniosły prawdziwą wartość. Potem i tak okaże się, że na produkcji wyszło jeszcze coś innego. To w skrócie o technicznym aspekcie testów niefunkcjonalnych.

Patrząc biznesowo, testy z tego kwandrantu mają nas zapewnić, czy system będzie potrafił przynosić nam wartość ($$$) nieprzerwanie, czy będziemy mogli zarabiać na wszystkich użytkownikach korzystających z naszego systemu w danej chwili, czy tylko na części tych użytkowników. Czy nasi użytkownicy będą wręcz błagać o możliwość korzystania z niego, czy też przejdą obojętnie do konkurencji z powodu swego niezadowolenia. Czy będziemy narażeni na kary umowne lub inne pozwy i skandale. Czy utrzymywanie i rozwijanie systemu nie będzie zbyt kosztowne i czy będzie żeby nam się to opłacać.

Wydawałoby się, że testy z kwadrantu czwartego są szalenie istotne. Jednak tak jak napisałem w akapicie drugim, zwykle spychane na sam koniec- z różnych powodów. Nie będę określał, czy to źle czy dobrze. Napiszę tylko, że jeśli mają być wykonywane na sam koniec, postarajmy się mieć świadomość z czym to się wiąże. Być może na dany moment fazy projektowej nie

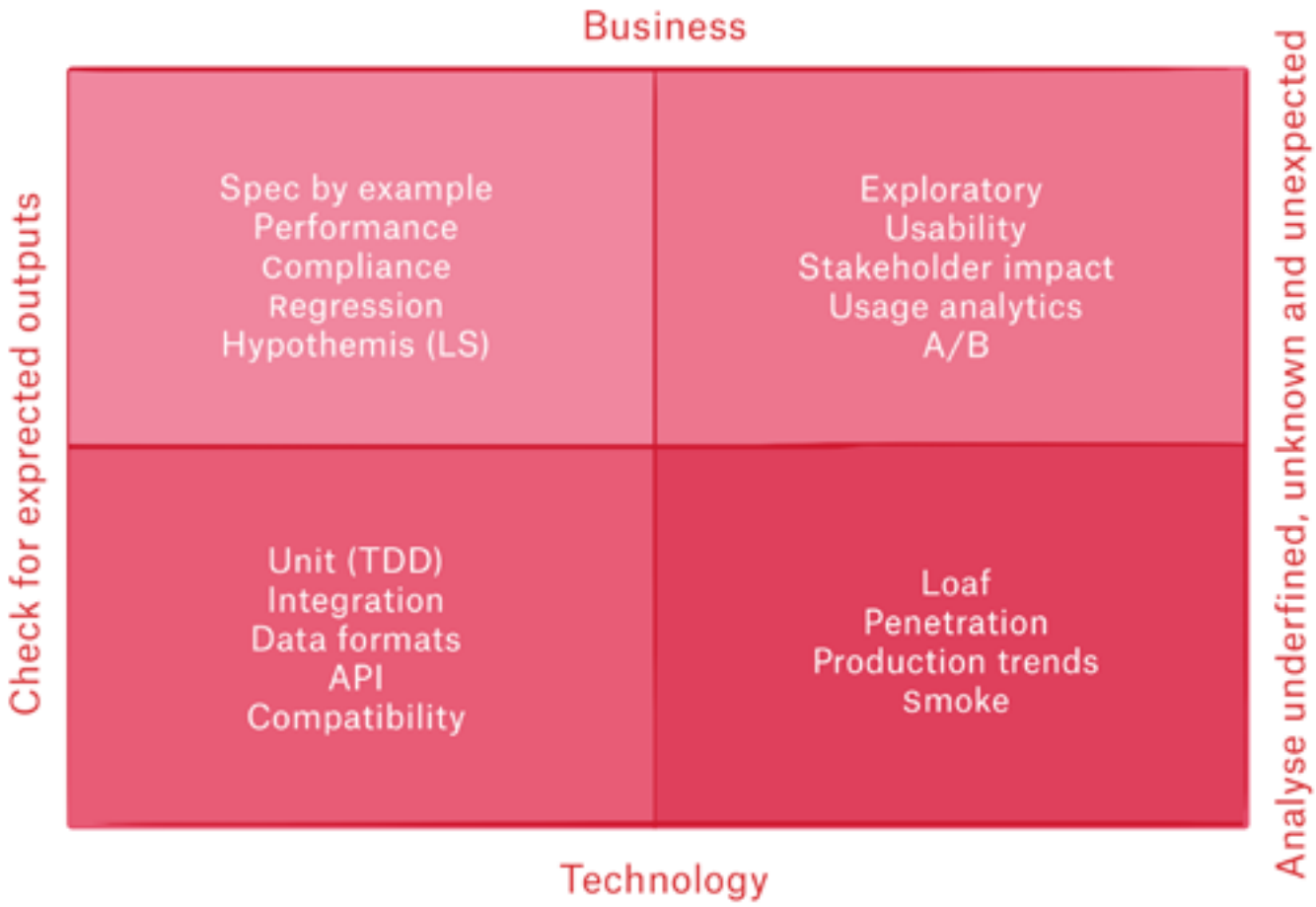
zależy nam aż tak bardzo na bezpieczeństwie, jak na dopięciu funkcjonalności. Być może ruch na początku będzie na tyle mały, że możemy o wydajności zacząć myśleć później. Jeśli nie, to starajmy się myśleć o wszystkich aspektach niefunkcjonalnych od początku projektu. Całe szczęście technologia częściowo nas od tego odciąża, zapewniając pewną dozę wydajności, czy też bezpieczeństwa out-of-the-box, tak więc to na ile zaangażujemy się w tego rodzaju testy będzie pewnie zależeć od naszych potrzeb. Jak to zwykle bywa, wszystko zależy od kontekstu.

Wpis #20 Podsumowanie kwadrantów

W ostatnich 5 feedach opisałem czym są kwadranty testowania i jak można je rozumieć. Tak jak już wspomniałem na samym początku, kwadranty te to tylko narzędzie, które ma pomagać w analizie i planowaniu testowania, uwzględniając różne potrzeby zespołu oraz biznesu. Narzędzie, które ma stanowić inspirację zamiast być wyrocznią.

Tak np. Gojko Adzic opublikował na swoim blogu własną (równie ciekawą) propozycję interpretacji tych kwadrantów. Gojko uznał rozgraniczenie na testy wspierające zespół i krytykujące produkt, za zbyt problematyczne i nieaktualne twierdząc, że większość z nich zarówno wspiera zespół jak i poddaje system krytyce w tym samym czasie. Stąd jego propozycja zamienia horyzontalne fronty na testy weryfikujące oczekiwane rezultaty i testy ujawniające rzeczy nieoczekiwane. Ja sam opisując kwadranty również przemyciłem to rozgraniczenie, ponieważ uważam je za bardzo pomocne; tym samym chciałem uniknąć błędnej interpretacji jaka często pojawia się w zespołach.

Poniżej znajduje się zdjęcie z propozycją Gojko Adzic'a a sam artykuł można znaleźć tutaj.



Opisując kwadranty tak naprawdę chciałem pokazać na czym polega planowanie jakości w zespole (częściowo). Najwięcej problemów i strat wynikających z testowania pojawia się w momencie, gdy go nie rozumiemy lub gdy źle go postrzegamy. Testowanie jest odpowiedzią na potrzeby, nie na wymagania (choć czasem może być i tak). Jeśli testujemy, bo trzeba testować, wtedy postrzegamy sam proces jako koszt. Jeśli testujemy, bo tego potrzebujemy, wtedy widzimy zyski. Oczywiście musimy

w pierwszej kolejności zdać sobie sprawę z naszych potrzeb (i z czego one wynikają), i do czego to narzędzie powinno nam służyć.

Wpis #21 Być albo nie być QA, a może testerem?

Ponieważ ostatnio sporo pisałem o testowaniu, tym razem chciałbym na jakiś czas przeskoczyć na stronę QA. Zwykle rozróżnienie czym jest QA, a czym jest testowanie stanowi problem. Co gorsza wiem, że istnieją osoby, które po kilku latach pracy jako QA wciąż nie potrafią wyjaśnić różnicy. Nie chciałbym prowadzić tutaj zbyt filozoficznych rozważań, uważam jednak, że rozumienie czym jest jedno, a czym drugie, zwyczajnie pozwala być lepszym w tym zagadnieniu. Przede wszystkim pozwala precyzyjniej określać swoją ścieżkę kariery. Mówiąc o ścieżce kariery nie do końca chodzi mi tutaj o zdobycie jak największej ilości zasobów pieniężnych (choć wiadomo, po to chodzimy do pracy), a bardziej o trafiony rozwój, który przełoży się na jakość tego, co robimy jako QA, czy jako tester. Sama dyskusja na ten temat może być długa i mieć wiele perspektyw. Ja przedstawię tutaj to jak ja widzę tę różnicę i mniej więcej (bardziej więcej niż mniej), pokrywa się ona z dostępnymi źródłami, które o tym traktują.

Tak więc, Quality Assurance ma na celu tak właściwie co? Zapewnienie jakości, ma się rozumieć. Wydaje się to oczywiste. Gorzej, gdy zapytamy: Czym jest jakość i jak mam ją zapewnić? Odpowiadając na to pytanie można zobaczyć różnicę między jednym a drugim. Testowanie to jeden ze sposobów zapewniania jakości - mniej lub bardziej opłacalny i skuteczny. Jeśli rozbijemy sobie cykl życia funkcjonalności w danej iteracji to możemy przyjąć, że ma ona co najmniej trzy fazy:

- Zanim zostanie zaimplementowana;
- Jest w trakcie implementacji;
- Została zaimplementowana.

Testowanie zwykle odbywa się po implementacji. Bierzemy to, co zostało zrobione i kontrolujemy jakość testując. Co znajdziemy to zgłaszamy, bezpośrednio do autora lub, np. robimy bugi w Jirze. Kończymy, otrzepujemy ręce i idziemy na kawę (albo mamy koniec sprintu i lecimy z kolejnym zadaniem). **Tylko to nie jest QA!**

Tak jak wspomniałem wcześniej, QA to zapewnianie jakości. Jednak, jak osoba na pozycji QA ma zapewnić jakość, jeśli zwykle nie przyczynia się nawet do jednej linijki kodu produktu? To kto tę jakość zapewnia w takiej sytuacji? Autor, oczywiście. Dlaczego zatem ja jestem nazywany QA? Ponieważ nie da się umieć wszystkiego. Chociaż to autor kodu zapewnia jakość systemu, to ja jestem osobą, która ma mu w tym pomóc. Jeśli moją specjalizacją w zespole cross funkcjonalnym jest QA, to moim zadaniem jest:

- Po pierwsze określić czym jest jakość w kontekście danego projektu. Brak defektów to nie wszystko. Możemy mieć produkt dobrej jakości, który wciąż ma defekty. Defekt defektowi nie równy;
- Po drugie wiedzieć jak skutecznie zapewniać jakość, jakich narzędzi użyć, jak je wprowadzić, jak z nich korzystać;
- Po trzecie wprowadzać swoją wiedzę w życie tak aby efektem pracy zespołu był produkt o wyższej jakości;
- Po czwarte na bieżąco obserwować pracę zespołu i optymalizować sposoby na zapewnianie jakości.

To co konkretnie składa się na metody i narzędzia zapewniania jakości będę opisywał w dalszych feedach. Chciałem na sam początek podkreślić, że o ile testowanie może być pracą indywidualną, o tyle zapewnianie jakości musi być pracą zespołową.

Jeszcze jako podsumowanie, aby dobrze sobie zapamiętać (w oryginalne bo ładnie brzmi):

`*Tester focuses on tests. QA focuses on quality. And quality improvement can be achieved in multiple ways including testing.*` ~

damn, chciałem podać autora ale nie mogę sobie przypomnieć z której to książki.

Wpis #22 Istota procesu

Zaczynając ujmijmy to, "serię informacji" o tym, czym jest zapewnianie jakości i jak się to robi, pomyślałem, że na samym początku warto zacząć od wspomnienia, jaką istotę pełni proces wytwarzania oprogramowania w zapewnianiu jakości.

Proces jest pierwszą i podstawową rzeczą, jaką wprowadza się rozpoczynając projekt - bez względu na to, czy jest to przemyślany proces czy nie, jakiś zawsze istnieje. Skoro proces zawsze istnieje, to nasuwa się pytanie, czy jest jakaś różnica, gdybyśmy spróbowali dotrzeć do celu drogą A albo drogą B. Różnica istnieje, i w zależności od procesu, może być większa lub mniejsza. Aby zdać sobie z tego sprawę za przykład posłuży mi banan. A w zasadzie proces jego krojenia.

Potrzebuje abyś teraz włączył/włączyła swoją wyobraźnie. Jest rano, chcesz zrobić owsiankę z bananem. Zaczynasz więc kroisz banana prostopadle do jego osi, w plastry. Ponieważ lubisz małe kawałki, chcesz teraz wszystkie te plastry przeciąć jeszcze na pół. Możesz to robić po kolei albo próbować zebrać wszystkie plastry, które już zdążyły się rozsypać i przekroić je razem. Następnie zbierasz wszystko i wsypujesz do miseczki. Spróbuj następnym razem **najpierw** przekroić banana na pół, a potem obie te części złączyć i dopiero zacznij kroić w plastry. Zobacz ile czasu mniej zeszło Ci na krojeniu. Zauważysz, że dokładnie ten sam cel jesteś w stanie osiągnąć szybciej, jeśli zrobisz to sprytniej. (Swoją drogą, ciekawe czy są jeszcze inne, może sprytniejsze sposoby na krojenie banana?)

Proces w jakim pracujemy może wpłynąć zarówno na czas, jak i na inne atrybuty w rezultatach, np. na jakość. Niestety, sama praca z procesem jest bardzo trudna z kilku powodów. Po pierwsze, układając proces nie zawsze jesteśmy w stanie przewidzieć, jak będzie działał we wszystkich jego aspektach. Po drugie, czasem ciężko jest zauważyć, że mamy problemy z wydajnością. Czasem widzimy, że je mamy, jednak nie do końca wiemy czemu. Dlatego praca z procesem to nieustająca obserwacja i optymalizacja. Grunt to mieć dobre założenia.

Założeniem QA- co do procesu- powinno być podnoszenie jakości. Tak więc naszym zadaniem jest obserwować jak pracujemy i identyfikować elementy, które nie pomagają nam zwiększać jakości, a czasem nawet ją obniżają. Spróbujmy kierować się poniższymi przesłankami:

- Proces ma być łatwy i przejrzysty, w miarę krótki. Im dłuższa i bardziej zawiła jest droga, tym większa szansa, że popełnimy błąd; celem jest unikanie złożoności;
- Proces ma umożliwiać współpracę i komunikację. Im lepszy przepływ informacji, tym większa wiedza, a możliwość, że coś pominiemy maleje;
- Proces ma dawać szybką i częstą informacje zwrotną. Dzięki temu możemy reagować na problemy szybko, co w niektórych przypadkach przekłada się na zyski czasowe, które możemy reinwestować, lub też na unikanie dalszych problemów;
- Proces czasem powinien narzucać pewne praktyki. Brzmi troszkę kontrowersyjnie (ze względu na słowo narzucać- tak myślę), jednak nie zawsze wolność i swoboda to prędkość i jakość... Dla przykładu podam dwa elementy: rozpisywanie wymagań i code review. Jeśli założymy (narzucimy), że każde zadanie ma mieć opisane wymagania, jest duża szansa, że przemyślimy sobie implementacje wcześniej, niż w jej trakcie. Dalej, jeśli robimy code review, jesteśmy w stanie mniej lub bardziej zidentyfikować jakieś problemy. Nie robiąc code review wcale, mamy na to nikłe szanse. Tak więc “narzucone” rzeczy niekoniecznie muszą być złe;
- Proces powinien wspierać wczesną integrację i minimalizować czas trwania pracy nad jedną rzeczą (np. zadaniem). Tutaj znów wczesna integracja to szybka reakcja na błędy, mniejszy koszt naprawy, unikanie rework’u, context swtichingu itd.

Łatwo mówić, trudniej zrobić. Niestety, są to jakieś poniekąd dobre praktyki, które w teorii przyczyniają się do poprawy jakości, jednak nie zawsze wszystko wygląda prosto. To, jak dany element procesu będzie się sprawdzał, w dużej mierze zależy od zespołu, projektu, aktualnego procesu i pewnie wielu wielu

innych czynników. Niemniej, wciąż możemy starać się proces optymalizować pod warunkiem, że regularnie będziemy doszukiwać się takich możliwości. Pamiętajmy, że zmiany nie zawsze są widoczne od razu. Ponadto, wielkie rewolucje też nikomu nie służą, czasem lepsze są małe kroczki. Korzystajmy z retrospektyw, które są świetnym miejscem, gdzie możemy albo adresować problemy związane z procesem, albo inicjować dyskusje, które pomogą takie problemy odnaleźć.

Wpis #23 Zarządzanie jakością

Do tej pory opisałem kilka tzw. narzędzi, z których możemy korzystać aby poprawiać jakość produktu. W mojej opinii są to najistotniejsze rzeczy, stanowiące fundament naszej pracy. Mają one na tyle szeroki zakres, że stosunkowo szybko będziemy w stanie uzyskać pozytywne rezultaty.

Trzeba jednak pamiętać, że zmiany tego typu wymagają czasu, pracy i zaangażowania. Jeśli zdecydujemy się wprowadzić coś do naszego życia projektowego, to jesteśmy dopiero na początku drogi pracy z danym narzędziem. Niestety, często na tym etapie kończymy czekając na to, aż wszystko nabierze tempa samo. Potem można spotkać się z czyimś niezadowoleniem, że w zespole mamy Definition of Done, jednak nic to nie zmienia.

Jaki zwykle jest z tego wniosek?

Początki często dają słabe wyniki lub nawet zerowe. Wszystko zależy od tego, jak dobrze nauczymy się już na samym początku korzystać z nowości. Dalej musimy monitorować, jakie rezultaty te nowości przynoszą i w których miejscach wymagają optymalizacji, np. optymalizacji czasowej lub powiększenia wiedzy.

Tym samym dochodzę tutaj do zarządzania jakością. Korzystam z następującej definicji:

“Quality management is the act of overseeing all activities and tasks needed to maintain a desired level of excellence. Quality ma-

nagement includes the determination of a quality policy, creating and implementing quality planning and assurance, and quality control and quality improvement."

Tylko, skąd bierze się ten "desired level of excellence"? Czy bierzemy, potocznie mówiąc, "klocek" o nazwie jakość i nad nim pracujemy, poprawiając go, aby potem móc się nim pochwalić? Otóż nie.

Niestety, zwykle nie rozumiemy zapewniania jakości lub dorównujemy zapewnianie jakości z testowaniem, albo- co gorsza- z weryfikacją. Dlatego spora część wpisów jest właśnie o takich różnicach. Nie jesteśmy w stanie realnie nadzorować jakości, ponieważ jakość nie jest czymś namacalnym, co możemy po prostu wziąć i ulepszyć. Jakość to też nie jest brak defektów, więc samo testowanie i "naprawianie" nie jest rozwiązaniem.

Jakość to produkt uboczny. Jest wynikiem tego co i jak robimy. To stopień, w jakim produkt spełnia nasze oczekiwania. Nasze tzn. różnych interesariuszy. Interesariuszem zwykle jest klient, a może użytkownik? Może programista. Może firma sprzedająca oprogramowanie? Tak naprawdę, każdy kto w jakimś stopniu ma do czynienia z oprogramowaniem, czegoś od niego oczekuje; częściowo te oczekiwania są rozbieżne, częściowo się pokrywają.

Same oczekiwania mogą dotyczyć różnych aspektów systemu- funkcjonalnych lub też nie- i wywodzić się z różnych źródeł, np. przemyślana analiza potrzeb, wcześniejsze doświadczenie, ukryta potrzeba, z której nie zdajemy sobie sprawy. Dlatego jeśli chcemy zapewniać jakość, warto zdawać sobie sprawę, o co tak w ogóle nam chodzi. Jakby to zapytał mój dobry znajomy: "Co Ty tak w ogóle chcesz przetestować?". Cóż... Jeśli nie umiemy odpowiedzieć na tego typu pytanie, jest szansa, że poprawimy/przetestujemy "coś tam", czego wynikiem też może być "coś tam", a potem okaże się, że my i nasz problem znajdujemy się na dwóch różnych krańcach miasta.

Wracając do zarządzania jakością, definiujemy jakość, obserwujemy jak rzeczywistość ma się do naszej definicji, wprowadzamy narzędzia i metody, które pomagają nam tę jakość zapewnić, identyfikujemy problemy i pracujemy nad nimi. Dopiero poprzez takie czynności, efektem ubocznym pisania oprogramowania

okazuje się, spełnienie naszych jakościowych oczekiwań.

Wpis #24 Czy jakość to koszt?

Lubię zapisywać ciekawe cytaty. Ciekawe nie znaczy zawsze właściwe, czasem po prostu warte przemyślenia. Ten akurat uważam za ciekawy i właściwy. Dzisiaj od takiego zacznę:

"Testing sometimes seems to slow down development. But speed is not the only factor we should aim to improve. It's the same as if a bus wanted to improve its speed by giving up taking passengers. It would improve its speed but it would also miss it's point."

Moim zdaniem powyższy cytat świetnie prezentuje to, dlaczego potrzebujemy QA'jów bardziej, niż testerów, oraz dlaczego QA'je muszą wiedzieć, na czym polega ich rola.

Zastanówmy się, czy jakość to koszt. Ja powiem, że tak. To jest koszt, mimo to często zauważam, jak inni testerzy próbują ukryć tę niewygodną prawdę, kręcą się w kółko albo opowiadają o rzekomej "darmowej" jakości. Rozliczamy się wszyscy na podstawie czasu i każda minuta poświęcona na aktywność zmierzającą ku podnoszeniu jakości to minuta, która kosztuje, więc jest kosztem. Moim zdaniem, co najwyżej możemy te koszty minimalizować lub optymalizować ROI z poświęconego czasu. Jest też druga skrajność, gdzie zapewnianie jakości widzi się tylko jako koszt. Ci "niewygodni" testerzy spowalniają nam development, trzeba im zapłacić, natomiast klienta przekonać, żeby można było zakontraktować ich czas.

Kiedyś na konferencji, po prelekcji jeden z uczestników zadał pytanie, jak przekonać biznes, że zapewnianie jakości to nie tylko koszt, bo u siebie w organizacji miał z tym ogromny problem. Prelegent natomiast zapytał, czy biznes wie, po co płaci, czy wziął ich do pracy tak po prostu, ponieważ tak się robi.

Trafił w sedno. Jeśli chcemy zapewniać jakość dla klienta, to przede wszystkim musimy wiedzieć, na czym klientowi zależy

najbardziej, a na co jest w stanie przymknąć oko. Komercyjne projekty zwykle (tak myślę), obarczone są kompromisami; gdzieś trzeba uciąć czas, gdzieś nadgonić, bo klient nie chce zapłacić więcej. Jak już cokolwiek przyciąć, warto wiedzieć gdzie. Od tego też jest zarządzanie jakością. Tak jak w przypadku autobusu, jeśli zdecydujemy się optymalizować pozbywając się pasażerów, to po co nam autobusy w pierwszej kolejności. Tak samo w przypadku oprogramowania, jeśli decydujemy się przycinać czas na zapewnianie jakości to wtedy pierwszą rzeczą jaką robimy jest ustanowienie priorytetów, aby nie skończyć jak pusty autobus (tzn. bez klienta lub bez użytkowników). Jeśli nie jesteśmy w stanie zająć się wszystkimi problemami, zajmujemy się w pierwszej kolejności tymi, które są kluczowe dla biznesu, a pozostałe mogą zejść na drugi plan. Być może zdążymy jeszcze się nimi zająć, prawda?

Wpis #25 Priorytet testera

Jakiś czas temu wspominałem, że testowanie gruntowne jest niemożliwe (istnieje taka zasada). Wspominałem też, z czego to wynika: zarówno ze złożoności systemu jak i ograniczonych zasobów (najczęściej czasowych). Choć o zasoby warto walczyć, nie zawsze jest to proste i nie zawsze jest skuteczne.

W takiej sytuacji, zamiast się martwić, naszym zadaniem powinno być priorytetyzowanie. Na czym te priorytety polegają:

- W przypadku poprawek czy refaktoru, w pierwszej kolejności testujemy obszar bezpośredniej ingerencji. Dopiero potem robimy testy regresji całego systemu (chyba, że mamy dobrą automatyzacje, wtedy system sprawdza się). Ryzyko ewentualnych problemów jest większe dla obszarów modyfikowanych, niż tych zależnych.
- Znając system z którym pracujemy, powinniśmy wiedzieć, jakie funkcjonalności są kluczowe dla klienta, a jakie są “nice-to-have”. Najpierw testujemy te kluczowe, dopiero

potem pomocnicze. W przypadku problemów, klient czy użytkownik przymknie oko, jeśli nie skorzysta z jakiegoś ułatwienia, ale będzie niezadowolony, gdy nasz system przestanie mu przynosić konkretną wartość biznesową.

- System przede wszystkim powinien zapewniać funkcjonalność, dopiero w drugiej kolejności powinien być wydajny. Wiadomo, najlepiej mieć jedno i drugie, ale testując, najpierw upewniamy się, czy pierwszy warunek został spełniony. Lepszy jest system funkcjonalny, jednak mało wydajny, niż wydajny system, który spełnia funkcje nie do końca.
- Testowanie funkcjonalności zaczynamy od głównych ścieżek, z których najprawdopodobniej użytkownicy będą korzystać. Dopiero potem przechodzimy do mniej prawdopodobnych sytuacji. Jeśli dana funkcjonalność jest wykorzystywana w sposób podstawowy przez np. 80% użytkowników, to upewniamy się, czy 80% będzie mogło czerpać wartość z systemu. Dopiero w dalszej kolejności martwimy się tymi pozostałymi 20%-tami. Ponadto, podstawowe ścieżki to świetny fundament dla testowania eksploracyjnego.
- Defekt defektowi nie równy, funkcjonalność funkcjonalności też nie. Jeśli mamy kilka takowych i wszystkie przynoszą wartość biznesową, to zaczynamy testować (lub w jakikolwiek sposób priorytetyzujemy) te, które w przypadku wystąpienia problemu stworzą najwięcej szkód.

Powyższe punkty to rodzaj heurystyki nad którymi warto się zastanowić. Np. w przypadku punktu nr 3, czasem brak wydajności całkiem załamuje sens funkcjonalności. Takie same odchylenia będzie można znaleźć w pozostałych przypadkach. Sens w tym, aby świadomie podchodzić do priorytetyzowania i robić to na podstawie sensownych założeń.

Wpis #26 Czy każdy defekt wymaga naprawy

To pytanie rozpoczyna temat zarządzania defektami w projekcie. Jesteśmy w trakcie sprintu, testując lub developując, znajdujemy jakiś problem. Zastanawiamy się, czy to już bug, czy jeszcze nie, no i co z tym zrobić, pozostawić czy zgłaszać. Idealnie byłoby naprawiać wszystkie możliwe nieścisłości, przede wszystkim te bardziej, ale też te mniej istotne. Skoro coś powoduje, że zapala nam się czerwona lub żółta lampka, to często oznacza, że mamy do czynienia z czymś, co jest sprzeczne z naszymi oczekiwaniami. Co prawda nasze oczekiwania mogą być błędne ale to osobny temat.

Zgłaszając defekt oczekujemy, że prędzej czy później się nim zajmiemy. Zwykle oznacza to dwa etapy. Po pierwsze, trzeba się zastanowić czy chcemy go naprawiać. Po drugie, trzeba przejść do działań, czyli albo go odrzucić, albo naprawić. Pozostawianie defektów to tak naprawdę decyzja, która mówi nam, że prędzej czy później i tak ten defekt odrzucimy, więc nie ma sensu zaprzątać sobie nim głowy. Co do defektów wartych naprawiania, możemy się zastanowić nad kilkoma rzeczami:

- Ile czasu zajęłoby klientowi dojście do tego problemu (złożoność)?
- Jak często ten problem może występować (powtarzalność)?
- Jak wielkie straty nas czekają w przypadku wystąpienia tego błędu (krytyczność)?

Do tego możemy pomóc sobie pytaniem kolejnym:

- Ile czasu zajmie mi naprawa defektu?

Błędy poważne, czyli najczęściej brak spójności z kryteriami akceptacji, najlepiej poprawiać od razu, a pomoże nam w tym wczesne testowanie.

Dyskusje zaczynają się przy błędach, które są poważne, ale których nie znajdziemy w kryteriach akceptacji. Nasza funkcjonalność działa, jednak tylko w danych warunkach. Lub w danych warunkach przestaje działać (np. przy sporym obciążeniu). Musimy odpowiedzieć sobie na pytania powyżej, albo zająć się defektem od razu, zgłosić go do Jiry.

Są defekty, które blokują naszą pracę, ale teoretycznie nie zakłócają funkcjonalności. W takiej sytuacji, najszybszym rozwiązaniem jest spróbować obejścia (tzw. workaraound'u). Jednak prędzej czy później te błędy naprawiamy, ponieważ nie powinniśmy ich zostawiać. Wtedy można zadać sobie pytanie: Czy moje workaroundy nie dodały mi przypadkiem czasu do pracy nad danym zadaniem? Lub: Skoro i tak prędzej czy później problem trzeba było naprawić, to może lepiej było to zrobić już na samym początku?

Są jeszcze defekty gorsze, takie które utrudniają naszą pracę. Bardzo zdradliwe. Odraczając naprawę takiego błędu myślimy o tym, że najważniejsze jest "dowieźć" aktualny sprint. Jednak bardzo rzadko myślimy o tym, ile czasu taki defekt odbiera nam z obecnego i każdego kolejnego sprintu. Dużo łatwiej jest dostrzec nam sukces, gdy kończymy sprint, niż przepalone przez defekt małe fragmenty czasu, które rozciągnięte na kilka tygodni, pomnożone przez każdego członka zespołu, mogą dać całkiem niezłą sumę. Takie defekty są jak próchnica. Idziemy do dentysty jak już bardzo boli, ale wtedy okazuje się, że przespaliśmy moment i trzeba wyrwać zęba albo zapłacić sporą kwotę za leczenie kanałowe. Nie widzę w tym problemu, jeśli tylko świadomie podejmujemy taką decyzję i jesteśmy gotowi zapłacić za nią cenę.

Idąc dalej, są defekty, tzw. trywialne, np. wpływające na postrzeganie jakości, czy też na wygodę z użytkowania. Jeśli naprawa takich defektów zajmie nam minimalną ilość czasu, można się tym zająć choćby dla własnej satysfakcji. Dużo lepiej pracuje się nam z systemem, który nie ma tzw. wystających nitek. Niby nic, ale irytuje.

Ujmując rzecz książkowo, każdy defekt jest zły i wymaga naprawy. W rzeczywistości, prócz defektów jest jeszcze wiele innych aspektów, które sprawiają, że projekt dociera do końca i klient jest zadowolony. Decydując, czy dany defekt jest wart naszej

uwagi, musimy pamiętać, że zysk z naprawy nie może przewyższyć kosztu jakim jest czas.

Gojko Adzić często przypomina, że w momencie, gdy klienci dostaną to, czego chcą, tzn. realizowaną wartość biznesową, i dostaną to w dodatku szybko, wtedy nawet nie zauważają pomniejszych defektów, lub są w stanie przymknąć na nie oko.

Zastanawiając się nad tym, postarajmy się mieć na uwadze nie to, czym jest defekt, np. serwis rzuca tam jakimś błędem (naprawiamy czy nie), tylko to, jaki efekt ma on na realizację wartości biznesowej lub na pracę projektową (serwis rzuca błędem, przez co przestaje realizować wartość biznesową dopóki nie zostanie zrestartowany, co może już być warte naszej uwagi).

Wpis #27 Adwokat diabła ... ekhm ... buga

Znane również jako bug advocacy. Gdy tester, QA, czy ktokolwiek z zespołu znajduje problem w systemie, zwykle najpierw ocenia, czy faktycznie jest to problem. Następnie komunikuje ten problem innym i ewentualnie robi task w Jirze, żeby się nim zająć.

Jednak, sam komunikat o tym, że mamy defekt w systemie może nie być na tyle wymowny, aby zmotywować zespół do naprawienia go, bądź chociaż zpriorytetyzowania zadania w Jirze, żeby ktoś wkrótce się nim zajął. Tak jak pisałem w innym feedzie, nie wszystko jest warte naprawy, ale jeśli mamy defekt, który ewidentnie tego potrzebuje, jednak nikt nie zdaje sobie z tego sprawy, wtedy potrzeba kogoś, kto uświadomi wszystkim sedno znalezionego owego defektu.

Tym jest właśnie bug advocacy. Nie jest to jeszcze perswazja, ale coś w rodzaju rzecznictwa. Nie zawsze defekt będzie mógł "wypowiedzieć się" sam za siebie. Jeśli tak jest, np. defekt powoduje całkowite załamanie funkcjonalności już na środowisku developerskim, wtedy nie potrzeba żadnego rzecznika. Defekt uderza w większość zespołu. Jednak jeśli defekt pokazuje się tylko w

określonych okolicznościach, lub gdy użytkownik jest postawiony w odpowiednim kontekście, wówczas potrzeba osoby, która pomoże zespołowi zdać sobie z tego sprawę. Taki "adwokat" buga, musi przede wszystkim:

- Poinformować zespół o tym na czym polega defekt;
- W jakich okolicznościach konkretnie może się on pojawić;
- Jak często może się pojawiać;
- W jaki sposób wpłynie na możliwość korzystania z systemu przez użytkownika oraz;
- Jak wpłynie na realizowanie wartości biznesowej.

Aby być skutecznym adwokatem, potrzeba umiejętności miękkich, myślenia systemowego, trochę retoryki i skutecznego przekazywania informacji.

Brzmi to trochę, jakby nikogo nie interesowały defekty. W zasadzie, pracując w projekcie często ma się takie wrażenie. Jednak jest to tylko wrażenie (taką mam nadzieję). W momencie gdy pojawiają się defekty, tester w pierwszej kolejności myśli o naprawie, podczas gdy pozostali członkowie zespołu myślą o dowiezieniu obecnego sprintu, oraz obecnej pracy, z którą powinni zdążyć na czas. Również o tym, że mamy ograniczone zasoby czasowe itd., itd.

Zanim dojdziemy do tego, jaką istotę pełni defekt i czy w ogóle jest wart naprawy, musimy przedrzeć się przez te wszystkie "ściany". Warto zapamiętać, że rolą testera nie jest przekonywanie zespołu do naprawiania bugów. Rolą testera jest pomóc przebić się zespołowi przez wszystkie ściany, aby dotrzeć do momentu, w którym zespół realnie spojrzy na defekt i świadomie- wyposażony we wszystkie istotne informacje o defekcie, które tester przekaże- oceni, czy warto się nim zajmować.

Wpis #28 Co robić z drobnymi błędami?

Mówiąc (lub pisząc) “drobne błędy”, mam na myśli błędy typu literówki, problemy z responsywnością, brak spójności w widokach czy komunikatach, brzydkie formatowanie wiadomości, niejasne komunikaty, niejasne elementy na stronie, których nie zauważyliśmy, lub takie elementy, które wprowadzają w błąd, ponieważ zapomnieliśmy dostosować je do konkretnego kontekstu itd.

Na tego rodzaju błędy zwykle nie zwracamy uwagi, myśląc o ważniejszych sprawach. Czymże jest literówka w porównaniu do wartości, jaką przynosi gotowy tracker! Niektóre tego typu błędy to kwestia kilku minut poprawek, nawet w tym samym otwartym PR’rze, a niektóre wymagają troszkę więcej zmian.

W naszych oczach, drobne problemy są... drobne, bo patrzymy ze swojej perspektywy i porównujemy je do pozostałej pracy jaką wykonaliśmy. W oczach użytkownika, który ani nie ma pojęcia, co dzieje się pod spodem, ani go to nie interesuje, drobne problemy interpretowane są jako niedogodności, niedociągnięcia, brak profesjonalizmu, wątpliwe działanie systemu etc. Niestety, spore grono użytkowników ocenia nasz system nie na podstawie tego, co realizuje, a na podstawie tego, jak wygląda i działa jego interfejs. Choć zespół wie, że system jest rzetelny, użytkownik niestety nie.

Literówki czy “brzydkie” komunikaty nie sprawiają, że użytkownik nie może uzyskać wartości z działania systemu, jednak mogą sprawić, że użytkownik zdecyduje się przejść na inną platformę, która wygląda bardziej profesjonalnie. Skala problemu zależy od kontekstu i domeny.

Wiadomo, jak to bywa... Projekt ma terminy, określony budżet i całą listę stories w backlogu. Nie ma czasu zajmować się drobnostkami Jednak, z punktu widzenia jakości, satysfakcja klienta (oraz jego użytkowników) to częściowy wyznacznik jakości, (tutaj zachęcam, aby nasza motywacja do wprowadzania tych

mniejszych poprawek nie była kierowana perspektywą zespołu developerskiego a perspektywą klienta i użytkowników). Jeśli małe poprawki to kwestia paru minut czy godziny, to może warto je poświęcić dla uzyskania satysfakcji klienta. Jeśli takie poprawki zajmują więcej czasu, to wciąż warto chociaż spojrzeć z innej perspektywy, zanim zdecydujemy się, czy coś pozostawić. Łatwiej jest odrzucić poprawkę, ponieważ lista w backlogu jest naszą teraźniejszością. Problemy jakościowe tego typu są problemami odroczonymi, o których mówi archetyp myślenia systemowego Balancing Process with Delay.

Z jednej strony zawsze staram się okazać zrozumienie dla realiów projektowych, a z drugiej- w końcu jestem od tego, żeby promować jakość, i niestety jakość sama się nie zapewni. Dlatego jedyne co mogę zrobić, to upewnić się, że gdy rezygnujemy z poprawek, robimy to mając na uwadze klienta i użytkowników.

Wpis #29 Czym jest a czym nie jest risk-based testing

Risk-based testing, czyli testowanie w oparciu o analizę ryzyka. Z jednej strony, sama nazwa jest na tyle wymowna, że można od razu załapać, o co chodzi: analizujemy prawdopodobieństwo, co może pójść nie tak najprędzej i jakie będą tego konsekwencje. Dalej, właśnie te rzeczy testujemy. Proste.

Jest jednak i druga strona medalu. Rozważmy następujące pytanie: *Po co mi risk-based testing, jeśli zaraz przetestuje wszystkie ryzykowne rzeczy, a dalej przystępuje do testowania user stories lub do wykonywania testów akceptacyjnych?* Być może nie wiedziałem, ile będę mieć czasu na testowanie lub jak dużo czasu mi to zajmie. Dlatego najpierw wykonałem risk-based testing, aby otestować najbardziej kluczowe rzeczy; potem okazało się, że czasu mam jednak więcej. Nie jest to złe podejście do tego rodzaju testowania. Mimo wszystko, wciąż nie jest to jego kwin-

tesencja.

Wspomnianą kwintesencje można zobrazować za pomocą pączka (kradnę metaforę zapożyczona). Jeśli damy dziecku pączka i powiemy, że będzie mogło go ugryźć tylko z jednej strony, to pytanie, którą stronę wybierze? Tutaj wybór dziecka może zależeć od jego upodobań, np. woli dżem więc ugryzie ten kawałek, gdzie jest go więcej. Może po kilku takich razach ma już dość dżemu i chciałby zjeść troszkę więcej ciasta? A może wybierze lukier, bo go kocha?

Risk-based testing nie traktuje o tym, co będziemy testować, a czego nie będziemy testować. Analizę ryzyka robimy w celu zorientowania się, co możemy poświęcić, aby dostarczyć coś szybko i stosunkowo dobrej jakości. Czasem skalowalność testowania manualnego i automatycznego w górę dochodzi do punktu, gdzie nie możemy już dalej korzystać z członków zespołu, czy też równoległych buildów i musimy przyjąć inną strategię. Musimy zacząć rezygnować z pewnych obszarów, a żeby to zrobić potrzebujemy analizy ryzyka.

Dokładnie takie podejście zaczęło w pewnym momencie stosować Google wdrażając tzw. Google-Risk Assessment Tools, gdzie członkowie organizacji lub poszczególnych zespołów wspólnie mapowali ryzyko systemu i podejmowali decyzje dotyczące tego, co będzie testowanie, a co nie. Oczywiście to nie jest tak, że “machnęli ręką” na to, czego nie będą testować... Rozwiązaniem na te niepokryte obszary było przerzucenie kontroli jakości na system monitoringu, metryk i alertów oraz odpowiednią strategię wprowadzania fixów i ewentualnych rollbacków nowych wersji, które wdrażali.

Możemy sobie z tego zdawać sprawę lub nie, dostarczanie szybciej i lepiej to nasza nieustanna mantra i w ten sposób analiza ryzyka to nasza codzienność. Robimy to bardziej lub mniej świadomie więc osoboście zachęcam do bardziej świadomego podejścia. Bo jeśli już mamy coś pozostawić, lepiej robić to minimalizując problemy naszych przyszłych nas.

Wpis #30 Kalibracja strategii testowania

Podejmując rolę testera naszym zadaniem jest, krótko mówiąc, testować. Pytanie tylko, czy testowanie zawsze wygląda tak samo? Otóż nie.

Z biegiem czasu poznajemy nowe narzędzia, nowe techniki testowania, nowe miejsca, w których pojawiają się bugi, i do których zaczynamy zaglądać. Co za tym idzie, stajemy się bardziej doświadczeni i teoretycznie - lepsi w tym, co robimy.

Jednak to nie wszystko, co powinno dziać się wraz z upływem czasu. Każdy bug jaki do nas (do zespołu) trafia, to informacja o tym jak pracujemy. Bardziej; że nasza praca nie jest doskonała. Bugi, które znajdujemy zanim trafią na produkcję, dają nam informacje o skuteczności naszej strategii testowania. Jeśli je znaleźliśmy, wszystko, co doprowadziło nas do nich, jest warte kontynuacji w kontekście testowania. Jeśli chodzi o development- tu możemy zastanowić się, czy da się zrobić coś w celu uniknięcia podobnych bugów w przyszłości.

Bugi, które znajdują klienci dają nam informacje, że zawalił pracę zarówno development, jak i testowanie. To nie znaczy, że od razu się załamujemy... Zawsze może zdarzyć się sytuacja, że coś przeoczymy. Jednak, jeśli już to zrobiliśmy, możemy się zastanowić nad tym, czy można coś zmienić w developmencie i testowaniu, aby ponownie unikać takich bugów.

Bugi, których nikt nie znalazł (bo być może ich nie ma), informują nas o skuteczności naszego testowania. Jeśli podejmujemy się weryfikacji i wszystko działa zgodnie z wymaganiami to w porządku- to jest cenna informacja, a założenia zostały zrealizowane.

Natomiast jeśli zaczynamy testować eksploracyjnie i spędzamy czas nie znajdując żadnych problemów, być może szukamy w zły sposób lub w złych miejscach. Na przetestowanie funkcjonalności mamy określoną ilość i tak już ograniczonego czasu, więc

musimy korzystać z tych technik i zaglądać do tych obszarów, które najprędzej zwrócą nam jakieś bugi. W przeciwnym wypadku, kręcimy się w kółko aż skończy nam się czas.

Tym jest właśnie kalibracja strategii testowania. Bugi są informacją dla całego zespołu. Każdy może z nich wyciągnąć informacje również dla siebie. Tester przede wszystkim powinien na bieżąco kalibrować swoją strategie testowania biorąc pod uwagę to co dzieje się w projekcie. Finalnie: Celem jest testować skutecznie możliwie najmniejszym nakładem pracy.

Wpis #31 5 spojrzeń na jakość

Kilka feedów wcześniej rozmawialiśmy o tym, jak zdefiniować jakość i o tym, że może ona przejawiać się w wielu aspektach. W innych feedach wspominam również o jakości, i można by sądzić że się powtarzam. Nie chodzi mi o to, aby w kółko poruszać jeden i ten sam temat, a bardziej- pokazać problem z różnych stron. Ponieważ zrozumienie jakości jest trudne, każde kolejne ugryzienie tematu pomaga zrozumieć go jeszcze lepiej.

Dziś chciałbym to zrobić wprowadzając 5 spojrzeń na jakość, które zaprezentował David Garvin w swoim artykule pod tytułem “What does product quality really mean.”

Napiszę je po angielsku (w oryginalne):

Product-based - jakość opisująca produkt i jego atrybuty. Jest to odpowiedź na pytanie, jak dobry jest produkt; nie kod, nie system, nie infrastruktura, tylko finalny produkt. Jak ten produkt działa, czy dobrze działa, czy jest bezpieczny, czy ma jakieś ograniczenia itd. Czyli wszystkie te cechy niefunkcjonalne.

User-based - to spojrzenie na jakość z perspektywy użytkownika. Różni użytkownicy będą mieli różne potrzeby i oczekiwania. Jakość w tej perspektywie to spełnienie tych oczekiwań, nawet jeśli użytkownik nie jest ich świadomy. Tutaj pojawia się kwestia UI/UX i dobrze zaprojektowanego produktu (nie systemu).

Manufacturing-based - jakość kontrolowana na podstawie wymagań projektowych. Klient powiedział, że mamy zrobić A i B i to właśnie weryfikujemy. Taką jakość mierzymy właśnie za pomocą weryfikacji. Sprawdzamy, czy nasz system spełnia wymagania, nasze ustalenia, oraz czy jest zgodny ze specyfikacją.

Value-based - prezentuje cenę lub koszt (zależy jak na to spojrzeć) produktu. To znaczy, że nie tylko chcemy osiągnąć jakieś zamierzone cele, ale chcemy to zrobić stosunkowo niskim nakładem pieniężnym. To też znaczy, że nie każda praktyka z tej perspektywy jakości jest słuszna, nawet jeśli daje świetne rezultaty dla innych. Aby produkt był jakościowy tutaj, nasze rozwiązania muszą również być opłacalne. Pamiętajmy, że czas to również pieniądz, więc nawet jeśli nasze rozwiązania nie są kosztowne na fakturze z AWS'a, to jeśli poświęcamy im zbyt dużo czasu, wciąż jest to wysoki koszt.

Transcendent-based - to najciekawsza perspektywa, ponieważ nie da się jej łatwo zdefiniować. Wszystko to co czujemy, że jest niewłaściwe, ale nie potrafimy skategoryzować bądź uzasadnić, podpada tutaj. Czasem pojawiają się takie rzeczy: wiesz że coś jest nie tak dopiero jak to zobaczysz.

Najważniejsza lekcja jaka płynie z artykułu Garvina to, że na jakość nie można patrzeć z jednej perspektywy. Perspektyw może być więcej, niż te powyższe. Możemy też je jeszcze inaczej skategoryzować. Jest to nieistotne. Grunt, że jakość = testowanie, i że jakość = brak błędów.

Czasami to samo pytanie możemy zadać z dwóch różnych perspektyw i uzyskać tę samą odpowiedź, a czasem dany aspekt jakości będzie właściwy tylko dla jednej perspektywy. Widać tutaj też to, co już powtarzałem, że jakości nie jest w stanie zapewnić tester czy QA. Tester może przyczynić się do zwiększenia jakości Manufacturing-based - wykonując weryfikacje i licząc na to, że odnalezione błędy zostaną poprawione. Cała reszta zależy od wspólnej pracy całego zespołu.

Czy tego chcemy czy nie, każdy z nas, każdego dnia, gdy rozpoczyna pracę, decyduje o tym, jakiej jakości będzie nasz produkt. Przekazując informacje, podejmując decyzje, projektując rozwiązanie, konfigurując infrastrukturę itd..., przyczyniamy się do tego,

że coś będzie działać, ale też do tego jakiej jakości będzie. Pytanie, jak często bierzemy to pod uwagę?

Wpis #32 Trzy pytania dla strategii testów

O strategii testów już wcześniej wspominałem. Osobiście, spodobało mi się jedno spojrzenie na tworzenie strategii testów, dlatego też pozwolę sobie napisać o tym ponownie.

Testujemy, ponieważ obawiamy się, że coś może pójść nie tak. Czasem wiemy od razu, co o może być dokładnie, czasem tylko się domyślamy. Strategia testów to inaczej przemyślenie sobie, co możemy zrobić oraz jak możemy testować, aby nie musieć martwić się właśnie tym, że coś się nie uda. Bardziej realnie- aby pojawiło się jak najmniej fakapów.

Zatem: co, dla kogo i ile?

Jak już wiemy, za testowanie się płaci. Testujemy więc tylko to, co faktycznie adresuje nasze obawy. Jeśli testujemy rzeczy, na których nam nie zależy- po co to robić? Jeśli obawiamy się, że coś faktycznie pójdzie nie tak, oznacza to jednocześnie, że ktoś na tym ucierpi. W przeciwnym wypadku zwyczajnie byśmy się tym nie przejmowali.. Tak więc, trzeba sobie odpowiedzieć na pytanie kto. Wtedy testujemy już tak, aby uniknąć “ofiar”.

Na koniec, to ile będziemy testować zwykle możemy zdefiniować zgodnie z wielkością naszych obaw.

W tych wszystkich aspektach, obawa jest tym, co kieruje nami przy sporządzaniu strategii testów. Obawy również nie pojawiają się znikąd. Pojawiają się, gdy zdamy sobie sprawę z konsekwencji tego co robimy- do tego, potrzeba się nad czymś zastanowić, coś przeanalizować, coś przewidzieć.

A co jeśli nie mamy obaw? Co to może oznaczać? Jak byśmy testowali, gdybyśmy się niczego nie obawiali? Zachęcam do dyskusji.

Wpis #33 Błąd krytyczny, tak ciężko ogarnąć.

Co to znaczy, że błąd jest krytyczny i po co nam klasyfikować błędy?

Pozwolę sobie zacząć od tyłu, czyli po co je klasyfikować. Przede wszystkim po to, żeby nadać im jakiś priorytet. Błąd krytyczny zwykle będzie potrzebował naprawy natychmiastowej. Jednak, finalnie to zespół decyduje, jak będzie wyglądał proces naprawiania błędów i co będzie robić z błędami krytycznymi. Być może podzieli je na takie, które wymagają naprawy od razu i na takie, gdzie możemy z tym jeszcze chwilę poczekać.

Odpowiedzią na pytanie, czym jest błąd krytyczny to klasyczne stwierdzenie “to zależy”. Błędem krytycznym będziemy opisywali defekty zgodnie z tym, co ustalimy z zespołem bądź, będziemy błędy rozpatrywać indywidualnie.

Jeśli o mnie chodzi, gdy znajduje defekty, czasem konsultuje się z innymi, czasem od razu raportuję. Jeśli widzę, że podstawowa funkcjonalność nie działa, to od razu raportuje. Pozostaje jeszcze kwestia zastanowienia się, czym jest podstawowa funkcjonalność.W mojej opinii to problem, który uniemożliwia wykonanie pełnej ścieżki e2e funkcjonalności.

Są takie błędy, które bardzo chętnie nazwałbym krytycznymi, ponieważ utrudniają lub uniemożliwiają dalsze testowanie. Niestety tak zrobić nie mogę. Nie zawsze błąd hamujący testowanie to brak poprawnej realizacji ścieżki e2e. Czasem to specyfika środowiska developerskiego. Z drugiej jednak strony, jeśli ja muszę walczyć z jakimś problemem, i z tego powodu poświęcić kilka godzin więcej na testowanie, to mogę to zrobić. Jednak tracimy wtedy czas. Szczególnie, gdy problem powraca przy kolejnym testowaniu. Czas wtedy ulatuje jak powietrze z dziurawej dętki. Niby da się jechać dalej, ale jedziemy coraz wolniej. Oszczędzamy czas na naprawę, jednocześnie marnując czas na wydłużone testowanie.

Dodam, że trzeba pamiętać, że błędem krytycznym są też błędy niefunkcjonalne. Jeśli ścieżka e2e potrafi być poprawnie zrealizowana, jednak odbywa się to okropnie wolno, jest narażona na podatności z zakresu bezpieczeństwa.

Na sam koniec wspomnę jeszcze, że błędem krytycznym są również błędy dotyczące infrastruktury. Brak możliwości wykonania poprawnego deploymentu, bądź przeprowadzenia testów regresji może być równie poważne, co niedziałająca funkcjonalność.

Ciekawy jestem, czy macie inne spojrzenie na to, czym jest błąd krytyczny?

Wpis #34 Po co planować testowanie

Z planowaniem ogółem wiąże się jeden, stały problem. Zwykle postrzegamy planowanie jako próby przewidywania przyszłości w celu ustalenia jakiś działań, które następnie chcemy realizować. Np.: planując wakacje myślimy, gdzie pojedziemy, ile to będzie kosztować, co musimy załatwić, wziąć ze sobą etc.

Czym różni się IT od naszych wakacji? Najpierw przytoczę jeden cytat: *"Zmiana jest jedyną stałą"*. Nawet planując wakacje, po pierwsze, nie przewidzimy wszystkiego, po drugie, zawsze coś ulegnie zmianie. Jeśli chodzi o IT, biorąc pod uwagę naturę projektów, tzn. ich złożoność i niską przewidywalność, planowanie na zasadzie przewidywania przyszłości jest szczególnie trudne. W tym momencie musimy zdać sobie sprawę, że planowanie to być może niekoniecznie przewidywanie przyszłości, a bardziej myślenie o przyszłości (ponieważ w IT wielu rzeczy nie da się przewidzieć, można jednak na wiele z nich się przygotować).

Planując, możemy zyskać nie tylko na samym planie, ale także na aktywności planowania. To ona daje nam okazję do tego, aby uruchomić myślenie systemowe, analityczne, krytyczne. Napędza nas do tego, aby lepiej zrozumieć domenę projektu, wymagania, naszego klienta, i oczywiście potencjalne problemy (tzn. wyzwa-

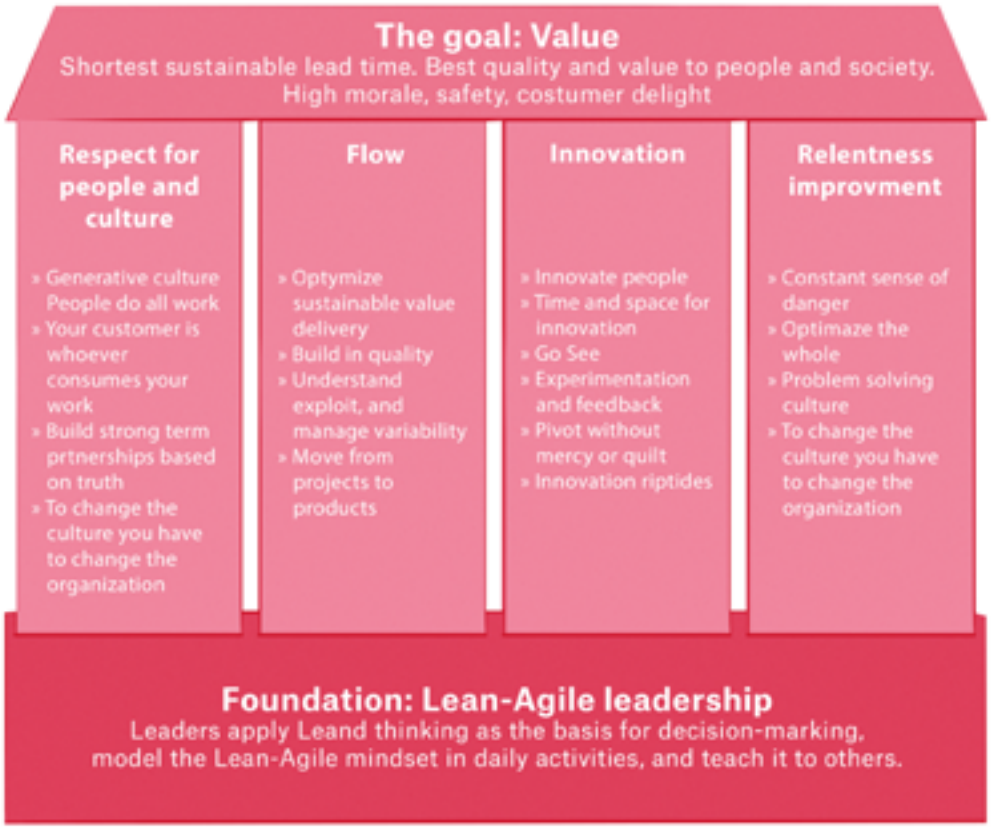
nia).

Niewątpliwą zaletą planowania będzie nie tylko ustanowiony plan (który swoją drogą może i tak się zmienić), lecz także zespół, który jest przygotowany na przyszłość, jaką przyniesie projekt. Zaczynając więc, warto nie tylko oczekiwać gotowego planu, a również angażować się w sam proces jego tworzenia. Każdy robiąc to, ma szansę poszerzyć swoją wiedzę i perspektywę.

Co do testowania samego w sobie... Planując możemy położyć fundament pod to, co będziemy robić, ale to przyszła rzeczywistość będzie tak naprawdę dyktować, co mamy robić. Mówiąc prościej: mogę sobie zaplanować wcześniej co będę robił, jednak po implementacji i tak będę musiał to skalibrować. Grunt, żebym był na to przygotowany.

Wpis #35 Lean - ściągawka

Dla wszystkich zwolenników szczupłego podejścia, prezentuję fajną formę na regularne odświeżanie sobie wszystkich istotnych wartości:



Gdyby ktoś chciał poczytać więcej na ten temat, dzielę się linkiem

Wpis #36 Strategia testowania na różnych poziomach

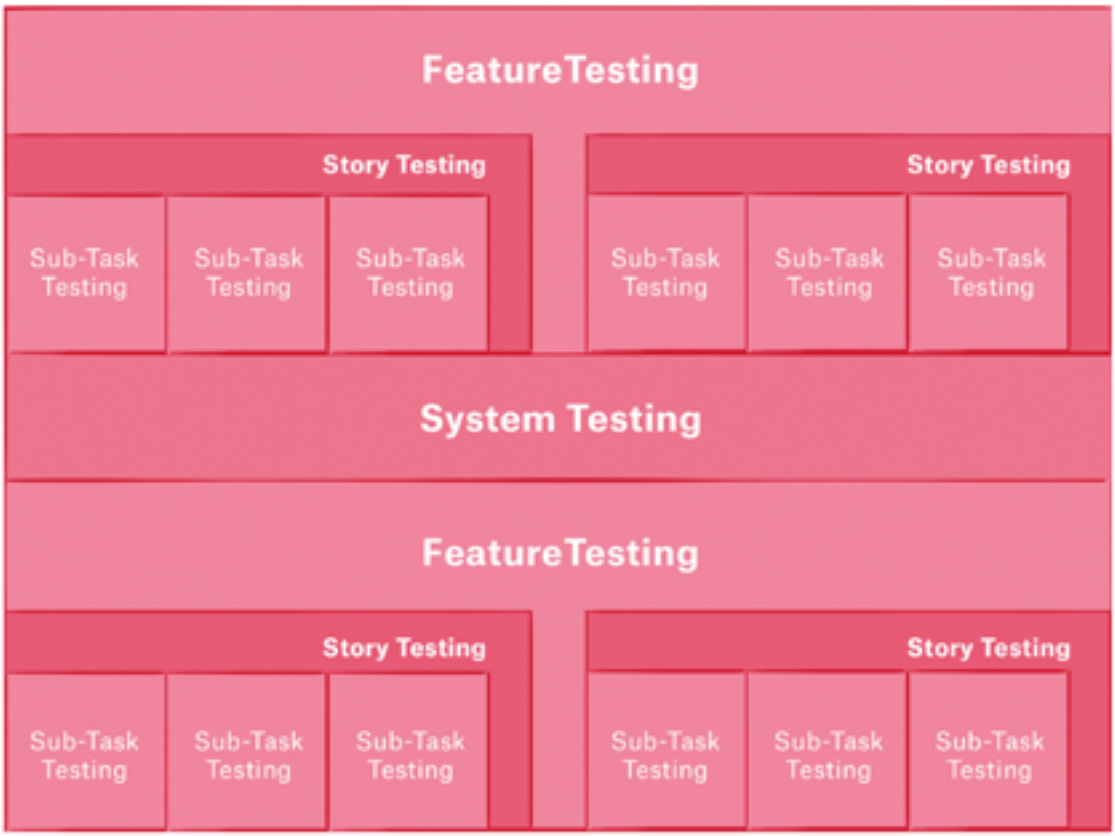
W tym feedzie chciałbym się podzielić wizualizacją, która pomoże lepiej zrozumieć zakres testowania na różnych poziomach systemu.

W moim teamie wprowadziliśmy sobie jakiś czas temu “fazę”, która nazywa się task review, co w praktyce oznacza, że w granulacji story -> sub-taski, po wykonaniu sub-taska decydujemy się, czy trzeba go testować, czy nie. Najczęściej testujemy sub-taski, ale nie zawsze widzimy taką potrzebę. Na pewno są zespoły, które testują w podobny sposób, jednak używają innego nazewnictwa, bądź ich proces troszkę się różni. Niemniej testowanie sprowadza się do tego, że dzielimy je na różne poziomy realizowania projektu.

Pamiętam początki wprowadzenia tego elementu do naszego procesu. Najwięcej problemów stwarzało zrozumienie, czym różni się testowanie w ramach task review od innego testowania, czy czasem nie dublujemy swojej pracy, czy ma to sens.

Co prawda trochę późno przychodzę z pomocą, ale wciąż, myślę że warto sobie uzmysłowić na czym ona polega.

Powyżej widzimy sposób na granulację zadań i poziomy testowania odpowiadające każdemu z etapów. W praktyce, oznacza to, że na każdym poziomie testujemy i to testowanie nazywa się inaczej. Rozumiem, że nazwa może komuś zbyt wiele nie mówić. Dlatego lepszym sposobem na przedstawienie różnic będzie kolejna wizualizacja:



Patrząc na powyższy diagram, zacznijmy od najniższego poziomu, czyli sub-taska. Developer wykonał sub-task i trzeba go przetestować. Testując, jesteśmy w stanie wykonać 100% pokrycia, ponieważ cały sub-task jest dla nas dostępny. Tak samo możemy zrobić w przypadku każdego kolejnego.

Jeśli jednak spojrzymy na to z perspektywy story, kilka tasków złączonych ze sobą to nie jest to samo, co każdy task z osobna postawiony obok. Żółty obszar pokazuje nowe pole do testowania, które nie było widoczne, gdy patrzyliśmy w kontekst każdego sub-taska z osobna.

Analogicznie, patrząc na testowanie na poziomie ficzera, mamy tę samą sytuację. Dwa story zebrane razem tworzą ficzer, który również może mieć obszar, wcześniej nie przetestowany, ponieważ jest to możliwe tylko w momencie, gdy mamy gotowe wszystkie stories z danego ficzera.

Ostatnim poziomem jest poziom systemowy i ponownie ta sama sytuacja.

Trzeba sobie uświadomić, że nasze zadania to nie zawsze puzzle (choć mogłoby się wydaważ, że tak jest). A+B nie daje AB tylko C (już kiedyś używałem tej metafory). Po dopasowaniu zadania A do zadania B, często wyłania się dodatkowa przestrzeń, która też musi zostać przetestowana. Najprędzej możemy ten obszar podzielić na:

- Przestrzeń konfiguracji - np. wiele sub-tasków musi ze sobą współgrać konfiguracyjnie. Powiedzmy, że mam na myśli poprawne interfejsy lub niezakłócony przepływ danych.
- Przestrzeń logiczna - ponownie dla przykładu użyję sub-tasków, prócz tego że kod działa poprawnie, musi też poprawnie realizować wymagania i wartości biznesowe z logicznego punktu widzenia. Ten problem jest tym większy, im bardziej rozłożona jest realizacja wartości biznesowej, na wiele serwisów lub modułów. Jeden moduł - prosta sprawa; kilka modułów (w dodatku oddalonych od siebie) - wymagania biznesowe potrzebują wszystkich powiązanych modułów, aby współgrały ze sobą, a zdarza się tak że o jakimś zapomnimy.

Pytanie, czy będziemy dublować swoją pracę? Oczywiście!Zakładam, że pierwsze, co zrobimy pracując w takim trybie, to dublowanie swojej pracy, o ile nie będziemy zastanawiać się nad tym, co robimy. Testując na wyższych poziomach, trzeba robić to tak, aby właśnie swojej pracy nie dublować. Musimy więc rozumieć, na czym polega ta różnica: między testowaniem sub-taska a testowaniem story. Musimy wiedzieć czego szukamy na poziomie story. Nie będzie miało to sensu, jeśli szukamy tego samego, co na poziomie sub-taska.

Na koniec mała uwaga. Często dobrze rozbite sub-taski sprawiają, że testując drugi i dalszy w kolejności, uzupełniamy również to żółte pole, które pojawia się z dołożeniem każdego kolejnego. To sprawia, że na poziomie story już nie musimy za wiele robić. Inna sprawa jest taka, że są sub-taski, które są na tyle niezależne, że ten żółty obszar się nie pojawia, bądź jest minimalny. Dodatkowo, warto zwrócić uwagę na to, że bardzo często testowanie ostatniego sub-taska umożliwia również testowanie na poziomie story. Reasumując, trzeba się umówić z zespołem, co dla nas oznacza testowanie sub-taska. Jeśli umówimy się na weryfikowanie tylko happy-paths, to naturalnie, że na poziomie testowania story będzie coś do zrobienia, ponieważ trzeba sprawdzić zarówno ten żółty obszar, jak i nieprzetestowane case'y z poziomu każdego sub-taska.

Wpis #37 Kiedy nie opłaca się testować

Jak i wszystkie wcześniejsze feed'y, ten również postaram się oprzeć na tym jak pracujemy z moim teamem. Konkretnie chodzi mi o granularność zadań. Standardowo dla wszystkich sprintów, do tej pory rozdzielaliśmy sobie zadania w taki sposób: epics -> stories -> sub-tasks. Planujemy i estymujemy stories, a w trakcie sprintu tworzymy sub-taski i na nich pracujemy. Każdy sub-task mergujemy do story brancha i w momencie, gdy wszystkie sub--taski są już gotowe, story brancha mergujemy do mastera. Tak w skrócie.

Teraz kwestia testowania. Z punktu widzenia jakości, najlepiej byłoby testować zawsze, wszystko. Tzn. w danym sprincie można by testować każdy sub-task, a po zakończeniu wszystkich sub--tasków z danego story, można jeszcze przetestować wersję ze story brancha. Zdaje sobie sprawę, że brzmi to jak dublowanie swojej pracy, ale o tym, że niekoniecznie tak jest, pisałem w feedzie #35 Strategia testowania na różnych poziomach.

Wróćmy do perfekcji; testujemy zawsze wszystko. Gdybyśmy mieli nieograniczony czas w sprincie i w dodatku nikt na nic by nie czekał, pewnie dałoby się zrobić w ten sposób. Wiadomo, czas jest ograniczony, nieraz bardzo mocno. Prawie zawsze ktoś czeka na tego taska, którego właśnie się testuje. W takim razie, lepiej wziąć się za strategię i priorytety.

Jak zrobić, żeby było dobrze - moja wersja.

Osobiście czasem testuje do upadłego, czasem testuje tylko trochę, a zdarzają się i takie przypadki, kiedy wcale czegoś nie testuje. Idąc tą kolejnością, do upadłego testuje wszystko to, co jest dla mnie niezmiernie ważne. Umówmy się, implementacja implementacji nie równa. Zawsze jednak muszę dobrze ocenić sytuację, zanim zdecyduje jak wytrwale będę testował. Powiedzmy, że mam funkcjonalność i wiem, że jeśli jej dobrze nie przetestuje, może być ciężko zauważyć, że coś nie działa jak trzeba. Klient po miesiącu zgłosi się do nas i powie, że mamy błąd, który przez cały miesiąc zaniżał zyski.

Idąc dalej, obszernie testuje też wtedy, gdy chciałbym uniknąć sytuacji, w której coś nie działa jak trzeba, a my rozszerzamy taki kod tylko po to, aby potem okazało się, że trzeba go mocno przepisać. Może już teraz podsumuję, super obszernie testuje wszystko co może odbić się sporymi konsekwencjami. Najczęściej, z takim obszernym testowaniem przychodzę na story branch'e.

Kiedy testuję tylko trochę? Wtedy, jeśli mam naprawdę problem z ilością czasu. Niby nie powinniśmy wydawać czegoś szybciej kosztem jakości. Jednak powiedzmy sobie szczerze, ile razy było tak, że nie potrafiliśmy oprzeć się presji (tak, nie pokusie tylko presji)? Jednak, to nie tak, że po prostu przeskakujemy dalej. Zawsze, kiedy decyduje się coś zostawić nieprzetestowane, jest

to obszar, którego jestem stosunkowo pewien; jest mało kluczowy, mało ryzykowny, bądź wiem, że nawet jeśli tam będą błędy, to szybko się o nich dowiemy. Trudno, będziemy naprawiać... , ale jest duża szansa, że nie będziemy i wydamy na czas, a wszyscy będą zadowoleni.

Takie testowanie wybiórcze, a czasem nawet całkowity brak testowania najłatwiej jest zastosować na kodzie, który i tak idzie do story brancha. Czasem, aby coś przetestować, potrzeba czasu nie dlatego, że sama ta czynność jest czasochłonna, ale dlatego, że nie mamy środowiska, danych, czy nawet BACKENDU! Tworzenie sztucznych danych, powygimnastykowane setupowanie zintegrowanych serwisów, bądź mockowanie funkcjonalności byłoby na tyle czasochłonne, że szkoda na nie czasu. Jak nie będzie działać, to wyjdzie to na story branchu albo, na story branchu setup testowania zajmie mi niewiele czasu, ponieważ wszystkie potrzebne rzeczy już tam będą.

Na ten temat można by na pewno dłużej i obszerniej rozmawiać. Podsumowując, jeśli podejmuje decyzje o tym, czy coś testować czy nie, muszę wziąć pod uwagę:

- po co testuje, czego chce się dowiedzieć: może nic się nie stanie jeśli z tym zaczekam?
- ile razy jeszcze wrócę do danej funkcjonalności zanim trafi to na produkcję: może gdy będzie więcej kodu, moje testowanie będzie łatwiejsze i bardziej skuteczne?
- jakie ponoszę ryzyko jeśli pojawią się defekty: może szybki fix będzie lepszy, niż godziny setupowania-kombinowania?
- dlaczego coś jest czasochłonne, i czy będzie szansa zrobić to szybciej, jeśli tylko trochę zaczekam: może pod koniec story, albo sprint później, będę mieć więcej kodu, lepszą integracje i środowisko bliższe produkcyjnemu?

W ten sposób, jeśli dobrze zadziałamy, możemy zaoszczędzić trochę czasu i sprawić, że sprint z punktu widzenia testowania będzie bardziej płynny. Aczkolwiek, nie ma co się bać wstrzymywać stories jeśli naprawdę uważamy, że jest taka potrzeba. Wiele razy tak robiłem, dlatego, że oceniłem wysokie ryzyko. Nawet jeśli ktoś czekał, czasem musi się tak zdarzyć.

Czy wy macie jakieś swoje strategie na omijanie testowania?
A może macie jakieś komentarze do tego co napisałem?

Część III

Testowanie w praktyce

Wpis #38 Dlaczego testowanie nie powinno mieć deadline'u

W światku testerskim istnieje taka zasada, że testowanie gruntowne jest niemożliwe. Często jak kandydat na rozmowie rekrutacyjnej chce zabłysnąć tajemną wiedzą, to właśnie o tym wspomina. Zapewne każdy tester w trakcie swojej pracy niejednokrotnie był w stanie potwierdzić tę tezę.

Produkty komercyjne mają to do siebie, że zwykle są bardziej złożone niż proste funkcje, czy programy składające się z kilku funkcji pisane na potrzeby własne dla ułatwienia sobie życia. Ta złożoność zwykle rośnie. Wraz z tą złożonością rośnie złożoność przypadków, kiedy brzydko mówiąc coś może się wydupić.

I choć czasem moglibyśmy przetestować coś gruntownie (o ile złożoność jest niewielka), o tyle nawet w takim przypadku agile nam na to nie pozwala: "dostarczaj często i szybko, klient czeka, pieniądze uciekają, konkurencja nie śpi".

Co z tym deadlinem? Ustawianie deadline'u np. do 13.01.2019 oznaczałoby, że co? Do 13.01 oddam w pełni przetestowaną funkcjonalność? A co, jeśli znajdziemy błędy 14.01? Tester do zwolnienia. A release leży, bo nikt się tego nie spodziewał, w końcu 13 stycznia coś zostało 'przetestowane'. Tego typu postrzeganie testowania stoi w sprzeczności z zasadą wspomnianą na początku.

Zamiast traktować testowanie zadaniowo i wyznaczać mu deadline, po którym zakładamy, że praca testera została zakończona i tester może iść dalej, polecam rozkminiać testowanie jako proces ciągły.

Testuj często, nie przestawaj testować. Nie traktuj czegoś jako przetestowane, ponieważ możesz zacząć bazować na błędnym założeniu, że wszystko jest ok, kiedy faktycznie może nie być. Jeśli testowałem i nie znalazłem żadnych problemów, to nie znaczy, że ich tam nie ma. To znaczy, że to, co byłem w

stanie przetestować nie wykazało żadnych problemów - biorąc pod uwagę to, co wiem o produkcie, funkcjonalności, moją pomysłowość, ograniczenia czasowe i inne zmienne. Ponadto, jeśli teraz nie znalazłem żadnych problemów to nie znaczy, że kiedyś tam się nie pojawią (jeśli system jest rozwijany).

Oczywiście nie kłócę się całkowicie z zadaniowością. Jest ona przydatna, żeby testowanie w jakiś sposób zaplanować i ustrukturyzować. Chodzi o to, żeby nie wprowadziła ona błędnego postrzegania testowania ze szkodą dla całego zespołu.

Wpis #39 Case Study: dlaczego testowanie nie ma końca

Wcześniej pisałem o tym, że testowanie powinno być ciągłe. Niedługo trzeba było czekać, aby pojawił się świeży przykład z projektu.

Krótko o funkcjonalności, bo niedawno wypuściliśmy funkcjonalność eksportu. Architektura bazuje na tym, że na s3 (AWS) trafia plik, s3 wysyła powiadomienie na topic, że plik jest gotowy do pobrania, a ten wysyła powiadomienie do serwisu, który ten plik pobiera. System notyfikacji s3->topic->serwis jest ustawiany automatycznie przy starcie serwisu.

Eksport przetestowałem już wcześniej, ale nie śpieszyło się z zamknięciem zadania, więc postanowiłem jeszcze pogrzebać i poczekać na jakieś dodatkowe pomysły. Okazało się jednak coś innego. Dlatego, że grzebałem w tej funkcjonalności dorywczo przez kilka dni (w przeciwieństwie do jednej sesji) zauważyłem, że topic dla notyfikacji jest zmieniany z jakiegoś powodu. Zauważyłem to tylko dlatego, że w trakcie sesji wszystko działało, przy drugim podejściu przestało działać. Co się zatem stało?

Na jednym z naszych komputerów serwis był startowany lokalnie do developmentu i lokalny setup korzystał częściowo z konfiguracji, z której nie powinien korzystać (zwykła pomyłka, każdego może spotkać). To sprawiało, że serwis na serwerze wstawał, konfigurując AWS'a poprawnie, a lokalny start serwisu (w późniejszym czasie) podmieniał te konfiguracje.

Eksport jest w naszej aplikacji kluczową funkcjonalnością. Z pozoru można by powiedzieć, że działa. Tak wynikło z jednej sesji testowania. A co pokazała druga? Że mamy problem, który nie został wykryty od razu i raczej nie mógł być wykryty w trakcie jednej sesji.

Czego uczy nas ten przypadek?

Po pierwsze potwierdza tezę, że podchodzenie do testowania zadaniowo może być szkodliwe. I tak jak wspomniałem nie chodzi o to, aby trzymać zadanie otwarte w nieskończoność. Nie chodzi też o to, aby zamykać zadania i o nich zapominać. Jakiś czas temu doszedłem do wniosku, że w zasadzie płaci mi się (częściowo) za regularne używanie systemu - przemyślane, zaplanowane, mające miejsce na różnych poziomach, ale wciąż to po prostu używanie.

Druga sprawa w kontekście wspomnianych heurystyk. To już jest kolejny raz, gdy problemy sprawiają zmienne środowiskowe. Co prawda ten przypadek jest dość nietypowy ale wciąż.. Skoro problemy z nimi się powtarzają, testując kolejny raz na pewno zwrócę większą uwagę na to, co się dzieje ze zmiennymi, tzn. gdzie i jak są definiowane i jakie procesy od nich zależą.

Wpis #40 Niebezpieczne frameworki

Clickbait. Mówiąc o frameworkach nie mam na myśli technologii. Odnoszę się do ram, w które często jesteśmy wkładani lub, w które sami siebie wkładamy.

Jedną z odpowiedzialności QA jest nie dać się obramować. Żeby pokazać, co mam na myśli posłużę się przykładem:

Przykład:

- Kiedy skończysz testować?
- A czemu potrzebujesz to wiedzieć?

To pytanie, a także inne pytania tego typu, często padają podczas normalnej codziennej pracy. Jest to pytanie, które wprowadza ramy, czy sobie zdajemy z tego sprawę, czy nie.

Jeśli jestem zapytany o to, kiedy kończę testować, to nieumyślnie jestem zmuszany do podania konkretnego terminu. Pytanie, czy adresata interesuje tylko termin? A może chce wiedzieć jak się ma to, co napisał i dodatkowo zastanawia się kiedy uzyska tę odpowiedź? Jeśli odpowiem, że skończę testować za godzinę, to odpowiem na pytanie, ale czy odpowiem na oczekiwania adresata? Być może tak, być może nie. Być może zaraz po odpowiedzi dostanę kolejne pytania, bo adresat zdał sobie sprawę, że właściwie to nie termin go interesuje, a wyniki testów. Może już część wyników mogę mu przekazać. Bez względu na intencje często albo jesteśmy spychani w kontekst, albo my to robimy nieumyślnie i nieświadomie.

Jak się to ma do bycia QA? W podobnej sytuacji jesteśmy stawiani pracując z klientem, z wymaganiami, podczas planowania, podczas implementacji, podczas code review i w każdej innej fazie iteracji i całego cyklu projektowego.

Jesteśmy spychani w ramy, które upośledzają nasze myślenie. W trakcie komunikacji to nie problem, zawsze możemy skorygować treść informacji, aż finalnie wymienimy się pożądaną treścią. Problem jest o tyle większy, że od np. dokumentacji nie dostaniemy feedbacku mówiącego, że źle ją interpretujemy albo, że jest niekompletna. Musimy sami na to wpaść.

Zadaniem wszystkich członków zespołu, a szczególnie QA, jest wychodzić poza te ramy - kwestionować. QA ma wiedzieć, że rzeczywistość może być inna niż ta, którą otrzymuje (np w postaci zadania z określonymi Acceptance Criteria). Dociekliwość i nieufność może być upierdliwa, czasem sprawia,

że możemy dyskutować sami ze sobą ale to dzięki niej jesteśmy w stanie dostrzegać rzeczy, które wychodzą poza to, co mamy przed oczami.

To samo ma miejsce w trakcie testowania funkcjonalności, która każe nam wybrać jeden element i następnie go wysłać. Tester zadaje pytanie “dlaczego jeden, a mogę więcej?”. Co się stanie, jak nie wyślę żadnego? A co jeśli wyślę dwa elementy jednocześnie? A może mogę wysłać coś innego niż to, co przewidujemy? Kwestionujemy ramy.

Możemy sobie wyobrazić jak ‘szerokie’ byłoby testowanie, gdybyśmy zaakceptowali wprowadzoną ramę:

- Program: wybierz jeden element i wyślij
- Tester: wybiera jeden element i wysyła
- Program: element został wysłany
- Tester: działa, idę na kawę

Jeśli rezultatem obecnej sytuacji są problemy, zakwestionujmy rzeczywistość i zastanówmy się czego nie dostrzegamy, wyjdźmy poza aktualne ramy.

Wpis #41 Wczesne testowanie oszczędza czas i pieniądze

W tym feedzie rozszerzę treść trzeciej zasady testowania.

Wczesne testowanie oszczędza czas i pieniądze.

Po pierwsze, czym jest wczesne testowanie? Popularne shift-left, agile testing, (częściowo) quality assurance, czyli testowanie od samego początku trwania projektu oraz każdej iteracji.

Klasyczne postrzeganie testowania mówi o tym, że po napisaniu funkcjonalności przychodzi czas na jej testowanie. Wszystkie powyższe hasła odnoszą się do innego postrzegania testowania i powstały jako odpowiedź na transformację w metodyki zwinne, w których klasyczne testowanie zwyczajnie się nie odnajduje. Zakładają one, że testowanie nie musi wcale mieć miejsca tylko po tym, jak funkcjonalność została stworzona, a testowaniu może podlegać więcej rzeczy niż tylko funkcjonalności.

Jeśli testowanie to konfrontowanie naszych oczekiwań względem rzeczywistości i eksploracja rzeczywistości w poszukiwaniu rzeczy nieoczekiwanych, to możemy to robić na każdym etapie trwania projektu oraz iteracji: zbieranie wymagań, projektowanie, planowanie, estymowanie, research, implementacja, code review itd... Zwykle to, co powstrzymuje nas przed wczesnym testowaniem, to brak odpowiedniego nastawienia (obserwacja, myślenie krytyczne, kwestionowanie) oraz brak oczekiwań. Niech pierwszy rzuci kamień, kto nigdy nie zaczął realnie myśleć o funkcjonalności dopiero w trakcie jej implementacji.

Gdzie te oszczędności?

Zaczynając od początku, czyli pierwsze fazy typu planowanie, projektowanie. Znalezienie błędów na tym etapie pomaga nam uniknąć ogromnego nakładu pracy. Jeśli implementujemy coś, co potem okazuje się nietrafione to nie dosyć, że straciliśmy czas, to jeszcze potrzebujemy dodatkowego czasu na nową implementację.

Czasem znajdziemy błędy, które niekoniecznie przekreślą całą implementację ale wymuszą pewne zmiany. Znajdowanie takich błędów wcześnie również będzie oszczędnością czasu i kosztów. Prócz tego, że uniknęliśmy konieczności poprawek, oszczędziliśmy sobie problemów pisząc dalsze linijki kodu w oparciu o błędne założenia.

W przypadku testowania na otwartym PR inaczej wygląda poprawa błędów, gdy zgłaszamy autorowi to, czy tamto, a ten jest w stanie szybciutko wprowadzić poprawki. Inaczej, gdy trzeba tworzyć bugi, otwierać nowe branche, implementować,

robić nowe code review (pamiętajmy, że code review angażuje pozostałych członków zespołu).

Poza tym zadanie, które leży i czeka na testera stoi w sprzeczności z koncepcją lean. Problemy znalezione w takim zadaniu wymagają powrotu do niego, co wymusza zmiany kontekstu i równoległą pracę, jeśli programista już zaczął pracować nad innym zadaniem, a tak zwykle jest. Ponadto opóźniamy integrację tego zadania z innymi i możliwość zidentyfikowania problemów, które mogą wyniknąć z integracji. Ostatecznie zwykle robimy takie poprawki na koniec sprintu, dochodzi tutaj presja czasu i spadający na głowę deadline. Śpieszymy się, wyłączamy myślenie, a to dobra droga do kolejnych błędów.

No dobra ale skoro będę testować na poziomie PR, to potem będę tracił czas na powtarzaniu testów, gdy już całe story będzie gotowe. Niekoniecznie. Egzekucja testów jest często najszybsza z całego procesu testowania. Zwykle najwięcej czasu poświęcamy na to, aby zorientować się w sytuacji, poznać dany kawałek kodu, przygotować sobie dane testowe, środowisko, jakieś narzędzia, przemyśleć co chcemy sprawdzić. Jeśli raz to zrobimy, samo sprawdzanie łatwo i szybko można powtórzyć (nie zawsze).

Oczywiście błąd błędowi nie równy. Nie wszystkie zasługują na priorytet lub nawet na naszą uwagę i nie wszystkich błędów tyczy się powyższe. A co wy myślicie o tym? Znacie jakieś przykłady, które udowadniają w jaki sposób testowanie (wczesne) oszczędza czas i pieniądze? A może są takie sytuacje w których jest wręcz przeciwnie?

Wpis #42 Tanie testowanie

Robią to prawie wszyscy (nie mam pewności, że wszyscy wszyscy). Jest to stały element procesu, niepodważalny, wręcz oczywisty. Czy to dobrze? Dobrze, bo zawsze z niego korzystamy. Źle bo czasem nie zdajemy sobie sprawy, jak dużo wartości wnosi i być może nie zawsze przykładamy się tak, jak

trzeba. A szkoda bo jest to tani i efektywny sposób na wczesne testowanie.

Tajemniczo. Mam na myśli code review. Code review to pierwsza aktywność testowa, która odbywa się z udziałem osób innych niż autor kodu (ok, no może w przypadku pair programming'u da się jeszcze wcześniej). Niemniej jest to sposób na super wczesne testowanie. W końcu ile razy byliście zmuszeni tworzyć nowe zadania w Jirze, otwierać nowe branche, robić retesty tylko z powodu błędów znalezionych podczas code review? Pewnie były i takie sytuacje (uzasadnione), ale generalnie nie zdarza się to prawie wcale. Problemy poprawiane w ramach code review to najtańsza forma tested -in quality (vs built-in) - nie mam pojęcia jak to po polsku.

Code review to forma testowania statycznego (stojąca naprzeciw testowaniu dynamicznemu), która pozwala nam w najszybszy i najbardziej elastyczny sposób przetestować kod źródłowy. Łatwo do tego zaangażować kilka osób i generalnie zysk z poświęconego czasu jest zwykle bardzo bardzo duży. Zwykle, niestety nie zawsze.

Nie zawsze, ponieważ żeby zysk był duży, trzeba robić code review sumiennie. Przejrzenie kodu na szybko będzie jeszcze tańsze, ale również dużo mniej wartościowe. A jeśli robimy je sumiennie zyskujemy:

- Wczesne wykrywanie problemów - tym samym minimalizujemy koszty defektów
- Wykrywanie problemów związanych z funkcjonalnością, utrzymywalnością, rozszerzalnością, testowalnością, stabilnością itd. Całkiem sporo prawda?
- Wykrywanie problemów jest możliwe bez konieczności uruchamiania kodu, narzut na rozpoczęcie testowania jest okropnie niski
- Możliwość wspólnej dyskusji nad efektywnością zaimplementowanego rozwiązania danego problemu
- Kilka różnych punktów widzenia poddających kod krytyce

- Możliwość poznawania nowych rozwiązań i dzielenia się wiedzą
- Możliwość nauki na cudzych błędach, dostrzegania problemów, których potem staramy się nie powtarzać w naszym kodzie np. czytelności
- Jest kilka zasad, które pozwalają zwiększyć skuteczność naszego code review:
- Dowiedz się czego dotyczy PR, jakie założenia ma realizować, poznaj funkcjonalność - będzie Ci łatwiej zrozumieć, co czytasz
- Bądź strumieniem danych, podążaj od inputu do outputu przechodząc tę samą drogę, które przejdą dane
- Nie czytaj kodu ot tak, zastanów się jakich problemów chcesz szukać
- Korzystaj z timebox'ów, uciekanie myślami podczas długiego code review może wiele nie pomóc
- Świadomie zarządzaj czasem poświęcanym na PR'y, znajdź swój własny sposób na to, jak efektywnie czytać kod, ile czasu realnie jesteś w stanie się skupić
- Namawiaj do tworzenia małych PR'ów

Choć mało osób pewnie zdaje sobie z tego sprawę, ale code review to jedna z pierwszych form (jak nie pierwsza) angażowania całego zespołu w zapewnianie jakości, czyli ukłon w stronę whole-team approach.

Wpis #43 Skąd biorą się bugi

Choć w branżach opartych o wytwarzanie (z ang. production) istnieją ambitne poglądy dążenia do wbudowanej jakości, zasadniczo jeszcze ani razu nie miałem okazji usłyszeć, że komuś się udało. Udało tzn. ogłosić wszem i wobec, że ich

oprogramowanie (czy produkty) są nieskazitelne zaraz po tym, jak powstaną i nie wymagają totalnie żadnej aktywności z zakresu inspekcji, czy weryfikacji.

Łatwo więc można stwierdzić, że o ile wbudowana jakość to taki święty graal produkcji, o tyle istnieje szansa, że jest ona niemożliwa do zrealizowania. Możliwe natomiast jest dążenie do minimalizowania błędów, które można znaleźć na wspomnianych etapach weryfikacji, czy inspekcji.

Świadomość tego skąd pochodzą błędy, może w dużym stopniu pomóc przy analizie naszego procesu, a dzięki temu możemy modyfikować go tak, aby unikać tych źródeł (lub znów, minimalizować ich natężenie).

Stąd, chciałbym przytoczyć pięć takich źródeł. Nie są to wszystkie możliwe, ale są godne uwagi:

- Problemy komunikacyjne. Ile razy zarówno w karierze, jak i w życiu okazało się, że zwyczajnie się nie dogadaliśmy? Mnie osobiście tak się zdarza. Z czego to wynika? Na pewno powodów może być cała lista, a na problemy komunikacyjne można poświęcić książki. Niemniej, jeśli chodzi o tzw. kontekst lokalny, czy nawet konkretną sytuację raczej jesteśmy w stanie stwierdzić, dlaczego się nie dogadaliśmy pod warunkiem, że poświęcimy temu odrobinę autorefleksji lub wspólnej analizy. Faktem jest, że problemy komunikacyjne to czasem rozjechane lub źle zaimplementowane wymagania, problemy z interfejsami, niespójny format kodu itd.
- Złożoność systemu. Czytałem kiedyś o takiej zasadzie, że wraz z rozwojem systemu maleje czas, który musimy poświęcać na development, a rośnie czas potrzebny na utrzymanie i inspekcje. Ciekawy jestem, czy każdy się z tym zgodzi. Ja sam widzę, że jeśli chodzi o to, jak złożoność przekłada się na potrzeby testowe to fakt, rosną one wraz z systemem (jak zwykle - nie w każdym przypadku). Im bardziej złożony jest system tym łatwiej się w nim pogubić, proste. Dlatego tak istotne jest pilnowanie prostoty, dokumentacji, czy innych praktyk, które wspierają ogarnianie tej złożoności.

- Zmieniające się wymagania. Jeśli wymagania zmieniają się zanim napiszemy pierwsze linijki kodu to spoko. Wiele problemów to nie wprowadzi. Ale jeśli już mamy system i nagle okazuje się, że zmiana wymagań to nie tylko roszady w backlogu ale także modyfikacje istniejącego kodu, które czasem polegają na unikaniu przepisywania i próbach naciągania tego, co jest pod wymagane zmiany może okazać się, że skomplikujemy sobie rozwiązanie i wprowadzimy błędy. Ta sama sytuacja ma miejsce w przypadku gdy nasze wymagania są ulotne jak para wodna ale skraplają się po pewnym czasie, zaskakując nas ponownie swoją obecnością. Tzn. że albo zapomnieliśmy o pewnych wymaganiach, albo nie były one wystarczająco oczywiste, żeby uwzględnić je planując architekturę. Wtedy implementacja takich wymagań może wyglądać podobnie jak implementacja tych zmieniających się - będziemy naciągać.
- Słaba dokumentacja. Dokumentacja bardzo często uważana jest jako strata (z ang. waste). Trudno się z tym nie zgodzić, jeśli mamy jej sporo i nikt z niej nie korzysta, a pisana jest jako sztuka dla sztuki. Często jednak strzelamy sobie takim postrzeganiem w kolano, nie dokumentując prawie nic. W ten sposób na początku jest spoko, ale przychodzi moment, gdzie przestajemy ogarniać co się dzieje, jesteśmy skłonni do popełniania błędów. Czasem wystarczy jak używamy docstringów albo posiadamy aktualny graf z architekturą. Wszystko zależy od potrzeb.
- Czynnik ludzki. Tak naprawdę jest to najczęstsze źródło błędu (a jednak spadło na sam koniec). Stoi on na końcu dlatego, że najciężej jest go sklasyfikować. Czynnik ludzki to po prostu nasza skłonność do popełniania pomyłek. Wystarczy gorszy dzień i już. Jako remedium można próbować oszukiwać samych siebie korzystając z narzędzi pomagających w pracy. Np. mam tendencje do chaotycznego planowania pracy dziennej i zapominam o spotkaniach - używam przypominajki. Mam tendencje do zapominania o domergowywaniu branchy - używam checklisty (to są akurat moje problemy, z którymi walczę - czasem zapominam ustawić przypominajke), itd. Każdy powinien znaleźć coś dla siebie.

Wpis #44 Sztuka zadawania pytań

Kolejnym narzędziem używanym do zapewniania jakości jest zadawanie pytań (umiejętne). Bo oczywiście nie chodzi mi o to, że nagle junior QA zacznie zadawać pytania, a jakość wystrzeli w górę niczym rakieta.

Jak już zdążyłem wspomnieć wielokrotnie, najpoważniejsze problemy to te, o których nie wiemy. Dlatego, że wraz cyklem życia oprogramowania oraz fazami iteracji rośnie nie tylko nasza baza kodu, ale także ryzyko i ponoszone koszty. Czyli im później problem zidentyfikujemy, tym konsekwencje mogą być bardziej kosztowne. Sztuka umiejętnego zadawania pytań jest cennym narzędziem, którego używamy by te problemy zidentyfikować.

Co mam na myśli poprzez umiejętne zadawanie pytań? Inaczej jest, gdy pytamy, bo wiemy, że adresat zna odpowiedź. Inaczej, gdy wiemy, że może jej nie znać. Pytań możemy używać, aby rozjaśniać sytuację, aby skłaniać do myślenia, aby wyprowadzać z błędu i wiele innych.

Istnieją pytania rozjaśniające (clarifying questions). Podczas dyskusji, w której udziela się wiele osób może zdarzyć się tak, że niby mówimy o tym samym, a jednak każdy używa innych założeń i interpretuje tę samą dyskusję inaczej. Zadając pytania rozjaśniające możemy skłonić do ponownego przedstawienia konkluzji naszej dyskusji w taki sposób, aby była ona jednoznaczna i posiadała wszelkie potrzebne szczegóły. Czasem w trakcie zadawania takich pytań niektóre osoby zdają sobie sprawę z tego, że jednak nie rozumieją do końca tematu, bądź wysnuły złe wnioski. Właśnie do tego dążymy. Jedna wspólna wizja to jest to, co pomaga nam unikać błędów.

Przykłady:

- Czy chodziło Ci o...?
- Masz na myśli...?

Pytania eksplorujące (adjoining questions) służą nam do tego, aby wyłonić luki lub błędy logiczne w myśleniu. Dyskutując nad jakimś rozwiązaniem mamy tendencje do skłaniania się ku końcowi w momencie, gdy zaczynamy się w czymś zgadzać. Bywa to jednak pułapką, tzn. przestajemy skupiać się na całym obrazie problemu, a zaczynamy skupiać się na tylko tym aspekcie, w którym jesteśmy zgodni. Stąd takie pytania pozwalają na chwilkę oderwać się rozmówcom od wniosków i jeszcze raz zastanowić nad swoim rozwiązaniem.

Przykłady:

- Jaki konkretnie problem to rozwiązuje?
- Dlaczego uważasz, że to dobre rozwiązanie?

Pytań drążących (funneling questions) używamy wtedy, gdy chcemy wyjąć na powierzchnię głębokie założenia, na których ktoś bazuje. Jeśli ktoś coś stwierdza nie robi tego ot tak. Najczęściej bazuje na jakichś założeniach, które wierzy, że są właściwe. A co jeśli nie są? Próbując dokopać się do czyichś założeń robimy przysługę sobie i całemu zespołowi. Szybciej dojdziemy do wspólnych wniosków, a szansa, że te wnioski będą właściwe rośnie.

Przykłady:

- Co masz na myśli mówiąc, że...?
- Czemu uważasz, że to dobry pomysł?

Ostatnie są pytania systemowe (elevating questions). Tego rodzaju pytania bazują na myśleniu systemowym. Czasem już na samym początku rozpoczynamy dyskusję z pominięciem pełnego obrazu, a czasem wraz z dyskusją schodzimy coraz niżej, zapominając o myśleniu systemowym. Pytania systemowe są tak zbudowane, aby wyciągnąć rozmówców z niższych poziomów i skłonić do myślenia systemowego.

Przykłady:

- Jak zintegrujemy to z innym serwisem?

- Czy to rozwiązanie może wpłynąć na inne części systemu?

Zapewne sposobów na skategoryzowanie pytań jest więcej. Tutaj chciałem tylko podkreślić, że choć wydaje się być to błahe, rozwijanie swoich umiejętności w odpowiednim zadawaniu pytań może przynieść wiele cennych efektów i sprawić, że unikniemy kilku fakapów. Nie zachęcam od razu do literatury (choć można poczytać sobie np. o sokratycznej metodzie dochodzenia do prawdy), zachęcam do świadomego zadawania pytań, bo to już będzie duży krok w kierunku poprawy jakości.

Wpis #45 Perspektywa ma znaczenie

Nasze postrzeganie świata polega na jego interpretacji. Tutaj w grę wchodzi kognitywistyka ale nie będę się zbytnio w to zagłębiać. Chodzi mi o to, że sposób, w jaki będziemy coś postrzegać jest mocno uzależniony od tego, w jaki sposób to coś zinterpretujemy. Na interpretacje wpływa wiele czynników, np. nasze dotychczasowe doświadczenie, nasze wewnętrzne wartości lub właśnie perspektywa, czyli nasze podejście do interpretacji.

To by było na tyle, jeśli chodzi o nauki poznawcze. Wracając do IT, co nam daje taka informacja? Jeśli sposób postrzegania świata (systemu) zależny jest między innymi od perspektywy to znaczy, że świat (system) zostanie zinterpretowany na tyle sposobów, ile mamy perspektyw.

Przechodząc jeszcze bardziej do konkretów: analiza wymagań. Jeśli nasza analiza wymagań może być różna ze względu na nasze różnice poznawcze, to możemy mieć problem. I zwykle mamy, ponieważ wymagania nie mogą być kwestią interpretacji, mają być jednoznaczne, a to oznacza, że musimy działać wbrew naszej kognitywistycznej naturze aby takie one były. Dlatego też istnieją role, które mają na celu wyeliminowanie możliwości interpretacji i sporządzanie wymagania w sposób jednoznaczny,

tak aby 10 różnych perspektyw zinterpretować je w ten sam sposób.

Płynie z tego krótki wniosek, choć nie przeskoczymy lat treningu i doświadczenia w pracy z wymaganiami, identyfikowanie elementów dwuznacznych, które można poddać interpretacji jest pierwszym i całkiem sporym krokiem w stronę płynnej pracy z wymaganiami.

Wpis #46 Zarządzanie własną uwagą

Właściwie zarządzanie tzw. okresem skoncentrowanej uwagi. Zarządzanie nim może być zarówno dobre, złe, jak i nieistniejące. Nieistniejące nie od razu oznacza złe. Możemy intuicyjnie pracować w sposób efektywny tj. zgodny z ograniczonym okresem koncentracji. Możemy też nieświadomie sabotować efekty naszej pracy, gdy narzucimy na siebie zbyt wiele do zrobienia podczas jednego posiedzenia (czy tam sesji).

Zarówno w przypadku pracy z wymaganiami, jak i generalnie podczas jakiejkolwiek pracy, warto znać siebie i wiedzieć ile czasu realnie jesteśmy w stanie się skupić, a od którego momentu zaczynamy uciekać myślami do innego świata.

W ten sposób pracując nad wymaganiami, zaczynamy omawiać lub spisywać potrzeby naszego klienta. Jedno story za drugim. Z każdym kolejnym coraz ciężej jest nam się skupić, zrozumieć sedno tych potrzeb i realnie je analizować (szukając problemów, zależności czy luk). Finalnie mija godzina, mamy ostatnie story do zrobienia, ustalamy coś tam na szybko, przytakujemy sobie nawzajem, żeby wreszcie skończyć. Widzimy dokąd to zmierza? A czemu jako QA mnie to obchodzi? Bo jeśli takie sytuacje finalnie przekładają się na niższą jakość, to tego właśnie chciałbym unikać.

Jeśli świadomie podzielimy sobie pracę w taki sposób, aby wymagać od siebie krótkich okresów koncentracji, ten sam czas,

jaki poświęcimy na pracę będzie wykorzystany lepiej, uważniej, efektywniej. Czy to pracując z wymaganiami, czy testując coś, jeśli chcemy dbać o jakość, to unikajmy sytuacji, które tę jakość nam obniżają.

Istnieją jeszcze inne korzyści dzielenia sobie pracy (np. wychodzenie z pułapek myślowych) ale dziś chciałem podać tylko jedną korzyść zarządzania uwagą. Są też sposoby na to, aby tę uwagę troszeczkę postymulować lub odświeżać do pewnego stopnia np. uzewnętrznianie własnych myśli za pomocą rozmowy, bądź robienia notatek (zależy od zadania) lub praca wspólna i interakcja. Choć dzięki takim zabiegom będziemy w stanie troszkę wydłużyć ten efektywny czas skupienia, prędzej czy później i tak się wypalimy.

Tak więc zgodnie z zasadą divide and conquer dzielmy pracę nad wymaganiami zgodnie z powyższym.

Wpis #47 Od tego jestem specjalistą żeby wiedzieć lepiej

Ostatnio na nowo przeglądam książkę `Lessons Learned in Software Testing`. Czytałem ją dość dawno i teraz, gdy wracam do niej i przeglądam spisane w niej lekcje, okazuje się, że większość z nich dość mocno ugruntowały się w mojej głowie. Głównie dlatego, że doświadczenie (choć krótkie) wiele razy pokazało, że to wszystko ma sens. Stąd książkę polecam. Napisana przez weteranów świata testerskiego i szkoły kontekstowej.

Jedna z lekcji brzmi:

`Don't expect anyone to understand testing or what you need to do it well`

Na samym początku swojej drogi byłem przekonany, że to środowisko, w którym się pojawiłem powinno zdefiniować

moją rolę. Oczekiwałem, że będę prowadzony za rękę i od zespołu dowiem się, co mam robić. Następnie przyszło odrobinę frustracji, gdy nie dowiadywałem się zbyt wiele, a zespół okazał się wyposażony w postrzeganie testera jako klikacza po GUI. Nie winię zespołu. Testowanie wciąż jest promowane w ten sposób. Zwykle przez osoby, które nie mają zbyt dużej wiedzy o testowaniu i co gorsza przez samych testerów, którzy próbują sprzedać 'przystępne' szkolenia dla ludzi chcących się przebranżowić wciskając im, że każdy może testować. Są też firmy, które wciąż używają testerów w ten właśnie sposób, ale tu nie o tym.

W kontekście agile'owym, gdzie tester, czy QA pracuje jako członek interdyscyplinarnego zespołu to on jest odpowiedzialny za to, jak zespół będzie zapewniał jakość i testował. To on ma być specjalistą w tej dziedzinie (a przynajmniej do tego dążyć) i źródłem wiedzy dla zespołu.

Dlaczego o tym mówię? Jako tester w zespole nie można liczyć na to, że zespół podrzuci mi taska, a ja go sprawdzę i tylko na tym będzie polegała moja praca. Nie można też oczekiwać od wszystkich, że będą się o tak znali na testowaniu, nie można winić za złe postrzeganie testowania. Warto natomiast zdać sobie z tego sprawę, wziąć odpowiedzialność za rolę, jaką się przyjęło i poszerzać swoją wiedzę, aby służyć zespołowi jako specjalista a nie klikacz. W przeciwnym wypadku cała wiedza i doświadczenie pozostaje nieużywane, a zespół podejmuje mniej lub bardziej (zależy od członków zespołu) świadome decyzje dotyczące zapewniania jakości. Pamiętajmy, że to cały zespół zapewnia jakość, a nie jeden jedyny QA czy tester w zespole.

Wpis #48 Jak myśli tester

Moim zdaniem, jeśli jesteśmy świadomi tego, w jaki sposób działamy (czy myślimy) możemy efektywniej się rozwijać i wykorzystywać swoje umiejętności.

Jeśli chodzi o testowanie wyróżnia się cztery sposoby myślenia sprzyjające testowaniu:

Myślenie techniczne:

Pozwala zrozumieć technologie i łączyć fakty techniczne. Aby myśleć techniczne, potrzebujemy do tego technicznej wiedzy. W momencie, gdy zaczynamy tę wiedzę rozszerzać, będziemy w stanie coraz lepiej rozumieć i identyfikować związki przyczynowo skutkowe, jeśli chodzi o nasze systemy. Jeśli rozumiemy jak działają nasze systemy, łatwiej nam odnaleźć w nich nieścisłości.

Myślenie kreatywne:

Jest to umiejętność 'produkowania' pomysłów i dostrzegania możliwości. Myśląc kreatywnie wychodzimy poza rzeczywistość, którą mamy przed sobą. Dzięki temu, nasze testowanie będzie w stanie pokryć zarówno te rzeczy, które mamy przed sobą (jesteśmy w stanie je zobaczyć) i także te, których nie widzimy. Do tych rzeczy doprowadza nas myślenie kreatywne.

Myślenie krytyczne:

Myślenie krytyczne to umiejętność kwestionowania rzeczywistości. Kwestionowanie, które niekoniecznie wiąże się z brakiem zaufania, a bardziej wynika ze sceptyczności (choć jedno może kłócić się z drugim). Dzięki tej umiejętności dochodzimy do problemów i nie odpuszczamy mimo, że wszystko wydaje się być w porządku.

Myślenie praktyczne:

Jak to mówią last but not least. Dzięki myśleniu praktycznemu przekładamy to, co powstaje w naszej głowie na rzeczywistość. Robimy to w sposób sensowny, pragmatyczny, czyli dostosowany do naszych możliwości, naszego kontekstu i realiów. Myślenie praktyczne pozwala nam dostrzegać sposoby na to, jak zaimplementować koncepcje powstające w naszej głowie.

Myśląc jak tester trzeba przede wszystkim zakładać, że rzeczywistość, którą mamy przed sobą może być inna. Najczęstszym źródłem problemów jest tzw. tunnel vision (widzenie tunelowe). Chodzi o to, że patrząc na rzeczywistość, patrzymy w sposób ograniczony. Mamy tendencje do wpadania w wąskie zakresy myślenia, nie dostrzegając pełnego obrazu.

Bez względu na to, czy spędzimy na testowaniu godzinę, czy dwie, jeśli zawężamy swój obszar myślenia nie zobaczymy niczego nowego, o ile z niego nie wyjdziemy.

Wpis #49 Odpowiednie nastawienie jest istotne

W branży testerskiej istnieje takie powiedzenie, że testerzy zajmują się psuciem oprogramowania i jeszcze im się za to płaci. Taka zachęta od szkoleniowców dla ludzi, którzy chcą się przebranżowić. “Superancko, będę psuć”. Następnie pojawia się opozycja, która twierdzi, że testerzy niczego nie psują. Defekty, na które natrafiają testerzy nie są efektem ich pracy tylko pracy autora kodu, który taki defekt do systemu wprowadził (nie mówię, że celowo). Tester tylko go odnalazł. Więc w sumie faktem jest, że tester niczego nie psuje.

Niemniej tego rodzaju przekonanie, choć być może nieprawdziwe, jest bardzo pomocne. Testując można założyć, że system jest sprawny i pozbawiony błędów, a my tylko weryfikujemy. Możemy też założyć, że system na bank posiada błędy i musimy je znaleźć. Przyjęcie pierwszego, czy drugiego podejścia sprawia, że rezultaty naszego testowania są inne.

W metodach badawczych korzysta się z przypuszczeń i obaleń (conjecture and refutation). Okazuje się, że jeśli nasze badanie oprzemy na tym, aby udowodnić, że nasza hipoteza jest trafna, cały nasz proces będzie oparty na szukaniu dowodów, aby tę hipotezę potwierdzić. Jednocześnie wszystkie argumenty stanowiące przeciwnie, mogą być przez nas ignorowane. Zależy nam na tym, aby udowodnić, że mamy rację, a nie, że jej nie mamy. I odwrotnie, jeśli szukamy dowodów na to, że nasza hipoteza jest nietrafna albo lepiej, hipoteza naszego kolegi, którego np. nie lubimy, zaprzęgniemy nasz umysł w poszukiwaniu jej niedoskonałości, wszelkich luk lub nawet sprzeczności. A wszystko po to, aby utrzeć komuś nosa.

Dokładnie ten sam proces wykonujemy przyjmując rolę testera. Jako tester badamy. Jeśli podejdziemy do testowania zakładając, że wszystko działa, będziemy starali się tego dowieść. Jeśli przyjmiemy założenie, że nasz system nie działa, zaczniemy być sceptyczni i zaczniemy doszukiwać się problemów. O ile możemy być postrzegani negatywnie przez resztę zespołu (sceptycy często tak są postrzegani), o tyle pozwoli nam to wykonać dobrze naszą pracę.

Testując próbujmy udowodnić, że w projekcie znajdują się problemy.

Wpis #50 Czy tester powinien brać się za debugowanie?

Tak zwany gorący temat, z którego zawsze powstaje gównoburza. Mniej więcej to samo, co dyskusja, czy tester powinien umieć programować. Obserwując wiele takich dyskusji zauważyłem, że najczęściej można wyróżnić dwie strony, gdzie jedna jest skrajnie za, a druga skrajnie przeciw. Ci pośrodku siedzą cicho i być może mają fun. Na pewno ani razu nie słyszałem pytania o kontekst, np. ‘a co robi wasz system’ albo ‘jak pracuje wasz zespół’.

Jestem przekonany, że takie dyskusje za i przeciw są bez sensu, bo zawsze można znaleźć jakieś argumenty za i przeciw. Zamiast skupić się na tym, w które argumenty mocniej wierzymy, aby sprowadzić te dyskusje do jednej wspólnej wersji, można zastanowić się, czy korzyści płynące z tego, że np. tester debuguje są większe niż gdyby nie debugował.

Jeśli chodzi o testera, który debuguje wyobraźmy sobie taką sytuację:

- Tester znajduje problem. Musi go przekazać do kogoś, kto będzie ten problem rozwiązywał. Może uderzyć do autora,

żeby ten na szybko wrzucił poprawkę, a może też zrobić zadanko w Jirze i problemem zajmie się albo autor, albo ktoś inny z zespołu. Aby cały proces był efektywny najlepiej byłoby, aby nasz majster miał możliwie najwięcej informacji, żeby szybko ten defekt poprawić. Źródeł tych informacji jest kilka. Na pewno można wyróżnić testera, który przekaże to, co do tej pory udało mu się ustalić. Jest też reprodukcja defektu, gdzie programista sam na własne oczy wykmin,i co się dzieje. Jest również wiedza i doświadczenie, które pozwala programiście wyciągnąć jakieś wnioski.

- Choć źródła są różne niewątpliwie, aby zebrać potrzebne informacje trzeba poświęcić na to trochę czasu. Pytanie, kto zrobi to szybciej? Odpowiadając na to, możemy rozważyć cztery okoliczności:

- Siedzieliśmy już długo nad jakimś obszarem i natrafiliśmy na problem. Znamy kontekst, mamy już jakieś dane testowe, czy narzędzia, mamy nawet pootwierane wszystkie okienka, terminale, logi, czy z czymkolwiek pracowaliśmy. Jeśli po odnalezieniu defektu pozamykamy wszystko i przekażemy problem programiście, to najprawdopodobniej ten będzie musiał wszystko to odtwarzać, co może być czasochłonne. Jeśli programista spędzi np. 60 min na to, aby wejść w kontekst problemu i go zreprodukować tylko po to, aby dalej przejść do debugowania, to może lepiej jak zrobi to tester? O ile jest w stanie oczywiście. Wtedy programista dostanie info o źródle problemu. Znając życie będzie wiedział, gdzie popełnił błąd i od razu zajmie się poprawką. Brzmi spoko.

- Odnajdujemy problem, który okazuje się być wierzchołkiem góry lodowej. Tzn. coś okazało się być zupełnie innym problemem, a to co znaleźliśmy było tylko symptomem. Jeśli przekażemy informacje o tym wierzchołku (w dodatku źle ją sformułujemy) możemy sprawić, że ktoś właśnie na tym się skupi. Będzie rozkminiał do momentu, gdy zorientuje się, że zupełnie nie o to chodziło. Np. symptomem jest to, że obiekt się nie tworzy, gdzie problemem jest źle spięta kolejka gdzieś tam dalej w procesie. Programista zacznie analizować kod tworzenia obiektu w poszukiwaniu problemu. Dopiero potem zorientuje się, że problem leży

w innym miejscu. Oczywiście to, jak on podejdzie do debugowania to inna kwestia, jednak osobiście zdarzyło mi się kilkakrotnie spotkać z taką sytuacją. Gdyby tester poświęcił troszkę więcej czasu na debugowanie jest szansa, że programista nie straciłby swojego czasu tylko przez to, że został zwiedziony złą informacją.

- Są też takie defekty, które są mocno wyrafinowane. Pokazują się tylko w pewnych okolicznościach i gdy tak już się stanie, trzeba łapać chwile. Jak już trafiliśmy na ten problem, fajnie wykorzystać tę sytuację i sprawdzić logi, pogrzebać i dowiedzieć się, dlaczego taki błąd się pojawia.
- Na koniec nie raz zdarza się tak, że błąd jest winą testera. Czasem odpalimy złą wersje, użyjemy złej konfiguracji, coś przeoczymy, coś popsujemy. Zdarza się. Ja sam wciąż pracuje nad tym, aby wyplenić z siebie nawyk zgłaszania problemów jak tylko się pokażą. Staram się doszukać, co się stało, z czego problemy wynikają i czasem faktycznie okazuje się, że to moja wina.

Osobiście jestem za tym, aby tester zabierał się za debugowanie pod warunkiem, że nie przewyższa to jego umiejętności i zrobi to efektywnie. Efektywnie to znaczy, jeśli jego debugowanie pozwoli zaoszczędzić czas programiście ale nie dlatego, że programisty czas jest cenniejszy niż czas testera, więc lepiej niech tester nad tym siedzi (tyle razy to słyszałem, że nie mogę o tym zapomnieć). Efektywnie dlatego, że pozbywamy się zbędnego dodatkowego nakładu pracy.

Wpis #51 Kto to testował?

W tym feedzie chciałem poruszyć temat budowania kultury jakości i otwartości, która jest bardzo istotna pracując w zespołach zwinnych.

'Kto to testował?!'

To pytanie zazwyczaj pojawia się w sytuacji, gdy ktoś odkrył problem w funkcjonalności, która teoretycznie powinna być

pozbawiona defektów. W końcu została przetestowana prawda? Jednak nie możemy spodziewać się znalezienia wszystkich możliwych defektów.

Oprócz szukania winnego sytuacji, która jest nieunikniona, to pytanie wprowadza jeszcze jedną niekorzystną implikację. Rozpoczynamy proces wyplewiania otwartości i siania strachu przed pomyłką. Jest to bardzo bardzo niekorzystne dlatego, że okazje do popełniania błędów to okazje do nauki. Wiadomo, nie chodzi o to, aby celowo wprowadzać błędy. Natomiast jeśli takowe się już zdarzą, to trzeba się na nich uczyć, koniecznie!

Nauka na błędach jest szalenie istotna w przypadku rozwoju testera. A testerem w zespole zwinnym może być każdy. Jako tester teoretyk, mól książkowy, amator kursów i innych form suchej nauki, mogę nauczyć się ogromnej ilości technik testowania. Choćby nie wiem jak obszerne było moje projektowanie testów, 5 przypadków, 10 przypadków, 20 przypadków testowych dla tej samej funkcjonalności, w tym wszystkim nie chodzi o obszerność, a o skuteczność.

Możemy przetestować coś na setki sposobów i wciąż nie znajdziemy problemu, jeśli będziemy szukać go w złym miejscu. Dlatego popełnione błędy są takie istotne. Za każdym razem, gdy jakiś błąd zostaje pominięty i następnie odkryty w późniejszym etapie to dowód na to, że projektując przypadki testowe pominęliśmy jakiś istotny aspekt. Stworzyliśmy 100 przypadków, które nie wykrywają problemu i zapomnieliśmy o 101, który ten problem wykrywa. Tym samym precyzja naszych przypadków testowych była słaba.

Projektowanie przypadków testowych to sztuka kombinowania. Niestety sztuczna sztuka. Jeśli mamy obiekt A, obiekt B, obiekt C i pomiędzy nimi jakieś ścieżki stanów, to klasyczne podejście do projektowania przypadków testowych polega na wyznaczeniu wszystkich możliwych kombinacji i przetestowaniu ich. Potem okazuje się, że to i tak nie działa na Firefoxie. Zamiast projektować więcej, tzw. sztucznych, czy wymuszonych przypadków testowych, trzeba korzystać ze skarbów doświadczenia i szukać błędów tam, gdzie one NAPRAWDĘ występują.

Z kolei, żeby uczyć się na błędach potrzeba nam otwartości. Jeśli ktoś odkrył błąd, a inna osoba go popełni to dojście do błędu, jego genezy i sposobu wykrycia jest mega cenną wiedza, którą wszyscy powinniśmy się dzielić i na pewno nie powinniśmy się bać o tych rzeczach mówić. Samo stwierdzenie ‘nie bójmy się mówić o porażkach’ wiele nie da. Żeby faktycznie wszyscy byli otwarci trzeba budować środowisko otwartości i eliminować to, co te otwartość zakłóca np. cięte uwagi. A robić to powinien cały zespół.

Wpis #52 Prawie działa

Wyobraźmy sobie, że mamy funkcjonalność, która jak zwykle opakowana jest innymi zależnościami. Np. mamy stworzyć obiekt A, aby następnie wykorzystać go w funkcjonalności B. Funkcjonalność B już mamy, w tym sprincie implementujemy tworzenie obiektu A. Jak to przetestować? Stworzę obiekt A i sprawdzę czy istnieje. Jeśli istnieje, to świetnie, mamy prawie działającą funkcjonalność. Ale czemu prawie?

Problemem w tej historii jest testowanie implementacji, zamiast testowanie wartości biznesowej. I to oczywiście jest kolejnym grzechem na liście testera. Po co nam obiekt A, jeśli nie możemy go dalej wykorzystać? Naturalnie, odpowiedź nasuwa się prosta. Taki obiekt jest nam niepotrzebny. Potrzebujemy obiektu, który będzie można następnie wykorzystać.

Aha! Niby proste. Jednak nawracającym problemem jest fakt, że często każdemu z nas, mnie też, zdarza się skupiać na implementacji samej w sobie i zapominamy o przetestowaniu wartości biznesowej. To tak jakbyśmy się zatrzymali przed drzwiami i zapomnieli sprawdzić, czy się otwierają i pozwalają nam wejść do mieszkania.

Zdarza się tak, ponieważ nie zawsze łatwo jest dostrzec zależności znajdujące się na końcu procesu. Nie zawsze taki błąd da nam liścia w twarz. Czasem pojawi się on dopiero gdy przejdziemy długi proces, żeby na samym jego końcu dowiedzieć się, że Obiektowi A brakuje jakiegoś atrybutu, który

jest niezbędny, aby proces ten sfinalizować na samym jego końcu. Często też dostrzegamy je tylko wtedy, gdy uruchomimy myślenie systemowe, czy dogłębnie przeanalizujemy cały proces. Dlatego można ten problem rozkminiać bardziej jako pułapkę niż niedbałość.

Jeśli więc wiemy, że w taką pułapkę można wpadać i jeśli po zastanowieniu się stwierdzimy, że my też faktycznie w nią wpadamy to nie pozostaje nam nic innego, jak powzięcie środków zapobiegawczych. Jakaś checklista, przypominajka, heurystyka, czy cokolwiek będzie nam służyć. Grunt to już wiedzieć, że mamy taki problem.

Wpis #53 Jeśli nie możesz przodem, próbuj tyłem do przodu

Ostatnio było o inputach, dziś będzie o outputach. Pamiętacie z jakiego filmu jest ten cytat? Podpowiem, że to od Bogusia, Bogusia Lindy.

Jeśli jest input, to zwykle jest też output, prawda? Zwykle (a być może zawsze?), gdy implementujemy kod, to po to, aby coś przetworzył. Output jest wynikiem tego przetworzenia, a input jest inicjatorem.

Myśląc o projektowaniu przypadków testowych, możemy dywersyfikować input, aby sprawdzić jak nasza implementacja będzie się zachowywać. Jednak dywersyfikacja wyniku końcowego naszej implementacji może być równie owocna.

To co konkretnie mamy zrobić? Zerknijmy na to, jaki output wypluwa nasz kod. Zróby dokładnie to samo, co przy inpucie, tzn. zastanówmy się z czego ten output się składa, jaki ma format, jaki ma rozmiar, czy długość. Czy istnieje w nim jakakolwiek mnogość, zakres i finalnie walidacja, czyli kiedy nasz output jest niepoprawny?

Jak już zidentyfikujemy wszystkie atrybuty i zajmiemy się ich dywersyfikacją (różne rozmiary, zakresy czy niepoprawne jego wersje) zastanówmy się, co musimy zrobić, aby taki output osiągnąć. Wiadomo, zwykle systemy są wypchane różnymi walidatorami, serializatorami itd., co pozwala nam odfiltrowywać wszystko, co niepoprawne. Jednak nie zawsze wszystko przewidzimy, więc takich błędów szukamy.

Co prawda nie chodzi tutaj o to, aby sztucznie wstrzykiwać niepoprawne dane gdzieś w środku procesu, żeby udowodnić, że się da. Tutaj bardziej interesuje nas wywoływanie pożądanych output'ów poprzez interakcje z interfejsem graficznym, api naszego backendu, środowiskiem oraz zewnętrznymi serwisami i źródłami danych. Chodzi o to, aby pomóc sobie znaleźć błędy, które mogą się pojawić, gdy system pracuje w naturalny dla siebie sposób. Ewentualnie znaleźć podatności, które mogą zostać wykorzystane poprzez niepożądane zewnętrzne interakcje z systemem.

Wpis #54 W poszukiwaniu prawdy

W dzisiejszym feedzie poruszę temat krytycznego QA. Krytycznego w sensie potrafiącego myśleć krytycznie - niekoniecznie mającego tendencję do wyrażania negatywnych ocen. Oczywiście to co napiszę równie dobrze można przełożyć na wszystkie inne role.

Myślenie krytyczne jest jednym z wielu sposobów postrzegania rzeczywistości wykorzystywanych w pracy projektowej. Myślenie analityczne, logiczne, systemowe, czy właśnie krytyczne to rodzaj sięgania po informacje, jakie kryją się w tej rzeczywistości. Nie bez powodu mówię, że te informacje się kryją, bo właśnie tak można to nazwać. To znaczy, że takie informacje nie są łatwo dostępne, czy dostrzegalne z uwagi na nasze przyzwyczajenia, uprzedzenia, emocje, bazowanie na danych historycznych i

wiele wiele innych cech natury ludzkiej, które wpływają na postrzeganie rzeczywistości.

Oczywiście po przeczytaniu jednego feed'a nie możemy spodziewać się rewolucji. Nie zaczniemy nagle lepiej myśleć krytycznie. Niemniej według mnie zrozumienie, na czym to dokładnie polega i co utrudnia nam pozyskiwanie informacji pomoże nam zrobić pierwszy krok.

Wobec tego mówiąc o myśleniu krytycznym odnoszę się do umiejętności, która pozwala nam dostrzegać fakty, jednocześnie skutecznie omijając założenia, hipotezy, czy opinie. W tym momencie, odrobinkę rozważań filozoficznych:

W myśleniu krytycznym chodzi o dążenie do odnalezienia prawdy. Prawda to jest nic innego jak fakt. Ponieważ świat zbudowany jest na faktach, opinie formują ludzie. Dobrze jest umieć dostrzegać różnice pomiędzy faktami i ... nie-faktami.

Tzn. spierając się, czy skoda jest fajna, czy nie, nie używamy faktów, bo fajność nie jest faktem tylko subiektywnym odczuciem bądź opinią. Możemy używać faktów, np. koloru, aby argumentować swoje stanowisko. Wynikiem takiej dyskusji jednak nigdy nie będzie prawda.

Spierając się, czy skoda jest awaryjna jesteśmy bliżej szukaniu prawdy, ponieważ stopień awaryjności może być prawdą. Ta prawda jednak może mieć różne oblicze w zależności od wyniku dyskusji. Czasem będzie on zależeć od ilości argumentów 'za', jakie udało nam się zebrać, w porównaniu do argumentów 'przeciw'. Czasem zostaniemy przegadani. Istnieją też dyskusje, które nie koniecznie bazują na prawdzie (faktach) ale tej prawdy szukają. To znaczy, że nie ma co spierać się o temperaturę na dworze w oparciu o subiektywne odczucie, lepiej sprawdzić termometr (odnaleźć prawdę).

Dlaczego to jest takie istotne? Dyskutując bardzo trudno jest opierać się wyłącznie na faktach i odpuścić wszelkie uprzedzenia, opinie, emocje, relacje międzyludzkie. Jeśli prowadzimy dyskusję przy kawie, to jej wynik być może nie będzie tak istotny, jak wynik spotkania projektowego, od którego będzie sporo zależało. (no chyba, że przy kawie omawiamy ważne sprawy).

Będąc na spotkaniu musimy uważać, czy nie formujemy opinii tylko dlatego, że ktoś jest charyzmatyczny, ma wyższy status (tech lead), rządzą nami emocje itd. W dodatku bardzo łatwo jest samemu wpaść w pułapkę, gdzie zaczynamy mylić fakty z opiniami. Zarówno tymi wyrażanymi przez innych, jak i nas samych. W tym momencie przypominają mi się słowa kolegi, który czasem zwracał uwagę w trakcie trudnej dyskusji, że rozmówca postrzegał swoje opinie jako fakty. Z takimi ludźmi lub w takiej sytuacji dyskutować jest bardzo trudno.

Gdy zaczynamy opierać się na faktach trzeba obserwować argumentacje. Bardzo łatwo jest zdominować postrzeganie rzeczywistości przedstawiając tylko argumenty pozytywne. Tak robią media, marketing no i my - cedzimy informacje, żeby zmanipulować rzeczywistość.

Gdy nasze dyskusje zaczynają być mocno oparte na faktach, tutaj też czyhają pułapki. Skąd wiemy, że coś jest faktem? A może to opinia? A może dochodzenie do faktu było obarczone błędem?

Spotkania to tylko przykład. Myślenie krytyczne, czyli posługiwanie się faktami sprzyja większości tego co robimy.

Wpis #55 Emerging dependencies

Znowu po angielsku, bo po polsku to zależności powstające/kształtujące się/nowo objawione. Nie pisałem nic, co by poruszało temat samej praktyki testowania. Wobec tego w tym feedzie będzie o zależnościach, które pojawiają się dopiero w konkretnym kontekście. Chciałem tylko na początku przypomnieć technikę testowania używaną przy projektowaniu przypadków testowych. Chodzi o identyfikowanie elementów i ich różnicowanie. Tzn. podchodząc do testowania identyfikujemy obiekty i akcje, a następnie dywersyfikujemy je w różnych wymiarach: ilość, czas, stan, relacje etc.

Zależności te pojawiają się tylko w pewnych okolicznościach. To znaczy, że aby się pojawiły musimy im w tym pomóc. Można je znaleźć praktycznie we wszystkich obszarach i na różnych poziomach, tzn. zarówno w aplikacji, jak i w infrastrukturze. Bądź zarówno na poziomie systemowym, jak i na poziomie jednego modułu.

Testując moduł 'a' oraz moduł 'b' z osobna, oba będą działać świetnie. Oba te moduły mogą być używane z osobna i oba będą spełniać swoje funkcje biznesowe. Jeśli jednak proces biznesowy przewiduje również korzystanie z obu w tym momencie, mogą pojawić się relacje, które sprawią, że mamy więcej case'ów do przetestowania.

Dla uproszczenia taki przykład: wybierając datę: wybieramy **dzień** 1-31, testujemy również negatywnie wybierając 0, bądź 32 itd. Pole `dzień` w swoim kontekście działa. Potem testujemy pole **miesiąc**. Znów 1-12, oraz negatywnie 0, 13 itd. Wszystko również działa.

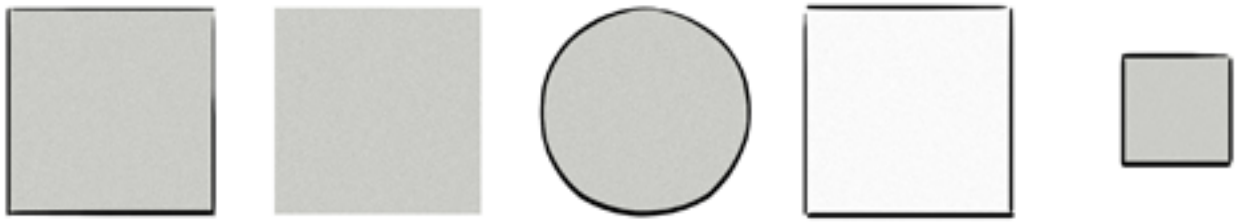
Następnie w grę wchodzi zależność, która wprowadza pewne zmiany w walidacji. W momencie, gdy próbujemy wybrać 31 kwietnia - to nie zadziała. Nie zadziała również 30 lutego. 29 lutego też nie zadziała. Zadziała dopiero biorąc pod uwagę kolejną zależność czyli **rok**. Jak widać skuteczność naszego testowania jest zależna od tego, jak szeroko będziemy testować.

Przykład z datą jest bardzo prosty i wybrałem go celowo. Chciałem jasno pokazać na czym polega problem, a wiadomo z naszymi systemami nie będzie tak łatwo. W przypadku systemów, z którymi my pracujemy, zwykle te relacje będą bardziej wyrafinowane i oczywiście bardziej oddalone od siebie.

Niemniej jeśli poświęcimy trochę czasu na analizę, jesteśmy w stanie je zidentyfikować. Uwaga - taka analiza przydaje się zarówno przy testowaniu, jak i przed implementacją. Im lepiej przemyślimy sobie, co będziemy implementować, tym więcej problemów zabezpieczymy. Pamiętajmy, że system to nie tylko całość całości. Dwa, czy trzy elementy systemu tworząc podsystem same w sobie też są systemem. Pamiętajmy o myśleniu systemowym na różnych jego poziomach.

Wpis #56 Modelowanie w praktyce - ćwiczenie

Powyżej widać pięć figur. Naszym zadaniem jest określić, która z nich nie pasuje do reszty.



Znów daje szansę abyście mogli to zrobić sami.

Udało się? Jeszcze nie? Próbuj dalej

Czy na pewno chcesz już zobaczyć odpowiedź?

Mam nadzieję, że próbowałeś/aś.

Ok no to rozwiązanie i analiza.

Na pierwszy rzut oka - trudne zadanie. Część z nas, gdy była małymi dziećmi pewnie spotykała się z takimi zagadkami. Ale autorzy zagadek kierowanych do dzieci byli na tyle mili, że wraz z zagadką dostarczali nam również wyrocznie (z ang. oracle.)

Krótko o tym, czym jest wyrocznia, a szczególnie w kontekście testowania, wyrocznia testowa.

Wyrocznia to nasze odniesienie do tego, co jest prawdą/faktem/oczekiwaniem. Wyrocznia jest nam potrzebna, aby porównując rzeczywistość, wykorzystać ją do stwierdzenia jak bardzo odbiega ona od oczekiwań/prawdy/faktu. Krótko mówiąc, skąd wiemy, że błąd to błąd? Wiemy to, ponieważ porównujemy dane zachowanie z modelem (który np. utworzyliśmy sobie w naszej głowie) mówiącym, jak powinno wyglądać prawidłowe zachowanie danej funkcjonalności.

Jeśli chodzi o to zadanie, tutaj wyrocznia nie została podana. Pierwszym krokiem jest jej odnalezienie. Analizując wszystkie te figury można dojść do wniosku, że prezentują one pewne atrybuty:

- obramowanie
- kształt
- rozmiar
- Kolor

Mając atrybuty możemy przejść do opisania każdej figury tymi atrybutami. Skorzystajmy sobie z matrycy:

	Figura 1	Figura 2	Figura 3	Figura 4	Figura 5
Obramowanie	Czarne	Brak	Czarne	Czarne	Czarne
Kształt	Kwadrat	Kwadrat	Koło	Kwadrat	Kwadrat
Rozmiar	Duży	Duży	Duży	Duży	Mały
Kolor	Szary	Szary	Biały	Szary	Szary

Teraz, do analizy!

Patrząc na wszystkie figury i ich atrybuty można zauważyć pewne anomalie. Na poziomie atrybutowym w każdym atrybucie

istnieje dokładnie jedna figura, która odstaje od reszty. Czy możemy na podstawie atrybutów wyróżnić któryś z nich? Nie. Wszystkie, bez wyjątku mają jedną odstającą figurę.

W tym momencie pozwolę przemycić sobie kolejną dygresję. Patrząc na wszystkie figury można było zobaczyć, że są różne i coś się z nimi dzieje. Po zamodelowaniu ukazał nam się pewien wzór, który jasno pokazał, że te różnice to anomalie. Polecam wszystkim modelowanie problemów. Wiele razy pomogło mi to dostrzec pewne rzeczy w trakcie testowania.

To co teraz? Zmieńmy wymiar! Przejdźmy do wymiaru figur.

Patrząc nie z perspektywy atrybutów i ich figur, a odwrotnie, czyli z perspektywy figur i ich atrybutów można zauważyć, że każda z figur posiada jedną anomalie. Każda prócz jednej!

Okazuje się, że pierwsza figura nie posiada żadnych anomalii. Można by rzec, że figura idealna. Pierwsza, taka którą zapewne wzięliśmy za wzór do porównywania reszty, okazuje się być wystającym gwoździem. Doskonała wśród przeciętniaków. Żartuje sobie troszkę ale to jest kolejny dowód na pewien błąd myślowy. Bardzo często jesteśmy ofiarami błędnych założeń i wrażeń, które odbieramy. Potem po głębszej analizie okazuje się, że świat wygląda troszkę inaczej niż nam się wydawało.

Ponoć z psychologicznego punktu widzenia większość osób (ja też) zaczyna analizować figury wybierając pierwszą jako odniesienie. Dzieje się tak dlatego, że podświadomie uznaliśmy ją za wzór. Tak już mamy. Trzeba uważać na te pułapki, bo przy testowaniu robimy te same błędy. A potem defekty nam umykają. Testerzy z jednej strony muszą kierować się doświadczeniem i intuicją ale z drugiej muszą bardzo uważać na to, aby nie wpadać w pułapki swoich własnych założeń.

I jeszcze raz, modelowanie jest spoko!

Wpis #57 Pułapki założeń w praktyce, for dummies

Używanie założeń może nam wyrządzić dużo złego. Tzn. założenie samo w sobie nie jest takie złe pod warunkiem, że je zweryfikujemy i stanie się faktem. W przeciwnym wypadku może nam sporo napsuć i w tym feedzie chciałem podzielić się historią z wczoraj, gdzie padłem ofiarą błędnego założenia i miałem z tego powodu niezłą zadymę w głowie.

Wczoraj miałem do czynienia z testowaniem danych i job'a do transformacji danych. Wyzwaniem tego taska była ilość danych w tabeli, miliony danych i setki kolumn. Pracujemy z AWS'em. W celu ułatwienia sobie życia wyciągnąłem z bazy danych 10k rzędów (limit 10000). Athena zapisała mi to jako plik .csv do bucket'u wynikowego, a ja sobie ten plik przekopiowałem do swojego bucket'a. Miałem potem z tego zrobić tabele używając crawler'a AWS Glue, a następnie te tabele podać jako parametr do job'a, którego chciałem przetestować. Job miał sobie tabele wczytać, coś tam zrobić i wypluć nową tabele.

Dobra, po szarpaninie z AWS'em i uprawnieniami w końcu mi się to udało. Uruchomiłem job'a, a ten się wysypał z jakimś nieoczekiwanym błędem - brakowało mu jakby kolumny. Zacząłem więc debugować ten problem korzystając z notebook'a AWS Sagemaker. Przekopiowałem skrypt z job'a i wkleiłem do notebook'a, żeby ręcznie to odpalić i zobaczyć w którym konkretnie momencie się to wysypuje.

Skrypt job'a miał sekcję w kodzie, która filtrowała dane. Filtrowanie polegało na tym, aby z milionów danych po pierwsze usunąć dane, które mają null'e w konkretnej kolumnie oraz aby wyciągnąć z tych danych tylko te, których kolumna event miała dwie określone wartości. Reszta event'ów była nam nie potrzebna. Jak się okazało w tym konkretnym momencie po filtrowaniu dostawałem pusty DF (dynamic frame - tabela w glue).

Skoro dostawałem pusty DF to stwierdziłem, że upewnię się, czy te 10k testowych danych w ogóle ma jakieś rekordy, które pasują do warunków filtrowania. Użyłem do tego Atheny i okazało się, że faktycznie pusto. Mówię "shit", na 10k profili nie ma ani jednego pasującego. Trochę to niewiarygodne, no ale ...

Powoli do sedna problemu. W dalszej kolejności wyciągnąłem nowe dane testowe. Tym razem upewniłem się, że są w nich dane, które powinny znaleźć się po filtrowaniu. Udało się. Dostałem plik .csv i załadowałem go do tabeli. Jako prawilny tester mówię do siebie tak: "zanim puszczę job'a na tych danych upewnię się, że w tej tabeli faktycznie jest to czego potrzebuję".

Zrobiłem SELECT ... WHERE na tych danych, no i ...nic. Pusto. Z jednej strony, ucieszyłem się, że sprawdziłem, bo tak znowu bym się kręcił w kółko. Jednak czemu tam nie ma tych danych? Chwile pogłówkowałem. Okazało się, że te dane są. Były zapisane jako stringi i zczytanie danych z pliku .csv dodało do każdej wartości typu string dodatkowe cudzysłowie.

W tym momencie popełniłem fuckup, mówię: "Aha! Więc to tutaj pewnie był problem". Wiedziałem, że to niemożliwe, żeby moje 10k poprzednich danych testowych nie miało ani jednego rekordu, którego potrzebuje. Wiedziałem, że to na pewno problem tego dodatkowego cudzysłowu i dlatego job się wysypał. Szukał wartości bez cudzysłowu, a ja po prostu wcześniej na to nie wpadłem.

Zamiast sprawdzić, czy job w ogóle sobie z tym poradzi założyłem, że to właśnie jest problemem i w dalszej kolejności próbowałem wyczytać dlaczego tak się dzieje, jak to naprawić w AWS'ie, konsultowałem się też z zespołem.

Wiecie co się okazało? Że to nie ma znaczenia dla job'a. Że on sobie z tym i tak radzi. Że moje pierwsze 10k danych NAPRAWDĘ nie miało żadnego rekordu, którego potrzebowałem, a zrobienie SELECT ... WHERE, po którym okazało się, że nie dostaję żadnych wyników o ile nie dodam dodatkowego cudzysłowu do wartości, NIE MA W OGÓLE ZNACZENIA w tym całym cyrku.

Nie wiem, czy po tym, co napisałem jesteście w stanie sobie wyobrazić tę sytuację tak, jak wyglądała ona u mnie w głowie. Jed-

nak pod koniec dnia miałem totalny mindfuck, bo prócz tego, że błądziłem w tym wszystkim, to jeszcze po drodze miałem dużo szarpania się z AWS'em, który też nie ułatwiał wykonywania operacji płynnie. Być może te postoje na rozwiązywanie problemów AWS'owych też przyczyniły się do tego, że nie myślałem jasno.

Nieważne. Co jest istotne okazuje się, że nawet jeśli jestem świadomy takich pułapek, często staram się ich unikać, to i tak przyjdzie moment, gdzie dam się nabrać.

W pierwszym akapicie napisałem, że padłem ofiarą. Bullshit. Nie padłem ofiarą, tylko zamuliłem - moja wina i muszę zdawać sobie z tego sprawę. Fakt, wina moja ale nie było to też nic unikalnego. Takie sytuacje są normalne, więc nie oskarżam się za to za bardzo. Chciałbym jednak jeszcze mocniej się na nie wyczulić. Pewnie PESEL mnie tego nauczy lepiej, niż bycie tylko świadomym.

Część IV

Agile z punktu widzenia QA

Wpis #58 Definition of Ready

Definicja gotowości to taki element projektu, który niby jest, ale go nie ma. To znaczy, że czasem zespół zreflektuje się, żeby stworzyć sobie DoR, mając na uwadze podążanie za dobrymi praktykami zwinnymi i oczywiście dobrem zespołu, ale finalnie taki dokument zostaje odłożony na półkę w celu dojrzewania. Za każdym razem, gdy wspominam o sytuacjach, w których zespół, czy poszczególni jego członkowie robią coś, co wydaje się być niekorzystne, czy bezowocne nigdy nie podchodzę do tego roszczeniowo o ile widzę, że wszyscy równo staramy się dowozić najlepiej jak potrafimy. To znaczy, że (ostatnio zasłyszane) nie bimbamy sobie, tylko pracujemy. Wychodzę wtedy z założenia, że jeśli nie podążamy za jakimiś praktykami, to być może nie wiemy jaką wartość tę praktyki wnoszą. A może mamy z nimi złe doświadczenia, widzimy, że w sumie to szkoda czasu, bo i tak nic to nie zmienia. To nie znaczy, że praktyki są złe. Być może ich implementacja nie jest najlepsza albo nie widzimy korelacji między problemami, a praktykami.

Definition of Ready można nazwać narzędziem procesowym. Jest to forma checklisty, która ma potwierdzić, że zadanie jest gotowe, aby móc je przenieść z etapu zbierania wymagań (lub dojrzewania w kolejce) do sprintu i zacząć nad nim pracę. To teraz pytanie. Ile razy w trakcie trwania sprintu okazało się, że blokują nas jakieś zależności, brakuje nam wiedzy, źle zrozumieliśmy zadanie, czy nie wyrabiamy się z implementacją? Często się zdarza. Równie często winą można obarczyć problemy związane z DoR.

Życie pokazuje, że praca nad wielkimi zadaniami jest nieefektywna pod wieloma aspektami i granulacja `divide and conquer` to sposób na większość naszych problemów. Czasem na spotkaniu planującym sprint mamy jednocześnie przejrzeć sporą ilość zadań, przeanalizować, przedyskutować i zrozumieć. Potem jeszcze wybrać te z nich, które damy radę zrealizować w najbliższym sprincie. Wybrać musimy również zadania, które pozwolą uniknąć blokujących zależności, oraz te, które będą

sprzyjać efektywnej kolejności budowania systemu. Naturalnie, wówczas szybko się zmęczymy i zaczniemy skracać czas na pewnych czynnościach (oraz przytakiwać) mimo, że do końca wszystkiego nie rozumiemy. Zasiadając więc do spotkania powinniśmy mieć większość z powyższych rzeczy za sobą, aby móc się skupić na efektywnym planowaniu. W tym właśnie ma nam pomagać wykonywanie czynności zdefiniowanych w Definition of Ready.

Jednak, według mnie, najgorszą możliwą interpretacją DoR jest postrzeganie jej jako listy punktów, które mamy zrobić przed estymacją, czy planowaniem. Skopiujemy taką listę od kogoś albo z internetu i mamy. Czasem sobie o nich przypominamy, czasem nie. Czasem nawet staramy się za nimi podążać ale to zupełnie nie o to chodzi. Nie chodzi o to, żeby podążać za checklistą, bo sam fakt, że mamy przeanalizować zadanie nie gwarantuje, że zrobimy to dobrze. Będziemy to robić dobrze pod warunkiem, że zdamy sobie sprawę jak nasza analiza wpłynie potem na przebieg iteracji. Inaczej mówiąc, jeśli zrobimy to na odwal się to jest duża szansa, że wpakujemy cały zespół w problemy i opóźnienia. Co z tego, że podążamy za DoR, skoro nie przynosi to żadnych efektów? Jeszcze raz, kluczem nie jest podążanie za DoR, a rozumienie istoty DoR.

Brak lub niestaranne wykonanie elementów z DoR np. wykonanie opisu zadania lub analizy zależności powoduje, że nie posiadamy pełnego obrazu tego, co będziemy robić. Stąd nasze estymacje mogą być nietrafione, planowanie może być częściowo losowe, a iteracje skazane na niespodzianki. Takie niespodzianki mogą kończyć się błędami, opóźnieniami (uwaga: spowolniona praca to jednocześnie pośpiech, który też często kończy się błędami) albo nawet zmianami architektury, czy narzędzi.

Jednak samo przekonywanie zwykle niewiele daje. Stąd za każdym razem, gdy sprint zaczyna się sypać warto zastanowić się, czy czasem nie jest to wynikiem niestarannego (lub braku) stosowania się do DoR (np. na retrospekcji). Im częściej dojdziemy do wniosku, że tak jest, istnieje szansa, że zaczniemy bardziej przykładać się do analizy zadań, zamiast tylko skupiać się nad ich implementacją.

Wpis #59 Acceptance Criteria

Zaraz po tym, jak klient wyraził swoją potrzebę biznesową, która w następnej kolejności trafiła do naszego zespołu, rozpoczął się cykl życia tej potrzeby. Cykl życia, ponieważ od tego momentu wspomniana potrzeba biznesowa przejdzie kilka etapów, zanim klient otrzyma jej (bardziej lub mniej) namacalną wersję. Acceptance Criteria przydają się na wielu płaszczyznach pracy z taką potrzebą i ułatwiają tzw. dowożenie.

AC to nic innego, jak zestaw spisanych oczekiwań względem tej potrzeby. Mogą być spisane w różnej formie np. GivenWhenThen, bądź jako lista oczekiwań wyrażona z perspektywy użytkownika. Jest kilka zasad jak pisać dobre AC. Jedną z nich jest np. fakt, że powinny prezentować etap, w którym funkcjonalność już została zaimplementowana, a my opisujemy to, co widzimy i możemy wykonać. Ja jednak chciałem się skupić nie na tym jak, a dlaczego pisać AC.

Zapewniamy jakość jeszcze przed implementacją. ACki piszemy na możliwie wczesnym etapie życia zadania (TDD na poziomie zbierania wymagań tzw. ATDD). Stąd jest to świetny moment, aby zacząć myśleć o tym, co my tak naprawdę będziemy robili i w jaki sposób możemy to zrobić. Choć dobrą praktyką jest nieuwzględnianie sposobu implementacji w kryteriach akceptacji, to analizując potrzebę szczegółowo chcąc nie chcąc od razu zastanawiamy się nad możliwą implementacją. To dobrze, jeśli włączamy myślenie. Myśląc, jak realizacja potrzeby będzie wyglądać od początku do końca, możemy identyfikować dwuznaczności, czy luki w wymaganiach i od razu je adresować. Ponadto, wymagania w takiej formie mogą bardzo dobrze sprawdzać się w komunikacji z klientem. Pozwalają w łatwy sposób sprawdzić, czy się dogadaliśmy.

ACki pomagają nam też lepiej estymować, jeśli dają nam sporo wiedzy o potrzebie biznesowej. Z dokładnie tego samego powodu pomagają nam odnajdywać problemy, zależności, czy ryzyka, co znacznie ułatwia nam planowanie. Z kolei w trakcie

implementacji ACki stanowią mapę, którą możemy podążać, aby trafić do celu i nie zboczyć z trasy. Jeśli ACki są na tyle szczegółowe, że prezentują dodatkowe sytuacje, na które trzeba uważać (zwykle identyfikowane w trakcie testowania) to sprawi, że nie tylko dojdziemy do celu ale zrobimy to możliwie najbardziej optymalną drogą. Minimalizujemy ryzyko powstawania błędów i szczęśliwy jest zarówno tester, który nie musi zgłaszać błędów, jak i autor, który już nie musi wracać do danego zadania.

Dalej, ACki wspierają weryfikację implementacji, którą można potraktować jako checklistę (pod warunkiem, że przyłożyliśmy się do tworzenia kryteriów). Są dobrą bazą do tworzenia zautomatyzowanych testów regresji i dużym wsparciem przy testowaniu eksploracyjnym stanowiąc pewnego rodzaju fundament, na którym można budować dalsze testowanie.

Na sam koniec, gdy mamy problem z organizacją wymagań i ich dokumentacją, czasem spisywanie przejrzystych kryteriów akceptacyjnych wystarczy, aby uznać system za udokumentowany, przynajmniej pod kątem funkcjonalnym.

Jedno narzędzie, a tyle korzyści. Pisanie kryteriów akceptacyjnych uważam za jedną z najbardziej korzystnych i opłacalnych praktyk. Prócz tych wszystkich korzyści, które wnosi, okazuje się, że koszty są nikłe. Faktem jest, że nie da się implementować nie mając pojęcia o tym, co implementujemy. Etap analizy zadania i tak trzeba w którymś momencie wykonać. Jednak wykonywanie go w trakcie implementacji zwykle przynosi sporo problematycznych niespodzianek. I to nie jest tak, że ktoś z lenistwa nie opisze zadania i wszystko spoko, on sobie to rozkmini po swojemu. Każdy członek zespołu, który w jakikolwiek sposób ma do czynienia z zadaniem, traci czas wykonując mniej więcej tę samą analizę. Narażony jest też na pomyłki wynikające ze swojej interpretacji, a jeśli nie jest w stanie tego zrobić, potrzebuje dodatkowej komunikacji.

Wpis #60 Definition of Done

Definition of Done to jeszcze jedno narzędzie znane większości, jak nie wszystkim. W tym przypadku wspieramy kontrolę jakości bardziej, niż jakość wbudowaną.

Definition of Done podobnie jak Definition of Ready może mieć różny format, ale zwykle składa się z wypunktowanych elementów, które należy wykonać, aby uznać zadanie za skończone. Np. zadanie uważa się za zakończone, jeśli zostało zintegrowane z masterem lub/i przeszło weryfikację acceptance criteria.

Na co to komu?

Człowiek z reguły jest zapominalski. Niech nikt nie próbuje się bronić. Wraz z liczbą rzeczy, o których trzeba pamiętać rośnie liczba rzeczy, o których możemy zapomnieć. Właśnie z tego powodu korzystamy z kalendarzy, przypominajek i list ‘to do’.

Definition of Done jest niczym innym, jak profesjonalnie brzmiącą listą przypominajek. Posiadanie DoD pozwala na conajmniej dwie rzeczy, o których warto wspomnieć:

- Po pierwsze pozwala **wspólnie** z zespołem ustalić ile jesteśmy w stanie poświęcić czasu na kontrolę jakości i podzielić się dobrymi praktykami. Wytłuściłem słowo “wspólnie”, ponieważ kluczowym jest, aby każdy miał wpływ na decyzje o zakresie DoD, miał świadomość jakie rzeczy się na takiej liście znajdują i rozumiał, dlaczego warto realizować każdą z tych rzeczy. Jeśli świadomość o DoD w zespole kończy się na tym, że ponoć gdzieś tam jest, to wiele z niej nie wyciśniemy.
- Po drugie pozwala odciążyć naszą pamięć i jednocześnie zapewnia coś, co po ang. brzmi **mistake-proofing** (znów nie mam fajnego odpowiednika). To znaczy, że nawet jeśli mamy najgorszy dzień ever, o ile wykonamy wszystko z tej listy, będziemy pewni, że o niczym nie zapomnieliśmy. Dlatego

w DoD powinny znajdować się w szczególności takie elementy, które pozwolą na kontrolę jakości nie wymagającą zbytnio myślenia np. wcześniej wspomniane domergowanie mastera (integracja), czy odpalenie testów jednostkowych.

Ponadto, jeśli wiemy, że zespół ma ustalone dobre praktyki i regularnie ich przestrzega, nabieramy zaufania do jakości produktu i tym samym nie poświęcamy czasu żeby na własne oczy zobaczyć, że wszystko jest OK. Osobiście podążanie za DoD lub nawet checklistami w trakcie testowania, niejednokrotnie przypomniało mi o czymś, co okazało się być istotne. Staram się realnie patrzeć na swoje możliwości i zdaje sobie sprawę z tego, że mam tendencje do zapominania. Zamiast więc na siłę próbować o wszystkim pamiętać, regularnie zapisuje rzeczy, o których powinienem pamiętać lub, o których zwykle zapominam i uczę się zaglądać do tych notatek regularnie. To ostatnie jest też bardzo ważne. Same checklisty to połowa sukcesu. Drugą połową jest nauczyć się z nimi pracować, czyli regularnie uzupełniać i korzystać z nich. To już wymaga dyscypliny, ale naprawdę warto.

Pewnie fani Kent'a Beck'a już to słyszeli, nieraz więc tylko zacytuję w formie przypominajki: "I'm not a great programmer; I'm just a good programmer with great habits."

Wpis #61 Czego unikać tworząc Acceptance Criteria - część pierwsza

Ponieważ kryteria akceptacji są bardzo ciekawym narzędziem, które sprawdza się na wielu płaszczyznach, pomyślałem że skupię się na nim przez kilka dni. Chciałem rozszerzyć co nieco informacje na ten temat i zahaczyć o kilka powszechnych błędów w pracy z nimi.

Na pierwszy ogień pójdzie błąd, który mnie osobiście najczęściej rzuca się w oczy (to nie znaczy, że sam go nigdy nie powtarzam). Ostatnio nawet miałem okazję porozmawiać o tym z koleżanką z

pracy, i dlatego pomyślałem, że warto o tym napisać. Dodatkowo problem wynika z perspektywy (albo jej braku), o czym już pisałem, więc jest to nawet fajne płynne przejście między tematami.

Co oznacza zła lub brak perspektywy w tworzeniu Acceptance Criteria? Ponieważ Acceptance Criteria bardzo często przyjmują formę wypunktowanych warunków, niestety sprzyja to myśleniu w dokładnie ten sam sposób. Zamiast skupiać się na systemowym postrzeganiu funkcjonalności, której klient sobie życzy skupiamy się na wylistowaniu wymagań, które od klienta usłyszymy. W trakcie rozmowy wyłapujemy co się da, staramy się to wypunktować. Z jednej strony spoko, bo ACki zawierają dokładnie to, czego sobie życzy klient. Z drugiej strony nie spoko, bo przestajemy zwracać uwagę (albo robimy to w mniejszym stopniu) na to, jak będzie zachowywać się dana funkcjonalność, jak będziemy z niej korzystać, czemu ma dokładnie służyć i uciekamy od całego flow end-to-end.

Ja do tej pory spotkałem się z dwoma konsekwencjami takiej pracy z AC.

Albo ktoś zrobi dodatkową analizę i znajdzie luki, w których wymagania zwyczajnie się nie spinają w całość. Wtedy okazuje się, że trzeba jeszcze dorobić jedno, czy dwa zadania.

Albo ktoś takiej analizy nie wykona do momentu implementacji. Wtedy finał jest taki sam. Trzeba wykonać jedno, czy dwa zadania więcej. Im później się zorientujemy o takiej potrzebie, tym gorzej dla nas.

- Jeśli zrobimy to przed implementacją możliwe, że trzeba będzie przeorganizować sprint albo delikatnie się z nim spóźnić.
- Jeśli zrobimy to w trakcie implementacji, wtedy musimy zostawić pracę rozgrzebaną i wrócić do punktu pierwszego.

Mając na uwadze lean całość zwykle wiąże się z opóźnieniami, dodatkowymi dyskusjami, reorganizacją, niepotrzebnym dodatkowym narzutem na prostowanie spraw, czasem oczekiwania na odpowiedzi, czy skakaniem pomiędzy

kontekstami. Można samemu zastanowić się, w jakim stopniu utrudnia i wydłuża to pracę w sprincie.

Jeśli w trakcie sprintu natrafiamy na takie luki, być może właśnie problem leży w listowaniu wymagań zamiast ich zrozumieniu. Warto zastanowić się nad tym na retrospektywi.

Wpis #62 Czego unikać tworząc Acceptance Criteria - część druga

Czy pracując z wymaganiami zdarza nam się dyskutować na temat danej funkcjonalności? Jeśli tak, to dobrze. Postępujemy zgodnie z założeniami agile'owymi, których punkt nr 6 brzmi:

"The most efficient and effective method of conveying information to and within a development team is face-to-face conversation".

Jeśli nie, to niedobrze. Stosowanie narzędzi, które mają rozwiązywać problemy potrafią generować nowe. W tym wypadku, gdzie stosowanie Acceptance Criteria albo User Stories miało być po pierwsze inicjatorem dyskusji, po drugie formą dokumentowania tego, co zostało przegadane, finalnie kończą jako forma komunikacji sama w sobie. To znaczy, że zamiast wspólnie przegadać, nad czym będziemy pracować, wymienić się spostrzeżeniami, obawami, pomysłami każdy z nas zapoznaje się z wymaganiami skazany na samotną interpretację.

W ten sposób pracując w zespole, jednocześnie pozbawiamy się korzyści wynikających z pracy zespołowej. W byciu zespołem chodzi o wspólną pracę, wymianę wiedzy i perspektywy, czyli krótko mówiąc o synergię, która jest kluczem do osiągania lepszych wyników zespołowo. Nawet Wikipedia o tym wie:

"Synergia, synergizm, synergiczność, efekt synergiczny[a] (z gr. *συνεργία* współpraca[1]) – współdziałanie różnych czynników,

1 https://pl.wikipedia.org/wiki/Synergia

którego efekt jest większy niż suma poszczególnych oddzielnych działań".

Oczywiście trzeba uważać na skrajności. Nieprzemyślane angażowanie członków zespołu w spotkania może przynosić spore straty czasowe. Trzeba wiedzieć ile osób na spotkaniu to liczba optymalna, jakie osoby powinny wziąć udział w spotkaniu, aby najwięcej z tego spotkania wycisnąć. Finalnie samo spotkanie też warto mieć pod kontrolą, aby unikać niepotrzebnych dyskusji (prowadzenie spotkań to temat obszerny, dlatego generalizuję).

Wpis #63 Problemy odkrywamy nie tylko podczas testowania

Jestem gorącym zwolennikiem zespołów zwinnych, w których osoba QA jest stałą częścią pracy projektowej i może w pełni skupić się nad jednym projektem.

Dlaczego?

Dlatego, że w ten sposób (według mnie) QA jest w stanie realnie adresować problemy obniżające jakość i efektywnie znajdować defekty.

Jeśli patrzymy na QA aka testera jako osobę, której podrzucamy zadania do testowania, to naturalnie pojawia się pomysł, że w sumie nie ma problemu, aby taka osoba była np. współdzielona między projektami. Lub w ogóle, aby utworzyć dział testerów, którzy tylko będą dostawać zadania do testowania, a potem je oddawać. Pomysł dobry pod warunkiem, że takie podejście do testowania działa. Rzeczywistość pokazuje, że nie do końca tak jest.

Jeśli podzielić defekty pod kątem złożoności, to wyróżnię sobie podstawowe i złożone. Podstawowe widzę jako te, które najczęściej wypisane są w acceptance criteria, czy jakiejś innej

specyfikacji. Tworzymy coś i to ma działać tak, a nie inaczej. Podrzucamy testerowi funkcjonalność, specyfikacje i być może jakąś instrukcję, co ma zrobić i on to weryfikuje. Jednak mając automatyczne testy jednostkowe, integracyjne, codzienne integracje (w dodatku automatyczne) zwykle taka weryfikacja specyfikacji staje się formalnością. Trzeba zerknąć, czy wszystko jest ok i zwykle tak jest. Nuda.

Problemy zwykle pojawiają się w innych miejscach. Tak jak pisałem już wielokrotnie, są to rzeczy nieprzewidziane, które zwykle nas zaskakują, bo nie ma ich nigdzie w wymaganiach. Czy taki tester, oddelegowany do zespołu na 1/6 etatu jest w stanie takie problemy znaleźć? Gdy jego wiedza o projekcie to 1/6 tego, co mógłby wiedzieć? Czy taka osoba jest w stanie odnajdywać problemy w wymaganiach, gdy zna domenę lub potrzeby klienta w 1/6 części? Czy jest w stanie obserwować pracę zespołu, rozmawiać z członkami zespołu, współpracować z nimi, żeby wpływać na wyższą jakość produktu w stopniu 1/6 ? Widać do czego dążę.

Prócz błędów, które zwykle pojawiają się w trakcie standardowego testowania przed zamknięciem zadania, regularnie odkrywam błędy dzięki temu, że rozmawiam z klientem, analizuję wymagania (do tego muszę dobrze rozumieć potrzeby klienta i nasz system), biorę udział w spotkaniach, słucham i dyskutuję ze wszystkimi członkami zespołu. Pracuję razem z nimi, korzystam z systemu w innych celach, niż cele testowe dla zamknięcia zadania, wykonuję testy regresyjne i je automatyzuję. Fun fact, okazuje się, że samo pisanie automatycznych testów e2e już kilka razy pomogło odkryć problem, który przy testach manualnych dokładnie tego samego scenariusza nie wyszedł.

Założenie 1/6 etatu może wydawać się skrajnością, ale 1/3 już okazała się być możliwością. Stąd widać, że delegując QA do trzech projektów, upośledzamy wartość, jaką do nich wnosi w stopniu 2/3? Pewnie matematycznie można to lepiej wyliczyć ale widać, że jest to stopień istotny. Już nie chce przypominać o stratach, o których mówi Lean.

Wpis #64 Pojawiające się wymagania

Czy kiedykolwiek w trakcie jakiegoś refinementu, analizy wymagań danego story, w trakcie normalnej pracy nad czymkolwiek, a może nawet w trakcie mycia zębów zdarzyło wam się doznać objawienia, które dało wam do zrozumienia, że wymagania są niekompletne, błędne, czy np. dwuznaczne? Niedługo po tym pojawia się taka myśl: "Cholera, czemu wcześniej o tym nie pomyślałem?".

Okazuje się, że to normalne, a w zasadzie nawet oczekiwane. W ten sposób skonstruowane jest zbieranie wymagań w zespołach zwinnych. Agile został skonstruowany tak, aby szczegóły dotyczące ficzerów nie były wynikiem ogromnej analizy całego systemu wykonanej już na samym początku każdego projektu, dlatego, że to zwyczajnie nie działa. Przynajmniej tak przedstawia to środowisko tzw. adżajlowców (agilists).

Zgodnie z tym taka forma zbierania wymagań i tak finalnie sprowadza się do nieścisłości, luk i innych problemów, które trzeba przegadać i naprawić. Agile więc sugeruje, aby zrezygnować z tej analizy, założyć sobie jakiś zarys systemu (w postaci story cards) i pracować nad ich szczegółami wtedy, kiedy to potrzebne, np. na refinementach, czy innych spotkaniach omawiających te karteczki. Dzięki temu dyskusje nad wymaganiami, które wcześniej były wynikiem odkrycia problemów, teraz są normalnym trybem pracy nad wymaganiami. Jednak warunkiem sukcesu jest to, że te dyskusje się odbywają, co może nie mieć miejsca, gdy zbyt mocno polegamy na spisanych wymaganiach. W ten sposób wracamy do punktu wyjścia.

Jeśli chodzi o te objawienia, jest na to ładna angielska nazwa: **emerging requirements**. Chodzi o to, że pewne rzeczy docierają do nas dopiero w określonych okolicznościach np. gdy widzimy gotowy fragment systemu, bierzemy udział w dyskusjach, zobaczymy gotowy diagram architektury i nie będzie inaczej, choćbyśmy nie wiadomo jak intensywnie nad tym myśleli.

Dlatego nie ma sensu traktować tego jako naszą porażkę, a lepiej założyć, że jest to nieuniknione i pomóc sobie doznawać takich objawień, częściej budując odpowiednie środowisko do tego.

Wpis #65 Jak radzić sobie z pojawiającymi się wymaganiami

W tym feedzie przejdę do krótkich konkretów pokazujących, jak można ujarzmić te tzw. emergent requirements, aby działały na naszą korzyść.

Pierwszym sposobem jest iteracyjne i zarazem inkrementacyjne podejście do pracy z wymaganiami. Iteracyjne to znaczy, że wymagania dla danego story przerabiane są kilkukrotnie (w etapach). Inkrementacyjnie to znaczy, że za każdym razem, gdy podchodzimy do pracy z wymaganiami dla tego story, dokładamy coraz więcej szczegółów. Ilość iteracji zależy od kilku czynników, z których można wyróżnić przede wszystkim złożoność wymagań ale też skala problemu, jaką zespół ma z wymaganiami. Jeśli zespół pracuje z wymaganiami dla danego story tylko raz i radzi sobie z tym świetnie, nie ma powodu, żeby przejść na taki model.

Aby dodać jeszcze kilka słów do akapitu powyżej, warto doczytać o metaforze góry lodowej w odniesieniu do backlogu produktu.

W skrócie chodzi o to, że na samym szczycie znajdują się najbardziej priorytetowe i uszczegółowione stories, a schodząc niżej priorytet i ich szczegółowość spada. Zespół z kolei, w trakcie trwania kolejnych sprintów pracuje nad tym, aby przesuwać zadania w górę backlogu, tym samym pracując nad ich priorytetem i szczegółami.

Drugim sposobem jest generowanie dyskusji na temat wymagań. Dyskusje to nic innego jak wymiana informacji i konfrontacja

perspektyw. Każda informacja i perspektywa danego członka zespołu da mi nowy kontekst i nowe możliwości spojrzenia na wymagania, czego nie uświadczymy w przypadku czytania wymagań spisanych.

Spotkania generują też tzw. what-if discussions. Jeśli zaangażujemy członków zespołu w dyskusje, pojawią się tendencje do dostrzegania problemów i kwestionowania czyjegoś spojrzenia. Kwestionowanie tego, co ktoś mówi brzmi negatywnie, jednak jeśli robione jest profesjonalnie, bez emocji, skutkuje wartościowymi spostrzeżeniami. Znów, czytając spisane wymagania, potrzeby kwestionowania tego, co czytamy są dużo mniejsze. Każdy z was może sam się zastanowić nad tym, jak często na spotkaniu pojawiły się dyskusje, które wciągnęły (zaangażowały) kilku członków zespołu, a jak często ta sama sytuacja miała miejsce w sekcji komentarze. Uwaga, samo pojawienie się dyskusji jest spoko pod warunkiem, że jest produktywna, a wiadomo, że nie każda taka może być.

Ponadto ile razy zdarzyło wam się tak, że finalnie wszyscy zgodzili się, że rozumieją dane story, po czym ktoś podniósł jakąś wątpliwość i okazało się, że jednak to ‘rozumienie’ nie do końca było prawdziwe. Choć ta sytuacja jest całkiem naturalna warto uważać, aby nie wpaść w tę pułapkę. To, że wszyscy są zgodni wcale nie świadczy o tym, że faktycznie wszystko jest jasne. Może są zmęczeni, może mają sporo pracy i czekają na koniec spotkania, może zostali przytłoczeni ilością informacji i przeczytali wymagania tylko pobieżnie. Przyczyny są różne, efekt ten sam, dlatego warto dyskutować.

Ja sam u siebie widzę dokładnie te same problemy. Czytając wymagania muszę się skupić, zarezerwować sobie na to czas, pilnować aby nie myśleć o zadaniu, do którego za chwilę wrócę. Muszę uważać, aby nie przytłoczyć samego siebie ilością informacji, uważać na pułapki myślowe i akceptowanie dwuznaczności, samemu próbować generować różne perspektywy i je kwestionować. Brzmi jak sporo wymuszonej pracy.

Na pewno proces w którym ktoś o roli product ownera spisuje wymagania i przekazuje je członkom zespołu do zapoznania się i weryfikacji jest szybszy. Pytanie tylko, czy efektywny. Odpowiedź

jest prosta. Jeśli nie mamy problemów z wymaganiami to gicik. Jeśli natomiast je mamy i chcemy je w jakiś sposób rozwiązać pierwszą rzeczą, która przyniesie sporo wartości jest wprowadzenie rozmowy.

Wpis #66 Co tester robi lepiej od pozostałych członków zespołu?

Testuje. A tak poważnie w tym wpisie chciałem zawrzeć to, co niedawno zauważyłem. W poprzednich wpisach mocno skupiłem się na tym, aby promować zaangażowanie całego zespołu w testowanie systemu, nad którym wszyscy pracujemy. Doszło do mnie, że w zespole, w którym każdy członek zespołu zaangażowany jest w testowanie, mogłoby się wydawać, że tester w sumie nie jest potrzebny. Jak można się domyślić, tak się tylko wydaje.

Kiedyś w artykule, który napisałem o whole-team approach podkreśliłem, że zaangażowanie całego w zespołu w testowanie warto postrzegać bardziej jako wkład, a nie przejęcie odpowiedzialności. Co to znaczy? Każdy członek zespołu testuje, dzięki czemu pomaga unikać powstawania wąskich gardeł, które często pojawiają się, gdy jeden tester pracuje nad weryfikacją wszystkich zadań jakie zostaną wykonane w sprincie. Dodatkowo każdy testujący zyskuje na tej czynności na kilka sposobów - o tym w artykule na blogu firmowym Clearcode.

Tester w takiej sytuacji wciąż ma swoje zadania do wykonania. To, na co warto zwrócić uwagę, to fakt, że te zadania dotyczą aspektów jakościowych całego projektu - nie tylko wybranych fragmentów. Z jednej strony taka jest jego rola, z drugiej przynosi mu to wiele korzyści.

Im więcej QA wie o projekcie i systemie, tym łatwiej jest mu ocenić ryzyko, dostrzec problemy i podejmować trafne decyzje

co do zakresu testowania, bądź wesprzeć zespół w tych czynnościach służąc swoim holistycznym spojrzeniem.

Dalej dzięki wytrenowanym umiejętnościom, które wynikają z faktu, że tester poświęca 100% swojego czasu na aspekty jakościowe oraz dzięki generalnej wiedzy, którą QA pozyskuje, jeśli proces pozwala mu angażować się we wszystkie elementy projektu, taka osoba przede wszystkim będzie wiedziała co przetestować, jak to zrobić, w jakim stopniu i z jakim priorytetem. Ponadto zwykle dzięki temu, że tester regularnie stawia sobie środowisko w różnych konfiguracjach, czy ładuje dane testowe, bądź ogólnie mówiąc setupuje pożądany stan aplikacji, zrobienie tego po raz 1001 będzie szybkie i bezbolesne. Szczególnie, gdy do tego posiada narzędzia, które tworzył w czasie pracy nad projektem.

Finalnie, jeśli chodzi o testowanie regresji, która najczęściej okazuje się być najbardziej żmudnym, nudnym, i kosztownym zajęciem, na które zwykle nie ma wystarczająco zasobów, podejmowanie decyzji o tym, co testować w ramach regresji, czy nawet co automatyzować, jest proporcjonalnie prostsze do wiedzy jaką posiadamy o projekcie i systemie.

Takie spojrzenie jest istotne dlatego, że w momencie, gdy decydujemy się zaangażować cały zespół w testowanie musimy pilnować tego, aby było to z korzyścią dla zespołu. Korzyści wynikające z whole-team approach są spore ale bardzo łatwo mogą zostać przyćmione negatywnymi konsekwencjami, gdy nagle od wszystkich członków zespołu zaczniemy oczekiwać tego samego wkładu jaki wnosi QA, a każdy członek zespołu będzie poświęcał sporo czasu na to, na co QA poświęci niewiele. Jak zwykle, wszystko trzeba wypośrodkować.

Wpis #67 Czy zawsze warto mieć QA w zespole? Nie...

Dzisiejszy temat jest jednym z istotniejszych i myślę, że warto promować problem, który za chwilę opiszę.

- Jeśli zespół ma problem z jakością, to zatrudniamy QA.
- QA zatrudniamy, aby wpłynął na poprawę jakości.

Wiele razy słuchając innych lub czytając, co mają do powiedzenia, natknąłem się na ciekawy problem. Zespół przez jakiś czas radził sobie bez QA, jakość nie była idealna ale nie była też zła (mówiąc ogólnikowo). Jednak organizacja lub zespół postanowili zatrudnić QA, aby dzięki temu jakość podnieść jeszcze bardziej. Ku zaskoczeniu wszystkich minęło pół roku i okazało się, że liczba problemów urosła, zamiast zmaleć. Kto by pomyślał, przecież to nielogiczne.

Taki scenariusz można spotkać od czasu do czasu. W czym więc tkwi problem? Mówiąc krótko: brak odpowiedzialności za jakość.

Zespół pracujący wspólnie bez dedykowanego QA jest świadomy tego, że o ile każdy poszczególny członek zespołu nie zadba, aby wszystko działało jak należy, nikt inny tego nie zrobi. W momencie, gdy taki zespół dostaje QA i taka osoba przejmuje większość odpowiedzialności za testowanie i sprawy jakościowe (w końcu po to ją zatrudniliśmy, prawda?), w zespole pojawia się sztuczne uczucie pewności, że skoro mamy QA, to on o wszystko zadba, a my nie musimy się już tym martwić. Staranność, jaką przykładamy do testowania i upewniania się, że wszystko działa jak należy w różnych sytuacjach, spada. QA zaczyna robić za tzw. safety net albo quality police, a jeden QA to za mało, aby taką odpowiedzialność przyjąć na swoje barki (nie wspominam już o skalowalności takiego podejścia).

W takiej sytuacji zespół o wiele lepiej poradziłby sobie bez QA, niż mając go w zespole. Ale to nie znaczy, że mamy nie

zatrudniać QA. Tu zupełnie nie o to chodzi. Chodzi o to, aby nie dać się nabrać, że QA ma supermoce zapewniające jakość. Musimy pamiętać, że nawet po zaangażowaniu takiej osoby do zespołu, zespół wciąż powinien być odpowiedzialny za jakość mniej więcej w takim samym stopniu, w jakim był nie mając dedykowanej roli. Jeśli faktycznie zależy nam na tym, aby podnieść jakość, a nie zdejmować odpowiedzialność za jakość z członków zespołu.

Możemy też oszukiwać się, że przecież zawsze testujemy to, co napiszemy i upewniamy się, że działa. Niestety to nie wszystko. Nie chodzi o to, aby kod z zadania działał, ale żeby system działał po wdrożeniu tego kodu - i to właśnie może przestać nas interesować w momencie, gdy mamy QA.

Od początku pracy w swoim zespole zauważyłem spore różnice, gdy zaangażowanie wszystkich jego członków w testowanie z czasem się zwiększało. Choć nigdy nie doświadczyłem dokładnie tego scenariusza, jaki opisuje powyżej (mam nadzieję, że nigdy nie doświadczę) wyraźnie widzę, jak zmiany tego zaangażowania przekładają się na jakość i jak naprawdę działa rola QA w zespole. Bez wahania promuję takie podejście, bo widzę, że ono działa.

Trzeba też uważać, żeby nie podchodzić do sprawy zero jedynkowo, jak zawsze. Większość tego typu problemów rozwiązuje się korzystając z równoważni. Czasem może nie mieć sensu, aby po zaangażowaniu QA tę odpowiedzialność zostawić na dokładnie takim samym poziomie ale nie ma też sensu całkowicie zrzucać ją na QA. Trzeba obserwować, ile można zrzucić na QA, żeby jednocześnie wesprzeć efektywność pracy zespołu i podnosić jakość tego, co tworzymy.

Wpis #68 Tester na spotkaniu

Dostaliśmy w zespole nowy projekt i jak to na początku każdego (chyba) projektu, zostaliśmy zasypani spotkaniami. To też zainspirowało mnie do tego, żeby podzielić się branżowym podejściem i opowiedzieć, co QA tak naprawdę robi na spotkaniach.

Rozpoczynam ten temat dlatego, że nie zawsze tester, czy QA uczestniczący na spotkaniach to standard. Słyszy się o czasach, gdzie trendem było wręcz odwrotnie, a żeby QA uczestniczył w spotkaniach i innych czynnościach okołoprojektowych trzeba było krzyczeć, błagać, machać rękami i generalnie nieustannie dawać znać o swoich potrzebach (odnoszę się do branży).

W branży testerskiej od jakiegoś czasu promuje się podejście, aby QA zawsze uczestniczył w spotkaniach. Zarówno tych omawiających design, architekturę, czy inne rozwiązania, jak i w tych dotyczących estymowania czy planowania.

Tak więc pierwszym powodem, dla którego testerzy tak bardzo chcą uczestniczyć w spotkaniach jest syndrom wiecznego studenta. My, testerzy, jesteśmy na spotkaniach żeby się uczyć. Musimy znać produkty, które testujemy. Im więcej wiemy, tym lepsze będą nasze wyniki. Od razu uprzedzam Twoje myśli, tak jest granica ROI z tej nauki. Ale generalnie, jeśli wiemy dużo, to działa na naszą korzyść - no i oczywiście na korzyść zespołu.

Ponieważ testując robimy dwie rzeczy: weryfikujemy i eksplorujemy. Tyle, że nie tylko napisany kod. Testujemy cały produkt. Od pomysłów, planów, dokumentów, architektury, prototypów po implementacje, środowiska, monitoringi itd. Chodzi o to, aby wszystko, co powstaje w ramach działalności projektowej weryfikować z wizją projektu, jego domeną, wymaganiami interesariuszy, ramami czasowymi, czy nawet możliwościami budżetowymi. Żeby weryfikować dobrze, trzeba to wszystko znać.

Na spotkaniach więc uczymy się, kto jest tym interesariuszem, czego sobie życzy, w jakiej domenie pracuje, jakie ma oczekiwania względem naszego zespołu, co myśli o swojej wizji. Uczymy się też, jak zespół podchodzi do tej wizji, jakie rozwiązania ma na horyzoncie, z jakich narzędzi będzie korzystać i wiele, wiele innych.

Wszystko po to, aby identyfikować problemy i ryzyka wcześnie, aby testować dobrze i finalnie wspierać development przez cały czas trwania projektu

Wpis #69 Tester - na spotkaniu cz.2

Z poprzedniego wpisu już wiemy, że QA siedzący na spotkaniu nie tylko układa kostkę Rubika, ale przede wszystkim uczy się kontekstu.

Oprócz tego QA uczestniczy w spotkaniu będąc w stanie gotowości, aby adresować wszelkie potrzeby związane z zapewnieniem jakości, gdy tylko uzna taką potrzebę. To znaczy dba o testowalność i dba o kulturę jakości.

W kontekście testowalności za każdym razem, gdy na spotkaniu omawiamy, co będziemy implementować, jaką architekturę planujemy zastosować, jakie możliwe rozwiązania i narzędzia - QA myśli, jak to wszystko będziemy mogli testować. Z jednej strony można powiedzieć, że cały zespół powinien myśleć zarówno o rozwiązaniach, jak i o jakości, ale to chyba nie takie proste. Gdy architekt jest pochłonięty projektowaniem rozwiązania, aby po pierwsze działało, a po drugie było wydajne, na pewno bierze w jakimś stopniu pod uwagę problemy jakościowe, ale jednak jego pierwszorzędnym celem jest funkcjonalność i wydajność.

QA z kolei robi odwrotnie. Jego pierwszorzędnym celem jest jakość. Mówi sobie w głowie: "No fajnie to wygląda ale jak to będziemy testować? Co w ogóle będziemy potrzebowali

testować, czyli jakie problemy i ryzyka mogą się pojawić przy takim rozwiązaniu?". Dzięki temu możemy potem konfrontować oba spojrzenia.

Co do kultury jakości, mimo, że nie jesteśmy w stanie myśleć zarówno o rozwiązaniach, jak i o jakości równie efektywnie (tak mi się wydaje), czasem problemem jest nie myślenie o jakości wcale. Wtedy QA promuje spojrzenie jakościowe i upewnia się, że development będzie odbywał się z jakością w tle. Zwraca więc uwagę na potrzeby, jakie mogą się pojawić przy zapewnianiu jakości i prowokuje dyskusje, aby te potrzeby od początku można było zapewniać. Krótko mówiąc, upewnia się, aby każdy pamiętał, że testowanie jest częścią developmentu.

Wpis #70 Tester na spotkaniu - część 3

Chciałem jeszcze rozwinąć temat kwestionowania. Tak jak wspomniałem, bardzo ciężko jest jednocześnie myśleć o wszystkim. Nawet jeśli próbujemy i nam się udaje, to być może efekty naszej pracy nie są tak kreatywne, tak innowacyjne i tak świetne jak mogłyby być, gdybyśmy skupili się na mniejszej ilości aspektów tego, co robimy. Czasem potrzebujemy się skupić lub dać się ponieść wyobraźni, aby zrobić coś wyjątkowego.

Pracując w projekcie można dostrzec dwie rolę:

- Budującą
- Testującą

Rolą QA jest dostarczanie krytycznego spojrzenia na to, co robimy, na produkt i projekt. Jest też zadawanie pytań w stylu: "co my tutaj pomijamy?", bądź "o czym mogliśmy zapomnieć?". Jesteśmy od tego, żeby być sceptyczni i podejrzliwi. Przewidujemy problemy i szukamy ryzyka.

Z kolei rolą budujących jest tworzenie, szukanie rozwiązań. Tam, gdzie developerzy podają odpowiedzi, tam QA jest od tego, żeby zadawać pytania.

Czasem taka postawa może być postrzegana jako destrukcyjna, czy uciążliwa. Trzeba więc starać się to robić zawsze na zasadzie bardziej demokracji niż dyktatury. Trzeba pamiętać, że za każdym razem, gdy QA coś kwestionuje, to jest moment, aby zwalidować, czy jego obawy są słuszne, czy też nie.

Spotkania są takim miejscem, w którym obie te role mają okazję się skonfrontować, ponieważ każda z nich wnosi inną perspektywę. Rola budująca ciągnie rozwiązania w jedną stronę, tę bardziej kreatywną, innowacyjną. Rola testująca sprowadza na ziemię, ocenia i kwestionuje.

Naszym celem jest znalezienie balansu i osiąganie rozwiązań, które są zarówno innowacyjne, jak i bezpieczne przed wszelkim ryzykiem i problemami.

Wpis #71 10 sposobów na granulacje do poziomu user stories

Wcześniej było o rozbijaniu epiców na user stories. Teraz ponownie, ale temat ugryziony od innej strony. Jak się okazuje, jest co najmniej 10 sposobów na to, jak podzielić wizję wymagań (bo epic to jeszcze nie wymagania) na user stories. Myślę, że można znaleźć, bądź wymyślić więcej takich sposobów w zależności od tego, z jakim projektem mamy do czynienia. W moim zespole mogę wyróżnić 3, z których na pewno korzystaliśmy i dobrze się sprawdziły.

Pozostaje jeszcze pytanie dlaczego w ogólę piszę o rozbijaniu user stories. Być może niektóre feedy przekazywały odpowiedź na to pytanie ale chętnie przypomnę.

Jako QA moim zadaniem jest zapewniać jakość - zupełnie jak sugeruje tytuł mojej roli. Jednak sam jakości nie zapewnię, ponieważ to nie ja tworzę kod produkcyjny. Dlatego wszystko, co jest w stanie usprawnić pracę zespołu i co finalnie przekłada się na lepszą jakość to moje zadanie.

“Dobre” user stories to wiele korzyści. Przejrzyste wymagania, szybka droga do walidacji, skupienie na testowalności, łatwość w dostrzeganiu problemów, porządna implementacja, taka (short and accurate) (taki jest cel ale w rzeczywistości wiadomo, różnie to bywa). Niemniej, gdy praca zespołu charakteryzuje się takimi cechami jak powyżej, jakość zaczyna rosnąć, a ja się cieszę.

Wpis #72 Acceptance Criteria na sterydach

Miałem już parę wpisów o acceptance criteria, tzn. konkretnie o tym, do czego służą i jak je tworzyć, aby służyły nam jak najlepiej. Teraz rozszerzenie do tego tematu, aby ACki służyły nam jeszcze lepiej!

Co możemy zrobić, aby podrasować nasze Acceptance Criteria? Wcześniej pisałem o tym, że ACki powinny być formułowane w formie dokonanej, czyli powinny prezentować stan systemu po implementacji. Przydaje się też, jeśli są dość obszerne i np. są formułowane tak, jakby były przypadkami testowymi.

Jednak mimo, że taki sposób tworzenia AC’kow już i tak wybiega poza standard, czyli wypisanie tego, co przyjdzie mi do głowy i wsio, to wciąż można lepiej. Konkretnie chodzi mi o przypadki testowe.

Spisując Acceptance Criteria mogę pomyśleć nad przypadkami testowymi, co by znaczyło, że dane story po zakończeniu implementacji w teorii powinno przejść moje przypadki. Przede wszystkim chodzi o to, aby pomóc osobie, która będzie dane story implementować, podsunąć podpowiedź typu: “słuchaj, jak będę testować to na pewno sprawdzę to, to i

to". Wtedy "implementator" pochłonięty projektowaniem i implementowaniem rozwiązania ma szansę nie zapomnieć o tych mniej oczywistych sytuacjach, które to zwykle sprawiają, że testerzy wracają do programistów mówiąc, że mamy bugi. W takiej sytuacji wiele razy już nie trzeba wracać z poprawkami i mamy trochę czasu w kieszeni.

Nie jest to jednak takie "hop siup". Problemem przy spisywaniu test case'ów, o czym wcześniej nie pomyślałem, jest przeskoczenie momentu analizy use case'ów. Tzn. jeśli spisując acceptance criteria zacznę od razu zastanawiać się nad test case'ami, to mogę pominąć sporo kluczowych rzeczy dlatego, że podchodzę do tego od złej strony.

Zasadniczo sprawa wygląda tak, że żeby coś dobrze przetestować, muszę się zastanowić co ma robić nasz system, komu ma służyć, jak ten ktoś będzie tego systemu używał (i w jakich warunkach) i dopiero potem projektować pod to przypadki testowe. Zwykle taki proces ma miejsce w momencie, gdy dostajemy już gotowe story, kawa jest już w kubku i siadamy do testowania. Wtedy nagle okazuje się, że właściwie jakby się tak zastanowić, to mamy dodatkowe pytania, czegoś nam brakuje, czegoś nie rozumiemy i musimy dopytać.

Mimo, że już wcześniej uczestniczyłem w pisaniu acceptance criteria i nawet starałem się już pomyśleć o jakichś przypadkach testowych, to porządnie robię to dopiero teraz.

Wiele razy pisałem. że najwięcej błędów (w przekonaniu moim i wielu) pojawia się dlatego, że autor kodu czegoś nie przewidział. Zwykle jest tak, że autor nie przewidział danych sytuacji nie dlatego, że jest głupi, czy ma wszystko w nosie. Dzieje się tak dlatego, że to są naprawdę nieoczywiste rzeczy, które wychodzą dopiero, gdy mocno wdrożymy się w daną funkcjonalność, czy w kontekst systemu.

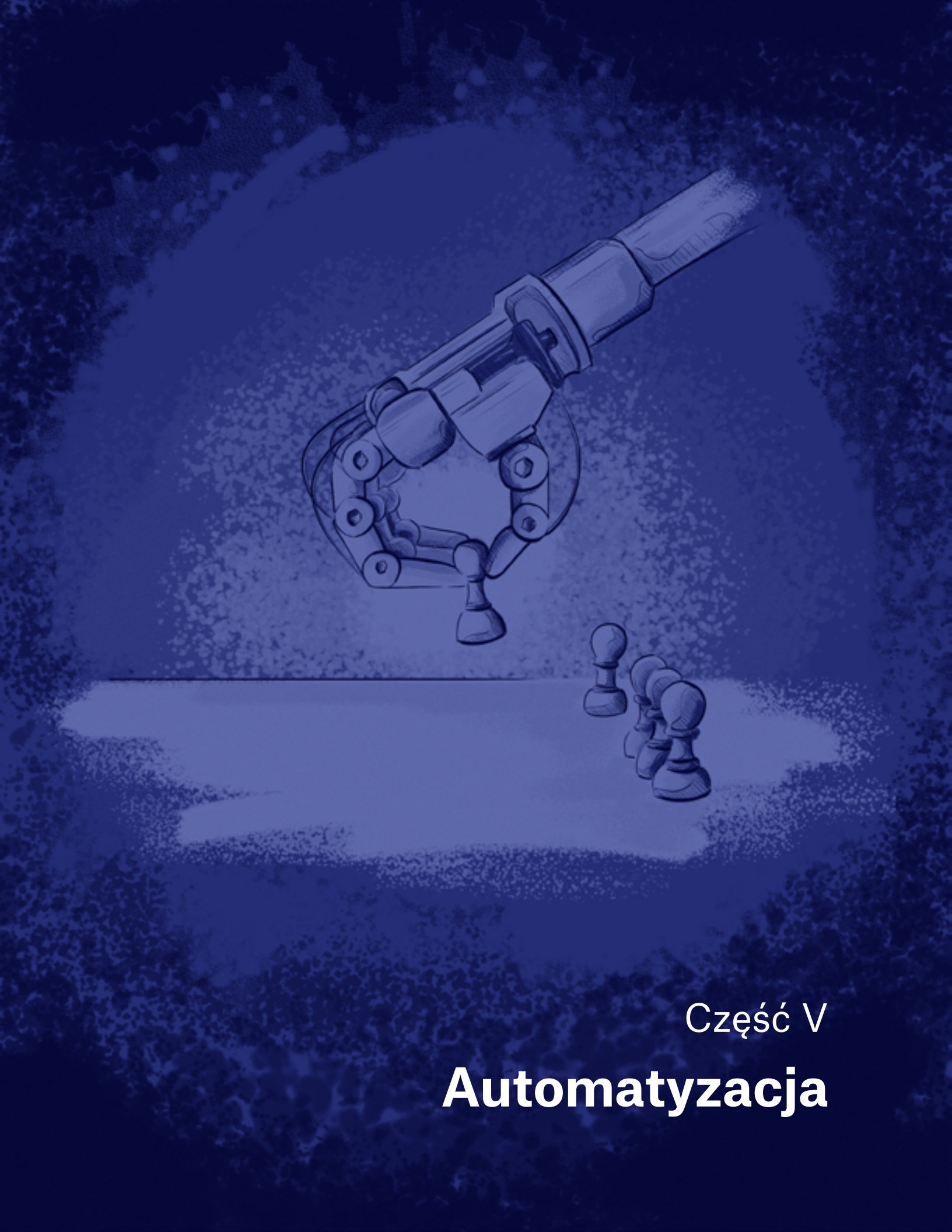
Jeśli więc próbujemy rozszerzyć nasze Acceptance Criteria o jakieś przypadki testowe tworzone na szybko, to takie przypadki mogą być mało pomocne. To, co odróżnia słabe przypadki testowe, które faktycznie znajdują błędy to dogłębne zrozumienie systemu i problemu. Co prawda o tym, że nasze

przypadki testowe są słabe dowiadujemy się dopiero po całym procesie testowania ale o tym może napiszę innym razem.

Morał jest taki, że jeśli chcemy mieć dobre Acceptance Criteria, które pomagają zapewniać jakość wbudowaną, warto je spisywać po tym, jak już przemyślimy sobie różne rzeczy, wykonamy analizę, uruchomimy swoją empatię i kreatywność, wejdziemy głęboko w problem.

Jako bonus, jedna teoria. Ciekawy jestem czy wy też to u siebie zauważyliście. Nasz potencjał kreatywności zmienia swój poziom, wręcz maleje wraz z upływem jednej sesji myślenia - bez względu na to, co robimy. Jeśli podchodzę do analizy i projektowania przypadków testowych i zastaję tabula rasę, to muszę od początku wszystko porozkminiać, aż dojdę do pewnego miejsca, z którego trudno już iść mi dalej. Mam już zmęczony, zaplątany umysł i dalsze myślenie wiele więcej mi nie przyniesie.

Jeśli natomiast podejdę do testowania i mam już sporo roboty wykonanej, np. w formie dobrych, obszernych, przemyślanych Acceptance Criteria, to ten sam kubełek energii i kreatywności przepalam na rzeczy, do których nie doszedłbym zaczynając od zera. To pozwala mi spojrzeć na problemy z zupełnie innej perspektywy, zajrzeć do innych, nowych miejsc i daje szanse na znalezienie zupełnie innych problemów.



Część V

Automatyzacja

Wpis #73 Myśl i automatyzuj, nie odwrotnie

Istnieje ogrom tematów i problemów związanych z aspektami zapewniania jakości której automatyzacja stanowi tylko część.

Mówiąc cokolwiek o automatyzacji trzeba wziąć poprawkę na to, że tester nie jest specjalistą od wszystkiego, co ma w sobie nazwę ‘test’, czy ‘jakość’. Dlatego też, jeśli chodzi o doświadczenie w automatyzacji na poziomie strukturalnym (poziom kodu źródłowego) lepszym specjalistą będzie osoba, która na co dzień z takimi testami pracuje. Niemniej, ponieważ testy zautomatyzowane to wciąż testy, sposób w jaki powinniśmy do nich podejść jest bardzo podobny i pewne zasady wypracowane na wyższych poziomach można wykorzystać również na poziomie strukturalnym.

Automatyzacja, jak sama nazwa wskazuje, polega na wykorzystaniu innego sposobu wykonywania danej czynności. Możemy automatyzować testy, ale równie dobrze możemy zautomatyzować napełnianie miski psa każdego dnia o godzinie 8 (przy odrobinie umiejętności).

Czy fakt, że wykorzystam automatyzacje pozwoli mi lepiej nakarmić psa? Albo sprawi, że będzie on bardziej zadowolony? To zależy. Jeśli np. mam tendencję do zapominania i pies chodzi głodny, wtedy będzie on mega szczęśliwy, że w końcu ma jedzenie codziennie.

Niemniej, jeśli chodzi o inne aspekty tej konsumpcji, to wciąż musimy obserwować co lubi, jaką karmę kupić, w jakiej ilości mu ją podawać i być może o której godzinie?

Dokładnie ta sama zasada tyczy się testów. Automatyzacja, ponieważ promowana jest jako złoty środek na nasze problemy jakościowe, sprawiła, że naszą pracę z automatyzacją rozpoczynamy od ...automatyzacji. To ten

moment, w którym używanie narzędzia przyćmiło jego założenia i stało się celem samym w sobie. To sprawia, że zapominamy zadawać sobie podstawowego pytania: “po co mi te testy?”. Automatyzujemy bo tak się robi, a nie dlatego, że potrzebujemy. Oczywiście nie wszyscy, nie zawsze ale często pojawiają się takie tendencje. Wtedy powstają testy, które są mało pomocne albo średnio pomocne lub trudne, czy kosztowne w utrzymaniu.

Inna strona tego problemu jest taka, że jeśli nie rozumiemy po co nam testy, to jest większa szansa, że ich nie napiszemy.

Często piszę, że w obliczu ograniczonych zasobów muszę zdecydować się na to, co testować, tzn. co jest priorytetem, co niesie za sobą ryzyko, kiedy wiem, że to co zrobiłem jest wystarczające itd. Patrząc na to w ten sposób, testy automatycznie nie różnią się zbytnio. One też nam są do czegoś potrzebne i w obliczu ograniczonych zasobów trzeba podejmować podobne decyzje. W obu sytuacjach, czasem nawet i bez ograniczonych zasobów, lepiej ograniczyć się w testowaniu - więcej nie zawsze znaczy lepiej. Jak się nad tym zastanowić zawsze jest jakiś zakres, który w większym stopniu warto pokryć testami i taki, który w sumie można pokryć testami ale nie przyniesie to zbytnio wartości - a za takie testy my wciąż płacimy naszym czasem.

Zanim przejdziemy do automatyzacji, odpowiedzmy sobie na pytanie dlaczego chcę napisać testy, co chcę nimi osiągnąć, jakie rzeczy chcę sprawdzić: funkcjonalność, czy może swoją implementację? Testowanie implementacji pewnie nie zawsze jest złe, pewnie znajdą się uzasadnione przypadki, gdzie dobrze jest mieć testy implementacji, ale jeśli testujemy implementację zamiast testować funkcjonalności, to możemy zacząć się zastanawiać, co poszło nie tak.

Wpis #74 Automatyzacja ma wspierać testowanie

Piszę ten wpis dlatego, że złe postrzeganie automatyzacji zdarza się ogólnie w branży i nawet, jeśli znajdą się poszczególne jednostki, które po tym feedzie zmienią swoje postrzeganie, to już będzie sukces.

Zapewnianie jakości zawiera w sobie testowanie, z kolei testowanie zawiera w sobie automatyzacje. Testowanie jest inwestycją często opłacalną, czasem mniej, gdy stanowi bardziej formę ubezpieczenia, za które płacimy, a mimo wszystko 'nic z tego nie mamy' (bo całe szczęście nic złego się nie wydarzyło).

Obszar testów regresji jest tak, jak ostatnio napisałem żmudny, nudny i często szalenie kosztowny. Celem automatyzacji jest zwolnić nas z tego obszaru do jakiegoś stopnia, abyśmy mogli albo zredukować koszty testowania, albo zwolnić czas, który będziemy mogli lepiej wykorzystać.

Należy jednak pamiętać, że automatyzacja zwalnia nas tylko z obszarów testowania regresyjnego i w żaden sposób nie zwalnia nas z testowania ogółem.

Dlaczego?

Automatyzując nigdy nie pokryjemy wszystkiego. Nie chodzi o code coverage, tylko case coverage. Zawsze będą 'okoliczności', które pominiemy, np. na chromie działa, na firefoxie działa, ale na firefoxie odpalonym na windowsie już nie. To tylko przykład, który ma pokazać na czym polegają okoliczności. Kod może spełniać swoją funkcjonalność, a system wciąż może nie działać w pełni z uwagi na wiele innych zależności, które można znaleźć poza kodem źródłowym.

Inna sprawa jest taka, że starając się zautomatyzować testy dla jak największej liczby tego rodzaju okoliczności, możemy stracić kupę czasu, a wyniki tego nie będą zwyczajnie warte

zachodu, jeśli np. pół roku developmentu testów pozwoliło nam odkryć, że rok później na firefoxie odpalonym na windowsie coś się popsuło.

Teraz do czego zmierzam. Posiadanie kogoś, kto (jak już pisałem kiedyś) będzie używał systemu w kontrolowany sposób, jest dużo bardziej opłacalne niż automatyzowanie przypadków całego świata. Testowanie manualne jest potrzebne. Tylko dzięki temu, jesteśmy w stanie korzystać z intuicji, kreatywności, obserwować kontekst i czerpać z tych tzw. emerging problems (odsyłam do wcześniejszego feeda). Wiele razy znalazłem problem, który 'przeciekł' przez filtry testów automatycznych, a znalazłem go przy okazji, bo zwyczajnie testowałem...

Są tylko dwa warunki:

- Potrzebuję automatyzacji, dzięki której będę mieć na to czas, a nie automatyzacji, która mi ten czas odbiera.
- Moje testowanie naprawdę polega na używaniu systemu i nie ograniczam się tylko do weryfikacji tego, co zostało zrobione.

Wpis #75 Automatyzacja powtarzalnych czynności

Czy QA powinien umieć programować, czy nie, zawsze jest kwestią dyskusyjną. Pomijając testy regresji, rozumienie technologii, korzystanie z code review itd. uważam, że **zawsze** jest spoko, jak QA potrafi zautomatyzować sobie swoją własną pracę.

Jeśli QA jest stałą częścią zespołu i regularnie pracuje nad projektem, bardzo często pojawiają się czynności powtarzalne. Setupowanie danych testowych, doprowadzenie systemu do pożądanego stanu, czy stawianie customowych instancji serwisów. Jeśli takie czynności się powtarzają albo wiemy, że będą się powtarzać,

warto zastanowić się na ile pomogłaby nam automatyzacja tych czynności.

Niektóre z tych rzeczy warto zautomatyzować chociaż po części albo tak pisać kod, aby służył czynnościom testowym. Aby poprzeć to przykładem podam sytuację z naszego projektu w zespole. Dla dwóch serwisów frontend i backend mamy osobne pipeline'y i osobne deploymenty. Używamy gitops, więc bez względu na to, skąd odbywa się deployment najważniejsze jest wskazać to, skąd pipeline ma pobrać sobie kod do zbudowania serwisu. U nas każdy branch na każdym repo ma swój deployment i na początku projektu udało mi się przewidzieć, że w związku z tym przyda się, gdy będziemy mogli w prosty sposób spinać różne wersje frontendu z backendem.

Odpowiednio zmodyfikowałem plik yamlowy i patrząc na przestrzeni całego projektu, pozwoliło mi to zaoszczędzić bardzo dużo czasu, BAAARDZO dużo.

Przydaje się również automatyzowanie ładowania danych testowych. Jest pewna część systemu, do której regularnie wracam i udało mi się przewidzieć również to, już na samym początku, jak ta część powstawała. W pierwszej kolejności napisałem sobie skrypty, które ładują mi dane. Dalej te skrypty rozszerzałam i modyfikowałem w zależności od potrzeb i rozwoju systemu. Znów, patrząc na cały projekt, zaoszczędziłem sobie dzięki temu sporo czasu + bardzo łatwo mogłem te skrypty wykorzystać testując nasze serwisy pseudowydajnościowo, dodając jakieś asyncio i inne tego typu rzeczy. Pseudowydajnościowo to znaczy, że testowałem na większej ilości requestów/s i większej ilości danych ale nie były to prawdziwe testy wydajnościowe, gdzie dok,ładnie sprawdzałem czasy odpowiedzi, wykorzystanie zasobów etc. Zależało mi na tym, aby sprawdzać jak system realizuje funkcjonalność przy większym obciążeniu.

Aby nie przedstawiać automatyzacji tylko w świetle korzyści znalazłem również powody, aby nie automatyzować wszystkiego, bądź setupować niektóre rzeczy ręcznie, mimo posiadanych skryptów. Ręczna robota ma swoje korzyści. Używam systemu jak zwykły użytkownik, regularnie wracam

do teoretycznie zakończonych funkcjonalności i zwyczajnie ich używam. Mam okazję do tego, aby obserwować co się dzieje z systemem i adresować różne problemy na bieżąco. Nie miałoby to miejsca, gdybym polegał wyłącznie na automatyzacji i mogę się pochwalić, że dzięki temu udało nam się (dalej udaje i wierzę, że będzie nam się udawać) odnaleźć różne problemy, które pojawiły się z czasem, gdy pracowaliśmy nad systemem.

Jeśli testowanie nie jest wykonywane z doskoku, a jest regularnie powtarzane, tak jak ma to miejsce w przypadku testerów i QA, warto automatyzować sobie różne rzeczy o ile wiemy, że będziemy do tego wracać. Wiem, że zawsze kłócimy się z własnymi myślami: tworzenie skryptów zajmie mi więcej czasu niż setup ręczny i bardzo łatwo jest iść na skróty. Jednak za 5 czy 10 razem robienia tego samego okaże się, że w sumie tracimy czas i lepiej było poświęcić parę godzin na samym początku.

Wpis #76 Asynchroniczność w automatycznych testach e2e - część 1

Z okazji zakończenia kolejnej fazy projektu w moim zespole i rozpoczęciu pracy nad całkiem nowym projektem, naszły mnie refleksje nad różnymi elementami związanymi z testowaniem, między innymi nad testami e2e. W branży można znaleźć różne zasady pisania dobrej automatyzacji na poziomie e2e. Staram się do tych zasad stosować, aczkolwiek sam też mam pewne przemyślenia, którymi chciałem się podzielić. Chciałem napisać o tym kilka wpisów i zacznę od asynchroniczności.

Asynchroniczność w dzisiejszym kontekście będzie dotyczyła SPA. Standardowy html polega na przeładowywaniu strony, gdy chcemy wykonywać zapytania. SPA robi to asynchronicznie, tzn. że strona nie przeładowuje się, gdy wykonuje szereg zapytań do backendu i zmienia elementy interfejsu. To generuje nam trzy problemy:

- elementy na widokach pojawiają się po tym, jak strona zostanie załadowana
- elementy, nawet jeśli się pojawią, mogą nie być gotowe, tzn. można je zobaczyć ale jeszcze czekają na jakiś konkretny stan
- zapytania do backendu wykonywane bez przeładowania strony determinują dalszy stan aplikacji

Powyższe problemy obsługiwane są lepiej lub gorzej, w zależności od framework'a do automatyzacji. Do tej pory pracowałem z selenium i cypress'em, więc opowiem jak one sobie z nimi radzą.

Selenium generalnie posiada najmniejsze wsparcie dla SPA. Całą obsługę asynchroniczności trzeba pisać ręcznie. Ma to swoje plusy i minusy. Jeśli zależy nam na elastyczności, jest to plus. Jeśli zależy nam na szybkości pisania testów, jest to minus.

Cypress wspiera radzenie sobie z asynchronicznością na dwa sposoby:

- zakłada, że elementy mogą być w trakcie renderowania, więc gdy szuka jakiegoś elementu czeka na niego przez określony czas, zanim finalnie powie nam, że nie udało mu się go znaleźć;
- daje dostęp do zapytań XHR, które można śledzić, modyfikować i wyciągać z nich informacje.

Jak więc pisać testy mając na uwadzę powyższe problemy?

W pierwszej kolejności powinniśmy przyjąć zasadę, że jeśli chcemy manipulować jakimś elementem, musimy upewnić

się, że on istnieje i że ma pożądany stan zanim przejdziemy do interakcji. Cypress niby czeka jakiś czas, aż element pojawi się na widoku. Jednak my wciąż możemy napisać kod tak, że szukanie elementu zostanie wywołane w złym momencie i framework nigdy tego elementu nie znajdzie. Zaoszczędzi to sporo problemów, które ciężko potem debugowac. Oczywiście nie wszystko jest warte funkcji czekających, więc naszym zadaniem jest identyfikować miejsca problematyczne i je zabezpieczać

W drugiej kolejności warto skorzystać z dostępu do zapytań XHR, jeśli pracujemy z cypress'em. Zawsze zakładajmy, że request może nie rozwiązać się szybko i jeśli polegamy na jego odpowiedzi warto na nią zaczekać. Weryfikując napisane testy poprzez odpalanie ich lokalnie, możemy wpaść w pułapkę. Środowisko lokalne może działać wolniej, to daje aplikacji więcej czasu. Requesty dostaną odpowiedzi i uznamy, że nie trzeba na nic czekać. Następnie odpalimy cały zestaw testów w CI i dopiero tam okaże się, że jednak jakieś requesty nie zdążyły się rozwiązać, a driver już przeszedł do kolejnych czynności. Debugowanie takich problemów jest strasznie męczące i czasochłonne.

Mam wrażenie, że doświadczenie w automatyzacji testów poprzez GUI to zarówno umiejętności pisania czystego kodu (które w przypadku takich testów nie muszą być szalenie zaawansowane) ale również znajomość problemów narzędziowych i specyfiki działania aplikacji w przeglądarkach. Bardzo łatwo jest napisać testy, które przechodzą lokalnie w sztuce jeden. Dużo ciężej jest napisać testy, które będą przechodzić zawsze, tzn. będą stabilne i tym samym rzetelne.

Feed #77 Asynchroniczność w automatycznych testach e2e - część 2

W drugiej części feeda chciałem powiedzieć o asynchroniczności w procesach systemowych aplikacji, którą testujemy używając zautomatyzowanych testów e2e.

Poziom problematyczności wynikającej z asynchroniczności zależy w dużej mierze od systemu, jaki chcemy pokryć testami. Asynchroniczność może pojawić się zawsze i wynika przede wszystkim z luki pomiędzy momentem zainicjowania jakiegoś procesu w systemie, a momentem, w którym ten proces się zakończy, a jego wyniki staną się dostępne do dalszej manipulacji przez nasze testy automatyczne.

Z mojego punktu widzenia specjalizacja Clearcode daje nam więcej procesów, które są asynchroniczne i tym samym tworzenie testów automatycznych dla tych procesów jest dużo bardziej wymagające.

Przechodząc do konkretów, każdy test zwykle składa się z co najmniej dwóch rzeczy:

- Inicjowania procesu
- Weryfikacji wyniku.

Zwykle łatwo jest nam ocenić co trzeba zrobić, aby zweryfikować wynik procesu i czy istnieje konieczność ‘czekania’ na niego. Testujemy więc inicjowanie i weryfikację, dodajemy jakąś funkcję czekającą na wynik.

Gdyby świat był taki prosty nie mielibyśmy żadnych problemów. Niestety, nie jest. Do tych dwóch rzeczy przychodzi jeszcze nieszczęsny setup, czyli wprowadzanie

danych testowych, tworzenie obiektów i doprowadzanie ich do konkretnego stanu tylko po to, aby finalnie przejść do tego, co chcemy przetestować i wykonać te dwa wyżej wspomniane kroki.

Tutaj pojawiają się schody dlatego, że do setupu danych zwykle przykładamy mniej uwagi, a potrzeba obsługiwania asynchroniczności jest w nich dokładnie tak samo duża, jak i na innych etapach. Efektem tego są niestabilne testy. Takie, które raz przechodzą, a raz failują.

Pisząc setup frywolnie wprowadzamy dane, tworzymy obiekty, manipulujemy stanem zakładając, że to wszystko po prostu się wykona. Potem dochodzimy do sytuacji, gdzie chcemy utworzyć obiekt, a danych jeszcze nie ma. Albo chcemy manipulować stanem obiektu, który jeszcze się nie utworzył. Te mankamenty to piekło kogoś, kto pisze testy e2e. Siedzi potem godzinami analizując co jest nie tak.

Bez względu na to, czy nasze testy rozpoczynają od GUI, czy od API, pisząc automatyzację trzeba bardzo ostrożnie podchodzić do wszystkich procesów i obsługiwać asynchroniczność. To takie lessons learned z ostatniego mojego projektu. Napisałem o tym dlatego, że większość źródeł z jakimi miałem do czynienia traktuje stabilność testów głównie w kontekście selector'ów, czy danych testowych (o nich też napiszę). Nigdy nie spotkałem się ze źródłami, które traktowały o problemach asynchroniczności, a patrząc na swój ostatni projekt, to te problemy stanowiły na oko 70% wszystkich problemów z jakimi musiałem się uporać. Teraz definitywnie widzę, że im częściej piszę kod, który weryfikuje, że stan jakiego oczekuje w testach faktycznie istnieje, tym mniej niespodzianek pojawia się na CI.

Wpis #78 Dane Testowe w automatyzacji e2e

Droga do stabilnych testów jest długa. Obsługa asynchroniczności to jedna z potrzeb, którą trzeba zapewnić, porządne dane testowe to druga potrzeba.

Dane testowe pojawiają się często w kontekście omawiania dobrych praktyk pisania automatów. Osobiście też miałem okazję pocierpieć trochę z powodu ładowania danych, które było słabej jakości.

Tutaj chciałbym rozróżnić dwie istotne rzeczy: z jakich danych testowych korzystamy i jak te dane testowe ładujemy.

Dobrze jest, żeby każdy test miał swoje dane testowe, niezależne od pozostałych testów. Dzięki temu unikamy problemu, że któryś z testów modyfikuje dane dla innego testu. Wiadomo jak to się skończy.

Ładowanie danych testowych przed wykonaniem każdego testu jest kolejną dobrą praktyką. Daje nam przejrzystość na jakich konkretnie danych pracuje nasz test i ułatwia debugowanie. Na pewno nie chcemy sytuacji, w której, gdy test się wysypie musimy przejrzeć jeden wielki setup danych i dojść do tego, które konkretnie dane zostały wykorzystane w naszym failującym teście. Debugowanie to jest niestety to, na co poświęcimy sporą ilość czasu pracując z automatyzacją e2e, no... chyba, że napiszemy je dobrze.

Jeszcze gorszym pomysłem jest manualne tworzenie danych testowych i trzymanie ich na jakimś środowisku ‘pod testy’. Taki sposób szybko okaże się problematyczny.

Bonusem jest fakt, że niezależne dane testowe pozwolą nam wykonywać testy równolegle, jeśli będziemy czuli taką potrzebę.

Inną sprawą jest sposób, w jaki ładujemy dane testowe. Tutaj pojawiają się kwestie efektywności i wiarygodności.

Efektywność oznacza “o ile czasu ładowanie danych wydłuży nam wykonywanie danego testu”. Szczególnie, że przy większej ilości testów może się okazać, że 20-40% czasu wykonywania testów ładujemy dane testowe.

Dodatkowo efektywność odnosi się do tego, czy jesteśmy w stanie załadować wszystkie dane testowe jakich potrzebujemy. Jeśli nie, to ciężko będzie nam zautomatyzować potrzebne testy. Niestety takie możliwości trzeba sobie zapewnić, nie jest to ficzer out-of-the-box, który każdy system posiada.

Na koniec wiarygodność to podjęte próby względem udanych prób. Używajmy takiego sposobu ładowania danych, które zawsze da nam sukces. Niezaładowane dane to failujący test. Failujący test to czas poświęcony na debugowanie i -1 do zaufania dla testów. Ładując dane testowe używajmy interfejsu, który da nam rzetelne wyniki i pamiętajmy, że problemy asynchroniczności są tak samo obecne przy ładowaniu danych, jak i w dalszych etapach pisania testów.

Wiadomo, że są sytuacje, w których nie potrzeba nam martwić się powyższymi problemami. Jednak te problemy pojawiają się bardzo szybko, gdy myślimy o szerszej automatyzacji. Jeśli już planujemy architekturę naszych testów, warto zrobić to biorąc pod uwagę to, co u wszystkich powoduje utratę włosów

Wpis #79 Po co nam stabilne automaty e2e

Bardzo podoba mi się jedno stwierdzenie:

If you have bad tests, automation can help you do bad testing faster.

To stwierdzenie w pierwszej kolejności zachęca, aby myśleć nad wartością przypadków testowych, które

automatyzujemy. Można również sprowadzić to do prostego przekazu:

- Automatyzacja nie oznacza sukcesu, wręcz przeciwnie zła automatyzacja może nas pogrążać.
- Czego zwykle chcemy od automatyzacji
- Chcemy zyskać czas, aby nie tracić go na powtarzalne manualne testy regresji
- Chcemy dostawać informacje zwrotną szybko
- Chcemy mieć pewność, że testy regresji zawsze wykonają się tak samo, bo człowiek jest skłonny do pomyłek przy powtarzalnych czynnościach

Automatyzacja wydaje się być prosta, to fakt. Napisać drivera, który klika po GUI jest bardzo prosto i szybko można się tego nauczyć. No ale hej, postawić serwer we flasku też można szybko prawda? Dopiero potem pojawiają się wyzwania. To samo jest z automatyzacją.

Co się stanie, jeśli pomyślimy, że automatyzacja jest prosta i można zatrudnić studenta, który się tym zajmie?

- Będziemy mieli testy, które nie są wiarygodne. Czasem będą przechodzić od razu, zwykle jednak będą failować. Finalnie nie będziemy wiedzieli, czy testy dają nam jakąś rzetelną informacje. Nie będziemy też wiedzieli, czy jeśli failują to mamy problem z systemem, czy to może sprawa słabych testów.
- Zamiast zyskiwać na czasie będziemy musieli coraz więcej czasu poświęcać na debugowanie testów, które failują. Również na ewentualne poprawki, jeśli się w końcu zreflektujemy. Poświęćmy zatem sporo czasu, aby takie testy napisać i drugie tyle, aby się z nimi szarpać. Na sam koniec okaże się, że lepiej byłoby testować wszystko manualnie.
- Na informacje zwrotną będziemy czekać. Niech będzie, że testy lecą 2-4h. Kolejne 2-4h trzeba poczekać, aż ktoś

zdebuguje co się stało z tymi testami, które nie przeszły. Tak czekamy i czekamy... Jeśli przyjdzie moment, że będziemy chcieli skrócić czas wykonywania, to trzeba będzie wszystko przepisać. Finalnie zaczną się pojawiać myśli, czy nie lepiej zrezygnować z tych testów skoro i tak nie da się z nich normalnie korzystać.

Potrzeba czasu, aby pisać testy automatyczne, wykonywać je, debugować te, które nie przechodzą, a następnie je poprawiać. Często zaleca się spojrzenie na automatyzacje e2e większej ilości przypadków testowych w kategorii osobnego projektu, który potrafi mieć równie duże potrzeby, co funkcjonalny projekt, jaki chcemy pokryć takimi testami. Wszystko zależy od tego, jak duża ta automatyzacja powstanie. Im większa, tym poważniej powinniśmy do tego podejść. Zapewnianie dobrej architektury testów i dobrego code design pozwala zmienić proporcję czasu pisania nowych testów w stosunku do utrzymywania tych istniejących. Pozwala wyciągać z testów to, czego potrzebujemy i dlaczego w pierwszej kolejności zdecydowaliśmy się je pisać. Przede wszystkim natomiast pozwala stwierdzić: “fakt, manualnie zajęłoby mi to więcej czasu, całe szczęście, że mamy te testy automatyczne”. Ja osobiście podchodzę do tego w taki sposób: nawet jeśli testy, które sam piszę nie są idealne na ten moment i być może testując manualnie poświęciłbym tyle samo czasu albo nawet troszkę mniej (choć uważam, że w tej chwili bilans mam na plus), staram się wciąż poprawiać ich jakość patrząc przyszłościowo. Im dłużej dbam o jakość, tym lepsze testy jestem w stanie pisać i coraz mniej potrzebuje czasu na to, aby się z nimi szarpać. Może przyjdzie kiedyś moment, że będę tylko spijał śmietankę z tego, co napisałem.

Wpis #80 Jak dobierać test case'y do automatyzacji

Automatyzując przypadki testowe prędzej, czy później natrafiamy na problem doboru testów. Czyli wybór co testować, a czego nie testować.

Podejmując decyzje musimy pamiętać o tym, że automatyzacja nie powinna wykrywać błędów w nowych funkcjonalnościach. Tzn. naszym celem nie jest zautomatyzowanie tego, co robi tester, gdy pojawia się nowa funkcjonalność. Głównie dlatego, że nigdy nam się to nie uda. Stracimy sporo czasu, a i tak nie przetestujemy tego równie efektywnie.

Uwaga, oczywiście są sytuacje, w których testy automatyczne można wykorzystać do testowania nowych funkcjonalności. Pewne implementacje mają tak techniczne podłoże, że testowanie tego manualnie może nie być efektywne albo nawet możliwe. Do takich case'ów lepiej faktycznie pisać automaty.

Do wszystkich innych case'ów tester (zakładam, że jest dobry) skutecznie przetestuje nowe funkcjonalności. Zrobi to szybciej, porównując do czasu pisania testów automatycznych i zrobi to efektywniej. Znajdzie to, czego testy automatyczne nigdy nie znajdą, bo testowanie to proces intelektualny, który ewoluuje. Każde powiedzmy 10 minut testowania daje nam nowy pogląd i nowe informacje, które stanowią input do dalszego działania.

Z kolei automatyzacja ma sens wtedy, gdy to, co tester będzie musiał zrobić manualnie jest powtarzalne i czasochłonne - testy regresji, czyli testowanie tego, czy nie zmieniliśmy czegoś, co do tej pory działało. Jeśli nie wiesz jakie testy napisać, to wyobraź sobie, że przy każdym merge'u, albo po każdym push'u musisz przetestować coś manualnie. Jeśli zaczniesz siwieć już na samą myśl to znaczy, że to dobry materiał na automatyzację.

Warto się jeszcze zastanowić nad ryzykiem, które chcemy zminimalizować. Gdybyśmy się bardzo mocno skupili możemy wymyślić całą listę testów, których nie chcemy powtarzać ręcznie. Ale to nie znaczy, że wszystko jest warte uwagi. Więc w następnej kolejności, zastanów się jak bardzo zależy Ci na tym, aby coś działało i jak duża jest szansa na to, że to może przestać działać. Inaczej mówiąc, jak bardzo boisz się, że dany hipotetyczny scenariusz może pojawić się w rzeczywistości.

Trzeba pamiętać, że automatyzacja kosztuje nas sporo czasu. Zarówno rozwijanie, uruchamianie, jak i utrzymywanie to koszty. Jak to mówią, at the end of the day, automatyzacja musi nam się opłacać.

Wpis #81 Testy e2e z punktu widzenia utrzymania

Testy automatyczne na poziomie e2e warto traktować jako projekt i traktować go z powagą rosnącą wraz z jego rozmiarami.

Projekt testów e2e może stawiać przed nami równie problematyczne wyzwania, co projekt produktu, który chcemy pokryć tymi testami. Fajnie jest pisać nowy kod, nowe testy = nowe ficzery. Jesteśmy wtedy kreatywni, projektujemy rozwiązania, spełniamy się i cieszymy się, gdy to działa. Ale tak samo jak z pisaniem produktów, spora część czasu idzie w utrzymanie takiego kodu. Również i tutaj warto dbać o dobre praktyki, aby ułatwić sobie życie naszych przyszłych nas.

Po pierwsze testy dobrze jest zaprojektować tak, aby dopisywanie nowych, czy modyfikowanie istniejących zajmowało nam stosunkowo niewiele czasu. W tym celu możemy korzystać z różnych wzorców projektowych, np.

page object pattern. Praca z dobrym projektem jest całkiem przyjemna pod warunkiem, że dobrze się go zastosuje.

Ponieważ debugowanie failujących testów jest jedną z większych bolączek, trzeba sprawić, aby było ono łatwiejsze. Możemy starać się pisać kod do testów w taki sposób, aby odzwierciedlał flow tego, co te testy robią. Czasem jednak ładne nazwy funkcji, czy metod nie wystarczą. Musimy celowo zrobić tak, aby flow wyciągnąć na najwyższą warstwę abstrakcji i w tym celu możemy korzystać z enkapsulacji. Łatwiej nam będzie przypomnieć sobie, co ten test konkretnie robi bez konieczności przeglądania wszystkiego co importujemy.

Przydaje się dobre logowanie. Dobre to znaczy takie, które pokaże nam co się wysypało. Nie zawsze jednak życie jest takie proste, więc możemy chociaż zadbać o to, aby wiedzieć, w którym momencie coś się wysypało. W tym celu powinniśmy albo pisać kod na tyle prosty, aby traceback pokazał nam w którym momencie test się wysypał, albo dodawać do niego breakpointy w postaci właśnie logów czy asercji. Cokolwiek, co da nam informacje, w którym miejscu ostatni raz coś udało się zrealizować zanim pojawił się błąd.

Finalnie przydaje się dodawanie screenshotów, które potem będą dostępne dla nas w celu debugowania. Screenshoty nie rozwiążą naszego problemu ot tak. Niestety często powodem dla failujących testów są jakieś subtelne problemy. W takich sytuacjach zarówno obrazek, jak i logi nie pokażą nam problemu wprost ale będą stanowić doskonałą pomoc, aby rozpocząć inwestygowanie problemu od jakiegoś solidnego fundamentu.

Na koniec najważniejsze, jeśli piszemy testy lokalnie i wszystko nam działa, to poziom niezawodności jest mniej więcej taki sam, jak w przypadku kodu produktowego. Dopiero jak zintegrujemy nasz kod z resztą testów i odpalimy je na docelowym środowisku, wtedy zaczynają się niespodzianki. Dlatego to, że nasze testy przechodzą odpalone lokalnie, to nie jest informacja, że nasza praca dobiegła końca.

Wpis #82 Suche testy automatyczne

DRY - Don’t Repeat Yourself, czyli krótko mówiąc dobra praktyka, którą wszyscy pewnie znają, więc nie będę rozpisywać się
w jakim celu się jej używa i jakie korzyści ona przynosi.

Choć podkreślałem podobieństwa w pisaniu kodu dla testów automatycznych na poziomie e2e i naszej aplikacji, którą chcemy takimi testami pokryć, zasada DRY może być strzałem w kolano w przypadku testów.

Testy z natury zawierają w sobie powtórzenia. Testujemy jedną funkcję, funkcjonalność, widok, czy proces na wiele różnych sposobów. Każdy test testujący ten sam element tylko na inny sposób, chciał nie chciał, będzie częściowo korzystać z tego samego kodu. Bardzo szybko pokusimy się o wydzielenie tego kodu do jakiejś reużywalnej funkcji.

Niestety może (nie musi - jednak zwykle tak się dzieje) to skończyć się super skomplikowanym kodem, który wcale nie jest bardziej czytelny, czy łatwiejszy w utrzymaniu.

Wręcz przeciwnie, zrobimy sobie problem w postaci:

- Reużywalnych funkcji, które będą miały mega dziwne nazwy, ponieważ chcemy dostosować te jedną reużywalną funkcje do różnych kontekstów jej użycia,
- Reużywalnych funkcji przeładowanych różnymi warunkami, parametrami i innymi kombinacjami, bo każdy przypadek użycia tej funkcji wymaga jednak odrobinkę innej konfiguracji
- Modułów wypełnionych całą pseudo reużywalnych funkcji, które tak naprawdę i tak są customowe, gdyby się tak głębiej nad tym zastanowić

Na sam koniec jak spojrzymy, co żeśmy zrobili zobaczymy mega

brzydkie, nieczytelne (ale reużywalne) funkcje, które przypominają wyciskanie sztangi obciążone gumami treningowymi wykonywane stojąc na piłce gimnastycznej.

Ja już to przerabiałem, inni też to przerabiali - nie ma sensu. W przypadku testów lepiej pozwolić sobie robić copy-paste z małymi modyfikacjami, bo jest to czytelniejsze, łatwiejsze w używaniu i utrzymywaniu oraz zdrowsze dla samego developera testów.

Oczywiście pewne rzeczy będzie można wydzielić, a nawet warto będzie wydzielić, jednak jest to bardziej wyjątek niż reguła.

Wpis #83 Jeszcze raz o suchych testach

Z uwagi na powtarzalną naturę testów być może ciężej jest pisać abstrakcje i dlatego one nie wychodzą za dobrze. Być może większość przypadków źle napisanych abstrakcji pochodzi od osób, u których doświadczenie w programowaniu jest mniejsze - testerzy w przeważającej części zajmują się tematami około testowymi.

Bez względu na to, co i kto jest przyczyną powstawania brzydkich abstrakcji faktem jest, że takowe powstają. Powstają do tego stopnia, że pojawiły się zalecenia, aby mocno zastanowić się, czy w przypadku testów są one słuszne. To jak dobre są abstrakcje wychodzi tak naprawdę z czasem, gdy wracamy do tego kodu i próbujemy go zrozumieć, rozszerzyć albo przepisać. Im więcej czasu poświęcamy na pracę z automatyzacją, tym bardziej rozumiemy, co robimy i jak to się przekłada na późniejszą pracę.

Moje wnioski są takie: być może nie warto za wszelką cenę wystrzegać się DRY, ale nie warto też za wszelką cenę

próbować się do niej stosować. Trzeba dobrze ocenić swoje umiejętności zanim zaczniemy kombinować z wzorcami projektowymi i dobrymi praktykami oraz używać ich, gdy wiemy co robimy. Warto też wziąć pod uwagę fakt, że nasze doświadczenie wywodzące się z tworzenia systemów nie zawsze ma przełożenie 1:1 przy pisaniu testów. Oceniając więc gdzie mogą przydać się abstrakcje i jak powinny one wyglądać, być może powinny wynikać z naszego doświadczenia z pracy z testami, a nie doświadczenia pracy z systemami.

Dlatego wciąż nie zmieniam zdania co do tego, jak edited (źle użyta) zasada DRY może utrudnić nam życie. Widziałem ile problemów generowały abstrakcje tworzone na siłę, sam też takie tworzyłem. Zachęcam tylko do ostrożności.

Wpis #84 Jak dobierać test case'y do regresji e2e?

Wiele razy opisywałem ile mamy możliwych rodzajów testów, technik testowych, poziomów testowania - generalnie testować można sporo. Mimo tego znów muszę rozszerzyć słowniczek, aby można było omówić problem doboru testów dla regresji.

Do słowniczka dokładam testowanie pozytywne i testowanie negatywne oraz testowanie ekstremalne i testowanie krytyczne. Nie są to jakieś akademickie pojęcia, od czasu do czasu spotykam się z taką kategoryzacją. Głównie chodzi o założenia, nie terminy.

Testowanie pozytywne, czyli happy paths. Testujemy, że system działa, gdy użytkownik korzysta z niego poprawnie, np. logując się podaje poprawne dane.

Testowanie negatywne jako kontra dla happy paths. Testujemy, że system radzi sobie, gdy użytkownik korzysta

z niego niepoprawnie, np. podaje niepoprawne dane do logowania.

Druga para, czyli testowanie krytyczne po jednej stronie. Bez względu jaką ścieżkę (pozytywną czy negatywną) przechodzi użytkownik, system musi sobie z nią poradzić, bo bez tego leżymy. Np. nie możemy zalogować użytkownika, gdy poda poprawny login, ale poda niepoprawne hasło.

Po drugiej stronie testowanie ekstremalne.Testujemy, że system radzi sobie, gdy użytkownik korzysta z niego ekstremalnie źle. Kombinuje, robi setki niezamierzonych akcji i ma niesamowite pomysły na inputy. Są to sytuacje ekstremalne, ale niekoniecznie krytyczne. Aczkolwiek ekstremalny case może wywołać błąd krytyczny.

Ciężka sprawa co? Pisałem, że przydają się umiejętności analityczne.

Przechodząc do wybierania testów regresji, gdy piszemy sobie regression suite (manualne, czy automatyczne) nie możemy wrzucić tam wszystkiego, bo klient nie doczekałby się diplojów. Musimy coś wybrać.

Napiszę jak ja podchodzę do tego tematu. Ma to jakieś pokrycie z innymi źródłami, ale w żadnym wypadku nie twierdzę, że to złoty środek.

Najważniejsze zawsze są dla mnie testy/case'y krytyczne. Bez względu na to jakiej ścieżki dotyczą i czy są ekstremalne, jeśli case jest krytyczny to bardzo nie chciałbym, aby wydarzył się na produkcji. No... czasem pewnie znajdzie się jakiś wyjątek.

W dalszej kolejności rozważam dorzucenie happy paths. Zwykle są to wszystkie testy dlatego, że najczęściej ścieżki pozytywne to również case'y krytyczne.

Czy zawsze dorzucam do regresji wszystko co powyżej? Nie zawsze. Zwykle to zależy od tego ile testów się nazbiera i ile mamy realnie czasu. Może być tak, że będziemy musieli ocenić krytyczność funkcjonalności, a nie case'ów. W takiej sytuacji zdecydujemy, że niektóre z nich

pozostaną nieotestowane albo otestujemy je bardzo skąpo, a większą uwagę poświęcimy tym bardziej krytycznym funkcjonalnościom. Życie.

Na koniec case'y negatywne i ekstremalne. Tutaj już się mocno zastanawiam. Zwykle takie testy wykonuje raz, gdy pojawia się coś nowego. Jeśli uważam, że te case'y nie są krytyczne i nawet gdyby się pojawiły na produkcji, to aż tak wiele się nie stanie, to nie dodaje ich do regresji. Niestety czas jest skończony, zwykle kończy się zbyt szybko. Trzeba pogodzić się z tym, że zawsze coś się może popsuć. Ekstremalne case'y, mają tę zaletę, że rzadko się pojawiają, dlatego podejmujemy ryzyko i zostawiamy je bez testów. Negatywne, które nie są krytyczne zwykle też nie zrobią nam krzywdy. Jeśli już się pojawią no to hotfiksik, trudno. Jest to tańsze niż pisanie, wykonywanie i utrzymywanie tych testów.

Wpis #85 Automatyzacja potrafi zaskoczyć

Ostatnio miałem parę przemyśleń związanych z automatyzacją testów. W zasadzie planuje zrobić feed o tym, w jaki sposób w ogóle podejmować decyzje, czy tworzyć testy automatyczne, czy nie (konkretnie na poziomie e2e). Ten feed jest zachętą do tego, dlaczego testy automatyczne często się przydają.

Zwykle, gdy myślimy o automatyzowaniu testów e2e to myślimy o testach regresji, których nie chce nam się powtarzać ręcznie. Sam też często to podkreślam, że testy automatyczne to nic innego jak zautomatyzowane testy regresji dlatego, gdy myślimy o czymkolwiek związanym z automatyzacją to w pierwszej kolejności powinniśmy pomyśleć o problemach regresji - może nie mamy ich wcale?

Powiedzmy, że podjęliśmy decyzje o automatyzowaniu testów regresji i zabieramy się za ich tworzenie. Taka

decyzja po pierwsze da nam jakąś siatkę bezpieczeństwa, możemy rozwijać nasz system i wprowadzać w nim zmiany mając z tyłu głowy, że jak coś zepsujemy, to zawsze mamy testy, które to wyłapią.

Jednak! Jest jeszcze inny pozytywny aspekt automatyzacji. Okazuje się, że posiadanie testów automatycznych oraz sam proces ich tworzenia wiele razy pozwolił mi odnaleźć problemy, o których w życiu bym nie pomyślał i tym samym nigdy bym nie sprawdził testując manualnie.

Ciekawy jestem, czy wy macie podobne doświadczenia? Nie wiem jak to wygląda w przypadku testów kodu bezpośrednio ale testy pisane na 'działającym systemie', bez względu, czy przez GUI, czy API, czy na innym poziomie właśnie tak się sprawdzają.

Często są to różne anomalie w zachowaniu systemu, które można zobaczyć, gdy coś się wykona odpowiednio szybko, odpowiednio długo, z odpowiednim obciążeniem lub odpowiednio wiele razy. Czasem też testy automatyczne potrafią szybko wykonać żmudną, niemożliwą do odtworzenia ręcznie robotę i właśnie tam pokazują się problemy.

Podejmując decyzję o tym, czy tworzymy jakąś automatyzację testów, czy nie logicznie rzecz biorąc musimy przemyśleć sobie wszystkie za i przeciw, czy właśnie koszty i zyski. Jednak nigdy nie przewidzimy wszystkich korzyści, jakie ta automatyzacja przyniesie i jakie konkretnie błędy pomoże nam wykryć. Nie chciałbym abyśmy podejmowali takie decyzje tylko i wyłącznie na podstawie intuicji ale z pewnością trzeba wziąć pod uwagę korzyści, które mogą, ale nie muszą, ukazać się z czasem, niż tylko twardo rozpisane za i przeciw. Pewnych rzeczy nie przewidzimy, ale nasza intuicja często dobrze nam podpowiada, gdy czegoś nie jesteśmy w stanie udowodnić na moment podejmowania decyzji ale wiemy, że z czasem będziemy mogli to zrobić.

Wpis #86 Automatyzować czy nie automatyzować?

Często, gdy rozpoczynamy nowy projekt pojawia się dylemat, jak w tytule tego wpisu. Wszyscy wiemy, że istnieje coś takiego jak testy automatyczne i zwykle zastanawiamy się nad tym, czy je tworzyć, czy ich nie tworzyć.

W ostatnim feedzie wspomniałem, że właśnie kolejny będzie o podejmowaniu decyzji dotyczącej automatyzacji. Dzisiaj odnoszę się konkretnie do automatyzacji na poziomie end-to-end, czyli testowania systemu, który jest ‘żywy’.

Stojąc na linii startu, jak tu sobie odpowiedzieć na to pytanie? To co z tymi automatami? Tworzymy, czy nie tworzymy? Moim zdaniem, jeśli nasza decyzja co do automatyzacji to odpowiedź na pytanie “automatyzować, czy nie”, to podejmujemy ją źle.

Dlaczego źle? Podejmowanie decyzji, czy automatyzować na podstawie tego, co wiemy o automatyzacji to jest podchodzenie do tematu od złej strony. Zamiast zastanawiać się, jaki problem chcemy rozwiązać, zastanawiamy się nad potencjalnymi plusami i minusami takiej automatyzacji zwykle bazując na wszechobecnej opinii i utartych za i przeciw.

Jaki problem możemy chcieć rozwiązać? Zwykle to problem systemu, który wymyka się spod naszej kontroli. Gdy system jest mały i np. dodajemy do niego mały klocek, łatwo jest przewidzieć co się stanie i równie łatwo jest przetestować wprowadzoną zmianę. Sytuacja wygląda bardzo podobnie, gdy system jest prosty.

Ale, jeśli system jest złożony, bądź duży, bądź jedno i drugie, przychodzi taki moment, gdzie przestajemy ogarniać. Nie jesteśmy w stanie przewidzieć wszystkich konsekwencji wprowadzanych zmian, przestajemy kontrolować co się dzieje w naszym systemie, a jego stan (działa vs nie działa) staje się dla nas coraz bardziej mglisty. Dokładnie wtedy

rozpoczyna się etap przypuszczeń, etap ‘być może’, etap ‘w sumie to nie wiem, nie pamiętam’.

Tutaj pojawia się pierwsze pytanie, na które musimy sobie odpowiedzieć. Czy taki stan rzeczy nam odpowiada? Czy możemy pozwolić sobie na niewiedzę, co dzieje się w naszym systemie? To, że przy mergowaniu do mastera kiedyś tam działało, wcale nie daje nam 5-letniej gwarancji, że tak już będzie.

Jeśli odpowiemy sobie - tak, możemy sobie na to pozwolić, to luzik. Automatyzacja to coś, o czym powinniśmy zapomnieć.

Jeśli odpowiemy sobie - nie, to idziemy dalej.

Skoro wiemy, że potrzebujemy wiedzieć co dzieje się z naszym systemem, to teraz pytanie jak możemy to zrobić? Co da nam taką informację? Może regresja manualna? Może jakiś monitoring? W zasadzie to, czy musimy wiedzieć o wszystkim, czy może wystarczy nam znać stan tylko jakiś newralgicznych obszarów naszego systemu?

Jeśli odpowiadając sobie na te pytania okazuje się, że w zasadzie wystarczy nam wiedzieć, co się dzieje w kilku takich obszarach, to może wystarczy jak od czasu do czasu ktoś je odwiedzi i sprawdzi co słychać, bądź ustawi sobie odpowiedni monitoring.

Co jeśli okaże się, że tych punktów jest sporo? No cóż, czasem trzeba się nachodzić i to w dodatku dość regularnie. Dopiero tutaj jest miejsce, żeby powiedzieć sobie : “a może by tak te spacery zautomatyzować...”.

Oczywiście nie chodzi tutaj, aby z automatyzacją czekać aż przestaniemy ogarniać. Chodzi o to, że jeśli na początku projektu mamy podejrzenia, że taki moment może nadejść, to podejmujemy odpowiednie środki zapobiegawcze.
W końcu mamy jakąś road mapę, a co najmniej backlog wypełniony ficzerami na ileś miesięcy do przodu i możemy się zorientować, co będzie się działo za 2-3 miesiące, czy nawet pół roku.

Jeśli dobrze sobie wszystko przemyślimy na samym początku i małymi kroczkami będziemy przygotowywać się pod ten moment, w którym system zacznie nas przerastać, to w takim momencie możemy mieć już gotowe środowisko i być może nawet kilka testów, które już od jakiegoś czasu są uruchamiane. Dodanie kolejnych będzie stosunkowo proste, tak samo jak i odzyskanie kontroli nad naszym systemem.

A jeśli sobie tego nie przemyślimy? Stanie się to co zwykle się dzieje. Wyjście z problemu będzie kosztowne, czasochłonne, bolesne, bądź niemożliwe.

Wpis #87 Sztuka pracy z lokatorami i selektorami

Automatyzacja testów, gdzie eventy generuje się na poziomie interfejsu graficznego użytkownika wydaje się być pozornie prosta. Z mojego punktu widzenia jest tak, ponieważ kod dla wykonywania tych eventów nie jest jakiś super skomplikowany i wypakowany algorytmami. Jest wręcz banalnie prosty i pozbawiony złożoności:

- Wejdź na stronę
- Kliknij tu
- Kliknij tam
- Zapisz

Nie znaczy to, że automatyzacja jest faktycznie prosta. Wyzwania czekają w innych miejscach.

Jednym z takich wyzwań jest budowanie dobrych lokatorów i selektorów.

Najpierw jednak, czym są lokatory, a czym są selektory. Co któreś źródło w Internecie podaje, że jest to jedno i to samo. Dla mnie jest to absurdalne nieporozumienie.

Żeby wykonać event symulujący zachowanie użytkownika musimy programatycznie zlokalizować element na stronie, aby następnie wykonać na nim jakąś akcję. W tej lokalizacji mamy dwie składowe, które rządzą się swoimi prawami i tymi składowymi są właśnie lokatory i selektory.

Lokatory - to sposób, w jaki będziemy lokalizować elementy na stronie. Szczególne rozróżnienie ma tutaj miejsce w selenium. Jest to sposób, w jaki będziemy szukać elementu, czy np. skorzystamy z xpath, css, nazwy klasy bądź id, czy będziemy szukać po jednym, czy po wielu atrybutach, a może będziemy szukać po tekście elementu. To, jaki lokator dobierzemy będzie zależało od tego, jak zbudowany jest frontend. Przede wszystkim chodzi o to, aby osiągnąć nasz cel, który brzmi: znajdź mi element, znajdź mi go zawsze, bez względu na to w jakim frontend jest stanie i jak się zmienia. Frontend może się zmieniać w taki sposób, że programista doda kolejne funkcjonalności do widoku, na którym już czegoś szukamy, bądź nasze eventy dodadzą te elementy. Jedno i drugie bardzo często może nam popsuć lokator, z którego skorzystaliśmy, więc musimy stosować strategie, które dają nam stabilność i powtarzalność.

Przykładowe lokatory dla cypress'a:

```
cy.get()
cy.find()
cy.contains()
cy.first()
cy.siblings()
```

Cypress pozwala też chain'ować te lokatory i na tym właśnie polega budowanie.

Selectory - selektor to nie tyle sposób, w jaki szukamy, a konkretnie ścieżka, z której korzystamy, aby odnaleźć dany element. Wszystkie elementy frontowe są zbudowane na DOM'ie, czyli strukturze tag'ów (div, span, td etc.). Struktura ta polega na poziomach i zagnieżdżeniach. Selektor to nic innego jak string, w którym podajemy ścieżkę czyli: div > span > td[atrybut="test"]. Selektory musimy budować

tak, aby odnajdywać elementy unikalne. Szczególnym wyzwaniem jest widok, na którym jest mnóstwo powtarzających się elementów, a my musimy zrobić tak, aby z tych powtarzających się elementów wyróżnić ten konkretny, który nas interesuje. Tutaj też musimy uwzględniać zmiany, które są powodowane albo zmianami w kodzie, albo naszymi eventami.

Tak naprawdę budowanie skutecznej strategii szukania elementów to mix dobierania odpowiednich lokatorów i selektorów. Czasem jest tak, że spotykasz widok, na którym jest sporo powtarzających się elementów. Patrząc na DOM wszystkie elementy są niemal takie same i nie ma w nich niczego unikalnego, a my musimy uruchomić kreatywność, aby mimo tego odnaleźć właśnie ten jeden, na którym nam zależy. Czasem jest tak, że widok ma sporo złożonych komponentów, które zależą od siebie, a wręcz po wykonaniu eventu, np. click na jednym powoduje, że pojawia się drugi lub drugi zmienia swój stan, a my budując lokatory musimy się do tego dostosować. Co projekt to inne wyzwanie, co technologia to inny sposób tworzenia DOM'a, co programista to inny sposób opisywania elementów. Szukanie elementów na stronach to często miejsce, w którym się zatrzymujemy i musimy pogłówkować. Jasne, możemy polecieć po najmniejszej linii oporu i zbudować swoje lokator bez specjalnego zastanawiania się ale może to prędzej, czy później doprowadzić do failujących testów ponieważ:

- Frontend się zmienił i zburzył nam selektor, który sobie zbudowaliśmy.
- Doszły nowe case'y, które wymagają bardziej wyrafinowanych selektorów.

lokatory zbudowane przez nas opierają się na 'farcie' tzn. raz będzie działać, raz nie będzie działać (chodzi o specyfikę renderowania elementów i asynchroniczność js'a).

Jest jednak jedna rzecz, która może nam ogromnie ułatwić życie. Mając dostęp do frontu albo do fajnego frontenda

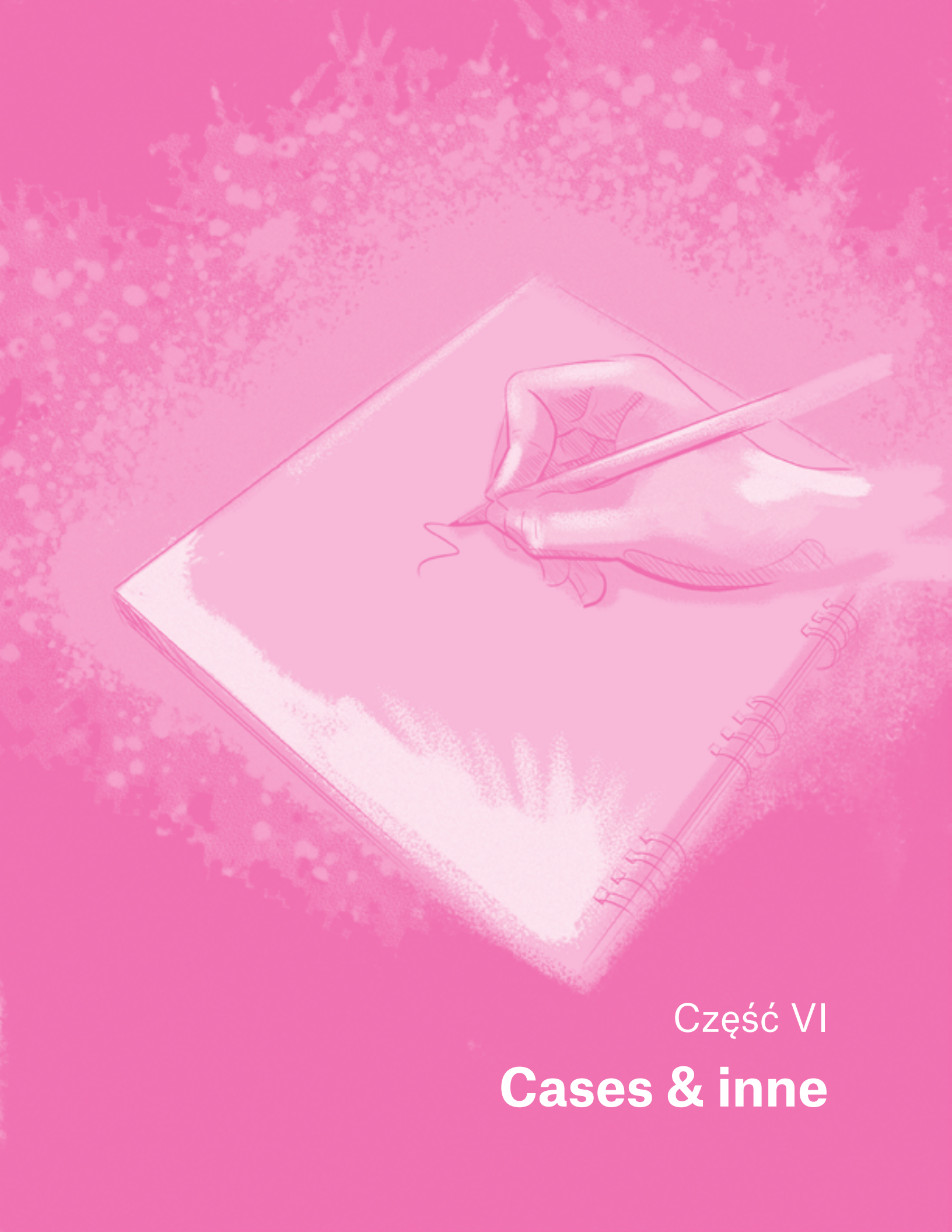
można zaplanować strategię dodawania atrybutów testowych do kluczowych elementów tak, aby te elementy identyfikować, a powtarzające elementy zindeksować. Jeśli tester ma ogarnięcie frontowe to jest w stanie sam sobie takie elementy dodawać. Tak też robiłem w poprzednim projekcie, ale zwykle zajmuje to sporo czasu. Można też pogadać z programistą, żeby dodawał takie elementy pisząc frontend, bo to jest naprawdę szybka sprawa.

Przykład dobrze zbudowanego elementu:

```
<button data-testid="audience-switch-button-0"
id="audience-form_groups_0_segments_0_exclude"
type="button" role="switch" aria-checked="false" class="ant
switch"><span class="ant-switch-inner"></span></button>
```

oraz lokator do niego:

```
cy.get(`button[data-testid="audience-switch-button
${index}]`)
```



Część VI

Cases & inne

Wpis #88 Z życia wzięte

Lubię jak życie podsuwa przykłady, które potwierdzają moje założenia co do testowania. W tym wpisie podam Wam przykład podejścia do automatyzacji.

Wczoraj rozwiązaliśmy w zespole jeden problem, który był stricte związany z nową wersją przeglądarki (chrome). Problem polegał na tym, że od którejś wersji chrome zmienił wartości dla ustawień defaultowych (nieistotne jakich konkretnie). Jeśli masz chrome'a zainstalowanego już jakiś czas, to Twoje ustawienia defaultowe będą miały inne wartości, niż ustawienia defaultowe u osób, które chrome'a zainstalują dziś. Mimo, że będziecie mieć tę samą wersję. Nam akurat przysporzyło to sporo problemów z zapisywaniem ciasteczek.

Co by było gdybyśmy zbyt mocno polegali na automatyzacji?

Scenariusz pierwszy: jeśli mielibyśmy automatyzację, która taki problem nam wykryła (chociaż nie wyobrażam sobie automatyzacji, która instaluje na świeżo przeglądarkę) to oznaczałoby, że mocno przegięliśmy z projektowaniem testów regresyjnych do automatyzacji. Jeśli mamy otestowane tego typu case'y, to nie chcę nawet myśleć o tym, jakie inne dziwne sytuacje również mamy otestowane. Pytanie ile czasu straciliśmy na pisanie, uruchamianie i utrzymywanie tego typu testów.

Scenariusz drugi, bardziej realny, nie mamy na to testów, ale mamy sporo innych, bo mocno polegamy na automatyzacji. Zbyt mocno, żeby podejmować się obszernych testów manualnych. Oczywiście o takim błędzie nikt nie wie, dopóki klient się nie zorientuje, że coś jest nie tak. Albo metryki pokażą nam, że coś jest nie tak. Jednak debugowanie takiego problemu na podstawie metryk może być okropnie trudne.

To jest tylko przykład, jest on dość specyficzny. Jednak problemy, które znajdujemy testując zwykle są specyficzne.

Rozsądna automatyzacja jest spoko, testowanie manualne jest równie spoko.

Jeszcze śmieszna dygresja, ostatnio jedna osoba zainteresowana zostaniem QA zapytała mnie, czy już automatyzuje, czy jeszcze nie. Czytając materiały w internetach i słuchając różnych ludzi odniosła wrażenie, że automatyzacja w byciu QA jest niczym awans społeczny i postanowiła, że ona też kiedyś chciała by automatyzować.

Wpis #89 Jak nie robić szybkich fixów

Teraz lesson learned. Co prawda to nie jest tak, że dopiero teraz odkryłem Amerykę,ale liczę na to, że pisząc tego feed'a zapamiętam sobie raz na zawsze, żeby dobrze testować swoje szybkie fixy.

Zakładam, że to nie tylko mój problem. Ile razy mieliście taki scenariusz?

Coś nie działa.

"O, wiem co nie działa! Zaraz to poprawię i wszystko będzie cacy".

Cyk, cyk, szybki fix, push, i sprawdzamy.

"O, tutaj jest więcej błędów...".

Jeśli feedback z takiego procesu jest szybki, to luzik. Ale jeśli czekacie 20 minut albo i dłużej na deployment, czy może jakiegoś joba, to trochę słabo.

Bardzo często jak robię szybkiego fixa to daje się nabrać na to, że to ten jeden jedyny problem. Żadnych już nie będzie i jeśli tylko go poprawię, to już wszystko będzie śmigać. Potem dostaje w pysk czterema kolejnymi zanim w końcu wszystko będzie faktycznie działać.

Co prawda zawsze mogę jeszcze sprawdzić, czy nawet nie tyle moje poprawki działają ale prócz tego, co poprawiałem nie ma więcej problemów, bo często trafia się tak, że jest ich więcej.

Czemu tak się dzieje? Pewnie powody są różne. Ostatnio spędzałem już kolejny dzień nad testowaniem JEDNEGO taska. Po drodze było sporo poprawek i niby już wszystko miało działać, byłem też zmęczony tym taskiem i chciałem go w końcu zamknąć. Dałem się więc nabrać swojemu optymizmowi, że błąd, który zepsuł mi całą zabawę w testowanie to ten jedyny i żadnych kolejnych już nie będzie.

Lekcja na dziś, lepiej testować coś jak najszybciej się da, zamiast kręcić się w kółku: poprawka - czekanie - fail - poprawka - czekanie - kolejny fail.

Wpis #90 Jak ulepszyć swój warsztat testerski

Mając na myśli testowanie nie chodzi mi o tzw. przeklikanie wymagań, czy kryteriów akceptacji, a chodzi mi o proces nauki systemu, eksperymentowania, obserwacji i wyciągania odpowiednich wniosków.

Aby rozwijać swój warsztat możemy studiować różne poziomy, podejścia, czy metody testowania. Jednak jest ich ograniczona ilość i po pewnym czasie okaże się, że właściwie każda kolejna ‘metoda’ to tak naprawdę ubranie w inne słowa tego, co już znamy. Możemy też poznawać różne techniki i narzędzia wspierające projektowanie przypadków testowych, czy prowadzenie eksploracji. Tych technik i narzędzi też jest ograniczona ilość i czasem nawet będzie lepiej, gdy nie będziemy przytaczać się ich mnogością. To co możemy rozwijać w sposób nieograniczony to umiejętności poznawcze, które stanowią fundament dobrego testowania.

Na te umiejętności składają się:

- Pamięć
- Uczenie się
- Logiczne myślenie
- Kojarzenie faktów
- Rozpoznawanie wzorców
- Kategoryzowanie rzeczywistości
- Podejmowanie decyzji
- Wyciąganie wniosków
- Samoświadomość i autorefleksja

Im lepiej znamy system, tym większa szansa, że nasze testowanie będzie trafne (z ang. accurate). Do tego przyda nam się umiejętność szybkiego i efektywnego uczenia się, szczególnie jeśli chodzi o wiedzę techniczną. Prócz poznania systemu, z którym pracujemy dobrze byłoby go też rozumieć. Na sam koniec będzie świetnie, gdy cała ta wiedza zostanie w naszej pamięci na dłuższy czas, a my z łatwością będziemy w stanie do niej wracać. Testując niejednokrotnie zdarza się, że potrzebujemy jakiegoś zaplecza wiedzy domenowej, czy wiedzy o systemie, aby poznać i zrozumieć nowo wprowadzaną funkcjonalność. Ponadto to, co już wiemy o systemie pozwoli nam lepiej taką funkcjonalność ocenić, o ile jesteśmy w stanie kojarzyć fakty i skutecznie odnosić się do tego, co już wiemy o systemie.

Eksperymentując z systemem, obserwując co się dzieje i wyciągając wnioski robimy to korzystając z naszych przekonań, uprzedzeń, czy ogólnie mówiąc "z naszej perspektywy". Dlatego przydaje się samoświadomość i autorefleksja. Część naszej perspektywy jest przydatna ale w pewnym momencie musimy od niej odejść i skorzystać z innej. To daje nam szerokie spektrum dla projektowania przypadków testowych. Dodatkowo musimy wystrzegać się naszych uprzedzeń i założeń, które mogą wprowadzić nas w różne pułapki myślowe. Żeby to zrobić musimy najpierw zdać sobie sprawę z ich istnienia.

Korzystając z technik testerskich, takich jak np. analiza wartości brzegowych będziemy w stanie sprawdzić... wartości brzegowe. Super. Ile razy zdarzyło wam się mieć problem z wartościami brzegowymi ? Niestety o ile te techniki brzmią dobrze na papierze, o tyle w rzeczywistości stosowanie ich to metody infantylne, tzn. że nie dojrzeliśmy jeszcze do tego, aby zrozumieć gdzie pojawiają się problemy. Dlatego umiejętność logicznego myślenia, kojarzenia faktów, rozpoznawania wzorców, czy kategoryzowania rzeczywistości to właśnie te umiejętności, które pozwolą nam realnie wycisnąć wartość z naszego testowania.

Każdy z nas lubi techniki, czy narzędzia (może jakąś nową biblioteczkę w js'ie), które rzekomo rozwiązują nasze problemy. Może dlatego, że wprowadzanie i korzystanie z narzędzi wydaje się prostsze niż praca nad sobą. Jednak to nie narzędzia rozwiązują problemy, a umiejętne korzystanie z nich. W tym właśnie pomagają nam umiejętności poznawcze. Dzięki nim poprawnie ocenimy kiedy i jak korzystać z tych narzędzi, jak je wdrożyć i jak z nimi pracować. Jeśli chodzi o projektowanie przypadków testowych już jest troszkę ciężej, bo nie ma gotowych narzędzi, które skutecznie zrobią to za nas. W pewnym momencie to umiejętności poznawcze stają się naszym narzędziem pracy.

Wpis #91 Dlaczego tester to nie klikacz

To, że testerzy postrzegani są jako klikacze chyba nikogo nie dziwi. Zapewne część osób nawet pracowała z tego rodzaju testerami. O co chodzi z tym klikaniem?

Generalnie klikacz jest postrzegany ...źle. Głównie dlatego, że sama nazwa umniejsza testerom ich wiedzę i doświadczenie. Sprowadza tę rolę do powtarzalnej bezmyślnej czynności. Problem w tym, że rola postrzegana

w ten sposób nie jest żadną abstrakcją. Nie mam 20 lat doświadczenia i nie przeżyłem rewolucji na rynku IT, żeby podawać informacje z pierwszej ręki. Tyle, co udało mi się wysłuchać od starszych i mądrzejszych kolegów po fachu. Czy to w jakiejś rozmowie, czy słuchając ich prelekcji pozwala mi wnioskować, że pewnego czasu, ileś lat temu wstecz, grono firm postanowiło zapewniać jakość poprzez testowanie manualne wykonywane przez armię testerów przeklikujących scenariusze, które dostawali. Taka praca faktycznie była krótko mówiąc nieskalana myśleniem. Zatrudniano osoby nietechniczne, niewykwalifikowane, bo nie było takiej potrzeby.

Praca jak praca. Nie wiem jak w tamtych czasach postrzegało się testerów, ale wciąż nie widzę powodu, aby wytykać im specyfikę ich pracy. Wnioskuję jednak, że negatywne postrzeganie testerów rozpoczęło się wraz z pojawieniem się nowych metodologii wytwarzania oprogramowania, które wymuszały na testerach współpracę z resztą zespołu. W tym momencie tester, który do tej pory nauczony był przeklikiwać przygotowane przez kogoś scenariusze testowe na przygotowanym przez kogoś środowisku, używając przygotowanych przez kogoś danych testowych zaczynał być problematyczny. Brak umiejętności technicznych spowodowały, że taki tester stawał się mało użyteczny, a wręcz problematyczny w utrzymaniu i współpracy. W momencie, gdy zespoły przechodziły transformację, działy testerskie na siłę próbowały pozostać przy swoich starych sposobach i współpraca zespół - testerzy zaczęła być nieefektywna.

To wszystko to tylko domniemania. Być może tak dokładnie było, być może było podobnie. Na pewno problem testera “klikacza” jest wciąż obecny, gdy tester przychodzi do zespołu agile’owego licząc na to, że będzie klikał albo firma zatrudnia właśnie takich testerów, żeby klikali... w zespole zwinnym.

Do ognia oliwy dolewają firmy szkoleniowe i pojedynczy konsultanci, którzy tak jak pisałem wcześniej, wykorzystują nieaktualne już postrzeganie testerów, aby przekonać rzesze

chętnych na przekwalifikowanie się osób, że testowanie jest dla każdego. “Mamo testuj” i te sprawy (tylko nie zapomnij zapłacić nam 5k czy 10k). Fakt, dalej są firmy, które zatrudniają i wykorzystują testerów w ten sposób. Super, nie widzę w tym problemu. Widzę problem w tym, że na rynku jest mocna dezinformacja i testująca mama kończy jako QA w zespole. Swoją drogą, żeby nie zostać źle odebranym - nie widzę problemu, gdy osoba się przebranżawia. Sam to zrobiłem mając niewielką wiedzę techniczną. Nigdy jednak nie wierzyłem w to, że to będzie łatwa praca za worki pieniędzy i od początku byłem przygotowany na sporo nauki.

Ponadto są też testerzy, którzy dalej nie skumali, że świat się zmienił i oni też powinni. Takie osoby aplikują na stanowiska QA, mają właśnie tego typu klikające testerskie doświadczenie i takie też osoby promują testowanie w taki właśnie sposób.

Na sam koniec dodam, że są osoby, które nie mając zielonego pojęcia czym zajmuje się tester łykają informacje o klikaczach i przekazują ją dalej. Co gorsza nawet przekonują innych, że na testera to łatwo się dostać, nic nie trzeba umieć i zachęcają ich żeby spróbowali - byłem świadkiem takiej rozmowy. To raz, a dwa - przykro mi to rzec ale osoba, którą znam przysłała do mnie chętnego na testera, żebym przekonał go, że warto i że nic nie trzeba umieć - wystarczy tylko aplikować.

Wpis #92 Mapa rozwoju testera

Całkiem niedawno trafiłem na bardzo fajną mapę, która pozwala planować swój testerski rozwój. Pomyślałem, że warto się nią podzielić: http://thetestingmap.org. Jest świetnie zrobiona, tzn. oprócz tego, że jest atrakcyjna wizualnie, zawiera w sobie sporo kategorii i podaje również źródła.

Zamiast wypisywać co znajduje się na tej mapie zachęcam, żeby każdy zerknął sam, dlatego ten feed jest króciutki. Na pewno część z tych umiejętności nie jest dedykowana tylko dla testerów.

Wpis #93 Czy Whole Team approach wspiera refaktoryzacje?

Całkiem niedawno na blogu firmowym Clearcode pojawił się artykuł, w którym opisałem próbę wprowadzenia whole team approach w moim zespole. Konkretniej opisałem korzyści wynikające z zaangażowania całego zespołu w testowanie tego, nad czym pracujemy.

Dzisiaj rano w książce “Succeeding with Agile - Mike Cohn”, natrafiłem na fragment, który mówił o tym jaką istotę pełni refaktoring w zespołach zwinnych. Zgodnie z założeniami refaktoryzacje są potrzebne, a wręcz są nieustanną częścią pracy z długoterminowymi projektami. Ciężko się z tym nie zgodzić. Skoro refaktoryzacja jest tak istotna, to nasza zdolność do refaktoryzacji może okazać się równie ważna, co umiejętność projektowania i implementowania nowych rzeczy.

We wspomnianej książce autor mówi również o koncepcji zbiorowej własności kodu (collective ownership), która dzięki pracy z nie-swoim kodem przyczynia się do powstawania czystego i (stosunkowo) bezbłędnego kodu. Założenia są takie:

- Mając świadomość, że kod, który powstaje będzie za chwilę reużywany lub rozszerzany przez kogoś innego, staramy się bardziej. Nie chcemy, aby ktoś za chwile znalazł błędy w naszej pracy i w grę wchodzi ambicja, czy poczucie wartości.

- Nawet jeśli pojawią się błędy i ‘nieczystości’ w cudzym kodzie, zdolność do refaktoryzacji i własność zbiorowa sprawi, że ktoś inny to poprawi.

Oczywiście warunek jest taki, że wszyscy zabierający się za pracę z obcym kodem wiedzą, z czym mają do czynienia.

W ten właśnie sposób objawiła się kolejna zaleta whole team approach, o której pisząc artykuł nie pomyślałem. Zaangażowanie całego zespołu w testowanie to nauka tego, jak działają różne części systemu. Przeważnie te, nad którymi sami nie pracowaliśmy. W związku z tym wydaje się, że zdolność do refaktoryzacji rośnie wraz z regularnym nabywaniem takiej wiedzy.

Wpis #94 Komunikacja o niskiej i wysokiej roli kontekstu

W tym feedzie z pozoru może temat troszkę nietypowy ale jak najbardziej właściwy dla problemów komunikacyjnych, które pojawiają się w zespołach i które wpływają na cały obszar developmentu projektów (i na błędy).

Komunikacja o niskiej i wysokiej roli kontekstu jest częścią modelu komunikacyjnego zaproponowanego przez Edwarda Halla. Wiele to pewnie nikomu nie mówi. Chciałem tylko podkreślić, że ktoś nad tym prowadził prace naukowe i nie jest to fanaberia.

Według tego modelu komunikujemy się na dwa sposoby, tak jak w tytule, a każdy ze sposobów można w większym lub mniejszym stopniu zauważyć u poszczególnych członków zespołu. Wyróżnia się nawet społeczności lub społeczeństwa, które charakteryzują się większym udziałem jednego z tych modeli.

Komunikacja o niskiej roli kontekstu polega na tym, że wymieniając się informacjami przekazujemy możliwie największą ilość faktów i informacji. Komunikacja jest jasna i bezpośrednia, a odbiorca otrzymuje wszystko, co niezbędne, aby w pełni zrozumieć komunikat.

Przeciwnie stawia sprawę sposób komunikacji o wysokiej roli kontekstu. W takiej komunikacji mniejszą rolę odgrywa przekazywanie pełnych informacji. Tutaj dominuje język ciała, pozycja społeczna, wiedza kontekstowa i wiele innych. Tym samym komunikacja staje się niejasna, niebezpośrednia i mniej dosłowna.

Co więcej często spotyka się również sposób komunikacji o ekstremalnie wysokiej roli kontekstu, w którym nadawca przekazuje minimum faktów i informacji ale również informacji niewerbalnej, mocno bazując na własnym kontekście, którego brak u adresata.

Już na pierwszy rzut oka klaruje się prosty wniosek. W pracy projektowej potrzebujemy komunikacji o niskiej roli kontekstu, ponieważ to ona wspiera skuteczne dowożenie projektów do końca. Pomaga nam zrozumieć klienta, użytkownika, product ownera. Pomaga formułować lepsze user stories, czy wymagania, pomaga skuteczniej przekazywać sobie informacje pomiędzy członkami zespołu i finalnie pomaga unikać fakapów na różnych etapach pracy projektowej.

Jeśli każdy z nas zastanowi się nad swoim sposobem komunikacji oraz nad sposobami komunikacji osób, z którymi pracujemy na co dzień okaże się, że zarówno u siebie, jak i u innych możemy dostrzec tendencje do komunikacji o wysokiej roli kontekstu. U jednych jest silniejsza niż u innych ale tak samo u danej osoby, danego dnia przejawia się silniej niż w przypadku innego dnia.

Zdając sobie z tego sprawę możemy zarówno pracować nad tym, w jaki sposób przekazujemy informacje (rozmawiając, czy tworząc dokumenty) ale także nad tym, w jaki sposób te informacje odbieramy wiedząc, że nasz rozmówca może nie

mówić nam wszystkiego, choć to co mówi wydaje się mieć sens.

Wpis #95 Testowanie to biznes informacyjny

Jak zapytamy co robi tester, to odpowiemy, że testuje. Jak zapytamy po co testuje, to zwykle odpowiemy, że testuje, bo szuka bugów. To prawda! Ale tylko częściowo...

Tester nie testuje tylko dlatego, że szuka bugów. Tester przede wszystkim szuka informacji.

System ma defekty - to informacja, z którą możemy coś zrobić. System nie ma defektów - nigdy tak nie powiemy. Powiemy natomiast, że nie znaleźliśmy żadnych defektów grzebiąc w danej funkcjonalności albo wykonując jakieś operacje. To też informacja, z którą możemy coś zrobić, np. Deployment.

Planując więc testowanie lub strategie testową możemy podejść do tego na zasadach zbierania informacji. Musimy uzmysłowić sobie po co zbieramy informacje i dla kogo. Patrząc na to inaczej, musimy zdać sobie sprawę jakich informacji nie chcemy zbierać (czego nie chcemy testować) oraz dla kogo tych informacji nie chcemy zbierać (jakie informacje o systemie są nieistotne). Jeśli już pozyskamy informacje, np. znajdziemy bugi, to znając naszych interesariuszy i ich potrzeby możemy ocenić, czy ta informacja jest cenna, czy nie oraz czy mamy z nią coś zrobić.

Wpis #96 Wyzwania jakościowe pracy przy data pipelines

Mój zespół otrzymał nowe wyzwanie (nowy projekt), który będzie w dużej mierze polegał na budowaniu data lake'a i etl'i pozwalających na transformowanie danych pochodzących z wielu różnych źródeł.

W ramach przygotowań do projektu ostatnimi czasu robiłem sporo riserczu na temat tego, z jakimi problemami można się spotkać pracując przy takich rozwiązaniach.

W trakcie natknąłem się na całkiem fajny godzinny podcast.

Całkiem niedawno powstało narzędzie wspierające analityków danych i generalnie wszystkich, którzy pracują z transformacjami - https://greatexpectations.io. Swoją drogą to klient podrzucił nam tę sugestię, a po sprawdzeniu co to jest okazało się, że jest to jedno (jak nie jedyne) z seksowniejszych rozwiązań, w którym odchodzimy od testowania kodu, a skupiamy się na testowaniu... danych.

W podccaście bierze udział autor framework'a i opowiada o wyzwaniach w pracy z data pipelines i o tym, co skłoniło go do napisania Great Expectations.

Polecam dla zainteresowanych. Sam jeszcze chciałbym pogrzebać w tym temacie. Zobaczymy, czy staniemy przed podobnymi wyzwaniami, które pojawiają się w podcaście. W razie czego, będziemy mieli na horyzoncie jakieś wsparcie.

Wpis #97 Dlaczego jakość oprogramowania jest trudna

Procesy zapewniania jakości, czy kontroli jakości mają swój początek w procesach produkcyjnych, ponieważ są one starsze niż tworzenie oprogramowania. W takim przypadku jakość można było zrozumieć trochę prościej. Co roku produkowało się coś, co miało spełniać jakieś z góry określone kryteria. Proces ten był (i jest) powtarzalny i tak naprawdę jakość była definiowana jako brak odchyleń od standardu, czyli produktu wytwarzanego w wielu kopiach.

Z oprogramowaniem jest troszkę ciężej, ponieważ każdy tworzony system jest na swój sposób unikatowy nawet, jeśli realizuje podobne potrzeby biznesowe. Co więcej tworzenie oprogramowania to nie jest kwestia produkcji, a bardziej kwestia rozwiązywania problemów.

Można by powiedzieć, że przecież mamy wymagania, więc jakość to będzie stopień odchylenia od tych wymagań. Ale to nie takie proste. Ponieważ wymagania skupiają się tylko na realizacji potrzeby biznesowej i to w sposób mocno abstrakcyjny. A przecież ta potrzeba może być realizowana na różne sposoby, a interesariuszy może być wielu. Ponadto sama potrzeba biznesowa i proces jej realizacji zwykle niesie za sobą całą listę nieznanych problemów, które pojawiają się dopiero w konkretnym czasie.

Biorąc pod uwagę tylko wymagania możemy zbudować system, który będzie je spełniać, a wciąż będzie niskiej jakości. Więć jak to tak? Dlatego, że prócz tego, że system realizuje te wymagania może on być przykry w użytkowaniu, może być zawodny, może być nieprecyzyjny, może być niewydajny, może być zbyt drogi w utrzymaniu, może być zbyt trudny w rozszerzaniu.

Dlatego nie da się łatwo określić jakości. Łatwiej jest na bieżąco kalibrować definicje jakości zgodnie z tym co wiemy,

a nasza wiedza jak wiadomo ewoluuje wraz z cyklem życia projektu.

Wpis #98 Testerzy kontra agile manifesto

Kilka dni temu (z jakiegoś powodu) ponownie przeglądałem sobie agile manifesto i jego 12 zasad. Cały manifest skupia się na dowożeniu produktów i utrzymywaniu zadowolonych interesariuszy. W zasadzie doszło do mnie, że nie ma tam nic, co by promowało jakość.

Istnieją dwie zasady, które mogłyby traktować o jakości, ale tego nie robią:

Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

Working software is the primary measure of progress.

Working software - hmm. Problem w tym, że określenie oprogramowania jako działające jest strasznie... nieprofesjonalne. Szczególnie z perspektywy ludzi zajmujących się jakością. Od czasu powstania manifestu napisano mnóstwo artykułów, książek, nagrano wiele podcastów i udzielono wiele prelekcji na temat tego, czym jest i jak definiować jakość w projekcie, a dla manifestu jakość to po prostu działające oprogramowanie. Czyli: działa? Działa... po co drążyć temat.

Być może trochę się przyczepiłem do tego przymiotnika, ale właśnie chodzi o to, że ten przymiotnik to jest to, co totalnie ignoruje całe zagadnienie jakości i na temat czego powstaje wiele materiałów, które próbują wyprowadzić z błędu osoby uważające, że jeśli coś działa to znaczy, że jesteśmy we właściwym miejscu.

Całe szczęście powstało kilka podobnych przykazań traktujących konkretnie o jakości w świecie agile i nazywają się one Modern Testing Principles. Zostały sporządzone przez dwóch programistów-testerów pracujących przez większość swojej kariery dla Microsoftu i, jak sami twierdzą, oni tego nie stworzyli, oni podjęli próbę nazwania tego, co wszyscy już rozkminili dawno temu.

Również i ja chciałbym dołożyć cegiełkę i popchnąć ten temat dalej, dlatego zachęcam do zerknięcia na te zasady.

Wpis #99 Pipelines debt

W ramach ostatniego riserczu dotyczącego data lake, transformacji danych i podobnych rzeczy, natrafiłem na pojęcie **pipeline debt**, więc pomyślałem, że zwrócę na nie tutaj uwagę.

W obszarze data analytics, gdzie nałogowo korzysta się z big data, transformacji danych i modeli do machine learning, prócz poprawnie napisanego kodu pojawia się dodatkowe wyzwanie, jakim jest jakość danych. Kod kodem ale jeśli mamy słabe dane to i wyniki naszych transformacji i predykcji będą słabe.

Transformacje danych mogą odbywać się na wielu poziomach. Każdy poziom może transformować dane na swój sposób i przekazywać go do dalszych transformacji. Może się tak zdarzyć, że nasze transformacje, czy tzw. czyszczenie danych nie robi tego, czego oczekujemy. Nie jest to łatwe do zauważenia, szczególnie kiedy złożoność naszego kodu rośnie razem z ilością transformacji. Pipeline debt dokładnie określa sytuacje, w której mamy jakiś kod (legacy), który coś ma robić ale w sumie nie do końca wiemy co konkretnie, ani nie mamy na to dobrych testów.

Rezultatem takiego problemu jest wynik predykcji, który budzi nasze wątpliwości. Z tego, co czytałem debugowanie takich problemów jest mega trudne, ponieważ nigdy nie

wiadomo gdzie szukać problemu. Tak więc wszystko jest dobrze, jak jest dobrze. Ale jak jest niedobrze, to musimy szukać, na którym poziomie mamy problem i wyniki nie są takie jak być powinny.

Wpis #100 Cztery umiejętności, które przydadzą Ci się w roli QA

Poprzez cztery umiejętności mam na myśli umiejętności:

- Kierownicze
- Miękkie
- Analityczne
- Techniczne

Rozwijanie umiejętności kierowniczych pomaga QA planować, estymować i koordynować cały proces zapewniania jakości włącznie z procesem testowania.

Tworzenie strategii testów, planowanie testowalności, dobieranie narzędzi, modyfikowanie procesu wytwarzania oprogramowania tak, aby jego wynikiem była wysoka jakość. To tylko kilka z wielu możliwych aktywności. QA zarówno kieruje tymi czynnościami, jak i bierze udział w ich wykonywaniu, tzw. hands-on.

Umiejętności miękkie także przydają się w byciu QA (i nie tylko). QA sam jakości nie zapewni. Potrzebuje całego zespołu, który się w ten proces zaangażuje. Jeśli chcemy zmodyfikować nasz proces, wdrożyć jakieś narzędzie, podjąć się jakichś praktyk (wszystko w celu podnoszenia jakości), fajnie jest umieć się dogadać z zespołem i znaleźć wspólny grunt.

Istnieje taka rola jak analityk testów, który patrząc na wymagania, planowaną architekturę i wszystkie inne możliwe wskazówki próbuje zaplanować proces testowy i zaprojektować test case'y, jeszcze zanim powstanie pierwsza linijka kodu. QA robi zasadniczo to samo plus wiele innych, taki człowiek orkiestra (zupełnie jak nasz PM/PO/Scrum Master itd.) Analiza to podstawa. Bez tego nie zrozumiemy wymagań, architektury, czy implementacji.

No i na koniec umiejętności techniczne. Tutaj zwykle jest spór, czy tester powinien być techniczny, czy nie. Spotkałem się nawet z dywagacjami na temat tego co to znaczy, że tester jest techniczny i kiedy technicznym się staje. Moim zdaniem umiejętności techniczne, tzn. umiejętności szeroko pojętego programowania na poziomie junior'a, bardzo się przydają. Przynajmniej mi (nie twierdzę, że jestem juniorem ale do tego zmierzam). Gdy rozumiem architekturę, rozwiązania techniczne i specyfikę dobieranych narzędzi wiem, że lepiej projektuje swoje testy, nie robię tego w ciemno, nie opieram się tylko na intuicji. Ponadto łatwiej jest mi się poruszać po środowiskach, setupować potrzebne komponenty do testowania, automatyzować, nawet dogadywać się z programistami, a czasem nawet coś zdebugować.

Wpis #101 Wąska specjalizacja czy obszerna generalizacja?

W zasadzie w każdym obszarze: managerskim, jakościowym, czy developerskim istnieją wyspecjalizowane role. Czasem narzekamy sobie, że musimy robić różne rzeczy, a gdzie indziej zatrudnia się od tego konkretnych ludzi, np. architekta baz danych, testera wydajnościowego, product owner'a itd.

Myślę, że potrzeba specjalizacji jest zależna od organizacji. W jednej będziemy potrzebowali specjalistów z każdej dziedziny, w innej ludzi, którzy ogarną kilka rzeczy w ramach jednej roli.

Dla przykładu patrząc na nasze zespoły mam wrażenie, że biorąc pod uwagę projekty, z jakimi pracujemy, ich wielkość, czas trwania itd. bardziej sprawdzają się osoby, które potrafią zarówno kodzić na frontendzie, jak i backendzie, czy zarówno planować jakość, jak i brudzić swoje ręce testowaniem manualnym.

Ciekawy jestem tylko czy fakt, że są osoby, które zmierzają w kierunku bardziej generalnym niż specjalistycznym, to z waszego punktu widzenia zaleta czy wada? Czy jest jeden kierunek, w którym powinniśmy zmierzać? Na przykład generalizacja? No i inna sprawa, kto jest więcej warty na ten moment, komu powinniśmy więcej płacić?

Wpis #102 Test selekcji Wasona

Jest to z jednej strony zagadka logiczna, a z drugiej dobry dowód na to, że selektywne wybieranie test case'ów ma sens.

Zagadka polega na tym, że mamy przed sobą cztery karty leżące na stole. Każda karta ma po jednej stronie liczbę, a po drugiej kolor. Dwie karty położone są liczbami do gór,y a dwie kolorami do góry. Karty z cyframi pokazują cyfry 3 i 8, a karty z kolorami pokazują kolory: czerwony i brązowy.

Założenie: jeśli karta zawiera parzystą liczbę z jednej strony, to jej druga strona jest czerwona.

Pytanie: którą kartę (z czterech możliwych) musimy koniecznie odwrócić, aby udowodnić, powyższe założenie?

Spróbujcie najpierw sami odgadnąć!

Wiadomo, że aby sprawdzić to założenie można odwrócić wszystkie karty. A co, jeśli tych kart byłoby 1k? Będziemy odwracali wszystkie? A co, jeśli przypadków testowych jest 300? Będziemy wykonywać wszystkie?

Rozwiązanie: po pierwsze musimy sprawdzić, czy założenie jest prawdziwe. W tym celu odwracamy kartę z cyfrą 8. Ona nam to potwierdzi. Jeśli okaże się, że pod 8 kryje się kolor brązowy, możemy skończyć testować, mamy błąd... no i kawa.

Jednak na tym jeszcze nie koniec. Jeśli trójka okaże się mieć kolor czerwony to nic się nie stanie, założenie wciąż jest ważne. Jeśli kolor czerwony, będzie mieć liczbę nieparzystą też będzie ok, bo założenie mówi wyraźnie, że liczby parzyste mają mieć kolor czerwony. Liczby nieparzyste - whateva.

Pozostaje więc kolor brązowy. Co, jeśli się okaże, że pod kolorem brązowym będziemy mieć liczbę parzystą? Wtedy założenie nie zostaje spełnione. Testowanie negatywne w najczystszej postaci.

Wnioski: testując weryfikujemy pewne założenia. Założenia można potwierdzić albo obalić. Aby to zrobić jak widać warto przemyśleć sobie test case'y, zamiast wykonywać wszystko jak leci. Jeśli test case'ów jest niewiele i aby sprawdzić wszystko potrzeba nam 10 minut więcej, to może lepiej nie myśleć? Jeśli test case'ów jest sporo, a każdy z nich wymaga przygotowania innych danych i generalnie wykonanie wszystkich to kwestia kolejnych kilku dodatkowych godzin? Może nie warto skoro te test case'y dadzą nam takie same wyniki, jak i inne test case'y albo nie są w stanie obalić założenia.

Wpis #103 W świecie wątpliwości

Sceptycyzm to niewątpliwa zaleta testera. Gdyby wszystkiemu i wszystkim można było ufać na słowo nie mielibyśmy bugów. Nie twierdzę, że dokumentacja, user stories, czy to, co mówi do mnie kolega/koleżanka z zespołu to same kłamstwa. Chodzi o to, że nie zawsze wszystko jest poprawne i bycie sceptycznym pomaga te niepoprawności odnajdywać.

A więc tester (czy też QA) w świecie wątpliwości. Będąc testerem zawsze możemy napotkać wątpliwości związane z:

- Informacjami otrzymanymi od klienta.
- Wymaganiami.
- Dokumentacją.
- Poprawnością komunikacji.
- Informacjami otrzymanymi od kogokolwiek z zespołu.
- Kompletnością napisanych testów.
- Poprawnością danych testowych (nawet swoich własnych).
- Zbieranym metrykom dotyczącym tego jak system działa.

...i wiele innych?

Jako dowód załóżmy, że ktoś z zespołu mówi testerowi na czym polega dana funkcjonalność i jakie wiążą się z nią ryzyka (co warto by sprawdzić). Jaką mamy pewność, że ta osoba jest w 100% nieomylna, że wszystko przekazała poprawnie i że o niczym nie zapomniała? Zwykle nie mamy takiej pewności. Co by się stało, jeśli faktycznie coś zostało przekazane niepoprawnie albo pominięte? Będąc sceptycznym i tak staralibyśmy się zweryfikować te informacje i wyjść ponad nie. Będąc ufnym łyknęlibyśmy

to, co jest i najprawdopodobniej pominęli ważny błąd (pod warunkiem, że on gdzieś tam jest).

Dlatego bycie sceptycznym popłaca. Ufni możemy być na kawie, czy przy lunchu.

Wpis #104 Brain teasers - rozwijaj się jako tester

Czytając nawet wybiórczo feedy możecie zauważyć, że fundamentem testowania nie są narzędzia, templatki, checklisty, konkretne techniki itd. Fundamentem testowania jest umiejętne myślenie. Tzn. myślenie kreatywne, analityczne, systemowe, logiczne, czy krytyczne. Każdy rodzaj myślenia determinuje to, jak dobre będzie nasze testowanie począwszy od dokumentacji, a skończywszy na linijkach kodu.

Oczywiście wiedza narzędziowa, domenowa i techniczna to pozostałe elementy układanki, niemniej myślenie to podstawa do odnajdywania 'niedoskonałości'.

Oczywiście nie tyczy się to tylko testerów, a doskonałym przykładem jest coding dojo, prowadzone przez naszych rycerzy pracowników, którego motywem przewodnim jest trenowanie rozwiązywania problemów. Testerzy mają swoje katy i teraz chciałbym zachęcić do wzięcia udziału w jednej z nich.

Dzisiejsza kata bardzo dobrze prezentuje na czym polega analizowanie wymagań i dokumentacji w poszukiwaniu wcześniej wspomnianych 'niedoskonałości'.

Kata przedstawia się następująco (będzie po ang. ponieważ słowo married jest jednoznaczne gdzie zamężna != żonaty):

Jack is looking at Anne, but Anne is looking at George. Jack is married, but George is not. Is a married person looking at an unmarried person?

1. Yes.

2. No.

3. Cannot be determined.

Spróbujcie najpierw rozwiązać to zadanie sami.

Udało się? Jeszcze nie? Próbujcie dalej.

Czy na pewno chcesz już zobaczyć odpowiedź?

Mam nadzieję, że próbowałaś/próbowałeś :)

Ok no to rozwiązanie i analiza.

W rozwiązaniu pomaga myślenie logiczne oraz ‘zamodelowanie’ sobie problemu (dwie pieczenie na jednym ogniu).

Jack(żonaty) -> Anne(nie wiadomo) -> George(kawaler).

Na pierwszy rzut oka ciężko stwierdzić czy osoba stanu married patrzy na osobę stanu unmarried.

Mamy tutaj dwie niewiadome - jakiego stanu cywilnego jest Anne i na kogo patrzy George.

To na kogo patrzy George jest nieistotne - analiza pozwala nam wyeliminować ten element, ponieważ nawet jak dowiemy się na kogo patrzy George, jest on wciąż osobą nie żonatą, więc nie odpowie nam to na pytanie, czy osoba stanu married patrzy na osobę stanu unmarried. Wczesne eliminowanie niepotrzebnych informacji pozwala nam zaoszczędzić czasu, nie szukając odpowiedzi w złym miejscu i tym samym dojść do rozwiązania szybciej skupiając się tylko na informacjach istotnych.

Skupmy się więc na Jacku, bo to on jest żonaty. Nie wiemy nic o stanie cywilnym Anne. Spróbujmy więc skorzystać z hipotezy, czyli co by było, gdyby. Gdyby Anne była zamężna to Jack patrzyłby na osobę stanu married. Ale Anne patrzyłaby na osobę stanu unmarried. To sugerowałoby odpowiedź A, czyli ‘Tak’.

Co natomiast, gdyby Anne była niezamężna? To by również sugerowało odpowiedź A, czyli ‘Tak’ - Jack (żonaty) patrzy na Anne (niezamężną).

Przy analizowaniu wymagań często musimy logicznie spojrzeć na ich strukturę. Zadać sobie pytanie jakie informacje mamy, a jakich nie mamy, które informacje są istotne a które nie. Nawet w przypadku braku pewnych informacji (co często się zdarza) jesteśmy w stanie sprawdzić, czy wymagania są jasne, jednoznaczne, bez konfliktów i oczywiście, czy pozwolą nam płynnie je zaimplementować.

Gratulacje dla wszystkich, którzy spróbowali rozwiązać tę zagadkę i dla wszystkich, którym udało się wybrać właściwą odpowiedź!

Wpis #105 Jak byś przetestował krzesło?

Testowanie krzesła to takie obcykane zadanie rekruterskie, kiedy zatrudnia się testerów. Coś w stylu ‘jak byś sprzedał długopis’, gdy chcesz zatrudnić się na sprzedawcę.

Sednem tego pytania nie jest podanie poprawnej odpowiedzi, a raczej pokazanie, że jako tester dostrzegam rzeczy, które są pomijane przez osoby nie testujące. Bardziej liczy się to, co tester będzie chciał wziąć pod uwagę przy testowaniu, niż podanie konkretnych test case’ow, bo bez dodatkowych informacji ciężko będzie to zrobić.

Sama analiza tego przypadku jest w stanie pokazać, że testowanie to nie jest przeklikanie happy path, że nie każdy może testować, że trzeba myśleć o rzeczach, o których normalnie się nie myśli, a do tego potrzeba doświadczenia i konkretnych umiejętności.

Krzesło samo w sobie nie jest specjalnie wypchane funkcjonalnościami. Krzesło stoi i ktoś na nim powinien móc usiąść. Z punktu widzenia funkcjonalności jest to kiepski temat na wykazanie się kreatywnością. Testowanie funkcjonalności to najprostsza część projektowania testów, dlatego chodzi o to, aby odsiać ludzi, którzy nie widzą niczego poza tym.

Czym jest ta trudniejsza część? Trudniejszą częścią jest postawić się po stronie odbiorcy, który może wziąć pod uwagę rzeczy takie jak:

- Rodzaj krzesła, bo jest ich wiele i każdy rodzaj będzie miał pewne cechy odmienne i może też służyć w innym celu.

- Kim będzie użytkownik krzesła, np. czy będzie to dorosły, czy dziecko, jakiś domownik, czy może pracownik biurowy?
- Jak będziemy tego krzesłą używać? Czy będziemy na nim siedzieć, czy może też stać i wkręcać żarówki? Czy będziemy się na nim bujać, czy nie? Czy będziemy się przesuwać razem z tym krzesłem?
- Jak często krzesło będzie używane? Czy będzie stało w domu, w biurze, czy może w galerii handlowej?
- Gdzie to krzesło będzie używane? Czy będzie to krzesło do biurka, a może krzesło barowe? Typowa kwestia kompatybilności.
- W jakich warunkach środowiskowych będziemy tego krzesła używać?
- Tyle chyba wystarczy żeby pokazać, że krzesło to wcale nie jest oczywista sprawa.

Po co nam tyle pytań?

Jeśli klient zamawiał krzesło do kuchni, a my mu dostarczymy krzesło obrotowe, to może być słabo.

Jeśli krzesło ma być dla dorosłych, to musi być odpowiednich rozmiarów i też wytrzymywać odpowiedni ciężar.

Jeśli krzesło będzie używane przez licealistę, który regularnie będzie się na nim bujać to fajnie byłoby, aby nie rozbił sobie głowy (pamiętacie krzesła ze szkoły z wygiętymi nogami z tyłu?).

Jeśli krzesło będzie stało w galerii handlowej, to będzie na nim dziennie siadać setka ludzi.

Jeśli krzesło będzie używane przy barze, to musimy pamiętać, aby nie było zbyt niskie.

Jeśli krzesło będzie regularnie wystawiane na deszcz, to dobrze by było, aby np. nie było drewniane.

Powyżej to tylko przykłady, można by tak dłużej analizować. Osobiście nie lubię tego pytania, ponieważ taka analiza jest troszkę wymuszona. Moim zdaniem specyfika wytwarzania oprogramowania jest inna i bez sensu jest ją porównywać do produkcji krzesła. Wiele z tych problemów powyżej jest zabezpieczanych out-of-the-box. Krzesło ma wytrzymać tyle a tyle kg, bez względu na to, kto będzie z niego korzystał. Inna sprawa, że rzadko kiedy produkujemy krzesła całkowicie customowo. Klient, który chce krzesła do ogródka przy restauracji raczej wybierze krzesła z kategorii ogrodowej.

Dlatego rozkminianie takich case'ów jest strasznie sztuczne i na ten moment rzadko spójne z realiami.

Jeśli chodzi o oprogramowanie wszystkie takie case'y mogą okazać się istotne, bo oprogramowanie to nie produkt hurtowy. Klient przychodzi do nas z często unikalnym problemem, potrzebuje rozwiązania, a my musimy wziąć pod uwagę wszystkie aspekty produktu, który chcemy stworzyć. Kolejna sprawa jest taka, że musimy też pamiętać, aby nie polecieć z tymi wymaganiami. Tak jak w przypadku krzesła, które jest dostosowane do np. 150 kg, tak oprogramowanie dostosowane do szerokiego zakresu wymagań (tak na wszelki wypadek) może okazać się zbyt kosztowne. Funkcjonalność to bardziej oczywista sprawa, choć też zdarzają się funkcjonalności, gdzie trzeba się nagłówkować. Niemniej bardzo cenną umiejętnością testera jest dostrzeganie rzeczy poza funkcjonalnościami.

Mateusz Stachurzewski

QA z zespołu Py03 w Clearcode, który zawsze stawia na pragmatyczną jakość.

Za dzieciaka nie chciał być ani strażakiem, ani astronautą, ani testerem.

Jednak to świat testowania całkowicie go pochłonął.

W pracy daje z siebie wszystko, a po pracy dalej drąży i zgłębia zakamarki bycia QA.

Codziennie i niezmiennie towarzyszy mu pytanie: Jak zapewnić jakość, aby każdy był zadowolony?