



Magazine

Strategie testowania i jakości Agile: dyscyplina ponad retoryką

Cześć III z IV: Strategie testowania Agile

Autor: Scott Ambler



O autorze: W lipcu 2006 r. Scott dołączył do zespołu IBM w Kanadzie jako Główny Metodolog Agile i Lean dla IBM Rational. Scott pracuje dla klientów IBM, w szczególności w obszarach coachingu i usprawniania procesów wytwórczych w IT. Scott pracuje w przemyśle IT od połowy lat 80-tych, a z technologią

obiektywą od wczesnych lat 90-tych. Napisał kilka książek i *white papers* traktujących o obiektywnym wytwarzaniu oprogramowania, procesie oprogramowania, Modelu Skalowania Agile (ASM), Wytwarzaniu Agile Kierowanym Modelem (AMDD), Technikach Baz Danych Agile, Agile UP (ang. *Unified Process*). Przemawiał na wielu konferencjach na całym świecie.

Kontakt: Scott_Ambler@ca.ibm.com twitter: <http://twitter.com/scottwambler>

Intermediate

Level

5

Magazine Number

Inżynieria oprogramowania

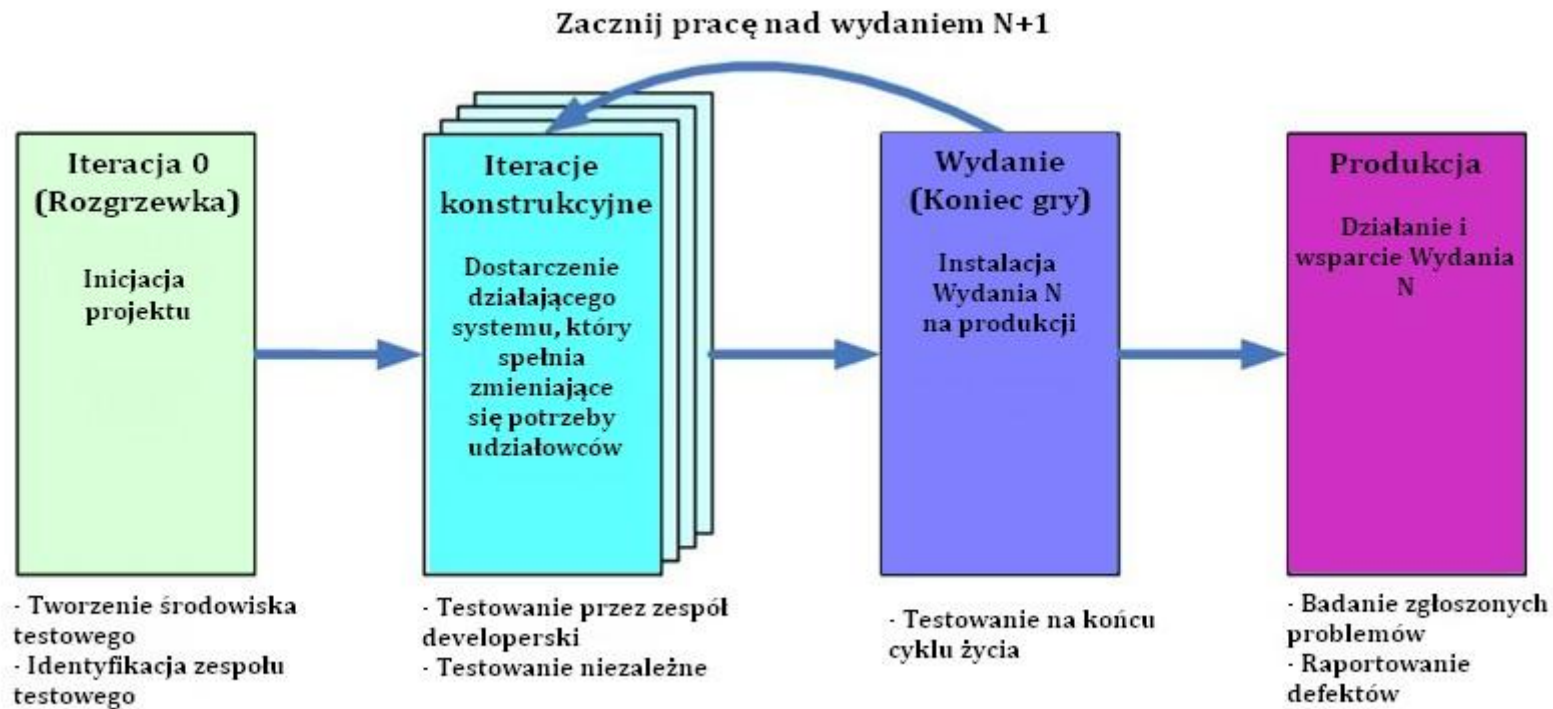
Section in the magazine

Wprowadzenie

Aby zrozumieć, jak czynności testowania dopasowują się do zwinnego wytwarzania systemu, korzystnie jest przyjrzeć się temu z punktu widzenia cyklu życia dostarczenia systemu. Rysunek 10 jest wysoko-poziomym obrazem zwinnego SDLC, wskazując czynności testowe w różnych fazach SDLC. Niniejsza sekcja jest podzielona na następujące tematy:

Niniejszy artykuł dzieli się na następujące tematy:

- Inicjacja projektu
- Kompletny zespół
- Niezależny zespół testowy
- Tworzenie środowiska testowego
- Testowanie przez zespół developerski
- Ciągła integracja
- Wytwarzanie Kierowane Testami (TDD)
- Podejście „Testuj natychmiast po”
- Równoległe niezależne testowanie
- Zarządzanie defektami
- Testowanie na końcu cyklu życia
- Kto to robi?
- Implikacje dla praktyków testowania



Copyright 2006-2009 Scott W. Ambler

Rysunek 1 Testowanie w trakcie SDLC

Inicjacja projektu

Podczas inicjacji projektu, zwanej często „Sprintem 0” w Scrum lub „Iteracją 0” w innych metodach Agile, twoim celem jest spowodować, że zespół pójdzie we właściwym kierunku. Mimo, że społeczność głównego nurtu Agile nie lubi o tym dużo mówić, w rzeczywistości faza ta może trwać od kilku godzin do kilku tygodni, w zależności od natury projektu i kultury twojej organizacji. Z punktu widzenia testowania, głównym zadaniem jest zorganizowanie podejścia do testów i rozpoczęcie tworzenia środowiska testowego, o ile ono jeszcze nie istnieje. Podczas tej fazy twojego projektu, będziesz wykonywać wstępne przewidywanie wymagań (jak opisano wcześniej) i przewidywanie architektury. W wyniku tych prac powinieneś nabyć lepsze zrozumienie zakresu, tego, czy

twój projekt musi spełniać zewnętrzne regulacje takie, jak akt Sarbanes-Oxley czy przewodnik FDA CFR 21 Część 11, i potencjalnie pewnych wysoko-poziomowych kryteriów akceptacji dla twojego systemu – wszystko to stanowi ważne informacje, które powinny pomóc ci zdecydować, ile testowania będziesz musiał wykonać. Ważne jest, by pamiętać, że proces jednego rozmiaru nie musi pasować wszystkim i że różne zespoły projektowe będą miały inne podejścia do testowania, ponieważ odnajdują się w innych sytuacjach – im bardziej złożona sytuacja, tym bardziej złożone podejście do testowania (między innymi). Zespoły odnajdujące się w prostych sytuacjach mogą stwierdzić, że podejście „kompletnego zespołu” do testowania wystarczy, podczas gdy zespoły w bardziej złożonej sytuacji odkryją, że potrzebują niezależnego zespołu testowego pracującego równoległe do zespołu developerskiego. Bez względu na to, zawsze będzie nieco pracy w tworzeniu twojego środowiska testowego.

Strategia organizacji „kompletnego zespołu”

Powszechną w społeczności Agile strategią organizacji, spopularyzowaną przez Kenta Becka w drugiej edycji „*Extreme Programming Explained*”, jest włączanie do zespołu właściwych osób, tak by mieli umiejętności i perspektywy wymagane do tego, by zespół osiągnął sukces. Aby skutecznie dostarczać działający system w regularnych odstępach czasu, zespół musi mieć ludzi z umiejętnościami analitycznymi, projektowymi, programistycznymi, przywódczymi i – tak – nawet ludzi z umiejętnościami testowymi. Oczywiście, nie jest to ani kompletna lista umiejętności wymaganych przez zespół, ani nie implikuje, że każdy w zespole ma wszystkie te umiejętności. Co więcej, każdy w zespole Agile przyczynia się w każdy sposób, jaki może, tym samym zwiększając ogólną produktywność zespołu. Ta strategia nazywa się „kompletny zespół”.

Z podejściem kompletnego zespołu testerzy są „wbudowani” w zespół developerski i aktywnie uczestniczą we wszystkich aspektach projektu. Zespoły Agile odchodzą od tradycyjnego podejścia, w którym ktoś ma pojedynczą specjalizację, na której się skupia – na przykład, Sally tylko programuje, Sanjiv tylko tworzy architekturę a John tylko testuje – do podejścia, w którym ludzie dążą do stania się specjalistami ogólnymi, z szerszym zakresem umiejętności. Więc, Sally, Sanjiv i John będą chcieli być zaangażowani w czynności programowania, tworzenia architektury i testowania, i co ważniejsze, będą chcieli pracować razem i uczyć się od siebie nawzajem, by stać się lepszymi w miarę upływu czasu. Siła Sally może nadal leżeć w programowaniu, Sanjiva w tworzeniu architektury a Johna w testowaniu, ale to nie będą jedyne rzeczy, które będą robić w zespole Agile. Jeśli Sally, Sanjiv i John są nowi w Agile i obecnie są tylko specjalistami – w porządku, poprzez zastosowanie praktyk developmentu *non-solo* i pracy w krótkich cyklach informacji zwrotnej, szybko nabędą nowych umiejętności od swoich kolegów z zespołu (i również przekażą swoje obecne umiejętności kolegom).



To podejście może być wyraźnie odmienne od tego, co zwykły robić zespoły tradycyjne. W zespołach tradycyjnych powszechne jest to, że programiści (specjaliści) piszą kod a następnie „przerzucają go przez mur” do testerów (również specjalistów), którzy następnie go testują i raportują podejrzewane defekty z powrotem do programistów. Mimo że lepsze to, niż brak testowania w ogóle, często okazuje się to strategią kosztowną i czasochłonną z powodu przekazywania prac pomiędzy dwoma grupami specjalistów. W zespołach Agile programiści i testerzy pracują ramię w ramię i z biegiem czasu różnica pomiędzy tymi dwoma rolami zaciera się do pojedynczej roli developera. Interesującą filozofią w społeczności Agile jest, to że prawdziwi profesjonalści IT powinni walidować swoją pracę najlepiej, jak potrafią i dążyć do doskonalenia się.

Strategia kompletnego zespołu nie jest doskonała, oto kilka potencjalnych problemów:

Myślenie grupowe. Kompletny zespół – tak samo, jak każdy inny typ zespołu – może cierpieć na coś, co jest znane jako „myślenie grupowe”. Podstawowym problemem jest to, że ponieważ członkowie zespołu pracują blisko siebie (do czego dążą zespoły Agile), zaczynają oni myśleć podobnie i w wyniku tego stają się ślepi na pewne problemy, jakie napotykają. Na przykład, system nad którym pracuje twój zespół, może mieć kilka poważnych problemów z użytecznością, ale nikt z zespołu nie jest tego świadom, ponieważ albo nikt z zespołu nie ma umiejętności związanych z użytecznością, albo zespół w pewnym momencie błędnie zdecydował się zbagatelizować problemy z użytecznością i członkowie zespołu zapomnieli o nich.

Zespół może nie mieć umiejętności, których potrzebuje. Jest całkiem łatwo zadeklarować, że przestrzegasz strategii „kompletnego zespołu”, ale czasami nie tak łatwo rzeczywiście to robić. Niektórzy ludzie mogą przeceniać swoje umiejętności, albo nie doceniać, jakich umiejętności naprawdę potrzebują i tym samym narażać zespół na ryzyko. Dla przykładu, powszechną słabością wśród programistów są umiejętności projektowania baz danych, z powodu braku szkoleń w takich umiejętnościach, nad-specjalizacji wewnątrz organizacji, gdzie „profesjonaliści danych” skupiają się na umiejętnościach związanych z danymi a programiści – nie oraz kulturowym niedopasowaniu pomiędzy developerami i profesjonalistami danych, co utrudnia nabycie umiejętności związanych z danymi. Gorzej – strategii enkapsulacji baz danych, takie jak obiektowo-relacyjne środowiska mapowania np. Hibernate, powodują, że ci kwestionujący dane (ang. *data-challenged*) programiści wierzą, iż nie muszą wiedzieć nic o projekcie bazy danych ponieważ dostęp do tego mają zamknięty. Tak więc, pojedynczy członkowie zespołu wierzą, że posiadają umiejętności, by wykonać pracę a w rzeczywistości ich nie mają – w takim przypadku ucierpi system, ponieważ zespół nie ma umiejętności do odpowiedniego zaprojektowania i używania bazy danych.

Zespół może nie wiedzieć, jakie umiejętności są potrzebne. Nawet gorzej, zespół może nie pamiętać, jakie umiejętności są właściwie potrzebne. Dla przykładu, system mógł mieć kilka bardzo istotnych wymagań bezpieczeństwa, ale jeśli zespół nie wiedział, że bezpieczeństwo będzie prawdopodobnie problemem, mógł całkowicie pominąć te wymagania lub zrozumieć je źle i przypadkowo narażić projekt na ryzyko.

Na szczęście korzyści z podejścia kompletnego zespołu znacznie przeważają nad potencjalnymi problemami. Po pierwsze, okazuje się, że kompletny zespół zwiększa ogólną produktywność poprzez zmniejszenie, a często wyeliminowanie, nieodłącznego w podejściach tradycyjnych czasu oczekiwania pomiędzy

programowaniem a testowaniem. Po drugie, jest mniejsza potrzeba pracy papierkowej, takiej jak szczegółowe plany testów, z powodu braku przekazywania pracy pomiędzy odrębnymi zespołami. Po trzecie, programiści zaczynają się szybko uczyć umiejętności testowania i jakości od testerów i w wyniku tego wykonują lepszą pracę – jeśli developer wie, że będzie aktywnie zaangażowany w prace testowe, jest bardziej zmotywowany do pisania kodu o wysokiej jakości i testowalnego.

Niezależny zespół testowy (Strategia zaawansowana)

Podjęcie kompletnego zespołu w praktyce działa dobrze, jeśli developerzy Agile odnajdują się we względnie prostych sytuacjach. Jednak kiedy środowisko jest złożone, lub jeśli domena problemu sama w sobie jest skomplikowana, system jest duży (często w wyniku wspierania złożonej dziedziny) lub jeśli jest potrzeba integracji z ogólną infrastrukturą, która obejmuje miriady innych systemów, wtedy podjęcie kompletnego zespołu do testowania okazuje się niewystarczające. Zespoły w złożonych środowiskach, jak i zespoły, które znajdują się w sytuacjach podporządkowania się regulacjom, często stwierdzają, że potrzebują uzupełnienia ich kompletnego zespołu niezależnym zespołem testowym. Zespół testowy będzie wykonywać równoległe niezależne testowanie podczas całego projektu i zwykle będzie odpowiedzialny za testowanie na końcu cyklu życia, wykonywane podczas fazy wydania/przejścia w projekcie. Celem tych prac jest odkrycie miejsc, w których system się psuje (testowanie kompletnego zespołu często skupia się na testach potwierdzających, które pokazuje, że system działa) i zaraportowanie takich awarii zespołowi developerskiemu, tak by mogli je naprawić. Niezależny zespół testowy będzie się skupiał na bardziej złożonych formach testowania, które są zwykle poza możliwościami realizacji przez sam „kompletny zespół” (więcej o tym – później).

Twój niezależny zespół testowy będzie wspierać wiele zespołów projektowych. Większość organizacji ma wiele zespołów developerskich pracujących równoległe, często dziesiątki zespołów a czasem nawet setki, tak więc możesz osiągnąć ekonomię skali posiadając niezależny zespół testowy wspierający wiele zespołów developerskich. To umożliwi minimalizację ilości licencji na narzędzia testowe, których potrzebujesz, dzielenie drogich środowisk sprzętowych i zapewnia, że specjaliści testowania (np. ludzie doświadczeni w testowaniu użyteczności czy testowaniu dochodzeniowym) wspierają wiele zespołów.

Ważne jest, by zauważyć, że niezależne zespoły testowe Agile pracują wyraźnie odmiennie, niż tradycyjne niezależne zespoły testowe. Niezależny zespół testowy Agile skupia się na mniejszej części prac testowych, najcięższej części, podczas gdy zespół developerski wykonuje większość podstawowych prac testowych. W podejściu tradycyjnym, zespół testowy często wykonuje zarówno testowanie podstawowe, jak i złożone formy testowania. Mając to na uwadze, stosunek ludzi w zespołach developerskich Agile do ludzi w niezależnym zespole testowym Agile często będzie rzędu 15:1 lub 20:1, podczas gdy w świecie tradycyjnym proporcje te są bliższe 3:1 lub 1:1 (a w środowiskach regulacyjnych może być 1:2 i więcej).

Tworzenie środowiska testowego

Na początku twojego projektu będziesz musiał zacząć tworzenie środowiska, włączając określenie twojego obszaru pracy, sprzętu i narzędzi developerskich (by nazwać kilka rzeczy). Oczywiście będziesz musiał stworzyć swoje środowisko testowe – od początku, jeśli obecnie nie masz dostępnego takiego środowiska – lub poprzez dostosowanie istniejącego środowiska do spełnienia twoich potrzeb. Istnieje kilka strategii, które zazwyczaj sugerują, kiedy przychodzi do organizacji środowiska testowego:

Przyjmij dla developerów narzędzia *open source* (OSS). Dostępnych jest wiele świetnych narzędzi testowych OSS, takich jak środowisko testowe xUnit, które są przeznaczone dla developerów. Takie narzędzia są często łatwe w instalacji i nauczaniu się ich obsługi (choć nauczanie się, jak testować efektywnie okazuje się być inną kwestią).

Przyjmij dla niezależnych testerów narzędzia *open source* (OSS). Ponieważ twój niezależny zespół testowy często będzie adresować bardziej złożone kwestie testowe, takie jak testowanie integracji pomiędzy wieloma systemami (nie tylko pojedynczy system, na którym skupia się zespół developerski), testowanie bezpieczeństwa, testowanie użyteczności i tak dalej, będzie on potrzebował na tyle wyszukanych narzędzi testowych, aby objąć wszystkie te kwestie.

Miej dzielony system śledzenia błędów/defektów. Jak widać na Rysunku 2, i jak wspominałem w poprzedniej sekcji, niezależny zespół testowy będzie przysyłać raporty defektów z powrotem do zespołu developerskiego, którzy z kolei będą często postrzegać te raporty jako po prostu inny typ wymagań. Implikacją tego jest to, że muszą oni mieć wsparcie narzędziowe. Jeśli zespół jest mały i zlokalizowany w pobliżu, jak widzimy na poziomach 1 i 2 Modelu Dojrzałości Procesu Agile (ang. *Agile Process Maturity Model (APMM)*), możliwe że byłoby to tak proste, jak stos kart indeksowych. W bardziej złożonych środowiskach, będziesz potrzebować narzędzia opartego na oprogramowaniu, najlepiej jednego, które będzie używane do zarządzania zarówno wymaganiami zespołu, jak i defektami (lub co ważniejsze, ich całą listą elementów pracy). Więcej o tym – później.

Zainwestuj w sprzęt do testowania. Zarówno zespół developerski, jak niezależny zespół testowy będzie potrzebował sprzętu, na którym ma wykonywać testowanie.

Zainwestuj w narzędzia wirtualizacji i zarządzania laboratorium testowym. Sprzęt testowy jest drogi i nigdy nie ma go dość. Oprogramowanie do wirtualizacji, które umożliwi ci łatwe załadowanie środowiska testowego na twój sprzęt, oraz narzędzia do zarządzania laboratorium testowym, które umożliwi ci monitorowanie konfiguracji sprzętu, są krytyczne dla powodzenia twojego niezależnego zespołu testowego (szczególnie, jeśli wspiera on wiele zespołów developerskich).

Zainwestuj w narzędzia ciągłej integracji (ang. *continuous integration (CI)*) i ciągłej instalacji (ang. *continuous deployment (CD)*). Nie jest to dosłownie kategoria narzędzi testowych, ale narzędzia CI/CD są krytyczne dla zespołów developerskich Agile z powodu praktyk takich jak: Wytwarzanie Kierowane Testami (TSS), ogólnie developerskie testy regresji i potrzeba instalacji działających buildów na niezależnych środowiskach testowych, środowiskach demonstracyjnych czy nawet produkcji. Narzędzia *open source*, takie jak Maven i CruiseControl działają dobrze w prostych sytuacjach, jednak w bardziej złożonych sytuacjach (szczególnie w skali) odkryjesz, że potrzebujesz komercyjnych narzędzi CI/CD.

Strategie testowania w zespole developerskim

Zespoły developerskie Agile generalnie przestrzegają strategii kompletnego zespołu, w której ludzie z umiejętnościami testerskimi są efektywnie osadzeni w zespole developerskim i zespół jest odpowiedzialny za większość testowania. Taka strategia działa dobrze w większości sytuacji, ale kiedy twoje środowisko jest bardziej złożone, stwierdzisz, że potrzebujesz również niezależnego zespołu testowego pracującego równoległe z developmentem i być może wykonującego również testowanie na końcu cyklu życia. Bez względu na sytuację, zespoły developerskie Agile będą przyjmować takie praktyki, jak ciągła integracja, która umożliwi im wykonywanie ciągłych testów regresji, wraz z podejściami Wytwarzania Kierowanego Testami (TDD) lub „testuj natychmiast po”.

Ciągła integracja (CI)

Ciągła integracja (CI) jest praktyką, w której co najmniej raz na każde kilka godzin, a jeszcze lepiej częściej, powinienes:

Zbudować twój system. Obejmuje to zarówno kompilację kodu, jak i potencjalnie rebuild twojej testowej bazy danych (lub baz danych). To brzmi prosto, ale dla dużych systemów, które składają się z podsystemów, potrzebujesz strategii do określenia, jak zamierzasz zbudować zarówno podsystemy, jak cały system (możesz zrobić tylko jeden build kompletnego systemu w ciągu nocy, lub na przykład raz w tygodniu, i budować tylko podsystemy w regularnych odstępach czasu).

Uruchom twój zestaw testów regresji. Po udanym buildzie, kolejnym krokiem jest uruchomienie twojego zestawu testów regresji. Który zestaw uruchomisz, będzie określone przez zakres builda i rozmiar systemu. Dla małych systemów, prawdopodobnie będziesz miał pojedynczy zestaw zawierający wszystkie testy, ale w bardziej złożonych sytuacjach będziesz mieć kilka zestawów testów do celów wydajnościowych. Dla przykładu, jeśli robię build na mojej własnej stacji roboczej, wtedy prawdopodobnie uruchomię tylko ten podzbiór testów, które walidują funkcjonalność, nad którą zespół pracował ostatnio (powiedzmy, przez ostatnie kilka iteracji), ponieważ ta funkcjonalność jest najbardziej narażona na awarie i mogę uruchomić testy nawet z „zaślepkami”. A ponieważ prawdopodobnie będę uruchamiał ten zestaw testów kilka razy dziennie, musi on być wykonywany szybko, najlepiej w ciągu mniej, niż 5 minut,

choć niektórzy pozwolą, by ich developerski zestaw testów przekraczał 10 minut. „Zaśleпки” mogą być używane do symulowania części systemu, których testowanie jest drogie, z punktu widzenia wydajności – takie jak baza danych czy zewnętrzny podsystem. Jeśli używasz zaślepek, lub jeśli testujesz część funkcjonalności, wtedy będziesz potrzebować jednego lub więcej zestawów testów o większym zasięgu. Minimalnie będziesz potrzebować zestawu testów, który będzie testował rzeczywiste komponenty systemu (nie zaśleпки) i to uruchomi wszystkie twoje testy. Jeśli ten zestaw testów wykonuje się przez kilka godzin, wtedy może być uruchomiony w nocy; jeśli trwa to dłużej, wtedy będziesz chciał go przeorganizować tak, by bardzo długo wykonujące się testy były w osobnym zestawie testów, który wykonuje się nieregularnie. Dla przykładu, widziałem system posiadający długo wykonujący się zestaw testów, który był uruchamiany przez kilka miesięcy w celu wykonania złożonych testów obciążeniowych i dostępności (coś, co prawdopodobnie będzie robił niezależny zespół testowy).

Wykonuj analizę statyczną. Analiza statyczna to technika jakościowa, w której automatyczne narzędzie sprawdza obecność defektów w kodzie, często szukając takich typów problemów jak błędy zabezpieczeń lub problemy ze stylem kodowania (aby wymienić kilka).

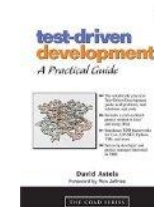
Twoje zadanie integracyjne może być wykonywane w określonych momentach, może raz na godzinę, lub za każdym razem, jak ktoś wprowadza nową wersję komponentu (taką jak kod źródłowy), który jest częścią builda.

Zaawansowane zespoły, szczególnie te w sytuacji „zwinności w skali”, odkryją, że jest także potrzeba rozważenia ciągłej instalacji. Podstawową ideą jest tu to, że automatyzujesz instalację twojego działającego builda – niektóre organizacje traktują to jako „promocję ich działającego builda” – do innego środowiska w regularnych odstępach czasu. Dla przykładu, jeśli na końcu tygodnia otrzymałeś udany build, możesz zechcieć automatycznie zainstalować go w stabilnym obszarze tak, by mógł być poddany niezależnym równoległym testom. Lub – jeśli na koniec dnia jest działający build, możesz chcieć zainstalować go w środowisku demonstracyjnym, by ludzie spoza twojego zespołu mogli zobaczyć postępy, jakie robicie.

Wytwarzanie Kierowane Testami (TDD)

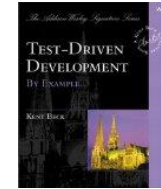
Wytwarzanie Kierowane Testami (ang. *Test-driven development (TDD)*) jest techniką developerską Agile, która łączy:

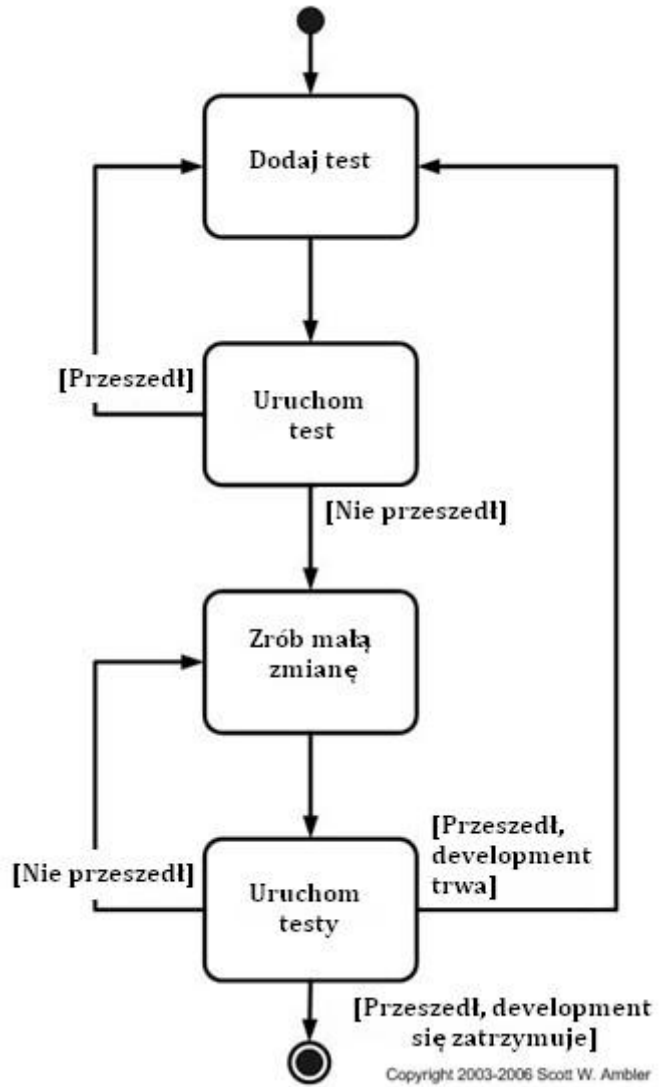
Refactoring. Refactoring jest techniką, w której w swoim istniejącym kodzie źródłowym lub w danych źródłowych wykonujesz małą zmianę, by ulepszyć jego projekt bez zmiany semantyki. Przykłady refactoringu obejmują: przeniesienie operacji wyżej w hierarchii dziedziczenia i zmiana nazwy operacji w kodzie źródłowym aplikacji; wyrównanie pól i zastosowanie spójnej czcionki w interfejsie użytkownika; zmiana nazw kolumny lub podzielenie tabeli w relacyjnej bazie danych. Kiedy pierwszy raz decydujesz się pracować nad jakimś zadaniem, patrzysz na istniejące źródło i pytasz, czy to jest najlepszy projekt, by umożliwić dodanie nowej funkcjonalności, nad którą pracujesz. Jeśli tak – kontynuuj z TDD. Jeśli nie – poświęć czas na naprawę części kodu i danych, tak by był to możliwie najlepszy



projekt dla ulepszenia jakości i tym samym zredukowania wad technicznych w czasie.

Wytwarzanie „najpierw test” (ang. *Test-first development (TFD)*). Z TFD piszesz pojedynczy test a następnie piszesz dokładnie tyle oprogramowania, by wypełnić ten test. Kroki wytwarzania „najpierw test” (TFD) są przedstawione na diagramie aktywności UML na Rysunku 11. Pierwszym krokiem jest szybko dodać test, po prostu tylko tyle kodu, by test się nie powiódł. Następnie wykonujesz swoje testy, często kompletny zestaw testów – chociaż przez wzgląd na szybkość możesz zdecydować się uruchomić tylko podzbiór, by upewnić się, że nowy test rzeczywiście się nie udaje. Następnie aktualizujesz kod funkcjonalny, by sprawić, że przejdzie nowe testy. Czwartym krokiem jest wykonanie testów jeszcze raz. Jeśli się nie powiodą, musisz aktualizować swój kod funkcjonalny i wykonać retesty. Po przejściu testów, kolejnym krokiem jest rozpoczęcie od nowa.





Rysunek 2 Kroki projektowania „najpierw test” (TFD).

Są dwa poziomy TDD:

Akceptacyjne TDD. Możesz wykonywać TDD na poziomie wymagań, przez pisanie pojedynczego testu klienta, równoważnego testowi funkcjonalnemu lub akceptacyjnemu w świecie tradycyjnym. TDD akceptacyjny jest często nazywany „story TDD”, w którym najpierw automatyzujesz ten story test, który się nie powiódł, następnie kierujesz projektowaniem poprzez TDD aż do momentu, w których story test przejdzie (story test jest testem akceptacji klienta).

Developerskie TDD. Możesz również wykonywać TDD na poziomie projektowym z testami developerskimi.

Z podejściem wytwarzania kierowanego testami (TDD), twoje testy efektywnie stają się szczegółowymi specyfikacjami tworzonymi „dokładnie na czas” (ang. *Just in time (JIT)*). Większość programistów nie czyta pisanej dokumentacji systemu, zamiast tego preferują pracę z kodem. I nic w tym złego. Kiedy próbują zrozumieć klasę czy operację, większość programistów najpierw spojrzy na przykładowy kod, który już to wywołuje. Dobrze napisane testy jednostkowe/developerskie dokładnie to robią – dostarczają działającej specyfikacji funkcjonalnego kodu – i w rezultacie testy jednostkowe efektywnie stają się istotną częścią dokumentacji technicznej. Podobnie, testy akceptacyjne mogą formować ważną część dokumentacji wymagań. Jeśli się zatrzymasz i pomyślisz o tym, ma to wiele sensu. Twoje testy akceptacyjne określają dokładnie, czego oczekują od systemu twoi udziałowcy, dlatego też specyfikują one krytyczne wymagania. Implikacją tego jest to, że akceptacyjne TDD jest techniką szczegółowej specyfikacji wymagań JIT a developerskie TDD jest techniką szczegółowej specyfikacji projektu JIT. Pisanie wykonywalnych specyfikacji jest jedną z najlepszych praktyk Modelowania Agile.

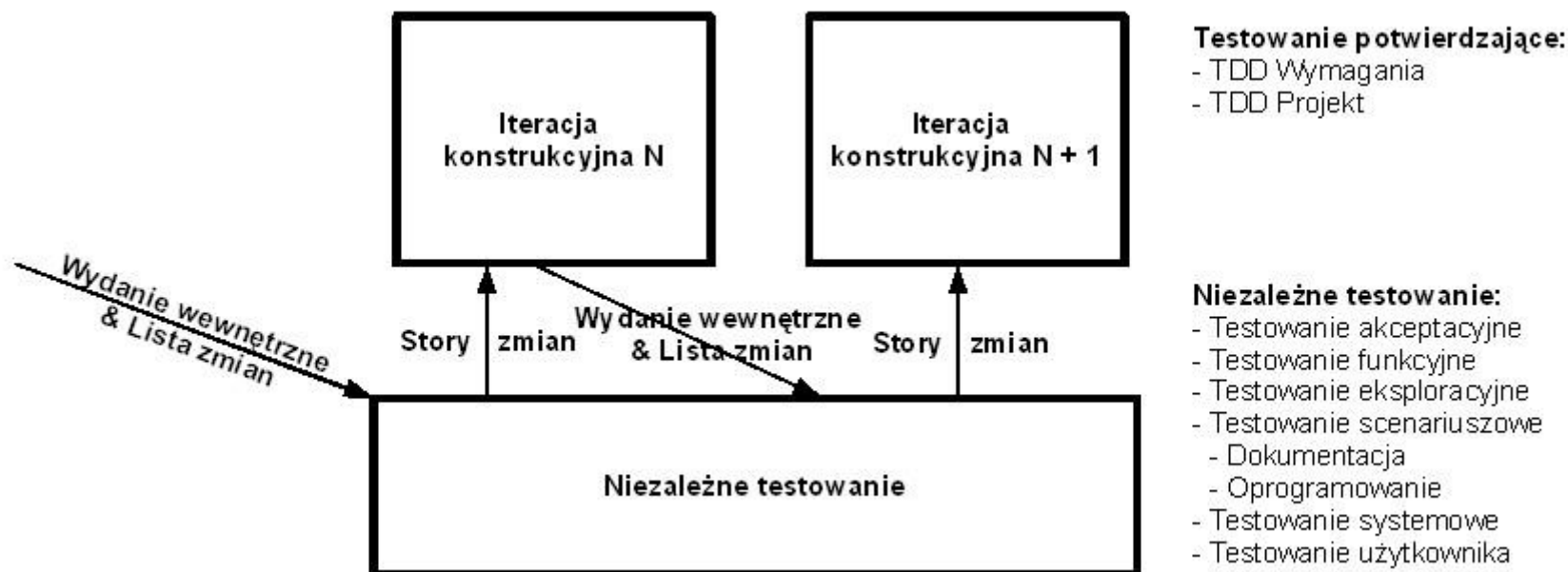
Podejście “testuj natychmiast po”

Chociaż wielu praktyków Agile mówi o TDD, w rzeczywistości wydaje się, że jest o wiele więcej wytwarzania „testuj po”, w którym piszą oni nieco kodu a następnie jeden lub więcej testów do walidacji. TDD wymaga znacznej dyscypliny, w rzeczywistości wymaga poziomu dyscypliny, który znajdujemy u ledwie kilku koderów, szczególnie koderów, którzy w developmencie przestrzegają podejść solo zamiast podejść *non-solo*, takich jak programowanie parami. Bez partnera, zapewniającego twoją uczciwość, całkiem łatwo jest wpaść w nawyk pisania kodu produkcyjnego przed pisaniem kodu testowego. Jeśli piszesz testy bardzo wcześnie po tym, jak napisałeś kod produkcyjny (innymi słowy „testujesz natychmiast po”), to jest prawie tak dobre jak TDD, problem pojawia się, gdy piszesz testy po kilku dniach czy tygodniach – o ile w ogóle.

Popularność narzędzi do badania pokrycia kodu, takich jak Clover i Jester wśród programistów Agile jest wyraźną oznaką tego, że wielu z nich naprawdę przyjmuje podejście „testuj po”. Te narzędzia ostrzegają cię, kiedy napisałeś kod, który nie ma pokrywających go testów zmuszając cię do napisania testów, które miejmy nadzieję napisałeś wcześniej poprzez TDD.

Niezależne równoległe testowanie

Podjęcie do wytwarzania “kompletnego zespołu”, gdzie zespoły Agile testują najlepiej jak potrafią, jest doskonałym startem, ale w pewnych sytuacjach nie wystarcza. W takich sytuacjach, opisanych poniżej, musisz rozważyć powołanie niezależnego równoległego zespołu testowego, który wykona niektóre z trudniejszych (lub może lepszym terminem jest zaawansowanych) form testowania. Jak możesz zobaczyć na Rysunku 12, podstawową koncepcją jest tu to, że w regularnych odstępach czasu zespół developerski udostępnia działający build niezależnemu zespołowi testowemu (lub automatycznie instaluje go poprzez swoje narzędzie ciągłej integracji), by mogli go przetestować. Celem tych prac testowych nie jest powtórzenie testów potwierdzających, co już zostało wykonane przez zespół developerski, ale identyfikacja defektów, które przeszły przez sito. Implikacją jest to, że ten niezależny zespół testowy nie potrzebuje szczegółowej specyfikacji wymagań, chociaż może potrzebować diagramów architektury, czy ogólnego zakresu i listy zmian od momentu, w którym zespół developerski ostatni raz oddał im build. Zamiast testować według specyfikacji, niezależne testowanie skupi się na testach integracyjnych systemu na poziomie produkcyjnym, testowaniu dochodzeniowym i formalnym testowaniu użyteczności, by nazwać parę rzeczy.



Rysunek 3 Niezależne równoległe testowanie

Niezależny zespół testowy raportuje defekty do developmentu, co jest określone jak „story zmian” (ang. *change stories*) na Rysunku 12 (ponieważ będąc Agile, mamy tendencję do zmieniania nazw wszystkiego). Te defekty są traktowane przez zespół developerski jako typ wymagań w tym, że są priorytetyzowane, szacowane i wkładane na stos elementów pracy.

Istnieje kilka powodów, dla których powinieneś rozważyć niezależne równoległe testowanie:

Testowanie dochodzeniowe. Podejścia testowania potwierdzającego, takie jak TDD, walidują, że zaimplementowałeś wymagania tak, jak zostały tobie opisane. Ale co dzieje się, jeśli wymagania są pominięte? User story, ulubiona technika pozyskiwania wymagań w społeczności Agile, są świetnym sposobem eksplorowania wymagań funkcjonalnych ale defekty towarzyszące wymaganiom nie-funkcjonalnym, takim jak bezpieczeństwo, użyteczność i wydajność, mają tendencję do tego, że w tym podejściu są pomijane.

Brak zasobów. Co więcej, wiele zespołów developerskich może nie mieć zasobów potrzebnych do wykonania efektywnego testowania integracji systemowej, zasobów, które z ekonomicznego punktu widzenia muszą być współdzielone w wielu zespołach. Implikacją jest to, że odkryjesz, iż potrzebujesz niezależnego zespołu testowego pracującego równoległe do zespołu developerskiego, który rozwiązuje te rodzaje problemów. Testy integracji systemowej często wymagają drogiego środowiska, co leży poza możliwościami indywidualnego projektu.

Duże albo rozproszone zespoły. Duże albo rozproszone zespoły są często podzielone na mniejsze zespoły i kiedy to się dzieje, testowanie integracji systemowej całego systemu może stać się na tyle złożone, że pojedynczy zespół powinien rozważyć wzięcie tego na siebie. Krótko mówiąc, testowanie „kompletnego zespołu” działa dobrze w Agile dla małych systemów, ale dla bardziej złożonych systemów lub dla Agile w skali musisz być bardziej wyrafinowany.

Złożone domeny. Kiedy masz bardzo złożoną domenę, być może pracujesz nad oprogramowaniem krytycznym dla życia lub procesującym finanse, podejścia kompletnego zespołu mogą okazać się niewystarczające. Posiadanie niezależnego równoległego zespołu testowego może zmniejszyć to ryzyko.

Złożone środowiska techniczne. Kiedy pracujesz z wieloma technologiami, systemami zastanymi lub zastanymi źródłami danych, testowanie twojego systemu może stać się bardzo trudne.

Stosowanie się do regulacji. Niektóre regulacje wymagają od ciebie zapewnienia niezależności prac testowych. Moje doświadczenie mówi, że najbardziej efektywnym sposobem, by to zrobić, jest tu wykonywanie testów równoległe do prac zespołu developerskiego.

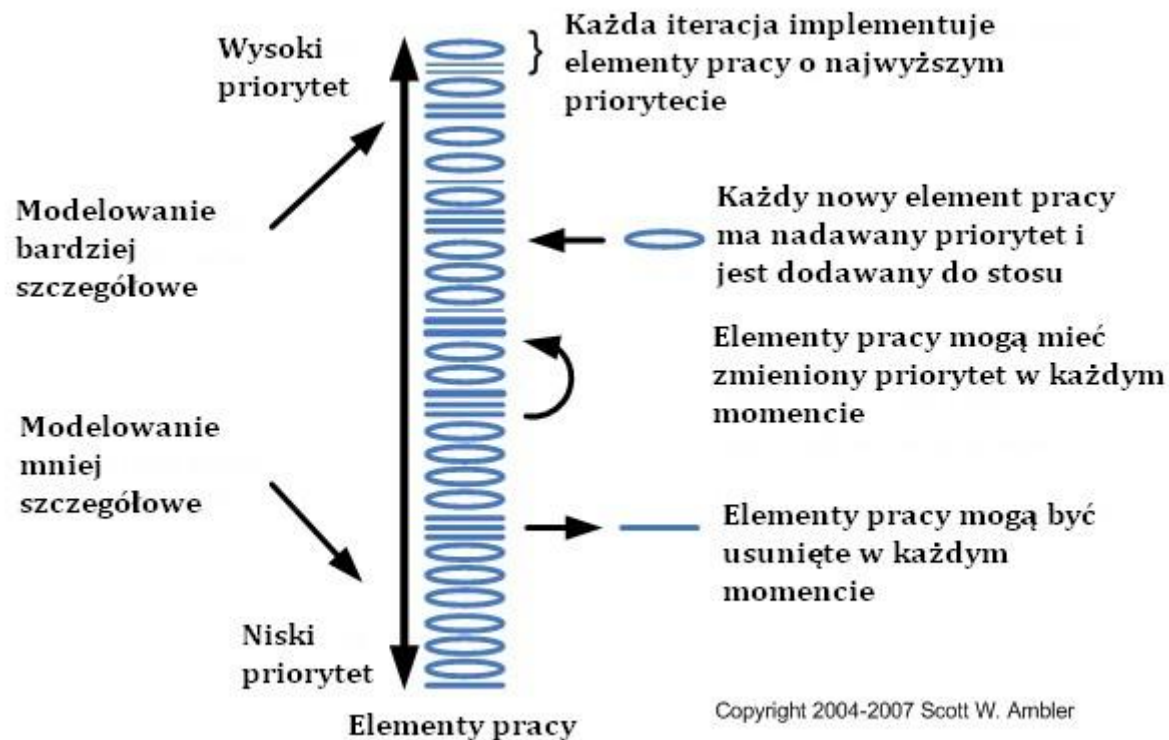
Testowanie gotowości produkcyjnej. System, który budujesz, musi „dobrze grać” z innymi systemami znajdującymi się obecnie na produkcji, gdzie ma być dostarczony twój system. Aby odpowiednio to sprawdzić, musisz testować z tymi wersjami innych systemów, które są obecnie w trakcie wytwarzania – co

oznacza, że potrzebujesz dostępu do zaktualizowanych wersji w regularnych odstępach czasu. W małych organizacjach jest to całkiem proste, ale jeśli twoja organizacja ma dziesiątki, jeśli nie tysiące projektów IT w toku, dla pojedynczego zespołu developerskiego zdobycie takiego dostępu staje się przytłaczające. Bardziej efektywnym podejściem jest posiadanie niezależnego zespołu testowego odpowiedzialnego za takie testowanie integracji systemowej na poziomie przedsiębiorstwa.

Niektórzy praktycy Agile będą twierdzić, że nie potrzebujesz równoległego niezależnego testowania i w prostych sytuacjach jest to zwyczajnie prawdą. Dobra wiadomość jest taka, że niewiarygodnie łatwo jest określić, czy twoje niezależne prace testowe dostarczają wartości, czy nie: po prostu porównaj prawdopodobny wpływ zaraportowanych *story* defektów/zmian z kosztem wykonywania niezależnego testowania.

Zarządzanie defektami

Zarządzanie defektami z dwóch powodów jest często o wiele prostsze w projektach Agile, niż w klasycznych/tradycyjnych projektach. Po pierwsze, z podejściem „kompletnego zespołu” do testowania w momencie, w którym znajdowany jest defekt, jest on zwykle naprawiany na miejscu, często przez osobę (osoby), która go wprowadziła. W tym przypadku cały proces zarządzania defektami jest co najwyżej rozmową pomiędzy kilkoma ludźmi. Po drugie, kiedy niezależny zespół testowy pracuje równoległe z zespołem developerskim nad walidacją ich pracy, zwykle używają narzędzia raportowania defektów, takiego jak ClearQuest czy Bugzilla, by poinformować zespół developerski o tym, co znaleźli. Zdyscyplinowane zespoły dostarczania Agile łączą swoje strategie zarządzania wymaganiami i zarządzania defektami, by uprościć ogólny proces zarządzania zmianą. Podsumowuje to Rysunek 13 (tak, jest taki sam jak Rysunek 6) pokazując, jak pracuje się nad elementami pracy w porządku określonym priorytetami. Zarówno wymagania, jak i raporty błędów są typami elementów pracy traktowanymi równoważnie – są one szacowane, priorytetyzowane i układane na stosie elementów pracy.



Rysunek 4 Proces zarządzania zmianą/defektami Agile

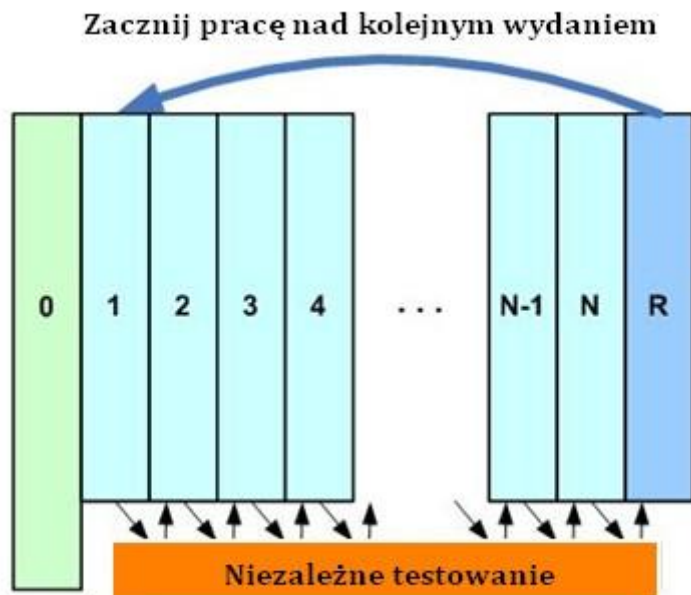
To działa, ponieważ defekty są po prostu innym typem wymagań. Defekt X może być zmieniony na wymaganie „proszę naprawić X”. Wymagania są również typem defektu – wymaganie jest po prostu brakującą funkcjonalnością. W rzeczywistości, niektóre zespoły Agile będą nawet zbierać wymagania przy użyciu narzędzia zarządzania defektami, przykładowo zespół developerski Eclipse wykorzystuje Bugzillę a zespół developerski Jazz – Rational Team Concert (RTC).

Główne utrudnienie w przyjęciu tej strategii – traktowania wymagań i defektów jako tej samej rzeczy – pojawia się, kiedy zespół dostarczenia Agile znajduje się w sytuacji „stałej ceny” (ang. *fixed price*) lub „stałego oszacowania” (ang. *fixed estimate*). W takich sytuacjach klient zwykle chce płacić za nowe wymagania, które nie były uzgadniane na początku projektu, jednak nie powinien płacić za poprawianie błędów. W takich przypadkach podejście biurokratyczne polegałoby na zwyczajnym zaznaczeniu danego elementu pracy jako czegoś, za co trzeba będzie zapłacić ekstra (lub nie). Naturalnie ja preferuję podejście pragmatyczne. Jeśli jesteś w sytuacji „stałej ceny”, może zainteresuje cię, że napisałem dość sporo o tej i ważniejszych alternatywach

budżetowania projektów Agile. Będąc bezceremonialnym, waham się pomiędzy stwierdzeniem, że strategie stałej ceny są nieetyczne lub zwyczajnie są oznaką groteskowej niekompetencji części organizacji nalegającej na nie. Łączenie procesów zarządzania wymaganiami i defektami w jeden, prosty proces zarządzania zmianą jest wspaniałą możliwością ulepszenia procesu. Wyjątkowo wątpliwe strategie budżetowania projektu nie powinny wstrzymywać cię przez czerpaniem korzyści z tej strategii.

Testowanie na końcu cyklu życia

Dla wielu zespołów Agile ważną częścią prac związanych z wydaniem jest testowanie na końcu cyklu życia, gdzie niezależny zespół testowy waliduje, czy system jest gotowy przejść na produkcję. Jeśli została przyjęta praktyka niezależnego równoległego testowania, wtedy testowanie na końcu cyklu życia może być bardzo krótkie, ponieważ dane kwestie zostały już istotnie pokryte. Jak widać na Rysunku 14, prace niezależnego testowania rozciągają się na fazę wydania cyklu życia dostawy, ponieważ niezależny zespół testowy będzie nadal musiał przetestować kompletny system, jak już będzie on dostępny.



Copyright 2006-2008 Scott W. Ambler

Rysunek 5 Niezależne testowanie w trakcie cyklu życia

Jest kilka powodów, dla których nadal musisz wykonywać testowanie na końcu cyklu życia:

Wykonywanie go jest profesjonalne. Będziesz chciał uruchomić ostatni raz wszystkie twoje testy regresji, aby znaleźć się w pozycji, w której możesz zadeklarować, że twój system jest w pełni przetestowany. Nastąpi to właśnie wtedy, gdy iteracja N – ostatnia iteracja konstrukcyjna – się skończy (albo będzie ostatnią rzeczą, którą zrobisz w iteracji N, nie dzielmy włosa na czworo).

Możesz być zobowiązany, by to zrobić. Jest tak albo z powodu kontraktu, jaki masz z klientem biznesowym, albo z powodu zobowiązań regulacyjnych (jak wykazało badanie State of the IT Union z listopada 2009 roku, wiele zespołów Agile znajduje się z takiej sytuacji).

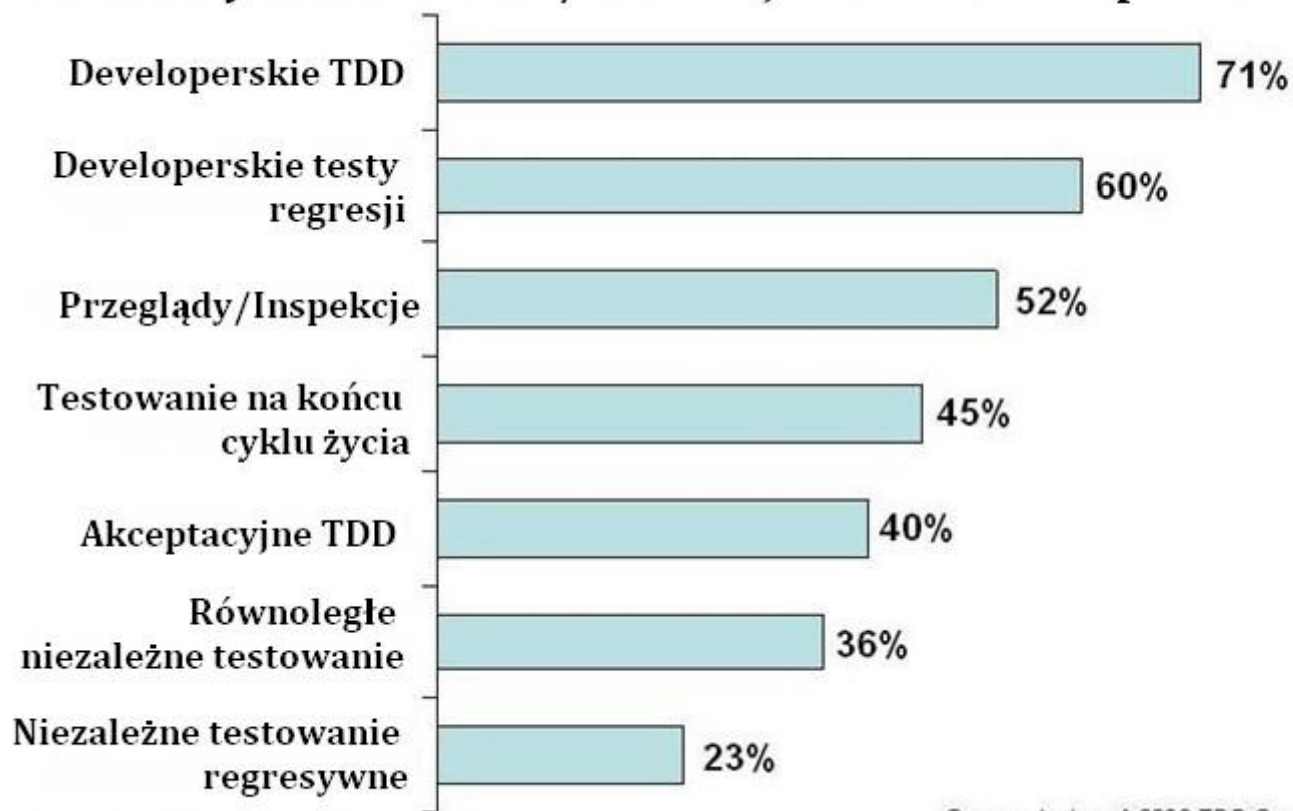
Twoi udziałowcy tego wymagają. Twoi udziałowcy, szczególnie twój departament operacyjny, prawdopodobnie będzie wymagał pewnego rodzaju testowania przed wydaniem rozwiązania na produkcję – aby poczuć się komfortowo co do jakości twojej pracy.

W głównym nurcie społeczności Agile odbywa się mała publiczna dyskusja o testowaniu na końcu cyklu życia; założeniem wielu ludzi przestrzegających głównych metod Agile (takich jak Scrum) jest to, że techniki, takie jak TDD są wystarczające. Może tak być, ponieważ większość z głównej literatury Agile skupia się na małych, zlokalizowanych wspólnie, zespołach developerskich, pracujących nad dość prostymi systemami. Ale jeśli znajduje zastosowanie jeden lub więcej czynników skali (np. rozmiar dużego zespołu, zespoły rozproszone geograficznie, wymagania prawne czy złożona dziedzina), wtedy potrzebujesz bardziej wyszukanych strategii testowania. Bez względu na pewną retorykę, jaką mogłeś usłyszeć, spora liczba praktyków TDD radzi prywatnie co innego – jak zobaczymy w następnej sekcji.

Kto to robi?

Rysunek 15 podsumowuje wyniki jednego z pytań z Badania TDD wykonanego przez Ambyssoft w 2008 r. , które zadano społeczności TDD, o to, jakich technik testowych używają w praktyce. Ponieważ to badanie było wysłane do społeczności TDD, w ogóle nie reprezentuje dokładnie wskaźnika przyswojenia TDD, ale interesujący jest fakt, że respondenci jasno wskazywali, że wykonywali nie tylko TDD (albo że wszyscy wykonywali TDD, co zaskakujące). Wiele robiło również przeglądy i inspekcje, testowanie na końcu cyklu życia i niezależne równoległe testowanie, czynności, o których purytanie Agile rzadko dyskutują.

Praktyki testowania/walidacji wśród developerów Agile



Copyright 2008 Scott W. Ambler

Source: Ambyssoft 2008 TDD Survey
www.ambyssoft.com/surveys/tdd2008.html

Rysunek 6 Praktyki testowania/walidacji w zespołach Agile

Implikacje dla praktyków testowania

Jest kilka krytycznych implikacji dla profesjonalistów testowania:

Stań się specjalistą uogólniającym. Implikacją testowania „kompletnego zespołu” jest to, że większość obecnych profesjonalistów testów będzie musiało przygotować się do robienia czegoś więcej, niż samo testowanie, jeśli chcą być zaangażowani w projekty Agile. Tak, ludzie w niezależnych zespołach testowych nadal będą skupiać się ściśle na testowaniu, ale zapotrzebowanie na ludzi w tej roli jest o wiele mniejsze, niż zapotrzebowanie na ludzi z umiejętnościami testowymi będącymi aktywnymi członkami zespołów dostarczenia Agile.

Bądź elastyczny. Zespoły Agile przyjmują iteracyjne i kolaboracyjne podejście, które wspiera zmieniające się wymagania. Implikacją tego jest to, że minęły dni posiadania szczegółowych spekulacji wymagań, nad którymi się pracuje; teraz każdy zaangażowany w testowanie musi być dość elastyczny, by testować przez cały cykl życia, nawet jeśli wymagania się zmieniają.

Bądź przygotowany do bliskiej współpracy z developerami. Większość prac testowych jest wykonywana przez sam zespół dostarczenia Agile, nie przez niezależnych testerów.

Bądź elastyczny. To jest warte powtórzenia ;)

Skup się na czynnościach o wartości dodanej. Straciłem rachubę, ile razy słyszałem profesjonalistów testów lamentujących, że nigdy nie ma dość czasu ani zasobów przydzielonych do prac testowych. Kiedy badam, co ci ludzie chcą robić, odkrywam, że chcą czekać na szczegółowe spekulacje wymagań dostępne dla nich, chcą tworzyć dokumenty opisujące ich strategię testowania, chcą pisać szczegółowe plany testów, szczegółowe raporty defektów, i – tak – chcą nawet pisać i uruchamiać testy. Nic dziwnego, że nie mają dość czasu na wykonanie całej tej pracy! Czynnościami o prawdziwej wartości dodanej, które wykonują testerzy, jest znajdowanie a następnie komunikowanie potencjalnych defektów do ludzi odpowiedzialnych za ich naprawianie. Aby to robić, muszą oni stworzyć i wykonać testy; wszystkie inne czynności, które wymieniłem poprzednio są dla tej pracy co najwyżej pomocnicze. Czekanie na spekulacje wymagań to nie testowanie. Pisanie strategii i planów testowych to nie testowanie. Pisanie raportów defektów może mieć wartość, ale są lepsze sposoby komunikacji informacji, niż pisanie dokumentacji. Strategie Agile skupiają się na czynnościach o wartości dodanej i minimalizują, jeśli nie eliminują biurokratyczne marnotrawstwo, które jest systemowe w wielu organizacjach podążających za klasycznymi/tradycyjnymi strategiami.

Bądź elastyczny. To jest naprawdę ważne.

c.d.n.

W następnym numerze: Strategie testowania i jakości Agile: dyscyplina ponad retoryką. Część IV z IV: Strategie jakości Agile.