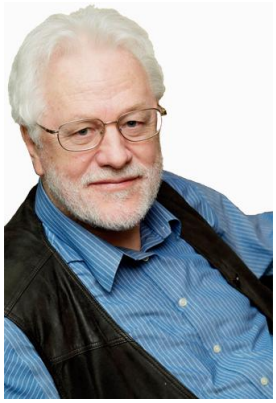
Zasady i wartości *Value-Driven Development* – Zwinność jest narzędziem, nie mistrzem

Część 1 z 2: Dziesięć kluczowych „zwinnych" zasad Gilba dla dostarczania wartości udziałowcom,unikania biurokracji i umożliwienia kreatywnej wolności.



Autor: Tom Gilb

O Autorze: Tom Gilb jest autorem dziewięciu książek i setek publikacji na tematy związane z inżynierią oprogramowania. Jego ostatnia książka – Competentive Engineering stanowi istotną definicję idei wymagań. Jego koncepcje dotyczące wymagań oparte są o CMMI poziomu 4. Tom gościnnie wykłada na uniwersytetach w Wielkiej Brytanii, Chinach, Indiach, USA, Korei i występuje jako prelegent na wielu znaczących międzynarodowych konferencjach.

Wprowadzenie

Manifest Agile [2] ma swoje *serce* we właściwym miejscu. Obawiam się tylko o jego "umysł". Jego pierwsza zasada „Naszym najwyższym priorytetem jest zadowolenie klienta poprzez wczesne i ciągłe dostarczanie wartościowego oprogramowania" jest centralną myślą tego artykułu. Nic dziwnego w tym, że zgadzam się, iż wielu praktyków Agile (np. Kent Beck) wyraźnie wskazuje na moją książkę z 1988r. jako źródło niektórych swoich idei [4, 5, 6, 7, 8, 9, 12].

Mój problem z Agile leży nie w jego pomysłach, ale w codziennym uczeniu i praktykach, które widzimy. To co się stało, jest tym samym problemem, który dotyka wszystkie projekty softwarowe i IT – na długo przed pojawieniem się Agile. Nasza technokratyczna kultura zapomniała o udziałowcach i ich wartościach. Praktyki są zbytnio „skupione na programiście" i zdecydowanie za mało skupione na „wartości udziałowca". Rezultatem jest to, że „działające oprogramowanie" [2] **jest dostarczone** do „klienta" [2].

Jednak konieczne *wartości* nie są koniecznie, a zwykle zbyt rzadko, *dostarczane* do *wszystkich krytycznych udziałowców*. Kod – sam w sobie – nie ma wartości. Możemy dostarczyć kod wolny od błędów, który ma zbyt mało, lub w ogóle nie ma żadnych przewidywanych wartości. Możemy dostarczyć do „klienta" funkcję systemu taką, jaką zdefiniowano w wymaganiach – ale ponieść całkowitą porażkę w dostarczeniu krytycznej *wartości* do wielu krytycznych *udziałowców.*

System biletowy nowej wspaniałej opery w Oslo jest tego prostym przykładem. W dawnych latach bilet był po prostu odrywany z bloczka i natychmiast dostarczany kupującemu. Nowy system - na otwarciu - spowodował,

że sprzedawcy biletów potrzebowali 10-20 minut na zorientowanie się, co trzeba robić i jak wykonać rzeczy takie, jak np. zniżki dla seniorów lub dzieci. Dodatkowo, często wymagana była obecność kierowników, którzy nie byli pewni co i jak. Żadnych błędów, wszystkie funkcje, ale coś było jednak katastrofalnie złe. „Działające oprogramowanie" – działające źle. Dzięki Bogu praktycy Agile nie zaprojektowali samego gmachu Opery! To było zrobione przez prawdziwych, pierwszej klasy, architektów! [3]

Obawiam się, że ten artykuł może nie zmienić ograniczeń społeczności programistów, a moje zasady stosują się do wyższego poziomu myślenia, niż kodowanie. Postaram się jednak sformułować o wiele jaśniejszy zbiór zasad, bardziej przejrzysty zestaw; a w kolejnym artykule, bardziej klarowne „wartości" niż te, które udało się spreparować praktykom Agile. Mam jedną zdecydowaną przewagę – nie jestem uzależniony od wspólnej polityki działań – tak jak oni byli – mogę więc bez przeszkód wyrażać swoją własna opinię. Dlatego niniejszym daję im specjalne pozwolenie na zmianę ich mętnych i niebezpiecznych ideałów na te bardziej skrystalizowane. A ponieważ oni zapewne nie będą przejawiać chęci (wyciosani w kamiennej, nie-zwinnej dokumentacji Manifestu) – daję czytelnikowi prawo do rozpowszechniania, aktualizacji i ulepszania tych zasad.

Zasady wartości

Oto podsumowane "Zasad wartości". Szczegółowy opis poniżej.

1. Kontroluj projekty przez kilka krytycznych i policzalnych wyników. Jedna strona - to wszystko! (żadnych historii, funkcji, właściwości, przypadków użycia, obiektów).
2. Upewnij się, że te rezultaty są rezultatami biznesowymi, nie technicznymi. Dostosuj projekt to interesów Twojego sponsora finansowego!
3. Daj wolność programistom, aby mogli dowiedzieć się, *jak* dostarczać owe rezultaty.
4. Oszacuj wpływ Twoich projektów, rozwiązań na *Twoje* cele ilościowe.
5. Wybierz projekty i rozwiązania o największym znaczeniu dla wartości przy uwzględnieniu kosztów - wykonaj je w pierwszej kolejności.
6. Podziel workflow na tygodniowe (lub o wartości 2% budżetu) ramy czasowe.
7. Zmieniaj projekty i rozwiązania opierając się na wartości ilościowej i koszcie implementacji.
8. Zmieniaj wymagania opierając się na wartości ilościowej, kosztach oraz nowych informacjach wejściowych.
9. Zaangażuj udziałowców – co tydzień – w ustalanie celów wartościowych.
10. Zaangażuj udziałowców – co tydzień – w faktyczne korzystanie z przyrostów wartości.

Oto ogólny opis **moich** Wartości Agile:

Prostota

1. Skoncentruj się na prawdziwych wartościach udziałowców.

Komunikacja

2. Komunikuj wartości udziałowców ilościowo.

3. Oszacuj oczekiwane wyniki i koszty w tygodniowych krokach i zdobądź mierzalną ilościowo informację zwrotną na temat oszacowań w tym samym tygodniu.
4. Wdrażaj prawdziwe ilościowe ulepszenia co tydzień – dla prawdziwych udziałowców.
5. Mierz krytyczne aspekty w ulepszonym systemie – co tydzień.
6. Analizuj odchylenie od szacowanych wartości i kosztu.

Odwaga

7. Zmieniaj plany tak, by odzwierciedlić cotygodniowe uczenie się .
8. Natychmiast wdrażaj najbardziej wartościowe potrzeby udziałowców – już w następnym tygodniu. Nie czekaj, nie studiuj (paraliż analizowania), nie tłumacz się. Po prostu to zrób!
9. Przekaż udziałowcom, jakie dokładnie ilościowe ulepszenia wprowadzisz w następnym tygodniu.
10. Użyj dowolnego projektu, strategii, metody, procesu, który działa ilościowo – by osiągnąć rezultaty.

Bądź inżynierem systemów, nie po prostu programistą (rzemieślnikiem). Nie bądź ograniczony swoim podłożem programistycznym w pracy dla swoich płatników.

A oto bardziej szczegółowe komentarze do powyższych zasad.

Zasada 1. Kontroluj projekty przez kilka krytycznych i policzalnych wyników. Jedna strona - to wszystko!

Większość z naszych tak zwanych wymagań funkcjonalnych, nie jest naprawdę wymaganiami. Są one rozwiązaniami (*designs*) mającymi spełnić niewyartykułowane, wysoko-poziomowe i krytyczne wymagania [14]. Większość prawdziwie krytycznych celów projektu jest prawie zawsze pogrzebana w slajdach i sformułowana w mętny i nietestowalny sposób. Nie są one nigdy używane w architekturze, kontraktowaniu lub testowaniu. To główna przyczyna porażek projektów [14]. Kierownictwo i sponsorzy projektu są zapewniani, że projekt dostarczy określonych usprawnień. W praktyce kultura Agile nie ma mechanizmów monitorowania i dostarczania oczekiwanych wartości. Scrum wysunie argument, iż jest to zdanie Właściciela Produktu (*Product Owner*). Ale nawet najwięksi guru Scrum otwarcie przyznają, że w praktyce taka sytuacja jest daleka od tego, jaka powinna być. Po prostu nie uczymy i nie praktykujemy potrzebnych mechanizmów. Ludzie z IT zawsze byli w tym słabi, a Agile nie dostarczyło swoich własnych jasnych ideałów.

Zasada 2. Upewnij się, że te rezultaty są rezultatami biznesowymi, nie technicznymi. Dostosuj projekt do interesów Twojego sponsora finansowego!

Ludzie nie rozwijają projektów aby uzyskać funkcje czy właściwości. A wydaje się to być głównym celem w obecnych metodach Agile. Potrzebujemy funkcji i być może „właściwości" by upewnić się, że aplikacja wykonuje podstawowe czynności biznesowe, jakich oczekiwano. Tak jak „kup bilet do opery". „Daj dziecku zniżkę". Ale te podstawy nie są nigdy głównymi motywami inwestowania w projekt IT. Zwykle udziałowcy już mają takie funkcje na miejscu, w obecnych systemach. Jeśli spojrzymy na dokumentację projektu, wynika z niej, że po prostu ktoś „sprzedał" kierownictwu lepszy system – parę usprawnień. Szybciej, taniej, bardziej wiarygodnie etc.

Takie rzeczy są zawsze specyficznie określonymi – i zawsze ilościowymi – ulepszeniami. Niestety my, w Agile, unikamy bycia precyzyjnymi na tym poziomie. Używamy przymiotników takich, jak „lepiej", „ulepszony", „udoskonalony" i zostawiamy to tak. Nauczyliśmy się dawno temu, że nasz klient jest zbyt niewykształcony, i za głupi (logika powinna skompensować brak wiedzy), by stawiać nam wyzwania na tych polach. I gotów jest z radością płacić dużo pieniędzy za system gorszy, niż już miał.

Musimy uczynić częścią naszej kultury produkcji: starannie analizowanie wymagań biznesowych ("oszczędzić pieniądze"), starannie analizowanie potrzeby udziałowców ("ograniczyć koszt szkolenia pracowników"), starannie analizowanie wymagania jakościowe aplikacji („lepsza użyteczność"). Musimy wyrażać te wymagania ilościowo. Musimy systematycznie wyodrębniać wymagania udziałowców z wymagań biznesowych. Musimy wyodrębniać wymagania jakościowe systemu z wymagań udziałowców. Następnie – musimy zaprojektować i stworzyć architekturę systemu taką, która dostarczy ilościowe poziomy wymagań na czas. Nie jesteśmy nawet blisko tego próbując to zrobić za pomocą konwencjonalnych metod Agile. Konsekwentnie więc ponosimy klęskę biznesową (znaczącą dla udziałowców) próbując dostarczyć produkt na wymaganym i określonym poziomie jakości [15].

Pozwólcie mi tu wyjaśnić. Możemy to zrobić, kiedy system ewoluuje, i może to być wyrażone na jednej stronie wymagań najwyższego poziomu [przykłady 14]. Więc nie próbujcie na mnie argumentu „zbędnej biurokracji"!

Zasada 3. Daj wolność programistom, aby mogli dowiedzieć się, *jak* dostarczać owe rezultaty.

Najgorszym scenariuszem, jaki mogę sobie wyobrazić jest taki, w którym pozwalamy prawdziwym klientom, użytkownikom i naszym własnym sprzedawcom dyktować programistom funkcje i właściwości, ostrożnie maskując je jako „wymagania klienta" (być może przekazane przez naszego Właściciela Produktu).

Jeśli zajrzysz pod powierzchnię tych fałszywych „wymagań", natychmiast zauważysz, że nie są one prawdziwymi wymaganiami. Są one naprawdę złym, amatorskim designem dla „prawdziwych" wymagań – domniemanych, lecz nie zdefiniowanych w dobry sposób [17]. Podałem wcześniej jeden przykład (prawdziwy), gdzie „Hasło" było wymagane, ale „Bezpieczeństwo" (czyli właściwe wymaganie) nie zostało w ogóle zdefiniowane. Jesteśmy w tym tak słabi, że możesz spokojnie założyć, iż niemal wszystkie tak zwane wymagania nie są prawdziwymi wymaganiami – są tylko złymi opracowaniami. Jedyne co musisz zrobić, by to zobaczyć, to zadać pytanie: **DLACZEGO?**.Dlaczego „Hasło" ? (Bezpieczeństwo głupku!) – aha! Odpowiedź zapewne będzie brzmieć: „Z powodu bezpieczeństwa!". A gdzie jest wymaganie odnośnie bezpieczeństwa? Nie tu (w wymagalności hasła), lub nawet gorzej – umieszczone w slajdach jako „Supernowoczesne Bezpieczeństwo" – i pozostawione dalej do zaprojektowania totalnym amatorom.

Spróbuj sobie wyobrazić, czy *Test Driven Development (TDD)* naprawdę przetestował poziomy jakości, takie jak poziomy bezpieczeństwa dla przykładu. Daleko do tego i TDD to kolejne rozczarowanie w skrzynce z narzędziami Agile.

Analizuję prawdziwe wymagania około raz w tygodniu, w środowisku międzynarodowym, i znajduję bardzo mało wyjątków – na przykład sytuacje, kiedy prawdziwe wymagania są zdefiniowane, zmierzone ilościowo,

później zaprojektowane (pod względem inżynierii, architektury). Kultura Agile nie ma żadnego pojęcia o prawdziwej inżynierii. Rzemieślnictwo [4] – owszem. Ale nie inżynieria – to rzecz kompletnie obca.

Nie możemy poprawnie zaprojektować niewyraźnego wymagania („Lepsze bezpieczeństwo"). Skąd mogę wiedzieć, czy hasło jest dobrym rozwiązaniem? Jeśli wymaganie bezpieczeństwa jest jasne i ilościowe (upraszczam tutaj), jak np. „Mniej niż 1% ryzyka spenetrowania systemu przez hakera w ciągu godziny", wtedy możemy podjąć inteligentną dyskusję o 4-znakowym kodzie, które ktoś uważa za dobre hasło.

Mam klienta (Confirmit [16]), który konsekwentnie odmawia zaakceptowania wymagań dotyczących funkcji i właściwości pochodzących od swoich klientów lub sprzedawców. Klient ten skupia się na kilku krytycznych cechach produktu (jak użyteczności, intuicyjność) i pozwala swoim programistom tworzyć rozwiązania techniczne takie, które w mierzalny sposób zrealizują ilościowe wymagania jakościowe.

Takie podejście powoduje, że właściwa praca (projekt) jest wykonana przez właściwych ludzi (programiści) i realizuje właściwe wymagania (ogólne, wyższego poziomu widoki wartości jakościowych w aplikacji). Programiści nawet wykonują „refactoring" poprzez iteracyjne realizowanie zbioru długofalowych wymagań jakościowych dotyczących utrzymywalności i testowalności. Czy to tylko zbieg okoliczności, że ich liderzy mają prawdziwe tytuły inżynierskie?

Zasada 4. Oszacuj wpływ Twoich projektów, rozwiązań na *Twoje* cele ilościowe.

Przyjmuję wymagania ilościowe za rzecz oczywistą. Podobnie inżynierowie. Praktycy Agile zdaje się, że nie słyszeli o koncepcji „jakości ilościowej". Koncepcja „projektu" również wydaje się dla nich obca. Jedyną wzmianką dotyczącą projektu lub architektury w Manifeście Agile jest „Najlepsze architektury, wymagania i projekty wynikają z samo organizujących się zespołów" [2]. W tej idei jest pewna zaleta. Jednak pogląd Agile na architekturę i projekt pomija większość kluczowych idei prawdziwej inżynierii i architektury [18].

Musimy projektować i tworzyć architekturę uwzględniając wielu udziałowców, wiele celów związanych z jakością i wydajnością, wiele ograniczeń i sprzecznych priorytetów. Musimy robić to w ewoluującym morzu zmian dotyczących wymagań, udziałowców, priorytetów, potencjalnych architektur. Proste wskazanie na „samo-organizujące się zespoły" to metoda daleka od podstawowych koncepcji opisujących, jak projektować i realizować złożone, wielkoskalowe krytyczne systemy. W rzeczy samej, to metoda nieodpowiednia nawet dla o wiele mniejszych systemów takich jak system Conformitu, przy którym pracuje 13 programistów [16].

Każdy proponowany projekt lub architektura muszą być porównane liczbowo, z oszacowaniami, następnie z pomiarami, jak dobrze realizują wymagania jakościowe i te, związane z wykonywaniem; oraz do jakiego stopnia pochłoną zasoby, czy też mogą naruszyć ograniczenia. Ja rekomenduję użycie tabeli Estymacji wpływu (*Impact Estimation*) jako podstawowej metody wykonywania takiego liczbowego porównania wielu projektów do wielu wymagań [19, 10, 4]. Udowodniono że metoda ta jest spójna z ideałami i praktykami Agile i daje wyniki o wiele lepiej zaraportowane, niż inne metody [16]. Nawet jeśli inne metody mają lepsze wyniki, nie umożliwiają przedstawienia ich w odpowiedni sposób, ponieważ nie operują na wartościach liczbowych wyznaczników wartości i jakości udziałowców. Jeżeli projektant nie jest w stanie oszacować wpływu proponowanego projektu na nasze wymagania, oznacza to, iż nie jest on kompetentny i nie powinien być

zatrudniony. Większość projektantów oprogramowania jest więc według tej definicji niekompetentnych. Nie tylko nie radzą sobie z oszacowaniem, ale nawet nie pojmują faktu, że powinni spróbować!

Zasada 5. Wybierz projekty i rozwiązania o największym znaczeniu dla wartości przy uwzględnieniu kosztów - wykonaj je w pierwszej kolejności.

Załóżmy, że uznamy powyższe stwierdzenie (iż powinniśmy szacować i mierzyć potencjalny i rzeczywisty wpływ projektów i architektury na nasze wymagania) za zdroworozsądkowe. Wysuwam argument, że nasza podstawowa metoda decydowania „które projekty zastosować" powinna opierać się na stwierdzeniu, które z nich mają najlepszy stosunek wartości do kosztu. Scrum, jak inne metody, skupia się wąsko na oszacowaniu wysiłku. To nie jest to samo, co szacowanie wielu wartości przyczyniających się do dziesięciu najważniejszych celów projektu (co Estymacja Wpływu realizuje rutynowo) [19]. Wydaje mi się dziwne, że metody Agile rozumieją podrzędną koncepcję szacowania kosztów, ale nigdy nie podejmują nadrzędnej koncepcji szacowania wartości ważnych dla udziałowców oraz ich wymagań. Zarządzanie kosztem ma mało sensu, jeśli nie umiesz wpierw zarządzać wartością. Głębszym problemem są tu prawdopodobnie nie metody Agile, ale kompletna porażka naszych uczelni biznesowych w uczeniu menedżerów o czymś więcej, niż finansach – o jakości i wartościach [20]. Gdyby menedżerowi byli czujni i wyważeni, zażądaliby o wiele lepszego rozliczania się z wartości dostarczanych przez programistów i projekty IT. Jednakże społeczność programistów już dawno uświadomiła sobie, że kierownictwo „śpi w pracy" – i leniwie czerpie z tego korzyści.

Zasada 6. Podziel workflow na tygodniowe (lub o wartości 2% budżetu) ramy czasowe.

Na jednym poziomie ja i praktycy Agile zgadzamy się – aby dzielić duże projekty na mniejsze kawałki, tygodniowe lub o zbliżonej długości, co jest o wiele lepsze, niż podejście kaskadowe (*Waterfall*) lub wielkiego wybuchu (*Big Bang*) [21].

Jednak będę twierdził, że musimy zrobić więcej, niż dzielić ze względu na "priorytety wymagań określone przez *Product Ownera*". Najpierw musimy podzielić sam przepływ wartości – nie tylko *story*/ funkcje/ przypadki użycia. To dzielenie wartości jest podobne do poprzedniej zasady ustalania priorytetów projektów pod względem wartości i kosztu. Musimy wybrać, jakie największe wartości możemy dostarczyć w małym kroku (nasz zespół pracujący przez tydzień) w ciągu następnego tygodnia (kolejny krok dostarczenia wartości udziałowcom). W zasadzie jest to właśnie to, co powinien robić *Product Owner* w Scrum. Jednak nie sądzę, by byli oni (Właściciele Produktów) nawet w znikomym stopniu wyszkoleni, aby robić to dobrze. Po prostu – nie mają ilościowych wymagań dotyczących wartości (patrz wyżej) oraz ilościowych oszacowań projektu (patrz wyżej), by wykonywać te zadania w logiczny sposób.

Jedyna rozmowa, której nie wyobrażam sobie z praktykami Agile jest o tym, czy można rzeczywiście podzielić jakikolwiek projekt na małe (2% budżetu) kroki. Ja odkryłem, że zawsze można, ale jest wielu ludzi silnie fałszywie przekonanych o tym, że to niemożliwe [21]. Może jednak taka dyskusja wyniknie, kiedy praktycy Agile zostaną skonfrontowani z potrzebą dzielenia systemu według wartości, nie według funkcji.

Zasada 7. Zmieniaj projekty i rozwiązania opierając się na wartości ilościowej i koszcie implementacji.

Jeśli dostajemy liczbową informację zwrotną o aktualnie dostarczonej wartości designu, porównaną z szacowaną wartością (jak czyni się w Confirmit [16]), z reguły będziesz rozczarowany osiągniętym wynikiem. To daje okazję ponownego rozważenia designu lub jego implementacji tak, by osiągnąć wartość jakiej potrzebujesz, bez względu na wcześniejszy brak jej zrozumienia. Możesz się nawet nauczyć, że „kodowanie samo w sobie to nie wszystko", by dostarczyć wartość udziałowcom.

Obawiam się, że ta możliwość jest w znacznej mierzej stracona – ponieważ metody Agile ani nie określają ilościowo wymaganej wartości, ani wartości oczekiwanej od danego kroku czy projektu na danym kroku dostarczenia systemu. Rezultat jest taki, że utkniemy ze złymi projektami aż do momentu, w którym jest za późno. Nie wydaje mi się, aby było to bardzo „zwinne".

Zasada 8. Zmieniaj wymagania opierając się na wartości ilościowej i koszcie oraz nowych informacjach wejściowych.

Czasami wymagania ilościowe dotyczące jakości i wartości są zbyt ambitne. Zbyt łatwo jest marzyć o perfekcji – a niemożliwe jest rzeczywiste jej osiągnięcie. Zbyt łatwo jest marzyć o wielkich ulepszeniach bez świadomości ich rzeczywistych kosztów czy ograniczeń. Czasami trzeba przyjąć rzeczywistość – to co możemy zrobić, lub czego możemy wymagać bazując na praktycznych doświadczeniach. Taki jest bieg normalnej inżynierii i nauki. Krok po kroku uczyć się rzeczywistości technicznej i ekonomicznej.

Jednak społeczność Agile, jak już zaznaczyliśmy, ma niewielkie pojęcie o ilościowym określaniu jakichkolwiek wymagań. Konsekwentnie nie mogą przyjąć tego, co jest rzeczywiste. Otrzymują tylko to, co mogą przypadkowo lub przez nawyk uzyskać.

Jeśli określaliby ilościowo swoje kluczowe wymagania i jeśli mierzyliby (w liczbach) wyniki przyrostów, w końcu – jeśli wymagania nie byłby zbyt ambitne lub kosztowne - mielibyśmy szanse reagować szybko (zwinnie!). Uczenie się i zwinne zmiany są zagrożone przez brak kwantyfikacji i pomiarów w normalnym procesie developmentu. Lecz dzisiejsza społeczność Agile pozostaje niefrasobliwa.

Zasada 9. Zaangażuj udziałowców – co każdy tydzień – w ustalanie celów wartościowych.

Metody Agile odnoszą się do użytkowników i klientów. Używane są terminy „sponsorzy", „programiści", „użytkownicy", „klienci". W inżynierii systemów niewątpliwie standardową koncepcją jest „udziałowiec". Niektóre elementy inżynierii oprogramowania przyjęły paradygmat udziałowca [22]. Jednak metody Agile nie wspominają o tej koncepcji. W prawdziwych projektach, średniej wielkości, istnieje od 20 do 40 interesujących ról udziałowców wartych rozważenia. Udziałowcy są źródłem kluczowych wymagań. Microsoft nie przejmował się zbytnio udziałowcem zwanym EU (Unia Europejska) – co okazało się kosztowną pomyłką. W każdym upadłym projekcie – a mamy ich stanowczo za dużo –w podstawach znajdziesz problem z udziałowcami. Udziałowcy mają priorytety, a ich różne wymagania mają inne priorytety. Musimy systematycznie je nadzorować. Wybaczcie, jeśli wymaga to wysiłku umysłowego. Nie możemy być leniwi ipóźniej ponosić

porażkę. Wątpię, że *Product Owner* w Scrum jest szkolony lub uposażony tak, by radzić sobie z bogactwem udziałowców i ich potrzeb. W rzeczywiści *PO* wydaje się tu być niebezpiecznym wąskim gardłem.

Jednak analiza wszystkich udziałowców i ich potrzeb, priorytetów owych potrzeb, nie zawsze może być prostą sprawą. Fakt dostarczania wartości w nieprzerwany sposób zmieni potrzeby i priorytety. Zewnętrzne środowisko udziałowców (politycy, konkurenci, nauka, ekonomia) będzie stale zmieniać swoje priorytety a nawet zmieniać to, kim ci udziałowcy są. Tak więc musimy trzymać pewnego rodzaju linę do prawdziwego świata – i to stale. Musimy próbować „wyczuć" wymagania o nowych priorytetach, jak tylko się pojawią przed wczesnymi zwycięzcami. Nie wystarczy myśleć o wymaganiach jak o prostych funkcjach i przypadkach użycia. Najbardziej krytyczne i wszechobecne wymagania są wymaganiami ogólnej jakości systemu i istnieją liczbowe poziomy aspektów („ilities") krytycznie trudnych do dostosowania tak, by były w równowadze z innymi rozważaniami.

Zręczny biznes w rzeczy samej, ale – czy zamierzamy naprawdę być "zwinni"? Jeśli tak – musimy być realistami – a obecne metody Agile nawet nie rozpoznają koncepcji udziałowca. Chowam głowę w piasek, jeśli mnie zapytasz!

Zasada 10. Zaangażuj udziałowców – co każdy tydzień – w tworzenie przyrostów wartości.

Wreszcie – udziałowcy to ci, którzy powinni przyrostowo otrzymywać wartości, w każdym przyroście developmentu. Wierzę, że to powinno być celem każdego przyrostu. Nie „dostarczanie działającego kodu do klientów". Oznacza to, że musimy dokładnie rozpoznać, który typ udziałowca ma otrzymać określone usprawnienie (wartość), planować dostarczenie tej wartości (lub jej użytecznej części) w danym przyroście oraz poddawać dostarczoną wartość ocenie. Obecne metody Agile nie umożliwiają tego a w rzeczywistości wydają się nie dbać w ogóle o wartości czy udziałowców.

W rzeczywistości programiści powinni zanalizować cały system, nie tylko kod, aby dostarczyć prawdziwą wartość – a programiści czują się bardzo niekomfortowo w czymkolwiek poza ich wąską dziedziną. Czyż nie jest zadziwiające, że to właśnie "menedżerowie" dali im tyle władzy, by skłócić środowisko?

Referencje

1. Gilb's Agile Principles and Values

   Wstęp do pełnego artykułu. Sformułowany na potrzeby konferencji XP Days Keynote w Londynie w listopadzie 2004. prezentacja jest ciągle dostępna pod http://xpday4.xpday.org/slides.php. Slajd 38-39 mówi o "*Principles and Values statements"*. Dlaczego powołuję się na materiał z 2004? Napisałem wartości i zasady na konferencji tuż przed moim wystąpieniem i najpierw pojawiły się w prezentacji. Zaktualizowałem zasady by położyć nacisk na „wartości, udziałowców i koszty" w 2007 i w 2010 – na użytek niniejszego artykułu.
2. Manifest Agile: URL http://agilemanifesto.org/principles.html
3. http://en.wikipedia.org/wiki/Oslo_Opera_House

4. Gilb, Principles of Software Engineering management, 1988. http://books.google.co.uk/books?q=gilb+principles+of+software+engineering+management&spell=1&oi=spell
5. Mike Cohn, "Zawsze uważałem Toma za prawdziwego praktyka Agile. W 1981r napisał o krótkich iteracjach (każda powinna nie trwać dłużej niż 2% całego harmonogramu projektu). To było na długo przed tym, jak reszta z nas to odkryła" http://blog.mountaingoatsoftware.com/?p=77
6. Komentarz Kenta Becka do artykułu Toma Gilba, Principles of Software Engineering Management: " A strong case for evolutionary delivery – small releases, constant refactoring, intense dialog with the customer". (Beck, strona 173). W mailu do Toma, Kent pisze "Cieszę się, że ja i ty mamy wspólne ideały. Ukradłem dość twoich, że byłbym rozczarowany, że ich nie mamy ☺, Kent" (2003)
7. Jim Highsmith (sygnatariusz Manifestu Agile) skomentował: "W latach 1980 szczególnie dwóch ludzi było pionierami ewolucji podejścia developmentu iteracyjnego - Barry Boehm ze swoim Modelem Spiralnym i Tom Gilb z modelem *Evo*. Czerpałem z idei Boehma i Gilba wczesną inspirację w tworzeniu Adaptacyjnego Wytwarzania Oprogramowania (ang. Adaptive Software Development). …. Gilb przed długi czas uważał to za bardziej dokładną (ilościową) wycenę w pozyskiwaniu wartości i zwiększaniu ROI" (strona 4, July 2004).
8. Ward Cunningham napisał w kwietniu 2005: Tom – dziękuję za udostępnienie twojej pracy. Mam nadzieję, że znajdziesz wartość w naszych. Cieszę się, że społeczność Agile przykłada uwagę do twojej pracy. Wiemy (teraz), że byłeś o wiele dalej, niż większość z nas. Najlepsze pozdrowienia - Ward, http://c2.com
9. Robert C. Martin (pierwszy sygnatariusz Manifestu Agile Manifesto, aka Wujek Bob): "Tom i ja rozmawialiśmy o wielu rzeczach i wiele się od niego nauczyłem. Kwestia, która najbardziej utkwiła mi w pamięci to definicja postępu", „Tom wynalazł zasadę planowania, nazwaną przez niego Planguage, która wychwytuje ideę potrzeby klienta. Myślę, że przeznaczę trochę czasu na przestudiowanie tego" z http://www.butunclebob.com/ArticleS.UncleBob.TomGilbVisit
10. Gilb, Competitive Engineering, 2005, http://books.google.co.uk/books?q=gilb+competitive+engineering&btnG=Search+Books
11. Scott Ambler w *Amazon reviews*, o *Competitive Engineering*: Tom Gilb, ojciec metodologii *Evo* dzieli się w tej książce swoim praktycznym, życiowym doświadczeniem z umożliwiania efektywnej współpracy pomiędzy programistami, menedżerami i udziałowcami. Mimo że książka szczegółowo opisuje *Planguage*, specyficzny język inżynierii systemowej, same rady metodologiczne są warte ceny książki. *Evo* jest jedną z naprawdę niedocenionych zwinnych metodologii i jako efekt prowokującej pracy Gilba nie jest tak dobrze znana, jak powinna być, choć podejrzewam, iż zmieni się to z tą książką. Książka opisuje efektywne praktyki specyfikowania wymagań i projektów, wysoce kompatybilne z zasadami i praktykami Modelowania Agile, obejmuje czynności planowania, jakość i szacowanie wpływu. Podejrzewam, że ta książka będzie jedną z „obowiązkowych" książek w tworzeniu oprogramowania w 2006r.

12. http://leansoftwareengineering.com/2007/12/20/tom-gilbs-evolutionary-delivery-a-great-improvement-over-its-successors/

    "Ale jeśli naprawdę chcesz pójść krok do przodu, powinieneś przeczytać Toma Gilba. Pomysły wyrażone w Principles of Software Engineering Management nie są całkiem dopasowane do ADD, które mogą używać dzisiejsi programiści, lecz bez wątpienia myślenie Gilba o definicji wymagań, wiarygodności, projekcie, inspekcjach kodu i metrykach projektowych są poza większością obecnych praktyk" Corey Ladas
13. Re Security Requirements: http://www.gilb.com/tiki-download_file.php?fileId=40 Artykuł o tym, jak określać ilościowo wymagania bezpieczeństwa.
14. Cele wysokiego poziomu (*Top Level Objectives*): http://www.gilb.com/tiki-download_file.php?fileId=180 podręcznik pełen prawdziwych studiów przypadku dotyczących wysokopoziomowych wymagań projektowych lub ich braku.
15. Przykład systematycznej ilościowej i połączenia poziomów wymagań biznesowych, udziałowca oraz jakościowych w *Norwegian Post case study* od Kai Gilba. http://www.gilb.com/tiki-download_file.php?fileId=277
16. Przypadek Confirmit: Artykuł http://www.gilb.com/tiki-download_file.php?fileId=32

    Przypadek Confirmit: prezentacja http://www.gilb.com/tiki-download_file.php?fileId=278
17. Prawdziwe wymagania (*Real requirements)*: jak określić, jakie są prawdziwe wymagania, artykuł. http://www.gilb.com/tiki-download_file.php?fileId=28
18. Architektuta http://www.gilb.com/tiki-download_file.php?fileId=47
19. Ocena projektu (*Design Evaluation*): Szacowanie krytycznych wpływów wykonywania i kosztów względem projektów http://www.gilb.com/tiki-download_file.php?fileId=58
20. *Hopper: The Puritan Gift: Reclaiming the American Dream Amidst Global Financial Chaos'*, http://www.puritangift.com/

    To nie ostatni bezpośredni i głęboki atak na szkoły biznesowe, by uczyły znacznie więcej, niż o finansach zapominając o szerszym zestawie wartości, które prowadzą do długoterminowej finansowej solidności. http://twitter.com/puritangift
21. Dekompozycja: http://www.gilb.com/tiki-download_file.php?fileId=41
22. Artykuły Susanne Robertson dotyczące udziałowców http://www.systemsguild.com/GuildSite/Guild/Articles.html