Agile. Nowy paradygmat rozwoju oprogramowania

1 WPROWADZENIE DO AGILE

CO TO JEST AGILE?

Agile to umiejętność tworzenia i reagowania na zmiany, aby odnieść sukces w niepewnym i burzliwym środowisku. (Źródło: www.agilealliance.org)

Agile Software Development jest ogólnym pojęciem zestawu metod i praktyk opartych na wartościach i zasadach wyrażonych w Manifeście Agile. Rozwiązania ewoluują dzięki współpracy między samoorganizującymi, międzyfunkcyjnymi zespołami wykorzystującymi odpowiednie praktyki w ich kontekście. (Źródło: www.agilealliance.org)

AGILE MANIFESTO

Na początku 2001 r. Grupa siedemnastu niezależnych myślących programistów - zwana Agile Alliance - spotkali się, aby zdefiniować Agile i nakreślić rozwój i dostarczanie frameworków Agile. Byli zmotywowani do stworzenia zwinnego środowiska, które umożliwiłoby szybkie projektowanie, opracowywanie i dostarczanie oprogramowania. Branża do tej pory (i wciąż jest) borykała się z bardzo powolnym dostarczaniem oprogramowania oraz powiązanymi procesami i metodami. Początkowe działania Agile Alliance pozwoliły na stworzenie w pierwszej kolejności Manifestu Zwinnego Sojuszu. Mimo że w manifeście napisano "Manifest dla zwinnego rozwoju oprogramowania", wartości i zasady Agile można z łatwością zastosować do wielu rodzajów opracowywania produktów.

Manifest z Agile Alliance:

Manifest dla Agile Software Development

Odkrywamy lepsze sposoby tworzenia oprogramowania, robiąc to i pomagając innym. Dzięki tej pracy doszliśmy do wartości

• **Osoby i interakcje** nad procesami i narzędziami

• **Oprogramowanie do pracy** nad obszerną dokumentacją

• **Współpraca z klientem** nad negocjacją umowy

• **Reagowanie na zmiany** w następstwie planu

Oznacza to, że chociaż w pozycjach po prawej stronie znajduje się wartość, cenimy przedmioty po lewej więcej

Poniższe 12 zasad opiera się na Agile Manifesto:

1. Naszym najwyższym priorytetem jest zadowolenie klienta poprzez wczesną i ciągłą dostawę cennego oprogramowania.

2. Witamy zmieniających się wymagań, nawet późno w rozwoju. Zwinne procesy zmiany uprzęży na korzyść klienta.

3. Regularnie dostarczaj działające oprogramowanie, od kilku tygodni do kilku miesięcy, z preferencją do krótszej skali czasowej.

4. Ludzie biznesu i programiści muszą współpracować codziennie przez cały czas trwania projektu.

5. Buduj projekty wokół zmotywowanych osób. Zapewnij im środowisko i wsparcie, którego potrzebują, i zaufaj im, aby wykonali zadanie.

6. Najbardziej wydajny i skuteczny sposób przekazywania informacji do i w ramach zespołu programistów to rozmowa twarzą w twarz.

7. Oprogramowanie robocze jest podstawową miarą postępu.

8. Zwinne procesy promują zrównoważony rozwój. Sponsorzy, deweloperzy i użytkownicy powinni mieć możliwość utrzymywania stałego tempa w nieskończoność.

9. Ciągła dbałość o doskonałość techniczną i dobry design zwiększa zwinność.

10. Prostota - sztuka maksymalizacji ilości pracy, która nie została wykonana - jest niezbędna.

11. Najlepsze architektury, wymagania i projekty powstają w samoorganizujących się zespołach.

12. W regularnych odstępach czasu zespół zastanawia się, jak stać się bardziej skutecznym, a następnie odpowiednio dostosowuje i dostosowuje swoje zachowanie.

WARTOŚCI AGILE

Cztery instrukcje lub wartości Agile Manifest zostały krótko opisane poniżej:

• Osoby i interakcje nad procesami i narzędziami. Ludzie są najważniejszym zasobem każdej organizacji i kluczowym składnikiem każdego udanego przedsięwzięcia. Najczęściej spotykaną w branży trylogią są ludzie, procesy i technologie (lub narzędzia). Spośród tych trzech najważniejsze są ludzie. Procesy a narzędzia mogą nam zająć tyle czasu i nie są wystarczające, by uratować projekt przed niepowodzeniem. Potrzebny jest zespół silnych graczy i różnorodnych umiejętności. Ludzie lepiej rozumieją, jaką wartość mogą dostarczyć klientowi; dlatego najlepiej nadają się do wykorzystania procesów i narzędzi do realizacji tej wartości. Kluczowym aspektem ludzi jest umiejętność współpracy z innymi. Silny członek zespołu może nie być ekspertem w swojej dziedzinie. Ale ważna jest umiejętność dobrej współpracy z innymi, komunikowania się i interakcji z nimi w przejrzysty sposób. Budowanie właściwego zespołu jest ważniejsze niż budowanie środowiska dzięki procesom i narzędziom. Dlatego w Agile osoby i interakcje są cenione bardziej niż procesy i narzędzia.

• Oprogramowanie do pracy nad obszerną dokumentacją

Każdy program ma dołączoną dokumentację (np. Użytkownika technicznego

przewodnik). Niestety, czasem większy nacisk kładziony jest na pisanie obszernej dokumentacji niż na kodowaniu działającego oprogramowania. Klienci czerpią wartość z produktu, który odpowiada ich potrzebom, a nie dobrze zarządzany dokument. Lepiej mieć działający produkt programistyczny, ale słabą dokumentację pomocniczą. Podczas gdy kluczową dokumentacją jest przydatna dokumentacja (oprogramowanie bez dokumentacji jest katastrofą), należy skupić się na pisaniu działającego kodu. Jeśli celem dokumentacji jest lepsze przekazywanie wiedzy pomiędzy członkami zespołu, jakie są lepsze sposoby osiągnięcia tego, niż napisanie kodu, który jest czytelny dla człowieka i pomaga zespołowi. Dlatego też Agile kładzie większy nacisk na pisanie kodu, który zapewnia wartość poprzez regularne, widoczne, przyrostowe dostarczanie działającego produktu. Agile również mocno podkreśla interakcję człowiek-człowiek, która jest uważana za najbardziej efektywny sposób przekazywania wiedzy o budowanym produkcie. Dokumentacja jest ważna, ale musi być krótka i ważna; musi być humanizowalny, aby mógł efektywnie opisywać system, jego struktury na najwyższym poziomie i racjonalny pod względem podejmowanych decyzji projektowych. Dlatego w Agile oprogramowanie robocze jest bardziej skoncentrowane niż obszerna dokumentacja.

• Współpraca z klientem nad negocjacją umowy

Skuteczny produkt powstaje, gdy istnieje ścisła i regularna współpraca między klientem a dostawcą. Wyobraź sobie budowanie swojego wymarzonego domu. Wykonujesz wszystkie prace przygotowawcze, aby zidentyfikować doświadczonego budowniczego. Negocjujesz z nim, aby utworzyć umowę, która określi Twoje wymagania, harmonogram i koszt. Obie strony podpisują umowę; pozostawiasz mu to, aby ukończyć pracę i wybrać się na wielomiesięczną wycieczkę po świecie, mając nadzieję, że twój wymarzony dom będzie gotowy do powrotu. Mijają miesiące i czas odwiedzić nowy dom. Horror okropności. Jesteś zszokowany widząc, że dom jest nie tylko niekompletny; Projekt jest odbiegający od tego, który podałeś, koszt wzrósł ponad twój budżet, a harmonogram wykroczył daleko poza to, co zostało uzgodnione w umowie. Jeszcze bardziej zabawne jest to, że budowniczy obwinia cię za wszystko; nie wziąłby żadnej odpowiedzialności za fiasko. W tym momencie nie jesteś nawet pewien, kiedy dom będzie gotowy i przekazany tobie. Brzmi znajomo? Zwykle dzieje się tak z rozwojem oprogramowania. Zbyt wiele projektów związanych z tworzeniem oprogramowania zawodzi głównie z powodów wymienionych powyżej. Pomyślne projekty polegają na regularnych i częstych opiniach klientów. Klient ściśle współpracuje z zespołem programistów, zapewniając częste informacje zwrotne na temat swojej pracy podczas podróży. Klient jest świadomy, co jest opracowywane, a deweloperzy wymagań klienta. Nie ma najmniejszego zaskoczenia w kryteriach testu akceptacji klienta ani zmian wymagań. Nie chodzi o to, że umowy nie są negocjowane i tworzone. W Agile zapisywane są umowy, które regulują sposób, w jaki klienci i dostawcy będą współpracować dla wzajemnie zadowalających wniosków. Sprawna umowa koncentruje się na umożliwieniu inspekcji i adaptacji produktu, priorytetyzacji i współpracy wszystkich zainteresowanych stron. Dlatego w Agile współpraca z klientem jest bardziej obciążona niż negocjacje kontraktowe.

• Reagowanie na zmiany w następstwie planu

Udana organizacja to taka, która może reagować na zmiany szybciej niż jej konkurenci. Podobnie, udane projekty oprogramowania to takie, które mogą lepiej reagować na zmiany. Często tworzymy plan z nadzieją, że przetrwa próbę czasu. Ale podstawową rzeczywistością jest to, że żaden plan nie jest wystarczająco inteligentny, by brać pod uwagę przyszłość. Wszystko zmienia się z każdą chwilą. A zatem, wykonanie na planie, który przechodzi niewidzialne zmiany, oznacza ustawienie się na porażkę. Plan projektu oprogramowania jest również taki jak powyżej; nie można jej zaplanować zbyt daleko w przyszłość. Rzeczy zawsze się zmieniają. Warunki w zmianach na rynku, zmiany strategii biznesowych, zmiany wymagań oprogramowania. Ludzie i koszt rzeczy się zmienia; z powodu nieprzewidzianych wydarzeń harmonogram może się zmienić. Dlatego jest prawie niemożliwe dostarczyć skutecznego oprogramowania, jeżeli mechanizm zmiany nie jest w nim wbudowany. Lepsza strategia planowania to taka, w której plan jest tworzony wokół pewności, niepewności, i najlepiej zgadnij. Wiemy na pewno, co wydarzy się w ciągu najbliższych dwóch tygodni; domyślamy się, co może się wydarzyć wstępnie w ciągu następnych 2-8 tygodni; i możemy zgadywać, co może się wydarzyć po upływie 8 tygodni. Wymagania i funkcjonalność projektu są również zgodne z tą samą ścieżką widoczności. Na pewno wiemy, jakie są wymagania w ciągu najbliższych 2-4 tygodni; z grubsza znamy wymagania na następne trzy miesiące; i ledwo znamy wymagania na 12 miesięcy Dlatego Agile uważa, że z góry planami projektu nie są plany zobowiązań; to jest zrozumiałe, że plany są jedynie najlepszymi przypuszczeniami opartymi na doświadczeniu i faktach; a zmiana jest nieunikniona wraz z rozwojem projektu. Przyrostowy rozwój oprogramowania oparty na krótkoterminowym szczegółowym planie jest lepszym składnikiem sukcesu. Stąd w Agile plany programów / projektów są określane na wysokim poziomie, mając na uwadze, że plany mają ulec zmianie. Agile zaleca, aby plan został zbudowany przy założeniu, że jest elastyczny i gotowy do dostosowania się do zmian w środowisku biznesowym lub technicznym. Dlatego też, w Agile, reagowanie na zmiany jest lepszym planem niż zastosowanie nieelastycznego planu.

ZASADY AGILE

1. Naszym najwyższym priorytetem jest zadowolenie klienta poprzez wczesne i ciągłe dostarczanie cennego oprogramowania. Zamiast podejścia big bang, Agile koncentruje się na dostarczaniu klientom wartości przyrostowej poprzez dostarczanie oprogramowania w sposób częsty, spójny, regularny i przyrostowy. Agile obala mit, że im więcej funkcjonalności jest pakowane w produkt, tym wyższa będzie jakość produktu. Wręcz przeciwnie, Agile uważa, że jakość produktu jest odwrotnie proporcjonalna do poziomu funkcjonalności. Im mniej funkcjonalna jest dostawa początkowa, tym wyższa jakość produktu końcowego. Jako konsekwencja Agile uważa również, że im częstsze dostawy, tym wyższa ostateczna jakość. To ostatecznie doprowadzi do bardziej zadowolonego i zaangażowanego klienta. Prawdopodobnie zwiększy to prawdopodobieństwo zaakceptowania dostarczanego produktu. A jeśli coś jest nie tak, chętnie raportują o zmianach, które chcą rozwiązać.

2. Witamy zmieniających się wymagań, nawet późno w rozwoju. Zwinne procesy zmiany uprzęży na korzyść klienta. Zwinny zespół zachęca i przyjmuje zmiany, jeśli zmiany są wymagane regularnie i przyrostowo. Zespół postrzega zmiany w wymaganiach jako zdrowe, ponieważ docenia fakt, że zespół biznesowy (właściciel produktu) zdobył więcej wiedzy

warunki na rynku, które doprowadziły do zmian. Zwróć uwagę, że Agile koncentruje się na współpracy między klientem a zespołem programistycznym. Skuteczne reagowanie na zmiany i dostarczanie klientom wartości przyrostowej są kluczowymi rezultatami takiej współpracy. W przypadku elastycznego dostarczania wymagania biznesowe są podzielone na mniejsze elementy lub komponenty, które można dobrze zrozumieć, zdefiniować, zaprojektować, przetestować i dostarczyć. Zespół utrzymuje elastyczną architekturę oprogramowania, dzięki czemu przy wprowadzaniu zmian wpływ na produkt jest minimalny. Takie podejście znacznie zmniejsza ryzyko złej jakości dostarczanego produktu;

3. Regularnie dostarczaj działające oprogramowanie, od kilku tygodni do kilku miesięcy, preferując krótszy czas. Zwinne frameworki koncentrują się na dostarczaniu działającego oprogramowania, wcześnie (po kilku pierwszych tygodniach) i regularnie (co kilka tygodni), które zaspokajają potrzeby klienta. Metryka kluczowa, która jest używana, to "koszt opóźnienia". Oznacza to, jaki jest koszt dla firmy, która nie wykona żądanej funkcji lub funkcji produktu. Pozwala to klientowi zidentyfikować koszt alternatywny wymagań. Najcenniejsze cechy produktu mogą nie być najlepszą decyzją ekonomiczną. Dzięki tym danym klienci mogą zrozumieć koszt opóźnienia implementacji jednej funkcji w stosunku do innej.

4. Ludzie biznesu i programiści muszą współpracować codziennie przez cały czas trwania projektu. Współpraca jest kluczową zasadą Agile. Agile zachęca do bezpośredniej komunikacji i częstych interakcji między klientem, programistami i interesariuszami, używając języka łatwego do zrozumienia dla wszystkich. Celem jest wyeliminowanie wszystkich niespodzianek i niejednoznaczności. Projekt oprogramowania wymaga stałego monitorowania i wskazówek, a współpraca to zapewnia.

5. Buduj projekty wokół zmotywowanych osób. Zapewnij im środowisko i wsparcie, którego potrzebują, i zaufaj im, aby wykonali zadanie. Agile uważa zespół właściwych ludzi za najważniejsze czynniki skutecznego dostarczania oprogramowania. Drugi czynnik, taki jak proces, otoczenie, zarządzanie i atmosfera, są uważane za drugorzędne, a zatem te czynniki opierają się na wymaganiach drużyna. Agile wierzy w wzmocnienie pozycji zespołu, aby był zmotywowany i produktywny. Jeśli członkowie zespołu mogą sobie ufać, Agile może odnieść sukces

6. Najbardziej wydajny i skuteczny sposób przekazywania informacji do i wewnątrz zespół programistów to rozmowa twarzą w twarz. W Agile domyślnym trybem komunikacji jest rozmowa twarzą w twarz. Wykorzystanie e-maili i innego rodzaju komunikacji poza człowiekiem jest ograniczone. Nawet w sytuacjach, w których zespoły są rozproszone geograficznie, zachęca się do współpracy wirtualnej lub kolokacji. Pisemna komunikacja nie jest w stanie przekazać wszystkiego; jest podatny na nieporozumienia. Wiele komunikacji odbywa się poprzez język ciała i wizualne wskazówki. Dlatego Agile podkreśla znaczenie komunikacji twarzą w twarz. Zwinny zespół (typowa wielkość zespołu wynosi od 3 do 10 osób) członkowie spotykają się codziennie przez 10-15 minut. Ten czat nazywa się codziennym staniem, o którym będziemy rozmawiać w dalszych częściach.

7. Oprogramowanie robocze jest podstawową miarą postępu.

Głównym celem Agile jest dostarczanie klientom wartości przyrostowych i regularnie. W programie Agile sukces projektu mierzy się ilością dostarczonego oprogramowania i jego potrzebami. Postęp nie jest mierzony liniami napisanego kodu, czasem spędzonym, zakończonymi fazami projektu lub wartością wytworzonej dokumentacji. Postęp jest naprawdę mierzony liczbą funkcjonalności, które są dostarczane i działają.

8. Zwinne procesy promują zrównoważony rozwój. Sponsorzy, deweloperzy,

a użytkownicy powinni mieć możliwość utrzymywania stałego tempa w nieskończoność. Agile wierzy w prowadzenie maratonu, aniżeli krótkie sprinty. Zespół Agile stara się utrzymać tempo realizacji projektu do momentu dostarczenia końcowego działającego oprogramowania. Zwinny zespół pracuje w zrównoważonym tempie, aby zminimalizować takie rzeczy, jak wypalenie, demotywacja, nieoczekiwane opuszczenie ludzi, nierealistyczny harmonogram lub ludzkie stresy. Zdrowe ,zrównoważone tempo minimalizuje ryzyko "technicznego długu" w produkcie. Dług techniczny jest "pojęciem w programowaniu, które odzwierciedla dodatkowe prace rozwojowe, które powstają, gdy kod, który jest łatwy do wdrożenia w krótkim okresie, jest używany zamiast stosowania najlepszego ogólnego rozwiązania." (Źródło: www.techopedia.com). Zrównoważony rozwój zapewnia, że najlepiej unikać długu technicznego, a jakość produktu roboczego jest regularnie dostarczana.

9. Ciągła dbałość o doskonałość techniczną i dobry design zwiększa zwinność. Zwinny zespół skupia się na doskonałości technicznej i dobrym wzornictwie od samego początku. Nie ma cięcia narożnego lub szybkie poprawki. Jeśli w kodzie wykryto jakiś poważny lub drobny problem lub produkt, zespół od razu się do nich zwraca. Jutro nie ma odkładających rzeczy. Zespół Agile próbuje napisać czysty kod, zbudować odpowiednie produkty we właściwej kolejności. Wszyscy członkowie zespołu Agile są zobowiązani do przestrzegania kodu najwyższej jakości, tak aby dług techniczny został zminimalizowany. Uważają, że wysokiej jakości kod poprawia zwinność i oszczędza czas w przyszłości, unikając dodatkowych wysiłków w naprawianiu jakichkolwiek nietypowych prac.

10. Prostota - sztuka maksymalizacji ilości pracy, która nie została wykonana - jest niezbędna. Zwinny zespół nie wierzy w podejście big-bang; utrzymują rzeczy proste. Członkowie zespołu ćwiczą prostotę, koncentrując się na istniejących wymaganiach i koncentrując się na wartości dostarczania, która jest odpowiednia i spełnia potrzeby klienta. Nie myślą zbyt wiele o przyszłości, ponieważ wiedzą, że zmiany są nieuniknione. Koncentrują się na dostarczaniu najwyższej jakości pracy w możliwie najbardziej efektywny sposób.

11. Najlepsze architektury, wymagania i projekty powstają w wyniku samoorganizacji zespołu. Zespół Agile to samoorganizujący się zespół z odpowiednim środowiskiem dostaw i kulturą organizacyjną. Członkowie zespołu to osoby dojrzałe i wiedzą najlepiej, jak dzielić się obowiązkami i stworzyć zdrowe środowisko pracy dla wysokiej jakości pracy. Pracują wspólnie nad wszystkimi aspektami projektów i wyznaczają najlepszą drogę do osiągnięcia pożądanego celu (tzn. Działającego oprogramowania).

12. W regularnych odstępach czasu zespół zastanawia się, jak stać się bardziej skutecznym

dostosowuje i odpowiednio dostosowuje swoje zachowanie. Zespół Agile poświęca wystarczająco dużo czasu na retrospekcję. W regularnych odstępach czasu oceniają, że elementy ich podejścia zadziałały, jakie elementy środowiska zmieniły się i co muszą poprawić, aby pozostać sprawnymi. Innymi słowy, takie podejście nazywa się ciągłym doskonaleniem.

HISTORIA AGILE

1948

Taiichi Ohno, Shigeo Shingo i Eiji Toyoda tworzą "Drogę Toyoty". Wiele koncepcji zwinnych odnosi się do myślenia Lean

1976

Publikacja "Software Reliability" autorstwa Glenford Myers, która określa "aksjomat", że "programista nie powinien nigdy testować swojego własnego kodu" (Dark Age of Developer Testing).

1977

Stworzenie narzędzia "make" dla systemów uniksowych - zasada automatyzacji kompilacji oprogramowania nie jest nowym pomysłem.

1980

Szczegółowe omówienie stopniowego rozwoju w Federalnym Systemie Systemów IBM można znaleźć w tomie wydanym przez Harlan Mills, "Principles of engineering engineering", a konkretnie w artykule autorstwa Dyera, który zaleca organizowanie "każdego kroku w celu zmaksymalizowania rozdziału jego funkcji (funkcji)" od funkcji w innych przyrostach "; Pomysł ten jest jednak w dużej mierze oparty na planowym, etapowym podejściu, a nie na reagującym na zmiany.

1980

Pojęcie "kontroli wizualnej", pochodzące z Toyota Production System, to antycypacja "grzejników informacyjnych".

1984

Wczesne badanie empiryczne Barry'ego Boehma dotyczące projektów wykorzystujących prototypowanie, w istocie z iteracyjną strategią, sugeruje, że iteracyjne podejścia zaczęły być poważnie traktowane w tym czasie, najprawdopodobniej spowodowane takimi czynnikami, jak wzrost liczby komputerów osobistych i graficznych interfejsów użytkownika.

1984

Pojęcie "faktoringu", przewidywanie refaktoryzacji, opisane jest w "Myśleniu do przyszłości" Brodiego, gdzie jest ono przedstawiane jako "organizowanie kodu w użyteczne fragmenty", które "występuje podczas szczegółowego projektowania i wdrażania".

1984

Chociaż krytyka podejścia sekwencyjnego "wodospad" zaczęła się znacznie wcześniej, coraz częściej pojawiają się sformułowania alternatywnych podejść inkrementalnych; dobrym przykładem jest wczesny artykuł na temat "Procesy komunikacji oparte na wiedzy w inżynierii oprogramowania" opowiadający się za stopniowym rozwojem z konkretnego powodu, że "kompletne i stabilne specyfikacje nie są dostępne".

1985

Być może pierwszą, wyraźnie nazwaną, alternatywną alternatywą dla podejścia "wodospadowego" jest Ewolucyjny model dostawy Toma Gilba, nazywany "Evo".

1986

W dobrze znanym artykule Barry Boehm przedstawia "Spiralny model rozwoju i ulepszania oprogramowania", iteracyjny model, którego celem jest identyfikacja i zmniejszenie ryzyka za pomocą wszelkich odpowiednich podejść (chociaż "typowy" przykład jest oparty na prototypowaniu).

1986

Termin "Scrum" pojawia się w artykule Takeuchi i Nonaka "The New Product Development Game", często cytowanym jako inspiracja dla Scruma; pojawia się jednak tylko raz (w tytule sekcji: "Przenoszenie scrum downfield") i nie odnosi się w tym momencie do żadnego spotkania.

1988

"Pole czasowe" jest opisane jako kamień węgielny podejścia "Rapid Iterative Production Prototyping" Scotta Schultza, stosowanego przez firmę Du Pont spin-off, Information Engineering Associates.

1989

Ward Cunningham opisuje technikę CRC we wspólnym artykule z Kentem Beckiem; Specyficzny format używanych kart pochodzi z aplikacji zaprojektowanej przez Cunningham do przechowywania dokumentacji projektowej jako stosu Hypercard.

1990

Bill Opdyke określa termin "refaktoryzacja" w dokumencie ACM SIGPLAN z udziałem Ralpha Johnsona "Refaktoryzacja: pomoc w projektowaniu ram aplikacji i ewoluujących obiektowych systemów"

1990

Rebecca Wirfs-Brock opisuje koncepcyjne aspekty CRC, które wymyśliła, podczas gdy ona i Cunningham pracowali w Tektronix, w swojej książce "Projektowanie obiektowego oprogramowania".

1990

Testowanie dyscypliny zdominowanej przez techniki "czarnej skrzynki", w formie narzędzi do testowania "przechwytywania i powtarzania"

Lata 90

Ze względu na wzrost popularności narzędzi RAD i IDE, narzędzia typu "make" zyskują mieszaną reputację

1991

RAD, prawdopodobnie pierwsze podejście, w którym time-boxing i "iteracje" w luźniejszym poczuciu "jednego powtórzenia całego procesu tworzenia oprogramowania" są ściśle ze sobą połączone, opisał James Martin w swoim "Rapid Application Development". Ta książka opisuje również szczegóły dotyczące czasu w jednym z jego rozdziałów.

1992

Kompleksowy opis "refaktoryzacji" został przedstawiony w pracy Opdyke pt. "Refaktoryzacja obiektowych ram"

1993

Jeff Sutherland wymyśla Scrum jako proces w Easel Corporation.

1995

Najwcześniejsze pisma na Scrumie wprowadzają pojęcie "sprintu" jako iteracji, chociaż jego czas trwania jest zmienny.

1996

Steve McConnell opisuje technikę "Daily Build and Smoke Test"

1998-2002

"Test First" jest rozwinięty w "Test Driven", w szczególności na Wiki C2.com.

1998

Ciągła integracja i "codzienna stand-up" są wymienione wśród rdzenia praktyki Extreme Programming.

1999

Praktyka "refaktoryzacji", wprowadzona kilka lat wcześniej do Extreme Programming, została spopularyzowana przez książkę Martina Fowlera o tej samej nazwie.

1999

Termin "Big Visible Chart" został wymyślony przez Kenta Becka w "Extreme Programming Explained"

2000

"Trzy pytania" codziennego formatu spotkań Scruma są w dużej mierze przyjęte przez zespoły Extreme Programming.

2000

Wykres pożaru został po raz pierwszy opisany przez Kena Schwabera

2000 Termin "prędkość" jest stosunkowo późnym dodatkiem do Extreme Programming

2001

Wprowadzono następujące koncepcje:

• Strukturalne podobieństwa między Agile i pomysłami znanymi jako Lean lub "Toyota Production System"

• Dwa różne smaki estymacji w użyciu w zespołach Agile, szacunki względne i bezwzględne.

• Niektóre techniki techniki eksploracyjnej

• Szybka sesja projektowa

• przyczyna roli-cechy

• Retrospektywa projektu

• warsztaty refleksyjne

• Grzejnik informacyjny

2002

Wspólnota Scrum wybiera praktykę pomiaru "prędkości"

2003

Materiały szkoleniowe Early Scrum wskazują na przyszłe znaczenie "Definition of Done", początkowo tylko w formie tytułu slajdu: "The Stories"

2003

Publikacja "Test Driven Development: By Example" autorstwa Kent Beck

2003

Pojęcie "projektowanie oparte na domenie" zostało ukute przez Erica Evansa

2 MYŚLENIE AGILE

Zanim przejdziemy do szczegółów Agile, niezwykle ważne jest, aby wcielić się w mentora zwinnego praktyka i zrozumieć myślenie stojące za Agile. Każda próba życiowa wiąże się z mechaniką tego działania oraz z nastrójem (lub procesem myślowym) z nim związanym. Podobnie w podróży Agile najważniejsze jest zrozumienie, czym jest Agile i jaki powinien być mentalny specjalista od Agile. Pierwszą i najważniejszą rzeczą w Agile Thinking jest to, że Agile to nie tylko robienie czegoś. To nie jest pakowane rozwiązanie, które kupujemy, aby

coś rozwiązać. Rzeczywiście chodzi o "bycie" Agile we wszystkich aspektach tworzenia i dostarczania oprogramowania. Nie chodzi tylko o zrozumienie "jak" robimy Zwinność; chodzi raczej o "dlaczego" robimy Agile. Aspekt "jak" Agile osiąga się dzięki licznym narzędziom i procesom, praktykom, zasadom i wartościom, które zapewnia struktura Agile. Oto niektóre aspekty myślenia Agile:

• Zwinni praktykujący uważają, że świat wiedzy jest dynamiczny i zmienny. Rzeczy nie są prawie stałe. Zmiana stanie się bez względu na wszystko. Stąd praktykujący Agile spodziewają się nieoczekiwanych zmian i jest przygotowany na ich obsługę. Jest wygodny w pracy w warunkach niepewności i niestabilności.

• Rzadko zdarza się, aby środowiska IT były proste i przewidywalne. Ponieważ IT jest ściśle powiązane z biznesem, ponieważ jest bliskie, każda zmiana środowiska biznesowego również będzie miała wpływ na IT. Środowiska IT zazwyczaj są złożone; istnieje niepewność w zakresie zmienności technologicznej i wymagań.

• IT nie jest jak inżynieria lądowa, gdzie istnieje ustalony określony wynik i określony proces. Na przykład budowanie mostu nie powoduje większych różnic. Po rozpoczęciu budowy zgodnie z wymaganiami i projektem jest bardzo mała zmienność w projektowaniu lub technikach budowlanych. Przed budową uruchomione, wymagania, analiza, projektowanie, budowa i dostawa są planowane i sfinalizowane. To jest przykład modelu Wodospadu, który omówimy później. Przeciwnie, projekty IT prawie nie podążają tą sztywną ścieżką.

• Najlepszym sposobem myślenia podczas Agile jest zastosowanie podejścia empirycznego. Dowiedz się i zastosuj uczenie się, jak idziesz dalej. Budujesz małe inkrementy, dostajesz recenzję i testowane przez właścicieli produktów, uzyskiwać opinie, stosować opinie i wydawać produkty do użytkownika końcowego. Po zakończeniu tego pierwszego cyklu rozpocznij od następnej iteracji i wykorzystaj to, co już dostarczyłeś. Ważne jest, aby eksperymentować, obserwować i zrozumieć wyniki, a następnie dostosować procesy, aby osiągnąć najlepsze wyniki. Dlatego empiryczny sposób myślenia jest ważną koncepcją zwinnego myślenia.

• Podejście empiryczne, które opiera się na doświadczeniu, składa się z trzech głównych filarów - przejrzystość, inspekcja i adaptacja. Przejrzystość to oznacza proces opracowywania produktu musi być łatwo widoczny dla wszystkich zaangażowanych stron w projekcie. Inspekcja wymaga, aby tworzony produkt był poddawany inspekcji regularne odstępy czasu, tak aby można było wykryć i poprawić każdą nieoczekiwaną wariancję. Adaptacja wymaga dostosowania procesu, aby zminimalizować nieoczekiwane i niedopuszczalne wariancje.

• Zwinne myślenie czerpie wiele ze swoich koncepcji z Toyota Production System (TPS). Według Wikipedii Toyota Motor Corporation opublikowała "oficjalny" opis TPS po raz pierwszy w 1992 r .; ta broszura została zmieniona w 1998 roku. W przedmowie, to mówi się: "TPS jest ramą dla oszczędzania zasobów poprzez eliminację odpadów. Osoby uczestniczące w systemie uczą się identyfikować wydatki materialne, wysiłek i czas, które nie generują wartości dla klientów, a ponadto unikamy podejścia "jak to zrobić". Broszura nie jest instrukcją. Jest to raczej przegląd koncepcji, które leżą u podstaw naszego systemu produkcyjnego. Jest to przypomnienie, że trwałe korzyści w zakresie produktywności i jakości

są możliwe zawsze i wszędzie, gdzie zarząd i pracownicy są zjednoczeni w zobowiązaniu do pozytywnych zmian ". TPS opiera się na dwóch głównych filarach koncepcyjnych:

* Just-in-time: "Tworzenie tylko tego, co jest potrzebne, tylko wtedy, gdy jest potrzebne, i tylko w takiej ilości, jaka jest potrzebna"

* Jidoka: - (Automominencja) oznaczająca "Automatyzację z ludzkim dotknięciem"

• Agile również stosuje się do koncepcji TPS dokładnie na czas, zwalniając oprogramowanie w małe przyrosty w oparciu o określone potrzeby / wymagania; Zwinny kładzie większy nacisk na temat komunikacji międzyludzkiej, a mniej na dokumentacji.

• Agile thinking podąża za cyklem rozwoju produktu. Cykl życia produktu rozpiętości od uruchomienia i likwidacji. Podróż między tymi dwoma celami odbywa się poprzez małe fazy dostawy w oparciu o specyficzny zestaw wymagań (lub zaległości w artykułach, które są nazywane światem rozwoju produktu). Zazwyczaj funkcje lub funkcje są tworzone jako karty historii dodawane jako zaległe produkty. Backlog istnieje przez cały cykl życia produktu. Historie są traktowane priorytetowo przez właścicieli produktów na podstawie tego, jak gruboziarniste lub drobnoziarniste są historie i biorąc pod uwagę aspekty czasu, kosztów i jakości.

• Agile postępuje zgodnie z koncepcją "time-boxing". Według Petera Measeya i książki Radtaca - Agile Foundations, "Time-boxing dzieli dostarczanie przyrostów produktu w krótkie, łatwe do kontrolowania okresy czasu (zwane sprintami lub iteracjami) dni lub tygodni i różni się, w zależności od priorytetu, funkcjonalnością, która ma być dostarczona w tych przedziałach czasowych. "

• Wreszcie, kluczowym aspektem myślenia Agile jest to, że przywiązuje on większą wagę do czasu, kosztów i jakości dostarczania oprogramowania niż funkcji / funkcjonalności. Agile traktuje czas, koszty i jakość jako elementy stałe, a funkcje / funkcjonalności jako elementy dynamiczne. Ten zmienny aspekt cech jest przedstawiony w procesie Agile, w którym funkcje są zwalniane w przyrostach. Z drugiej strony tradycyjny model wodospadu wierzy, że elementy są stałym elementem, a czas, koszt i jakość są zmiennymi elementami. Ta restrukturyzacja czasu, kosztów, jakości i funkcji działa lepiej w projektach zwinnych, ponieważ firmy oczekują dostarczenia oprogramowania na czas, w granicach kosztów i odpowiedniej jakości. Nie przejmują się tym, czy wszystkie funkcje są zaimplementowane, czy nie, jeśli istnieje zaległości w ich wymaganiach i mają przejrzystość, którą zespół programistów wyda im w kolejnych iteracjach. Podsumowując, Agile to podróż, a nie miejsce docelowe. Najlepsze wyniki można uzyskać dzięki zwinnej praktyce, gdy zespół i organizacja osiągają właściwy, zwinny sposób myślenia.

PODEJŚCIE WATERFALL

Zanim przejdziemy do podejścia Agile, ważne jest, aby zrozumieć, co doprowadziło do Agiley. We wczesnych latach branży IT rozwój oprogramowania następował sekwencyjny proces rozwoju. W 1970 r. Dr Winston W. Royce napisał artykuł zatytułowany "Zarządzanie rozwojem dużych systemów oprogramowania”, który opisał pierwszy formalny opis modelu wodospadu. Zauważ, że Royce nie użył terminu Waterfall w swoim artykule. Raczej

przedstawił swój model jako przykład wadliwego, niedziałającego modelu. W 1985 r. Departament Obrony Stanów Zjednoczonych uchwycił to podejście w DOD-STD-2167A, ich standardach dotyczących współpracy z wykonawcami oprogramowania, który stwierdził, że "wykonawca wdroży cykl tworzenia oprogramowania, który obejmuje następujące sześć faz: wstępny projekt, Szczegółowe projektowanie, kodowanie i testowanie jednostek, integracja i testowanie ". (źródło: Wikipedia - model wodospadu).

Jak stwierdzono powyżej, model wodospadu zawiera następujące kolejne fazy:

1. Wymagania: przechwytywanie wymagań dotyczących produktu / oprogramowania

2. Analiza: przygotowywanie modeli, schematów i reguł biznesowych

3. Projektowanie: przygotowanie architektury oprogramowania

4. Kodowanie: opracowanie / programowanie i testowanie jednostkowe oprogramowania

5. Testowanie: przeprowadzanie systematycznych testów ogólnego rozwiązania w celu wychwycenia błędów i usterek

6. Operacje: obejmuje instalację, migrację, obsługę i konserwację zakończonego oprogramowania

Powyższe fazy są uważane za sekwencyjne, ponieważ oczekuje się, że należy ukończyć fazę przed przejściem do następnej. Model wodospadu pochodzi z przemysłu wytwórczego i budowlanego, które są wysoko strukturyzowanymi środowiskami fizycznymi. Z powodu braku formalnych metodologii tworzenia oprogramowania w tym czasie, sprzętowy model Waterfall został przystosowany do tworzenia oprogramowania. Kluczowymi argumentami wspierającymi podejście Waterfall są:

• Jeśli więcej wysiłku poświęcimy na wczesnym etapie analizy i projektowania faz, aby lepiej zrozumieć wymagania, lepsza byłaby jakość wdrażanego rozwiązania / produktu.

• Wyższa jakość oznacza mniej błędów na późniejszych etapach, co zmniejsza koszty.

• Z punktu widzenia harmonogramu, użytkownicy wodospadów zwykle spędzają 30-40% czasu w pierwszych dwóch fazach (wymagania i analizy), 30-40% czasu na kodowanie i pozostają na testach i operacjach.

• Kluczowym sposobem komunikacji jest dokumentacja. Chodzi o to, czy którykolwiek członek zespołu opuszcza projekt, żadna wiedza nie jest tracona, a następny członek zespołu może szybko przejść przez całą dokumentację.

• Wodospad nadaje się do środowisk, w których wymagania i zakres są stałe, wariancja technologiczna jest mniejsza, a oczekiwana zmiana jest minimalna.

• Wodospad wspiera przewidywalność zazwyczaj poprzez szczegółowe fazy dostarczania podpisane jako kamienie milowe.

• Wodospad nie uwzględnia nieodłącznej niepewności i zmienności w przemyśle opartym na wiedzy, takim jak przemysł oprogramowania.

• Podsumowując, model Wodospadu kurtyna zwinność opierając się zmianom.

Dlatego zwolennicy rozwoju oprogramowania Agile twierdzą, że model kaskadowy jest nieskuteczny, jeśli chodzi o tworzenie oprogramowania z powodu dużej zmienności.

3 KORZYŚCI Z AGILE

Kluczowe korzyści podejścia Agile to:

• Szybka dostawa oprogramowania: Agile pomaga w dostarczaniu wczesnej wartości do projektu poprzez regularne i częste publikowanie kodu oprogramowania w małych jednostkach funkcji spełniających potrzeby właściciela produktu.

• Podejście iteracyjne: Uwalnianie kodu oprogramowania w małych jednostkach funkcjonalności daje korzyść z iteracyjnego podejścia.

• Fail fail: Uwalnianie kodu oprogramowania w małych jednostkach funkcjonalności pomaga w otrzymywaniu szybkiej informacji zwrotnej od klienta. Tak więc, jeśli coś nie działa, mogą być naprawione daleko. Oznacza to szybkie zawieszenie i poprawienie kursu.

• Wysoka reakcja na zmiany: Ponieważ szczegółowe planowanie projektu odbywa się w niewielkich fazach lub w krokach, projekt nie podlega znacznym zmianom ze względu na zmianę wymagań biznesowych.

• Lepsze zarządzanie ryzykiem: zarządzanie ryzykiem jest lepsze poprzez skupienie się na wdrażaniu niewielkich jednostek funkcjonalności i utrzymywanie uwagi na natychmiastowej i następnej wersji.

• Większa wartość biznesowa: ponieważ funkcjonalności są definiowane przez firmę, a oprogramowanie jest wydawane w małych porcjach funkcjonalności, klient lub właściciel produktu może szybciej i szybciej uzyskać wysoką wartość.

• Wyższa jakość: Z powodu małej iteracji wysiłek i czas opracowania są lepiej zaplanowane. Prace rozwojowe nie są gromadzone na końcu; nie ma pożyczania czasu od innych obszarów, takich jak testowanie. Zapobiega to wszelkim skrótom prowadzącym do wyższej jakości produktu końcowego.

• Przejrzystość: cały proces Agile jest całkowicie przejrzysty. Właściciel produktu widzi funkcjonalności wdrażane stopniowo w fazach; programiści mogą uzyskać szybką informację zwrotną od właściciela produktu i czerpią satysfakcję z tego faktu że zaakceptowana funkcjonalność jest cenna dla firmy.

• Zbiorowa własność: Agile kładzie nacisk na prawo własności do pracy. Drużyna określa tempo projektu i dostosowuje tempo, które jest zrównoważone dla pełny kurs. W związku z tym zespół Agile jest bardzo skuteczny, ponieważ członkowie zespołu mogą przejąć własność, wykazać się świetnym duchem pracy zespołowej współpracując z innymi i pracować w trybie konsensusu.

• Lepsze zarządzanie stresem: ponieważ Agile opiera się na zrównoważonym tempie i duchu wspólnej pracy, zespół może lepiej zarządzać swoim czasem i stresem w dłuższej

perspektywie. Ponieważ nakład pracy związany jest z niewielkimi jednostkami funkcji, harmonogram projektu i koszty są planowane w realistyczny sposób. Pozwala to uniknąć nadmiaru wysiłku ostatniej minuty lub ostatniej mili. Zespół wie, co zostanie wydane w obecnym wydaniu i następnym wydaniu. Dzięki temu mogą lepiej zarządzać swoim czasem osobistym. To zapobiega wszelkiemu wypaleniu i niepotrzebnemu stresowi.

5 OGÓLNY PROCES AGILE

1. Właściciel produktu (zwykle Klient) dostarcza listę pożądanych cech produktu, które są zestawiane w Backlog Produktu. Na podstawie danych uzyskanych od zespołu, przedmioty z Backlogu Produktu są traktowane priorytetowo i ponownie ustalane na początku każdego sprintu. Funkcje te stanowią wysokie wymagania projektu. Większość pozycji jest wyrażona w formie "historii użytkownika".

2. Zespół, wspólnie, wybiera historie użytkowników z zaległych produktów, które mogą zatwierdzić. Opracowywany jest projekt wysokiego poziomu i tworzony jest backlog sprintu. Zadania są identyfikowane, szacowane i samodzielne.

3. Zespół wykonuje planowanie na różnych poziomach (zwolnienie i / lub powtórzenie / sprint).

4. Zespół dostarczający zostaje wyposażony w zwinne praktyki techniczne, aby móc pomyślnie zrealizować projekt.

5. Stand-upy są wykonywane codziennie (zwykle na początku dnia) przez około 15 minut.

6. Wizualne plansze służą do monitorowania statusu iteracji / sprintu.

7. Rejestr RAID jest tworzony, gdy ryzyko, problemy, założenia i zależności są dodawane i monitorowane.

8. Spotkania "pokaż i powiedz" odbywają się na końcu sprintu, iteracji lub wydania zespołu, aby wykazać zainteresowanym stronom wszystkie historie użytkowników wykonane podczas iteracji / sprintu; od interesariuszy oczekuje się informacji zwrotnych na temat dostarczanego produktu.

9. Retrospektywa Sprintu została przeprowadzona w celu omówienia tego, co poszło nie tak, co nie poszło dobrze i co należy zrobić inaczej w następnym razem.

10. Agile Lead ułatwia i umożliwia proces Agile i trenuje zespół.

6 ROLE WSPÓLNE AGILE

Agile jest niekompletna bez udziału zaangażowanych w nią osób. Ludzie są paliwo, które utrzymuje silnik Agile. Agile ma własne zestawy kluczowych ról które są krytyczne dla dostarczania Agile. Na wysokim poziomie kluczową rolą w procesie zwinnym są:

Klient, Agile Lead, Agile Team i Interesariusze. Podczas gdy niektóre z tych ról oczekują określonych umiejętności, podejście Agile ma nadzieję, że członkowie zespołu mogą odgrywać inne role poza swoimi umiejętnościami. Innymi słowy, członkowie zespołu mogą

być "wyspecjalizowanymi generalistami". Poza tymi czterema rolami mogą istnieć inni uczestnicy realizacji Agile. Omówmy szczegółowo każdą z tych zwinnych ról.

• Klient: w Agile klient odnosi się do osoby, która jest właścicielem produktu, który jest tworzony, kto podejmuje decyzje o tym, jaki produkt zostanie zbudowany i jakie prace zostaną wykonane w jakiej kolejności do zbudowania produktu. Klient rozumie wizje i uzasadnienie biznesowe, dlaczego produkt jest budowany. Klient określa również, jakie historie mają być dostarczone, decyduje o kolejności opowiadań na zaległościach i podpisuje sklepy, gdy są one wykonywane na końcu każdej iteracji / sprintu lub zwolnij. Powszechnie używanym mnemonikem do zapamiętania kluczowych kryteriów dobrego klienta jest DARKA, która oznacza:

(D)esire: Klient powinien mieć żywą ochotę, aby zobaczyć produkt budowany zgodnie z wizją. Powinien być podekscytowany produktem.

(A)uthority: klient powinien mieć uprawnienia do podejmowania decyzji dotyczących produktu i jego dostawy oraz egzekwowania ich. Zespół Agile musi jasno zrozumieć, jaki poziom autorytetu ma klient i do kogo się zwrócić, jeśli potrzebne są decyzje wykraczające poza kompetencje klienta.

(R)esponsibility: Klient jest odpowiedzialny za określenie, co zespół dostarczy iw jakiej kolejności.

(K)nowledge: Klient musi posiadać wiedzę na temat dostarczanego produktu; powinien wiedzieć, gdzie znaleźć informacje w odpowiednim czasie.

(A)vailability: klient musi być dostępny w krytycznym momencie projektu, aby podjąć kluczowe decyzje.

• Drużyna Agile: Drużyna Agile zazwyczaj odpowiada za:

* Decydowanie, jak wykonać pracę. Wymaga to udoskonalania historii z klientem a następnie podzielenie na zadania.

* Definiowanie ram czasowych pracy. Obejmuje to szacowanie nakładu pracy wymaganego do wykonania każdego zadania.

* Podejmowanie decyzji, kto wykonuje pracę, sprawdzanie postępu prac i podejmowanie decyzji, jak najlepiej zorganizować się w celu dostarczenia produktu.

* Dostarczenie produktu.

Specjalistyczni specjaliści: wyspecjalizowany specjalista w zespole zwinnym to ktoś, kto ma specjalistyczne umiejętności (takie jak programowanie) i jest również świadomy tego, co robią inni członkowie zespołu w projekcie. Posiadanie specjalistycznych umiejętności w zespole sprzyja lepszej komunikacji, współpracy i pracy zespołowej. Typowe umiejętności specjalistyczne znalezione w zespole Agile to:

* Znajomość biznesowa

* Umiejętności architektoniczne

* Umiejętności analityczne

* Umiejętności projektowe

* Umiejętności kodowania

* Umiejętności testowania

Samoorganizujące się zespoły: Zespoły zwinne to samoorganizujące się zespoły, ponieważ mają one uprawnienia do decydowania, w jaki sposób wykonuje się pracę, kto to robi i ile czasu powinien zająć. Samoorganizujące się zespoły mogą podejmować inicjatywę, koncentrować się na wkładzie zespołu, koncentrować się na rozwiązaniach, współpracować ze sobą, wprowadzać innowacje w celu wypracowania lepszych metod pracy, znajdować sposoby na dostarczenie lepszego rozwiązania, zarządzać ograniczeniami i podejmować działania w celu zarządzania ewentualnymi sytuacjami kryzysowymi i sytuacjami kryzysowymi. Samoorganizujące się zespoły prowadzą do szybszego podejmowania decyzji, wyższej motywacji, zwiększonej siły mózgowej oraz zwiększonego poziomu inicjatywy i ciągłego doskonalenia. Grupy funkcyjne: Agile zespół funkcji są zazwyczaj zorganizowane jako zespołów funkcji, ponieważ umożliwia dostarczanie funkcji z zaległości. Zespół funkcji Agile koncentruje się na dostarczaniu klientowi wartości dodanej.

Zespoły komponentów: zespoły komponentów wchodzą w grę, gdy zespoły funkcyjne nie mogą dostarczyć funkcji do klienta, ponieważ cykl dostawy może wymagać wcześniejszego dostarczenia jednego lub więcej komponentów. Zespoły komponentów są odpowiedzialne za wytwarzanie komponentów, które można łączyć z innymi komponentami z innych zespołów i integrować w jedną funkcję.

Rdzeń i zespoły wsparcia: Zespół Agile może zostać podzielony na zespoły podstawowe i wspierające. Można to zwykle zaobserwować w dużych i złożonych projektach. Główny zespół koncentruje się na dostarczaniu w całym łańcuchu wartości; zespoły wsparcia zapewniają wsparcie dla podstawowych zespołów, aby robić wszystko dobrze za pierwszym razem. Przykładem obszarów, w których zespół wsparcia umożliwia zespołowi podstawowemu jest wsparcie architektoniczne, wsparcie interfejsu użytkownika lub wsparcie badań.

• Agile Lead: Zwinność leadów jest głównie odpowiedzialna za umożliwienie zespołowi selforganizacji i ciągłego doskonalenia. Ołów Agile rozumie ogólną dostawę cykl, zrozum zespół, ich umiejętności i sposób pracy. Zwinność ołowiu nie dowodzi i nie kontroluje zespołu; on lub ona raczej ułatwia i umożliwia zespół do samoorganizacji. Zwinny ołów usuwa przeszkody, które blokują zadanie wykonywane lub dostarczana funkcja. Moderator warsztatów: Agile Lead czasami stawia czoło Agile-facylitatorowi ułatwiającemu dostarczanie ram i działań Agile poprzez warsztaty. To wymaga koordynacji, planowania i organizacji warsztatów. Ponieważ facylitator musi być niezależny od osób biorących udział w warsztatach, wskazane jest, aby moderator pochodził z innego zespołu.

Funkcja ułatwiająca proces: Agile Lead pomaga również organizacji, programowi, projektowi i zespołowi w zdefiniowaniu procesu Agile i modelu operacyjnego. Agile Lead zapewnia, że procesy Agile są przestrzegane, działania Agile są respektowane, a wielkość sklepów jest

prawidłowa. Agile Lead współpracuje również ściśle z zespołem, aby określić wszelkie możliwości usprawnienia procesu Agile.

Trener lub trener: Agile Lead pełni także funkcję trenera lub trenera do propagowania znajomość zasad, procesów i praktyk Agile w zespole. Jeśli istnieje nowa praktyka lub proces, który musi zostać wdrożony, Agile Lead działa tak, jak pozwala pomagać zespołowi w ich wdrażaniu.

• Interesariusze: Interesariusze to osoby, które mają interes w projekcie, mogą być aktywnie zaangażowane w projekt, mogą wywierać wpływ na projekt, a ich interesy mogą pozytywnie lub negatywnie wpływać na wykonanie lub zakończenie projektu. Dlatego identyfikacja i efektywne zarządzanie interesariuszami jest kluczem do udanej realizacji Agile. Kluczową rolą interesariusza jest zapewnienie, że reprezentowane są interesy grupy, której są częścią. Na przykład, jeśli klient jest interesariuszem, byłby zainteresowany, aby zapewnić, że zespół projektowy dostarcza produkt zgodnie z wymaganiami ustalonymi przez klienta.

7 WSPÓŁCZESNE TECHNIKI PRAWDZIWE OPOWIEŚCI I OPROGRAMOWANIE WSTĘPNE

Historia użytkownika jest definicją bardzo wysokiego poziomu wymagań, zawierającą tylko tyle informacji, aby programiści mogli oszacować rozsądnie wysiłek wdrożenia. Przechwytuje opis funkcji oprogramowania z perspektywy użytkownika końcowego. Historia użytkownika opisuje typ użytkownika, jego potrzeby i przyczyny ich zainteresowania. Kluczowymi cechami karty opowieści są:

Podsumowanie

Jako <WHO>: Kto chce tę funkcję

Ja Chcę <WHAT>: jakiej funkcji chcą

Tak Więc <WHY>: Dlaczego oni chcą tej funkcji

Scenariusz

PODANE <WARUNKI URUCHOMIENIA>

KIEDY <WYSTĘPUJE AKCJA>

WTEDY <OCZEKIWANY WYNIK>

Dobre historie dotyczące jakości

NIEZALEŻNY

DO NEGOCJACJI

ZMIENNA (DO UŻYTKOWNIKA)

GODNY SZACUNKU

MAŁY

TESTOWALNE

Powyższe jest również nazywane 3Cs - Card, Conversation, Confirmation

Przykładem karty opowieści użytkownika jest:

PRZED KARTĄ

Jako klient - chcę mieć możliwość aktualizowania moich danych osobowych (hasła, adresu e-mail, adresu itp.) Na stronie internetowej. Tak więc - moje informacje są aktualne i otrzymuję terminową komunikację.

POWRÓT KARTY

Kryteria przyjęcia:

• Czy mogę zaktualizować swoje konto w dowolnym momencie - 24x7?

• Czy mogę uzyskać dostęp i aktualizować swoje konto w dowolnym trybie - na stronie internetowej, komórkowej itp.?

• Czy otrzymam potwierdzenie zmiany pocztą e-mail lub wiadomością tekstową?

• Czy moje aktualizacje wejdą w życie natychmiast?

Dobrze sformułowane historie będą spełniały kryteria akronimu BILLEST firmy Bill Wake:

Niezależny: Historie powinny być dostarczane niezależnie od siebie, oraz w dowolnej kolejności

Do negocjacji: Historie powinny być elastyczne i zbywalne, aby mogły być skorygowane w czasie

Cenne: wartość historii musi być zrozumiała dla klienta

Godny szacunku: Historie muszą być zrozumiałe przez zespół, aby tworzyć szacunki do planowania

Mały: "Małe jest piękne" powinno być motto. Duże historie są trudne do oszacowania i zaplanować. Dobrą praktyką jest udoskonalanie historii od 1 do 5-dniowego wysiłku

Testowalny: Historie muszą zawierać testowalne kryteria akceptacji do osiągnięcia Status "sporządzono".

Agile Planning Pyramid: Teraz, gdy mamy pojęcie o tym, co powinna zawierać karta historii, pytanie brzmi: jak zdefiniować karty historii. Czy są dostępne najlepsze praktyki? Odpowiedź brzmi tak. Poniżej przedstawiamy niektóre z najlepszych praktyk:

• Historie powinny być utrzymywane niezależnie od siebie

• Historie można określić jako gruboziarniste lub drobnoziarniste w zależności od charakteru projektów. Jeśli realizujesz duży złożony projekt, dobrze jest podzielić historie na pod-opowieści. W tym celu stosuje się piramidę planistyczną.

• Wraz z rozwojem dostawy historie są podzielone z gruboziarnistych na drobnoziarniste historie. Zasadniczo więcej szczegółów jest wprowadzanych, ponieważ priorytet historii rośnie.

• Z perspektywy osi czasu historie są precyzyjnie definiowane bliżej bezpośredniego sprintu iteracyjnego. Jeśli iteracje są daleko, lepiej pozostawić opowieści gruboziarnistą. Jest tak dlatego, że istnieje duże prawdopodobieństwo, że produkt ulegnie zmianie w trakcie cyklu rozwoju. Dlatego może być nieproduktywne definiowanie zbyt drobiazgowych opowieści, jeśli nie zostaną one podjęte w najbliższym wydaniu.

Agile Estimation: Po zdefiniowaniu kart historii, następnym działaniem jest ich oszacowanie. Szacowanie to proces uzgadniania pomiaru wielkości dla historii, a także zadań wymaganych do wdrożenia tych historii w zaległościach dotyczących produktu. Szacowanie to zazwyczaj a prognozę, ile wątków można dostarczyć w iteracji wydania / sprintu. Szacowanie odbywa się głównie w zespole, aby każdy członek zespołu był zaangażowany w planowaną pracę. Domyślnym typem jednostki używanym do oszacowania historii są Punkty. Przykładowy zestaw wartości punktowych to: 2, 3, 5, 8, 13, 20, 40 lub 100 punktów. To jest sekwencja Fibonacciego. Przed planowaniem iteracji oceny punktowe są wprowadzane w polu Planowana prędkość w iteracji. Ta wartość wskazuje całkowitą liczbę punktów fabularnych, które zespół uważa za możliwe do wykonania w iteracji. Różne zespoły będą miały inną prędkość. Innym sposobem obliczenia prędkości jest średnia całkowita liczba zaakceptowanych punktów z poprzednich iteracji. Przez pewien czas zespół będzie miał dość dobre pojęcie o swojej prędkości. Jeśli oszacowanie fabuły obliczone przez zespół jest większe niż całkowita prędkość dla nadchodzącej iteracji, jest prawdopodobne, że zespół nie będzie w stanie osiągnąć celu. W związku z tym wskazane jest rozbijanie dużych opowieści na mniejsze historie o dzieciach.

Jak oszacować historie: Popularną techniką szacowania punktów fabularnych jest warsztat "Planowanie pokera". Tutaj zespół decyduje o względnym wysiłku (punktach fabularnych) wymaganych do przekazania historii do stanu "ukończono". Planowanie pokera zazwyczaj wykorzystuje względną wielkość opartą na drobnych i grubych różnicach między liczbami. Względna różnica między historiami, które są bliskie dostarczenia, będzie drobnoziarnista, podczas gdy względna różnica między historiami, które są dalej od dostawy, będzie gruboziarnista. Typowe punkty fabuły w planowaniu pokera to:

• Historia jest w stanie 'done' = 0

• Drobnoziarniste rozmiary = ½ (xxs), 1 (xs), 2 (s), 3 (sm), 5 (m), 8 (ml), 13 (l)

• Gruboziarniste rozmiary = 20 (xl), 40 (xxl), 100 (xxxl)

Typowy proces planowania pokera wygląda następująco:

1. Utwórz lub kup zestaw kart planowania dla każdego członka zespołu. Masz kartę dla każdej wartości w systemie punktowym: 2, 3, 5, 8, 13, 20, 40, 100 (lub podobne)

2. Wyświetl proponowaną historię na monitorze lub tablicy. Wyjaśnij, jaką wartość zapewni historia i jakie są początkowe wymagania.

3. Daj członkom zespołu minutę lub dwie, aby wybrać kartę, zakryte.

4. Przy liczbie trzech wszyscy członkowie drużyny podnoszą rękę, wyświetlając kartę.

5. Znajdź najniższą i najwyższą wartość karty.

6. Każdy z najniższych i najwyższych posiadaczy kart zajmuje dwie minuty, aby wyjaśnić, dlaczego uważają, że historia ma określoną wielkość.

7. Podaj kilka momentów, w których inni mogą odważyć, jeśli dyskusja odkryje jakiekolwiek problemy lub wymagania.

8. Ponownie zagłosuj, dopóki nie osiągniesz porozumienia.

9. Zapisz wartość szacunkową.

PLANOWANIE AGILE

Planowanie w Agile odbywa się w celu umożliwienia zmiany. Mówiąc prosto, zwinne planowanie polega na mierzeniu szybkości, z jaką zespół może przekształcić historie użytkowników w działające, gotowe do produkcji oprogramowanie, a następnie wykorzystywać te informacje do określenia, kiedy zostaną wykonane. Proces planowania zwykle rozpoczyna się od głównej listy artykułów, która zawiera listę funkcji który klient chce zobaczyć w produkcie końcowym. Szybkość, z jaką historie użytkowników są przekształcane w działające oprogramowanie, nazywa się prędkością zespołu. Velocity określa oczekiwania co do terminów dostaw w przyszłości. Następnie za pomocą zwinnej iteracji planowanie odbywa się od jednego do dwóch tygodni sprintów pracy. Liczba iteracji jest określona przez następujący wzór:

iteracje = całkowity wysiłek (w punktach) / szacowana prędkość zespołu

Przykład:

iteracji = 200 pkt / 10 pkt na iterację = 20 iteracji

Po określeniu prędkości i liczby iteracji możemy określić, czy (a) będziemy szli szybciej niż oczekiwano, czy (b) będziemy wolniej niż pierwotnie przewidywano.

Planowanie z góry na dół:

Planowanie z góry na dół służy do tworzenia szacunków dla dłuższych ram czasowych. Plany odgórne są szybkie i nie zawsze dokładne (nie mają takiego charakteru); są one szczególnie dostosowane do środowisk, które są zmienne i nieco ryzykowne do szczegółowego planowania. Planowanie od góry do dołu można wykonać za pomocą "punktów fabularnych" lub "dni idealnych". Poprzednie doświadczenia lub przeszłe dane są również wykorzystywane w procesie planowania.

Planowanie od dołu: Planowanie od dołu jest używane, gdy wymagane jest bardziej szczegółowe oszacowanie. W tym procesie zespoły mają dobry pomysł, które historie prawdopodobnie zostaną dostarczone w iteracji / sprintu. Na tej podstawie zespół określa zdolność dostarczania (reprezentowaną w "całkowitych dostępnych godzinach w ramach

iteracji / sprintu"), planuje zadania wymagane do uzyskania statusu "gotowe" do wykonania i szacuje godziny potrzebne do wykonania zaplanowanych zadań . Jest to zwykle wyrażane jako "całkowita wymagana liczba godzin" w iteracji / sprincie. "Całkowite wymagane godziny" są następnie porównywane z "całkowitymi dostępnymi godzinami". Jeśli liczby się różnią, zespół może usuwać, wymieniać, dodawać lub dzielić niektóre artykuły, aż do osiągnięcia wymaganych godzin.

Planowanie iteracji / sprintu: planowanie iteracyjne / wiosenne jest zwykle wykonywane w dwóch częściach - "Planowanie część 1" i "Planowanie część 2".

Planowanie iteracji / sprintu część 1: To jest przykład planowania z góry i jest wykonywane w taki sam sposób, jak planowanie wydania - tzn. zespół planuje liczbę wątków, które mają być dostarczone w iteracji / źródle w oparciu o wielkość i szybkość wątku fabularnego.

Planowanie iteracji / sprintu część 2: Jest to przykład planowania od podstaw. Jeśli prognozowana liczba artykułów, które mogą być wyświetlane w dopasowaniu zstępującym i oddolnym, prawdopodobnie będzie to możliwa do oszacowania. Jeśli liczby nie są zgodne, zespół stwierdza przyczyny różnicy i wykonuje wymaganą kalibrację, aby uzyskać solidne oszacowanie.

TESTOWANIE AGILE

Sprawne testy oznaczają ogólnie praktyki testowania oprogramowania pod kątem wad lub problemów z wydajnością w kontekście zwinnego przepływu pracy. Zgodnie z praktyką Agile, cykl testowania musi być również zwinny. Pozostawienie testowania pod koniec wydania jest szkodliwe, ponieważ wykryte w tym miejscu poważne wady wykolejają jakość i dostawę produktu. Podobnie jak wielokrotne iteracje / sprint w cyklu dostarczania Agile, cykl testowania musi być częsty. Poniższy rysunek przedstawia prosty do zrozumienia model testowy Agile; zawiera wskazówki dotyczące rodzajów testów zwykle stosowanych w procesie dostarczania Agile, celu testowania i sposobu jego dostarczenia. Wiele praktyk programistycznych opartych na testowaniu jest używanych w zwinnych dostawach. Ich krótki opis znajduje się poniżej:

TFD (TEST PIERWSZY ROZWÓJ)

Jak sama nazwa wskazuje, praktyka TFD oznacza, że przypadki testowe są pisane przed pojawieniem się jakichkolwiek zmian, aby spełnić kryteria akceptacji fabuły. Tworzenie testów w pierwszej kolejności, przed kodem, znacznie ułatwia i przyspiesza tworzenie kodu. Utworzenie testu jednostkowego pomaga programistom zastanowić się, co należy zrobić. Wymagania są dostrajane przez testy. TFD eliminuje wszelkie nieporozumienia specyfikacji zapisanej w postaci kodu wykonywalnego. TFD również korzysta z projektowania systemu; tworząc pięść testową na projekt wpłynie chęć przetestowania wszystkiego, co ma wartość dla klienta. Po zapisaniu testów zaimplementowany zostaje cykl test-kompilacja, dopóki wszystkie błędy związane z kompilacją nie zostaną naprawione, a klient jest gotowy do podpisania historii. TFD potwierdza, że dostawa / produkt spełnia kryteria akceptacji fabuły uzgodnione przez klienta.

TDD (TEST DRIVEN DEVELOPMENT)

Rozwój oparty na testach (TDD) to proces tworzenia oprogramowania, który opiera się na powtórzenie bardzo krótkiego cyklu rozwojowego: wymagania zostają przekształcone w bardzo specyficzne przypadki testowe, a następnie udoskonalono oprogramowanie, aby tylko przejść do nowych testów. TDD jest zwykle implementowany na poziomie testowania jednostki lub komponentu. Obejmuje na praktyce "refaktoryzacji" Agile. Oznacza to, że testy i kod produkcyjny są pisane w celu kierowania projektem w trakcie naszej podróży. Zasadniczo, chcemy usprawnić i wyjaśnić. Założeniem TDD jest to, że trudno jest określić, który projekt będzie działał najlepiej, dopóki nie zostanie napisany kod. Gdy dowiadujemy się o tym, co działa, a co nie, jesteśmy w najlepszej możliwej sytuacji, aby zastosować te spostrzeżenia, gdy są one wciąż świeże w naszym umyśle. A wszystkie te działania są chronione przez zestawy zautomatyzowanych testów jednostkowych. W TDD zespół koncentruje się na warunkach w teście, które mogą spowodować awarię kodu. Kiedy nie ma już żadnych warunków awarii, rozwój uznaje się za zakończony. W TDD skupiono się na projektowaniu i zapewnieniu, że produkty są zaprojektowane w odpowiedni sposób.

ATDD (BADANIE PRZYJĘCIA NAPĘDU ROZWOJU)

Analogiczny do ATDD, Driven Development Driven Development (ATDD) jest bliżej użytkownika testy akceptacyjne (UAT). Test oparty na akceptacji Driven Development (ATDD) jest praktyką w którym cały zespół wspólnie omawia kryteria akceptacji, z przykładami, a następnie destyluje je w zestaw konkretnych testów akceptacyjnych przed rozpoczęciem rozwoju. Wspólne dyskusje, które pojawiają się w celu wygenerowania testu akceptacyjnego, są często nazywane trzema amigami, reprezentującymi trzy perspektywy klienta (jaki problem próbujemy rozwiązać?), Rozwojem (jak możemy rozwiązać ten problem?) I testowaniem (co powiesz na…). Te testy akceptacyjne reprezentują punkt widzenia użytkownika i działają jako forma wymagań, aby opisać, jak system będzie działać, a także służyć jako sposób sprawdzenia, czy system działa zgodnie z przeznaczeniem.

BDD (ZACHOWANIE ZWIĄZANE Z ZACHOWANIEM ZACHOWANIA)

Behavior Driven Development (BDD) to synteza i udoskonalenie praktyk wynikających z Test Driven Development (TDD) i Driven Test Driven Development (ATDD). BDD skupia się na testowaniu scenariuszy, aby upewnić się, że system zachowuje się w sposób, w jaki oczekuje od niego zachowanie. BDD rozszerza TDD i ATDD o następujące taktyki:

• Zastosuj zasadę "Five Why" do każdej proponowanej historii użytkownika, aby jej cel był wyraźnie powiązany z wynikami biznesowymi

• Myślenie "z zewnątrz", innymi słowy wdrażaj tylko te zachowania, które najbardziej bezpośrednio przyczyniają się do tych rezultatów biznesowych, aby zminimalizować marnotrawstwo

• Opisz zachowania w jednej notacji, która jest bezpośrednio dostępna dla ekspertów domeny, testerów i programistów, aby poprawić komunikację

• Zastosuj te techniki aż do najniższego poziomu abstrakcji oprogramowania, zwracając uwagę na rozkład zachowań, aby ewolucja pozostała tania

SPECYFIKACJA NA PRZYKŁADZIE

Specyfikacja według przykładów zapewnia, że to, co jest tworzone, odpowiada wymaganiom klienta, a testowanie koncentruje się na częściach systemu, które generują największą wartość biznesową. Gojko Adzic ukuł termin Specyfikacja przez Przykład i jest to tytuł jego książki (Adzic, 2011). Wybrał tę nazwę na nazwy, takie jak Behavior Driven Development (BDD), ponieważ wyrażenie kładzie nacisk na charakter wymagań, "przykłady", a nie na inne rzeczy, takie jak proces programowania lub interfejs użytkownika. Definiuje specyfikację poprzez przykład jako "... zestaw wzorców procesów, które ułatwiają zmianę w oprogramowaniu, aby zapewnić, że właściwy produkt jest dostarczany wydajnie. Kiedy mówię o właściwym produkcie, mam na myśli oprogramowanie, które zapewnia wymagany efekt biznesowy lub spełnia cel biznesowy ustalony przez klientów lub użytkowników biznesowych i jest na tyle elastyczne, że może być w stanie uzyskać przyszłe ulepszenia przy stosunkowo płaskich kosztach zmiany. "

Kluczowymi elementami specyfikacji na podstawie przykładów są:

• Wyprowadzanie zakresu systemu z jasno wyartykułowanych celów biznesowych

• Określanie kryteriów akceptacji dla opowieści współpracujących między zespołem, klientem i interesariuszem

• Uzgodnienie wymagań za pomocą przykładów

• Udoskonalanie specyfikacji przez cały okres użytkowania produktu

• Automatyzacja sprawdzania poprawności bez zmiany specyfikacji

• Częste sprawdzanie, czy budowany produkt spełnia specyfikacje

• Rozwój systemu dokumentacji, który składa się z prostych do zrozumienia wymagań i testów, które je potwierdzają

8 WSPÓŁCZESNE PRAKTYKI KRÓTKIE PASZE ZWROTNE

Pętle sprzężenia zwrotnego mają kluczowe znaczenie dla sukcesu Agile. Pętla sprzężenia zwrotnego to proces, w którym wyniki działania procesu mogą wpłynąć na to, jak sam proces będzie działał w przyszłości. Niektóre rodzaje pętli sprzężenia zwrotnego, które istnieją w zwinnych procesach programistycznych, to:

• Rozmowy twarzą w twarz

• Codzienne stroje

• Pokaż i opowiada

• Programowanie w parze

• Testów jednostkowych

• Para negocjacji

• Wydanie

KOMUNIKACJA TWARZĄ W TWARZ

Komunikacja twarzą w twarz umożliwia szybkie sprzężenia zwrotne, poprawia relacje, tworzy wzajemne zaufanie i definiuje centra oddziaływania w organizacji. W pracy zespołowej, takiej jak Agile, interakcja twarzą w twarz jest bardziej korzystnym i produktywnym wyborem. Dlatego zespoły Agile są fizycznie zlokalizowane, ponieważ umożliwiają lepszą komunikację twarzą w twarz.

CODZIENNY STAND-UPS

Codzienne stand-upy, jak sama nazwa wskazuje, to codzienne spotkania, podczas których cały zespół spotyka się codziennie, aby uzyskać szybką aktualizację statusu. Umożliwia to zespołowi sprawdzenie dokonanych postępów, zaplanowanie dnia pracy i wyeliminowanie wszelkich wąskich gardeł w pracy. Codzienne spotkania na stojąco są ograniczone do maksymalnie 15 minut. Spotkanie w większości przypadków odbywa się o pierwszej godzinie roboczej rano. Pomaga to ustalić porządek dzienny na resztę dnia. Zwykle te spotkania odbywają się w prostym formacie, a każdy członek odpowiada na każde z następujących trzech pytań:

• Co zrobiłem wczoraj, co pomogło zespołowi programistów w osiągnięciu celu sprint / iteracja?

• Co mam zrobić dzisiaj, aby pomóc zespołowi programistów w osiągnięciu celu sprint / iteracja?

• Czy widzę jakąś przeszkodę, która uniemożliwia mi lub zespołowi programistów osiągnięcie celu sprint / iteracji?

Spotkanie zazwyczaj koncentruje się wokół wizualnej planszy, na której układane są zadania i / lub opowieści. Zespół zaczynał od najdalej wysuniętego punktu na planszy i mówił o zadaniach, które są najbliższe ukończeniu. Potem przeciskają się po planszy. Pomaga to skoncentrować się na zadaniach, które są bliskie ukończenia.

POKAŻ I MÓW

"Show and tells" to spotkania, które odbywają się na końcu sprintu / iteracji / wydania. Celem tych spotkań jest zaproszenie zespołu do udziału w tych spotkaniach i zademonstrowanie wszystkich historii ukończonych podczas iteracji / sprintu. Zespół poszukuje natychmiastowej informacji zwrotnej od zainteresowanych stron. Daje także zespołowi możliwość pokazania najważniejszej pracy, jaką wykonali, porozmawiania o tym, czego się nauczyli, wyjaśnienia swoich planów na najbliższe tygodnie i odpowiedzi na pytania. Daje także innym zespołom szansę sprawdzenia, w jaki sposób ich praca odnosi się do innych. Show and tells może być również demonstracją na żywo nowego działającego produktu lub funkcji; zainteresowane strony mają szansę zobaczyć, dotknąć i poczuć prawdziwy działający produkt. Pokazywanie i opowiadanie spotkań jest dobrą okazją do podkreślenia ryzyka, problemów, przeszkód na drodze i założeń dla zainteresowanych stron, tak aby mogli oni przekazywać informacje zwrotne lub spostrzeżenia, które mogą pomóc w ich rozwiązaniu. Show and tells powinno być otwartą i przejrzystą dyskusją. Oferuje

uczestnikom szansę na zrozumienie i sami widzą postęp, jaki dokonał się na produkcie. Spotkania dają także zespołowi szansę na zamknięcie swojej pracy na czas sprintu. Daje zespołowi okazję do świętowania sukcesu w realizacji swoich celów i jest to szansa dla obecnych członków kierownictwa wyższego szczebla na rozpoznanie i potwierdzenie wysiłków zespołu. Pokaż i mów spotkania NIE są:

• Prezentacja PowerPoint

• Spotkanie przeglądu dokumentów

• Wykrywanie usterek i spotkanie z palcem

RETROSPEKTYWY

Retrospektywa Agile to spotkanie, które odbywa się pod koniec iteracji w Agile Software Development (ASD). Podczas retrospekcji zespół zastanawia się nad tym, co stało się w iteracji, i określa działania, które należy podjąć w przyszłości. Celem retrospektyw jest ciągłe doskonalenie produktu w miarę jego dostarczania i ciągłego doskonalenia sposobu dostarczania produktów. Podczas retrospekcji każdy członek zespołu odpowie na następujące pytania:

• Co działało dobrze dla nas? (Co powinniśmy dalej robić?)

• Co nie działało dobrze dla nas? (Co powinniśmy przestać robić?

• Jakie działania możemy podjąć, aby usprawnić nasz proces w przyszłości? (Co powinniśmy zacząć?)

Kluczowe kroki w procesie retrospektywnym to:

• Ustawianie sceny tak, aby każdy mógł mówić

• Zbieranie danych

• Generowanie wglądów

• Decydowanie, co robić

• Zamknięcie retrospekcji

DOKUMENTACJA ZWYCZAJNA

Manifest dla Agile Software Development ceni "działające oprogramowanie nad kompleksową dokumentacją". Dokumentacja powinna być zminimalizowana i może być wytwarzana tylko wtedy, gdy zwiększa wartość i powinna nadawać się do celu. Kluczem jest znalezienie właściwej równowagi między dokumentacją a dyskusją. Można zastosować trzy kryteria decydujące o tym, ile dokumentacji należy zapisać:

• Essential: Dokumentuj to, co jest niezwykle istotne

• Wartościowe: Dokument tylko jeśli zapewnia wartość

• Terminowo: Dokumentuj tylko wtedy, gdy jest potrzebny

Dlaczego nazywa się to emergentną dokumentacją? Jest tak dlatego, że kieruje się mottem "dokumentujemy, gdy odkrywamy". Zbyt duża dokumentacja to strata czasu. Zespół powinien udokumentować tylko to, co ZROBIŁ, w przeciwieństwie do tego, co według nich będą robić dalej. Dokumentacja powinna być częścią całego procesu, a nie osobnym działaniem. Powinien to być wspólny wysiłek.

TABLICE WIZUALNE

Wizualne tablice w Agile przybierają różne formy. Najważniejsze z nich wyjaśniono poniżej.

• Grzejnik informacyjny

Grzejnik informacyjny, znany również pod nazwą Big Visible Chart (BVC), to duża graficzna reprezentacja informacji o projekcie przechowywanych w widocznym miejscu w udostępnionej przestrzeni roboczej zwinnego zespołu programistycznego. Powinien przekazywać informacje każdemu, kto je widzi. Jest to świetny sposób na przekazanie informacji o aktualnym stanie dostawy. Grzejnikiem informacji może być odręczny, narysowany, wydrukowany lub elektroniczny wyświetlacz. Mogło być stosowane fizycznie, jak montaż na ścianie. Grzejniki informacyjne są kluczowym źródłem informacji podczas codziennych spotkań stand-up, show and tells i retrospektyw. Podczas codziennych spotkań stand-up członkowie zespołu mogą rozmawiać o codziennym postępie opowieści i zadań reprezentowanych na tablicy. Aby pokazać i powiedzieć, grzejniki informacyjne dostarczają informacji o zrealizowanych zadaniach i historiach. W przypadku retrospekcji, tablica może pokazywać informacje na temat tego, co wymaga uwagi. Co najmniej grzejnik informacyjny powinien pokazywać aktualny stan zadań / historii, nad którymi pracuje zespół oraz jak duży postęp robią, aby uzyskać status "zrobione". Powszechnie używane promienniki używane w środowiskach Agile to:

* Wizja produktu

* Plan zaległości / zwolnienia produktu

* Backlog iteracji

* Wykresy wypalania i wypalania

* Lista utrudnień

• Wykresy wypalenia i wypalenia

Wykres wypalania to wykres, który porównuje wysiłek wydany do tej pory z oczekiwanym wysiłkiem. Można go wykorzystać do przewidywania, czy zespołowi uda się dostarczyć wszystko w ramach iteracji / sprintu przez ekstrapolację linii "faktyczny wysiłek" w oparciu o najnowsze prognozy. Wykres wypalania to dobry sposób na pokazanie, co się dzieje, a ile postępu robimy podczas każdej iteracji / sprintu. Wykres wypalania śledzi całkowitą liczbę wymaganych prac a rzeczywistą wykonaną pracę. Kiedy dwie linie przecinają się, iteracja / sprint jest uważany za "zrobiony".

• Dziennik RAID

Dziennik RAID jest pojedynczym repozytorium wszystkich kluczowych informacji o dostarczeniu, który nie jest wyrażone w zaległościach lub w innej dokumentacji. RAID oznacza

* Risks (Ryzyka)

* Assumptions (Założenia)

* Issues/Actions (Problemy / działania)

* Dependencies (Zależności)

ZRÓWNOWAŻONE TEMPO

Zrównoważone tempo jest istotną częścią zarówno Extreme Programming, jak i Agile Manifesto. Oznacza to, że zespół dąży do zapewnienia tempa pracy, które będzie trwałe w dłuższej perspektywie. Ma to na celu poprawę wydajności poprzez wyeliminowanie wypalenia zawodowego. Podstawą zrównoważonego tempa był guru XP Kent Kent w pierwszej edycji "Extreme Programming Explained" (1999), gdzie zaproponował pracę nie więcej niż 40 godzin tygodniowo i nigdy nie pracował w nadgodzinach drugi tydzień z rzędu. Jeden z zasady stojące za Manifestem Agile były poświęcone "Zrównoważonemu Tempo", które można uznać za najszerzej akceptowaną definicję:

"Zwinne procesy promują zrównoważony rozwój. Sponsorzy, deweloperzy i użytkownicy powinni mieć możliwość utrzymywania stałego tempa w nieskończoność. "

Zrównoważone tempo nie polega na tym, aby brać to łatwo i powoli. Polega na mądrym wydatkowaniu energii i odzyskaniu sił poprzez odpowiedni odpoczynek.

FOCUS NA JAKOŚCI

Praktyka Agile koncentruje się na dwóch rodzajach jakości - funkcjonalnym i technicznym.

• Jakość funkcjonalna: Jakość funkcjonalna polega na dostarczaniu funkcji i funkcjonalności zgodnie z oczekiwaniami klientów. Oznacza to zapewnienie, że projekt / produkt działa tak, jak powinien i czego oczekują użytkownicy.

• Jakość techniczna: Jakość techniczna dotyczy dostarczania produktu lub oprogramowania zgodnie z wymaganiami funkcjonalnymi i niefunkcjonalnymi dostarczonymi przez klienta. Produkt powinien być odpowiedni do celu.

REFAKTOROWANIE

Refaktoryzacja oznacza zmianę projektu systemu (struktury wewnętrznej) bez zmiany jego zachowania zewnętrznego. Refaktoryzacja NIE oznacza przepisywania kodu, naprawiania błędów ani ulepszania obserwowalnych aspektów oprogramowania, takich jak jego interfejs. Refaktoryzacja koncentruje się głównie na poprawie jakości projektu systemu; powinna to być ciągła normalna praktyka programistyczna, aby uniknąć nadmiernego zadłużenia technicznego przez oprogramowanie. Refaktoryzacja poprawia obiektywne atrybuty kodu, które korelują z łatwością konserwacji; sprawia, że kod jest zrozumiały i łatwy w utrzymaniu, zachęca każdego programistę do myślenia i rozumienia decyzji projektowych w kontekście

zbiorowej własności / własności kodu zbiorowego, a także sprzyja pojawianiu się elementów wielokrotnego użytku (takich jak wzorce projektowe) i modułów kodu. Jednym ze sposobów wdrożenia refaktoryzacji jest Test Driven Development (TDD).

CIĄGŁA INTEGRACJA

Ciągła integracja (CI) polega na wytwarzaniu czystej struktury systemu kilka razy dziennie. Zwinne zespoły zazwyczaj konfigurują CI w celu uwzględnienia automatycznej kompilacji, wykonania testów jednostkowych i integracji kontroli źródła. Czasami CI obejmuje również automatyczne uruchamianie automatycznych testów akceptacyjnych. Ciągła integracja pozwala sprawnym zespołom programistycznym na częste sprawdzanie kodu w celu wczesnego wykrywania problemów. Ciągła integracja umożliwia członkom zespołu częstą i niezależną integrację ich pracy z podstawowym produktem, a tym samym zapewnia, że cały produkt nadal działa w sposób zintegrowany. Integracja odbywa się zazwyczaj kilka razy dziennie lub co najmniej raz dziennie. Przeprowadzanie integracji i testowanie pod koniec wydania lub projektu może prowadzić do nieoczekiwanych scenariuszy; może być za późno na naprawę, jeśli wykryta zostanie znacząca wada roblem. To z kolei prowadzi do wysokiego długu technicznego. Podczas wykonywania ciągłej integracji ważne jest, aby zespół korzystał z najnowszej działającej wersji produktu, która zwykle jest utrzymywana w centralnym systemie kontroli źródła. Zespół musi sprawdzić najnowszą wersję, wprowadzić zmiany i sprawdzić zaktualizowany kod. System ciągłej integracji wykrywa nowy kod sprawdzany, odbudowuje zmienione środowisko oprogramowania oraz przeprowadza testy jednostek i integracji, aby sprawdzić, czy wszystko działa poprawnie. Jeśli test się nie powiedzie, kompilacja nie powiedzie się, a programista, który zaznaczy kod, zostanie powiadomiony. Programista naprawia kod, aby rozwiązać problem. Dlatego kluczowe działania związane z ciągłą integracją to:

• Prowadzenie centralnego repozytorium kodu źródłowego

• Automatyzacja kompilacji i samodzielne testowanie kompilacji

• Codzienne sprawdzanie w opracowanym kodzie

• Przebudowywanie kodu linii podstawowej codziennie

• Tworzenie kopii lustrzanej środowiska testowego tak blisko środowiska produkcyjnego (na żywo), jak to możliwe

• Automatyzacja wdrażania wszędzie tam, gdzie to możliwe

AUTOMATYCZNE BUDOWANIA

Jak wspomniano w rozdziale o Continuous Integration, automatyczna kompilacja i testowanie mają kluczowe znaczenie dla pomyślnej ciągłej integracji. Ręczne budowanie testów wdrożeniowych i integracyjnych jest czasochłonne i ogranicza cel ciągłej integracji. Automatyczna kompilacja jest więc procesem tworzenia skryptów, które automatycznie wykonują kluczowe zadania programistyczne, takie jak kompilowanie kodu, uruchamianie testów, wykonywanie analizy kodu, wdrażanie w środowiskach i tworzenie dokumentacji systemu. Ważne jest, aby zespoły dostarczające Agile mogły uruchamiać automatyczną

kompilację w sposób ciągły. Automatyczne kompilacje powinny być uruchamiane regularnie, co najmniej raz dziennie.

PRZEGLĄD KODÓW I PRZEGLĄD ROBOCZY

Podczas gdy automatyzacja ma oczywiste zalety, nie może wykryć wszystkiego. Automatyczna kompilacja może pomyślnie zbudować i uruchomić test integracji. Ale wiele rzeczy może się wydarzyć na pierwszy rzut oka. Jednym z przykładów jest jakość pisanego kodu. W przypadku pożywienia ważne jest, aby kod był dobrze napisany, zrozumiały i możliwy do utrzymania. W Agile proces sprawdzania kodu odbywa się na dwa sposoby: przez przegląd partnerski i programowanie w środowisku równorzędnym. W recenzowaniu deweloper przesyła ukończony kod do peer do sprawdzenia przed zatwierdzeniem kodu. Recenzent może mieć podobny poziom doświadczenia, jak osoba, która napisała kod. W programowaniu parami dwóch programistów pracuje razem na jednej stacji roboczej. Jeden wpisuje kod, a drugi sprawdza każdą linię kodu, gdy jest wpisana. Pisząc na klawiaturze nazywa się kierowcą. Osoba przeglądająca kod nazywa się obserwatorem (lub nawigatorem). Dwaj programiści często zmieniają role. W obu tych przypadkach ważne jest, aby przegląd nie spowolnił procesu zwinności. Zdrowy rozsądek powinien przeważać przy określaniu kwoty i rygoru przeglądu kodu. Proces przeglądu powinien być zgodny ze złożonością i rygorem budowanego produktu.

9 GŁÓWNE RAMY AGILE

Zanim przejdziemy do innych frameworków Agile, pomocne byłoby określenie w kontekście wyzwań tradycyjnych (i niezwiązanych z Agile) metod opracowywania oprogramowania. Tradycyjne metodologie tworzenia oprogramowania, takie jak sztywne procedury Waterfall, wymagają dokładnego planowania z góry. Punktem wyjścia do zastosowania takich metodologii są jasne i określone wymagania. Poniższy rysunek przedstawia tradycyjny model wodospadu. Chociaż te tradycyjne metodologie mają oczywiste zalety (takie jak lepsze planowanie struktury zespołu i dystrybucji, przewidywalne budżety itp.), Mają również własne wyzwania. Metodologie te cechują się brakiem elastyczności, wyraźnym rozłączeniem się z oczekiwaniami klientów (jeśli projekty są zbyt duże), trudnościami z dostosowaniem się do zakresu lub zmian wymagań oraz niemożnością wykrycia wad projektowych do czasu fazy integracji i testów. Metoda wodospadu została opracowana przy użyciu podejścia inżynieryjnego. Zrozumiałe jest planowanie wszystkiego przed rozpoczęciem budowy mostów lub budynków. Niestety, tworzenie oprogramowania nie działa w ten sposób, ponieważ warunki są inne. Dlatego też, jako korekta niedociągnięć tradycyjnych metodologii, wprowadzono metodologię Agile. Agile zastępuje sztywny i przewidujący charakter tradycyjnej metodologii metodami adaptacyjnymi i zwinnymi. Są to określone kryteria określone przez klienta dla każdego wymagania funkcjonalnego. Kryteria akceptacji są napisane w prosty sposób i z perspektywy klienta. Kluczowe zasady metodologii Agile to:

- Wczesne i częste testy
- Przyrostowy projekt

• Częste (codzienne) wdrożenie

• Większa satysfakcja klienta

• Ciągła integracja

• Krótkie cykle rozwojowe

• Spójny zespół

• Lepsza przejrzystość wśród klientów i zespołu programistów

• Planowanie przyrostowe

• Pojedyncza podstawa kodu

• Proces samodostosowujący się

• Adaptacja do zmiany

Metody zwinne nie odnoszą się do jednego konkretnego podejścia, ale stanowią grupę indywidualnej metodologii, która wdraża zwinne zasady. Opracowano kilka metod zwinnych:

• Scrum

• Dynamic Development Development Method (DSDM)

• Metody krystaliczne

• Funkcja Driven Development

• Lean Development

• Extreme Programming (XP)

• Adaptacyjne tworzenie oprogramowania

• Agile Project Management

• Kanban

• Scaled Agile Framework (SAFe)

Extreme Programming (XP) to jedna z najbardziej znanych metodologii Agile.

EKSTREMALNE PROGRAMOWANIE (XP)

Extreme programming (XP) to metodologia opracowywania oprogramowania, która ma na celu poprawę jakości oprogramowania i reagowanie na zmieniające się wymagania klientów. Podkreśla zadowolenie klienta w stosunku do wszystkiego. Dostarcza oprogramowanie, tak jak potrzebuje tego klient, umożliwiając programistom reagowanie na zmieniające się wymagania klientów. W związku z tym XP zaleca częste publikacje w krótkich cyklach rozwojowych. Kluczowymi elementami XP są: programowanie w parach, obszerny przegląd kodu, testowanie jednostkowe całego kodu, unikanie programowania funkcji aż do ich potrzeby, płaska struktura zarządzania, prostota i klarowność kodu, oczekiwanie zmian w

wymaganiach klienta w miarę upływu czasu i problemu jest lepiej zrozumiany i częsty kontakt z klientem i programistami. XP ulepsza projekt oprogramowania na pięć podstawowych sposobów - komunikację, prostotę, informację zwrotną, szacunek i odwagę. Programiści XP nieustannie komunikują się ze swoimi klientami i członkami zespołu. Utrzymują projekt prosty i czysty. Opierają się na częstych opiniach klientów i testowaniu ich oprogramowania już na wczesnym etapie cyklu życia. Dostarczaj oprogramowanie do klientów tak wcześnie, jak to możliwe i często włączaj informacje zwrotne. Pogłębia to szacunek, jaki otrzymują od klientów; zwiększa także szacunek dla wkładów, które każdy członek zespołu tworzy. XP pozwala programistom odważnie reagować na zmieniające się wymagania i technologię. Rozróżniającym aspektem XP jest jego prosta zasada. Te zasady, w połączeniu, stanowią potężne narzędzie do osiągnięcia ekstremalnego programowania. Podsumowanie zasad XP znajduje się poniżej:

Planowanie

- Napisane są historie użytkowników.
- Planowanie wydań tworzy harmonogram wydań.
- Zrób częste małe wydania.
- Projekt podzielony jest na iteracje.
- Planowanie iteracji rozpoczyna się od iteracji.

Zarządzający

- Daj zespołowi dedykowaną otwartą przestrzeń roboczą.
- Ustal zrównoważone tempo.
- Spotkanie stand-up rozpoczyna się każdego dnia.
- Mierzona jest prędkość projekcyjna.
- Poruszaj ludźmi wokół.
- Napraw XP, gdy się zepsuje.

Projektowanie

- Prostota.
- Wybierz metaforę systemową.
- Używaj kart CRC do sesji projektowych.
- Twórz rozwiązania spajków, aby zmniejszyć ryzyko.
- Żadna funkcja nie zostanie dodana wcześniej.
- Refaktor zawsze i wszędzie, gdzie to możliwe.

Kodowanie

• Klient jest zawsze dostępny.

• Kod musi być zapisany w uzgodnionych standardach.

• Najpierw zakoduj test jednostki.

• Cały kod produkcyjny jest zaprogramowany w parach.

• Tylko jedna para integruje kod na raz.

• Integruj często.

• Skonfiguruj dedykowany komputer integracyjny.

• Użyj zbiorowej własności.

Testowanie

• Wszystkie kody muszą mieć testy jednostkowe.

• Cały kod musi przejść wszystkie testy jednostkowe, zanim zostanie zwolniony.

• Po znalezieniu błędu tworzone są testy.

• Testy akceptacyjne są często przeprowadzane, a wyniki są publikowane.

Extreme programming (XP) pochodzi z lat 90. został on skonceptualizowany przez Kenta Blacka, który próbował znaleźć lepszy sposób na rozwój oprogramowania. Zdał sobie sprawę, że rozwój oprogramowania, w przeciwieństwie do modelu kaskadowego, jest procesem płynnym; wymagania ulegną zmianie w trakcie projektu. Dlatego metoda tworzenia oprogramowania musi być w stanie dostosować się do zmieniających się wymagań. Cztery podstawowe czynności, które XP proponuje do rozwoju oprogramowania to:

• Kodowanie: w XP kodowanie rozpoczyna się bardzo wcześnie, ponieważ jest uważane za najbardziej ważny produkt procesu tworzenia oprogramowania.

• Testowanie: XP podkreślił skuteczność testów, aby sprawdzić, czy opracowany kod działa. Programiści korzystają z zautomatyzowanych testów jednostkowych i piszą testy możliwe złamanie napisanego kodu. Poprawia to odporność kodu.

• Słuchanie: Słuchanie uważane jest za niezwykle ważną miękką umiejętność w XP. Oczekuje się od każdego programisty uważnego słuchania, aby mógł zrozumieć potrzeby klienta i opracowywanie produktów jak najbliżej wymagań.

• Projektowanie: XP kładzie nacisk na tworzenie prostej i dopasowanej do celu struktury projektu która unika zbytniej zależności.

XP jest powiązany z 28 zasadami i praktykami. Najważniejsze z nich to:

• Historie użytkowników: Historie użytkowników są traktowane jako mniejsze przypadki użycia. To ułatwia planowanie i szacowanie kosztów.

• Małe wydania: XP podkreśla małe, proste i częste wydania zawierające jedno lub więcej wymagań w każdym wydaniu.

• Standardy: Zespół przestrzega standardów dotyczących nazw, nazw klas i metod śledzenia.

• Zbiorowa własność: Cały kod jest własnością całego zespołu, a nie jednostki. Kod jest sprawdzany i aktualizowany przez wszystkich.

• Prosta konstrukcja: projekt i wykonanie są utrzymywane tak proste, jak to możliwe, podczas gdy zapewnienie wymaganej funkcjonalności.

• Refaktoryzacja: Produkt jest stale dostosowywany i udoskonalany przez wszystkich członków zespołu.

• Testowanie: każde małe wydanie musi przejść pomyślnie testy. Testy są tworzone najpierw, a następnie wykonuje się kodowanie, aby przejść wstępnie napisany test.

• Programowanie w parach: programiści pracują w parach; pracują razem na jednej maszynie. Jedna osoba pisze kod, a druga przegląda; rola zostaje odwrócona.

• Ciągła integracja: Oprogramowanie jest budowane kilka razy dziennie. Pozwala to uniknąć ostatniej minuty wyzwania związane z integracją oprogramowania.

• 40-godzinny tydzień pracy: Praca zespołowa 40 godzin tygodniowo, aby uniknąć wypalenia zawodowego.

• Klient na miejscu: klient musi być dostępny dla zespołu tak często, jak możliwe, aby nie było luki w komunikacji.

Podsumowując, XP najlepiej nadaje się do projektów, które są narażone na zmieniające się technologie, w których możliwe są małe częste publikacje, gdzie projekty są małe i można je łatwo zarządzać.

SCRUM

Scrum to popularny układ Agile do wykonywania złożonych projektów. Termin "scrum" pochodzi od formacji scrumowej używanej przez zespoły rugujące, które mają wysoką skuteczność i są funkcjonalne. Pierwotnie przeznaczony do tworzenia oprogramowania, Scrum może być stosowany do każdego złożonego, innowacyjnego zakresu prac. To, co wyjątkowe dla Scrum, to jego zaangażowanie w krótkie iteracje pracy. Scrum używał iteracji o stałej długości zwanych Sprintami. Sprinty zwykle nie przekraczają 30 dni; zespół próbuje zbudować potencjalnie możliwy do uwolnienia produkt, zwiększając każdy Sprint.

Proces Scruma jest stosunkowo prosty:

• Backlog produktu:

*Właściciel produktu tworzy priorytetową listę życzeń nazywaną zaległym produktem.

• Planowanie sprintu:

* Podczas planowania sprintu zespół wyciąga kilka elementów ze szczytu listy życzeń, tworzy zaległe sprinty i decyduje, w jaki sposób zaimplementować te elementy.

* Zespół decyduje, ile czasu poświęcić na ukończenie pracy. Zwykle trwa od dwóch do czterech tygodni - zwany Sprintem.

• Daily Stand-up:

* Zespół spotyka się każdego dnia, aby ocenić jego postępy (codzienny Scrum). Ponadto, zwane codziennym stoiskiem, zwykle jest to szybkie 15-minutowe spotkanie.

* Scrum Master śledzi ogólny postęp i utrzymuje zespół w osiąganiu celu.

• Sprint Demo:

* Prezentacja jest prezentowana właścicielowi produktu i innym zainteresowanym podmiotom na temat tego, co będzie zawierać ostateczny produkt podlegający sprzedaży.

* Po zakończeniu sprintu praca powinna być potencjalnie możliwa do wysłania.

• Retrospektywa sprintu:

* Sprint kończy się przeglądem sprintu i retrospekcją. Zespół omówił, co poszło dobrze i nie poszło dobrze, i podejmuje działania, aby ulepszyć następny sprint.

Scrum ma trzy główne role - Product Owner, Scrum Master i Scrum Team.

• Właściciel produktu: Właściciel produktu ponosi odpowiedzialność za ogólną wizję produktu. Ponownie nadaje priorytet Backlogowi produktu i dostosowuje go do poszczególnych planów wydań. Określa priorytety zespołowi programistów. Ma wiodącą rolę i niesie znaczące autorytety. Rozważa interesy interesariuszy i sprawdza, co dzieje się w cyklu Sprintu. Dla zespołu Scrum, dostępność właściciela produktu jest bardzo ważna, ponieważ zapewnia przejrzystość wymagań klientów i może odpowiadać na pytania.

• Scrum Master: Scrum Master działa jako pomocnik dla właściciela produktu i zespołu. Nie zarządza zespołem, ale zapewnia usunięcie wszelkich przeszkód utrudniających zespołowi osiągnięcie jego celów sprintu. Współpracuje z właścicielem produktu i organizacją, aby umożliwić Scrum. Współpracuje z właścicielem produktu, jak zmaksymalizować zwrot nakładów inwestycyjnych na inwestycje. Scrum Master pomaga zespołowi zachować kreatywność i produktywność.

• Zespół Scrumowy: Zespół programistyczny Scrum jest zazwyczaj samodzielny i funkcjonalny. Zespół planuje jeden Sprint naraz z właścicielem produktu i ma autonomię i odpowiedzialność w osiąganiu celów Sprintu. Zespół intensywnie współpracuje i jest w jednym pokoju, aby być masowo produktywny. Konstytucja Zespołu Scrumowego liczy oficjalnie 3-9 członków; obejmuje członków z analizą biznesową, programowaniem, testowaniem, projektantem, domeną i umiejętnościami inżynierii oprogramowania.

METODA ROZWOJU SYSTEMÓW DYNAMICZNYCH (DSDM)

Metoda rozwoju systemów dynamicznych (DSDM) to zwinny system dostarczania projektów, wykorzystywany przede wszystkim jako metoda rozwoju oprogramowania. Jest to struktura, która zawiera wiele aktualnej wiedzy na temat zarządzania projektami. DSDM jest zakorzenione w społeczności programistów, ale konwergencja rozwoju oprogramowania,

inżynierii procesowej, a tym samym projektów rozwoju biznesu, zmieniła strukturę DSDM i stała się ogólną strukturą złożonych zadań rozwiązywania problemów. Ramy DSDM można wdrożyć dla zwinnych i tradycyjnych procesów rozwojowych. DSDM łączy razem zwinność, elastyczność i zarządzanie projektem. DSDM jest niezależny od dostawcy, obejmuje cały cykl życia projektu i zapewnia wskazówki dotyczące najlepszych praktyk w zakresie terminowego, budżetowego dostarczania projektów, ze sprawdzoną skalowalnością, umożliwiając obsługę projektów o dowolnej wielkości i dla dowolnego sektora biznesowego. Istnieje 9 zasad leżących u podstaw DSDM.

1. Aktywne zaangażowanie użytkownika - Imperatyw.

2. Zespoły muszą być uprawnione do podejmowania decyzji.

3. Skoncentruj się na częstym dostarczaniu.

4. Kryterium dla zaakceptowanego produktu (Fitness dla Firm).

5. Rozwój iteracyjny i przyrostowy - Obowiązkowe.

6. Wszelkie zmiany w trakcie rozwoju muszą być odwracalne.

7. Wymagania bazują na wysokim poziomie.

8. Testowanie jest zintegrowane przez cały cykl życia.

9. Podejście oparte na współpracy i współpracy.

Struktura projektu:

Projekt DSDM składa się z 7 etapów etapów, które są zorganizowane i osadzone w bogatym zestawie ról i obowiązków oraz wspierane przez kilka podstawowych technik.

• Role i obowiązki

• Organizacja i rozmiar drużyny

• 7 faz do ich regulacji

1. Wstępny projekt

2. Analiza wykonalności

3. Badanie biznesowe

4. Funkcjonalna iteracja modelu

5. Projektowanie i budowa Iteracja

6. Wdrożenie

7. Post-Project

DSDM zaleca stosowanie kilku sprawdzonych technik, w tym:

• Priorytetowanie MoSCoW: Ponieważ projekty DSDM są przygotowane na czas i budżet, nacisk kładziony jest na wymagania, których użytkownicy potrzebują najbardziej. Dlatego DSDM używa reguł MoSCoW do ważenia ważności wymagań. DSDM od samego początku ustala koszt, jakość i czas oraz wykorzystuje MoSCoW do ustalania priorytetów zakresu do moszczów, powinności, pojemników i nie musi dostosowywać dostarczanego projektu, aby spełnić określone ograniczenie czasowe.

* Musi: MUSI mieć ten wymóg, aby spełnić potrzeby biznesowe.

* Powinien: POWINIEN mieć ten wymóg, jeśli to możliwe, ale powodzenie projektu nie polega na tym.

* Mógł: MUSI mieć ten wymóg, jeżeli nie wpływa to na sprawność potrzeb biznesowych projektu.

* Chcesz: NIE BĘDZIE zawierał wymogu, że zainteresowane strony zgodziły się, że nie zostaną wdrożone w każdym wydaniu, ale mogą być brane pod uwagę w przyszłości.

• Boks czasowy: Boks czasowy to interwał, zwykle nie dłuższy niż 2, 4 lub 6 tygodni, w którym należy wykonać określony zestaw zadań. DSM zmniejsza funkcjonalność na rzecz dostarczania w czasie.

• Prototypowanie: w DSDM prototypowanie koncentruje się na dwóch zasadach - Częste dostarczanie

& Przyrostowy rozwój. Przeprowadzane są następujące rodzaje prototypów:

* Prototyp biznesowy: ocena wymagań biznesowych zostanie spełniona

* Prototyp użyteczności: oceń interfejs użytkownika

* Prototyp wydajności: sprawdza wymagania wydajnościowe rozwiązania

* Prototyp możliwości: oceń możliwe opcje

METODOLOGIA KRYSZTAŁOWA

Crystal Methodology to lekkie i elastyczne podejście do tworzenia oprogramowania. Opracowany przez Alistair Cockburn, Crystal Methodology koncentruje się przede wszystkim na ludziach i ich interakcji podczas projektowania oprogramowania. W słowach Cockburn, "Crystal to rodzina ludzkich, adaptacyjnych, ultralekkich metod opracowywania oprogramowania" stretchto-fit ".

• Kryształ jest "zasilany przez człowieka" - oznacza to, że ludzie są najważniejszym aspektem Kryształu, a wszystkie procesy i narzędzia są zbudowane wokół nich. Crystal podkreśla, że zespoły programistyczne są samowystarczalne i samoorganizujące się, dzięki czemu mogą usprawnić procesy w miarę postępu prac i zwiększania ich skuteczności.

• Kryształ jest "adaptacyjny" - procesy i narzędzia Crystal są dostosowane do wymagań i kontekst projektu. Są one dostosowane do potrzeb biznesowych i technicznych wymagania projektu.

• Kryształ jest "ultralekki" - Crystal stara się zachować prostotę (lub światło), minimalizując dokumentację, zarządzanie i raportowanie. Usuwa niepotrzebny bałagan, koncentrując się na wartości biznesowej i funkcjonalnym oprogramowaniu. Otwarta komunikacja w związku z tym między członkami zespołu zachęca się do przejrzystego przepływu informacji.

Rodzina metodologii Crystal składa się z następujących wariantów: Crystal Clear, Crystal Yellow, Crystal Orange, Crystal Orange Web, Crystal Red, Crystal Maroon, Crystal Diamond i Crystal Sapphire. Waga metodyki Crystal zależy od środowiska projektu i wielkości zespołu. Na przykład zespół sześciu programistów wybierze opcję krystalicznie czystą; zespół składający się z 10-40 członków i żywotność na okres 1-2 lat przejdzie na Crystal Orange. W przypadku bardzo dużych i ryzykownych projektów bardziej odpowiednie mogą być metody Crystal Sapphire lub Crystal Diamonds. W celu udanego wdrożenia podejścia Crystal, Crystal precyzuje, że podejście programistyczne powinno być iteracyjne i przyrostowe, zaangażowanie użytkownika powinno być aktywne, a zobowiązania w zakresie realizacji są spełnione.

FUNKCJA ROZWÓJ NAPĘDU (FDD)

Projektowanie zorientowane na funkcje (FDD) jest iteracyjnym i przyrostowym procesem tworzenia oprogramowania zwinnym. FDD to proces oparty na modelu i krótkiej iteracji. Zaczyna się od ogólnego kształtu modelu unikatowego dla każdego projektu, a następnie kontynuuje serię dwutygodniowych "projektów według funkcji, kompilacji według funkcji" iteracji. Funkcje muszą być oparte na wartości i przydatne dla użytkowników końcowych. FDD stosuje następujące osiem praktyk w procesie rozwoju:

• Modelowanie obiektów domenowych

• Opracowywanie według funkcji

• Własność komponentów / klas

• Zespoły Cech

• Inspekcje

• Zarządzanie konfiguracją

• Regularne konstrukcje

• Widoczność postępu i wyników

FDD składa się z pięciu podstawowych działań, mianowicie rozwoju całościowego modelu, tworzenie listy funkcji, planowanie według funkcji, projektowanie według funkcji i budynku według funkcji.

ROZWÓJ OPROGRAMOWANIA LEAN

Lean Software Development to iteratywna metodologia opracowana przez Mary i Tom Poppendieck. W oparciu o zasady i praktyki Lean firm takich jak Toyota, Lean Software Development koncentruje się na dostarczaniu Klientowi wartości, a także na wydajności

Strumienia Wartości - mechanizmach zapewniających tę Wartość. Główne zasady metodologii Lean obejmują:

• Eliminowanie odpadów

• Wzmacnianie nauki

• Decydowanie tak późno, jak to możliwe

• Zapewnienie tak szybkiego, jak to możliwe

• Wzmocnienie pozycji zespołu

• Integralność budynków w

• Oglądanie całości

Poniższa tabela pokazuje, w jaki sposób powyższe zasady Lean mają zastosowanie do Lean Software Development.

Eliminowanie odpadów

Unikamy:

• niepotrzebny kod lub funkcjonalność

• rozpoczęcie więcej niż można ukończyć

• opóźnienie w procesie tworzenia oprogramowania

• niejasne lub stale zmieniające się wymagania

• biurokracja

• powolna lub nieefektywna komunikacja

• częściowo wykonana praca

• Wady i problemy z jakością

• Przełączanie zadań

Wzmacnianie nauki

Zachęcamy:

• Programowanie parami

• Przeglądy kodu

• Dokumentacja

• Wiki - aby baza wiedzy rosła stopniowo

• Dokładnie skomentował kod

• Sesje dzielenia się wiedzą

• Trening

• Użyj narzędzi do zarządzania wymaganiami / historiami użytkowników

Decydowanie jako późno jak to możliwe

• Zdecyduj, które funkcje należy uwzględnić w każdej iteracji i przeanalizuj je w samą porę, aby je opracować

Szybka dostawa jak to możliwe

• Nie nadużywaj inżynierii

• Najpierw zbuduj proste rozwiązanie i stopniowo zwiększaj jego wartość

• Posiadać odpowiedni zespół, aby osiągnąć cele zespołu osiągnięte przy minimalnej biurokracji

• Mieć odpowiednich ludzi, aby mogli rozwiązywać problemy, rozwiązywać problemy i podejmować szybkie decyzje

Wzmocnienie zespołu

• Zachęcaj ludzi do wyrażania swoich opinii

• Okazuj empatię za ich punkt widzenia

• Daj ludziom uprawnienia do podejmowania decyzji dotyczących ich pracy

Budowanie uczciwości

Poniższe zwinne praktyki poprawiają jakość oprogramowania produktu:

• Programowanie w parze

• Testuj napędzany rozwój

• Stałe sprzężenie zwrotne - Sprawdź i dostosuj

• Minimalizuj czas między etapami

• Częsta integracja

• Automatyzacja

• Zarządzaj kompromisami

Oglądanie całości

• Podejmij zwinną adopcję od góry

• Buduj zespół według produktów lub projektów, zamiast ról lub umiejętności

• Zorganizuj zespoły tak, aby były kompletne, zdyscyplinowane, i kolokacja

• Zoptymalizuj cały strumień wartości

ADAPTACYJNY ROZWÓJ OPROGRAMOWANIA

Adaptive software development (ASD) to proces opracowywania oprogramowania, który ewoluował z pracy Rapid Application Development (RAD) autorstwa Jima Highsmitha i Sama Bayera. Uosabia zasadę, że ciągłe dostosowywanie procesu do pracy pod ręką jest normalne. Nazywa się to adaptacyjnym, ponieważ zapewnia możliwość dostosowania się do zmian i jest dostosowywany w turbulentnych środowiskach z produktami ewoluującymi przy minimalnym planowaniu i uczeniu się. Cykl życia ASD ma charakter cykliczny, a kluczowe fazy

• Spekuluj

• Współpracuj

• Ucz się

W ASD termin spekulacja jest używany zamiast terminu plan. To jednak nie robi wymień planowanie; ale uznaje rzeczywistość niepewności w złożonych problemach. Spekuluj zachęca do eksploracji i eksperymentowania. Zalecane są krótkie iteracyjne cykle. Współpraca jest kluczem do ASD, ponieważ umożliwia budowanie złożonych aplikacji w turbulentnych środowiskach. Współpraca wymaga umiejętności wspólnej pracy w celu gromadzenia, analizowania i stosowania dużej ilości informacji w złożonym problemie. Dowiedz się część ASD stale zwiększając wiedzę zespołu poprzez praktyki, takie jak przeglądy techniczne, retrospekcje projektu i grupy fokusowe. Podsumowując, ASD następuje po cyklu Spekuluj - Współpracuj - Ucz się.

AGILE PROJECT MANAGEMENT

Agile Project Management to podejście oparte na wartościach, które umożliwia kierownikom projektów (PM) wykonywanie pracy o wysokim priorytecie i wysokiej jakości. Pozwala to premierom efektywnie uwzględniać zmiany bez przechodzenia przez podatne na błędy podejście do zarządzania projektami. Zwinne zarządzanie projektami koncentruje się na dostarczaniu najpierw funkcji o największej wartości biznesowej, przy jednoczesnym zarządzaniu kosztami, czasem i zakresem zadań. Zgodnie z podejściem Agile, zarządzanie projektami Agile redukuje złożone projekty do małych, użytecznych funkcji, które są dostarczane w cyklu od dwóch do czterech tygodni. Osiąga to poprzez dzielenie odpowiedzialność między trzema kluczowymi rolami:

• Właściciel produktu zajmuje się ustalaniem celów projektu, kompromisem między harmonogramem a zakresem, dostosowywanie się do zmieniających się wymagań projektu i ustalanie priorytetów dla funkcji produktu.

• Scrum Master pomaga zespołowi nadawać priorytet zadaniom i eliminować przeszkody w wykonywaniu ich zadań.

• Członkowie zespołu bezpośrednio zajmują się ich codziennymi zadaniami, raportowaniem postępów i jakością kontrola ich produktu.

KANBAN

Kanban to metoda zarządzania tworzeniem produktów z naciskiem na ciągłe dostarczanie, a jednocześnie nie obciążanie zespołu programistów. Kanban opiera się na trzech podstawowych zasadach:

• Wizualizuj to, co robisz dzisiaj (workflow)

• Ogranicz ilość pracy w toku (WIP)

• Zwiększ przepływ

Dlatego Kanban kładzie nacisk na ciągłość dostaw, stałą współpracę i kontynuacja nauczania. Kanban wywodzi się ze świata Lean Manufacturing jako sposób produkcji "Just-in-Time". Został opracowany przez Toyotę w 1950 roku. Podobnie jak inne frameworki Agile, Kanban skupia się na iteracyjnym dostarczaniu, gdzie zespoły pracują w ciągu dwóch-czterech tygodni "sprintów", aby dostarczyć małą partię historii użytkowników. Przepływ pracy jest wizualizowany na tablicy Kanban. Możesz wizualizować, jak praca przepływa podczas sprintu poprzez bardzo szybkie iteracje projektowania, rozwijania i testowania, które odbywają się w każdym sprintu. Tablica Kanban pomaga nam również określić zdolność do dalszej pracy (w zależności od statusu pracy w toku). Jeśli istnieje pojemność, zespół ściąga następny element do pracy. Po zakończeniu zadania każdy element jest oznaczony jako Gotowe. Skoncentrowanie się na zwinnym przepływie procesu za pomocą Kanban może czasami być jeszcze bardziej wydajne, wynikające w krótszym czasie realizacji i wyższej wydajności. Uwaga, że Kanban jest podejściem usprawniającym proces, a nie procesem własnym. Może być zastosowany do Agile lub Scrum; Kanban uzupełnia je, zamiast je zastępować.

SKALOWANE AGILE FRAMEWORK (BEZPIECZNIE)

Scaled Agile Framework (SAFe) to swobodnie odkrywana baza wiedzy o sprawdzonych, zintegrowanych wzorcach dla rozwoju Lean-Agile w skali przedsiębiorstwa. Jest skalowalny i modułowy, pozwalając każdej organizacji na zastosowanie go w sposób zapewniający lepsze wyniki biznesowe i szczęśliwszych, bardziej zaangażowanych pracowników. SAFe synchronizuje wyrównanie, współpracę i dostarczanie dla dużej liczby zespołów Agile. Obsługuje zarówno oprogramowanie, jak i rozwój systemów, od skromnej skali od 100 praktyków do największych rozwiązań programowych i złożonych systemów cyber-fizycznych, systemów, które wymagają tysięcy ludzi do tworzenia i utrzymywania. SAFe został opracowany w terenie, w oparciu o pomoc klientom w rozwiązywaniu najtrudniejszych problemów związanych z skalowaniem. Wykorzystuje trzy główne obszary wiedzy: zwinny rozwój, rozwój produktów Lean i myślenie systemowe SAFe może być skonfigurowany z trzema lub czterema poziomami organizacyjnymi opisanymi powyżej, plus warstwa Fundacji, jak opisano poniżej:

• Poziom zespołu

• Poziom programu

• Poziom strumienia wartości

• Poziom portfela

• Poziom fundacji

Najważniejsze wartości SAFe są głównymi zasadami, które dyktują zachowanie i działanie. Wartości te mogą pomóc ludziom zrozumieć, co jest dobre, a co złe, gdzie skupić uwagę i jak pomóc firmom ustalić, czy są na właściwej drodze do realizacji swoich celów biznesowych.

• Wyrównanie

• Wbudowana jakość

• Przejrzystość

• Realizacja programu

Praktyki SAFe opierają się na dziewięciu podstawowych zasadach, które rozwinęły się z zasad i metod Agile, rozwoju produktu Lean, myślenia o systemach i obserwacji udanych przedsiębiorstw. Istnieje konkretny artykuł dla każdej zasady na stronie internetowej SAFe, a wcielenie zasad pojawia się w Ramach.

• Przyjrzyj się poglądowi ekonomicznemu

• Zastosuj myślenie systemowe

• Przyjmij zmienność; zachować opcje

• Buduj stopniowo dzięki szybkim, zintegrowanym cyklom uczenia się

• Kamienie milowe na obiektywnej ocenie systemów roboczych

• Wizualizuj i ograniczaj WIP, zmniejsz rozmiary partii i zarządzaj długością kolejki

• Zastosuj kadencję, zsynchronizuj z planowaniem crossdomain

• Odblokuj wewnętrzną motywację pracowników wiedzy

• Zdecentralizuj proces decyzyjny

10 GLOSARIUSZ

Kryteria akceptacji: są to określone kryteria określone przez klienta dla każdego wymagania funkcjonalnego. Kryteria akceptacji są napisane w prosty sposób i z perspektywy klienta.

Testowanie akceptacyjne: Testowanie akceptacyjne jest działaniem walidacyjnym przeprowadzanym w celu ustalenia, czy system spełnia kryteria akceptacji. Jest to ostatnia faza procesu testowania oprogramowania.

Agile: koncepcyjne ramy dla podejmowania projektów oprogramowania. Metody zwinne to rodzina procesów rozwojowych, a nie pojedyncze podejście do tworzenia oprogramowania.

Test na rozwój akceptacji (ATDD): analogiczny do rozwoju opartego na testach,

Test oparty na akceptacji Driven Development (ATDD) obejmuje członków zespołu z różnejperspektywy (klient, rozwój, testowanie) współpracują, aby napisać testy akceptacyjne przed wdrożeniem odpowiedniej funkcjonalności.

Zautomatyzowana kompilacja: Odnosi się do zautomatyzowanego procesu, który konwertuje pliki i inne zasoby objęte odpowiedzialnością twórców oprogramowania na produkt końcowy lub konsumpcyjny.

Backlog: Backlog to lista funkcji lub zadań technicznych, które zespół utrzymuje i które w danym momencie są znane jako niezbędne i wystarczające do ukończenia projektu lub wydania. Backlog jest podstawowym punktem wejścia dla wiedzy o wymaganiach i jedynym autorytatywnym źródłem określającym pracę do wykonania.

Backlog Grooming: Backlog grooming to sytuacja, w której właściciel produktu i część lub całość reszty zespołu dokonuje przeglądu pozycji w backlogu, aby upewnić się, że backlog zawiera odpowiednie pozycje, że są one priorytetowe oraz że przedmioty na górze zaległości są gotowe do dostawy.

Zachowanie oparte na zachowaniu: Rozwój oparty na zachowaniu (lub BDD) jest zwinną techniką tworzenia oprogramowania, która zachęca do współpracy między programistami, kontrolą jakości i uczestnikami nietechnicznymi lub biznesowymi w projekcie oprogramowania. BDD koncentruje się na uzyskaniu jasnego zrozumienia pożądanego zachowania oprogramowania poprzez dyskusję z interesariuszami. Rozszerza TDD, pisząc przypadki testowe w języku naturalnym, który nie mogą czytać programiści.

Wąskie gardło: wąskie gardło jest rodzajem zatorów w systemie, który pojawia się, gdy obciążenie dociera do danego punktu szybciej niż ten punkt może sobie z nim poradzić.

Błędy: Błąd oprogramowania powoduje awarię programu lub generuje nieprawidłowe wyniki. Jest to spowodowane niewystarczającą lub błędną logiką i może być błędem, błędem, defektem lub usterką.

Wykres Burndown: Wykres zagłębienia jest wizualnym narzędziem do mierzenia i wyświetlania postępów. Wizualnie, wykres pożegnania jest po prostu liniowym wykresem przedstawiającym pozostałą pracę w czasie. Wykresy Burndown służą do mierzenia postępu zwinnego projektu na poziomie iteracji i projektu.

Właściciel aplikacji biznesowych (właściciel produktu AKA): odpowiedzialny jest właściciel produktu dla wizji projektu, realizacji przypadku biznesowego, wymagań wysokiego poziomu, priorytetów i ostatecznej akceptacji. Oczekuje się, że właściciel produktu lub jego pełnomocnik będą w znacznym stopniu współpracować z głównym zespołem. Właściciele produktów czasami przekazują codzienne interakcje zespołowe do kierowania firmą. W obu przypadkach zespół musi mieć kogoś, kto może szybko odpowiedzieć na pytania o wymagania i podjąć decyzje dotyczące kompromisu.

Business Lead: The Business Lead współpracuje z Analitykami Rozwiązań, aby przetłumaczyć procesy biznesowe i wymagania dotyczące rozwiązań na produkty z zakresu metodologii dostarczania rozwiązań, które mogą być wykorzystywane do rozwoju, podpisują się pod procesami biznesowymi procesu projektowego i wymaganiami rozwiązania oraz zapewniają spójność wymagań w całym procesie , wymagania i interfejs użytkownika.

Wartość biznesowa: przedmioty o wartości materialnej i ogólnie wartości najbliższej firmie. Przykładem wartości biznesowej może być nowy ekran, raport lub funkcja dla zasobu. Ważne jest, aby pamiętać, że przedmioty nie oznaczone jako wartość biznesowa nadal generują wartość dla firmy. Jednak wartość jest długofalową korzyścią lub czymś, co nie jest namacalne dla użytkownika.

Zbiorowa własność: Zbiorowa własność kodu, jak sama nazwa wskazuje, jest wyraźną konwencją, że "każdy" członek zespołu jest nie tylko dozwolony, ale w rzeczywistości ma pozytywny obowiązek, aby wprowadzić zmiany do "dowolnego" pliku kodu, jeśli to konieczne: albo do zakończenia zadanie rozwojowe, naprawa wady, a nawet poprawa ogólnej struktury kodu.

Ciągły rozwój: Ciągłe wdrażanie może być uważane za przedłużenie ciągłej integracji, mające na celu zminimalizowanie czasu realizacji, czasu, jaki upłynął między opracowaniem jednej nowej linii kodu i nowego kodu używanego przez użytkowników na żywo do produkcji.

Ciągła integracja: ciągła integracja jest wykonywana w celu zminimalizowania czasu trwania i wysiłek wymagany przez każdy cykl integracji oraz dostarczenie wersji produktu odpowiedniej do wydania w dowolnym momencie.

Karty CRC: Karty CRC (dla klasy, obowiązków, współpracowników) to działanie łączące światy gier fabularnych i projektowania obiektowego.

Daily Standup / Daily Meeting: Daily Standup to całe spotkanie zespołu, które odbywa się o tej samej porze każdego dnia, który zwykle trwa 15 minut lub krócej. Spotkanie ma na celu umożliwić całemu zespołowi synchronizację ze sobą oraz zrozumienie przepływu i wyzwań związanych z procesem rozwoju. Każdy członek zespołu powinien podać następujące informacje: co zrobiłem wczoraj, co mam dziś zrobić i jakie przeszkody mam obecnie?

Wada: niepożądany wynik, który zostaje wykryty po tym, jak odpowiednia strona wyrejestrowała się, gdy działa jakaś funkcjonalność. Zazwyczaj znajduje się poza bieżącą iteracją.

Zależność: zależność powstaje, gdy jedno zadanie polega na ukończeniu innego zadania. Na przykład zakończenie funkcji X może zależeć od funkcji Y działającej. W tym przypadku funkcja X zależy od funkcji Y.

Gotowe: również określane jako "Gotowe", termin ten jest używany do opisania wszystkich różnych zadań, które muszą się wydarzyć, zanim historia zostanie uznana za potencjalnie możliwą do wydania.

Epicka: bardzo duża historia użytkownika, która ostatecznie rozpada się na mniejsze historie.

Szacowanie: proces uzgadniania pomiaru wielkości dla opowieści, a także zadań wymaganych do wdrożenia tych historii w zaległościach dotyczących produktu.

Testy eksploracyjne: Testowanie eksploracyjne jest stylem lub podejściem do oprogramowania testującego, które często kontrastuje z "testowaniem skryptowym". Zaleca wykonywanie różnych działań związanych z testowaniem (takich jak projektowanie testów,

wykonywanie testów i interpretacja wyników) w sposób przeplatany, przez cały projekt, a nie w ustalonej kolejności i fazie.

Ułatwienie: facylitatorem jest osoba, która wybiera lub ma wyraźną rolę prowadzenia spotkania. Ta rola zwykle oznacza, że moderator będzie miał niewielki udział w dyskusjach na temat spotkania, ale skupi się przede wszystkim na tworzeniu warunków dla efektywnych procesów grupowych, w dążeniu do celów, dla których zwołane zostało spotkanie.

Cecha: Funkcje są wysokopoziomowymi potrzebami biznesowymi. Funkcje zazwyczaj podlegają bardziej szczegółowej analizie i rozkładają się na bardziej w pełni zdefiniowane historie użytkowników przed wdrożeniem przez zespół.

Funkcja pełzania: Pełzanie cech występuje, gdy oprogramowanie staje się skomplikowane i trudne w użyciu z powodu zbyt wielu funkcji.

Częste wydania: proces, w ramach którego produkt jest często przesyłany do użytkowników końcowych w celu uzyskania opinii.

Given-When-Then: Formuła Given-When-Then jest szablonem mającym na celu przypomnienie pisania testów akceptacji dla historii użytkownika: (biorąc pod uwagę) pewien kontekst; (Kiedy) wykonywane są pewne działania; (Następnie) należy uzyskać zestaw możliwych do zaobserwowania konsekwencji.

Retrospektywa Heartbeat: Zespół spotyka się regularnie, zazwyczaj w rytm swoich iteracji, aby wyraźnie zastanowić się nad najważniejszymi wydarzeniami, jakie miały miejsce od poprzedniego spotkania i podjąć decyzje mające na celu naprawę lub ulepszenie.

Przyrostowy rozwój: środki, za pomocą których każda kolejna wersja produktu jest budowana na poprzedniej wersji poprzez dodanie funkcji widocznych dla użytkownika.

Informacje Radiatory: Termin ogólny używany do każdego z kilku odręcznych, wyciągniętych, drukowanych lub elektronicznych wyświetlaczy, które zespół umieszcza w bardzo widocznym miejscu, tak aby wszyscy członkowie zespołu, a także przechodnie, mogli zobaczyć najnowsze informacje w skrócie: liczba zautomatyzowane testy, prędkość, raporty o incydentach, stały status integracji i tak dalej.

Integracja: Odnosi się do wszelkich wciąż wymaganych wysiłków, po indywidualnych programistach lub podgrupach programistów pracujących nad oddzielnymi komponentami, dla zespołu projektowego, który dostarczy produkt odpowiedni do wydania jako funkcjonalna całość.

Invest: Akronimem jest ocena jakości historii użytkownika. To oznacza Independent (wszystkich innych), Negotiable (nie jest to konkretna umowa na funkcje), Valuable (lub vertical), Estimable (do dobrego przybliżenia), Small (w celu dopasowania iteracji) i Testable.

Iteracja: krótki, 2-4-tygodniowy cykl rozwojowy i testowy, podczas którego zespół opracuje działające, przetestowane oprogramowanie dla wybranych historii użytkowników i zaprezentuje działające oprogramowanie zainteresowanym stronom.

Backlit iteracyjny: krótka lista historii użytkowników, które zostały wybrane z zaległości produktu dla konkretnej iteracji.

Kanban: Kanban, wymawiane / 'kan'ban /, jest metodą opracowywania produktów za pomocą nacisk na dostawę "dokładnie na czas" i optymalizację przepływu pracy w zespole. To podkreśla, że programiści pobierają pracę z kolejki, a proces, od zdefiniowania zadania do jego dostarczenia do klienta, jest wyświetlany dla uczestników.

Czas oczekiwania: "Czas oczekiwania" to termin zapożyczony z metody produkcji znanej jako Lean lub Toyota Production System, w której zdefiniowano czas, jaki upłynął między złożeniem zamówienia przez klienta a otrzymaniem zamówionego produktu. W tłumaczeniu na dziedzinę oprogramowania czas realizacji można opisać bardziej abstrakcyjnie, jako czas, jaki upłynął między identyfikacją wymagania a jego realizacją.

Lean: Lean Software to tłumaczenie Lean Manufacturing i Lean IT zasady i praktyki w dziedzinie tworzenia oprogramowania. Przystosowany z systemu produkcyjnego Toyoty i jest zbiorem technik i zasad zapewniających więcej wartości przy użyciu tych samych lub mniejszych zasobów, eliminując odpady w organizacjach i procesach biznesowych.

Retrospektywa Milestone: wydarzenie obejmujące kilka dni, w które zainwestowano, aby przeprowadzić szczegółową analizę istotnych wydarzeń projektu.

Mock Objects: Technika powszechnie stosowana w kontekście wytwarzania testu automatów . Składa się z instancji wersji testowej komponentu oprogramowania (zazwyczaj klasy), która zamiast normalnych zachowań zapewnia wyniki wcześniej obliczone, a często sprawdza także, czy jest wywoływany zgodnie z oczekiwaniami testowanych obiektów.

Kalendarz Niko-niko: Drużyna instaluje kalendarz jednego ze ścian pokoju. Format kalendarza umożliwia każdemu członkowi zespołu zapisanie, pod koniec każdego dnia roboczego, graficznej oceny ich nastroju w tym dniu. Może to być albo "emotikon" rysowane ręcznie, albo kolorowa naklejka, po prostym kodzie koloru, na przykład: niebieski na zły dzień, czerwony na neutralny, żółty na dobry dzień. Z biegiem czasu kalendarz niko-niko ujawnia wzorce zmiany nastrojów zespołu lub poszczególnych członków.

Programowanie w parę: zwinna technika programistyczna, w której dwóch programistów pracuje razem na jednej stacji roboczej. Jeden wpisuje kod, a drugi sprawdza każdą linię kodu, gdy jest wpisana. Pisząc na klawiaturze nazywa się kierowcą. Osoba przeglądająca kod nazywa się obserwatorem (lub nawigatorem). Dwaj programiści często zmieniają role.

Motywy: Kiedy projekt wymaga tego - na przykład, gdy doświadczenie użytkownika jest głównym czynnikiem w wynikach projektu - zespół opracowuje szczegółowe, syntetyczne biografie fikcyjnych użytkowników przyszłego produktu: są to tak zwane "personas".

Planowanie pokera: Ponadto, zwany poker Scrum, jest opartą na konsensusie techniką szacowania, najczęściej wykorzystywaną do oszacowania wysiłku lub względnej wielkości zadań w tworzeniu oprogramowania.

Punkty: Jednostka używana do oszacowania wysiłków potrzebnych do opracowania historii użytkowników.

Backlog produktu: działa jako repozytorium dla wymagań, które mają zostać wydane w pewnym momencie. Są to zazwyczaj wymagania wysokiego poziomu z wysokimi szacunkami dostarczonymi przez interesariuszy produktu. Wymagania są wymienione na zaległości w kolejności pierwszeństwa i utrzymywane przez właściciela produktu.

Właściciel produktu: Właściciel produktu reprezentuje głos klienta i jest odpowiedzialny za zapewnienie, że zespół zapewnia wartość firmie. Właściciel produktu zapisuje elementy związane z klientem (zazwyczaj historie użytkowników), nadaje im priorytety i dodaje je do zaległego produktu. Zespoły Scrum powinny mieć jednego właściciela produktu.

Project Chartering: podsumowanie kluczowych czynników sukcesu projektu, które można wyświetlić na jednej ścianie sali zespołu jako arkusz papieru o rozmiarze flipchart.

Sesja szybkiego projektowania: sesja, podczas której spotykają się dwaj lub więcej deweloperów, aby omówić wybory projektowe, które mogą mieć daleko idące konsekwencje.

Aktywność związana z wydaniem: Każda czynność, która jest niezbędna do wdrożenia niektórych funkcji w produkcji.

Refaktoryzacja: Refaktoryzacja polega na poprawie wewnętrznej struktury kodu źródłowego istniejącego programu, przy jednoczesnym zachowaniu jego zewnętrznego zachowania. (Czerwony: Zielony: Refaktor). Jest to technika TDD, która opisuje zapisywanie najpierw testu błędu (czerwonego), zapisanie kodu wystarczającego do przejścia testu (zielony), a następnie modyfikację kodu do czystszego, bardziej czytelnego stanu (Refactor).

Względna ocena: Względna ocena jest jednym z kilku wyraźnych smaków estymacji stosowanych w zespołach Agile i polega na szacowaniu zadań lub historii użytkowników, a nie osobno iw bezwzględnych jednostkach czasu, ale przez porównywanie lub grupowanie elementów o równoważnych trudnościach.

Retrospektywa: Spotkanie zespołu, które ma miejsce po zakończeniu każdej iteracji dotyczącej rozwoju, w celu przeanalizowania zdobytych doświadczeń i omówienia, w jaki sposób zespół może być skuteczniejszy w przyszłości. Opiera się on na zasadach stosowania nauki od poprzedniego sprintu do nadchodzącego sprintu.

Role-Feature-Session: szablon "ról-cecha-powód" jest jedną z najczęściej zalecanych pomocy (często przerasta się po etapie nowicjusza) dla zespołów i właścicieli produktów zaczynających pisać historie użytkowników

Zasady prostoty: zestaw kryteriów, w kolejności pierwszeństwa, zaproponowany przez Kent Becka, aby ocenić czy jakiś kod źródłowy jest "wystarczająco prosty", że kod jest weryfikowany przez testy automatyczne, a wszystkie takie testy mijają, kod nie zawiera duplikacji, kod wyraża osobno każdy odrębny pomysł lub odpowiedzialność, kod składa się z minimalnej liczby składników (klasy, metody, linie) zgodne z pierwszymi trzema kryteriami.

Scrum: ramy, w których ludzie mogą rozwiązywać złożone problemy adaptacyjne, a jednocześnie produktywnie i twórczo dostarczać produkty o najwyższej możliwej wartości. Opiera się na adaptacyjnej i iteratywnej metodologii tworzenia oprogramowania.

Scrumban: Scrumban to mieszanka Scruma i Kanban, która podobno zawiera najlepsze cechy obu metod.

Scrum of Scrums: Technika skalowania Scruma do dużych grup (kilkanaście osób), polegająca na dzieleniu grup na zwinne zespoły 5-10. Każdy codzienny scrum w podgrupie kończy się wyznaczeniem jednego członka jako "ambasadora" do udziału w codziennym spotkaniu z ambasadorami z innych zespołów, zwanym Scrum ze Scrums.

Scrum Master: Scrum Master jest odpowiedzialny za usunięcie przeszkód w osiągnięciu przez zespół celów sprintu / rezultatów. Scrum Master nie jest liderem zespołu, ale działa jako bufor pomiędzy zespołem i wszelkie rozpraszające wpływy. Scrum Master zapewnia, że proces Scruma jest używany zgodnie z przeznaczeniem. Scrum Master jest egzekwatorem zasad. Kluczową rolą w roli Scrum Master jest ochrona zespołu i skupienie się na zadaniach, które są w zasięgu ręki. Rola została również określona jako przywódca-sługa w celu wzmocnienia tych podwójnych perspektyw.

Show & Tell: demonstracja działającego oprogramowania, która pojawia się na końcu każdej iteracji. Celem programu Show and Tell jest uzyskanie cennych opinii od właściciela produktu i innych zainteresowanych stron.

Zapisz się do zadań: Członkowie zespołu programistycznego Agile zwykle wybierają zadania do wykonania, zamiast przypisywać je menedżerowi.

Prosty projekt: Zespół przyjmujący praktykę "prostego projektu" opiera swoją strategię projektowania oprogramowania na następujących zasadach: projektowanie jest ciągłym działaniem, które obejmuje refaktoryzację i heurystykę, takie jak YAGNI (You Are Not Go Need It It); jakość projektu jest oceniana w oparciu o zasady prostoty kodu; wszystkie elementy projektu, takie jak "wzorce projektowe", architektury oparte na pluginach itp. są postrzegane jako mające zarówno koszty, jak i korzyści, a koszty projektowania muszą być uzasadnione; decyzje projektowe należy odłożyć do "ostatniej odpowiedzialnej chwili", tak aby zebrać jak najwięcej informacji o korzyściach wybranej opcji przed poniesieniem kosztów.

Wymiarowanie: proces powiązania wysiłku jednej rzeczy z drugą. Będzie mieć ten sam rozmiar, większy lub mniejszy. Wymiarujemy zamiast szacować, ponieważ oszacowanie jest oparte na czasie i trudne do zrobienia, a dobór jest oparty na wysiłku i łatwiej zrobić dobrze.

Spike: Krótka, czasowa część badań, zazwyczaj techniczna, w jednym opowiadaniu, który ma dostarczyć wystarczającej ilości informacji, aby zespół mógł oszacować rozmiar historii.

Sprint / Iteration: ustalony okres czasu, w którym historie użytkowników są wybierane do działania . Termin Sprint pochodzi od metodologii Scrum i jest analogiczny do terminu

Iteracja. Sprint definiowany jest jako 2-4-tygodniowy przyrost aktywności programistycznych, który zapewnia działające oprogramowanie i koniec inkrementacji. Zewnętrzne wpływy nie mogą zmieniać wymagań fabułów, nad którymi trwają prace.

Spring Backlog: Na początku każdego sprintu zespół ma planowanie sprintu, którego wynikiem końcowym jest zaległość pracy, którą zespół spodziewa się zakończyć po

zakończeniu sprintu. Są to przedmioty, które drużyna zapewni przeciwnikowi przez cały czas sprintu.

Planowanie sprintu: Czy spotkanie planowania przed sprintem z udziałem głównego zespołu agile. Podczas spotkania Właściciel Produktu opisuje zespołom najwyższy priorytet, zgodnie z opisem na zaległości produktu. Zespół następnie zgadza się na liczbę funkcji, które mogą wykonać w sprintu i planuje zadania wymagane do osiągnięcia dostarczenia tych funkcji. Grupa planowania obsługuje te funkcje w Historiach użytkowników i przypisuje kryteria akceptacji do każdej historii.

Sprint Review: Po każdym sprincie następuje przegląd Sprintu. Podczas tego przeglądu sprawdzane jest oprogramowanie opracowane w poprzednim Sprincie, aw razie potrzeby dodawane są nowe pozycje zaległości.

Story Card: miejsce na historię użytkownika. Karty historii to zazwyczaj 3 × 5 lub 5 × 7 kart lub kleistych kartek, które są umieszczane na ścianie, aby można je było łatwo śledzić.

Mapowanie historii: Mapowanie historii polega na porządkowaniu historii użytkowników według dwóch niezależnych wymiarów. "Mapa" organizuje czynności użytkownika wzdłuż osi poziomej w kolejności od priorytetu (lub "kolejność, w której opisano działania w celu wyjaśnienia zachowania systemu"). W dół osi pionowej oznacza rosnącą złożoność implementacji.

Story Point: jednostka miary złożoności i wielkości pomiaru.

Podział historii: Dzielenie polega na podzieleniu jednej historii użytkownika na mniejsze, przy jednoczesnym zachowaniu właściwości, że każda opowieść użytkownika ma osobną wartość biznesową.

Zrównoważone tempo: zespół dąży do osiągnięcia tempa pracy, które będą w stanie utrzymać w sposób nieokreślony.

Zadanie: Historię użytkownika można podzielić na jedno lub więcej zadań. Zadania są szacowane codziennie w godzinach (lub punktach fabularnych) pozostałych przez programistę pracującego nad nimi.

Taskboard / Storyboard: Tablica ścienna z kartami i karteczkami samoprzylepnymi, przedstawiająca całą pracę w każdym sprincie. Notatki są przesuwane po planszy, aby pokazać postęp.

Zespół: Drużyna jest odpowiedzialna za dostarczenie produktu. Zespół składa się zazwyczaj z 5-9 osób o umiejętnościach interdyscyplinarnych, wykonujących rzeczywistą pracę (analizowanie, projektowanie, opracowywanie, testowanie, komunikacja techniczna, dokument itp.). Zaleca się, aby zespół sam się umawiał i kierował, ale często pracuje z jakąś formą zarządzania projektem lub zespołem.

Team Room: Dedykowany zestaw przestrzeni dla zespołu Agile na czas trwania projektu, oddzielony od działań innych grup.

Trzy C: "Karta, rozmowa, bierzmowanie"; formuła, która przechwytuje komponenty historii użytkownika.

Karta techniczna: karta, która sama w sobie nie zapewnia wartości biznesowej. Zwykle jest to wykorzystywane do realizacji niektórych funkcji technicznych lub redukcji zadłużenia technicznego, które pomogą dostarczyć przyszłe karty historii.

Test Driven Development: rozwój oparty na testach (TDD) to program, proces polegający na powtórzeniu bardzo krótkiego cyklu rozwojowego: najpierw programista pisze wadliwy automatyczny przypadek testowy, który definiuje pożądaną poprawę lub nową funkcję, następnie tworzy kod do przejścia tego testu i ostatecznie refaktoryzuje nowy kod do akceptowalnych standardów.

Trzej Amigos: Trzej Amigos odnoszą się do podstawowych perspektyw, aby zbadać wzrost pracy przed, w trakcie i po rozwoju. Te perspektywy to: Biznes - Jaki problem próbujemy rozwiązać? Rozwój - Jak możemy zbudować rozwiązanie, aby rozwiązać ten problem? Testowanie - co z tego, co może się stać?

Trzy pytania: Codzienne spotkanie opiera się na kilku wariantach następujących trzech pytań: Co ukończyłeś od ostatniego spotkania? Co planujesz ukończyć do następnego spotkania? Co wchodzi ci w drogę?

Timeboxing: Timeboxing to technika planowania wspólna w projektach planistycznych (zazwyczaj w przypadku tworzenia oprogramowania), w której harmonogram jest podzielony na kilka oddzielnych okresów (przedziały czasowe, zwykle od dwóch do sześciu tygodni), przy czym każda część ma swoje własne wyniki, termin i budżet .

Wszechobecny język: podejście projektowe opisane w "Domain Driven Design" Eric Evansa (2003), polegające w szczególności na wykorzystaniu słownictwa danej domeny biznesowej, nie tylko w dyskusjach na temat wymagań dotyczących oprogramowania, ale w dyskusjach. projektowania, a także do samego "kodu źródłowego produktu".

Testowanie jednostkowe: Test jednostkowy, rozumiany przez zespoły zwinne, jest krótkim fragmentem programu napisanym i obsługiwanym przez programistów w zespole produktu, który wykonuje wąską część kodu źródłowego produktu i sprawdza wyniki. Wynik testu jednostkowego jest binarny: albo "przekazuj", jeśli zachowanie programu jest zgodne z zarejestrowanymi oczekiwaniami, albo inaczej "zawieść".

Testy użyteczności: Testy użyteczności to długo działająca, empiryczna i odkrywcza technika odpowiedzi na pytania typu "w jaki sposób użytkownik końcowy reaguje na nasze oprogramowanie w realistycznych warunkach?" Obejmuje to obserwowanie reprezentatywnego użytkownika końcowego wchodzącego w interakcję z produktem, biorąc pod uwagę celem jest osiągnąć, ale nie ma konkretnych instrukcji dotyczących korzystania z produktu.

Persona użytkownika: Personas to opis typowych użytkowników danego oprogramowania.

Historia użytkownika: historia użytkownika to definicja bardzo wysokiego poziomu wymagań, zawierająca tylko tyle informacji, aby programiści mogli oszacować rozsądnie wysiłek

wdrożenia. Historia użytkownika to jedno lub więcej zdań w języku codziennym lub biznesowym użytkownika końcowego, które przechwytuje to, co użytkownik chce osiągnąć. Historia użytkownika jest również miejscem, w którym można rozmawiać między użytkownikami i zespołem. Historie użytkowników powinny być pisane przez klientów lub dla nich w ramach projektu oprogramowania i są ich głównym instrumentem wpływającym na rozwój oprogramowania. Historie użytkowników mogą być również pisane przez programistów w celu wyrażenia niefunkcjonalnych wymagań (bezpieczeństwo, wydajność, jakość itp.)

Velocity: Jest to względna liczba, która opisuje ile pracy zespół może wykonać w danym okresie.

Pionowy plaster: pokazanie funkcji w aplikacji działającej od początku do końca, ale może mieć ograniczony zakres. Na przykład most linowy przekraczający przepaść jest natychmiast użyteczny i pozwala ludziom przejść. Posiadanie tego na miejscu może później pomóc w zbudowaniu lepszego mostu.

WIP: Również znana jako Praca w toku, jest to każda praca, która została rozpoczęta, ale jeszcze nie została ukończona.

XP: Metodologia tworzenia oprogramowania mająca na celu poprawę jakości oprogramowania i reagowanie na zmieniające się wymagania klientów. Jako rodzaj zwinnego rozwoju oprogramowania, zaleca częste "wydania" w krótkich cyklach rozwoju (timeboxing), które mają na celu poprawę wydajności i wprowadzenie punktów kontrolnych, w których mogą zostać przyjęte nowe wymagania klienta.