**Programowanie ekstremalne**

Pierwszy projekt programowania ekstremalnego rozpoczęto 6 marca 1996. Exstreme Programming jest jednym z kilku popularnych procesów Agile. To programowanie okazało się bardzo skuteczne w wielu firmach wszystkich branż różnej wielkości na całym świecie. Programowanie ekstremalne okazało się sukcesem, ponieważ podkreśla zadowolenie klientów. Zamiast dostarczać wszystkiego, czego można chcieć od jakieś daty daleko w przyszłości proces ten dostarcza oprogramowanie, na które jest zapotrzebowanie. XP pozwala programistom szybko reagować na zmieniające się wymagania klientów. Extreme Programming podkreśla pracę zespołową. Menedżerowie, klienci i deweloperzy są równymi partnerami w ramach współpracy zespołu. Programowanie ekstremalne realizuje proste, ale skuteczne środowisko pozwalające zespołom bardzo wydajnie pracować. Zespół zapoznaje się z problemem i zastanawia się jak skutecznie go rozwiązać. Programowanie ekstremalne poprawia projekt oprogramowania w pięciu podstawowych pojęciach : komunikacja, prostota, opinia, szacunek i odwaga. Ekstremalni programiści ciągle komunikują się z klientami i innymi programistami. Trzymają się konstrukcji prostej i czystej. Dostają informacje zwrotne poprzez testowanie ich począwszy od pierwszego dnia. Programowanie to zapewnia system dla klientów szybko i wprowadzane zmiany są zgodne z sugestią. Każdy mały sukces pogłębia poszanowanie unikalnych wkładów każdego członka zespołu. Ekstremalni programiści są w stanie odważnie reagować na zmieniające się wymagania technologii. Najbardziej zaskakujący aspekt Extreme Programming są jego proste zasady. Programowanie to można porównać do puzzli. Istnieje wiele małych kawałków. Indywidualne kawałki nie mają żadnego sensu. Jednak w połączeniu ze sobą tworzą sensowną całość. Zasady te mogą wydawać się niewygodne, nawet naiwne, ale są oparte na solidnych wartościach i zasadach. Warto sobie uświadomić, że te reguły określają środowisko, które sprzyja współpracy zespołowej i inicjacji, która jest celem. Osiągnięcie wydajnej pracy zespołowej będzie możliwe nawet wtedy gdy ulegną zmianie specyficzne potrzeby firmy. Schemat ten pokazuje jakie są zasady współpracy ekstremalnego programowania. Klienci cieszą się partnerami w procesie oprogramowania. Programiści aktywnie przyczyniają się bez względu na poziom doświadczenia i menedżerowie koncentrują się na komunikacji i relacji. Bezproduktywne działania zostały przycięte do zmniejszenia kosztów i frustracji wszystkich zainteresowanych.

**Definicja Extreme Programming:**

**Programowanie ekstremalne** – z języka angielskiego Extreme Programming, skrót XP, to paradygmat i metodyka programowania mająca na celu wydajne tworzenie małych i średnich „projektów wysokiego ryzyka”, czyli takich, w których nie wiadomo do końca, co tak naprawdę robi i jak to prawidłowo zrobić. Przyświeca temu koncepcja prowadzenia projektu informatycznego, wywodząca się z obserwacji innych projektów, które odniosły sukces.

Podstawą ekstremalnego programowania jest synergia wynikająca ze stosowania rozmaitych praktyk, które same w sobie mają wiele zalet, lecz mogą być trudne w zastosowaniu. Łączne użycie tych praktyk ma zapewnić wyeliminowanie niedogodności każdej z nich. Podstawowe założenia zostały sformułowane przez Kenta Becka. Stroną, która służyła do wymiany poglądów na temat programowania ekstremalnego było pierwsze na świecie Wiki Portland Pattern Repository założone przez Becka i Cunnighama.

**Zasady i praktyki programowania ekstremalnego.**

**Implementacja.**

**Odbiorca jest zawsze dostępny.**

Jednym z niewielu wymagań programowania ekstremalnego jest stała dostępność do klienta. Nie tylko, aby pomagać zespołowi wykonawców, ale by stać się jego częścią. Wszystkie fazy projektu opartego na metodologii XP wymagają komunikacji z odbiorcą, najlepiej twarzą w twarz, na miejscu. Najlepiej po prostu przypisać jednego lub kilku przyszłych użytkowników do zespołu. Warto zwrócić uwagę, aby były to osoby znające się na rzeczy, a nie praktykanci, których wyznaczył właściciel nie chcąc marnować czasu bardziej wykwalifikowanych pracowników. Potrzebni są eksperci, a nie aktorzy ich grający. Użytkownicy spisują opowiadania z pomocą developerów, by określić szacunkowy czas na ich wprowadzenie i ustalić ich priorytet.

Podczas spotkań związanych z planowaniem wersji klienci powinni uzgodnić, które z opowiadań mają być włączone do kolejnych wersji. Można także ustalić termin publikacji. Klienci podejmują te decyzje w oparciu o własne cele. Spotkanie te jest używane do określenia mniejszych, inkrementacyjnych publikacji, aby zapewnić jak najwcześniejsze dostarczenie pożądanej funkcjonalności do odbiorcy. Daje to użytkownikowi możliwość wcześniejszego wypróbowania systemu i szybszego odzewu developerów.

Ponieważ szczegóły są pomijane w opowiadaniach, developerzy będą musieli kontaktować się z odbiorcami w celu zebrania odpowiedniej liczby informacji, potrzebnych do opracowania zadań programistycznych. Projekty na większą skalę wymagają stałego zaangażowania ze strony klienta.

Użytkownik będzie także potrzebny do pomocy przy testach funkcjonalności. Trzeba będzie stworzyć dane testowe, a zamierzone wyniki oszacować i zweryfikować. Testy funkcjonalności sprawdzają czy system jest gotowy do publikacji i wdrażania. Może zdarzyć się, że system nie przejdzie wszystkich testów wymaganej funkcjonalności, wtedy klient będzie musiał przejrzeć ich wyniki i zdecydować, czy zgodzi się na jego odbiór cze też wstrzyma wydanie.

Może wyglądać, że klient będzie musiał poświęcić sporo swojego czasu, lecz należy pamiętać, że czas oszczędzany jest już na początku, nie wymagając szczegółowych specyfikacji, co do wymagań i zachowując go na później, nie dostarczając nieprzydatnego systemu.

Mogą wystąpić drobne problemy, gdy wielu użytkowników będzie dostępnych w różnym czasie. Eksperci z branży mają skłonności do posiadania sprzecznych poglądów. Jest to naturalne. Należy rozwiązać ten problem wymagając, żeby wszyscy klienci byli dostępni naraz podczas okazyjnych spotkań, aby móc wyłapać i pozbyć się różnic i nieścisłości wynikłych z ich opinii.

**Standardy kodowania.**

Kod musi być formatowany zgodnie z przyjętymi standardami. Standardy kodowania utrzymuje zapis spójnym i łatwym do odczytania oraz przebudowy dla całego zespołu. Projekty Smalltalk mogą użyć „Smalltalk – Best Practice Patterns” jako standardu kodowania.

**Najpierw arkusze testujące.**

Kiedy tworzone są testy najpierw, przed właściwym kodem, odkrywa się, że to co należało wykonać zaimplementowane zostanie znacznie szybciej i prościej. Wiele czasu oszczędza się łącząc implementację z testowaniem, które i tak należało by przeprowadzić później.

Tworzenie testów znacznie pomaga developerom uzmysłowić sobie, co tak naprawdę trzeba wykonać. Wymagania są stanowczo określone przez testy. Nie można podważać ich specyfikacji, gdy jest jasno określona za pomocą kodu.

Programista widzi postępy swojej pracy. Bez testów programista często może nie być pewny, czy to co wykonał jest wystarczające. Konsekwencją takich niedomówień mogą być zbędne dodatki i nowe błędy. Jeśli zostanie stworzony arkusz testujący dokładnie wiadomo, kiedy ukończono zadanie – gdy wszystkie testy wykonają się poprawnie.

Zyskuje także projekt systemu. Często spotkać można oprogramowanie, które bardzo trudno się testuje. Takie systemy przeważnie opierają się na wczesnej implementacji i późniejszym testowaniu, często przez inny zespół. Poprzez pierwsze tworzenie testów z góry wywierany jest nacisk na sprawdzenie wszystkiego, czego oczekuje klient. Odbija się to na projekcie, który sam staje się łatwiejszym do testowania.

Można wyróżnić pewną rytmikę przy tworzeniu arkuszy testujących. Najpierw tworzony jest jeden test by zdefiniować mały, ogólny aspekt problemu. Następnie pisze się najprostszy kod, który przejdzie ten test. Potem tworzy się drugi test. Później dodaje się do wcześniejszego kodu kolejne linie kodu, sprawiające zdanie drugiego testu. Nie należy pisać więcej dopóki nie będzie kolejnego testu. Te kroki kontynuuje się dopóki nie zostanie nic do sprawdzenia.

Tworzony kod powinien być prosty i zwięzły, implementując dokładnie tą funkcjonalność, która jest potrzebna. Pozostali developerzy będą mogli łatwo dowiedzieć się jak ją wykorzystać po prostu przeglądając arkusze testów. Jeśli pewien aspekt użycia nie został przewidziany to będzie od razu rzucał się w oczy.

**Programowanie w parach.**

Każdy kod dołączony do wydania jest tworzony przez dwie osoby pracujące razem na jednym komputerze. Programowanie w parach zwiększa jakość oprogramowania nie wpływając na czas jego dostarczenia. Jest to mało oczywiste, ale dwie osoby pracujące na jednym komputerze dodadzą tyle samo funkcjonalności, co dwie osoby pracujące oddzielnie, tyle, że będzie ona znacznie bardziej dopracowana. Ze zwiększoną jakością przychodzą późniejsze zyski.

Najlepszą metodą programowania w parach jest po prostu posadzenie obok siebie dwóch developerów przed monitorem, którzy będą przekazywać sobie myszkę i klawiaturę. Gdy jedna osoba pisze i myśli nad metodą, którą właśnie tworzy, druga w tej chwili myśli strategicznie o tym, jak ta metoda sprawdzi się w tej klasie. Potrzeba odrobiny czasu by przyzwyczaić się do takiego rozwiązania, więc nie można zrażać się, jeśli przy pierwszym podejściu poczuje się niezręczność.

**Sekwencyjna integracja.**

Bez kontrolowania integracji kodu źródłowego developerzy po lokalnym przetestowaniu swojego kodu będą kiedy tylko zechcą umieszczać wyniki swoich prac w publicznym repozytorium. Podczas równoległej integracji modułów kodu powstaje kombinacja, która nie została wcześniej przetestowana. Najprawdopodobniej zaistnieje wiele konfliktów, które ujawnią się później.

Pojawią się dalsze problemy, jeśli nie oczyszczono zależności ze starszej wersji. Odzwierciedlą się one na zestawie testów, który musi zweryfikować poprawki kodu. Jeśli nie można polegać na kompletnym, poprawnym i spójnym komplecie testów będą prześladować komunikaty o być może już nieistniejących błędach oraz nowopowstałych, które nie zostały w nim przewidziane.

W niektórych projektach developerzy opiekują się wydzielonymi klasami. Właściciele tych klas dbają o to, by ich kod był spójny i właściwie udostępniony. Zmniejsza to ryzyko, ale powiązania między klasami nadal mogą być niewłaściwe. Metoda ta nie rozwiązuje problemu w całości.

Inną drogą, prowadzącą do tego samego celu, jest wyznaczenie osoby-integratora lub całej grupy, zajmującej się integracją. Jeden człowiek może nie poradzić sobie z integracją kodu dostarczanego przez wielu pracowitych programistów. Zespół ten może być zbyt duży, by przeprowadzać integrację częściej niż raz w tygodniu. W takim środowisku developerzy pracują na przestarzałych wersjach, które byłyby wtedy błędnie, reintegrowane z głównym kodem.

Te rozwiązania nie odwołują się do źródła problemu. Chcąc by developerzy mogli równolegle, odważnie wprowadzać zmiany do każdej potrzebnej części systemu, ale pragnąc także uniknięcia logicznych konsekwencji takiej swobodnej integracji. Zamiast narzucać stricte sekwencyjny sposób programowania lub wymagać złożonych procedur integracyjnych należy przemyśleć jeszcze raz ten problem. Powinno się również w taki sposób przeprowadzać integrację kodu.

Ściśle sekwencyjna, samodzielna integracja kodu przez developerów połączona ze zbiorową własnością kodu będzie prostym rozwiązaniem problemu. Cały kod źródłowy jest publikowany do repozytorium w turach. Tj. tylko jedna para developerów integruje, testuje i wprowadza zmiany w repozytorium kodu w danej chwili. Kolejkowana integracja pozwala na jednoznaczne zidentyfikowanie ostatniej wersji systemu

Nie znaczy to, że nie można integrować poprawek z ostatnią wersją na własnej stacji roboczej, kiedy tylko programista zechce. Po prostu nie można publikować zmian do repozytorium dla zespołu, jeśli nie przyszła odpowiednia kolej.

Wymagany byłby pewien rodzaj mechanizmu blokującego. Najprostszy byłby fizyczny klucz przekazywany między developerami. Sprawdziłby się tutaj specjalnie przeznaczony do tego komputer, jeśli zespół pracuje w tym samym miejscu. Częsta integracja i publikacja kodu skraca czas potrzebny na trzymanie blokady, a zatem także czas oczekiwania na otrzymanie klucza.

**Dedykowany komputer publikujący.**

Pojedynczy komputer służący do kolejnych publikacji pracuje naprawdę dobrze, pod warunkiem, że zespół developerów jest usytuowany blisko siebie. Ten komputer odgrywa rolę fizycznego dowodu kontroli publikacji. Developerzy mają możliwość zobaczenia ostatniego słowa w bieżącej konfiguracji. Mają oni źródło ostatecznego arbitrażu w sprawie problemów integracyjnych. Komputer umożliwia projektantom przekonanie się, kto i kiedy publikuje. Kiedy publikujący komputer jest w trakcie pracy, nie można wprowadzać dodatkowych zmian. Należy zapewnić stabilność.

Najnowszy połączony arkusz ciągu testów może zostać uruchomiony jeszcze przed publikacją. Test suite jest zawsze aktualny, ponieważ używany jest tylko jeden komputer. Jeśli arkusze testujące wypełniają w 100% zadania, zmiany mogą zostać opublikowane. Jeśli nie - zmiany są debugowane lub cofnięte i debugowane na jednostkach komputerach developerów.

**Częsta integracja.**

Developerzy powinni integrować i publikować kod do repozytorium, co kilka godzin, kiedy to tylko możliwe. W żadnym razie, nigdy nie przetrzymywać zmienionej wersji dłużej niż dzień. Ciągła integracja często pozwala uniknąć rozbicia prac developerskich, kiedy to programiści nie porozumiewają się ze sobą na temat tego, co może zostać ponownie użyte albo, co powinno się dzielić. Wymaga się, aby każdy pracował na ostatniej, najnowszej wersji kodu. Wprowadzanie poprawek do i tak już nieaktualnego kodu źródłowego może tylko przyprawić o ból głowy w momencie integracji.

W miarę ciągła integracja pozwala na wczesne zapobieganie lub wykrywanie problemów z kompatybilnością. Chodzi o to, że jeśli znacznie łatwiej i bezpieczniej jest wprowadzać częściej drobne poprawki do głównego kodu niż za jednym zamachem, w chwili, gdy przychodzi czas oddania wyników, walczyć ze wprowadzeniem opracowywanych w ciągu kilku tygodni zmian do kodu, który sam już w tym czasie ewoluował. Dlatego zawsze należy pracować opierając się na aktualnej wersji systemu.

**Zbiorowa własność kodu.**

Zbiorowa własność kodu motywuje wszystkich do zasilania projektu nowymi pomysłami. Każdy programista może zmienić dowolną linijkę kodu dodając nową funkcjonalność, poprawiając błędy lub przebudowując istniejące rozwiązanie.

Może to być trudne do zrozumienia na początku. Prawie nie do pomyślenia jest by cała drużyna odpowiadała za architekturę systemu. Brak jednego dowódcy, który utrzymywałby nowe idee w ryzach może wydawać się założeniem, które się nie sprawdzi w praktyce.

Nie jest niezwykłą sytuacja, gdy na pytanie skierowane do głównego projektanta otrzymamy odpowiedź, która będzie po prostu niewłaściwa. Nie świadczy to o nie douczeniach czołowego developera. Każdy nieco mniej trywialny system może okazać się zbyt złożonym, aby nad całą jego strukturą zapanowała jedna osoba. Z kolei inni wykonawcy mogą mieć trudności z rozwijaniem systemu nie mając przedstawionego zarysu wizji projektanta. Czy zdając sobie sprawę z tego czy nie, architektura systemu jest już rozdzielona w obrębie drużyny. Jeśli natomiast cała ekipa jest już w pewnym stopniu odpowiedzialna za decyzje jej dotyczące to czy nie może również stanowić autorytetu w tym zakresie?

Drogą do realizacji tego założenia jest tworzenie przez każdego z developerów arkuszy testujących kod, w miarę jego pisania. Każdy kod publikowany do repozytorium kodu źródłowego zawiera arkusze testujące. Kod, który został dodany, poprawione błędy i starszy kod będzie objęty przez automatyczne testowanie. Teraz można polegać na zestawie testów, strzegących repozytorium kodu. Przed opublikowaniem jakiegokolwiek kodu musi on przejść cały zestaw testów w 100%.

Kiedy już zapewniono miejsce do działania, każdy będzie mógł wprowadzić zmiany w dowolnej metodzie którejkolwiek z klas i opublikować je do repozytorium gdy zajdzie taka potrzeba. Z częstą integracją developerzy nawet rzadko będą zauważać, że klasa została rozszerzona czy poprawiona.

W praktyce zbiorowa własność kodu właściwie lepiej się sprawdza niż umieszczanie pojedynczej osoby, mającej strzec odpowiednich klas. Zwłaszcza, że taka osoba może w każdej chwili opuścić projekt.

**Optymalizacja na końcu.**

Nie należy optymalizować czegoś, czego jeszcze nie ukończono. Nigdy nie należy próbować zgadywać, co może być wąskim gardłem systemu.

**Bez nadgodzin.**

Praca po godzinach wysysa ducha i motywację zespołu. Projekty wymagające nadgodzin, aby zostały ukończone na czas i tak się spóźnią, niezależnie od tego, co zostanie zrobione. Zamiast tego ważne jest wykorzystanie spotkania poświęconego planowaniu publikacji do zmian zakresu prac lub ich czasu. Zwiększanie zasobów poprzez przydzielanie większej ilości ludzi także jest nienajlepszym pomysłem, gdy projekt jest opóźniony.

**Planowanie.**

**Opowiadania użytkowników.**

Opowiadania użytkowników mają takie same zastosowanie, jak schematy tzw. przypadków użycia, ale znacznie się różnią. Są używane m.in. aby określić szacunkowy czas, jaki należy poświęcić na spotkanie w celu opracowania planu nowej wersji. Są także używane zamiast dużego dokumentu zawierającego wszystkie spisane wymagania. Opowiadania te są pisane przez klientów po prostu jako spis rzeczy, które nasz system powinien robić dla nich. Są podobne do scenariuszy użycia, ale nie ograniczają się do opisywania interfejsu użytkownika. Są przeważnie w formacie trój zdaniowych akapitów tekstu napisanych przez klienta, używających jego słownictwa, bez technicznej terminologii.

Opowiadania te sterują także tworzeniem testów akceptacyjnych. Testy te mają za zadanie stwierdzić, czy oczekiwania użytkowników zostały poprawnie zaimplementowane.

Jednym z największych powodów nieporozumień są różnice pomiędzy opowiadaniami, a specyfikacji wymagań. Główną cechą odróżniającą te dwa podejścia jest poziom szczegółowości opisu. Historie użytkowników powinny zapewniać tylko taką ilość szczegółów, jaka jest wystarczająca do rozsądnego oszacowania czasu potrzebnego na ich implementację. Gdy nadejdzie czas implementacji danego opowiadania wykonawcy po prostu skontaktują się z klientem i otrzymają dokładniejszy opis wymagań.

Projektanci ocenią jak długo będzie trwała implementacja oczekiwań. Każde opowiadanie zabierze 1, 2 lub 3 tygodnie pracy, oczywiście, jeśli wszystko pójdzie idealnie. Ten wzorcowy czas wykonania mówi jak długo zajęłoby jego wprowadzenie, gdyby nie było żadnych przerw, żadnych innych zadań i byłoby dokładnie wiadome, co jest do zrobienia. Jeśli czas może być dłuższy niż 3 tygodnie, to wymagane jest podzielenie naszej historii na mniejsze fragmenty. Jeśli jest krótszy niż 1 tydzień i wiadomo, że sprawa jest drobiazgowa, należy połączyć kilka streszczeń. Około 80 opowiadań jest doskonałą ilością do opracowania planu wersji.

Inną cechą różniącą opowiadania i specyfikację wymagań jest akcent na potrzeby użytkownika. Trzeba unikać szczegółów specyficznych technologii, struktury bazy danych i algorytmów. Należy próbować utrzymać historie skoncentrowane wokół potrzeb i korzyści klienta, nie wdając się na tym etapie w szczegóły samego interfejsu aplikacji.

**Planowanie publikacji.**

Zwoływane okresowe spotkania w celu opracowania planu publikacji. Służą one ustaleniu kierunku rozwoju projektu - plan ten jest następnie używany przy określaniu kolejnych iteracji. Zebraniem kieruje zbiór reguł, zezwalający wszystkim ludziom zaangażowanym w tworzenie projektu na podejmowanie decyzji w zależności od ich kompetencji. Zasady te definiują metody negocjacji planu i jego zatwierdzenia.

Opowiadania użytkowników i oblane testy są przekładane na zadania programistyczne, które je zapewnią. Zadania są zapisane na kartach. Karty te określają szczegółowy plan iteracji.

Podstawowym punktem spotkania jest ocenienie każdego opowiadania i przypisanie mu liczby, określającej szacunkową ilość tygodni potrzebnych do jego realizacji. Liczba ta nie uwzględnia żadnych innych obowiązków i wynikłych z nich opóźnień, zakładając idealną sytuację, gdy można poświęcić cały dostępny czas na jego realizację. Nie należy brać pod uwagę żadnej dodatkowej pracy, ale wliczyć testowanie. Odbiorca decyduje, które opowiadania są ważniejsze lub mają większy priorytet wykonania.

Historie użytkowników są drukowane lub spisywane na kartach. Developerzy i klienci rozmieszczają karty na stole wybierając zestaw opowiadań, które mają być zaimplementowane jako pierwsze, w tej, a które dopiero w kolejnej wersji. Użyteczny system do testowania, dostarczający wcześnie nowe wersje aplikacji byłby pożądany.

Można planować określając czas lub zakres realizacji. Za pomocą szybkości projektowania można wyznaczyć ile opowiadań może zostać zaimplementowanych w ciągu zadanego czasu lub jak długo zajmie ukończenie prac nad danym zbiorem opowiadań. Jeśli planowanie zawiera określany czas, mnożona jest liczba iteracji przez prędkość projektowania obliczając ilość opowiadań zdolnych do realizacji. Jeśli planowanie określa zakres opowiadań, należy podzielić sumę tygodni oszacowaną na ich podstawie przez prędkość projektowania. Uzyskana będzie wtedy liczba iteracji potrzebnych do ukończenia wersji.

Poszczególne iteracje są planowane dokładnie tuż przed ich rozpoczęciem, nigdy nie można ustalić ich przebiegu z góry. Spotkania prowadzone w celu opracowania planu wersji, zwane także grą mają swoje reguły, których opis można znaleźć w [repozytorium wzorców Portland](http://c2.com/cgi/wiki?PlanningGame).

Kiedy plan publikacji zostanie już ukończony, ale nie spodoba się kierownictwu kusząca staje się myśl o zmianie czasów realizacji opowiadań. Nie wolno tak robić! Opowiadania zostały dobrze ocenione i wymaga się, aby w takim stanie zostały uwzględnione na spotkaniu związanym z planowaniem iteracji. Zamiast tego trzeba opracować nowy plan publikacji, akceptowalny przez wszystkie strony. Negocjacje prowadzi się dotąd aż zarówno developerzy, klienci jak i kierownictwo zgodzą się na jego przebieg.

Główną filozofią planowania publikacji jest opisywanie projektu czterema wielkościami: zakresem, zasobami, czasem i jakością. Zakres określa jak dużo jest do zrobienia. Zasoby oznaczają ile jest dostępnych ludzi. Czas mówi kiedy projekt lub wersja będzie ukończona. Jakość natomiast opisuje jak dobre będzie oprogramowanie i jak dokładnie zostanie ono przetestowane.

Kierownictwo może określić tylko trzy z nich. Czwartą dostajemy już zawsze podczas samej realizacji. Warto zauważyć, że obniżenie jakości poniżej doskonałości ma nieprzewidywalny wpływ na trzy pozostałe cechy. W istocie więc tylko 3 parametry, które aktualnie można zmienić. Niech developerzy zapanują także nad pragnieniem klientów chcących, aby projekt został natychmiast ukończony poprzez zaangażowanie naraz zbyt wielu ludzi do pracy.

**Plan wydania.**

Po spisaniu opowiadań użytkowników można zwoływać spotkania w celu stworzenia planu publikacji. Plan wersji wyznacza dokładnie, które historie mają zostać wprowadzone w danej wersji systemu i kiedy można się tego spodziewać. Daje użytkownikom zestaw opowiadań do wyboru podczas planowania iteracji, które z nich zostaną zaimplementowane podczas najbliższej z nich. Wybrane historie są przekładane na odrębne zadania programistyczne do zaimplementowania w czasie iteracji.

Opowiadania są także przekładane na testy akceptacyjne podczas iteracji. Testy te są uruchamiane w czasie bieżącej i późniejszych iteracji w celu weryfikacji, kiedy historie zostały ukończone prawidłowo i można kontynuować dalszą pracę.

Gdy tempo projektowania drastycznie spadnie w przeciągu dwóch lub kilku iteracji wskazane jest wyprzedzenie kolejnego zdarzenia i wyznaczenie spotkania z klientami na którym ustalony zostanie nowy plan wersji.

Plan publikacji był wcześniej nazywany planem zobowiązań. Nazwa ta została zmieniona na taką, która dokładniej opisuje jego przeznaczenie i w bardziej logiczny sposób nawiązuje do planu iteracji.

**Regularna publikacja nowej wersji.**

Twórcy projektu muszą regularnie wypuszczać nowe wersje swojego systemu. Wynika to z iteracyjnego podejścia do projektowania. Częste spotkania poświęcone planowaniu nowych wersji służą ujawnieniu drobnych elementów funkcjonalności, które mogą być w najbliższym czasie zaimplementowane. Klient na bieżąco ma dostęp do aktualnej wersji środowiska i dzięki temu może dzielić się wartościowymi spostrzeżeniami z wykonawcami projektu. Im dłużej oczekuje się na wprowadzenie ważnego elementu do systemu, tym mniej czasu jest na jego dopracowanie.

**Szybkość projektowania.**

Prędkość projektowania jest miarą określającą jak szybko ukończony będzie projekt. Często używa się tutaj współczynnika obciążenia, ale prędkość projektowania wydaje się być znacznie prostszą w użyciu. Jeśli to pomoże, można użyć współczynnika obciążenia i na jego podstawie określić szacunkową prędkość projektowania.

Aby wyznaczyć tempo projektowania po prostu zlicza się opowiadania użytkowników lub ile zadań do zaimplementowania zebrano podczas bieżącej iteracji.

Podczas spotkania planującego wersję można użyć ilości opowiadań, które zostały już wprowadzone w życie do oszacowania ile z pozostałych historii zostanie zaimplementowanych w najbliższym czasie.

Podczas spotkania planującego iterację twórcy mają prawo przypisać odpowiednią liczbę dni do prac nad zadaniem, równą prędkości projektowania zmierzonej podczas poprzedniej iteracji.

Ten prosty mechanizm pozwala twórcom odnaleźć się po trudnej iteracji. Szybkość tworzenia projektu rośnie, gdyż pozwolono developerom na dopytywanie się klientów o kolejne historie, kiedy ich bieżąca praca zostanie wykonana wcześniej i nie ma żadnych innych zadań oczekujących na wykonanie.

Nie trzeba kłopotać się dzieląc prędkość projektowania przez długość iteracji czy liczbę programistów. Ta wartość w żadnym razie nie może służyć do porównania wydajności różnych projektów, ponieważ każdy zespół projektantów może wprowadzać różne odchylenia do szacunkowej sumy opowiadań i zadań, niekiedy zawyżają, a kiedy indziej mogą zaniżyć. Na dłuższą metę to nie ma znaczenia. Śledzenie ilości wykonanej pracy podczas każdej z iteracji jest kluczem do utrzymania tempa projektowania na tym samym poziomie.

Można spodziewać się kilku skoków prędkości projektowania, zarówno w górę jak i w dół. Ale jeśli prędkość ta dramatycznie zmieni się na czas dłuższy niż jedna iteracja, należy wykorzystać spotkania, związane z planowaniem nowej wersji, aby ponownie ocenić i przenegocjować plan ich tworzenia. Gdy ponownie rozpoczną się prace nad systemem, trzeba spodziewać się kolejnej zmiany prędkości, wynikającej z konieczności ponownego podjęcia się zadań.

**Współczynnik obciążenia.**

Współczynnik obciążenia jest równy rzeczywistym, kalendarzowym dniom niezbędnym do zrealizowania danego zadania, podzielonym przez, wyliczoną przez developerów, liczbę dni tzw. "idealnych". Aby zrozumieć to pojęcie, myśli się o zadaniu, którego rozwiązanie zajęłoby jeden dzień. A teraz wyobraża się staranie się to zrealizować w praktyce, w rzeczywistości. Współczynnik obciążeniowy określa liczbę dni, jakie rzeczywiście zajmie ukończenie tego zadania.

Współczynniki od 2 do 5 to norma. Jeśli odgaduje się, jakiego czynnika potrzeba, aby rozpocząć zadanie, powinno się rozważyć użycie odpowiedniej technologii i skorzystać z doświadczenia innych ludzi. Współczynnik 2 rokuje optymistycznie, 3 to technologia typowa. Czwórka i piątka natomiast używane są w projektach wymagających użycia technologii nietypowej. Ron Jeffries radzi używania trójki w początkowej fazie nowego projektu.

Następnie mierzy się i śledzi współczynnik obciążenia, a najlepiej jeszcze - tempo programowania w trakcie całego projektu.

Współczynnik obciążenia nie może być używany do porównywania dwóch projektów Każdy projekt, jak też zespół nad nim pracujący jest niepowtarzalny i będzie miał inne współczynniki obciążenia.

Jeśli współczynnik zmieni się w sposób dramatyczny, zwoływane są spotkania aby ponownie ocenić i przenegocjować plan publikacji. Należy się spodziewać, że kiedy system ruszy, współczynnik ponownie ulegnie zmianie z powodu zdań podtrzymujących.

**Iteracyjna rozbudowa.**

Rozbudowa projektu poprzez iteracje dodaje element zwinności procesowi tworzenia. Można podzielić swój plan zajęć na mniej więcej tuzin iteracji, każda o długości od 1 do 3 tygodni.

Nie planuje się swoich zadań z góry, na zapas. Zamiast tego na początku każdej iteracji stosuje się spotkania poświęcone jej planowaniu, aby określić co w danym kroku powinno być zrobione.

Jest to przeciwieństwo do reguły patrzenia naprzód i próby implementacji czegokolwiek, co nie jest zaplanowane na bieżącą iterację. Będzie mnóstwo czasu na implementację tej funkcjonalności, kiedy stanie się bardziej znaczącą sprawą, uwidocznioną w planie wersji.

Jeśli nigdy nie będzie wcześniej dodawana funkcjonalność i praktyka planowania na bieżąco, będzie łatwiej utrzymać się w czołówce zmieniających się wymagań użytkownika.

**Planowanie iteracji.**

Spotkanie poświęcone planowaniu jest zwoływane na początku każdej iteracji w celu opracowania jej planu, uwzględniającego zadania dla programistów. Każda iteracja trwa przez okres od 1 do 3 tygodni. Klient wybiera opowiadania z planu wersji, które powinny być zaimplementowane podczas bieżącej iteracji, w kolejności od najbardziej dla niego znaczących. Są także wybierane do załatwienia testy akceptacyjne, które zakończyły się niepowodzeniem.

Opowiadania użytkowników i oblane testy są przekładane na zadania programistyczne, które je zapewnią. Zadania są zapisane na kartach. Karty te określają szczegółowy plan iteracji.

Każde z zadań powinno wymagać od 1 do 3 "idealnych dni" do wykonania. Pisząc "idealny dzień" jest na myśli czas bez uwzględnienia niespodziewanych sytuacji, mogących przerwać pracę. Zadania, które trwają krócej niż 1 dzień, mogą zostać połączone w jedną grupę. Zadania trwające dłużej niż 3 dni powinny zostać ponownie przeanalizowane.

Developerzy podpisują się pod zadania i oceniają ile czasu im zajmie ich ukończenie. Ważny jest fakt, że osoba podejmująca się zadania sama określa jak długo zajmie jego wykonanie.

Wcześniej określona prędkość projektowania jest używana, aby określić czy iteracja jest już zamknięta czy jeszcze nie. Suma dni potrzebnych na realizację zadań nie może przekroczyć prędkości projektowania z poprzedniej iteracji. Jeśli bieżąca iteracja jest za duża, klient musi wybrać część opowiadań do odłożenia do czasu następnej.

Jeśli iteracja jest za mała można zaakceptować inne opowiadania. Prędkość w dniach i zadaniach ma większe znaczenie niż prędkość w tygodniach i opowiadaniach, gdyż jest bardziej dokładna.

Przerzucanie opowiadań użytkowników z jednej iteracji na drugą może być często odbierane jako alarmujące. Bez obawy. Teraz przypomni się znaczenie arkuszy testujących i refaktoryzacji. Długi i niedopełnienia na jednym z tych poziomów mogą znacznie spowolnić cały proces. Unika się dodawania jakiejkolwiek funkcjonalności dopóki nie zostanie ona zaplanowana. Po prostu dodawane jest to co potrzeba teraz. Dodawanie czegoś "ekstra" może tylko spowolnić pracę.

Niech nie kusi nikogo skrócenie czasu potrzebnego na wykonanie zadań i realizację opowiadań. Proces planowania polega na szacunkowych obliczeniach i nie uwzględnia wszystkich aspektów. Takie drobne zmiany mogą spowodować wiele problemów.

Obserwacja prędkości projektowania i przekładanie opowiadań. Może będzie potrzebne ponownie ocenienie wszystkich opowiadań i przenegocjowanie planu publikacji co każde 3 do 5 iteracji, to normalne. Jak długo będzie zawsze najpierw implementowane najbardziej wartościowe opowiadania, tak długo będzie robione najlepsze z możliwego dla swojego klienta i kierownictwa.

**Przenoszenie ludzi między zadaniami.**

Zmiana ludzi pracujących nad projektem, by uniknąć znaczącego odpływu wiedzy lub wystąpienia wąskiego gardła kodowania. Może zdarzyć się sytuacja, gdy jedyna osoba pracująca nad wyznaczonym zadaniem zdecyduje odejść z zespołu lub, gdy po prostu masz mnóstwo rzeczy czekających na zrealizowanie. Wówczas można stwierdzić, że projekt wlecze się w żółwim tempie.

Różne firmy często poważnie rozważają możliwość tzw. "szkoleń poprzecznych" by uchronić te obszary wiedzy, które łatwo jest utracić. Gdy zmieniani są ludzie pracujący w obrębie głównego kodu, a dodatkowo łączone jest to jeszcze z programowaniem w parach, to właśnie zastosowano metodę "szkoleń przemiennych". Sytuacja zmienia się z takiej, gdy jedna osoba wie wszystko o danej sekcji kodu, na taką, gdzie wszyscy znają większą część kodu w poszczególnych sekcjach.

Zespół jest o wiele bardziej "elastyczny”, jeśli wszyscy posiadają wystarczającą ilość informacji o każdej części systemu, który jest opracowywany. Zamiast posiadać małą grupkę osób przeciążonych pracą w trakcie, gdy reszta zespołu nie ma nic do roboty, można uczynić produktywnym cały zespół. Dowolna ilość developerów może zostać przypisania najbardziej pracochłonnym zadaniom. Płynne balansowanie takim przepływem jest spełnieniem marzeń menadżera.

Staranie się po prostu zachęcać, by każda osoba z zespołu spróbowała popracować nad nową częścią projektu, lub przynajmniej nad częścią danej iteracji. Taką możliwość daje, bez obawy utraty produktywności pracy, programowanie w parach. Metoda ta zapewnia ponadto ciągłość wymiany myśli i pomysłów. Jeśli zachodzi taka potrzeba, jedna osoba z pary może zostać "wymieniona" na kogoś innego i wówczas ta druga kontynuuje pracę z nowym partnerem.

**Codzienne "spotkania na stojąco".**

Podczas typowego spotkania dotyczącego projektu, większość uczęszczających na nie ludzi nie wnosi do niego niczego konkretnego, a przychodzi tylko by usłyszeć końcowe wnioski. Znaczna część czasu developera tracona jest więc na trywialne rozmowy. Gdy znaczna liczba osób uczęszcza na takie spotkania, niepotrzebnie marnotrawione są środki i energia potrzebna do tworzenia projektu a ponadto wyłania się koszmarna konieczność tworzenia planu.

Głównym założeniem "spotkań na stojąco" jest efektywna komunikacja pomiędzy członkami zespołu. Codzienne poranne spotkania, mają na celu mówienie o napotkanych problemach, możliwych rozwiązaniach a także motywowanie zespołu do koncentrowania się na sprawach najistotniejszych. Wszyscy zebrani stoją w kole, co pozwala uniknąć długich dyskusji i jest bardziej efektywne - lepiej zorganizować jedno krótkie ale obowiązkowe spotkanie, niż urządzać ich wiele, na których obecnych jest zaledwie kilku developerów.

Jeśli urządzane są takie spotkania, obecność na jakichkolwiek innych spotkaniach zależy od tego czy dana osoba jest tam naprawdę niezbędna i czy wnosi ona coś konkretnego do projektu. W takiej sytuacji możliwe jest nawet uniknięcie planowania większości spotkań. Mogą być one urządzane spontanicznie, przed komputerem, gdzie można na miejscu testować kod a nowe pomysły od razu wcielać w życie.

Takie spotkanie, to nie kolejne zebranie gdzie marnuje się czyjś czas. Zastąpi ono wiele innych, pozwalając ponadto na oszczędności kilka razy przewyższające czas jego trwania.

**Napraw XP, kiedy się zepsuje.**

Usprawnienie procesu, kiedy zawiedzie. Już teraz wiadomo, że projekt będzie wymagał pewnych zmian. Na dobry start ważne jest przestrzeganie zasad XP, ale nie wahanie się wprowadzając zmiany, jeśli coś nie funkcjonuje prawidłowo. Należy przestrzegać reguł do momentu, gdy zmieni je zespół. Nie znaczy to jednak, że może on robić cokolwiek zechce. Developerzy muszą dokładnie wiedzieć, czego mogą się spodziewać od siebie nawzajem. Trzymanie się ustalonego zestawu zasad to jedyny sposób, aby sprostać tym wymaganiom.

**Projektowanie.**

**Prostota stanowi klucz.**

Ukończenie prostego projektu zawsze zabiera mniej czasu niż złożonego. Dlatego zawsze wykorzystywany jest najprostszy sposób, który będzie działał zgodnie założeniami. Jeśli natrafi się coś skomplikowanego należy zamienić to na prostszy odpowiednik. Zawsze szybciej i taniej jest pozbyć się złożonego kodu teraz, niż później. W ten sposób zostanie zaoszczędzone wiele czasu zmarnowanego na upraszczanie już wykorzystywanego kodu.

**Wybór standardu nazewnictwa.**

To jak nazywane są obiekty jest bardzo znaczące dla zrozumienia całego projektu systemu, a także wielokrotnego używania kodu. Jeśli można domyślić się jak coś zostałoby nazwane, zaoszczędza dużo czasu na zgadywaniu. Należy wybrać odpowiedni system nazewnictwa dla obiektów, do którego wszyscy będą mogli się odnosić bez potrzeby poznania dokładnej wiedzy na temat działania projektu.

Np. system rozliczeń firmy Chrysler został zbudowany w oparciu o elementy linii produkcyjnej. U Forda natomiast samochody są sprzedawane jako struktury złożone z komponentów. Jest także nazewnictwo znane jako "naiwna", bazujące na własnej domenie. Ale nie można wybrać tej metody, jeśli nie jest wystarczająco prosta.

**Karty CRC.**

Największą wartością kart CRC jest pozwolenie ludziom na odrzucenie proceduralnego stylu tworzenia i docenienie w pełni możliwości technologii obiektowej. Karty CRC umożliwiają całemu zespołowi na współpracę podczas projektowania. Im więcej osób mogących pomóc zajmuje się projektowaniem systemu, tym więcej ciekawych pomysłów może zostać wprowadzonych w życie.

Do reprezentacji poszczególnych obiektów używa się oddzielnych kart CRC. Klasa obiektu może być napisana w górnej części karty, odpowiedzialność po lewej, a klasy, z którymi współpracuje po prawej stronie.

Jedną z większych wad systemu kart CRC jest brak opracowanego projektu na piśmie. Przeważnie da się bez niego obejść, ponieważ karty CRC sprawiają, że projekt jest przejrzysty. Jeśli niezbędne jest utrwalenie sesji należy wypełnić w całości po jednej karcie w każdej klasie zachowując je jako dokumentację.

**Utwórz rozwiązanie impulsowe.**

Rozwiązanie impulsowe to prostu program, którego zadaniem jest wyszukiwanie potencjalnych rozwiązań danego problemu. Większość tego typu programów nie jest wystarczająco dobra, by używać ich przez dłuższy czas, tak więc należy spodziewać się, że będzie trzeba się go pozbyć. Głównym założeniem jest zmniejszenie, na ile to możliwe ryzyka wystąpienia problemów typu technicznego, a także zwiększenie wiarygodności oceny opowieści użytkownika.

Kiedy trudności techniczne zagrażają rozwojowi systemu, niech dwoje developerów zajmie się tym przez tydzień lub dwa i spróbuje zmniejszyć potencjalne ryzyko wystąpienia problemu.

**Nie dodawaj funkcjonalności za wcześnie.**

Niech system nie będzie zaśmiecony dodatkowymi rzeczami. Tak naprawdę zaledwie 10% z nich zostanie kiedykolwiek użyte, a więc marnowane jest 90% czasu. Wszystkich kusi, aby dodać funkcjonalność raczej teraz niż później, ponieważ wiadomo dokładnie jak jej użyć. Poza tym to tak ulepszyłoby system... Wydaje się, że byłoby znacznie szybciej dodać ją właśnie teraz. Dodatkowa funkcjonalność jedynie spowolni naszą pracę i zmarnuje zasoby.

**Bez skrupułów przebudowuj istniejące rozwiązania.**

Ciągle używane są kody, z których tak naprawdę nie da się dłużej korzystać bezproblemowo i które wymagają nieustannego ulepszania. W przekonaniu jednak, nadal funkcjonują one w miarę dobrze, a nie chce się ich zmieniać. Extreme Programming próbuje przekonać, że nie. Kiedy usuwana jest nadmiarowość, eliminuje się nieużywaną funkcjonalność i odmładza się przestarzałe projekty, to właśnie wtedy przebudowywany jest system. Refaktoring stosowany w trakcie całego procesu projektowania to oszczędność czasu i poprawa jakości.

**Testowanie.**

**Arkusze testowe.**

Arkusze testów są jednym z kamieni węgielnych Extreme Programming. Jednakże testowanie w XP nieco różni się od zwykłego. Najpierw tworzone jest lub ściągane środowisko, które pozwoli je przeprowadzać w sposób automatyczny. Następne powinno się przetestować wszystkie klasy w systemie. Trywialne metody typu get/set są przeważnie pomijane. Później należy już praktykować tworzenie testów przed właściwym kodem.

Arkusze testów są publikowane do repozytorium razem z kodem, który testują. Kod nie pokryty testami nie może być opublikowany. Jeśli odkryjemy, że jakiegoś testu brakuje, trzeba natychmiast go opracować.

Największą trudnością w prowadzeniu arkuszy testów jest nieubłaganie zbliżający się deadline, termin oddania efektów pracy. W praktyce jednak automatyczne testy potrafią zaoszczędzić stokroć tyle czasu ile potrzeba przeznaczyć na ich stworzenie poprzez wynajdywanie błędów i zapobieganie przez nimi. Im trudniej jest opracować dany test tym bardziej go potrzeba, gdyż zaoszczędzi więcej czasu później. Koszty potrzebne na tworzenie testów zawsze zwracają się z nawiązką.

Innym popularnym błędnym przekonaniem jest, że testy mogą być pisanie w ciągu trzech ostatnich miesięcy projektowania. Niestety - bez arkuszy testów - prace developerskie przeciągają się, pochłaniając i te trzy miesiące, a może nawet więcej... Nawet, gdy znajdzie się trochę dostępnego czasu trzeba się liczyć z tym, że aby powstałe dobre testy potrzeba trochę czasu. One muszą dojrzeć. Odkrywanie potencjalnych problemów, które mogą wystąpić pochłania dalszy czas. Żeby testy były dostępne, kiedy będą potrzebne należy zacząć tworzyć je już teraz.

Arkusze testów wspierają zbiorową własność kodu. Kiedy tworzone są arkusze testów strzeżona jest funkcjonalność przed przypadkowymi uszkodzeniami. Wymagając, aby cały kod przeszedł wszystkie testy przed publikacją trzeba upewnić się, że cała funkcjonalność systemu jest sprawna. Wydzielanie odpowiedzialności za kod nie jest potrzebne, jeśli na straży klas stoją odpowiednie arkusze testów.

Arkusze testów umożliwiają również refaktoryzację. Po każdej drobnej zmianie testy mogą ją zweryfikować i upewnić się, że zmiana w strukturze rozwiązania nie wpłynie na niepożądaną zmianę funkcjonalności.

Zbudowanie pojedynczego, uniwersalnego zestawu arkuszy testów do walidacji i testowania regresyjnego pozwala na częstą integrację kodu. Można wtedy szybko zintegrować wszystkie ostatnie zmiany, po czym uruchomić najnowszą wersję zestawu testów. Jeśli kod nie przejdzie testu, oznacza to, że jest niekompatybilny z poprzednią wersją. Usuwanie małych problemów, w co kilka godzin zabiera mniej czasu niż naprawianie olbrzymich tuż przed wyznaczonym terminem. Wykorzystując automatyczne testowanie jest możliwe dołączenie zbioru poprawek do ostatnio wypuszczonej wersji i opublikowanie ich w krótkim czasie.

Często dodawanie nowych funkcji wymaga zmiany arkuszy testujących, aby te ogarnęły zmienioną funkcjonalność. Choć jest możliwe wprowadzenie błędu zarówno do kodu jak i do testu, w praktyce zdarza się to rzadko. Czasami test będzie błędny, ale kod okaże się poprawny. Ujawni się to dopiero, gdy problem będzie już wykryty i naprawiony. Tworzenie arkuszy testów niezależnie od kodu, najlepiej jeszcze przed nim, wyznacza ramy zadania i znacznie zwiększa szanse na jego dobrą implementację już za pierwszym razem.

Arkusze testujące zapewniają bezpieczną sieć testów regresyjnych i zatwierdzających, dzięki czemu można refaktoryzować i integrować kod bardzo efektywnie. Tworzenie arkuszy testów przed kodem ma jeszcze dodatkowe zalety: utrwala wymagania i pozwala na skupienie się developerów na konkretnym problemie.

**Schematy arkuszy testujących.**

Jednym z najczęściej spotykanych błędów jest rozumienie schematów arkuszy testujących dosłownie - jako narzędzi testujących. Są to narzędzia rozwojowe, podobnie jak edytor czy kompilator. Nie zalecane jest odkładajnie użycia tego potężnego narzędzia aż do ostatniego miesiąca projektu. Schematy te pomogą ustalić wymagania, uczynić architekturę bardziej przejrzystą, napisać, debugować i zintegrować kod, opublikować, zoptymalizować i, oczywiście przetestować.

**Gdy znajdziemy błąd.**

Po ujawnieniu się błędu tworzone są testy. Mają one ostrzegać przed jego ponownym jego wystąpieniem. Błąd w wykonaniu wymaga testu akceptacyjnego, który będzie sprawdzał nasz system pod jego kątem. Testy akceptacyjne tworzone jako pierwsze, jeszcze przed samym debugowaniem pozwalają użytkownikom na zwięzłe zdefiniowanie problemu i przedstawienie go programistom. Programiści mając wyszczególniony niespełniony test mogą wtedy skoncentrować swoje siły na rozwiązywaniu problemu i wiedzą kiedy efekt zostanie osiągnięty.

Otrzymawszy niezdane testy akceptacyjne, developerzy mogą utworzyć mniejsze arkusze testujące, aby móc opisać defekt za pomocą większej ilości kodu i ukazać błąd z różnych punktów widzenia, w różnym kontekście. Obserwując interesujące arkusze testów można na bieżąco kontrolować, kiedy błędy zostaną poprawione. Gdy wszystkie arkusze przechodzą testy pomyślnie można ponownie uruchomić odpowiednie testy akceptacyjne by upewnić się, że problem na pewno został zażegnany.

**Testy akceptacyjne.**

Testy akceptacyjne tworzone są na bazie opowieści użytkowników. Podczas iteracji opowieści użytkowników zostaną przetłumaczone na testy akceptacyjne. Kiedy opowieść zostanie prawidłowo zaimplementowana, użytkownik precyzuje scenariusz testu. Dana opowieść może mieć jeden lub więcej testów akceptacyjnych - tyle, aby mieć pewność, że funkcjonalność będzie działać prawidłowo.

Testy akceptacyjne to czarne skrzynki testów systemowych. Każdy z tych testów ma za zadanie podać pewien oczekiwany wynik systemowy. Użytkownicy odpowiedzialni są za sprawdzanie prawidłowego przebiegu testów, a także za przegląd ich wyników. Te informacje pomagają im zdecydować o tym, które z oblanych testów są najważniejsze. Testy akceptacyjne są także używane jako testy regresyjne stosowane jeszcze przed publikacją projektu.

Opowieść użytkownika nie jest uważana za kompletną, zanim nie przejdzie testów akceptacyjnych. Oznacza to, że dla każdej iteracji należy stworzyć nowy test akceptacyjny. W przeciwnym wypadku zespół developerów ogłosi zerowy postęp.

Gwarancja jakości, jest zasadniczą częścią programowania w XP. Przy niektórych projektach QA jest przeprowadzana przez członków specjalnie do tego powołanej grupy, podczas gdy przy innych - grupa ta jest po prostu częścią grupy developerskiej.

Testy akceptacyjne powinny być zautomatyzowane po to, by można je często uruchamiać. Wynik testu akceptacyjnego ogłaszany jest zespołowi, którego zadaniem jest takie zaplanowanie czasu każdej iteracji, by poprawić wszystkie z oblanych testów. Testy akceptacyjne poprzednio nazywane były testami funkcjonalnymi.

Nazwa ta została zmieniona na obecną, ponieważ lepiej oddaje ona cel testów, co daje gwarancję, że oczekiwania użytkowników zostaną w pełni zaspokojone a system we właściwy sposób spełnia swe zadania.