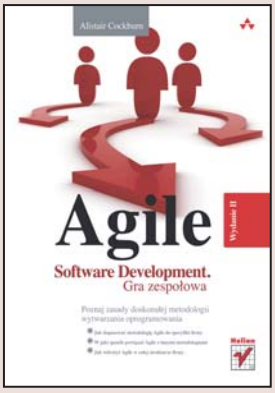
Agile Software Development. Gra zespołowa. Wydanie II



Autor: Alistair Cockburn
Tłumaczenie: Rafał Jońca
ISBN: 978-83-246-1503-2
Tytuł oryginału: Agile Software Development: The Cooperative Game (2nd Edition) (The Agile Software Development Series)
Format: 170x230, stron: 480

Poznaj zasady doskonałej metodologii wytwarzania oprogramowania

- Jak dopasować metodologię Agile do specyfiki firmy?
- W jaki sposób powiązać Agile z innymi metodologiami?
- Jak wdrożyć Agile w całej strukturze firmy?

Produkcja oprogramowania wymaga nie tylko doskonałej znajomości technologii, ale także metodologii zarządzania projektem. Kluczowym elementem jest tu umiejętność błyskawicznego reagowania na zmiany, sytuacje kryzysowe i błędy. Istnieje wiele usystematyzowanych metodologii wytwarzania oprogramowania, które jednak rzadko sprawdzają się w przypadku małych zespołów projektowych lub projektów realizowanych w krótkim czasie. Dla takich projektów opracowano metodologię Agile. To „zwinne programowanie" zdobywa coraz więcej zwolenników i jest wdrażane w wielu przedsiębiorstwach.

Książka „Agile Software Development: The Cooperative Game (2nd Edition) (The Agile Software Development Series)" to omówienie metodologii Agile i inżynierii oprogramowania. Czytając ją, poznasz założenia zwinnego programowania i sposoby zarządzania projektem, zgodne z wytycznymi tej metodologii. Dowiesz się, jakie ograniczenia posiada Agile i jak sobie z nimi radzić. Przeczytasz o programowaniu ekstremalnym, adaptacji tej metodologii do potrzeb konkretnych zadań i unikaniu błędów przy wytwarzaniu oprogramowania.

- Zasady inżynierii oprogramowania
- Dobór zespołu projektowego
- Komunikacja wewnątrz zespołu projektowego
- Wybór odpowiedniej metodologii
- Programowanie ekstremalne
- Zarządzanie zmianami
- Metodologie Crystal

Poznaj wydajne i efektywne zwinne programowanie!

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

SPIS TREŚCI

5

Zwinność i adaptacja

Mamy już wszystkie elementy układanki. Dowiedziałeś się, że:

- Tworzenie oprogramowania jest grą zespołową pomysłowości i komunikacji.
- Ludzie są dziwni, ale dobrzy w spoglądaniu wokoło, przejmowaniu inicjatywy i bezpośredniej komunikacji twarzą w twarz.
- Metodologia to zestaw konwencji stosowanych przez zespół, przy czym różne konwencje sprawdzają się w różnych rodzajach projektów.
- Lekkie metodologie pozwalają dostarczać oprogramowanie szybciej, ale wraz z powiększaniem się zespołu potrzeba cięższych metodologii.
- Projekty to unikatowe ekosystemy, więc trzeba tak dobierać metodologię, by pasowała do ekosystemu projektu.

Wszystko do siebie ładnie pasuje, ale… jaka lekkość jest odpowiednia dla danego projektu i jak *mamy* zastosować wspomniane zasady w *naszym* konkretnym projekcie**?**

Podrozdział „Lekko, aczkolwiek wystarczająco" wskazuje, jaka lekkość jest odpowiednia dla danego projektu, a w szczególności, co oznacza bycie zbyt lekkim. Naszym celem jest zrównoważenie lekkości i wystarczalności.

Podrozdział „Zwinność" omawia znaczenie pewnych „istotnych miejsc" projektu: kolokację, bliskość użytkowników, doświadczenie programistów itp. Gdy projekt oddala się od tych istotnych miejsc, trzeba w nim stosować mniej zwinne mechanizmy. W szczególności zespoły wirtualne znacząco utrudniają wprowadzenie w życie zasad zwinności, bo są od siebie mocno oddalone.

Podrozdział „Dostosowywanie się" opisuje technikę tworzenia lekkiej, aczkolwiek wystarczającej metodologii osobistej, która może przynieść korzyść projektowi. Kluczową sprawą okazuje się analiza co kilka tygodni, co poszło dobrze, a co należy zmienić.

Zwinność i adaptacja

Lekko, aczkolwiek wystarczająco

Przedstawiona do tej pory teoria wskazuje, by przede wszystkim stosować przekaz ustny do propagowania wszystkich informacji zebranych w trakcie realizacji projektu.

Nasza intuicja podpowiada nam, że przekaz ustny nie wystarcza.

Poszukiwanie dokumentacji

Pewien programista zaproponował swojej firmie ponowne napisanie jej flagowego produktu, ponieważ nie było do niego dokumentacji, żadna z pozostałych osób nie znała szczegółów powstania systemu i nie potrafiła wykonać niezbędnych modyfikacji. Powiedział również, że ma nadzieję, że po nowym projekcie pozostanie dokumentacja.

Ktoś inny opowiedział mi o trzech projektach, które miały bazować na jednym i tym samym oryginalnym projekcie. Wszystkie trzy miały być wytwarzane w rożnych miejscach. Stwierdził również, że w takiej sytuacji przekaz ustny po prostu nie ma prawa bytu.

Jest całkiem możliwe, że zdobyte informacje były za mało „lepkie". Warto jeszcze raz przyjrzeć się zasadzie gry zespołowej:

Głównym celem jest dostarczenie oprogramowania; celem drugorzędnym jest przygotowanie się do następnej rozgrywki.

Powody osiągnięcia pierwszego celu są oczywiste — jeśli nie dostarczymy oprogramowania, nie ma znaczenia, jak dobrze przygotowywaliśmy się do następnej gry.

Jeśli jednak dostarczymy oprogramowanie i słabo przygotujemy się do następnej rozgrywki, podcinamy sobie skrzydła.

Obydwa działania konkurują ze sobą. Zrównoważenie dwóch konkurujących działań wymaga dwóch umiejętności.

Pierwsza polega na prawidłowym zgadywaniu, w jaki sposób zaalokować zasoby dla każdego z celów. W idealnej sytuacji dokumentowanie należałoby odkładać na jak najpóźniejszy czas, a następnie tworzyć jak najmniejsze dokumenty. Zbyt rozbudowana dokumentacja wykonywana zbyt wcześnie opóźnia dostarczenie oprogramowania. Jeśli jednak zbyt małą dokumentację robimy zbyt późno, może się okazać, że odeszła osoba, która wiedziała coś istotnego dla następnego projektu.

Druga umiejętność polega na odgadnięciu, jak wiele można związać z tradycją ustną grupy, a ile należy umieścić w dokumentacji. Zauważ, że w pewnym momencie nie ma znaczenia, czy modele i inna dokumentacja są kompletne i czy idealnie odpowiadają rzeczywistemu systemowi lub czy są zgodne z aktualną wersją kodu. Ważne jest tylko, czy osoby, które będą czytały dokumentację, znajdą w niej potrzebne informacje.

Odpowiednia ilość dokumentacji to dokładnie tyle, ile potrzeba, by ktoś mógł wykonać następny ruch w grze. Każdy dodatkowy wysiłek włożony w dokumentowanie modeli ponad tę kwestię jest stratą pieniędzy.

Bardzo często osoby, które przesłuchiwałem w związku z udanym projektem, mówiły, że udało im się „**pomimo** oczywistych braków w dokumentacji i dziwacznego procesu" (to ich słowa, nie moje). Analizując te wypowiedzi w nowym świetle, nietrudno domyślić się, że odnieśli sukces dlatego, że dokonali dobrego wyboru i zaprzestali prac nad pewnymi kanałami komunikacji od razu po tym, jak osiągnęli wystarczający poziom zrozumienia. Wykonali odpowiednią pracę dokumentacyjną, ale jej nie doskonalili.

Adekwatność to doskonały warunek, jeśli zespół musi szybko gnać w kierunku głównego celu, a nie ma wielu zasobów.

Przypomnijmy programistę, który powiedział:

> „Jest dla mnie oczywiste, że gdy zaczynam tworzyć przypadki użycia, modele obiektów itp., moja praca ma jakieś znaczenie. Ale w pewnym momencie przestaje ona być użyteczna, stając się problemem i źródłem strat. Nie potrafię wykryć przekroczenia tego punktu i nigdy nie słyszałem dyskusji na ten temat. To bardzo denerwujące, ponieważ użyteczne działania zamieniają się w stratę czasu".

Poszukujemy punktu, w którym użyteczna praca staje się balastem. To druga z wymienionych umiejętności.

Ledwo wystarczające

Sądzę, że nie muszę dawać przykładów zbyt ciężkich lub zbyt lekkich metodologii. Większość programistów widziała lub słyszała wiele takich przykładów.

Z drugiej strony „jedynie trochę za lekkie" metodologie są trudne do znalezienia i wyjątkowo przydatne naukowo. To dzięki nim dowiadujemy się, co oznacza wyrażenie **ledwo wystarczający**.

We wcześniejszej części książki pojawiły się dwie tego rodzaju historie: „Ciągły brak dokumentacji" i „Przyklejanie myśli do ściany". W każdej z nich dobrze działający projekt w kluczowym momencie przechodził poniżej poziomu wystarczalności.

Ciągły brak dokumentacji (powtórka)

Ten zespół stosował wszystkie praktyki XP i dostarczał oprogramowanie klientowi w określonym czasie. Po kilku latach sponsor spowolnił prace projektowe, aż wreszcie je zatrzymał.

Gdy zespół przestał istnieć, po systemie nie pozostała żadna pisana dokumentacja, więc żadna z innych osób nie mogła poznać szczegółów jego budowy. Wystarczająca dawniej kultura werbalna przestała wystarczać.

W tej historii zespół osiągnął podstawowy cel gry, dostarczył działający system, ale nie udało mu się przygotować do następnej gry związanej z pielęgnacją i modyfikacją systemu.

Ktoś może wykorzystać moją własną logikę przeciwko mnie, argumentując, że ilość dokumentacji okazała się wprost idealna dla potrzeb firmy — projekt został anulowany, nigdy go nie wznowiono, więc poprawną i minimalną ilością dokumentacji okazał się jej **brak**!

Zauważmy jednak, że zgodnie z „programowaniem jako budowaniem teorii" Naura możemy stwierdzić, że zespół stworzył własną teorię w trakcie tworzenia oprogramowania, ale nie pozostawił wystarczających śladów, by następny zespół mógł skorzystać z ich doświadczeń.

Przyklejanie myśli do ściany (powtórka)

Analitycy nie mogli poradzić sobie z dziedziną, która była zbyt złożona. Właśnie przeszli z ciężkiego procesu na XP i sądzili, że nie wolno im tworzyć żadnej papierowej dokumentacji.

Mijały miesiące i mieli coraz to większe problemy ze zdecydowaniem się na szczegóły następnych etapów prac i wskazanie

implikacji swoich decyzji. Znaleźli się poniżej linii wystarczalności dla swojej gry. Aby uzdrowić projekt, musieli zacząć tworzyć więcej dokumentacji.

W pewnym momencie dostrzegli tę sytuację i zaczęli tworzyć dokumentację, która pozwoliła im osiągnąć poziom wystarczalności.

Warto zauważyć, że „niewystarczalność" nie jest związana bezpośrednio z metodologią, ale raczej z brakiem dopasowania metodologii i projektu jako ekosystemu. To, co jest ledwo wystarczające dla jednego zespołu, może być aż nadto wystarczające lub niewystarczające dla innego. Niewystarczalność pojawia się, gdy członkowie zespołu nie komunikują się ze sobą wystarczająco dobrze, by zapewnić efektywne przekazywanie informacji.

Wartość idealna, „ledwo wystarczalna", zależy od rodzaju projektu i wielu innych czynników. Ta sama metodologia może być nadmiarowa w jednej części projektu, a niewystarczająca w innej jego części.

Druga umiejętność polega na znalezieniu punktu „ledwo wystarczalny" przy każdej większej zmianie elementów projektu.

Zalecenia dotyczące dokumentacji

Oto kilka zaleceń związanych z tworzeniem dokumentacji.

- Nie proś o to, by wymagania były doskonałe, dokumenty projektowe zawsze aktualne i odzwierciedlające stan kodu, a plan projektu zawsze odpowiadał jego aktualnemu stanowi.
- Proś, by wymagania zawierały wystarczającą ilość informacji, by zapewnić wydajną pracę projektantów. Proś o zastępowanie dokumentacji szybszymi środkami komunikacji, jeśli to tylko możliwe, włączając w to wizyty osobiste lub krótkie klipy wideo.
- Jeśli projektanci są ekspertami w swej dziedzinie i siedzą blisko siebie, poproś o zastąpienie dokumentacji projektowej rysunkami na tablicy, a następnie zrób zdjęcia tych rysunków.
- Pamiętaj jednak, o tym, że inne osoby przychodzące po zespole projektowym mogą wymagać bardziej szczegółowej dokumentacji.
- Niech dokumentacja będzie zadaniem równoległym, odzwierciedlającym walkę o zasoby, a nie liniową ścieżką przez cały proces tworzenia oprogramowania.
- Staraj się być możliwie pomysłowy w kwestii osiągania obu celów, ale unikaj bycia idealnym, bo to kosztuje zbyt wiele.
- Szukaj (używam tu nieco przesadzonych przymiotników) **najlżejszej i najmniej perfekcyjnej** metodologii, która sprosta wyznaczonemu zadaniu. Z drugiej strony zapewnij wystarczająco mocną i rygorystyczną komunikację.

Zwinność

Zwinność zakłada efektywność i manewrowość. Proces zwinny jest zarówno lekki, jak i wystarczający. Lekkość pozwala zachować zwrotność. Wystarczalność ma związek z chęcią pozostania danej osoby w grze.

Pytaniem związanym z metodologią zwinną nie jest: „Czy mogę w tej sytuacji zastosować metodologię zwinną?", lecz: „Jak w tej sytuacji pozostać zwinnym?".

Czterdziestoosobowy zespół nie będzie tak samo zwinny jako sześcioosobowy, znajdujący się w jednym pokoju. Każdy zespół może jednak starać się zmaksymalizować zastosowanie

zasad metodologii zwinnej, by być możliwie lekkim i szybkim. Zespół czterdziestoosobowy będzie musiał zastosować cięższą metodologię zwinną; mniejszy zespół może sobie pozwolić na lżejszą. Oba zespoły powinny skupić się na komunikacji, społeczności, częstym wygrywaniu i informacjach zwrotnych.

Jeśli zespoły będą uważały, będą okresowo sprawdzały dopasowanie stosowanej metodologii do ekosystemu i ewentualnie przemieszczenia punktu „ledwo wystarczalny".

Punkty krytyczne

Częścią bycia **zwinnym** jest identyfikacja krytycznych punktów wydajnego tworzenia oprogramowania, a następnie przesuwanie projektu tak blisko tych punktów, jak to możliwe.

Zespół, któremu uda się dostosować do jednego z punktów krytycznych, zaczyna korzystać z zalet wydajnego mechanizmu. Jeśli zespołowi nie uda się dostosować do któregoś z punktów krytycznych, musi pogodzić się z mniejszą efektywnością. Zespół powinien myśleć kreatywnie, w jaki sposób zbliżać się do punktów krytycznych i jak radzić sobie z sytuacją, gdy punkty te nie są możliwe do spełnienia.

Oto pięć punktów krytycznych.

Od dwóch do ośmiu osób w jednym pokoju

W tym układzie przesył informacji jest najszybszy.

Ludzie mogą zadawać sobie pytania bez zbytniego podnoszenia głosu. Wiedzą, kiedy inni są gotowi odpowiadać na pytania. Co więcej, przysłuchują się rozmowom innych bez przerywania własnej pracy. Szkice projektu i jego plan znajdują się na tablicy w zasięgu wzroku.

Ludzie bardzo często mówią mi, że choć to środowisko jest czasem hałaśliwe, najbardziej wydajna praca projektowa odbywa się właśnie w małych zespołach, które pracują w tym samym pokoju.

Gdy nie uda nam się osiągnąć tego punktu krytycznego, znacząco wzrasta koszt przenoszenia informacji. Każde drzwi, każdy narożnik i każde piętro zwiększa ten koszt.

W historii „E-bycie i e-świadomość" opowiadam o jednym z zespołów, który nie mógł spełnić tego punktu. Programiści zastosowali jednak kamery internetowe, by przynajmniej w części zniwelować konieczność pracy w różnych miejscach. By szybko odpowiadać na krótkie pytania, wykorzystywali komunikatory internetowe. Starali się w jak najlepszy sposób odtworzyć zalety punktu krytycznego w sytuacji, która teoretycznie to uniemożliwiała.

Eksperci klienta na miejscu

Możliwość stałego korzystania z ekspertów klienta oznacza, że bardzo szybko uzyskujemy informacje zwrotne na temat wyników pracy. Często są to minuty, rzadziej godziny.

Tego rodzaju szybkie informacje zwrotne powodują, że zespół programistyczny lepiej rozumie potrzeby i przyzwyczajenia klienta, więc popełnia mniej błędów przy wprowadzaniu nowych pomysłów. Wypróbowuje również więcej pomysłów, co czyni końcowy produkt znacznie lepszym. Przy dobrej współpracy, programiści testują pomysły ekspertów i oferują kontrpropozycje. Pozwala to klientom lepiej zorientować się we własnych żądaniach dotyczących sposobu działania systemu.

Koszt związany z niezastosowaniem tego punktu krytycznego dotyczy obniżenia prawdopodobieństwa wykonania naprawdę użytecznego produktu i zwiększenia kosztu różnorakich eksperymentów.

Istnieje wiele alternatywnych mechanizmów radzenia sobie z szybką informacją zwrotną, gdy nie uda się wprowadzić opisanego punktu, ale są one mniej efektywne. W ciągu ostatnich lat dosyć dobrze opisano alternatywy — cotygodniowe sesje sprawdzające z udziałem użytkowników; analizy etnograficzne społeczności użytkowników; ankiety; wykorzystanie zaprzyjaźnionych grup testujących. Z pewnością są i inne alternatywy.

Niemożność wprowadzenia wspomnianego punktu krytycznego nie zwalnia od starań związanych z uzyskaniem dobrej i szybkiej informacji zwrotnej. Najczęściej jednak zwiększa koszt uzyskiwania tego rodzaju informacji.

Jednomiesięczne przyrosty

Nic nie zastąpi szybkich informacji zwrotnych, zarówno na poziomie produktu, jak i na poziomie procesu programistycznego. Przyrostowe tworzenie oprogramowania pozwala zapewnić dobrą informację zwrotną. Krótkie przyrosty zapewniają, że zarówno wymagania, jak i sam proces są poprawiane bardzo szybko. Pozostaje pytanie: „Jak długie powinny być przerwy między kolejnymi dostarczeniami?".

Prawidłowa odpowiedź bywa różna, ale zespoły projektowe, które miałem okazję przepytywać, wskazywały na okres od jednego do trzech miesięcy z możliwą redukcją do dwóch tygodni lub rozszerzeniem do czterech miesięcy.

Wydaje się, że ludzie mogą skupić swoje wysiłki na maksymalnie trzy miesiące, ale nie dłużej. W przypadku dłuższych okresów wydań wiele osób informuje mnie o zbytnim rozkojarzeniu, utracie intensywności działań i kreatywności. Co bardzo ważne, przyrostowe rozwijanie aplikacji daje zespołowi szansę na naprawienie swojego procesu. Im dłuższy okres między wydaniami, tym dłuższy czas między możliwymi naprawami.

Jeśli byłby to jedyny aspekt sprawy, idealnym okresem przyrostowym byłby jeden tydzień. Trzeba jednak pamiętać o koszcie wdrożenia produktu na zakończenie każdego okresu.

Wydaje mi się, że najlepszym punktem krytycznym w tym przypadku jest jeden miesiąc, ale widziałem również udane projekty wykorzystujące okres o długości dwóch i trzech miesięcy.

Jeśli zespół nie może dostarczyć produktu końcowym użytkownikom co kilka miesięcy (niezależnie od powodów), warto mimo to tworzyć system w sposób przyrostowy i przygotowywać go do dostarczenia w wyznaczonych okresach czasu (udając, że sponsor żąda jego natychmiastowego dostarczenia w obecnej postaci). Celem takiej pracy jest ćwiczenie każdej części procesu wytwarzania oprogramowania i poprawianie wszystkich elementów procesu co kilka miesięcy.

W pełni zautomatyzowane testy regresyjne

W pełni zautomatyzowane testy regresyjne (jednostkowe, funkcjonalne lub oba) oferują dwie zalety:

- Programiści mogą modyfikować lub usprawniać kod, a następnie testować go jednym naciśnięciem przycisku. Dzięki automatycznym testom ludzie są bardziej skłonni poprawiać kod innych osób, bo wiedzą, że testy nie pozwolą im na wprowadzenie subtelnych błędów.
- Ludzie zgłaszają mi, że bardziej relaksują się w weekendy, jeśli mają automatyczne testy regresyjne. Co poniedziałek uruchamiają testy i sprawdzają, czy ktoś bez ich wiedzy zmodyfikował system.

Innymi słowy, zautomatyzowane testy poprawiają zarówno jakość systemu, jak i jakość życia programistów.

Istnieją pewne części systemów (a nawet całe systemy), dla których trudno napisać zautomatyzowane testy.

Jednym z przykładów mogą być graficzne interfejsy użytkownika. Doświadczeni programiści doskonale zdają sobie z tego sprawę, więc starają się zminimalizować liczbę podsystemów, które muszą być testowane ręcznie.

Jeśli sam system nie ma zautomatyzowanych testów, doświadczeni programiści starają się znaleźć sposoby tworzenia testów dla opracowywanych przez siebie partii systemów.

Doświadczeni programiści

W idealnej sytuacji — punkcie krytycznym — zespół składa się tylko i wyłącznie z doświadczonych programistów. Analizowane przeze mnie zespoły, które składały się z doświadczonych programistów, miały inne i lepsze wyniki w porównaniu z zespołami średnimi i mieszanymi.

Ponieważ dobrzy i doświadczeni programiści mogą być od dwóch do dziesięciu razy bardziej wydajni od swoich kolegów, można drastycznie zmniejszyć rozmiar zespołu, jeśli składa się tylko i wyłącznie z doświadczonych programistów.

Przed wykonywaniem i w trakcie wykonywania projektu „Winifred" estymowaliśmy, że projekt do udanej realizacji w wyznaczonym przedziale czasu potrzebuje 6 dobrych programistów języka Smalltalk. Ponieważ w tym czasie nie udało nam się pozyskać sześciu dobrych programistów, musieliśmy zaangażować aż 24. Czterech doświadczonych programistów tworzyło najtrudniejsze części systemu i dużo czasu spędzało na pomocy mniej doświadczonym kolegom.

Jeśli nie możesz spełnić tego punktu, zastanów się nad wprowadzeniem pełno- lub półetatowego trenera, który poprawi efektywność pracy niedoświadczonych osób.

Problem z zespołami wirtualnymi

Wirtualny oznacza w tym przypadku, że **członkowie zespołu nie siedzą razem**. Ponieważ słowo to zyskało ostatnio na popularności, sponsorzy projektów starają się nim wytłumaczyć ogromne bariery komunikacyjne stawiane zespołom.

We wcześniejszej części książki przedstawiłem szkody dla projektu powodowane przez rozdzielenie osób wskutek umieszczenia ich w różnych częściach budynku. Szybkość tworzenia systemu jest związana z czasem i energią potrzebną do przeniesienia pomysłów. Jeśli z powodu dużej odległości osób zwiększa się koszt przekazania wiedzy, zwiększa się także koszt związany z niezadaniem kluczowych pytań. Dzielenie zespołu to proszenie się o zwiększony koszt projektu.

Rozproszone geograficznie zespoły dzielę na trzy grupy o różnym poziomie szkód wyrządzanych projektowi. Poszczególnym kategoriom nadałem nazwy: **wiele lokalizacji**, **zamorski** i **rozproszony**.

Programowanie w wielu lokalizacjach

Programowanie w wielu lokalizacjach ma miejsce, gdy większy zespół pracuje w stosunkowo niewielkiej liczbie lokalizacji, a poszczególne grupy programistyczne tworzące poszczególne podsystemy znajdują się w jednej lokalizacji. Dodatkowym warunkiem jest stosunkowo mocna separacja poszczególnych podsystemów.

Ten sposób programowania i prowadzenia projektów jest z sukcesami stosowany od dziesięcioleci.

Kluczem do sukcesu w takim przypadku jest posiadanie pełnego i kompetentnego zespołu w każdej z lokalizacji oraz zapewnienie wystarczająco częstych spotkań poszczególnych grup, by mogły się dzielić swoimi doświadczeniami i wizjami.

Choć w takim sposobie pracy wiele spraw może pójść nie tak, jest to rozwiązanie sprawdzone, ze stosunkowo dobrze wypracowanymi regułami pracy (w odróżnieniu od dwóch pozostałych modeli zespołów wirtualnych).

Programowanie zamorskie

Ten rodzaj pracy występuje, gdy „projektanci" z jednej lokalizacji wysyłają specyfikacje i testy do „programistów" z innej lokalizacji, najczęściej z innego kraju.

Ponieważ zamorska lokalizacja nie ma architektów, projektantów i testerów, w znaczący sposób różni się sposobem pracy od programowania w wielu lokalizacjach.

Oto, w jaki sposób można opisać programowanie zamorskie, używając określeń związanych z grą zespołową i przepływem powietrza.

Projektanci w jednej lokalizacji muszą przekazać pomysły przez cienkie kanały komunikacyjne osobom, które stosują inne słownictwo i znajdują się tysiące kilometrów dalej. Programiści potrzebują odpowiedzi na tysiące pytań. Jeśli znajdą błędy w projekcie, muszą wykonać trzy kosztowne zadania: zaczekać na następne spotkanie telefoniczne lub wideo, przekazać swoje obserwacje i przekonać projektantów o możliwych błędach w projekcie. Koszt w erg-sekundach na mem jest zastraszający, a same opóźnienia ogromne.

Testowanie programowania zamorskiego

Pewien projektant powiedział mi, że jego zespół musi określać program niemalże na poziomie kodu źródłowego i tworzyć testy, by mieć pewność, że programiści poprawnie zaimplementowali wszystkie wymagania. Projektanci wykonywali całą nieprzyjemną robotę papierkową, ale nie dostawali nagrody w postaci pisania kodu.

Ponieważ tworzenie projektu i testów zajmowało ogromną ilość czasu, mogliby równie dobrze sami pisać kod oraz odkrywać błędy w projekcie i to najczęściej szybciej.

Nie udało mi się jeszcze znaleźć projektów prowadzonych w ten sposób, które okazałyby się udane metodologicznie. Zawsze nie przechodziły trzeciego testu: przesłuchiwane przeze mnie osoby twierdziły, że nie chcą ponownie pracować w ten sposób.

Na szczęście w ostatnim czasie coraz więcej firm programistycznych stara się zamienić swoje projekty w coś na kształt programowania w wielu lokalizacjach, umieszczając w jednym miejscu architektów, projektantów, programistów i testerów. Choć więź komunikacyjna nadal jest długa i cienka, członkowie zespołów mają przynajmniej cień szansy na skorzystanie z informacji zwrotnych i szybkości komunikacji w projektach prowadzonych w wielu lokalizacjach.

Programowanie rozproszone

Programowanie rozproszone występuje wtedy, gdy zespół jest rozmieszczony w stosunkowo **wielu** lokalizacjach ze stosunkowo niewielką liczbą osób (bardzo często jest to tylko jedna osoba lub są to dwie osoby) w każdym z miejsc.

Programowanie rozproszone staje się coraz bardziej popularne, ale nie znaczy to, że jest efektywne. Koszt przenoszenia pomysłów jest znaczny, a koszt straconej szansy spowodowany niezadaniem pytania jeszcze większy. Model rozproszony działa stosunkowo dobrze, jeśli naśladuje model z wieloma lokalizacjami,

w którym jedna lub dwie osoby stanowią w miarę niezależny podzespół. W takiej sytuacji zadania powierzane programiście są jasne i spójne.

Niestety, najczęściej mamy do czynienia z sytuacją przedstawioną poniżej.

Rozproszenie krzyżowe

Firma tworzyła cztery powiązane produkty w czterech lokalizacjach, a każdy produkt składał się z wielu podsystemów.

Najlepszym rozwiązaniem byłoby umieszczenie zespołów tworzących wszystkie systemy jednego produktu w tej samej lokalizacji lub ewentualnie rozmieszczenie w tym samym miejscu zespołów jednego podsystemu dla wszystkich produktów tworzonych w firmie. W obu przypadkach ludzie byliby w fizycznej bliskości z innymi osobami, z którymi muszą się wymieniać informacjami.

W praktyce okazało się, że dziesiątki osób zaangażowanych w projekty przydzielono w taki sposób, że w tym samym mieście pracowali ludzie przydzieleni do różnych podsystemów rożnych produktów. Byli więc otoczeni przez osoby, które w niewielkim stopniu mogły im pomóc w zdobyciu potrzebnych informacji, a osoby z którymi musiały się komunikować znajdowały się daleko!

Od czasu do czasu programiści informują mnie o efektywnym tworzeniu oprogramowania z kimś znajdującym się w całkowicie innym miejscu. Oznacza to, że cały czas jest jeszcze coś nowego do odkrycia: co pozwala ludziom tak dobrze komunikować się za pomocą tak słabego kanału komunikacyjnego? Czy to po prostu szczęśliwe dopasowanie osobowości lub stylów myślenia? Czy te dwie osoby wypracowały miniaturowy model odpowiadający programowaniu w kilku lokalizacjach? Czy korzystają z czegoś, czego większość z nas nie jest w stanie nazwać?

Udane programowanie rozproszone

Spędziłem pewien wieczór, rozmawiając z kilkoma osobami, które z sukcesem zaangażowały jako grupę czterech lub pięciu programistów, którzy nigdy się nie spotkali.

Powiedziały, że poza bardzo uważnym podzieleniem problemu spędzają dużo czasu przy telefonie, dzwoniąc do siebie kilka razy w ciągu dnia.

Poza tą raczej oczywistą taktyką koordynatorka zespołu pracowała szczególnie ciężko, by zapewnić wysoki poziom zaufania i przyjazności. Co kilka tygodni odwiedzała każdego z programistów i starała się, by jej wizyta była użyteczna i nie przeradzała się w sesję obwiniania.

Za wszelką cenę starała się powielić udany model tworzenia oprogramowania.

Pod koniec spotkania stwierdziła, że musi znaleźć innego koordynatora prac, podobnego do siebie — kogoś, kto będzie miał podobny talent do zapewniania atmosfery zaufania i przyjazności.

Zdziwiły mnie dwa aspekty ich pracy:

- zwracanie dużej uwagi na budowanie wzajemnego zaufania,
- ogromna ilość energii poświęcana na codzienną komunikację, by zapewnić odpowiednią wiedzę, zaufanie i informacje zwrotne.

Tworzenie wolnego oprogramowania

Choć tworzenie oprogramowania *open source* przypomina model rozproszony, różni się od niego w kwestiach filozoficznych, ekonomicznych i struktury zespołu.

Większość firm programistycznych w trakcie prowadzenia projektu gra w grę zespołową o ograniczonych zasobach, ale projekty *open source* grają w grę zespołową o **nieograniczonych zasobach**.

Zespół w projekcie komercyjnym stara się osiągnąć pewien cel w wyznaczonym czasie, mając do dyspozycji określoną sumę pieniędzy. Ograniczenie finansowe i czasowe określa, ilu ludzi i w jak długim przedziale czasu może pracować nad projektem. W tych grach często słyszymy następujące stwierdzenia:

„Ukończ to, zanim zamknie się okno rynkowe!".

„Twoim zadaniem jest równoważenie jakości i czasu wytwarzania!".

„Wdrażaj!".

Z drugiej strony projekt *open source* działa według zasady, że wystarczy odpowiednio dużo oczu, umysłów, palców i czasu, by powstał dobry model i naprawdę dobrej jakości kod. W szczególności istnieje teoretycznie nieograniczona liczba osób zainteresowanych rozwojem programu i nie ma żadnego okna rynkowego, w którym trzeba się zmieścić. Projekt żyje własnym życiem. Każda z osób poprawia system tam, gdzie jest słaby, wykorzystując dostępną dla siebie energię i czas.

Struktura nagród również jest inna, bo bazuje na wewnętrznych potrzebach, a nie zewnętrznych kwestiach finansowych. (Więcej informacji na ten temat znajdziesz w rozdziale 2.). Ludzie tworzą oprogramowanie *open source* dla przyjemności, jako usługę dla społeczności, o którą dbają, lub w celu bycia rozpoznawanym i cenionym przez innych. Ten model motywacyjny jest dokładniej opisany w tekście *Homesteading the Noosphere* (Raymond, *http://catb.org/~esr/writings/cathe*
↪*dral-bazaar/homesteading*).

Celem typowego programisty w firmie jest stanie się drugim Billem Gatesem. Porównywalnym celem dla programisty *open source* byłoby stanie się drugim Linusem Torvaldsem.

Warto również pamiętać o innej strukturze zespołów *open source* i zespołów tradycyjnych. Choć każdy może napisać lub poprawić fragment kodu, istnieją wybrani ochroniarze, którzy chronią centralną bazę kodu. Ochroniarz nie musi być najlepszym programistą. Wystarczy, że jest dobrym programistą z łatwym nawiązywaniem kontaktów i z bardzo dobrym okiem na wykrywanie szczegółów. Po pewnym czasie kilku najlepszych współtwórców programu zaczyna zajmować centrum, stając się głównymi właścicielami intelektualnymi projektu. Wokół tych ludzi zbiera się niezliczona liczba innych osób, które przesyłają poprawki i sugestie, wykrywają i zgłaszają błędy, a także piszą dokumentację.

Sugeruje się i zaleca, by **cała komunikacja** programistów *open source* była **dostępna dla każdego**. Rozważmy to w świetle projektu komercyjnego.

W projekcie komercyjnym z wykorzystującym zespół umieszczony w jednym miejscu problemy rozpoczynają się, jeśli społeczność programistów zaczyna dzielić się na klasę wyższą i niższą. Jeśli analitycy siedzą po jednej stronie, a programiści po drugiej, rozdzielenie „my i oni" łatwo buduje wrogość między poszczególnymi grupami (tzw. „frakcje"). W dobrze zrównoważonym projekcie jest tylko „my", nie ma rozróżnienia na „my i oni". W występowaniu lub braku występowania tego podziału kluczową rolę gra sposób wymiany informacji między grupami i pogaduszki. Jeśli istnieją enklawy zbierające specjalistów z jednej dziedziny, pogaduszki niemal na pewno dotyczyć będą również komentarzy na temat „tamtych".

W modelu *open source* równoważna sytuacja miałaby miejsce, gdyby jedna z podgrup (znajdująca się w jednym miejscu), prowadziła

pewną dyskusję, w której nie mogą uczestniczyć inne osoby. Osoby z rozproszonej grupy mogłyby wyrobić w sobie przeświadczenie, że są obywatelami drugiej kategorii odciętymi od serca społeczności i od istotnych konwersacji.

Gdy cała komunikacja jest dostępna w internecie i widoczna dla każdego, trudno ukryć różnego rodzaju pogłoski, więc zawsze jest tylko „my".

Chciałbym pewnego dnia zobaczyć i przeanalizować dokładniej ten aspekt projektów *open source*.

Dostosowywanie się

Jeśli czytasz książkę od samego początku, zapewne nadal nie potrafisz znaleźć odpowiedzi na jeden problem.

Każda osoba jest inna, każdy projekt jest inny, a każdy projekt różni się od innych wieloma szczegółami, podsystemami, podzespołami i zakresem czasowym. Każda sytuacja wymaga zastosowania innej metodologii (zbioru konwencji).

Sekret polega na sposobie konstrukcji różnych metodologii dostosowanych do konkretnych sytuacji, ale w taki sposób, by nie zabierało to dużo czasu i nie przeszkadzało w dostarczeniu oprogramowania. Nie chcemy również, by każda osoba uczestnicząca w projekcie musiała być ekspertem w zakresie metodologii.

Zapewne domyślasz się, co będzie tematem tego podrozdziału.

Potrzeba analizy przeszłych działań

Sztuczka z dostosowywaniem konwencji do ciągle zmieniających się potrzeb wymaga wykonania dwóch rzeczy na poziomie osobistym i zespołu.

1. **Zastanawiaj się nad tym, co robisz**.
2. **Niech zespół co drugi tydzień spędza godzinę na analizie własnych nawyków**.

Jeśli wykonasz te dwa zadania, powstająca metodologia będzie efektywna, zwinna i dostosowana do konkretnej sytuacji. Jeśli nie możesz tego zrobić… cóż, pozostaniesz tam, gdzie jesteś.

Choć tajemniczy składnik jest naprawdę prosty, bardzo trudno wprowadzić go w życie z powodu dziwactw ludzkiej natury. Ludzie najczęściej nie chcą nawet słyszeć o spotkaniu. W niektórych firmach poziom braku zaufania jest tak wysoki, że pewne osoby nie rozmawiają ze sobą, a co dopiero, gdyby mieli spotykać się na wspólnych spotkaniach.

W takiej sytuacji można zrobić tylko jedno — zorganizować spotkanie, przesłać wszystkim wnioski i zobaczyć, czy spotkanie uda się powtórzyć.

Warto znaleźć w firmie osobę, która ma odpowiednie predyspozycje do organizacji tego rodzaju spotkań. Czasem trzeba poszukać kogoś spoza zainteresowanych grup, kogoś o odpowiednich zdolnościach i dużej akceptacji przez różnych członków zespołu.

Technika rozwoju metodologii

Oto technika konstrukcji i dostrajania metodologii „w locie". Przedstawię, co należy robić w pięciu różnych momentach:

- teraz,
- na początku projektu,
- w połowie pierwszej iteracji,
- po każdej iteracji,
- w połowie następnej iteracji.

Po tym opisie przedstawię przykładowe, jednogodzinne ćwiczenie z analizy wcześniejszych działań.

Teraz

Odkryj mocne i słabe strony firmy dzięki krótkim rozmowom o projekcie.

Możesz to zrobić na początku projektu, ale równie dobrze możesz to zrobić teraz, niezależnie od poziomu zaawansowania projektu. Uzyskane informacje pomogą na każdym etapie prac. Możesz więc w dowolnym momencie budować własny zbiór rozmów o projekcie.

W idealnej sytuacji kilka osób może porozmawiać z kilkoma innymi osobami, by powstał zestaw od 6 do 10 raportów z wywiadów. Bywa użyteczne, choć nie jest wymagane, przeprowadzenie wywiadu z więcej niż jedną osobą z danego projektu. Można na przykład porozmawiać z dwoma osobami na następujących stanowiskach: menedżer projektu, lider zespołu, projektant interfejsu użytkownika i programista. Dzięki ich różnym sposobom patrzenia na projekt można wyrobić sobie lepsze pojęcie o sposobie jego działania. Niemniej bardziej istotne okazuje się uzyskanie typowych odpowiedzi dotyczących wielu projektów.

Pamiętaj, że istotne może okazać się **każde** słowo wypowiedziane przez rozmówcę. W trakcie wywiadu nie wyrażaj żadnych własnych opinii, ale korzystaj ze swojego doświadczenia i oceny sytuacji, by formułować kolejne pytania.

Wydaje mi się, że powinieneś zastosować następującą kolejność poruszanych kwestii:

1. Zapytaj o podanie jednego przykładu każdego wytworzonego produktu pracy.
2. Poproś o podanie krótkiej historii projektu.
3. Zapytaj, co należałoby zmienić następnym razem.
4. Zapytaj, co należałoby powtórzyć następnym razem.
5. Określ priorytety.
6. Znajdź dowolne luki w produkcie pracy lub projekcie.

Krok 1. Zapytaj o podanie jednego przykładu każdego wytworzonego produktu pracy

Analizując ten aspekt, łatwo stwierdzić, ile biurokratycznego narzutu występowało w projekcie i jakie szczegółowe pytania o produkty pracy należy zadać w trakcie wywiadu.

Szukaj duplikacji pracy oraz miejsc, gdzie trudno było utrzymać aktualność efektów pracy.

Zapytaj, czy używano programowania przyrostowego, a jeśli tak, to w jaki sposób aktualizowano dokumenty w kolejnych iteracjach.

Szukaj opisów sposobu, w jaki używano nieformalnej komunikacji, by rozwiązać niejasności w dokumentacji.

Duplikacja efektów pracy

W jednym z projektów lider zespołu przedstawił mi 23 produkty pracy.

Zauważyłem, że wiele z nich pokrywało się, więc zapytałem, czy te późniejsze były generowane przez narzędzia na podstawie wcześniejszych.

Lider odpowiedział, że nie; były od podstaw tworzone przez ludzi.

Zadałem więc pytanie, jak pracujące nad tym osoby postrzegały swoją pracę. Powiedział, że jej nienawidziły, ale i tak zmuszał je do jej wykonywania.

Po zobaczeniu przykładów efektów pracy przechodzimy do następnego punktu.

Krok 2. Poproś o podanie krótkiej historii projektu

Zapisz datę rozpoczęcia, ewentualne zmiany wielkości zespołu (powiększenie lub zmniejszenie), strukturę zespołu, dobre i złe chwile związane z pracą nad projektem.

Wykonaj to, by móc oszacować rozmiar i typ projektu, a także by znaleźć interesujące pytania do zadania.

Odkrywanie programowania przyrostowego

Oto, w jaki sposób poznałem fascynującą historię projektu, któremu nadałem nazwę „Ingrid" (Cockburn 1998).

W początkowej fazie projektu, zespół uzbierał większość znanych mi w tym czasie wskaźników porażki. To, że ich pierwsza, czteromiesięczna iteracja okazała się katastrofą, nie było dla mnie żadnym zaskoczeniem. Zacząłem się nawet zastanawiać, dlaczego przebyłem tak długą drogę, by usłyszeć o tak oczywistej porażce.

Niespodzianką okazało się to, co zrobili po pierwszej porażce.

Po pierwszej iteracji zmienili w projekcie niemalże wszystko. Nigdy wcześniej nie słyszałem o takim przypadku.

Cztery miesiące później ponownie zbudowali system, używając innych rozwiązań; nie drastycznie różnych, ale wystarczających.

Co cztery miesiące dostarczali działające i przetestowane oprogramowanie, a następnie siadali razem, by dowiedzieć się, co zrobili i jak usprawnić proces (zrób to samo).

Co najbardziej zaskakujące, nie tylko rozmawiali o tym, co należy zmienić, ale również rzeczywiście zmieniali swój sposób pracy.

Lekcja, jaką wyniosłem z tego wywiadu, nie miała nic wspólnego z pośrednimi efektami pracy, ale raczej z fenomenem chęci osiągnięcia sukcesu i chęci zmian (nawet co cztery miesiące), byle tylko osiągnąć wymarzony sukces.

Po usłyszeniu historii projektu i opowieści na temat różnych wzlotów i upadków przechodzimy do następnej kwestii.

Krok 3. Zapytaj, co należałoby zmienić następnym razem

Zapytaj: „Jakie są najważniejsze błędy popełnione w trakcie projektu, których nie chciałbyś powtórzyć w następnym projekcie?".

Zapisz odpowiedzi i staraj się dodatkowymi pytaniami poznawać szczegóły.

Po usłyszeniu tego, czego ludzie nie chcą zrobić, ponownie przejdź do następnego punktu.

Krok 4. Zapytaj, co należałoby powtórzyć następnym razem

Zapytaj: „Jakie są kluczowe sprawy, które z pewnością chciałbyś powtórzyć w następnym projekcie?".

Zapisz odpowiedzi. Jeśli ktoś odpowie: „Ta czwartkowa gra w siatkówkę była naprawdę dobra", także to zapisz.

Wspólne upijanie się

Pewnego razu, gdy zadałem to pytanie (w Skandynawii), uzyskałem odpowiedź: „Wspólne upijanie się".

Postanowiłem spróbować tego na własnej skórze i tego samego wieczoru wybraliśmy się do pubu. Rzeczywiście, następnego dnia zauważyłem znaczną poprawę poczucia wspólnoty.

Odpowiedzi na to pytanie bywają naprawdę przeróżne: od jedzenia w lodówce, przez wspólne wyjścia na miasto, kanały komunikacji, narzędzia programistyczne, architekturę systemów po model dziedzinowy. Zapisuj wszystko, co usłyszysz.

Krok 5. Określ priorytety

Zapytaj: „Jakie są twoje priorytety z uwzględnieniem spraw, które lubiłeś w projekcie? Co jest najważniejsze do utrzymania, a co możesz negocjować?". Zapisz odpowiedzi.

Warto również zadać pytanie: „Co cię najbardziej zaskoczyło w projekcie?".

Krok 6. Znajdź dowolne luki w produkcie pracy lub projekcie

Zapytaj, czy powinieneś coś jeszcze wiedzieć na temat projektu, i zobacz, gdzie to doprowadzi.

W pewnej firmie wykonaliśmy dwustronicowy szablon wywiadu, by zapisywać wyniki i łatwo wymieniać się informacjami. Szablon zawierał następujące części:

1. Nazwę projektu, stanowisko przesłuchiwanej osoby (sam przesłuchiwany pozostawał anonimowy).
2. Dane związane z projektem (początek i koniec, maksymalna wielkość zespołu, docelowa dziedzina, używane technologie).
3. Historia projektu.
4. Co poszło źle i czego nie należy powtarzać.
5. Co poszło dobrze i co należy powtórzyć.
6. Priorytety.
7. Inne.

Wykonaj ćwiczenie, zbierz wypełnione szablony i przyjrzyj się im dokładniej. W zależności od sytuacji kilka osób może spotkać się razem i porozmawiać o wywiadach lub też tylko przeczytać zebrane notatki.

Poszukaj wspólnych elementów w analizowanych projektach.

Wzorzec komunikacji

W firmie, w której wykonaliśmy szablon, we wszystkich projektach pojawił się ten sam schemat:

„Gdy mamy dobrą komunikację z klientami i sponsorami oraz wewnątrz zespołu, osiągamy dobre wyniki. Gdy taka komunikacja nie występuje, nie osiągamy dobrych wyników".

Choć przedstawione stwierdzenie wydaje się trywialne, rzadko zostaje dostrzeżone i zapisane. W rzeczywistości rok po zebraniu przedstawionych wcześniej wyników w firmie miało miejsce następujące zdarzenie.

Wzorzec komunikacji w akcji

Uczestniczyłem w jednym z trzech prowadzonych w tamtym czasie projektów. Każdy z nich dotyczył małych zespołów i ludzi siedzących w różnych miastach.

Jak zapewne się domyślasz, spędziłem mnóstwo czasu i energii na komunikacji ze sponsorami i programistami.

Wszystkie trzy projekty ukończono mniej więcej w tym samym czasie. Dyrektor działu programistycznego zapytał się, co jest powodem różnicy — dlaczego projekt, w którym uczestniczyłem zakończył się sukcesem, a dwa pozostałe prowadzone w tym samym czasie okazały się porażką.

Przypomniawszy sobie wcześniejsze wywiady, stwierdziłem, że może to mieć coś wspólnego z jakością komunikacji między programistami i sponsorami lub między poszczególnymi osobami z zespołu.

Stwierdził, że to bardzo interesujący pomysł. Zarówno programiści, jak i sponsorzy z dwóch innych projektów zgłaszali problemy komunikacyjne z liderami projektów. Zarówno programiści, jak i sponsorzy czuli się odizolowani. Z drugiej strony sponsorzy w moim projekcie byli bardzo zadowoleni z poziomu komunikacji.

W innej firmie znalazłem inny wspólny motyw. Oto, co usłyszałem od jednej z przesłuchiwanych osób.

Motyw różnicy kulturowej

Wszyscy nasi projektanci interfejsów użytkownika mają doktoraty z psychologii i siedzą kilka pięter nad programistami.

Wystąpiła więc między nimi i programistami luka edukacyjna, kulturowa i fizyczna.

Mieliśmy problemy z powodu innego podejścia tamtych osób do pewnych spraw, a także z powodu znacznej odległości od nich.

Firma potrzebowała wzmocnić mechanizmy komunikacji między dwoma wspomnianymi grupami, a także zastosować dodatkowe oceny efektów pracy.

Z przedstawionych historyjek warto zapamiętać, że to, czego nauczysz się w trakcie wywiadów, może okazać się niezwykle ważne już w następnym projekcie. Zwracaj uwagę na ostrzeżenia pojawiające się w trakcie wywiadów.

Na początku projektu

Postaraj się w jakiś sposób skrócić standardową metodologię firmy. Należy to zrobić niezależnie od tego, czy bazową metodologią jest ISO9001, XP, RUP, Crystal lub własne rozwiązania.

Etap 1. Dostosowanie metodologii bazowej

Jeśli to możliwe, niech nad propozycją bazowej metodologii dla projektu pracują dwie osoby. Praca będzie przebiegać szybciej, osoby te prawdopodobnie odkryją błędy w myśleniu drugiej strony i pomogą sobie nawzajem w doszlifowaniu pomysłów.

Muszą przejść przez cztery kroki.

1. Określić, pracę ilu osób trzeba będzie koordynować, a także poznać ich rozmieszczenie geograficzne (patrz siatka z rysunku 4.22). Określić, jakiego poziomu poprawności oprogramowania się wymaga i jakie szkody mogą spowodować ewentualne błędy. Określić i zapisać priorytety projektu: czas wypuszczenia na rynek, poprawność, inne istotne czynniki.
2. Wykorzystując zasady projektowania metodologii z rozdziału 4., osoby te muszą określić podstawowe parametry metodologii: podać, jak ścisłe powinno być przestrzeganie standardów; podać, jak rozbudowaną dokumentację należy wykonać; wskazać poziom obrzędowości ocen i długość przyrostu lub iteracji (czas wymagany do dostarczenia działającego kodu rzeczywistym użytkownikom, nawet jeśli są oni tylko testerami).

Jeśli okaże się, że czas iteracji jest dłuższy niż cztery miesiące, trzeba znaleźć inny sposób na tworzenie przetestowanej i działającej wersji systemu w mniej niż cztery miesiące, by zasymulować rzeczywiste dostarczanie systemu.

3. Wybrać metodologię bazową, która nie jest znacząco różna od sposobu pracy preferowanego przez zespół.

Pamiętaj, że łatwiej modyfikować istniejącą metodologię, niż tworzyć własną. Można zacząć od standardów korporacyjnych, opublikowanych wersji RUP, XP, Crystal Clear, Crystal Orange lub metodologii użytej w poprzednim projekcie.

4. Skrócić metodologię do podstawowych przepływów — kto przekazuje co i komu — a także wskazać konwencje, na które grupa zapewne się zgodzi.

Przedstawione zadania mogą zająć od jednego do kilku dni w przypadku projektów małej i średniej wielkości. Jeśli zaczyna wydawać się, że dwie osoby spędzą nad wyborem metodologii bazowej więcej niż tydzień, dodaj do zespołu jeszcze dwie osoby, by prace zakończyć w ciągu następnych dwóch dni.

Krok 2. Metodologia początkowa

Zwołaj zebranie zespołu, na którym zostanie omówiona metodologia bazowa i konwencje. Zmodyfikuj ją, by stała się metodologią początkową. W większych projektach, gdzie zebranie całego zespołu jest niepraktyczne, zbierz reprezentantów każdego stanowiska.

Celem spotkania jest:

- wyłapanie ozdobników,
- znalezienie sposobów usprawnienia procesu i zmniejszenia kosztu komunikacji,
- wykrycie problemów, których nie ujawnił szkic metodologii bazowej.

W trakcie spotkania postaraj się odpowiedzieć na następujące pytania:

- Jak długie będą iteracje i przyrosty (i czym będą się różniły)?
- Gdzie będą siedzieć ludzie?
- Co zrobić, by utrzymać wysokie morale i dobry poziom komunikacji?
- Jakie będą potrzebne produkty pracy i systemy oceny, jaka musi być ich obrzędowość?
- Które standardy narzędzi, rysowania, testów i kodu są obowiązkowe, a które tylko zalecane?
- Jak zostanie rozwiązane raportowanie czasu?
- Które konwencje należy ustalić już teraz, a które można wprowadzić i rozwinąć w późniejszym terminie?

Podstawowym celem spotkania jest wskazanie sposobów wykrywania ewentualnych problemów komunikacyjnych i dotyczących morale.

Wynikami spotkania powinny być:

- podstawowy przepływ pracy,
- kryteria współpracy poszczególnych ról, w szczególności w kwestii nakładania się obowiązków i deklarowania kamieni milowych,
- ogólne standardy i konwencje do wprowadzenia,
- rozwiązania komunikacyjne do przećwiczenia.

To Twoja metodologia początkowa.

Spotkanie powinno zająć połowę dnia, ale nie więcej niż cały dzień.

W połowie pierwszej iteracji

Niezależnie od tego, czy iteracja ma długość dwóch tygodni, czy trzech miesięcy, przeprowadzasz krótkie wywiady z członkami zespołu, indywidualnie lub grupowo, w połowie iteracji. Poświęć na nie od jednej do trzech godzin.

Oto podstawowe pytanie, które należy zadać:

> **Czy uda nam się to zrobić, jeśli będziemy pracować tak, jak teraz?**

Nie należy oczekiwać, iż w pierwszej iteracji uda się całkowicie zmienić sposób działania zespołu, chyba że jest on całkowicie zepsuty. Szukamy raczej sposobu bezpiecznego dostarczenia pierwszej wersji. Jeśli metodologia początkowa będzie mogła wytrzymać tak długo, będziesz miał więcej czasu, więcej doświadczenia i lepszy moment na wprowadzanie zmian tuż **po** pierwszym udanym dostarczeniu systemu.

Z tego powodu celem pierwszego wywiadu lub spotkania jest wykrycie, czy nie dzieje się coś krytycznego, co mogłoby zagrozić dostarczeniu pierwszej wersji.

Jeśli zauważysz, że aktualny sposób pracy nie sprawdza się, **najpierw** rozważ ograniczenie zasięgu pierwszego dostarczenia.

Większość zespołów przeszacowuje to, co jest w stanie dostarczyć w pierwszej wersji. To normalne i nie stanowi powodu do odrzucania metodologii. Wynika to po części z ambicji zarządu układającego nierealistyczne harmonogramy i zbyt optymistycznych programistów, którzy często zapominają o potrzebie nauki, spotkań i poprawiania błędów. Ma na to również wpływ szybkość uczenia się

nowych technologii i kolegów z zespołu. Przesadzenie z liczbą rzeczy, którą można dostarczyć w pierwszym wydaniu, naprawdę jest całkowicie normalne.

Pierwszym krokiem powinna być redukcja zasięgu.

Może się okazać, że zmniejszenie zasięgu nie wystarcza. Może się okazać, że wymagania nie są wystarczająco jasne dla programistów lub że architekci nie są w stanie w odpowiednim czasie zakończyć specyfikacji architektury.

Jeśli tak się dzieje, musisz zareagować szybko i znaleźć nowy sposób pracy. To w połączeniu z drastycznie zmniejszonym zasięgiem powinno spowodować udane dostarczenie pierwszej wersji.

Można wprowadzić nakładanie się prac, posadzić ludzi bliżej siebie, zmniejszyć idealność początkowej architektury i w lepszym stopniu wykorzystać nieformalne kanały komunikacji. Konieczne mogą okazać się awaryjne zmiany zespołu, awaryjne szkolenia, konsultacje i zatrudnienie doświadczonych programistów z zewnątrz.

Głównym celem jest dostarczenie czegoś — pewnego niewielkiego, działającego i przetestowanego kodu w pierwszej iteracji. To bardzo istotny czynnik sukcesu projektu (Cockburn 1998). Po wydaniu pierwszej wersji będzie można chwilę odsapnąć i dokładnie zastanowić się nad powodem problemów.

Po każdej iteracji

Po każdej iteracji zwołaj zebranie na temat wniosków dotyczących sposobu pracy.

Analiza wniosków to bardzo istotny czynnik sukcesu w ewolucji metodologii, podobnie jak programowanie przyrostowe stanowi jeden z czynników sukcesu wytwarzania oprogramowania.

Długość i intensywność analizy zależy od firmy i kraju. Amerykanie są zawsze zajęci, mają mało pieniędzy i ciągle biegną. Nie dziwi więc, że Amerykanie na analizę poświęcają jedynie od 2 do 4 godzin. W innych częściach świata analiza wniosków trwa dłużej.

Pewnego razu uczestniczyłem w dwudniowym spotkaniu poza miejscem pracy, które łączyło analizę sposobu pracy, budowanie więzi zespołu i planowanie następnej iteracji. Co nie powinno dziwić, miało miejsce w Europie.

Głównym powodem przesunięcia analizy do momentu zakończenia pierwszej iteracji jest możliwość poznania pełnego efektu wszystkich elementów metodologii, ponieważ udało się dostarczyć działające i przetestowane oprogramowanie. Tylko wtedy można ocenić, czego zrobiło się za mało, a czego za dużo.

Istnieje również drugi powód, dla którego warto zaczekać z analizą do końca iteracji: ludzie bardzo często są wyczerpani tuż po wydaniu oprogramowania. Spotkanie daje szansę na złapanie oddechu. Przeprowadzane regularnie integruje się z rytmem projektu. Po każdej iteracji członkowie zespołu chętnie skorzystają z odmiany i uspokojenia umysłu.

Niezależnie od tego, czy spotkanie zajmuje dwie godziny, czy dwa dni, należy przede wszystkim zadać dwa pytania:

1. „Czego się nauczyliśmy?”.
2. „Co możemy zrobić lepiej?”.

Odpowiedzi mogą dotyczyć różnych kwestii związanych z projektem: od zarządzania przez mierzenie czasu, komunikację w grupie, układ biurek, oceny projektu, po standardy i skład zespołu.

Bardzo często zespoły po pierwszej iteracji zacieśniają standardy, mają więcej doświad-

czenia, lepiej przekazują sobie pracę, robią więcej testów i znają strukturę zespołu.

Zmiany w drugiej i kolejnych iteracjach będą znacznie mniejsze, ponieważ zespół dostarczył już oprogramowanie i wie, jak to zrobić ponownie.

W połowie następnej iteracji

Po pierwszej iteracji zespół ma już jeden (ledwie wystarczający) udany sposób pracy. Użyta metodologia staje się w tym przypadku metodologią awaryjną.

Mając plan ratunkowy, można nieco odważniej proponować zmiany w trakcie spotkania w połowie drugiej i kolejnych iteracji.

W trakcie spotkań w połowie następnych iteracji starajcie się szukać nowych i lepszych sposobów dostarczania oprogramowania.

Zobacz, czy możesz wykonać któreś z poniższych zadań:

- wyciąć cały fragment metodologii,
- zwiększyć równoległość prac,
- lepiej wykorzystać komunikację nieformalną do przekazywania informacji o projekcie,
- wprowadzić nowe i lepsze systemy testowania,
- wprowadzić nowsze i lepsze wytyczne dotyczące pisania testów.
- zapewnić bliższą współpracę różnych grup uczestniczących w projekcie: ekspertów dziedzinowych i użytkowych, programistów, testerów, trenerów, biura obsługi klienta i biura napraw.

Do uzyskania danych w trakcie iteracji wykorzystuj wywiady bezpośrednie lub spotkania grupowe. Przez ten czas zespół zbierze odpowiednie doświadczenie i będzie wiedział, czego dotyczą spotkania i co należy w nich zgłaszać.

Jeśli w projekcie są stosowane iteracje o długości trzech tygodni lub krótsze, można zrezygnować z późniejszych analiz w połowie iteracji.

Dlaczego w ogóle zajmować się analizą w połowie iteracji, skoro już udało się dostarczyć oprogramowanie i są planowane analizy działania po każdej iteracji?

W połowie cyklu iteracji to, co sprawia największe problemy, jest najlepiej widoczne. Szczegóły problemu nie będą tak dobrze pamiętane za 6 lub 8 tygodni, czyli w trakcie spotkania po iteracji. Lepiej jest więc od razu poznać szczegóły problemów i wypróbować sposoby ich rozwiązania, niż czekać na ich poznanie kilka tygodni, a nawet miesięcy.

A jeśli nowy pomysł okaże się niewypałem?

Czasem zespół próbuje nowego pomysłu w drugiej lub trzeciej iteracji, ale okazuje się, że nowe rozwiązanie nie funkcjonuje zgodnie z prognozami.

Zmiany struktury zespołu w połowie projektu

W jednym projekcie przeszliśmy przez trzy różne struktury zespołu w trakcie trzeciej iteracji.

Na początku trzeciej iteracji stwierdziliśmy, że stosowana do tej pory struktura zespołu nie sprawdza się najlepiej. Wybraliśmy więc nową strukturę do zastosowania w trzeciej iteracji.

Okazała się katastrofą. Już po dwóch tygodniach wiedzieliśmy, że musimy ją szybko zmienić.

Zamiast wracać do oryginału, dziwnej ale zapewniającej sukces struktury, zasugerowaliśmy nowe podejście i od razu wcieliliśmy je w życie.

Okazało się dobre, więc stosowaliśmy je aż do końca projektu.

Dzięki przetestowaniu kilku rozwiązań w jednej iteracji, udało się nam szybko usprawnić metodologię. Warto pamiętać o istnieniu takiej szansy.

Ocena po zakończeniu projektu

Stosowanie analiz w trakcie i po każdej iteracji zmniejsza potrzebę stosowania ocen poprojektowych. Wydaje mi się, że ocen należy dokonywać w trakcie projektu, gdy zmiany mogą jeszcze przynieść projektowi coś dobrego. Po projekcie jest już na to za późno.

Najczęściej okazuje się, że zespoły, które stosują oceny poprojektowe nie przejmują się analizą w trakcie trwania projektu, a następnie bardzo mocno chcą się dowiedzieć, co poprawić w następnym projekcie. Jeśli znajdziesz się na takim spotkaniu, zasugeruj, by następnym razem przeprowadzać spotkania w trakcie trwania projektu, a nie po nim.

Z drugiej strony ocena poprojektowa to dobry moment do podsumowania ogólnej pracy zespołu i sposobu zarządzania projektem. W takiej sytuacji proponuję zapoznać się z książką *Project Retrospectives* (Kerth 2001), która omawia, w jaki sposób prowadzić dwudniową sesję oceny projektu.

Zastanów się w trakcie analizy przeprowadzanej po projekcie, kto mógłby skorzystać ze zgromadzonych informacji i jakie wskazówki można przekazać osobom prowadzącym następne projekty. Warto wykonać krótką (dwustronicową) notkę dla następnego zespołu, podsumowując plusy i minusy aktualnego projektu.

Oczywiście, warto także napisać krótkie jednostronicowe uwagi na końcu każdej z analiz po iteracji jako standardowy wynik ich przeprowadzenia.

Sposoby analizy projektu

Namacalnym wynikiem sesji analizy w trakcie iteracji lub po niej jest lista spraw umieszczana później na ścianie, by była widoczna dla wszystkich osób uczestniczących w projekcie i przypominała im o nowych zadaniach.

Osobiście lubię od razu w trakcie spotkania pisać na docelowym arkuszu papieru. To właśnie on najlepiej utkwi w pamięci całej grupy. Inne osoby lubią kopiować wypracowane wyniki na czyste arkusze, by zapewnić ich lepszy wygląd. Osoby, które tworzyły analizę przedstawioną na rysunku 3.10 zdecydowały się na użycie samoprzylepnych karteczek zamiast bloku papieru.

Przykładowy sposób analizy projektu

Istnieje kilka technik prowadzenia spotkania analizującego projekt i przedstawiającego wyniki. Preferuję użycie najprostszego rozwiązania, jakie uda się wymyślić. Oto skrócona wersja mojej propozycji (bardziej szczegółowy opis znajduje się w: Cockburn 2005 i Tabaka 2006).

> Analiza projektu
>
> Cześć. Witam na spotkaniu, którego celem jest analiza projektu, by móc doskonalić nasze sposoby wytwarzania oprogramowania.
>
> Celem tego spotkania nie jest wytykanie palcami, obwinianie ani unikanie obwiniania. Ma ono na celu wskazanie, w których miejscach się zacinamy, i zaproponowanie rozwiązania tych problemów.
>
> Wynikiem tego spotkania będzie arkusz papieru z pomysłami do zastosowania w następnej iteracji i sprawami, o których po prostu chcemy pamiętać.
>
> Arkusz podzielimy na trzy części.
>
> Po lewej stronie umieścimy rzeczy, które robimy dobrze i których nie chcemy stracić w następnej iteracji.

Po prawej stronie umieścimy rzeczy, nad którymi musimy jeszcze mocno popracować.

Po lewej stronie na dole umieścimy przeciwwagę do tego, co robimy dobrze, czyli wskażemy problemy, z którymi staramy się uporać (patrz rysunek 5.1).

Rysunek 5.1. Przykład wyników spotkania analizującego projekt

Zacznijmy od tego, co zrobiliśmy dobrze. Czy jest coś, co robimy dobrze, i chcemy, byśmy wykonywali to w ten sam sposób w następnej iteracji?

W tym momencie rozpoczyna się dyskusja. Całkiem możliwe, że ktoś zacznie wymieniać problematyczne obszary zamiast tego, co robimy dobrze. Jeśli problem jest istotny, zapisz go od razu w części dotyczącej problemów. Zapewnij czas na zastanowienie się i dyskusję.

W pewnym momencie poprowadź dyskusję dalej.

Dobrze. Jakie problemy wystąpiły w ostatniej iteracji i jak chcemy się ich pozbyć?

W części dotyczącej problemów pisz możliwie niewiele: każdy problem opisuj tylko kilkoma słowami i w miarę możliwości łącz podobne problemy. Celem tej części jest zasugerowanie obszarów wymagających poprawy, a nie skupianie się na problemach.

Zbierz sugestie. Jeśli lista stanie się bardzo długa, zapytaj, ile z tych nowych rozwiązań zespół rzeczywiście chce wprowadzić w następnej iteracji. Spoglądanie na długą listę spraw do poprawienia może działać depresyjnie. Najlepiej, by lista spraw do poprawienia była krótka. Pisanie po arkuszu grubym flamastrem to doskonały sposób szybkiego pozbywania się wolnego miejsca na kartce.

Okresowo sprawdzaj, czy ktoś nie zgłosił dodatkowych rozwiązań, które grupa powinna pozostawić.

Pod koniec spotkania ponownie przeanalizujcie całą listę. Zauważ, czy ludzie rzeczywiście zgadzają się na nowe pomysły, czy po prostu siedzą cicho.

Po spotkaniu umieść arkusz papieru w widocznym miejscu.

Na następne spotkanie tego typu przynieś wcześniejszy arkusz papieru i zacznij od analizy tego, co udało się wykonać, co się sprawdziło, a co jeszcze wymaga sprawdzenia.

Przeprowadzanie tego rodzaju spotkań co 2 – 6 tygodni zapewnia rozwój lokalnej kultury — metodologii zwinnej.

Wartość oceny projektu

Fragment na temat **shu-ha-ri** z wprowadzenia pasuje do wniosku płynącego z oceniania projektu:

„Gdy uczysz się techniki i gdy asymptotycznie dociera ona do twojego umysłu, gdy widzisz innych ćwiczących, zaczynasz się zastanawiać nad jej poprawą. Warto zadać kilka interesujących pytań:

1. Jak działa ta technika?
2. Dlaczego ta technika działa?

3. Jak ta technika wiąże się z innymi praktykowanymi przeze mnie technikami?
4. Jakie są konieczne warunki początkowe i końcowe, by zastosować tę technikę w walce?

Gdy tworzysz sensowny repertuar technik, z których potrafisz dobrze korzystać, musisz spotykać się z możliwie wieloma innymi praktykami". Gdy oglądasz innych, musisz zadać sobie trzy pytania i odpowiedzieć na nie:

1. Których praktyków cenię i podziwiam?
2. Czy wykonują działania inaczej ode mnie?
3. Jak mogę zmienić moje działania (model mentalny i jego realizację), by zniwelować różnice, które wydają mi się istotne?

Pytania na temat przeciwnika, które należy sobie zadać:

1. Gdzie możesz kontrolować działania i ruchy przeciwnika?
2. Gdzie możesz się uspokoić i uczynić techniki bardziej efektywnymi dzięki wyciszeniu umysłu?
3. Czy przeciwnik wygląda podobnie do podziwianych praktyków?

W trakcie analizy musisz szczerze ocenić wyniki każdego testu. Wracaj do shu przez ha i ri, jakbyś chadzał ślepymi uliczkami".

Sam nie mógłbym określić tego lepiej.

Co zatem będę robił jutro?

Potraktuj **zwinność** jako nastawienie, a nie wzór. Spójrz na aktualny projekt i zapytaj: „W jaki sposób możemy pracować według zasad zwinności?".

- Przyjrzyj się, jak daleko Twój zespół znajduje się od punktów krytycznych. Przekonaj się, jak w kreatywny sposób możesz się do nich zbliżyć lub je zasymulować.
- Zobacz, gdzie zespół może uczynić metodologię lżejszą. Zobacz, gdzie to nie wystarcza.
- Przeprowadź jeden z opisanych wywiadów na temat projektu.
- Niech inne osoby również przeprowadzą wywiady. Podzielcie się wynikami. Znajdź w wywiadach wspólne wątki.
- Przeprowadź jednogodzinną analizę projektu. Gdy zaczną się opisy problemów, zapisz je. Porównaj je następnie z listą przedstawioną w rozdziale 3. Poszukaj rozwiązań i rozszerz listę. Wyniki analizy umieść na arkuszu papieru. Zauważ, ile osób się nim zainteresowało.
- Przekonaj się, czy łatwiej zebrać ludzi na drugą analizę projektu. Naucz się przekonywać ludzi do mniejszego narzekania, a częstszego zgłaszania sposobów naprawy sytuacji.

Staraj się stać projektantem metodologii drugiego poziomu. Tak, to część Twojej profesji.