
DAVID FARLEY
NOWOCZESNA INŻYNIERIA OPROGRAMOWANIA
Stosowanie skutecznych
technik szybszego rozwoju
oprogramowania wyższej jakości
Przedmowa TRISHA GEE
Helion

Tytuł oryginału: Modern Software Engineering: Doing What Works to Build Better Software Faster

Tłumaczenie: Tomasz Walczak

ISBN: 978-83-8322-594-4

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/nowoio
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
https://ftp.helion.pl/przyklady/nowoio.zip

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

4

Praca w modelu iteracyjnym

Iterację można zdefiniować jako „procedurę, w której powtarzanie sekwencji operacji daje efekty coraz bliższe pożądanemu wynikowi”[1].

Na podstawowym poziomie iteracja jest procedurą, która ułatwia uczenie się. Iteracje umożliwiają uczenie się, reagowanie i dostosowywanie do uzyskanych informacji. Bez iteracji i ściśle powiązanego z nimi zbierania informacji zwrotnych ciągłe uczenie się jest niemożliwe. Iteracje pozwalają popełniać błędy i je korygować lub robić postępy i je rozwijać.

Wspomniana definicja przypomina, że iteracje umożliwiają stopniowe podążanie w kierunku określonego celu. Rzeczywista ich wartość związana jest z tym, że możemy zbliżać się do celu nawet w sytuacji, gdy początkowo nie wiemy, jak się za to zabrać. O ile tylko potrafimy ocenić, czy zbliżamy się do celu, czy oddalamy się od niego, możemy nawet podejmować losowe decyzje w trakcie iteracji i mimo to osiągnąć cel. Można odrzucać kroki, które oddalają zespół od celu, i wybierać decyzje, które do niego przybliżają. Jest to istota ewolucji. Jest to także podstawa działania uczenia maszynowego.

Rewolucja podejścia zwinnego

Zespoły stosują iteracyjne i oparte na informacjach zwrotnych techniki przynajmniej od lat 60. XX wieku. Jednak dopiero po głośnym spotkaniu czołowych myślicieli i specjalistów w ośrodku narciarskim w Kolorado powstał *Manifest Agile*, w którym nakreślono ogólną filozofię będącą podstawą tych elastycznych, opartych na uczeniu się strategii stojących w kontraście do popularnych wówczas cięższych procesów.

Manifest Agile[2] jest prostym dokumentem. Obejmuje 9 wierszy tekstu i 12 zasad, wywarł jednak duży wpływ.

---

[1] Źródło: Merriam Webster Dictionary, *https://www.merriam-webster.com/dictionary/iteration*.

[2] *Manifest Agile*, *https://agilemanifesto.org/*.

Wcześniej powszechna wiedza (z którą nie zgadzali się tylko nieliczni milczący nonkonformiści) nakazywała, aby przy „poważnych" pracach nad oprogramowaniem stosować typowe dla produkcji techniki związane z podejściem kaskadowym.

Musiało minąć trochę czasu, aby podejście zwinne zyskało uznanie, jednak obecnie to ono, a nie model kaskadowy, jest dominującym podejściem — przynajmniej jeśli chodzi o sposób myślenia.

Jednak większość organizacji wciąż jest kulturowo zdominowana przez myślenie kaskadowe. Na pewno dzieje się tak na poziomie organizacyjnym, a często także na poziomie technicznym.

Mimo to myślenie zwinne jest zbudowane na bardziej stabilnych fundamentach niż wcześniejsze podejścia. Określenie, które najlepiej ujmuje idee (lub może raczej ideały) społeczności zwinnej, to „sprawdzaj i adaptuj".

Ta zmiana perspektywy jest znacząca, ale niewystarczająca. Dlaczego to podejście jest tak ważne? Ponieważ stanowi krok w kierunku postrzegania rozwoju oprogramowania jako zadania z obszaru uczenia się, a nie tylko jako problemu produkcyjnego. Procesy kaskadowe mogą być skuteczne w problemach produkcyjnych określonego typu, jednak bardzo słabo sprawdzają się w problemach wymagających eksploracji.

Zmiana jest ważna także z innego powodu. Choć dziesięciokrotna poprawa, o jakiej pisał Fred Brooks, wydaje się niemożliwa w obszarach technologii, narzędzi lub procesów, istnieją podejścia, które są tak niewydajne, że usprawnienie ich o rząd wielkości jest jak najbardziej wykonalne. Podejście kaskadowe stosowane do rozwoju oprogramowania jest jednym z kandydatów do tego.

Myślenie kaskadowe rozpoczyna się od następującego założenia: „Jeśli tylko będziemy myśleć lub pracować wystarczająco ciężko, uda nam się od początku poprawnie zaprojektować rozwiązanie".

Myślenie zwinne odwraca to nastawienie. Zaczyna się od założenia, że z pewnością popełnimy błędy. „Nie zrozumiemy, czego użytkownicy oczekują", „Nie stworzymy od razu poprawnego projektu", „Nie będziemy wiedzieć, czy wykryliśmy wszystkie błędy w napisanym kodzie" itd. Ponieważ zespoły zwinne zaczynają od przyjęcia, że popełnią błędy, mogą pracować w sposób, który celowo ogranicza koszty pomyłek.

Jest to idea wspólna dla myślenia zwinnego i nauki. Sceptyczne traktowanie pomysłów i dążenie do dowiedzenia ich fałszu zamiast prawdziwości (ważne jest tu słowo „falsyfikowalność") są typowe dla nastawienia naukowego.

Te dwie szkoły myślenia, oparte na przewidywaniu i na eksploracji, prowadzą do zupełnie odmiennych i niezgodnych sposobów organizowania projektów i funkcjonowania zespołów.

Jeśli przyjmiesz założenia typowe dla myślenia zwinnego, będziesz tak organizować zespoły, procesy i technologie, aby umożliwić bezpieczne popełnianie pomyłek, łatwe ich dostrzeganie, wprowadzanie zmian i, w idealnych warunkach, tworzenie lepszych rozwiązań w kolejnym podejściu.

Nieistotne są spory o to, czy należy stosować scrum czy programowanie ekstremalne, ciągłą integrację czy technikę feature branching, programowanie sterowane testami czy dogłębną analizę przez doświadczonych programistów. W swej istocie wszystkie prawdziwie zwinne procesy dotyczą „empirycznej kontroli procesu".

Ten model zdecydowanie lepiej pasuje do rozwoju oprogramowania dowolnego rodzaju niż stosowane wcześniej podejście kaskadowe, typowe dla produkcji i oparte na przewidywaniach.

Praca iteracyjna w fundamentalny sposób różni się od pracy w ściślej zaplanowany i sekwencyjny sposób. Jest też zdecydowanie bardziej efektywna.

Wielu czytelnikom może się to wydawać oczywiste, ale takie nie jest. Przez dużą część historii rozwoju oprogramowania zakładano, że iteracje są zbędne i że szczegółowe zaplanowanie wszystkich kroków jest celem wczesnych etapów rozwoju oprogramowania.

Iteracja jest istotą uczenia eksploracyjnego i podstawą każdego procesu zdobywania wiedzy.

Praktyczne zalety podejścia iteracyjnego

Jeśli potraktujesz inżynierię oprogramowania jak ćwiczenie z odkrywania i uczenia się, koniecznie musisz stosować podejście iteracyjne. Różne inne zalety iteracyjnej pracy mogą nie być od razu oczywiste.

Prawdopodobnie najważniejsza idea dotyczy tego, że jeśli zaczniesz zmieniać sposób pracy w kierunku iteracyjnego, automatycznie przełoży się to na zawężenie uwagi i zachęci do myślenia w mniejszych porcjach oraz do poważniejszego potraktowania modułowości i podziału zadań. Te aspekty początkowo są naturalnym skutkiem iteracyjnej pracy, a ostatecznie tworzą samonapędzający się mechanizm, który zwiększa jej jakość.

Jedną z cech wspólnych technik scrum i programowania ekstremalnego jest to, że należy wykonywać niewielkie porcje pracy. Proces myślenia w podejściu zwinnym wygląda tak: „Trudno jest mierzyć postępy w rozwoju oprogramowania, jednak można mierzyć ukończone funkcje, dlatego należy pracować nad mniejszymi funkcjami, co pozwoli im się przyjrzeć, gdy zostaną ukończone".

Ta redukcja wielkości porcji pracy była dużym krokiem naprzód. Jednak sytuacja się komplikuje, gdy chcesz ustalić, ile czasu zajmie Ci ukończenie danej porcji. To iteracyjne podejście do rozwoju oprogramowania różni się od tradycyjnych sposobów myślenia. Na przykład ciągłe dostarczanie wymaga, aby możliwe było udostępnienie każdej niewielkiej zmiany, co dzieje się kilka razy dziennie. Każda porcja pracy powinna być ukończona w takim stopniu, aby w każdym momencie można było bezpiecznie udostępnić oprogramowanie w środowisku produkcyjnym. Co więc oznacza słowo „ukończone" w takim kontekście?

Każda zmiana jest ukończona, gdy można ją udostępnić. Dlatego jedyną sensowną miarą poziomu ukończenia jest to, czy zapewnia jakąś wartość użytkownikom. Jest to bardzo subiektywna kwestia. Jak przewidzieć, ile zmian jest potrzebnych, by zapewnić „wartość" użytkownikom? Większość organizacji domyśla się, jaki zestaw funkcji w połączeniu będzie „wartościowy". Jeśli jednak oprogramowanie można udostępnić na dowolnym etapie cyklu jego życia, takie podejście jest nieco rozmyte.

Trudno jest określić zestaw zmian, które zapewniają „wartość", ponieważ zależy to od założenia, że od początku znasz wszystkie potrzebne funkcje i możesz ocenić postępy na drodze do jakiejś „kompletności". Jest to nadmierne uproszczenie tego, co twórcy ruchu zwinnego mieli na myśli. Jednak właśnie takie założenie przyjmowano w większości tradycyjnych organizacji, które chciały wprowadzić planowanie w duchu zwinnym.

Jedną z subtelnych zalet podejścia iteracyjnego jest możliwość wyboru. Można iteracyjnie modyfikować utworzone produkty na podstawie informacji zwrotnych od klientów i użytkowników, aby dążyć w kierunku rozwiązań o wyższej wartości. Jest to jeden z najcenniejszych aspektów tego podejścia, jednak często przeoczany przez tradycyjne organizacje, które próbują je stosować.

Niezależnie od tego, czy taki był cel, podejście oparte na pracy w małych porcjach zachęciło branżę do zmniejszenia wielkości i poziomu złożoności rozwijanych funkcji. Jest to bardzo istotny krok.

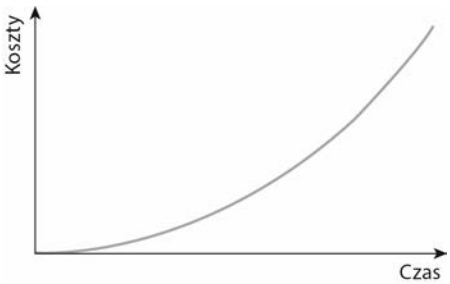
Planowanie w podejściu zwinnym zależy w dużym stopniu od podziału pracy na wystarczająco małe porcje, aby można było ukończyć funkcje w trakcie jednego sprintu, czyli jednej iteracji. Początkowo ten model opisywano jako sposób pomiaru postępów, jednak miał on znacznie większy wpływ, ponieważ zapewnił regularny dostęp do rzetelnych informacji zwrotnych na temat jakości i adekwatności prac. Ta zmiana zwiększyła możliwą szybkość uczenia się. Czy dany projekt się sprawdza? Czy użytkownikom podoba się dana funkcja? Czy system jest wystarczająco szybki? Czy udało się wyeliminować wszystkie błędy? Czy z kodem dobrze się pracuje? I tak dalej.

Iteracyjna praca nad niewielkimi, kompletnymi i gotowymi do zastosowania w środowisku produkcyjnymi etapami zapewnia świetne informacje zwrotne.

Podejście iteracyjne jako strategia projektowania defensywnego

Praca iteracyjna zachęca do zastosowania defensywnego sposobu projektowania. Szczegółowo omawiam to zagadnienie w części III.

Mój przyjaciel, Dan North, jako pierwszy przedstawił mi ciekawy punkt widzenia na podstawy myślenia zwinnego. Dan opisał różnicę między myśleniem w modelach kaskadowym i zwinnym, która odpowiada problemowi z dziedziny ekonomii. W podejściu kaskadowym obowiązuje założenie, że wraz z upływem czasu zmiany są coraz bardziej kosztowne. W klasycznym ujęciu używany jest model kosztów zmian (zobacz rysunek 4.1).



Rysunek 4.1. Klasyczny model kosztów zmian

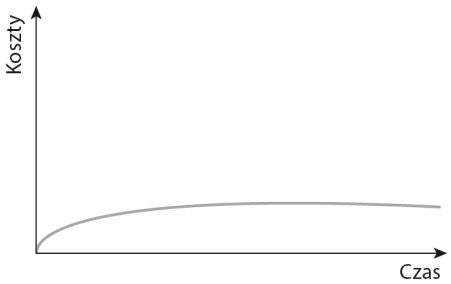
Ta perspektywa prowadzi do problemów. Jeśli ten model jest poprawny, oznacza to, że jedynym sensownym rozwiązaniem jest podjęcie najważniejszych decyzji projektowych na wczesnych etapach życia projektu. Jednak na początku prac wiemy o zadaniu najmniej. Dlatego podejmujemy najważniejsze decyzje w życiu projektu na podstawie niepewnych domniemań i to niezależnie od tego, jak dużo wysiłku włożymy w uzyskanie potrzebnych informacji.

Rozwój oprogramowania nigdy nie zaczyna się od stanu „w pełni rozumiemy wszystkie aspekty zadania". Nie ma przy tym znaczenia, jak starannie przeanalizujemy sytuację przed

rozpoczęciem prac. Tak więc ponieważ nigdy nie zaczynamy od dobrze zdefiniowanego zestawu danych wejściowych, to niezależnie od tego, jak starannie zaplanujemy działania, model oparty na ściśle zdefiniowanym procesie (czyli podejście kaskadowe) zawiedzie przy pierwszych przeszkodach. Nie da się dostosować rozwoju oprogramowania do tego niedopasowanego modelu.

Niespodzianki, nieporozumienia i pomyłki są w rozwoju oprogramowania normą, ponieważ jest to proces obejmujący eksplorację i odkrywanie. Dlatego musimy się skupić na tym, aby nauczyć się chronić przed błędami, które z pewnością się pojawią.

A oto pomysł Dana Northa — skoro klasyczny model kosztów zmian nie jest pomocny, co byłoby bardziej przydatne? Czy nie byłoby znacznie łatwiej, gdyby udało się wypłaszczyć krzywą kosztów zmian? Spójrz na rysunek 4.2.



Rysunek 4.2. Koszty zmian w podejściu zwinnym

Co by się stało, gdybyśmy mogli zmieniać zdanie, odkrywać nowe pomysły, wykrywać usterki i poprawiać je mniej więcej przy tych samych kosztach niezależnie od momentu, gdy się to dzieje? Co by było, gdyby krzywa kosztów zmian była płaska?

Zyskalibyśmy wtedy swobodę odkrywania nowych rzeczy i czerpania korzyści z tych odkryć. Umożliwiłoby to zastosowanie podejścia, w którym można stale zwiększać poziom wiedzy, kodu i wygody użytkowników z korzystania z naszych produktów.

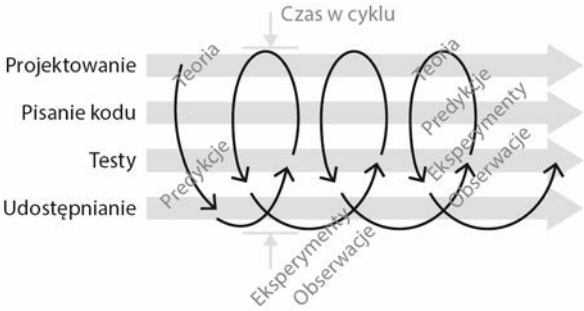
Co jest potrzebne, aby uzyskać płaską krzywą kosztów zmian?

Nie można pozwolić sobie na długą fazę analiz i projektowania bez tworzenia czegokolwiek, ponieważ oznacza to dłuższy czas, przez jaki nie uczymy się, co naprawdę się sprawdza. Trzeba więc skompresować proces i pracować iteracyjnie. Należy przeznaczyć tylko tyle czasu na analizy, projektowanie, pisanie kodu, testy i wdrażanie, aby przekształcić idee w produkt możliwy do zaprezentowania klientom i użytkownikom. W ten sposób dowiemy się, co rzeczywiście się sprawdza. Należy przemyśleć informacje zwrotne, a następnie na podstawie uzyskanej wiedzy ustalić dalszy plan działań i wykorzystać w nim nowe dane.

Jest to jedna z najważniejszych idei w ciągłym dostarczaniu (zobacz rysunek 4.3).

Pokusa tworzenia planu

Ludzie propagujący podejście kaskadowe mieli dobre intencje. Uważali, że jest to najlepszy sposób pracy. W naszej branży przez dziesięciolecia próbowano sprawić, aby to podejście zaczęło działać, ale bez powodzenia.



Rysunek 4.3. Iteracja w ciągłym wdrażaniu

Problem polega na tym, że opis podejścia kaskadowego brzmi bardzo sensownie: „dobrze się zastanów, zanim zaczniesz", „starannie zaplanuj, co będziesz robić, a następnie skrupulatnie realizuj ten plan". W kontekście doświadczeń epoki przemysłowej te pomysły wydają się uzasadnione. Jeśli masz dobrze zdefiniowany proces, tego rodzaju podejście do kontrolowania go sprawdza się świetnie.

Przy produkcji fizycznych obiektów problemy inżynierii produkcyjnej i skalowania często są ważniejsze niż problemy projektowe. Jednak obecnie zmienia się to nawet w dziedzinie wytwarzania fizycznych przedmiotów. Ponieważ firmy produkcyjne stają się coraz bardziej elastyczne, a niektóre fabryki mogą zmieniać to, co wytwarzają, nawet w branży produkcyjnej podważa się sztywne procesy i rezygnuje z nich. Jednak myślenie w kategoriach linii produkcyjnych dominowało w większości organizacji przynajmniej przez sto lat, dlatego na pewnym poziomie jesteśmy zaprogramowani, aby myśleć o problemach w ten właśnie sposób.

Konieczny jest trudny skok intelektualny, aby dostrzec, że paradygmat, w ramach którego pracujemy, jest fundamentalnie błędny. Jest to jeszcze trudniejsze, gdy cały świat zakłada, że ten paradygmat jest poprawny.

Wojny procesów

Skoro nie da się uzyskać dziesięciokrotnej poprawy za pomocą języków, formalnych metod lub diagramów, gdzie indziej można szukać?

Obszarami, którym warto się przyjrzeć, są sposób organizowania pracy oraz podejście do umiejętności i technik uczenia się i odkrywania, które to zjawiska są nieodłączne od naszej dziedziny.

Gdy branża rozwoju oprogramowania dopiero powstawała, pierwsi programiści mieli zwykle wykształcenie z zakresu matematyki, nauk ścisłych lub inżynierii. Pracowali nad systemami samodzielnie lub w małych grupach. Ci ludzie byli pionierami w nowej dziedzinie i podobnie jak większość eksploratorów wzięli ze sobą swoje doświadczenia i uprzedzenia. Dlatego pierwsze metody rozwoju oprogramowania były często wysoce matematyczne.

W wyniku rewolucji komputerowej rozwój oprogramowania rozpoczął się na znacznie większą skalę, a zapotrzebowanie na aplikacje wyraźnie przekraczało podaż. Trzeba było produkować więcej lepszego oprogramowania i to w krótszym czasie. Zaczęto obserwować inne branże, aby spróbować skopiować stosowane w nich sposoby wydajnej pracy na dużą skalę.

To w tym momencie popełniono straszliwy błąd dotyczący niewłaściwego zrozumienia fundamentalnej natury rozwoju oprogramowania. W efekcie wprowadzono nieodpowiednie techniki z branży produkcyjnej. Firmy zatrudniały rzesze programistów i próbowały zbudować programistyczne odpowiedniki linii do masowej produkcji towarów.

Ludzie, którzy za to odpowiadali, nie byli głupcami, ale popełnili poważny błąd. Musieli poradzić sobie z wieloaspektowym problemem. Oprogramowanie jest skomplikowane, a proces jego tworzenia nie przypomina tradycyjnego problemu produkcji, choć wiele osób najwyraźniej uznało inaczej.

Pierwsze próby „uprzemysłowienia" naszej dziedziny były bolesne, powszechne i bardzo szkodliwe. Efektem było utworzenie dużej ilości oprogramowania, jednak często wiązało się to z problemami. Aplikacje były powolne, niewydajne, dostarczane za późno i nie zapewniały tego, czego użytkownicy oczekiwali. Ponadto były niezwykle trudne w konserwacji. W latach 80. i 90. XX wieku w dziedzinie rozwoju oprogramowania nastąpił gwałtowny postęp, jednak podobnie wzrosła też złożoność związanych z nią procesów stosowanych w wielu dużych organizacjach.

Te kłopoty pojawiały się mimo tego, że wiele aspektów problemu było dobrze znanych czołowym myślicielom z naszej branży.

Fred Brooks (znów on) opisał te problemy i sposoby na ich uniknięcie w 1970 roku w książce *Legendarny osobomiesiąc*. Jeśli nie znasz jeszcze tej przełomowej pozycji z naszej dziedziny, prawdopodobnie zdziwisz się, jak trafnie Brooks przedstawił trudności, z którymi zapewne nadal zmagasz się prawie każdego dnia w pracy programisty. Dzieje się tak, choć książka Brooksa jest oparta na jego doświadczeniach w rozwoju systemu operacyjnego dla komputera mainframe IBM 360 z końca lat 60. XX wieku, do czego stosowano dość prymitywne technologie i narzędzia dostępne w tamtych czasach. Brooks ponownie poruszył kwestie ważniejsze i bardziej fundamentalne niż język, narzędzia lub technologie.

W tamtych czasach wiele zespołów rozwijało wysokiej klasy oprogramowanie, często kompletnie ignorując ówczesną „wiedzę" z zakresu planowania projektów i zarządzania nimi. W takich zespołach często można było dostrzec powtarzające się schematy. Te zespoły zwykle były małe. Programiści pozostawali w bliskim kontakcie z użytkownikami oprogramowania. Szybko testowali idee i zmieniali kierunek, jeśli rozwiązanie działało niezgodnie z oczekiwaniami. W tamtych czasach było to rewolucyjne podejście — w tym stopniu, że wiele takich zespołów nie przyznawało się do stosowanych metod, ponieważ ich organizacje posługiwały się ciężkimi procesami spowalniającymi pracę.

Pod koniec lat 90. XX wieku w reakcji na ciężkie procesy zaczęły pojawiać się próby zdefiniowania skuteczniejszych strategii. Popularność zaczęły zyskiwać różne konkurencyjne podejścia: crystal, scrum, programowanie ekstremalne i kilka innych technik, które odzwierciedlały zupełnie nowy sposób pracy. Te podejścia zostały w formalny sposób ujęte w *Manifeście Agile*.

W branży oprogramowania potrzebna była rewolucja podejścia zwinnego, aby odrzucić dawne normy. Jednak jeszcze dziś wiele organizacji (a może nawet ich większość) jest przesiąkniętych opartym na planach podejściem kaskadowym.

W organizacjach trzymających się modelu kaskadowego obok trudności z dostrzeżeniem problemu występuje myślenie życzeniowe. Byłoby wspaniale, gdyby taka organizacja potrafiła:

- poprawnie zidentyfikować potrzeby użytkowników;
- trafnie ocenić wartość oprogramowania dla organizacji, jeśli te potrzeby zostaną spełnione;
- precyzyjnie oszacować koszty spełnienia tych potrzeb;

- podjąć racjonalną decyzję dotyczącą tego, czy korzyści przeważą nad kosztami;
- opracować precyzyjny plan;
- bezproblemowo zrealizować ten plan;
- cieszyć się z zysku po zakończeniu prac.

Problem polega na tym, że jest to niemożliwe ani na poziomie biznesowym, ani na poziomie technicznym. Rzeczywisty świat i rozwój oprogramowania tak nie wyglądają.

Z danych branżowych wynika, że w najlepszych firmach informatycznych na świecie dwie trzecie pomysłów przynosi zerowe zyski lub nawet generuje straty[3]. Bardzo kiepsko radzimy sobie ze zgadywaniem, czego użytkownicy oczekują. Nawet gdy ich o to zapytamy, sami nie będą wiedzieć, czego tak naprawdę chcą. Najskuteczniejsze jest tu podejście iteracyjne. Należy zaakceptować, że niektóre, a nawet liczne pomysły okażą się błędne, i pracować w taki sposób, aby można je było wypróbować możliwe szybko, tanio i wydajnie.

Także ocena biznesowej wartości idei jest niezwykle trudna. Znany jest cytat Thomasa J. Watsona, prezesa firmy IBM, który prognozował, że światowe zapotrzebowanie na komputery pewnego dnia sięgnie aż pięciu sztuk!

Nie jest to problem z obszaru technologii, tylko kwestia ludzkich ograniczeń. Aby robić postępy, musimy próbować, zgadywać, podejmować ryzyko. Jednak słabo sobie radzimy ze zgadywaniem. Dlatego aby skutecznie robić postępy, musimy tak zorganizować pracę, aby nieudane próby nie doprowadziły do katastrofy. Musimy pracować ostrożniej, bardziej defensywnie. Musimy robić małe kroki i ograniczyć zakres (lub „promień rażenia") domysłów, a także wyciągać wnioski z ich efektów. Wymaga to zastosowania podejścia iteracyjnego.

Gdy masz już pomysł, który chcesz zrealizować, musisz znaleźć sposób na określenie, kiedy się zatrzymać. Na jakiej podstawie wstrzymać prace nad złym rozwiązaniem? Gdy już stwierdzisz, że pomysł jest na tyle dobry, że warto zaryzykować i spróbować go wdrożyć, w jaki sposób ograniczyć „promień rażenia" i nie stracić wszystkiego w wyniku realizacji fatalnej idei? Konieczna jest możliwość jak najszybszego wykrywania kiepskich rozwiązań. Jeśli uda się je wyeliminować tylko na podstawie analizy, to świetnie. Jednak wiele pomysłów nie jest błędnych w oczywisty sposób. Sukces jest ulotny. Nawet dobry pomysł może dać kiepskie efekty, jeśli zostanie zrealizowany w nieodpowiednim czasie lub w zły sposób.

Trzeba znaleźć sposób na to, aby móc wypróbować ideę minimalnym kosztem, i jeśli okaże się zła, dowiedzieć się o tym szybko i stosunkowo tanio. W 2012 roku McKinsey we współpracy z Uniwersytetem Oksfordzkim przeprowadzili ankietę na temat projektów z branży oprogramowania. Okazało się, że 17% dużych projektów (o budżecie powyżej 15 milionów dolarów) zakończyło się tak poważną porażką, że zagrażało to przetrwaniu firm, które je realizowały. W jaki sposób można zidentyfikować tego rodzaju kiepskie pomysły? Jeśli pracujesz małymi krokami, uzyskujesz prawdziwe informacje zwrotne na temat postępów lub ich braku oraz stale sprawdzasz i oceniasz rozwiązania, to najszybciej i najniższym kosztem dowiesz się o tym, że prace toczą się niezgodnie z planami i oczekiwaniami. Jeśli pracujesz iteracyjnie małymi krokami, koszt wykonania jednego błędnego kroku jest zdecydowanie niższy, co pozwala ograniczyć ryzyko.

[3] Źródło: *Online Controlled Experiments at Large Scale*, *https://stanford.io/2LdjvmC*.

W książce *The Beginning of Infinity* David Deutsch opisuje istotną różnicę między ideami, które mają ograniczony zakres a tymi nieograniczonymi. Porównanie podejścia kaskadowego z jego ujętym w planie i zdefiniowanym procesem prac oraz podejścia iteracyjnego, które jest eksploracyjne i eksperymentalne, to zestawienie tego rodzaju dwóch fundamentalnie odmiennych modeli. Modele kontroli zdefiniowanych procesów[4] wymagają „zdefiniowanego procesu". Z definicji mają więc ograniczony zakres. Granicą tego podejścia jest na pewnym poziomie zdolność ludzkiego mózgu do uchwycenia szczegółów całego procesu. Nawet jeśli ktoś jest inteligentny i korzysta z abstrakcji oraz modułów do ukrycia niektórych szczegółów, to ostatecznie zdefiniowanie kompletnego procesu w ramach jakiegoś rodzaju planu wymaga uwzględnienia wszystkiego, co może się wydarzyć. Jest to z natury podejście ograniczone do rozwiązywania problemów. Pozwala ono rozwiązać tylko te problemy, który można od początku zrozumieć.

Podejście iteracyjne działa zupełnie inaczej. Można zacząć od miejsca, w którym nie wiadomo prawie nic, a mimo to robić wartościowe postępy. Można zacząć od jakiegoś prostego, zrozumiałego aspektu systemu i na tej podstawie określić, jak zespół powinien pracować, zbadać pierwsze pomysły dotyczące architektury systemu, wypróbować kilka technologii, które wydają się obiecujące, i tak dalej. Żaden z dokonanych wyborów nie musi być niezmienny. Nawet jeśli stwierdzisz, że dana technologia jest nieodpowiednia lub że pierwsza koncepcja architektury jest błędna, i tak oznacza to postępy. Wiesz teraz więcej niż wcześniej. Jest to z natury otwarty i nieskończony proces. O ile tylko istnieje jakiegoś rodzaju funkcja przystosowania, która pozwala stwierdzić, czy zbliżasz się do celu, czy się od niego oddalasz, możesz kontynuować pracę w tym modelu w nieskończoność — stale dopracowywać, wzbogacać i rozwijać wiedzę, idee, umiejętności oraz produkty. Możesz nawet zdecydować się na zmianę funkcji przystosowania w trakcie prac, jeśli uznasz, że istnieją lepsze cele, do których warto dążyć.

Początek nieskończoności

W poszerzającej horyzonty książce *The Beginning of Infinity* fizyk David Deutsch opisuje naukę i oświecenie jako poszukiwanie „dobrych wyjaśnień". Pokazuje też, że różne idee w historii ludzkości reprezentują „początek nieskończoności" i pomagają nam radzić sobie z wszelkimi wyobrażalnymi zastosowaniami tych dobrych wyjaśnień.

Dobrym przykładem jest tu różnica między alfabetem a piktograficznymi systemami pisma.

Ludzkość początkowo posługiwała się systemami piktograficznymi. W językach chińskim i japońskim nadal część znaków to piktogramy. Są one piękne, jednak mają poważną wadę. Jeśli usłyszysz nowe słowo, nie będziesz wiedzieć, jak je zapisać, dopóki ktoś Ci tego nie pokaże. Piktograficzne systemy pisma nie są inkrementalne. Musisz znać odpowiednie symbole dla wszystkich słów. W piśmie chińskim istnieje mniej więcej 50 000 znaków.

[4] Ken Schwaber opisał podejście kaskadowe jako „model kontrolowania zdefiniowanego procesu". Oto jego słowa: „Model kontrolowania zdefiniowanego procesu wymaga, aby każdy fragment prac był w pełni zrozumiały. Gdy zestaw danych wejściowych jest dobrze zdefiniowany, za każdym razem generowane są te same dane wyjściowe. Zdefiniowany proces można uruchomić, a następnie wykonywać do momentu ukończenia i za każdym razem otrzymać te same wyniki". Schwaber porównuje to podejście do „modelu kontroli procesu empirycznego" reprezentowanego przez podejście zwinne. Zobacz: *https://bit.ly/2UiaZdS*.

Alfabet działa zupełnie inaczej. Koduje dźwięki, a nie słowa. Możesz zapisać dowolne słowo (nawet niepoprawnie) w taki sposób, że każdy zrozumie Twoje intencje.

Jest to prawda nawet wtedy, jeśli nigdy wcześniej nie zdarzyło Ci się usłyszeć danego słowa ani zobaczyć jego zapisu.

Możesz też przeczytać słowo, którego nie znasz, a nawet wyraz, którego nie rozumiesz lub który nie potrafisz poprawnie wymówić. Pismo piktograficzne tego nie umożliwia. To oznacza, że alfabet daje nieskończony zakres możliwości, natomiast pismo piktograficzne jest ograniczone. Pierwsze z tych podejść jest skalowalnym podejściem do reprezentowania idei, a drugie już nie.

Idea nieskończonego zasięgu lub zakresu jest zgodna ze zwinnym podejściem do rozwoju oprogramowania, natomiast nie przystaje do podejścia kaskadowego.

Podejście kaskadowe jest sekwencyjne. Trzeba odpowiedzieć na pytania na danym etapie i dopiero potem można przejść do następnego. To oznacza, że w tym modelu nawet bardzo inteligentne osoby w jakimś momencie natrafią na granicę, poza którą złożoność całego systemu będzie przekraczać możliwości jego zrozumienia.

Ludzkie zdolności umysłowe są ograniczone, jednak niekoniecznie dotyczy to naszej zdolności rozmienia. Możemy radzić sobie z fizjologicznymi granicami mózgu dzięki stosowaniu odpowiednich technik. Można na przykład stosować abstrakcje i dzielić system na fragmenty (moduły), aby podnieść możliwości zrozumienia go do niezwykłego poziomu.

Zwinne podejście do rozwoju oprogramowania ma aktywnie zachęcać do tego, by rozpoczynać pracę nad rozwiązywaniem problemów w małych porcjach. Zachęca też do rozpoczynania pracy, zanim poznamy odpowiedzi na wszystkie pytania. Ten model umożliwia robienie postępów, choćby przez wykonywanie nieoptymalnych lub nawet złych kroków, ponieważ każdy etap pozwala dowiedzieć się czegoś nowego.

W ten sposób można doprecyzować myślenie, zidentyfikować następny mały krok i wykonać go. Model zwinny jest nieograniczonym, nieskończonym podejściem, ponieważ polega na pracy nad małymi porcjami problemu przed ruszeniem dalej ze znanego i rozumianego miejsca. Jest to dużo bardziej organiczne, ewolucyjne i nieograniczone podejście do rozwiązywania problemów.

Opisana różnica jest bardzo istotna i wyjaśnia, dlaczego myślenie zwinne stanowi ważny krok naprzód, jeśli chodzi o możliwość robienia postępów w rozwiązywaniu — w idealnym scenariuszu — coraz to trudniejszych problemów.

Nie oznacza to, że myślenie zwinne jest idealną lub ostateczną techniką. Jest to jednak ważny duży krok w kierunku bardziej wydajnej pracy.

Pokusa tworzenia planu jest zwodnicza. Plan nie prowadzi do bardziej starannego, kontrolowanego i profesjonalnego podejścia. Taki model jest natomiast ograniczony i w dużym stopniu oparty na przeczuciach i domysłach. W praktyce sprawdzi się tylko dla niewielkich, prostych, zrozumiałych i dobrze zdefiniowanych systemów.

Ma to istotne implikacje. Oznacza, że musimy, jak ujął to Kent Beck w znanym podtytule swojej przełomowej pracy *Extreme Programming Explained*, „zaakceptować zmiany".

Musimy nauczyć się odważnie rozpoczynać projekt w momencie, w którym jeszcze nie znamy odpowiedzi i kiedy nie wiemy, ile pracy będzie on wymagać. Dla niektórych osób i firm jest to trudne do zaakceptowania, jednak niczym nie różni się od realiów dużej części ludzkiej egzystencji. Gdy firma rozpoczyna nowy projekt, nie wie, kiedy, a nawet czy w ogóle, przyniesie on korzyści. Nie wie, ilu ludziom spodoba się dany produkt i czy będą chcieli za niego zapłacić.

Nawet w przypadku czegoś tak prostego jak podróż samochodem nie można mieć pewności, ile czasu ona zajmie lub czy wybrana trasa okaże się najlepsza, gdy już ruszysz w drogę. Obecnie dostępne są fantastyczne narzędzia, takie jak satelitarne systemy nawigacji korzystające z łączności radiowej. Takie systemy potrafią nie tylko zaplanować trasę przed podróżą, ale też iteracyjnie aktualizować ją na podstawie informacji o ruchu. Pozwala to „badać" zmieniające się warunki na drodze i „dostosowywać się" do nich.

Iteracyjne podejście do tworzenia planów i ich realizowania sprawia, że stale masz możliwie aktualny obraz sytuacji zamiast jakiejś prognozowanej, teoretycznej i zawsze niepoprawnej wizji. Dzięki temu możesz się uczyć, reagować i adaptować do zmian. Podejście iteracyjne jest jedyną skuteczną strategią w zmiennym środowisku.

Praktyczne aspekty podejścia iteracyjnego

Jak więc wygląda praca w tym modelu? Przede wszystkim należy zajmować się mniejszymi porcjami pracy. Trzeba ograniczyć zakres wszystkich zmian i wprowadzać modyfikacje w mniejszych krokach. Ogólnie można przyjąć, że im mniejsze porcje, tym lepiej. Dzięki temu możesz częściej wypróbowywać różne techniki, idee i technologie.

Praca w niewielkich porcjach oznacza też, że ograniczasz horyzont czasowy, w jakim przyjęte założenia muszą być prawdziwe. Wszechświat ma wtedy mniejsze okno czasowe, w którym może zakłócić Twoją pracę, dzięki czemu zmniejszasz ryzyko szkodliwych zmian spowodowanych zewnętrznymi czynnikami. Ponadto jeśli wykonujesz niewielkie kroki, to nawet jeśli któryś z nich okaże się błędny z powodu zmiennych okoliczności lub błędnego zrozumienia sytuacji, oznaczać to będzie mniejszą stratę. Dlatego wykonywanie małych kroków jest naprawdę istotne.

Oczywistym ucieleśnieniem tej idei w zespołach zwinnych są iteracje i sprinty. W podejściu zwinnym zaleca się tworzenie w krótkim, ustalonym czasie kompletnego kodu gotowego do zastosowania w środowisku produkcyjnym. Daje to wiele korzystnych efektów opisanych w tym rozdziale. Jest to jednak tylko jeden i to ogólny sposób realizacji podejścia iteracyjnego.

Na zupełnie innym poziomie można potraktować ciągłą integrację i programowanie sterowane testami jako z natury iteracyjne procesy.

W ciągłej integracji zmiany są zatwierdzane często, wielokrotnie w ciągu dnia. To oznacza, że każda zmiana musi być atomowa, nawet jeśli funkcja, której ona dotyczy, nie jest jeszcze gotowa. Zmienia to sposób pracy, ale daje więcej okazji do nauki i oceny, czy Twój kod nadal współdziała z kodem rozwijanym przez inne osoby.

Programowanie sterowane testami jest często opisywane za pomocą etapów używanych w tej technice: czerwony, zielony, refaktoryzacja.

- Czerwony: napisz test, uruchom go i sprawdź, czy kończy się niepowodzeniem.
- Zielony: napisz wystarczającą ilość kodu, aby test zakończył się powodzeniem. Uruchom test i sprawdź, czy kończy się powodzeniem.
- Refaktoryzacja: zmodyfikuj kod i test, aby były przejrzyste, zwięzłe, eleganckie i bardziej ogólne. Po każdej nawet drobnej zmianie uruchamiaj test i sprawdzaj, czy kończy się powodzeniem.

Jest to bardzo precyzyjne, iteracyjne podejście. Zachęca do stosowania wysoce iteracyjnych podstawowych technik pisania kodu.

Ja sam w trakcie pisania kodu prawie zawsze tworzę nowe klasy, zmienne, funkcje i parametry w wieloetapowym procesie niewielkich refaktoryzacji. W trakcie pracy często uruchamiam testy, aby sprawdzić, czy kod nadal działa.

Tak wygląda iteracyjna praca na bardzo szczegółowym poziomie. Dzięki temu podejściu wiadomo, że przez większość czasu kod jest poprawny i działa. Oznacza to, że każdy krok jest bezpieczniejszy.

W każdym miejscu procesu mogę dokonać ponownej oceny sytuacji i łatwo zmienić zdanie oraz kierunek prac nad projektem i kodem. Dzięki temu trzymam wszystkie opcje otwarte.

Opisane cechy sprawiają, że podejście iteracyjne jest tak przydatne i stanowi niezwykle ważną podstawową technikę inżynierii rozwoju oprogramowania.

Podsumowanie

Iteracje są ważną ideą i to dzięki nim można wdrożyć bardziej kontrolowany model uczenia się, odkrywania oraz rozwoju lepszego oprogramowania i produktów informatycznych. Jednak, jak zawsze, nie ma nic za darmo. Jeśli chcesz zastosować podejście iteracyjne, musisz zmienić wiele aspektów pracy, by to umożliwić.

Podejście iteracyjne ma wpływ na projekt rozwijanych systemów, organizację pracy i strukturę firm. Idea iteracji jest ściśle wpleciona w sposób myślenia zalecany w tej książce i prezentowany tu model inżynierii oprogramowania. Wszystkie omawiane tu pomysły są ze sobą mocno powiązane, choć czasem trudno jest ocenić, gdzie kończy się iteracja, a zaczynają informacje zwrotne.

PRZEKONAJ SIĘ, JAK DZISIAJ NAJLEPSI PRAKTYCY PROJEKTUJĄ OPROGRAMOWANIE!

Inżynieria oprogramowania polega na stosowaniu empirycznego podejścia do szukania wydajnych i ekonomicznie uzasadnionych rozwiązań podczas tworzenia oprogramowania. Dziedzina ta ma na celu organizowanie optymalnego procesu tworzenia aplikacji – od koncepcji, poprzez kodowanie, wdrażanie, utrzymywanie, po wycofywanie. Uzyskanie biegłości w tym obszarze nie jest łatwe: rozwój oprogramowania wymaga zrozumienia filozofii inżynierii i stosowania określonych zasad.

Tę książkę docenią programiści, menedżerowie, inżynierowie i liderzy techniczni. Znajdziesz w niej cenne informacje o filozofii inżynierii oprogramowania, jak również o postępach w sposobie myślenia na jej temat. Na tej bazie oparto zestaw zasad ułatwiających skuteczne radzenie sobie z dwoma podstawowymi procesami inżynierii oprogramowania: *uczenia się i eksploracji* oraz *radzenia sobie ze złożonością*. W ten sposób dowiesz się, jak usprawnić wszystkie aspekty swojej pracy, a także jak stosować sprawdzone podejścia prowadzące do sukcesu z uwzględnieniem uwarunkowań ekonomicznych. Dzięki tej przełomowej publikacji nauczysz się technik rozwiązywania problemów z wykorzystaniem zarówno obecnych, jak i przyszłych technologii. W efekcie będziesz szybciej tworzyć lepsze oprogramowanie, i to w bardziej przyjemny i satysfakcjonujący sposób.

Poznaj i stosuj zasady nowoczesnej inżynierii oprogramowania:

- określaj jasne cele i sensownie wybieraj narzędzia
- organizuj pracę i systemy tak, aby móc oceniać ciągłe postępy
- zachowuj kontrolę nawet po zwiększeniu złożoności systemu
- utrzymuj dyscyplinę z zachowaniem właściwej elastyczności
- ucz się z historii i doświadczenia
- naucz się oceniać nowe idee rozwoju oprogramowania

DAVID FARLEY jest wizjonerem i ekspertem w obszarze rozwijania oprogramowania; wiele razy podważał konwencjonalne myślenie w tej dziedzinie. Kierował zespołami tworzącymi oprogramowanie najwyższej klasy. Zbudował jedną z najszybszych na świecie giełd papierów wartościowych, jest autorem manifestu Reactive i laureatem Nagrody Duke'a za bibliotekę LMAX Disruptor.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-8322-594-4
9 788383 225944
Cena: 69,00 zł

Pearson
Addison-Wesley