Helion

Czysty Agile

Powrót do podstaw

Przedmowy: **Jerry Fitzpatrick, Tim Ottinger, Jeff Langr, Eric Crichlow, Damon Poole** *oraz* **Sandro Mancuso**

Robert C. Martin

Tytuł oryginału: Clean Agile: Back to Basics

Tłumaczenie: Wojciech Moch

ISBN: 978-83-283-6304-5

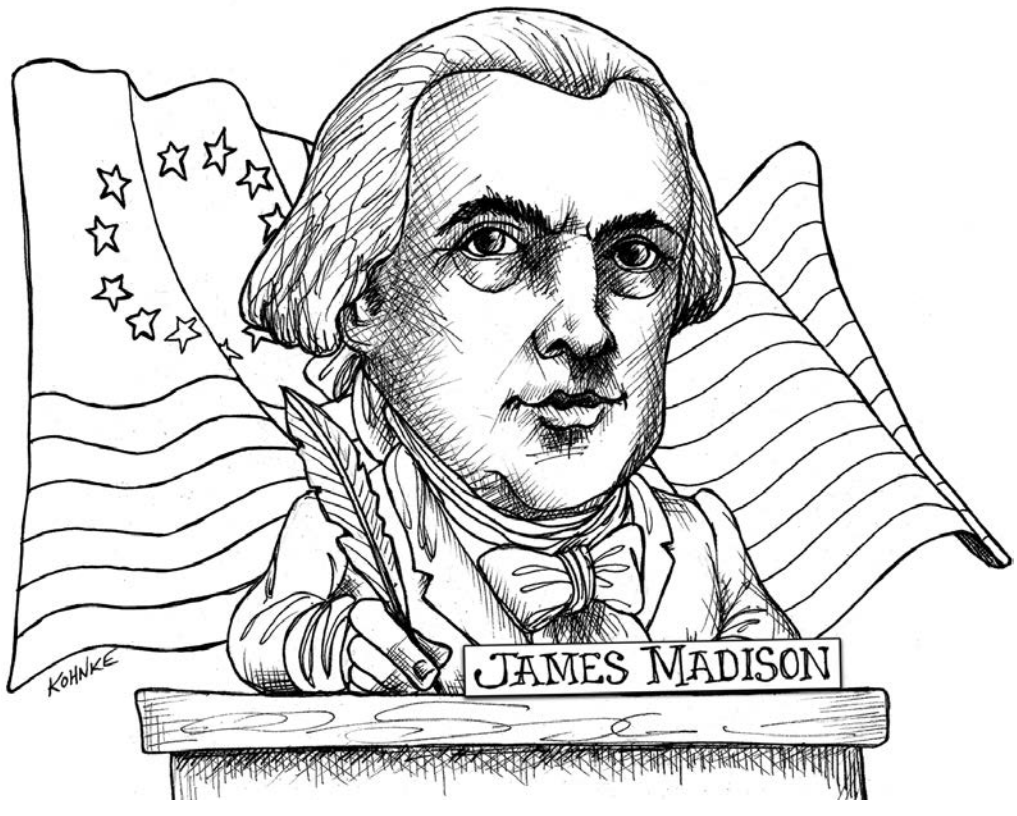
Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

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

Printed in Poland.

SPIS TREŚCI

2 Dlaczego Agile?



Zanim zagłębimy się w szczegóły zwinnego tworzenia oprogramowania, chciałbym wyjaśnić, o co toczy się gra. Zwinny rozwój oprogramowania jest ważny nie tylko dla samej branży, ale również dla całego przemysłu, społeczeństwa i cywilizacji.

Programiści i menedżerowie zaczynają stosować metody Agile z najróżniejszych powodów. Chcą je wypróbować, bo z jakiejś przyczyny wydają im się one poprawne, a być może zostają zachęceni obietnicami zwiększenia szybkości prac i poprawy jakości. Są to efekty niematerialne, trudno mierzalne i łatwo uznać, że się ich nie uzyskało. Wiele osób porzuciło zwinne metody tylko dlatego, że nie odczuło od razu tych efektów, jakie sobie obiecywało.

To nie te powody sprawiają jednak, że zwinne metody tworzenia oprogramowania są tak istotne. Waga tych metod wynika ze znacznie głębszych filozoficznych i etycznych przesłanek. Wiąże się z profesjonalizmem oraz rozsądnymi oczekiwaniami naszych klientów.

Profesjonalizm

Do Agile przyciągnęły mnie przede wszystkim mocny nacisk na dyscyplinę i niskie poważanie ceremonii. By prawidłowo stosować metody Agile, trzeba było pracować w parach, najpierw pisać testy, dokonywać refaktoryzacji i stosować proste rozwiązania. Należało pracować w krótkich cyklach, na końcu których trzeba było dostarczać działające oprogramowanie. Konieczne było regularne komunikowanie się z interesariuszami biznesowymi.

Spójrz ponownie na koło życia i przyjrzyj się znajdującym się na nim praktykom, traktując je jako *obietnice* lub *zobowiązania*. Postępując w ten sposób, dowiesz się, z jakiego środowiska się wywodzę. Z mojego punktu widzenia zwinne rozwijanie oprogramowania jest zobowiązaniem do ciągłego poprawiania się — do bycia profesjonalistą, do promowania profesjonalnych zachowań w całej branży zajmującej się tworzeniem oprogramowania.

W naszej branży bardzo potrzebujemy zwiększenia poziomu naszego profesjonalizmu. Zbyt często działamy nieprofesjonalnie. Tworzymy zbyt wiele produktów o marnej jakości. Akceptujemy zbyt wiele defektów. Decydujemy się na straszliwe kompromisy. Zbyt często zachowujemy się jak niesforne nastolatki zabawiające się kartą kredytową. W czasach, gdy było prościej, takie zachowania były tolerowane, ponieważ nie graliśmy o wysokie stawki. W latach 70., 80., a nawet 90. XX wieku koszt porażki projektu oprogramowania był ograniczony i choć wysoki, to jednak akceptowalny.

Oprogramowanie jest wszędzie

Dzisiaj wszystko się zmieniło.

Rozejrzyj się wokół siebie. Tak, właśnie teraz. Usiądź sobie i popatrz na swoje otoczenie. Ile komputerów znajduje się razem z Tobą w jednym pokoju?

No dobrze, ja zacznę. Aktualnie jestem w swoim domku letniskowym w północnych lasach stanu Wisconsin. Ile komputerów jest teraz wokół mnie?

- 4: Piszę ten tekst na komputerze MacBook Pro z czterema rdzeniami. Wiem, mówią, że jest ich osiem, ale wolę nie liczyć „wirtualnych" rdzeni. Poza tym nie liczę też wszystkich wspomagających procesorów umieszczonych w tym MacBooku.
- 1: Moja myszka Apple Magic Mouse2. Jestem pewien, że ukrywa się w niej więcej niż tylko jeden procesor, ale policzę jako jeden.
- 1: Mój iPad działający w trybie Duet jako drugi monitor. Wiem, że w iPadach umieszczono wiele innych małych procesorów, ale jego również policzę jako jeden.
- 1: Kluczyki do samochodu (!).
- 3: Słuchawki Apple AirPods. Po jednym na każde ucho i jeden w etui. Prawdopodobnie jest ich więcej, ale…
- 1: Mój iPhone. Tak, tak, rzeczywista liczba procesorów w iPhonie prawdopodobnie przekracza tuzin, ale ograniczmy się do jednego.
- 1: Ultradźwiękowy czujnik ruchu. (W domu jest ich więcej, ale z miejsca, gdzie siedzę, widzę tylko jeden).
- 1: Termostat.
- 1: Panel systemu bezpieczeństwa.
- 1: Telewizor LCD.
- 1: Odtwarzacz DVD.
- 1: Urządzenie do strumieniowania telewizji (Roku).
- 1: Apple TV.
- 5: Różne piloty.
- 1: Telefon (tak, standardowy telefon).
- 1: Sztuczny kominek. (Gdybyście mogli zobaczyć, w jakich trybach może pracować!).
- 2: Stary teleskop sterowany komputerem, model Meade LX 200 EMC. Jeden procesor sterujący napędem, a drugi w zewnętrznej jednostce sterującej.
- 1: Pendrive w mojej kieszeni.
- 1: Piórko Apple.

Naliczyłem przynajmniej 30 komputerów, które mam przy sobie lub są ze mną w pokoju. Ich rzeczywista liczba prawdopodobnie jest dwa razy większa, ponieważ większość urządzeń ma w sobie po kilka procesorów. Ale na tę chwilę pozostańmy przy 30.

A Tobie ile udało się naliczyć? Założę się, że większość osób również uzyska wartość podobną do 30. Co więcej, obstawiam, że większość spośród 1,3 miliarda ludzi żyjących w cywilizacji

zachodniej nieustannie ma w pobliżu siebie przynajmniej 12 komputerów. To zupełna nowość. Na początku lat 90. XX wieku ta liczba była niemalże równa zero.

Jaką wspólną cechę mają te wszystkie otaczające nas komputery? Wszystkie wymagają oprogramowania. Oprogramowania napisanego przez nas. A jak sądzisz, jak wygląda jakość tego oprogramowania?

Pozwól, że przedstawię to nieco inaczej. Ile razy dziennie Twoja babcia musi korzystać z oprogramowania? I tu informacja dla wszystkich mających jeszcze żyjące i sprawne babcie: być może tysiące razy, ponieważ w dzisiejszym świecie nie da się zrobić niczego, żeby nie natknąć się na oprogramowanie. Nie można na przykład:

- rozmawiać przez telefon,
- kupić ani sprzedać czegokolwiek,
- skorzystać z mikrofalówki, lodówki ani nawet z tostera,
- wyprać lub wysuszyć ubrań,
- umyć naczyń,
- posłuchać muzyki,
- prowadzić samochodu,
- złożyć podania o wypłatę ubezpieczenia,
- zwiększyć temperatury w pokoju,
- obejrzeć telewizji.

Jest nawet gorzej. Dzisiaj w naszym społeczeństwie praktycznie niczego ważnego nie można zrobić bez kontaktu z jakimś systemem programowym. Nie można ustanowić żadnego prawa ani monitorować jego przestrzegania. Nie da się debatować nad planami rządu. Nie da się sterować samolotem. Nie da się kierować samochodem. Nie można wystrzelić żadnego pocisku rakietowego. Nie można płynąć statkiem. Nie da się budować dróg, zbierać zboża z pól. Huty nie mogą produkować stali. Fabryki samochodów nie są w stanie produkować aut. Producenci słodyczy nie mogą tworzyć swoich produktów. Nie można obracać akcjami…

W naszym społeczeństwie bez oprogramowania nie da się już zrobić zupełnie niczego. Każde nasze działanie jest jakoś powiązane z oprogramowaniem. A wielu z nas używa też oprogramowania do monitorowania swojego snu.

Rządzimy światem

Nasz świat stał się całkowicie i nieodwracalnie uzależniony od oprogramowania. Oprogramowanie jest krwią krążącą w naszym społeczeństwie. Bez niego znana nam cywilizacja nie byłaby możliwa.

A kto pisze te wszystkie programy? Ty i ja. My, programiści, rządzimy tym światem.

Inni ludzie sądzą, że to oni rządzą światem, ale oni tylko przekazują nam ustalone przez siebie reguły, a my zapisujemy rzeczywiste zasady, według których działają maszyny monitorujące i kontrolujące niemal każdy aspekt nowoczesnego życia.

To my, programiści, rządzimy tym światem.

I robimy to w naprawdę fatalny sposób.

Jak sądzisz, jaka część oprogramowania, które steruje absolutnie wszystkim, została odpowiednio przetestowana? Ilu programistów może powiedzieć, że dysponuje zestawem testów *dowodzących* z dużym prawdopodobieństwem, że tworzone przez nich oprogramowanie działa?

Czy sto milionów wierszy kodu napędzających Twój samochód rzeczywiście działa? Udało Ci się znaleźć w nich jakieś błędy? Mnie się udało. A co z kodem odpowiadającym za działanie hamulców, pedału gazu i kierownicy? Czy w nim też są jakieś błędy? Czy istnieje zbiór testów, które można uruchomić w dowolnym momencie, aby *udowodnić* z dużym prawdopodobieństwem, że po naciśnięciu nogą pedału hamulca samochód się naprawdę zatrzyma?

Ilu ludzi zginęło, ponieważ oprogramowanie w ich samochodach nie przekazało informacji o nacisku stopy kierowcy na pedał gazu do systemu hamulcowego? Nikt tego nie wie na pewno, ale z pewnością *wielu*. W jednym z przypadków z 2013 roku Toyota musiała zapłacić miliony dolarów odszkodowania, ponieważ w oprogramowaniu samochodów tego producenta występowały: „błędy umożliwiające przełączanie bitów, awarie zadań doprowadzające do wyłączenia zabezpieczeń, umożliwiające zniszczenie zawartości pamięci, pojedyncze punkty awarii, niewystarczająca ochrona przed przepełnieniem stosu lub bufora, nieodpowiednie regiony zabezpieczenia przed awariami [oraz] tysiące zmiennych globalnych", a to wszystko ukryte w tak zwanym „kodzie spaghetti"[1].

Nasze oprogramowanie zabija ludzi. Ani ja, ani Ty nie zajęliśmy się tworzeniem oprogramowania, żeby zabijać ludzi. Wielu z nas zostało programistami, ponieważ jako dzieci napisaliśmy nieskończoną pętlę wypisującą na ekranie nasze imiona i uznaliśmy, że to świetna zabawa. Teraz jednak nasze działania mogą nieść zagrożenie dla życia i majątku ludzi. I każdego dnia coraz więcej powstającego kodu może powodować zagrożenie dla kolejnych osób i ich majątków.

[1] Safety Research & Strategies Inc., *Toyota unintended acceleration and the big bowl of „spaghetti" code*, wpis na blogu, 7.11.2013. Dostępny pod adresem *http://www.safetyresearch.net/blog/articles/toyota--unintended-acceleration-and-big-bowl-%E2%80%9Cspaghetti%E2%80%9D-code*.

Katastrofa

Nadejdzie w końcu dzień (o ile nie nadszedł, zanim udało Ci się przeczytać te słowa), w którym jakiś głupi programista zrobi coś naprawdę idiotycznego i przez chwilę swojej nieuwagi zabije dziesięć tysięcy osób. Zastanów się nad tym przez chwilkę. Wymyślenie przynajmniej kilku takich scenariuszy naprawdę nie jest trudne. A gdy to się stanie, to politycy z całego świata uniosą się pełni oburzenia (powinni to zrobić) i swoimi palcami wskażą właśnie nas.

Możesz sądzić, że tymi palcami będą pokazywać naszych szefów albo prezesów naszych firm, ale wszyscy widzieliśmy, co się stało, gdy te palce wskazywały prezesa amerykańskiego oddziału firmy Volkswagen zeznającego przed Kongresem. Politycy zapytali go, dlaczego Volkswagen umieścił w swoich samochodach oprogramowanie, które celowo wykrywało stanowiska do badania poziomu emisji spalin używane w stanie Kalifornia i fałszowało wynik. Odpowiedział wtedy: „Z mojego punktu widzenia i według mojej wiedzy to nie była decyzja korporacji. To dwóch programistów ze znanych tylko sobie powodów przygotowało takie oprogramowanie”[2].

I w ten sposób palce zaczną pokazywać nas. Prawidłowo! Ponieważ to nasze palce na klawiaturach i nasz brak dyscypliny, a także nasza beztroska okażą się główną przyczyną katastrofy.

To właśnie z tego powodu tak wielkie nadzieje pokładałem w Agile. Już wtedy wierzyłem i wierzę nadal, że narzucenie sobie dyscypliny w tworzeniu oprogramowania wymaganej przez metody Agile będzie pierwszym krokiem w kierunku zmiany pracy programisty w realną i szanowaną profesję.

Rozsądne oczekiwania

W dalszej części tego podrozdziału przedstawię całkowicie rozsądną listę oczekiwań, które menedżerowie, użytkownicy i klienci powinni mieć wobec nas. Czytając tę listę, zauważ, że jedna połówka Twojego mózgu będzie się zgadzać z tym, że każdy z tych punktów jest całkowicie uzasadniony. Zwróć też uwagę, że druga połówka Twojego mózgu — połówka programistyczna — będzie reagowała na nie przerażeniem. Programistyczna część Twojego mózgu nie jest w stanie sobie wyobrazić, że takie oczekiwania można spełnić.

A jednak zaspokojenie tych oczekiwań jest jednym z podstawowych celów stosowania zwinnych metod rozwijania oprogramowania. Zasady i praktyki związane z metodami Agile niemal bezpośrednio odwołują się do wszystkich oczekiwań z mojej listy. Każdy dobry dyrektor

[2] S. O’Kane, *Volkswagen America’s CEO blames software engineers for emissions cheating scandal*, „The Verge”, 8.10.2015. Dostępny pod adresem *https://www.theverge.com/2015/10/8/9481651/volkswagen-congressional-hearing-diesel-scandal-fault*.

techniczny (ang. CTO — *chief technology oficer*) powinien wymagać od swoich pracowników przedstawionych poniżej zachowań. Żeby było dosadniej, wyobraź sobie, że to ja jestem Twoim dyrektorem technicznym. I tego oczekuję od Ciebie.

Zakaz dostarczania podrzędnych produktów!

To, że w ogóle muszę wspomnieć o tego rodzaju oczekiwaniach, pokazuje, jak tragiczna jest sytuacja w naszej branży. Niestety muszę. I jestem pewien, że wielu spośród moich czytelników już wielokrotnie nie było w stanie spełnić tego oczekiwania. Mnie się to przytrafiło.

Aby zrozumieć, jak poważny to problem, zastanów się nad awarią systemu kontroli lotów nad Los Angeles wywołaną przepełnieniem 32-bitowego licznika. Albo nad wyłączeniem generatorów zamontowanych na pokładzie samolotów Boeing 787, również spowodowanym przepełnieniem licznika. Albo nad setkami osób zabitych przez błąd w oprogramowaniu systemu MCAS samolotów 737 MAX.

A może przyjrzyjmy się moim doświadczeniom z systemem *healthcare.gov* zaraz po jego uruchomieniu? Po pierwszym zalogowaniu, podobnie jak wiele innych dzisiejszych systemów, poprosił mnie on o przygotowanie zestawu pytań zabezpieczających. Jedno z tych pytań mówiło o „ważnej dla ciebie dacie". Wprowadziłem datę mojego ślubu, czyli `7/24/73`. System odpowiedział mi błędem `Nieprawidłowe dane`.

Jestem programistą. Wiem, jak myślą programiści. Spróbowałem zatem innych formatów daty: `07/21/1973`, `07-21-1973`, `21 July, 1973`, `07211973` itd. Zawsze w odpowiedzi dostawałem ten sam błąd: `Nieprawidłowe dane`. To było niezwykle frustrujące. Jakiego formatu daty wymagał ten przeklęty system?

I wtedy doznałem olśnienia. Programista piszący tę funkcję systemu nie wiedział, na jakie pytania będziemy odpowiadać. Po prostu odczytywał pytania z bazy danych i zapisywał tam odpowiedzi. Bardzo prawdopodobne, że nie pozwalał na stosowanie w odpowiedzi żadnych znaków specjalnych, a może i cyfr. Wpisałem zatem `Rocznica ślubu`. I ta odpowiedź została przyjęta.

Uważam, że każdy zgodzi się ze mną, że dowolny system zmuszający użytkowników do zastanawiania się nad tokiem rozumowania programistów przy wprowadzaniu danych należy uznać za wielką katastrofę.

Mógłbym tu podać całą listę anegdotek o różnych fatalnie przygotowanych programach. Ale inni już to zrobili — znacznie lepiej ode mnie. Jeżeli chcesz dowiedzieć się, z jak wielkim

problemem mamy do czynienia, to zainteresuj się książką Gojko Adzica *Humans vs. Computers*[3] oraz książką Matta Parkera *Humble Pi*[4].

Całkiem rozsądnym żądaniem naszych menedżerów, klientów i użytkowników jest to, że oczekują od nas systemów, które będą miały bardzo dobrą jakość i niewiele błędów. Nikt nie chce korzystać z fatalnego systemu, szczególnie jeśli zapłacono za niego wysoką cenę.

Zwróć uwagę na to, że w metodach Agile kładzie się nacisk na testowanie, refaktoryzację, proste projekty i kontakty z klientami. To oczywisty sposób na zapobieganie powstawaniu złego kodu.

Ciągła gotowość techniczna

Ostatnią rzeczą, jakiej będą oczekiwać nasi klienci i menedżerowie, będzie to, że my, programiści, będziemy sztucznie opóźniać wydanie systemu. Niestety takie sztuczne opóźnienia są bardzo powszechne w pracy zespołów programistycznych. Ich powodem często jest chęć zbudowania od razu wszystkich funkcji systemu. Dopóki w systemie istnieją funkcje gotowe w połowie, w połowie przetestowane lub w połowie udokumentowane, nie można przygotować oficjalnego wydania.

Kolejnym źródłem sztucznych opóźnień jest potrzeba stabilizacji. Często zespoły korzystają ze specjalnych okresów ciągłego testowania, w czasie których obserwują tworzony system, sprawdzając, czy nie pojawią się jakieś błędy. Jeżeli po X dniach nie zostaną znalezione żadne błędy, to programiści czują się na tyle bezpiecznie, że mogą przekazać system do wdrożenia.

Agile rozwiązuje ten problem, wprowadzając prostą regułę, według której na zakończenie każdej iteracji system musi być *technicznie* gotowy do wdrożenia. Techniczna gotowość oznacza, że z punktu widzenia programistów system jest na tyle solidnie zbudowany, że możliwe jest jego wdrożenie. Kod systemu jest czysty, a wszystkie testy zostały zaliczone.

To wszystko znaczy, że praca wykonana w danej iteracji musi obejmować nie tylko przygotowanie kodu i jego testów, ale również dokumentację oraz stabilizację dla wszystkich historii zaimplementowanych w tej iteracji.

Jeżeli na końcu iteracji system jest *technicznie* gotowy do wdrożenia, to decyzja o jego wdrożeniu jest decyzją *biznesową*, a nie techniczną. To interesariusze biznesowi mogą zdecydować, że system nie ma wystarczającej liczby funkcji, by można było go wdrożyć. Mogą też opóźnić wdrożenie z powodów rynkowych lub z powodu konieczności

[3] G. Adzic, *Humans vs. Computers*, Neuri Consulting LLP, Londyn 2017. Dostęp przez stronę *http://humansvscomputers.com*.

[4] M. Parker, *Humble Pi: A Comedy of Maths Errors*, Penguin Random House UK, Londyn 2019. Dostęp przez stronę *https://mathsgear.co.uk/products/humble-pi-a-comedy-of-maths-errors*.

przeprowadzenia szkoleń. Niezależnie od tego, jakość systemu zawsze spełnia *techniczne* wymagania dla wdrożenia.

Czy to możliwe, żeby system co tydzień lub dwa był technicznie gotowy do wdrożenia? Oczywiście, że tak. Zespół musi jedynie wybrać na tyle mały zestaw historii, by można było wykonać wszystkie zadania związane z uzyskaniem gotowości do wdrożenia, mieszcząc się w jednej iteracji. I najlepiej byłoby, gdyby większość zadań związanych z testowaniem została zautomatyzowana.

Z punktu widzenia interesariuszy biznesowych i klientów ciągła gotowość techniczna jest czymś naturalnym. Gdy interesariusze widzą, że jakaś funkcja działa, to automatycznie zakładają, że jest ona gotowa. Nie spodziewają się, iż usłyszą, że trzeba poczekać jeszcze miesiąc na ukończenie testów i stabilizacji. Nie spodziewają się, że funkcja działa tylko dlatego, że prezentujący ją programiści sprytnie pominęli te elementy, które nie działają.

Stabilna produktywność

Z pewnością nietrudno zauważyć, że nowe zespoły programistów potrafią pracować bardzo wydajnie w trakcie pierwszych kilku miesięcy trwania nowego projektu. Jeżeli nie istnieje żaden kod, który mógłby nas spowalniać, możemy w krótkim czasie napisać wiele nowego działającego kodu.

Niestety z czasem bałagan w kodzie zaczyna się kumulować. Jeżeli kod systemu nie jest porządkowany i oczyszczany na bieżąco, to zespół zacznie odczuwać presję, która będzie spowalniać dalsze prace. Im większy bałagan w kodzie, tym większa presja i tym wolniej pracuje zespół. Im wolniej pracuje zespół, tym bardziej zwiększa się presja związana z koniecznością dotrzymania terminów z harmonogramu, a wraz z nią rośnie pokusa wprowadzenia jeszcze większego bałaganu. Ta pętla może doprowadzić do sytuacji, że zespół nie będzie w stanie wykonać żadnej pracy.

Zaskoczeni takim spowolnieniem menedżerowie mogą w końcu zdecydować o dołączeniu do zespołu nowych ludzi, żeby zwiększyć jego produktywność. Jak jednak dowiedziałeś się w poprzednim rozdziale, zatrudnianie nowych ludzi poważnie spowalnia prace zespołu na kilka tygodni.

Zawsze istnieje nadzieja, że po upływie tych kilku tygodni nowi ludzie będą w stanie zacząć skutecznie pracować, przez co zwiększy się prędkość zespołu. Pamiętajmy jednak, kto szkoli nowych pracowników. Te same osoby, które już wcześniej pozwoliły na powstanie tego całego bałaganu. Oznacza to, że nowi pracownicy niemal na pewno będą powtarzać zachowania dotychczasowych członków zespołu.

Co gorsza, istniejący kod bywa skutecznym instruktorem. Nowi ludzie będą przeglądać stary kod i uczyć się metod pracy swojego zespołu. W ten sposób utrwalane będą zachowania

prowadzące do powstania bałaganu. I tak produktywność będzie nadal spadać mimo dołączenia nowych pracowników do zespołu.

Kadra zarządzająca będzie próbowała tego samego rozwiązania jeszcze kilka razy, ponieważ w niektórych organizacjach definicją zdrowego rozsądku menedżerów jest powtarzanie tych samych zachowań i oczekiwanie innych rezultatów. Ostatecznie mimo wszystko prawda wyjdzie na jaw. Żadne działania menedżerów nie są w stanie powstrzymać spadku produktywności zespołu. Będzie się zmniejszać aż do zera.

W akcie desperacji menedżerowie zapytają programistów, co można zrobić, żeby ponownie zwiększyć produktywność. A programiści znają odpowiedź na to pytanie. Już od jakiegoś czasu wiedzieli, co trzeba zrobić. Czekali tylko, aż ktoś o to zapyta.

Programiści powiedzą wtedy: „Trzeba przeprojektować system od zera".

Wyobraź sobie przerażenie menedżerów. Wyobraź sobie, ile pieniędzy i czasu do tej pory zainwestowano w system. A teraz programiści proponują, żeby całą tę pracę wyrzucić do śmieci i zacząć wszystko od zera!

Czy menedżerowie uwierzą programistom twierdzącym, że „tym razem wszystko będzie inaczej"? Oczywiście, że nie. Musieliby być głupcami. Ale jaki mają wybór? Produktywność zespołu jest praktycznie zerowa. W ten sposób nie da się prowadzić biznesu. Dlatego po wielu narzekaniach, zaciskając zęby, zgadzają się na przeprojektowanie systemu.

I programiści zaczynają świętowanie. „Hura! Wracamy do początków, gdzie życie jest przyjemne, a kod jest czysty!". Oczywiście nic takiego się nie dzieje. Bardzo szybko okazuje się, że zespół zostaje podzielony na dwa. Dziesięciu najlepszych — zespół tygrysów, czyli ludzi, którzy wytworzyli początkowy bałagan — zostaje wydzielonych i przeniesionych do osobnego pokoju. To oni mają prowadzić pozostałych na drodze do wspaniałej krainy przeprojektowanego systemu. Pozostali są na nich wściekli, ponieważ ich zadaniem jest utrzymywanie starego bałaganu.

Skąd zespół tygrysów ma pobierać wymagania dla nowego systemu? Czy istnieje dokument, w którym zapisana jest aktualna lista wymagań? Owszem. To kod starego systemu. Stary kod jest jedynym dokumentem dokładnie opisującym to, co powinien robić nowy system.

A zatem zespół tygrysów zaczyna przeglądać stary kod, próbując zrozumieć, co on właściwie robi, i na tej podstawie przygotować nowy projekt. W tym czasie pozostali programiści ciągle zmieniają stary kod, naprawiając w nim błędy i dodając nowe funkcje.

I tak zaczyna się wyścig. Zespół tygrysów cały czas próbuje trafić w poruszający się cel. Jak wykazał Zenon z Elei w swojej opowieści o Achillesie i żółwiu, dogonienie poruszającego się celu jest prawdziwym wyzwaniem. Za każdym razem gdy zespół tygrysów dociera do miejsca, w którym był stary system, okazuje się, że ten już jakiś czas temu przeniósł się w inne miejsce.

Wykazanie, że Achilles ostatecznie wyprzedzi żółwia, wymaga wykonania odpowiednich obliczeń. W przypadku oprogramowania takie obliczenia nie zawsze się sprawdzają. Pracowałem w firmie, w której po dziesięciu latach prac nowy system ciągle nie był gotowy do wdrożenia, mimo że osiem lat wcześniej obiecano klientom jego stworzenie. Niestety nowy system nigdy nie miał wystarczającej liczby funkcji, a stary system cały czas był w stanie zrobić więcej niż nowy. W takiej sytuacji klienci odmawiali przesiadki na nowy system.

Po kilku latach klienci zaczęli po prostu ignorować obietnicę, że kiedyś dostaną nowy system. Z ich punktu widzenia taki system nigdy nie istniał i nigdy istnieć nie będzie.

W tym czasie firma opłacała dwa zespoły programistów: zespół tygrysów i zespół utrzymujący stary system. Ostatecznie zarząd firmy zdecydował się na radykalny krok, mówiąc klientom, że nowy system zostanie wdrożony pomimo ich obiekcji. Klienci wszczęli z tego powodu awanturę, ale prawdziwą burzę wywołali programiści z zespołu tygrysów, choć powinienem raczej powiedzieć — ci, którzy zostali z tego zespołu. Pierwotni członkowie zespołu tygrysów niemal co do jednego otrzymali awans i piastowali pozycje menedżerskie. Z kolei nowi członkowie tego zespołu jednym głosem stwierdzili, że: „Tego nie można przeznaczyć do wdrożenia, bo to kupa gnoju. Całość trzeba przeprojektować".

OK, masz rację. To kolejna przerysowana historia opowiedziana przez Wujka Boba. Ta opowieść bazuje na prawdziwych wydarzeniach, choć trochę ją podkoloryzowałem dla zwiększenia efektu. Mimo to jej przesłanie jest prawdziwe. Realizacja idei wielkiego przeprojektowania systemu zawsze jest niezwykle droga, a nowe systemy prawie nigdy nie mogą się doczekać wdrożenia.

Klienci i menedżerowie nie spodziewają się, że zespoły programistów będą z czasem pracowały coraz wolniej. Oczekują raczej, że skoro tworzenie pewnej funkcji zajęło dwa tygodnie na początku projektu, to zaimplementowanie podobnej funkcji rok później również zajmie dwa tygodnie. Wszyscy spodziewają się, że produktywność będzie wartością stałą w czasie.

Sami programiści nie powinni oczekiwać niczego innego. Ciągłe starania o zachowanie jak największej czystości architektury i kodu pozwalają na utrzymywanie wysokiej produktywności i zapobieganie powstaniu spirali spadającej produktywności prowadzącej do pojawienia się chęci przeprojektowania systemu.

Jak wkrótce się przekonasz, takie zwinne praktyki jak testowanie, praca w parach, refaktoryzacja i proste projekty są kluczowymi elementami pozwalającymi na przerwanie tej spirali. Z kolei gra w planowanie jest antidotum na presję związaną z koniecznością dotrzymania terminów z harmonogramu, która bardzo skutecznie napędza tę spiralę.

Niedroga zdolność do adaptacji

Angielskie słowo „software" jest złożeniem dwóch słów. Słowo „ware" oznacza produkt, natomiast słowo „soft" oznacza łatwość wprowadzania zmian. To znaczy, że software, czyli

oprogramowanie, jest produktem, który można łatwo zmieniać. Oprogramowanie zostało wynalezione dlatego, że chcieliśmy uzyskać możliwość łatwego i szybkiego zmieniania zachowań naszych maszyn. Gdybyśmy chcieli, żeby zmiany takich zachowań były trudne, to używalibyśmy słowa „hardware".

Programiści często narzekają na zmieniające się wymagania wobec produktu. Nierzadko słyszałem takie stwierdzenia jak: „Ta zmiana całkowicie odwraca naszą architekturę". No to mam dla Ciebie wiadomość, misiu. Jeżeli jakaś zmiana wymagań systemu powoduje niezgodność z architekturą, to znaczy, że ta architektura jest do bani.

Jako programiści powinniśmy cieszyć się ze zmian, ponieważ zmiany są istotą naszego zawodu. Zmiany wymagań systemu są podstawą całej tej zabawy. To właśnie takie zmiany są bazą dla naszych karier i naszych pensji. Nasza praca wymaga od nas zdolności do przyjmowania i przetwarzania zmieniających się wymagań, tak żeby ich wprowadzenie było względnie niedrogie.

Jeżeli wprowadzanie zmian do oprogramowania przygotowanego przez dany zespół jest trudne, to znaczy, że ten zespół zaprzecza samej przyczynie istnienia oprogramowania. Klienci, użytkownicy i menedżerowie oczekują, że systemy programowe będą łatwo poddawały się zmianom, a koszt takich zmian będzie niewielki i proporcjonalny do wielkości wprowadzanej zmiany.

Wykażę tutaj, że takie praktyki Agile jak TDD, refaktoryzacja i proste projekty współgrają ze sobą, a ich wykorzystanie umożliwia bezpieczne wprowadzanie zmian do systemów programowych przy minimalnym nakładzie pracy.

Ciągłe usprawnienia

Ludzie z czasem stają się lepsi w tym, co robią. Malarze tworzą coraz lepsze obrazy, piosenkarze coraz lepiej śpiewają, a właściciele domów cały czas poprawiają swoje siedziby. To samo powinno dotyczyć też oprogramowania. Systemy programowe powinny być im starsze, tym *lepsze*.

Projekt i architektura systemu programowego powinny poprawiać się z czasem. Poprawie powinna ulegać też struktura powstającego kodu, podobnie jak jego wydajność i przepustowość. Czy to nie oczywiste? Czy nie tego oczekiwalibyśmy od dowolnej grupy ludzi pracujących nad wspólnym projektem?

To najważniejsze oskarżenie wobec branży tworzenia oprogramowania. To najbardziej dobitny dowód naszej porażki. Nasze produkty z czasem stają się coraz gorsze. To, że jako programiści spodziewamy się, że nasze systemy będą z czasem stawały się coraz bardziej bałaganiarskie, coraz brzydsze, delikatniejsze i bardziej kruche, jest oznaką naszego ogromnego braku odpowiedzialności.

Użytkownicy, klienci i menedżerowie oczekują od nas ciągłych powolnych usprawnień. Oczekują, że początkowe problemy zostaną rozwiązane, a system będzie już tylko lepszy i lepszy. Takim oczekiwaniom możemy sprostać, stosując zwinne praktyki pracy w parach, TDD, refaktoryzacji i prostego projektu.

Nieustraszona kompetencja

Dlaczego większość systemów nie staje się z czasem coraz lepsza? Powodem jest strach. A dokładniej, strach przed zmianami.

Wyobraź sobie, że patrzysz na jakiś stary kod na ekranie. Twoją pierwszą myślą jest: „Ależ paskudny ten kod, trzeba go oczyścić". Ale zaraz pojawia się kolejna myśl: „Ja tego nie dotknę!". Z góry wiesz, że jeżeli coś zmienisz, to coś zepsujesz, a jeżeli coś zepsujesz, to na Ciebie spadnie odpowiedzialność za to. I w ten sposób cofasz się przed jedyną rzeczą, która mogłaby pomóc Ci usprawnić ten kod: przed oczyszczeniem go.

Jest to reakcja powodowana strachem. Boisz się kodu, a ten strach sprawia, że zachowujesz się niekompetentnie. Nie chcesz podjąć się niezbędnego zadania oczyszczenia kodu, ponieważ obawiasz się, jakie będą uzyskane wyniki. Dopuszczasz do powstania sytuacji, w której tworzony przez Ciebie kod tak bardzo wymyka się spod Twojej kontroli, że boisz się podejmować działania mające na celu poprawienie go. To ekstremalny brak odpowiedzialności.

Klienci, użytkownicy i menedżerowie oczekują od nas *nieustraszonej kompetencji*. Oczekują, że gdy zobaczymy coś brzydkiego lub niepoprawnego, to od razu przystąpimy do poprawiania i oczyszczania. Nikt nie chce, żebyśmy pozwalali na utrwalanie i powiększanie się problemów. Oczekuje się od nas, że będziemy w pełni kontrolować nasz kod, dbając o jego czystość.

Jak zatem możemy wyeliminować ten strach? Wyobraź sobie, że masz pod ręką przycisk sterujący dwoma światłami: zielonym i czerwonym. Przyjmij, że po naciśnięciu tego przycisku zielone światło zapali się w przypadku, gdy system pracuje prawidłowo, a czerwone w przypadku, gdy w systemie istnieją jakieś problemy. Wyobraź sobie, że możesz nacisnąć taki przycisk i w ciągu kilku sekund uzyskać tę cenną informację. Jak często korzystałbyś z tego przycisku? Z pewnością przez cały czas. Po każdej zmianie wprowadzonej do kodu naciskałbyś przycisk, aby się upewnić, że nic nie zostało zepsute.

Teraz wyobraź sobie, że na swoim ekranie widzisz jakiś paskudny kod. W pierwszej chwili myślisz: „To trzeba oczyścić". I od razu przystępujesz do pracy, a swój przycisk naciskasz po każdej zmianie, jaką wprowadzasz do kodu, aby upewnić się, że system nadal działa prawidłowo.

I strach od razu znika. Możesz spokojnie oczyścić kod. Możesz korzystać z praktyk Agile, takich jak refaktoryzacja, praca w parach i tworzenie prostego projektu, aby za ich pomocą poprawiać jakość systemu.

Tylko skąd wziąć taki przycisk? Okazuje się, że w taki przycisk wyposaża nas praktyka TDD. Jeżeli będziesz ją stosować z zapałem i determinacją, to uzyskasz swój przycisk informujący o stanie systemu, a tym samym staniesz się nieustraszenie kompetentny.

Testerzy nie powinni niczego znaleźć

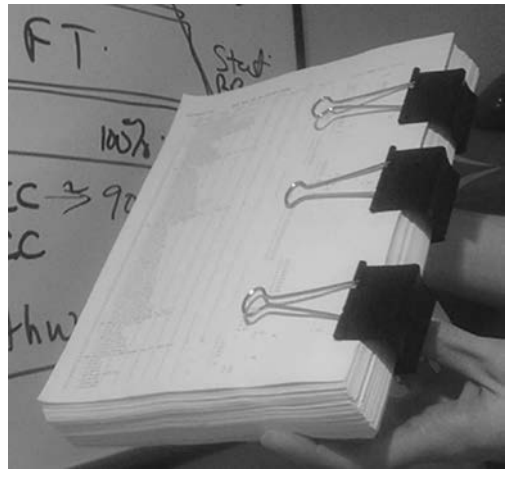
Dział kontroli jakości nie powinien znaleźć żadnych błędów w systemie. Kiedy testerzy przeprowadzą swoje testy, powinni nas tylko poinformować, że wszystko działa zgodnie z założeniami. Za każdym razem gdy testerzy znajdą jakiś błąd, zespół programistów powinien sprawdzić, co poszło źle w stosowanym przez nich procesie, i poprawić to, tak żeby następnym razem testerzy już niczego nie znaleźli.

Dział kontroli jakości powinien zadać sobie pytanie, dlaczego znajduje się na samym końcu procesu i ma za zadanie kontrolowanie systemu, który zawsze działa. Jak się zaraz przekonasz, dla kontroli jakości można znaleźć znacznie lepsze miejsce.

Takie praktyki Agile jak testy akceptacyjne, TDD i ciągła integracja bezpośrednio wiążą się z tym oczekiwaniem.

Automatyzacja testów

Ręce widoczne na rysunku 2.1 należą do menedżera z działu kontroli jakości. Trzyma on dokument będący *spisem treści* planu *ręcznego* testowania. Wymienionych jest w nim ponad 80 tysięcy ręcznie przeprowadzanych testów, które muszą być wykonywane co sześć miesięcy przez małą armię testerów w Indiach. Przeprowadzenie tych testów kosztuje ponad milion dolarów.



Rysunek 2.1. *Spis treści planu* ręcznych *testów*

Menedżer działu kontroli jakości pokazuje mi ten dokument, ponieważ właśnie wraca z biura swojego szefa. Jego szef z kolei właśnie wrócił z biura dyrektora finansowego. Mamy 2008 rok. Zaczęła się wielka recesja. Dyrektor finansowy zmniejszył budżet na testy do pół miliona co pół roku. Menedżer kontroli jakości pokazuje mi ten dokument, pytając jednocześnie, której połowy testów ma nie wykonać.

Powiedziałem mu, że niezależnie od tego, jak zmniejszy liczbę wykonywanych testów, nigdy nie będzie miał pewności, czy połowa systemu jeszcze działa.

To całkowicie normalny rezultat bazowania na testach ręcznych. Takie testy w którymś momencie zawsze są ograniczane. To, co właśnie opisałem, to najbardziej oczywisty mechanizm ograniczania testów ręcznych: one po prostu są *drogie* i dlatego są redukowane.

Istnieje jednak znacznie mniej jawny mechanizm powodujący ograniczanie wykonywania testów manualnych. Programiści zazwyczaj nie dostarczają na czas swoich produktów do działu kontroli jakości. To oznacza, że testerzy mają mniej czasu, niż planowano, żeby przeprowadzić wszystkie niezbędne testy. W takiej sytuacji konieczne jest *wybranie*, które spośród tych testów pozwolą dotrzymać terminu wydania produktu. I przez to część testów nie jest wykonywana. Zostają pominięte.

Poza tym ludzie nie są maszynami. Zmuszanie ludzi do wykonania pracy, z którą lepiej radzą sobie maszyny, jest drogie, nieefektywne i *niemoralne*. Dział kontroli jakości może zatrudnić pracowników do wykonywania znacznie lepiej dobranych zadań, które wymagają wykorzystania ludzkiej kreatywności i wyobraźni. Ale do tego jeszcze dojdziemy.

Klienci i użytkownicy oczekują, że każde nowe wydanie systemu będzie dokładnie przetestowane. Nikt nie spodziewa się tego, że zespół tworzący system będzie pomijał pewne testy tylko dlatego, że kończą się czas lub pieniądze. Oznacza to, że każdy test, który można skutecznie zautomatyzować, musi zostać zautomatyzowany. Użycie testów manualnych powinno ograniczać się do tych elementów, których nie można skontrolować automatycznie, ręcznie należy wykonywać także kreatywne testy eksploracyjne[5].

W spełnianiu oczekiwań związanych z testowaniem wspomagają nas zwinne praktyki TDD, ciągłej integracji i testów akceptacyjnych.

Krycie się nawzajem

Jako dyrektor techniczny oczekuję, że zespoły będą się zachowywały jak zespoły. A jak zachowują się zespoły? Wyobraź sobie piłkarzy ruszających z piłką na bramkę przeciwnika. Jeden z nich potyka się i upada. Co robią pozostali piłkarze? Pokrywają przestrzeń niezabezpieczaną przez kolegę i nadal *biegną z piłką do bramki drużyny przeciwnej*.

[5] Agile Alliance, *Exploratory testing*. Dostępne pod adresem *https://www.agilealliance.org/glossary/exploratory-testing*.

Na pokładzie statku każdy ma jakieś zadania. Każdy wie również, jak wykonywać zadania innych członków załogi. Po prostu na statku wszystkie zadania muszą zostać wykonane.

W zespole programistów, jeżeli Bartek zachoruje, to Kasia przejmie i wykona jego zadania. To oznacza, że Kasia musi wiedzieć, nad czym pracował Bartek i gdzie przechowuje on swoje pliki źródłowe, skrypty itp.

Oczekuję, że każdy członek zespołu programistów będzie w stanie wykonać zadania innych członków. Oczekuję też tego, że każdy członek zespołu programistów upewni się, że w przypadku jego niedyspozycji znajdzie się ktoś, kto przejmie jego pracę. To Twoim zadaniem jest upewnienie się, że przynajmniej jeden członek Twojego zespołu będzie w stanie pokryć lukę wywołaną Twoją nieobecnością.

Jeżeli Bartek jest specjalistą od baz danych, to nie spodziewam się, że prace nad projektem zostaną zatrzymane w przypadku jego choroby. Ktoś inny, nawet jeżeli nie jest „specjalistą od baz danych", powinien jednak być w stanie przejąć pałeczkę. Nie chcę, żeby wiedza była przechowywana w silosach. Ona powinna być współdzielona przez cały zespół. Jeżeli będę musiał przydzielić połowę zespołu do innego projektu, to nie zakładam, że zespół utraci połowę swojej wiedzy.

Na spełnienie tych oczekiwań pozwalają nam praktyki programowania w parach, całego zespołu i wspólnej własności.

Szczere szacowanie

Oczekuję od programistów szacunków i spodziewam się, że będą one szczere. Trudno o bardziej szczerą odpowiedź niż: „nie wiem". Takie szacowanie jest jednak niepełne. Z pewnością nie wiesz wszystkiego, ale na pewno niektóre elementy są Ci znane. Oczekuję zatem, że przedstawisz mi szacunki na podstawie tego, co wiesz, i tego, czego *nie wiesz*.

Na przykład możesz nie wiedzieć, ile czasu zajmie wykonanie pewnego zadania, ale możesz porównać jedno zadanie z drugim, stosując pojęcia względne. Możesz nie wiedzieć, jak długo potrwają prace nad stroną logowania, ale już teraz możesz stwierdzić, że tworzenie strony zmiany hasła zajmie połowę czasu potrzebnego na zbudowanie strony logowania. Takie względne szacunki również mają ogromną wartość, o czym przekonasz się w dalszej części tego rozdziału.

Zamiast podawać szacunki względne, możesz też przedstawić mi pewien zakres prawdopodobieństwa. Na przykład możesz mi powiedzieć, że prace nad stroną logowania potrwają od 5 do 15 dni, przy czym uśredniony czas ukończenia prac to 12 dni. Takie szacunki łączą w sobie to, co wiesz, z tym, czego nie wiesz, przez co szczerze określasz prawdopodobieństwo, a z tym menedżerowie mogą już pracować.

Na spełnienie tych oczekiwań pozwalają nam praktyki gry w planowanie oraz całego zespołu.

Mówienie „nie”

Oczywiście bardzo ważne jest poszukiwanie rozwiązań różnych problemów, ale oczekuję, że powiesz „nie”, jeżeli rozwiązania nie uda się znaleźć. Musisz sobie uświadomić, że powodem zatrudnienia Cię jest Twoja zdolność do mówienia „nie”, a umiejętność tworzenia kodu ma tu mniejsze znaczenie. To Ty, jako programista, wiesz, czy możliwe jest to, o co Cię proszę. Jako Twój dyrektor techniczny oczekuję, że poinformujesz mnie o tym, że właśnie zmierzamy w kierunku przepaści. Liczę, że niezależnie od presji związanej z terminami, jaką aktualnie odczuwasz, niezależnie od liczby menedżerów żądających od Ciebie konkretnych wyników, powiesz nam „nie”, gdy zajdzie taka potrzeba.

Na spełnienie tego oczekiwania pozwala nam praktyka całego zespołu.

Ciągłe agresywne uczenie się

Jako Twój dyrektor techniczny oczekuję, że nigdy nie przestaniesz się uczyć. Nasza branża zmienia się przez cały czas. Musimy zachować zdolność do zmieniania się razem z nią. A to oznacza naukę, naukę i jeszcze raz naukę! Czasami firma może sobie pozwolić na to, żeby wysyłać programistów na różne kursy i konferencje. Niekiedy firma może kupić dla nich książki i filmy treningowe. Ale jeżeli tak nie jest, to musisz znaleźć własne sposoby na dalszą naukę bez pomocy swojej firmy.

Na spełnienie tego oczekiwania pozwala nam praktyka całego zespołu.

Mentoring

Jako Twój dyrektor techniczny oczekuję, że będziesz uczyć innych. Okazuje się, że najlepszą metodą uczenia się jest nauczanie innych. A zatem gdy w zespole pojawiają się nowi ludzie — ucz ich. Nauczcie się uczyć się nawzajem.

Na spełnienie tego oczekiwania pozwala nam praktyka całego zespołu.

Lista praw

W trakcie spotkania w Snowbird Kent Beck powiedział, że celem korzystania z metod Agile jest zasypanie podziałów pomiędzy interesariuszami biznesowymi a programistami. W związku z tym Kent, Ward Cunningham i Ron Jeffries przygotowali specjalną listę praw.

Czytając te prawa, zauważ, że lista praw klienta i lista praw programisty wzajemnie się uzupełniają. Pasują do siebie jak kawałki układanki. Powoduje to, że pomiędzy klientami i programistami panuje równowaga oczekiwań.

Lista praw klienta

Na liście praw klienta znajdują się następujące pozycje:

- Masz prawo domagać się ogólnego planu prac, aby dowiedzieć się, co może zostać wykonane, kiedy i jakim kosztem.
- Masz prawo w każdej iteracji uzyskać najlepsze możliwe wartości.
- Masz prawo widzieć postępy prac nad działającym systemem, których poprawność dowodzona jest za pomocą powtarzalnych, zdefiniowanych przez Ciebie testów.
- Masz prawo zmienić zdanie, zmodyfikować funkcje systemu oraz zmienić priorytety prac bez narażania się na gigantyczne koszty.
- Masz prawo być informowany o zmianach w harmonogramie odpowiednio wcześniej, tak abyś miał możliwość wprowadzenia zmian do zakresu prac, które pozwolą na ukończenie zadań w wyznaczonym czasie. W dowolnym momencie możesz anulować projekt i otrzymać system działający w zakresie adekwatnym do poczynionych inwestycji.

Lista praw programisty

Na liście praw programisty znajdują się następujące pozycje:

- Masz prawo wiedzieć, co jest niezbędne i jakie są priorytety.
- Masz prawo do ciągłego realizowania prac o wysokiej jakości.
- Masz prawo prosić o pomoc swoich kolegów, menedżerów i klientów i otrzymać ją.
- Masz prawo tworzyć i aktualizować swoje własne szacunki.
- Masz prawo brać odpowiedzialność za swoje działania, nie powinieneś być do tego zmuszany.

To bardzo ważne deklaracje. Przyjrzyjmy się im po kolei.

Klienci

W naszym kontekście słowo „klient” oznacza ogólnie osoby związane ze sferą biznesową. Zaliczają się do nich rzeczywiści klienci, menedżerowie, dyrektorzy, liderzy projektów i inne osoby, które w jakimkolwiek stopniu odpowiadają za harmonogram lub budżet albo czerpią korzyści z działania systemu.

> *Klienci mają prawo domagać się ogólnego planu prac, aby dowiedzieć się, co może zostać wykonane, kiedy i jakim kosztem.*

Wiele osób uważa, że wstępne planowanie nie jest częścią zwinnego rozwijania oprogramowania. Ale już pierwsze z praw klienta zadaje kłam takiemu twierdzeniu. Oczywiście, że klient potrzebuje jakiegoś planu. To jasne, że w tym planie muszą znaleźć się

informacje o harmonogramie i kosztach prac. I nie dziwi to, że taki plan powinien być jak najbardziej dokładny i szczegółowy.

I z tym ostatnim wymogiem mamy największe kłopoty, ponieważ jedyną metodą na przygotowanie dokładnego i szczegółowego planu jest zrealizowanie projektu. Nie ma możliwości, żeby uzyskać wymaganą dokładność i szczegółowość, stosując inne metody. Oznacza to, że my, jako programiści, możemy przestrzegać tego prawa klienta, upewniając się, że nasze plany, szacunki i harmonogramy właściwie odzwierciedlają poziom naszej niepewności i definiują metody przeciwdziałania jej.

W skrócie, nie możemy się zgadzać na realizację określonego zakresu prac przy z góry założonej dacie ich zakończenia. Musimy mieć pole manewru albo po stronie daty albo po stronie zakresu prac, dlatego w harmonogramach posługujemy się krzywą prawdopodobieństwa. Na przykład szacujemy, że istnieje 95% prawdopodobieństwo, że do wyznaczonej daty zrealizujemy dziesięć pierwszych historii. Istnieje też 50% prawdopodobieństwo, że do wyznaczonej daty uda się nam zrealizować pięć kolejnych historii. A także 5% prawdopodobieństwo, że w wyznaczonym czasie damy radę zrealizować jeszcze pięć dodatkowych historii.

Klienci mają prawo domagać się takiego planu bazującego na prawdopodobieństwie, ponieważ bez takich planów nie są w stanie zarządzać swoim biznesem.

Klienci mają prawo w każdej iteracji uzyskać najlepsze możliwe wartości.

W metodach Agile prace nad systemem dzieli się na części o stałej wielkości, które nazywane są *iteracjami*. Klienci mają zatem prawo oczekiwać, że programiści w każdym momencie będą pracowali nad najważniejszymi w tym czasie rzeczami, a oni sami w każdej iteracji uzyskają maksymalne możliwe *użyteczne* wartości biznesowe. Priorytety tych wartości definiowane są przez klientów w ramach sesji planowania na samym początku każdej iteracji. Klienci wybierają historie, które gwarantują im największy zwrot z inwestycji i mieszczą się w szacowanym zakresie prac, jakie programiści mogą wykonać w danej iteracji.

Klienci mają prawo widzieć postępy prac nad działającym systemem, których poprawność dowodzona jest za pomocą powtarzalnych zdefiniowanych przez nich testów.

Gdy spoglądasz na tę sprawę z punktu widzenia klienta, to wydaje się to oczywiste. Oczywiście, że klienci mają prawo widzieć przyrostowe postępy. To jasne, że mają prawo zdefiniować kryteria akceptacji wszystkich wprowadzanych zmian. Zrozumiałe jest też to, że mają prawo szybko uzyskać dowód na to, że ich kryteria akceptacji zostały spełnione.

Klienci mają prawo zmienić zdanie, zmodyfikować funkcje systemu oraz zmienić priorytety prac bez narażania się na gigantyczne koszty.

Mówimy przecież o *oprogramowaniu*. Sensem istnienia oprogramowania jest możliwość łatwego zmieniania zachowania naszych maszyn. To właśnie ta elastyczność była powodem,

dla którego ktoś wymyślił oprogramowanie. A to oznacza, że klienci muszą mieć prawo do wprowadzania zmian w wymaganiach.

Klienci mają prawo być informowani o zmianach w harmonogramie odpowiednio wcześniej, tak aby mieli możliwość wprowadzenia do zakresu prac zmian pozwalających na ukończenie zadań w wyznaczonym czasie.

Klienci mają prawo w dowolnym momencie anulować projekt i otrzymać system działający w zakresie adekwatnym do poczynionych inwestycji.

Zauważ, że klienci nie mają prawa żądać ścisłego trzymania się harmonogramu. Ich prawo ogranicza się do zarządzania harmonogramem poprzez wprowadzanie zmian do zakresu prac. Bardzo ważną rzeczą, jaka wynika z tego prawa, jest prawo do *informacji*, że harmonogram jest zagrożony, tak żeby odpowiednio wcześnie można było zająć się jego korygowaniem.

Programiści

W kontekście, który nas interesuje, programistą jest każdy, kto pracuje nad kodem rozwijanego systemu. Do tej grupy zaliczają się programiści, kontrolerzy jakości, testerzy i analitycy biznesowi.

Programiści mają prawo wiedzieć, co jest niezbędne i jakie są priorytety.

Tutaj również koncentrujemy się na *wiedzy*. Programiści mają prawo otrzymywać szczegółowe wymagania dla systemu i informacje o wadze tych wymagań. Oczywiście wymóg *praktyczności* stawiany przed szacunkami dotyczy również wymagań. Nie zawsze możliwe jest zachowanie pełnej dokładności przy definiowaniu wymagań. Trzeba też pamiętać o tym, że klienci mają prawo zmieniać zdanie.

Poza tym muszę zaznaczyć, że to prawo dotyczy działań zamykających się *w ramach jednej iteracji*. Poza tą iteracją zarówno wymagania, jak i ich priorytety mogą się dowolnie zmieniać. Jednak w ramach danej iteracji programiści mają prawo uznać je za niezmienne. Pamiętajmy jednak, że programiści mogą zdecydować się odstąpić od tego prawa, jeżeli uznają, że proponowana zmiana nie niesie za sobą większych konsekwencji.

Programiści mają prawo do ciągłego realizowania prac o wysokiej jakości.

To chyba najważniejsze ze wszystkich przedstawionych praw. Programiści mają prawo do wykonywania dobrej roboty. Klienci nie mają zatem prawa zmuszać ich do wybierania drogi na skróty albo wykonywania pracy o niskiej jakości. Mówiąc inaczej, klienci nie mają prawa zmuszać programistów do niszczenia swojej reputacji zawodowej albo do naruszania etyki zawodowej.

Programiści mają prawo prosić o pomoc swoich kolegów, menedżerów i klientów i otrzymywać ją.

Pomoc może mieć wiele różnych form. Programiści mogą na przykład prosić się nawzajem o pomoc przy rozwiązywaniu pewnego problemu, przy kontrolowaniu wyników albo przy nauce nowego frameworka. Programiści mogą prosić klientów o dokładniejsze objaśnienie wymagań albo o ustalenie priorytetów. Przede wszystkim to prawo daje programistom możliwość *komunikowania się*. A z prawem do proszenia o pomoc wiąże się też obowiązek udzielania pomocy.

Programiści mają prawo tworzyć i aktualizować swoje własne szacunki.

Nikt nie jest w stanie oszacować złożoności Twojego zadania. A gdy dokonasz takiego szacunku, to zawsze możesz go zmienić, jeżeli pojawi się jakiś nowy czynnik wpływający na wykonanie tego zadania. Szacowanie złożoności zadań to w zasadzie przypuszczenia. Oczywiście takie przypuszczenia bazują na pewnych informacjach, ale nadal są tylko przypuszczeniami. Dokładność szacunków poprawia się jednak z czasem. Szacunków nigdy nie można traktować jako zobowiązań.

Programiści mają prawo brać odpowiedzialność za swoje działania, nie powinni być do tego zmuszani.

Profesjonaliści *przyjmują* pracę do wykonania. Praca nie jest im przydzielana. Zawodowy programista ma pełne prawo odmówić wykonania określonej pracy lub zadania. Może nie czuć się wystarczająco kompetentny, żeby dobrze wykonać prace związane z danym zadaniem. Może uważać, że ktoś inny lepiej nadaje się do wykonania tej konkretnej pracy. Ale może się też zdarzyć, że programista odmówi wykonania zadania z powodów osobistych lub moralnych[6].

W każdym wypadku prawo do przyjmowania pracy wiąże się z pewnym kosztem. Przyjmowanie pracy oznacza też wzięcie na siebie odpowiedzialności za jej wykonanie. Programista przyjmujący pewną pracę staje się odpowiedzialny za jakość wykonywanych zadań, za ciągłe aktualizowanie szacunków, tak żeby można było dostosowywać harmonogram, za przekazywanie całemu zespołowi informacji o stanie realizacji zadania. Jest też zobowiązany do proszenia o pomoc, jeżeli jest to konieczne.

Programowanie w zespole oznacza ścisłą współpracę z innymi programistami, zarówno młodszymi, jak i starszymi. Zespół ma prawo wspólnie zdecydować, kto będzie wykonywać określone zadania. Lider techniczny może zapytać programistę, czy ten przyjmie dane zadanie, ale nie ma prawa narzucać zadań komukolwiek.

Wnioski

Agile to zestaw praktyk wspomagających profesjonalne rozwijanie oprogramowania. Osoby stosujące te praktyki akceptują rozsądne oczekiwania menedżerów, interesariuszy i klientów.

[6] Zastanówmy się nad pracą programistów w firmie Volkswagen, którzy „zgodzili się" przygotować oprogramowanie oszukujące testy EPA w Kalifornii. Więcej na *https://pl.wikipedia.org/wiki/Afera_Volkswagena*.

Z drugiej strony korzystają też z praw, jakie podczas stosowania metod Agile przysługują wszystkim programistom i klientom. To wzajemne korzystanie z praw i akceptowanie oczekiwań jest podstawą *etycznego* standardu tworzenia oprogramowania.

Agile nie jest procesem. Agile nie jest przejściową modą. Agile nie jest tylko pewnym zbiorem reguł. Agile jest raczej zbiorem praw, oczekiwań i praktyk, które wspólnie stają się bazą naszej etycznej profesji.

Agile: ponadczasowa idea zwinnego rozwoju projektu!

W lutym 2001 roku grupa kilkunastu pasjonatów programowania zebrała się w Snowbird w stanie Utah, aby podyskutować na temat efektywności tworzenia oprogramowania i określić nowe zasady pracy. Nie był to cel łatwy do osiągnięcia. Siedemnaście osób o całkowicie różnych doświadczeniach i przekonaniach raczej rzadko wypracowuje wspólne stanowisko. A jednak w tym przypadku tak się stało i powstał *Manifest Agile*. Wtedy właśnie narodziła się jedna z najważniejszych idei w świecie oprogramowania, która stopniowo zyskiwała zwolenników również poza światem programistów. Zwiększająca się popularność Agile sprawiła, że wielu starało się upiększać, rozmywać czy modyfikować jego koncepcje. Wokół Agile narosło sporo nieporozumień.

Ta książka jest niezwykle potrzebnym powrotem do podstaw Agile. Przyda się nowemu pokoleniu programistów, nieprogramistów i osób, które kierują zespołami. Znalazł się tu jednoznaczny opis sedna Agile jako prostej i spójnej koncepcji zwinnego zarządzania małym projektem w niewielkim zespole. To bardzo ważny aspekt Agile, gdyż wielkie projekty składają się z szeregu małych projektów. Pokazano tu, jak zaimplementować metodyki Agile w poprawny, czysty sposób. Zaprezentowano kluczowe koncepcje w ich pierwotnej formie. Wyjaśniono zasady poszczególnych praktyk. Przy tym wszystkim książka jest szczerą, osobistą opowieścią o historii Agile i jego fundamentalnej idei.

W tej książce między innymi:

- **czym jest Agile i jaka jest jego istota**
- **właściwe stosowanie metodyki Scrum**
- **najważniejsze praktyki biznesowe Agile**
- **najważniejsze praktyki techniczne Agile**
- **wartości i rzemieślnictwo a praca zwinnych zespołów**

Robert C. Martin (dla przyjaciół: Wujek Bob) programuje od 1970 roku. Jest założycielem firmy Uncle Bob Consulting LLC, która zajmuje się konsultingiem w zakresie oprogramowania, szkoleniami oraz świadczeniem usług rozwoju umiejętności dla największych firm z całego świata. Opublikował dziesiątki artykułów w różnych czasopismach, regularnie bierze udział w międzynarodowych konferencjach. Jest też twórcą serii filmów edukacyjnych dostępnych na stronie *cleancoders.com*. Napisał wiele cenionych książek. Był też pierwszym przewodniczącym organizacji Agile Alliance.

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 978-83-283-6304-5

9 788328 363045

INFORMATYKA W NAJLEPSZYM WYDANIU

Cena: 47,00 zł