Robert C. Martin

MISTRZ CZYSTEGO KODU

KODEKS POSTĘPOWANIA PROFESJONALNYCH PROGRAMISTÓW

Helion

Tytuł oryginału: The Clean Coder: A Code of Conduct for Professional Programmers

Tłumaczenie: Wojciech Moch

ISBN: 978-83-283-3131-0

Authorized translation from the English language edition, entitled: THE CLEAN CODER:
A CODE OF CONDUCT FOR PROFESSIONAL PROGRAMMERS; ISBN 0137081073;
by Robert C. Martin; published by Pearson Education, Inc, publishing as Prentice Hall.

Wydawnictwo HELION
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/mckkov
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

SPIS TREŚCI

4 Kodowanie

W poprzedniej książce[1] napisałem całkiem sporo o strukturze i naturze *czystego kodu*. W tym rozdziale będę omawiać sam *akt* tworzenia kodu oraz kontekst otaczający ten akt.

Gdy miałem 18 lat, całkiem nieźle szło mi pisanie na klawiaturze, ale nadal musiałem patrzeć na klawisze. Nie byłem w stanie pisać bezwzrokowo. Któregoś wieczoru spędziłem kilka godzin nad klawiaturą komputera IBM 029, starając się nie patrzeć na swoje palce w czasie, gdy wpisywałem program, który zapisałem na kilku formularzach. Po zakończeniu wprowadzania sprawdziłem każdą kartę z osobna i wyrzuciłem te, które zawierały błędy.

[1] [Martin09].

Początkowo zdarzało mi się całkiem sporo błędów, ale pod wieczór mogłem zapisywać karty niemal perfekcyjnie. W ciągu długiej nocy przekonałem się, że pisanie bezwzrokowe wymaga przede wszystkim *pewności siebie*. Moje palce wiedziały, gdzie są poszczególne klawisze, i musiałem tylko zyskać pewność, że nie popełniam błędów. Jedną z rzeczy, które mi bardzo przy tym pomogły, było to, że dokładnie czułem, kiedy popełniam błąd. Pod wieczór od razu wiedziałem, kiedy popełniłem błąd, i mogłem zaraz wyrzucić kartę bez patrzenia na nią.

Umiejętność wyczuwania własnych błędów jest niezwykle ważna. Nie tylko podczas pisania na klawiaturze, ale w każdym elemencie. Ten dodatkowy zmysł oznacza, że bardzo szybko zamykasz pętlę sprzężenia zwrotnego i jeszcze szybciej jesteś w stanie uczyć się na swoich błędach. Od czasu tego dnia spędzonego nad klawiaturą dwudziestki dziewiątki wiele się nauczyłem i pogłębiłem swoją wiedzę. Za każdym razem przekonywałem się, że kluczem do sukcesu są pewność siebie i umiejętność wyczuwania błędów.

W tym rozdziale opiszę mój osobisty zbiór zasad i reguł dotyczących tworzenia kodu. Te reguły i zasady nie odnoszą się do samego kodu, ale raczej do mojego zachowania, nastawienia i humoru podczas jego pisania. Opisują mój mentalny, moralny i emocjonalny kontekst pisania kodu. To właśnie one są podstawą mojej pewności siebie i umiejętności wyczuwania błędów.

Z pewnością nie zgodzisz się ze wszystkim, co tutaj napiszę, to w końcu sprawy bardzo osobiste. Co więcej, całkiem prawdopodobne jest, że gwałtownie zaoponujesz przeciwko niektórym moim zasadom. To zupełnie normalne. Nie mają być one prawdami absolutnymi dla nikogo poza mną. Są tylko moją własną metodą profesjonalnego tworzenia kodu.

Być może analizując moje środowisko tworzenia kodu, będziesz w stanie lepiej zrozumieć całą zawartość tej książki.

Przygotowanie

Tworzenie kodu jest bardzo wyczerpującą czynnością i ogromnym wyzwaniem intelektualnym. Wymaga ono osiągnięcia pewnego poziomu koncentracji i skupienia, niezbędnego tylko w kilku innych dziedzinach. Wynika to z tego, że tworzenie kodu wymaga od programisty żonglowania wieloma sprzecznymi zasadami.

1. Twój kod musi działać. Musisz poznać problem, który masz rozwiązać, i zdefiniować sposób jego rozwiązania. Musisz mieć pewność, że napisany przez Ciebie kod będzie wiernym odwzorowaniem tego rozwiązania. Musisz zadbać o każdy szczegół tego rozwiązania, pozostając w zgodzie z językiem programowania, platformą, używaną architekturą oraz kruczkami systemu.
2. Kod musi rozwiązywać problem zdefiniowany przez klienta. Często jest tak, że wymagania określone przez klienta wcale nie rozwiązują jego problemów.

Twoim zadaniem jest wychwycenie tych rozbieżności i takie prowadzenie negocjacji z klientem, żeby zostały zaspokojone jego rzeczywiste potrzeby.

3. Twój kod musi dopasować się do istniejącego systemu. Nie powinien zwiększać jego sztywności, delikatności ani nieprzejrzystości. Wszystkie zależności muszą pozostać pod kontrolą. W skrócie: Twój kod musi być zgodny z podstawowymi zasadami inżynierii[2].
4. Twój kod musi być czytelny dla pozostałych programistów. I nie chodzi tu tylko o pisanie odpowiednich komentarzy, a raczej o takie ukształtowanie kodu, żeby sam ujawniał Twoje zamiary. To jest najtrudniejsze. Wydaje się, że dla programisty właśnie to może być najtrudniejsze do opanowania.

Żonglowanie tymi wszystkimi zasadami może być naprawdę trudne. Już samo fizjologiczne utrzymywanie niezbędnego skupienia przez dłuższy czas jest niełatwe. Dodaj do tego cały zbiór problemów i elementów rozpraszających wynikających z pracy w zespole, w większej organizacji, nie wspominając nawet o typowych rozterkach codziennego życia. Chodzi o to, że na każdym rogu czai się coś, co może zmniejszyć Twoje skupienie.

Jeżeli nie możesz się odpowiednio skoncentrować i skupić, to z pewnością napisany przez Ciebie kod będzie nieprawidłowy. Będzie zawierał błędy. Będzie miał nieodpowiednią strukturę. Będzie nieprzejrzysty i zagmatwany. Nie będzie rozwiązywał problemów klienta. Oznacza to, że będzie musiał zostać przebudowany i napisany na nowo. Praca bez skupienia tworzy tylko śmieci.

Jeżeli jesteś zmęczony lub brakuje Ci skupienia, to *nie twórz kodu*. Ostatecznie staniesz przed koniecznością ponownego wykonania tej samej pracy. Lepiej będzie od razu wyeliminować elementy rozpraszające i uspokoić swój umysł.

Kod z godziny 3 nad ranem

Najgorszy kod w życiu napisałem o godzinie 3 nad ranem. Było to w roku 1988, kiedy to pracowałem dla młodej firmy telekomunikacyjnej o nazwie Clear Communications. Wszyscy w firmie pracowaliśmy przez długie godziny, żeby wytworzyć „kapitał" (ang. *sweat equity*). Oczywiście wszyscy marzyliśmy o bogactwie.

Pewnego bardzo późnego wieczoru (a może raczej bardzo wczesnego ranka) w ramach rozwiązywania problemu czasowego nakazałem swojemu kodowi wysyłać do samego siebie komunikat poprzez system rozprowadzania zdarzeń (nazywaliśmy to „wysyłaniem poczty"). To było bardzo *niewłaściwe* rozwiązanie, ale o 3 nad ranem wyglądało bardzo obiecująco. Co więcej, po 18 godzinach tworzenia kodu (nie wspominając o 60 – 70 godzinach pracy w tygodniu) na nic więcej nie było mnie stać.

[2] [PPP2002].

Pamiętam, że byłem dumny z tego, że tak długo pracowałem. Pamiętam, że czułem swoje *poświęcenie*. Pamiętam też, że sądziłem, iż zawodowcy zawsze pracują do 3 nad ranem. Oj, jak bardzo się myliłem!

Tamten kod kopnął nas jeszcze wielokrotnie. Zdefiniowałem wadliwą strukturę, z której korzystali wszyscy, ale jednocześnie zmuszeni byli tworzyć ciągłe obejścia. Wytworzyła ona najdziwniejsze problemy czasowe i niezwykłe pętle sprzężenia zwrotnego. Wpadaliśmy w nieskończone pętle wiadomości, gdy jeden komunikat powodował wysłanie innego i jeszcze kolejnego, *ad infinitum*. Nigdy nie mieliśmy czasu na napisanie na nowo fragmentów powodujących te problemy (a przynajmniej tak się nam wydawało), ale zawsze znajdowaliśmy czas na utworzenie kolejnej łatki i poprawki obchodzących wykrywane problemy. Całość ciągle rosła, opakowując ten kod napisany o 3 nad ranem w coraz większy bagaż efektów ubocznych. Później stało się to podstawą żartów w zespole. Za każdym razem, gdy byłem zmęczony lub sfrustrowany, mówili do siebie: „Patrzcie! Bob zaraz wyśle do siebie maila!”.

Morał tej historii jest taki: nie pisz kodu, jeżeli jesteś zmęczony. Poświęcenie i profesjonalizm są bardziej związane z odpowiednią dyscypliną, a nie z godzinami pracy. Upewnij się, że Twój styl życia, zdrowie i dawka snu są wystarczające, żeby pracy poświęcić osiem *dobrych* godzin.

Kod ze zmartwieniami

Czy kiedykolwiek zdarzyło Ci się ostro pokłócić ze współmałżonkiem lub przyjacielem, a zaraz potem przystąpić do pisania kodu? Pamiętasz zapewne, że gdzieś z tyłu Twojej głowy trwał mały proces, który starał się ponownie analizować przebieg kłótni. Stres generowany przez ten mały proces można czasami poczuć w klatce piersiowej lub w żołądku. Sprawia to, że czujesz niepokój jak po wypiciu zbyt wielu kaw lub coli. To bardzo rozprasza.

Jeżeli martwię się kłótnią ze swoją żoną albo problemami z klientem albo chorym dzieckiem, to nie jestem w stanie się odpowiednio skupić. Moja koncentracja cały czas odpływa. Okazuje się, że mimo oczu skierowanych na ekran, a palców leżących na klawiaturze nie jestem w stanie nic zrobić. Katatonia. Paraliż. Jestem tysiąc kilometrów stąd, analizując palący mnie problem, a nie zastanawiając się nad tworzonym kodem.

Czasami zmuszam się do *myślenia* na temat kodu. W ten sposób jestem w stanie dopisać wiersz lub dwa. Mogę zmusić się do napisania jednego testu lub dwóch, ale nie jestem w stanie dłużej koncentrować się na pracy. Ostatecznie zawsze popadam w to niezwykłe odrętwienie, w którym nie widzę nic mimo otwartych oczu i przetwarzam moje najróżniejsze troski.

Nauczyłem się już, że nie jest to dobry czas na tworzenie kodu. Każdy kod, który w tym czasie napiszę, będę mógł wyrzucić do kosza. Dlatego właśnie zamiast pisać, staram się rozwiązywać problemy powodujące zmartwienia.

Oczywiście istnieje wiele trosk, których nie da się usunąć z godziny na godzinę. Co więcej, nasz pracodawca raczej nie będzie chętnie tolerował naszej bezczynności, w czasie gdy

będziemy starać się rozwiązywać prywatne problemy. Sztuczka polega zatem na tym, żeby nauczyć się zamykać ten działający w tle proces, a przynajmniej zmniejszać jego priorytet, tak aby nie stawał się on nieustającym elementem rozpraszającym.

Sam robię to, odpowiednio dzieląc swój czas. Zamiast zmuszać się do pisania kodu z ciągle atakującymi mnie myślami, poświęcam im pewien wybrany wycinek czasu — może nawet godzinę — starając się pracować nad powodem mojego niepokoju. Jeżeli moje dziecko jest chore, to dzwonię do domu, aby dowiedzieć się, czy wszystko jest w porządku. Jeżeli pokłóciłem się z żoną, to dzwonię do niej i staram się obgadać powód tej kłótni. Jeżeli mam problemy finansowe, to myślę nad tym, jak sobie z nimi poradzić. Wiem, że najprawdopodobniej nie uda mi się rozwiązać problemu w czasie tej godziny, ale jest bardzo możliwe, że zmniejszę w ten sposób swoje napięcie i uciszę działający w tle proces.

Najlepiej byłoby, gdyby czas poświęcony na walkę z osobistymi problemami był Twoim czasem prywatnym. Spędzenie w ten sposób godziny w pracy byłoby niewłaściwe. Zawodowi programiści wykorzystują swój prywatny czas tak, żeby czas spędzony w biurze był jak najbardziej produktywny. Oznacza to, że jeszcze w domu należy poświęcić trochę czasu na zmniejszenie wewnętrznych niepokojów, tak aby nie przynosić ich ze sobą do biura.

Jednak jeżeli jesteś już w pracy, a wewnętrzne niepokoje nie pozwalają pracować z pełną skutecznością, to może lepiej poświęcić godzinę na ich uspokojenie, niż zmuszać się do pisania kodu, który później i tak będzie trzeba napisać od nowa. Albo co gorsza dalej z nim żyć.

Strefa

Wiele już napisano o tym stanie hiperproduktywności nazywanym „przepływem” (ang. *flow*). Niektórzy programiści nazywają ten stan „strefą” (ang. *zone*). Niezależnie od nazwy każdy z nas z pewnością się już z tym stanem zetknął. To stan świadomości z dużym skupieniem i widzeniem tunelowym, w który mogą wejść programiści tworzący kod. Wówczas czują się naprawdę produktywni. To właśnie wtedy czują się całkowicie *nieomylni*. W efekcie starają się jak najdłużej pozostawać w tym stanie, a miarą ich wartości staje się spędzony w nim czas.

Oto mała rada od kogoś, kto dokładnie przerobił już ten temat: *unikaj strefy*. W tym stanie świadomości tak naprawdę nie jest się hiperproduktywnym, a już na pewno nie jest się nieomylnym. Tak naprawdę jest to tylko pewien stan medytacyjny, w którym niektóre czynniki racjonalne zostają umniejszone w celu uzyskania większej prędkości pracy.

Muszę tu wyrazić się jasno. Będąc w strefie, *na pewno* napiszesz więcej kodu. Jeżeli stosujesz praktyki TDD, to na pewno szybciej przejdziesz pętlę czerwone – zielone – refaktoryzacja. I dodatkowo *poczujesz* lekką euforię. Problem polega na tym, że będąc w strefie, stracisz szerszy pogląd na sprawy, przez co możesz podejmować decyzje, które później będzie trzeba zmieniać. Kod napisany w strefie na pewno powstaje szybciej, ale później trzeba go częściej poprawiać.

Aktualnie, gdy czuję, że sam zaczynam wpadać w strefę, na kilka minut odchodzę od komputera. Oczyszczam umysł, odpisując na kilka e-maili i czytając kilka wiadomości na Tweeterze. Jeżeli jest tuż przed południem, to idę na obiad. Jeżeli pracuję w zespole, to idę poszukać sobie partnera.

Jedną z zalet programowania w parach jest to, że dla pary praktycznie niemożliwe jest wejście w strefę. Jest to stan niekomunikatywny, podczas gdy programowanie w parach wymaga ciągłej i intensywnej komunikacji. Co ciekawe, często słyszę, że jedną z wad programowania w parach jest to, że nie pozwala ono na wejście w strefę. O rany! Przecież nikt *nie* powinien dążyć do tego, żeby znaleźć się w strefie.

To jednak *nie do końca* prawda. Czasami strefa jest miejscem, w którym chciałoby się znaleźć. Ale zdarza się to tylko podczas *ćwiczeń*. O tym opowiem jednak w innym rozdziale.

Muzyka

W latach 70. w firmie Teradyne miałem prywatne biuro. Byłem wtedy administratorem systemu na naszym PDP 11/60 i jako jeden z niewielu programistów dostałem swój własny terminal. Terminalem tym był VT100 podłączony do PDP-11 za pomocą 25 metrów kabla RS-232 działającego z prędkością 9600 bodów, który przypiąłem do sufitu swojego biura i w ten sposób poprowadziłem do pokoju komputerowego.

W biurze miałem zestaw stereo. Składał się ze starego gramofonu, wzmacniacza i sporych głośników. Miałem też dużą kolekcję płyt winylowych, w tym Led Zeppelin, Pink Floyd i… chyba już wiesz, o co chodzi.

Często korzystałem z tego zestawu w czasie pisania kodu. Sądziłem, że muzyka ułatwiała mi koncentrację. Niestety, byłem w błędzie.

Pewnego dnia musiałem wrócić do modułu, który pisałem, słuchając sekwencji otwierającej *The Wall*. W komentarzach do kodu zobaczyłem kawałki tekstu tego utworu, a także wzmianki o bombowcach nurkujących i płaczących dzieciach.

To wtedy zrozumiałem, że czytelnik mojego kodu dowie się z niego więcej o moich gustach muzycznych niż o problemie, który ten kod ma rozwiązywać.

Okazało się, że słuchając muzyki, wcale nie piszę lepszego kodu. Muzyka wcale nie pomaga mi się skoncentrować. Co więcej, słuchanie muzyki zdaje się pochłaniać całkiem sporą część zasobów mojego umysłu, których potrzebuję do tworzenia czystego i dobrze zaprojektowanego kodu.

Być może w Twoim przypadku działa to inaczej. Być może Tobie muzyka *ułatwia* pisanie kodu. Znam w końcu wiele osób, które piszą kod ze słuchawkami na głowie. Jestem w stanie zaakceptować fakt, że muzyka ułatwia im pracę, ale mam dziwne podejrzenia, że tak naprawdę pomaga im ona wejść do strefy.

Przerwy

Wyobraź sobie, że właśnie tworzysz kod na swojej stacji roboczej. Jak odpowiadasz, gdy ktoś zadaje Ci pytanie? Odpowiadasz niemiło? Patrzysz gniewnie? Czy język Twojego ciała informuje rozmówcę, że ma się szybko oddalić, bo masz inne zajęcie? W skrócie: czy zachowujesz się niegrzecznie? A może przerywasz swoją pracę i chętnie pomagasz komuś, kto właśnie ma jakiś problem? Czy traktujesz intruza tak jak w Twoim wyobrażeniu on powinien potraktować Ciebie, gdy przyjdziesz z jakimś problemem?

Takie niegrzeczne zachowania często wiążą się ze strefą. Mogą wynikać z rozgoryczenia powodowanego wyciągnięciem ze strefy albo przerwaniem prób wejścia do niej. W obu przypadkach niegrzeczne zachowania są powiązane ze strefą.

Czasami jednak powodem wcale nie jest strefa, ale prosty fakt, że próbujesz zrozumieć coś złożonego, co wymaga sporej koncentracji. W takim przypadku masz do wyboru kilka rozwiązań.

Dobrą metodą radzenia sobie z takimi przerwami może być praca w parach. Twój partner może nadal pamiętać kontekst Waszych prac, podczas gdy Ty odbierasz telefon albo odpowiadasz na pytania innego kolegi. Po powrocie partner szybko pomoże Ci odtworzyć stan umysłu sprzed tej przerwy.

Doskonałą pomocą może być też technika TDD. Jeżeli masz niedziałający test, to z pewnością przechowuje on cały kontekst zadania, nad którym pracujesz. Po przerwie możesz łatwo wrócić do pracy i sprawić, żeby test zaczął działać.

Oczywiście zawsze *będą zdarzały się przerwy*, które będą Cię rozpraszały i powodowały straty czasu. Gdy coś takiego się zdarzy, pamiętaj, że następnym razem to Ty możesz potrzebować pomocy i będziesz komuś przerywać pracę. Oznacza to, że do profesjonalnego zachowania należy uprzejmość i chęć udzielania pomocy.

Blokada twórcza

Czasami kod po prostu nie chce powstać. Zdarzyło mi się to kilka razy i wiele razy widziałem taki stan u innych. Siedzi się wtedy przed komputerem i zupełnie nic się nie dzieje.

Często znajdujesz wtedy jakieś inne zajęcie. Czytasz e-maile albo wiadomości z Tweetera. Przeglądasz różne książki, dokumenty lub kalendarze. Zwołujesz zebranie albo zajmujesz się rozmową ze współpracownikami. Robisz wtedy *cokolwiek*, byle tylko nie siedzieć przed komputerem i nie patrzeć, jak kod uparcie nie chce się urodzić.

Co powoduje taką blokadę? O wielu czynnikach wspominałem już wcześniej. Dla mnie jedną z najważniejszych rzeczy jest sen. Jeżeli się dobrze nie wyśpię, to najzwyczajniej

w świecie nie jestem w stanie pisać kodu. Innymi podobnie działającymi czynnikami mogą być strach, zmartwienia lub depresja.

Najdziwniejsze jest to, że istnieje dla tego problemu bardzo proste rozwiązanie, które sprawdza się niemal za każdym razem. Jest naprawdę proste, a może dać Ci rozpęd wystarczający do napisania całkiem sporych ilości kodu.

Rozwiązanie jest takie: znajdź sobie partnera do programowania w parze.

To niesamowite, jak dobrze sprawdza się ta metoda. Gdy tylko usiądziesz przy kimś innym, wszystkie problemy powodujące blokadę same znikają. Podczas pracy z inną osobą następuje zmiana *fizjologiczna*. Nie mam pojęcia, na czym ona polega, ale za każdym razem czuję ją doskonale. W moim mózgu lub w moim ciele następuje zmiana chemiczna, która pozwala mi się przebić przez blokadę i wrócić do normalnej pracy.

Nie jest to niestety rozwiązanie doskonałe. Czasami na zmianę trzeba czekać godzinę lub dwie, po których następuje tak poważne zmęczenie, że muszę opuścić swojego partnera i znaleźć jakiś kącik, aby odpocząć. Czasami nawet siedząc z kimś przy komputerze, nie jestem w stanie zrobić nic poza przytakiwaniem. Najczęściej jednak moją typową reakcją w takiej sytuacji jest odzyskanie swojego typowego rytmu.

Kreatywne wejście

Istnieje jeszcze kilka innych rzeczy, które robię, żeby zapobiegać takim blokadom. Już dawno temu przekonałem się, że kreatywne wyjście zależy od kreatywnego wejścia.

Sporo czytam, i to na wiele różnych tematów. Czytam teksty poświęcone oprogramowaniu, polityce, biologii, astronomii, fizyce, chemii, matematyce i wielu innym zagadnieniom. Co więcej, uważam, że literatura science fiction najlepiej pobudza we mnie kreatywne wyjście.

W Twoim przypadku elementem pobudzającym może być coś całkiem innego. Być może dobra powieść detektywistyczna, poezja albo nawet romans. Jak sądzę, chodzi tu o to, że kreatywność tworzy kreatywność. Może też chodzić o swego rodzaju ucieczkę, eskapizm. Godziny, które spędzam z dala od moich normalnych problemów, stymulując się i wchłaniając różne kreatywne idee, wywołują niemal nieodpartą potrzebę samodzielnego tworzenia.

Nie działają na mnie jednak wszystkie rodzaje kreatywnego wejścia. Na przykład zupełnie nie pomaga mi oglądanie telewizji. Wypad do kina działa lepiej, ale tylko troszkę lepiej. Słuchanie muzyki nie pomaga mi w tworzeniu kodu, ale bardzo ułatwia tworzenie prezentacji, przemówień i filmów. Jednak ze wszystkich rodzajów kreatywnego wejścia nic nie działa na mnie lepiej niż stara dobra space opera.

Debugowanie

Jedna z najgorszych sesji debugowania w mojej karierze przytrafiła mi się w 1972 roku. Terminale podłączone do systemu księgowego firmy Teamsters zawieszały się raz lub dwa razy na dzień. Nie dało się jednak w żaden sposób wymusić tego stanu. Błąd nie był związany z konkretnym rodzajem terminala ani z określonymi aplikacjami. Nie miało też znaczenia, co użytkownik robił tuż przed zawieszeniem. W jednej chwili terminal działał doskonale, a w następnej nie reagował już na nic.

Zdiagnozowanie tego problemu zajęło nam całe tygodnie, w czasie których firma Teamsters stawała się coraz bardziej nerwowa. Za każdym razem osoba, której terminal się zawiesił, musiała skoordynować się z innymi użytkownikami, tak żeby zakończyli oni pracę, a następnie zadzwonić do nas z prośbą o restart. To był prawdziwy koszmar.

Pierwsze dwa tygodnie spędziliśmy, zbierając dane z wywiadów prowadzonych z osobami, którym przytrafiła się blokada. Pytaliśmy, co robiły w czasie wystąpienia blokady i wcześniej. Pytaliśmy, czy zauważyły cokolwiek na *swoich* zawieszonych terminalach. Wszystkie te pytania zadawaliśmy przez telefon, ponieważ wszystkie terminale były umiejscowione w centrum Chicago, a my pracowaliśmy 50 kilometrów na północ od miasta, w środku pola kukurydzy.

Nie mieliśmy żadnych protokołów, żadnych liczników, żadnych debuggerów. Jedynym dostępem, jaki mieliśmy do systemu, były światełka i przełączniki na przednim panelu. Mogliśmy zatrzymać komputer i przeglądać jego pamięć po jednym słowie. Nie mogliśmy tego jednak robić dłużej niż pięć minut, ponieważ firma Teamsters musiała korzystać ze swojego systemu.

Kilka dni poświęciliśmy na napisanie prostego inspektora działającego w czasie rzeczywistym, którym mogliśmy sterować z teletekstu ASR-33 służącego nam za konsolę. W ten sposób mogliśmy zaglądać do pamięci komputera w czasie pracy systemu. Dodaliśmy komunikaty protokołu, które w krytycznych momentach były wypisywane na teletekście. Przygotowaliśmy liczniki zliczające zdarzenia i zapamiętujące historię stanów, którą mogliśmy sprawdzać za pomocą inspektora. Oczywiście to wszystko zostało napisane od zera w asemblerze i było testowane wieczorami, gdy system nie był używany.

Terminale były sterowane przerwaniami, natomiast znaki przesyłane do nich były umieszczane w cyklicznych buforach. Za każdym razem, gdy port szeregowy kończył wysyłanie znaku, uruchamiane było przerwanie, w którym do wysłania był przygotowywany kolejny znak z cyklicznego bufora.

Ostatecznie okazało się, że terminale zawieszały się wtedy, gdy rozsynchronizowały się trzy zmienne zarządzające stanem cyklicznego bufora. Nie mieliśmy pojęcia, dlaczego tak się działo, ale to była już jakaś wskazówka. Gdzieś w 5 000 wierszy kodu superwizora znajdował się błąd powodujący nieprawidłowe działanie tych wskaźników.

Ta wiedza pozwalała nam też ręcznie odblokować terminale! Mogliśmy wpisać do tych trzech zmiennych domyślne wartości za pomocą inspektora, a terminale zaczynały znowu działać. W końcu napisaliśmy mały programik, który kontrolował wartość tych liczników i w razie stwierdzenia nieprawidłowości przywracał im wartości domyślne. Początkowo program ten był wywoływany przez naciśnięcie specjalnego przełącznika na panelu komputera za każdym razem, gdy ktoś z firmy Teamsters zgłaszał zawieszenie terminala. Później uruchamialiśmy go automatycznie co sekundę.

Po mniej więcej miesiącu z punktu widzenia firmy Teamsters sprawa zawieszających się terminali została zamknięta. Czasami jakiś terminal zatrzymywał się na mniej więcej pół sekundy, ale przy prędkości 30 znaków na sekundę nikt tego nawet nie zauważał.

Ale dlaczego te liczniki się rozsynchronizowywały? Miałem wtedy 19 lat i byłem zdeterminowany, żeby znaleźć przyczynę.

Kod superwizora został napisany przez Richarda, który teraz był już na studiach. Nikt z pozostałych członków zespołu nie znał tego kodu zbyt dobrze, ponieważ Richard był w tym względzie dość zaborczy. Ten kod był *jego*, a my mieliśmy się od niego trzymać z daleka. Teraz jednak Richarda nie było już w firmie, dlatego wyciągnąłem gruby wydruk programu i zacząłem przeglądać go strona po stronie.

Cykliczne kolejki w tym systemie były po prostu strukturami typu FIFO, czyli najzwyklejszymi kolejkami. Programy umieszczały znaki na jednym końcu kolejki, aż została ona zapełniona. Przerwania zdejmowały z kolei znaki z drugiego końca kolejki, za każdym razem gdy drukarka była gotowa na ich przyjęcie. Jeżeli kolejka była pusta, to drukarka się zatrzymywała. Drobny błąd sprawiał, że aplikacje były przekonane o tym, iż kolejka jest pełna, natomiast przerwania stwierdzały, że kolejka jest pusta.

Przerwania były wykonywane w innym „wątku" niż pozostały kod programu. Oznaczało to, że liczniki i inne zmienne aktualizowane zarówno przez przerwania, jak i pozostały kod musiały być odpowiednio zabezpieczone przed jednoczesnym zapisem. W naszym przypadku oznaczało to wyłączanie przerwań w pobliżu kodu manipulującego tymi zmiennymi. W tym momencie wiedziałem już, że muszę znaleźć w kodzie miejsce, które zmienia zawartość feralnych zmiennych, nie wyłączając uprzednio przerwań.

Dzisiaj mamy do dyspozycji całą paletę różnych narzędzi pozwalających wyszukać te miejsca w kodzie, które interesują się określoną zmienną. W ciągu kilku sekund można wyświetlić wszystkie podejrzane wiersze kodu. Znalezienie wycinka, w którym zapomniano o wyłączeniu przerwań, zajęłoby zaledwie minutę. Jednak w roku 1972 nie istniały takie narzędzia, a do dyspozycji miałem wyłącznie swoje oczy.

Dokładnie przeglądałem kod zapisany na każdej stronie, poszukując feralnych zmiennych. Niestety, były one używane właściwie *wszędzie*. Niemal na każdej stronie były w ten lub inny sposób modyfikowane. W wielu przypadkach przerwania nie były wyłączane, ponieważ

zmienne były tylko odczytywane, a przez to cała operacja była niegroźna. Problem polegał na tym, że w tym szczególnym asemblerze nie dało się stwierdzić, czy referencja zmiennej była niegroźnym odczytem, bez dokładniejszego sprawdzenia kodu. Po każdym odczycie zmiennej mogły nastąpić jej aktualizacja i zapisanie. Jeżeli zdarzyło się to w czasie, gdy przerwania były włączone, to zawartość zmiennej mogła zostać uszkodzona.

Dokładne sprawdzenie kodu zajęło mi kilka dni, ale w końcu znalazłem przyczynę. W samym środku kodu ukryło się jedno miejsce, w którym aktualizowana była jedna z trzech zmiennych bez uprzedniego wyłączenia przerwań.

Dokonałem wtedy pewnych obliczeń. Czas trwania podatności to mniej więcej dwie mikrosekundy. W firmie działało jakieś 12 terminali pobierających dane z prędkością 30 znaków na sekundę, czyli jedno przerwanie pojawiało się mniej więcej co 3 milisekundy. Biorąc pod uwagę wielkość superwizora, częstotliwość zegara procesora, można było się spodziewać, że terminal zostanie zablokowany jeden raz lub dwa razy dziennie. Bingo!

Usunąłem ten błąd, ale nigdy nie miałem odwagi wyłączyć automatu sprawdzającego i korygującego liczniki. Do dzisiaj nie mam pewności, czy w programie nie ukrywał się jeszcze inny błąd.

Czas debugowania

Z nieznanych mi powodów część programistów nie uznaje czasu poświęconego na debugowanie za czas tworzenia kodu. Twierdzą, że czas debugowania jest po prostu wymogiem natury, czyli czymś, co trzeba zrobić. Jednak z punktu widzenia firmy czas debugowania jest tak samo kosztowny jak czas tworzenia kodu, dlatego wszystko, co zrobimy, żeby uniknąć debugowania albo przynajmniej skrócić jego czas, będzie pozytywnym działaniem.

Dzisiaj spędzam na debugowaniu znacznie mniej czasu niż jeszcze 10 lat temu. Oczywiście nie mierzyłem różnicy, ale sądzę, że jest to mniej więcej jeden rząd wielkości. Tę radykalną redukcję czasu debugowania uzyskałem przez przyjęcie praktyki TDD (*Test Driven Development*), o której będę mówił w innym rozdziale.

Niezależnie od tego, czy stosujemy praktykę TDD, czy inne rozwiązania o podobnej skuteczności[3], na każdym zawodowym programiście ciąży zadanie dążenia do jak najskuteczniejszego redukowania czasu spędzanego na debugowaniu. Zerowy czas jest tutaj celem asymptotycznym, ale mimo to zawsze powinien być naszym celem.

Chirurdzy nie lubią ponownie otwierać swoich pacjentów, żeby poprawić coś, co zrobili źle. Prawnicy nie lubią powtarzać pozwów tylko dlatego, że przy poprzednim palnęli głupotę.

[3] Nie znam innej techniki o skuteczności zbliżonej do TDD, ale może wiesz coś, czego ja nie wiem.

Chirurg lub prawnik, który zbyt często popełnia takie błędy, nigdy nie będzie uznany za profesjonalistę. Podobnie programista robiący zbyt wiele błędów działa nieprofesjonalnie.

Wyznaczanie sobie rytmu

Tworzenie oprogramowania to bardziej maraton niż sprint. Tego wyścigu nie da się wygrać, biegnąc od samego początku tak szybko, jak się da. Kluczem do zwycięstwa jest zachowywanie niezbędnych zasobów i wyznaczanie sobie odpowiedniego rytmu. Maratończyk dba o siebie zarówno przed wyścigiem, jak i *podczas* niego. Zawodowi programiści w podobny sposób zachowują swoją energię i kreatywność.

Wiedzieć, kiedy odejść

Nie możesz wrócić do domu, dopóki nie rozwiążesz tego problemu? Oczywiście, że możesz, a najprawdopodobniej tak należałoby zrobić. Kreatywność i inteligencja to bardzo ulotne stany umysłu. Jeżeli uginasz się od zmęczenia, to na pewno się ich pozbywasz. Jeżeli wtedy zmuszasz swój przemęczony mózg do pracy w późnych godzinach nocnych, próbując rozwiązać jakiś problem, to tylko zwiększasz swoje zmęczenie, a tym samym zmniejszasz szanse na to, że prysznic lub samochód pomogą Ci znaleźć rozwiązanie.

Jeżeli utkniesz w pracy i zmęczenie nie pozwoli Ci dalej pracować, to spróbuj się na chwilę odłączyć. Daj swojej podświadomości szansę na podjęcie próby rozwiązania problemu. Jeżeli odpowiednio zadbasz o swoje zasoby, to uzyskasz więcej w krótszym czasie przy znacznie mniejszym wysiłku. Odpowiednio wyznaczaj rytm sobie i swojemu zespołowi. Naucz się stosować wzorce kreatywności oraz błyskotliwości i staraj się je wykorzystać do swoich celów, a nie walczyć przeciwko nim.

Jazda do domu

Miejscem, w którym udało mi się rozwiązać zadziwiająco wiele problemów, jest mój samochód wiozący mnie z pracy do domu. Kierowanie autem wymaga zaangażowania wielu niekreatywnych zasobów umysłowych. Wykonywaniu tego zadania musisz poświęcić swoje oczy, ręce oraz część umysłu. Oznacza to, że musisz odłączyć się od problemów z pracy. Takie właśnie *odłączenie* pozwala Twojemu umysłowi na poszukiwanie rozwiązań w inny, znacznie bardziej kreatywny sposób.

Prysznic

Równie wielką liczbę problemów udało mi się rozwiązać pod prysznicem. Być może ten strumień wody wcześnie rano pobudza mnie na tyle, że mogę ponownie przeanalizować wszystkie rozwiązania, na jakie wpadł mój umysł w czasie, gdy spałem.

Pracując nad jakimś problemem, czasami wchodzisz w niego tak głęboko, że nie jesteś w stanie dostrzec wszystkich możliwych opcji. Pomijasz eleganckie rozwiązania, ponieważ kreatywna część Twojego umysłu jest tłumiona przez intensywność skupienia na zadaniu. Czasami najlepszym sposobem na rozwiązanie problemu jest powrót do domu, zjedzenie kolacji, obejrzenie czegoś w telewizji, pójście do łóżka i skorzystanie z prysznica po przebudzeniu się następnego dnia.

Spóźnienia

Spóźnienia *będą* Ci się przytrafiać. To zdarza się najlepszym spośród nas. Zdarza się tym najbardziej oddanym pracy. Czasami po prostu nasze szacunki nie są dość dokładne i wtedy się spóźniamy.

Właściwą metodą radzenia sobie z opóźnieniami jest wczesne ich wykrywanie i otwarte informowanie o nich. Nie ma nic gorszego niż utrzymywanie do samego końca, że zdążysz na czas, i sprawienie zawodu w ostatnim momencie. Tak się *nie* robi. Zamiast tego *regularnie* mierz swoje postępy i porównuj je z celem, a następnie podawaj trzy[4] wynikające z tego porównania daty: najlepszy przypadek, normalny przypadek i najgorszy przypadek. *Do tych szacunków nie dodawaj swoich nadziei!* Trzy wyliczone daty zaprezentuj całemu zespołowi i wszystkim zainteresowanym, a następnie codziennie je aktualizuj.

Nadzieja

Co zrobić, jeżeli z tych szacunków wynika, że *możesz* przestrzelić termin? Dla przykładu załóżmy, że za 10 dni są targi i do tego czasu musimy mieć gotowy produkt. Załóżmy też, że Twoje trzy szacowane daty ukończenia funkcji, nad którą pracujesz, to 8/12/20.

Nie opieraj się na nadziei, że zdołasz zrobić wszystko w 10 dni! Nadzieja jest zabójcą projektów. Nadzieja niszczy plany i rujnuje reputację. Nadzieja wpędzi Cię w poważne tarapaty. Jeżeli targi zaczynają się za 10 dni, a Twoja szacunkowa, normalna data ukończenia to 12 dni, to znaczy, że *nie* zdążysz na czas. Upewnij się, że cały zespół i udziałowcy znają szczegóły sytuacji, i nie odpuszczaj, dopóki nie powstanie plan awaryjny. Nie dawaj innym fałszywej nadziei.

Pośpiech

Co zrobić, jeżeli przełożony usadza Cię i prosi o próbę dotrzymania terminu? Co zrobić, jeżeli kierownik nastaje, żeby „zrobić, co trzeba”? *Nie koryguj swoich szacunków!* Twoje pierwotne szacunki będą o wiele dokładniejsze niż wszelkie zmiany, jakie wprowadzisz do nich podczas takiej konfrontacji z szefem. Powiedz szefowi, że wszelkie możliwości zostały

[4] Więcej na ten temat opowiem w rozdziale „Szacowanie”.

już wzięte pod uwagę (bo tak się stało) i jedyną metodą poprawienia planu jest redukcja zakresu projektu. *Nie ulegaj pokusom przyspieszenia prac.*

Biada biednemu programiście, który ugnie się pod presją i zgodzi się *spróbować* dotrzymać terminu. Taki programista zacznie wykorzystywać skróty i pracować w nadgodzinach w próżnej nadziei na cud. To jest najprostsza recepta na katastrofę, ponieważ daje to zespołowi i udziałowcom fałszywą nadzieję na sukces. Przez to nikt może nie zauważać problemu, co opóźnia podjęcie niezbędnych, choć trudnych decyzji.

Pośpiech nie jest żadnym rozwiązaniem. Nie jesteś w stanie szybciej pisać kodu. Nie zmusisz się do szybszego rozwiązywania problemów. Jeżeli zaczniesz podejmować takie próby, to tylko spowolnisz swoje działania, tworząc przy okazji bałagan, który zacznie spowalniać pozostałych członków zespołu.

Musisz zatem udzielić szefowi i zespołowi uczciwej odpowiedzi, która pozbawi ich fałszywej nadziei.

Nadgodziny

Zdarza się, że szef mówi Ci: „A jeżeli popracujesz dwie dodatkowe godziny dziennie? A jeżeli przyjdziesz do pracy w sobotę? Przecież musi być jakiś sposób, żeby wygospodarować dodatkowe godziny i przygotować tę funkcję na czas".

Nadgodziny mogą być dobrym rozwiązaniem, a czasami są nawet nieodzowne. Czasami można dotrzymać niemożliwego terminu, pracując po 10 godzin dziennie, a nawet w jedną sobotę lub dwie. To jednak bardzo ryzykowne założenie. Nie jesteś w stanie wykonać 20% więcej pracy, poświęcając na to 20% więcej czasu. Co więcej, nadgodziny *na pewno* stracą jakikolwiek sens, jeżeli taki tryb pracy potrwa dłużej niż dwa lub trzy tygodnie.

Oznacza to, że nie należy zgadzać się na nadgodziny, chyba że (1) możesz sobie na to pozwolić, (2) jest to rozwiązanie krótkoterminowe, maksymalnie na dwa tygodnie i (3) *Twój szef ma już plan awaryjny* na wypadek, gdyby praca w nadgodzinach nie wystarczyła.

Ten ostatni warunek jest tutaj najważniejszy. Jeżeli Twój szef nie jest w stanie stwierdzić, co zrobi, jeżeli nadgodziny nie przyniosą oczekiwanych efektów, to nie zgadzaj się na dodatkowe godziny pracy.

Fałszywa dostawa

Ze wszystkich nieprofesjonalnych zachowań, na jakie może sobie pozwolić programista, chyba najgorsze jest twierdzenie, że prace są zakończone, choć ma się pełną świadomość tego, że tak nie jest. Czasami jest to najzwyklejsze kłamstwo, co samo w sobie jest złe. Jednak znacznie bardziej podstępnym przypadkiem jest próba stworzenia nowej definicji

„gotowego”. Próbujemy się wtedy przekonać, że jesteśmy *wystarczająco* gotowi, i od razu przechodzimy do następnego zadania. Uznajemy, że praca, która została jeszcze do wykonania, może zostać zrobiona później, kiedy będzie na to czas.

Takie zachowanie jest bardzo zaraźliwe. Jeżeli robi tak jeden programista, to pozostali na pewno to zauważą i zaczną postępować podobnie. Jeden z nich jeszcze bardziej rozciągnie definicję tego, co „gotowe”, a reszta się do niej szybko przystosuje. Miałem nieprzyjemność oglądać już ekstremalne przypadki takiego działania. Jeden z moich klientów definiował „gotowe” jako „zapisane w systemie”. Kod nawet nie musiał się kompilować. Jeżeli nic nie musi działać, to bardzo łatwo powiedzieć „gotowe”!

Gdy zespół wpada w tę pułapkę, menedżerowie słyszą, że wszystko w projekcie jest w porządku. Raporty o stanie wskazują, że wszyscy pracują zgodnie z planem. Przypomina to piknik ślepców na torach kolejowych. Nikt nie zauważa pociągu towarowego wiozącego niedokończoną pracę, aż w końcu jest za późno.

Definicja „gotowego”

Problemu fałszywych dostaw można uniknąć, tworząc całkowicie niezależną definicję „gotowego”. Najlepszą metodą jej uzyskania jest współpraca analityków i testerów w celu przygotowania zautomatyzowanych testów akceptacyjnych[5], które muszą zostać zaliczone, aby dana funkcja mogła zostać uznana za gotową. Takie testy powinny zostać zapisane w wyspecjalizowanym języku, takim jak FitNesse, Selenium, RobotFX, Cucumber lub podobnym. Poszczególne testy muszą być zrozumiałe dla udziałowców projektu oraz dla biznesmenów, a przede wszystkim muszą być uruchamiane odpowiednio często.

Pomoc

Programowanie to *ciężka* praca. Im ktoś jest młodszy, tym trudniej mu w to uwierzyć. W końcu to tylko zbiór instrukcji `if` i `while`. Jednak zbierając doświadczenie, zaczynasz dostrzegać, że najważniejszy jest sposób, w jaki łączone są ze sobą te wszystkie instrukcje. Nie możesz ich umieścić po prostu w sekwencji i mieć nadzieję, że taka konstrukcja zadziała. Konieczne jest ostrożne dzielenie całego systemu na mniejsze, zrozumiałe części, które mają ze sobą jak najmniej wspólnego. I to właśnie jest najtrudniejsze.

Programowanie jest tak trudne, że jedna osoba nie jest w stanie zrobić tego dobrze. Niezależnie od tego, jak wielki masz talent, to na pewno skorzystasz na przemyśleniach i pomysłach innych programistów.

[5] Zajrzyj do rozdziału 7., „Testy akceptacyjne”.

Pomaganie innym

Z tego właśnie powodu każdy programista powinien w miarę możliwości pomagać innym. Izolowanie się w swoim boksie lub biurze i odmawianie odpowiedzi na pytania zadawane przez innych jest naruszeniem etyki zawodowca. Twoja praca nie jest tak ważna, że nie możesz poświęcić chwili na to, by pomóc kolegom. Co więcej, każdy profesjonalista podejmuje honorowe zobowiązanie do udzielania pomocy wszędzie tam, gdzie jest ona niezbędna.

Nie oznacza to, że nie potrzeba Ci czasu spędzonego w samotności. Oczywiście, że tego potrzebujesz. Ale musisz o tym otwarcie i spokojnie informować. Na przykład możesz przekazać zespołowi, że pomiędzy godziną 10 rano a południem nie należy Ci przeszkadzać, ale już w godzinach od 13 do 15 Twoje drzwi są otwarte i każdy może przyjść z pytaniami.

Musisz mieć pełną świadomość tego, w jakim stanie są członkowie Twojego zespołu. Jeżeli zauważasz, że ktoś ma problemy, to zaoferuj mu swoją pomoc. Możesz się naprawdę zdziwić, jak wielki skutek może odnieść Twoja pomoc. Nie chodzi o to, że ta inna osoba jest mniej inteligentna od Ciebie. Po prostu spojrzenie z innej perspektywy może mieć ogromne znaczenie przy rozwiązywaniu problemów.

Próbując komuś pomóc, najlepiej jest usiąść razem z nim i razem zacząć tworzyć kod. Zaplanuj na ten cel jakąś godzinę, a może i więcej. Być może potrzeba będzie mniej czasu, ale też nie chodzi o popędzanie kogokolwiek. Uznaj dane zadanie za własne i przyłóż się do niego solidnie. Całkiem możliwe, że na końcu nauczysz się więcej, niż dasz od siebie.

Przyjmowanie pomocy

Jeżeli ktoś proponuje Ci swoją pomoc, okaż mu wdzięczność. Przyjmij ją i staraj się ją jak najlepiej wykorzystać. *Nie chroń swojego terytorium.* Nie rezygnuj z oferowanej pomocy, nawet jeżeli masz nóż na gardle. Poświęć na ten cel jakieś pół godziny. Jeżeli przez ten czas okaże się, że kolega nie jest w stanie Ci pomóc, to pięknie mu podziękuj i zakończ sesję. Pamiętaj, że tak jak honor zobowiązuje Cię do udzielania pomocy, tak i zobowiązuje Cię do jej przyjmowania.

Naucz się też *prosić* o pomoc. Jeżeli utkniesz, dopadnie Cię zamroczenie albo po prostu nie będziesz w stanie ogarnąć problemu, to poproś kogoś o pomoc. Jeżeli siedzisz w pokoju z całym zespołem, możesz po prostu wstać i powiedzieć: „Potrzebuję pomocy”. W innych okolicznościach wykorzystaj Tweetera, e-mail albo telefon stojący na biurku i poproś kogoś o pomoc. Powtarzam raz jeszcze, że jest to kwestia etyki zawodowej. Zdecydowanie nieprofesjonalne jest stanie w miejscu, podczas gdy pomoc jest tak łatwo dostępna.

W tym momencie może Ci się wydawać, że zaraz chóralnie zaśpiewam *Kumbaya*, a w tle różowe króliczki będą wskakiwać na grzbiety jednorożców i wspólnie poszybujemy ponad tęczą nadziei i zmian. No, nie do końca. Okazuje się, że programiści *bywają* aroganckimi,

pochłoniętymi sobą introwertykami. Tego zawodu nie wybiera się dlatego, że tak bardzo lubi się *ludzi*. Większość z nas została programistami, ponieważ wolimy koncentrować się na sterylnych szczegółach, jednocześnie przerzucając różne pomysły, i w ogóle udowadniać, że mamy mózgi wielkości małej planety, a jednocześnie unikać skomplikowanej interakcji z *innymi ludźmi*.

To oczywiście stereotyp i wielka generalizacja, z wieloma różnymi wyjątkami. Ale rzeczywistość jest taka, że programiści raczej nie są najlepszymi współpracownikami[6]. A mimo to współpraca jest bardzo istotnym czynnikiem skutecznego programowania. Oznacza to, że choć dla wielu z nas współpraca nie jest instynktowna, musimy wyrobić sobie dyscyplinę, która będzie nas do tego zmuszała.

Mentor

W dalszej części książki poświęcę temu tematowi cały rozdział. Na razie powiem tylko tyle, że szkolenie mniej doświadczonych programistów jest zadaniem tych o większym doświadczeniu. I nie liczą się tutaj kursy programowania. Nie liczą się książki. Nic nie jest w stanie szybciej wynieść młodego programisty na wyżyny wydajności niż własna motywacja i pomoc starszych kolegów. I znów poświęcenie czasu, żeby wziąć młodych pod swoje skrzydła i odpowiednio ich szkolić, należy do etyki zawodowej doświadczonych programistów. Z tego wynika też, że zadaniem młodszych programistów jest poszukiwanie możliwości skorzystania na doświadczeniu starszych kolegów.

Bibliografia

[**Martin09**]: Robert C. Martin, *Czysty kod. Podręcznik dobrego programisty*, Gliwice, Helion, 2009.

[**PPP2002**]: Robert C. Martin, *Agile Software Development. Principles, Patterns, and Practices*, Upper Saddle River, NJ: Prentice Hall, 2002.

[6] To znacznie częściej sprawdza się w przypadku mężczyzn niż kobiet. Prowadziłem kiedyś bardzo interesującą rozmowę z @desi (Desi McAdam, założycielką witryny DevChix) na temat tego, co motywuje kobiety programistki. Powiedziałem, że gdy udaje mi się w końcu uruchomić prawidłowo działający program, czuję się, jakbym powalił wielką bestię. Ona odpowiedziała, że dla niej i wielu innych kobiet, z którymi rozmawiała, akt tworzenia kodu jest aktem wychowawczej kreacji.

Skorowidz

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

T

U

W

X

Z

Robert C. Martin, znany jako Uncle Bob, to jeden z prawdziwych gwiazdorów branży IT, człowiek o niezwykłej charyzmie, rewelacyjnym podejściu do słuchaczy i poczuciu humoru. O jego czas wciąż biją się konferencje branżowe. Poza działalnością ekspercką Martin zajmuje się pisaniem książek — jest autorem m.in. znanego każdemu programiście *Czystego kodu*. Książka, którą trzymasz w rękach, jest udaną kontynuacją tamtej pozycji.

W trakcie lektury dowiesz się, jakie cechy charakteryzują profesjonalnego programistę, a jest ich sporo! W pierwszej kolejności musisz nauczyć się mówić „nie". Są też sytuacje, kiedy trzeba powiedzieć „tak" — dowiesz się, kiedy i jak to robić. Ponadto poznasz najlepsze techniki zarządzania czasem oraz przekonasz się, jak presja, zmęczenie i pośpiech wpływają na jakość Twojego kodu. W kolejnych rozdziałach Robert C. Martin zapozna Cię z różnymi sposobami podejścia do testowania kodu oraz współpracy między programistami a innymi ludźmi. Książka ta jest długo wyczekiwaną pozycją na rynku — nie pozwól, żeby ktoś miał ją przed Tobą!

Zobacz, jak Uncle Bob:

- radzi sobie z presją
- mówi „nie" i „tak"
- zarządza czasem
- tworzy kod wysokiej jakości

Lektura obowiązkowa dla każdego programisty!

Helion
księgarnia internetowa
http://helion.pl
zamówienia telefoniczne
0 801 339900
0 601 339900

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

ISBN 978-83-283-3131-0

Informatyka w najlepszym wydaniu

cena: 39,90 zł