Przedmowa

Programista, koder, deweloper – to tylko niektóre z nazw używanych do opisania ludzi tworzących oprogramowanie, ale co to znaczy być inżynierem oprogramowania? Niezależnie od tego, jak zwykle przebiega kształcenie, zostanie inżynierem oprogramowania to znacznie więcej, niż umiejętność tworzenia programów poprawnych syntaktycznie.

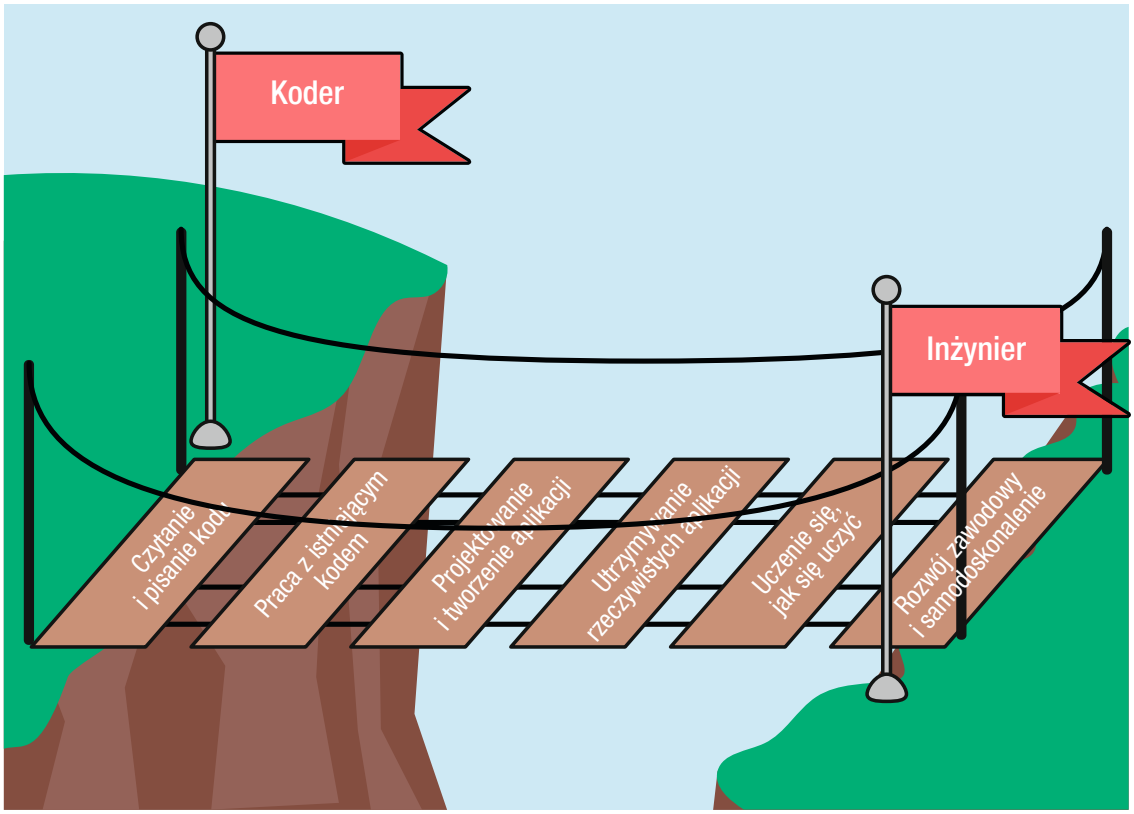
Bootcampy i uniwersytety zazwyczaj koncentrują się na mechanicznych aspektach pisania kodu, kształcąc osoby biegłe w programowaniu. Wiedza konieczna dla odniesienia sukcesu jako inżynier oprogramowania znacząco wykracza poza naukę języka programowania. Musisz doskonale znać cały cykl życia produktu programistycznego. Musisz mieć wszechstronną wiedzę wykraczającą poza składnię i gramatykę języków programowania; opanować testowanie, architekturę, modelowanie i wiele innych. Musisz wiedzieć, jak efektywnie pracować z istniejącym kodem oraz jak niezawodnie i wielokrotnie wdrażać go w środowisku produkcyjnym. Żeby osiągnąć sukces, awansować i pracować przy najciekawszych projektach, musisz wyjść poza samo pisanie kodu i stosować zasady inżynierii w całym procesie wytwarzania. Nawet w erze sztucznej inteligencji i agentowych narzędzi do kodowania musisz dobrze znać podstawy, aby prawidłowo posługiwać się narzędziami AI.

Istnieje wiele dróg do zostania inżynieriem oprogramowania, od studiów informatycznych, przez intensywne bootcampy i samodzielną naukę (rysunek 0-1). Na początku kariery może się wydawać, że dopiero wkraczasz do znacznie większego świata. To może onieśmielać, wiemy, bo sami to przeżyliśmy! Niezależnie od przygotowania, jako świeżo upieczony praktyk szybko odkryjesz, że istnieje mnóstwo kluczowych zagadnień, o których nic nie wspominano w standardowym programie studiów, a także umiejętności, których ci brakuje, co uniemożliwia awans na wyższe stanowiska.

Istnieje duża luka pomiędzy tym, czego uczono cię na studiach lub bootcampach, a tym, co musisz umieć, aby odnieść sukces jako inżynier oprogramowania[1]. (Firmy technologiczne, takie jak Thoughtworks, mają rygorystyczne praktyki rekrutacyjne, a mimo to wysyłają świeżo zatrudnionych pracowników na szkolenia i nie są w tym osamotnione). Ta książka ma wypełnić tę lukę, dając ci kontekst i podstawy konieczne do wykreślenia swojej ścieżki kariery i pomagać w identyfikowaniu możliwości rozwoju osobistego i zawodowego. Potraktuj ją jak poradnik wdrożeniowy dla początkującego inżyniera

1 Jest to jeden z głównych czynników, dla których syndrom oszusta (patrz rozdział 14) jest tak rozpowszechniony w branży oprogramowania.

oprogramowania. Ta książka ma być przewodnikiem w tej podróży, pokazując rzeczy, o których mogłeś nie wiedzieć, że ich nie znasz.



Rysunek 0-1 *Istnieje luka pomiędzy tym, czego uczy się początkujących programistów, a tym, czego potrzebują, aby odnosić sukcesy i awansować w swojej karierze*

Dla kogo jest ta książka

Ta książka powstała z myślą o nowych inżynierach oprogramowania. Naszym celem jest pokazanie szerszego obrazu tego, co potrzebne, aby zostać prawdziwym inżynierem, wykraczającym poza „samo kodowanie" i co jest potrzebne, aby rozwijać karierę.

Jednak podstawy omawiane w tej książki nie są przeznaczone tylko dla nowicjuszy. Doświadczeni inżynierowie również mogą skorzystać z doskonalenia kluczowych umiejętności, szczególnie gdy pragną awansować i objąć wyższe stanowiska. Zbudowanie solidnego zestawu umiejętności jest tym, co otwiera możliwość zdobycia większej odpowiedzialności i awansu.

Czego się nauczysz

Czym są te umiejętności, których potrzebujesz, aby rozkwitnąć i odnosić sukcesy? Co odróżnia nowicjusza od doświadczonego inżyniera oprogramowania? Od czytania kodu, przez pisanie kodu, który jest czytelny, testowanie, zachowanie równowagi pomiędzy

pracą a życiem, po nauczenie się, jak się uczyć – powiemy ci wszystko, co powinieneś wiedzieć (a nawet kilka rzeczy, o których nie miałeś pojęcia, że powinieneś je znać). Większość inżynierów uczy się tego metodą prób i błędów, co niekiedy drogo kosztuje. Ale nie musi tak być! Te umiejętności to fundamenty inżynierii oprogramowania, które otworzą ci drogę do udanej kariery i – w przeciwieństwie do modnych dziś frameworków czy popularnych języków – pozostaną z tobą na całe życie.

Choć książka nie ma być wyczerpującym przewodnikiem po każdym zagadnieniu, pokaże uniwersum tematów wchodzących w skład inżynierii oprogramowania, tak byś miał dość informacji, aby zrozumieć podstawowe koncepcje.

W książce dzielimy się historiami i doświadczeniami z naszego życia zawodowego, dając ci okazję do uczenia się na naszych błędach. Każdy rozdział kończy się zestawieniem przydatnych źródeł (głównie innych książek) oraz praktycznymi ćwiczeniami, pozwalającymi utrwalić zdobytą wiedzę. A jeśli któryś rozdział poruszy cię silniej, zachęcamy, aby zagłębić się w ten temat i zbadać go bliżej!

Zawartość książki

Tak jak nie ma jednej drogi do zostania inżynierem oprogramowania, jest też wiele sposobów podejścia do tego materiału. Choć większość czytelników zapewne zacznie od przeczytania książki od deski do deski, możesz także „wybrać własną przygodę", wybierając rozdziały na podstawie tego, co interesuje cię w tym momencie. W miarę postępów kariery, wchodzenia w nowe projekty i napotykania nowych, powiązanych zagadnień, możesz zechcieć wrócić do konkretnych przykładów po wskazówki.

Książka podzielona jest na cztery części. Część 1 obejmuje podstawowe umiejętności, niezbędne, by stać się inżynierem oprogramowania. Część 2 bada różne praktyki techniczne spotykane w różnych projektach, zaś w części 3 zagłębimy się w niuanse projektowania i budowania oprogramowania. Na koniec część 4 koncentruje się na rozwoju kariery i krokach, które możesz podejmować, aby zachować kontrolę nad swoją podróżą.

Oto, co znajdziesz w tej książce:

Część 1: Podstawowe umiejętności

- Rozdział 1, „Od programisty do inżyniera", omawia różne ścieżki prowadzące do zostania inżynierem oprogramowania i zwraca uwagę na ważność umiejętności podstawowych.
- Rozdział 2, „Czytanie kodu", dotyczy tego, że inżynierowie oprogramowania spędzają więcej czasu na czytaniu kodu, niż na jego pisaniu. Zawiera strategie poruszania się po nieznanym kodzie.
- Rozdział 3, „Pisanie kodu", pokazuje, jak ważne jest pisanie kodu, aby był czytelny i łatwy w utrzymaniu. Mówiąc w skrócie, ilustruje, dlaczego ważniejsze jest pisanie dla (innego) programisty, niż dla komputera.

Część 2: Praktyki techniczne

- Rozdział 4, „Modelowanie", zagłębia się w rolę modelowania oprogramowania, czyli diagramów i schematów, z którymi spotykałeś się (albo spotkasz) w wielu projektach, jako kluczowego aspektu komunikacji pomiędzy inżynierami oprogramowania.
- Rozdział 5, „Zautomatyzowane testowanie", omawia zalety automatycznego testowania dla jakości kodu i poczucia pewności siebie.
- Rozdział 6, „Badanie i modyfikowanie nieznanych systemów", pokazuje strategie umożliwiające efektywną pracę z istniejącymi bazami kodu. Niezależnie od tego, czy dołączasz do zespołu pracującego nad jakimś projektem od pewnego czasu, czy utrzymujesz odziedziczony system, te umiejętności pozwolą ci wnosić wkład do dowolnej bazy kodu.

Część 3: Projektowanie i tworzenie aplikacji

- Rozdział 7, „Projektowanie interfejsu użytkownika", to przegląd projektowania interfejsów użytkownika. Wyjaśnia, dlaczego zrozumienie użytkownika i stosowanie zasad projektowych pomoże zagwarantować, że twoje oprogramowanie zaspokoi potrzeby użytkowników.
- Rozdział 8, „Praca z danymi", poświęcony jest pracy z danymi, czyli temu, co w jakiś sposób realizuje niemal każda aplikacja. Obejmuje podstawowe umiejętności niezbędne do podejmowania świadomych decyzji dotyczących danych.
- Rozdział 9, „Architektura oprogramowania", bada złożoności architektur oprogramowania i zapewnia wgląd w kompromisy architektoniczne, nieuniknione w każdym projekcie programistycznym.
- Rozdział 10, „Do produkcji", omawia wprowadzanie kodu do środowiska produkcyjnego. Przedstawiamy tu nieprzewidywalność rzeczywistych użytkowników i środowisk, a także szkicujemy strategie budowania kodu gotowego do produkcji.

Część 4: Rozwój osobisty

- Rozdział 11, „Zwiększanie produktywności", powraca do codziennej pracy i tego, jak ważne jest budowanie osobistego zestawu narzędzi.
- Rozdział 12, „Nauka uczenia się", omawia ważność ciągłej nauki. Ten rozdział oferuje strategie dotrzymania kroku nieustannym zmianom.
- Rozdział 13, „Kompetencje miękkie w świecie techniki", zagłębia się w ważność kompetencji miękkich dla twojej kariery. Omawiamy tu komunikację ze współpracownikami, wywieranie wpływu i zarządzanie interesariuszami.
- Rozdział 14, „Zarządzanie karierą", koncentruje się na planowaniu i realizowaniu kariery inżyniera oprogramowania. Zajmiemy się tu odkrywaniem, co naprawdę cię pasjonuje, badaniem różnych ścieżek kariery i świadomym zdobywaniem umiejętności. Omawiamy tu również budowanie sieci kontaktów zawodowych, sposoby

odnoszenia sukcesów na rozmowach kwalifikacyjnych i dbanie o równowagę pomiędzy pracą a życiem.

- Rozdział 15, „Inżynier oprogramowania ze wsparciem AI” porusza jeden z najbardziej przełomowych tematów we współczesnej branży oprogramowania: sztuczną inteligencję.

Choć sztucznej inteligencji poświęcony jest odrębny rozdział, jej ogromny wpływ na cykl rozwojowy oprogramowania oznacza, że wzmianki i uwagi na temat AI znajdziesz w całej książce.

Konwencje użyte w tej książce

W książce zastosowano następujące konwencje typograficzne:

Kursywa

Wskazuje nowe terminy, adresy URL, adresy e-mail, nazwy plików i rozszerzenia plików.

`Stała szerokość`

Stosowana do listingów programów, a także w akapitach, aby odwoływać się do elementów programu, takich jak nazwy zmiennych lub funkcji, bazy danych, typy danych, zmienne środowiskowe, instrukcje i słowa kluczowe.

`Stała szerokość z kursywą`

Oznacza tekst, który powinien zostać zastąpiony wartościami podanymi przez użytkownika lub wartościami określonymi przez kontekst.



Ten znak oznacza uwagę ogólną.



Ten znak oznacza wskazówkę lub alternatywny sposób realizacji zadania.



Ten element wskazuje ostrzeżenie lub przestrogę.

Używanie przykładów kodu

Uzupełniający materiał (przykłady kodu, ćwiczenia itp.) są dostępne do pobrania pod adresem *https://github.com/Fundamentals-of-Software-Engineering/book*.

W przypadku pytań technicznych lub problemów z wykorzystaniem przykładów kodu, prosimy o przesłanie wiadomości e-mail na adres *support@oreilly.com*.

Niniejsza książka ma służyć pomocą w wykonywaniu pracy. Ogólnie rzecz biorąc, jeśli przykładowy kod jest oferowany z książką, można go używać we własnych programach i dokumentacji. Nie trzeba się z nami kontaktować w celu uzyskania pozwolenia, o ile nie reprodukuje się znacznej części kodu. Na przykład pisanie programu z zastosowaniem kilku fragmentów kodu z niniejszej książki nie wymaga zgody. Sprzedaż lub dystrybucja przykładów z książek O'Reilly wymaga zgody. Udzielenie odpowiedzi na pytanie przez cytowanie niniejszej książki i cytowanie przykładowego kodu nie wymaga zgody. Wcielenie znaczącej ilości kodu przykładowego z niniejszej książki do dokumentacji własnego produktu wymaga zgody.

Doceniamy, ale na ogół nie wymagamy, atrybucji. Atrybucja zazwyczaj obejmuje tytuł, autora, wydawcę i ISBN (oryginalnego wydania angielskiego). Na przykład: „*Fundamentals of Software Engineering* by Nathaniel Schutta and Dan Vega (O'Reilly). Copyright 2026 Code Monkey LLC and Nathaniel Schutta, 978-1-098-14323-7."

W razie uznania, że korzystanie z przykładów kodu wykracza poza dozwolony użytek lub zgodę podaną powyżej, prosimy o kontakt z nami: *permissions@oreilly.com*.

Jak się z nami skontaktować

Komentarze i pytania dotyczące niniejszej książki prosimy kierować do wydawcy:

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-889-8969 (Stany Zjednoczone lub Kanada)

707-827-7019 (numer międzynarodowy lub lokalny)

707-829-0104 (faks) *support@oreilly.com*

https://oreilly.com/about/contact.html

Mamy stronę internetową dla niniejszej książki, na której zamieszczamy erratę[2], przykłady i wszelkie dodatkowe informacje. Dostęp do tej strony można uzyskać pod adresem *https://oreil.ly/fundamentals-of-software*.

2 Poprawki potwierdzone w chwili redagowania polskiego wydania tej książki (według stanu z 30.08.2026 r.) zostały już uwzględnione, ale mogą pojawić się nowsze, zatem warto zaglądać do tej strony (przyp. red. wydania polskiego).

Podziękowania

Gdyby sądzić tylko po okładce, każda książka wydaje się być wyłącznym dziełem autora lub dwóch, ale w rzeczywistości są to efekty pracy rozległych zespołów. Od redaktorów, przez recenzentów, po wybaczających członków rodziny, do powstania książki przyczynia się wiele rąk. Jesteśmy niewymownie wdzięczni za wsparcie, cierpliwość i wskazówki naszych cudownych redaktorek: Rity, Virginii, Louise i całej reszty zespołu O'Reilly. Nigdy nie traciłyście wiary i zrobiłyście wszystko, co w waszej mocy, opiekując się tymi dwoma dzikimi kotami. Dziękujemy bardzo!

Książka nie byłaby nawet w połowie tak dobra bez naszych recenzentów; ich czas i poświęcenie to dar, za który nie zdołamy się szybko odwdzięczyć. Książka jest znacznie lepsza dzięki waszym komentarzom, pytaniom i przykładom z życia. Christopher M. Judd zapewne zasługuje na uznanie za współautora za jego prorocze spostrzeżenia, które bezpośrednio wpływały na kierunek kilku rozdziałów. Jesteśmy wdzięczni za czas, jaki dr Venkat Subramaniam poświęcił na ukształtowanie tego, co masz w rękach. Dziękujemy też Chrisowi Kramerowi za jego rady dotyczące AI oraz Premanandowi Chandrasekaranowi za wniesienie na te strony swoich wieloletnich doświadczeń. Na szczególne podziękowania zasługuje James Erler, który nie tylko jest świetnym studentem, ale także pilnym recenzentem. Chcemy również wyrazić wdzięczność Raju Gandhi, Sruthi Sathyamoorthy i Muruganowi Lakshmananowi za ich czas i wkład.

Chcielibyśmy również podziękować wielu programistom, architektom i innym specjalistom, z którymi współpracowaliśmy w trakcie naszej kariery. Nie sposób wymienić was wszystkich z imienia i nazwiska, ale ukształtowaliście nas w ludzi, którymi jesteśmy dzisiaj. Chcielibyśmy podziękować Jakubowi Pilimonowi, który odegrał kluczową rolę w kształtowaniu propozycji książki (przy espresso), a także tematyki pierwszych rozdziałów. Choć nie wszyscy to wiedzą, znaczna część tej książki powstała jako zajęcia online na platformie O'Reilly. Tym z was, którzy uczestniczyli w tych sesjach, dziękujemy za pomoc w opracowaniu tego materiału!

Tu Nate. Chciałbym wykorzystać tę chwilę, by podziękować mojemu drogiemu przyjacielowi, współkonspiratorowi, partnerze do golfa, a teraz również współautorowi, Danowi Vega. Dzięki za ciężką pracę, historie i pomysły. Wziąłeś luźny szkielet paru luźno powiązanych myśli i przekształciłeś je w moje ulubione rozdziały. Mam nadzieję, że niedługo znów uda nam się umówić na partyjkę golfa! Jestem szczęśliwy, że twoje panie zgodziły się, abyś przeznaczył trochę swojego cennego czasu na ten szalony projekt. Dziękuję, przyjacielu!

Chciałbym też podziękować mojej ukochanej rodzinie, Christine i Everettowi, za ich cierpliwość i zrozumienie w długim, często zagmatwanym procesie pisania kolejnej książki. Przepraszam za opóźnione kolacje i brak czasu dla rodziny, ale mam nadzieję, że powstała książka to wynagrodzi w jakimś stopniu. Nie mogę obiecać, że to ostatni raz, gdy biorę się za pisanie, ale zrobię sobie przerwę na jakiś czas.

Tu Dan. Chciałbym podziękować mojej cudownej rodzinie, Jen, Isabelli i Julianie, za ich cierpliwość i wsparcie przy pisaniu tej, pierwszej dla mnie książki. Bez was nie byłoby to możliwe. Co do mojego współautora, Nate, niektórzy ludzie po prostu pojawiają się w naszym życiu i jesteś jedną z takich osób. Jestem wdzięczny, że możemy być współpracownikami, a teraz współautorami, ale przede wszystkim za to, że mogę cię nazywać przyjacielem.

Chciałbym również podziękować tym, którzy wywarli trwały wpływ na moją karierę, w tym Johnowi Kimowi, Philowi Rodopoulosowi, Jasonowi Delmore'owi, Davidowi Wintrichowi i Tashy Isenberg. W istocie jest ich zbyt wielu, by ich tu wszystkich wymienić, ale proszę, wiedzcie, że wszyscy zajmujecie szczególne miejsce w moim sercu.

Część 1

Podstawowe umiejętności

Rozdział 1

Od programisty do inżyniera

> Fundamentalne umiejętności, których nauka zawsze była żmudna, wydają się być przestarzałe. I mogłyby takie być, gdyby istniała jakaś krótsza droga do zostania ekspertem. Jednak droga do poziomu eksperckiego wymaga osadzenia na faktach.
>
> – Ethan Mollick, ze wstępu do *Co-Intelligence* (Portfolio, 2024)

Zostanie inżynierem oprogramowania wymaga szerokiej palety umiejętności rozciągających się na różnorodne obszary. Zrozumienie, o co naprawdę prosi nasz klient. Przekładanie tych potrzeb na kod dający się zarządzać i utrzymywać. Pisanie testów gwarantujących, że oprogramowanie robi to, co naszym zdaniem powinno robić. Budowanie interfejsów użytkownika, które się sprawdzają. Architektura. Praca z danymi. Przenoszenie kodu do środowiska produkcyjnego. Jeśli chcesz rozwijać swoją karierę jako praktyk oprogramowania, musisz skupić się na więcej, niż tylko pisaniu kodu. Musisz ogarnąć całe rzemiosło inżynierii oprogramowania.

Aby z programisty stać się inżynierem, musisz opanować podstawowe umiejętności z zakresu całego cyklu życia wytwarzania oprogramowania (SDLC – *software development lifecycle*), mądrze pracować, przyznać się samemu przed sobą, czego nie wiesz i znaleźć sposoby załatania tych luk. W tym rozdziale dostaniesz kilka wskazówek i porad, które pozwolą rozpocząć tę drogę. Poznasz różne drogi prowadzące do zostania inżynierem, a także kluczową wiedzę, którą te podejścia często pomijają. Ostatecznie ten rozdział dostarczy z trudem zdobytych rad, które pomogą ci w podróży, wygładzając przejście od programisty do inżyniera. Ale najpierw porozmawiajmy o tym, co to znaczy być inżynierem oprogramowania.

Inżynier pod (dowolną) inną nazwą

Świat oprogramowania pełen jest przeciążonych (często przedefiniowanych) pojęć, co pozostawia mnóstwo miejsca na niejednoznaczność. Gdy spytasz pięciu programistów o znaczenie jakiegoś słowa, zapewne otrzymasz co najmniej pięć odpowiedzi. Słowo „computer" oryginalnie oznaczało człowieka wykonującego obliczenia. Teraz słyszymy takie pojęcia, jak programista, koder, deweloper i inżynier oprogramowania. Czy są to synonimy?

Nie, nie są. *Programista* lub *koder* zwykle oznaczają kogoś skupionego na pojedynczym zadaniu generowania, no właśnie, kodu[1]. W ogólności mogą być wysoce wykształceni w zakresie konkretnego języka i/lub platformy, ale nie muszą rozumieć całego SDLC. Wielu profesjonalistów rozpoczyna swoje kariery od programisty lub kodera, ale ścieżka do awansu wymaga wyjścia poza samo implementowanie przypadkowych funkcjonalności czy poprawianie błędów.

Deweloperzy typowo mają lepsze pojęcie ogólnego obrazu dostarczania oprogramowania. Zwykle znają kilka języków i frameworków, a także mają doświadczenie w kilku dziedzinach biznesowych. Mając zwykle więcej doświadczenia niż programiści czy koderzy, zaczynają poznawać szerszy zakres SDLC i mogą służyć jako mentorzy lub partnerzy dla mniej doświadczonych członków zespołu.

Jednak wejście do królestwa inżynierii oprogramowania wymaga, abyś wniósł *dyscyplinę inżynieryjną* do całego SDLC. Od *inżyniera oprogramowania* oczekuje się nie tylko tego, że będzie sprawny i efektywny w pisaniu kodu, ale że będzie też myślał o skalowalności, niezawodności, wydajności i bezpieczeństwie. Musisz wyjść poza pobieżne rozumienie algorytmów i zasad projektowych. Nie tylko musisz znać zasady, ale także wiedzieć, kiedy należy je naginać, a nawet kiedy muszą zostać złamane. Inżynierowie oprogramowania zajmują się najbardziej złożonymi i krytycznymi systemami produkcyjnymi istniejącymi obecnie.

Nauczanie sformalizowane, od kursów szkoleniowych po uniwersytety typowo skupia się na mechanicznych aspektach pisania kodu, kształtując ludzi biegłych w programowaniu. Kształcą deweloperów lub programistów. To punkt startowy dla kariery w programowaniu. Aby móc się wyróżnić, awansować i pracować przy najciekawszych projektach, musisz wyjść poza samo pisanie kodu.

Tworzenie oprogramowania to więcej, niż tworzenie kodu poprawnego składniowo. Obszar wiedzy wymaganej dla osiągnięcia sukcesu jako inżynier oprogramowania daleko wykracza poza opanowanie jakiegoś języka czy nawet kilku. Trzeba również być biegłym we wszystkich aspektach cyklu życia produktu programistycznego. Ta książka ma być twoim przewodnikiem na tej drodze, pokazując ci rzeczy, o których możesz nawet nie wiedzieć, że ich nie wiesz.

Podstawy są ważne

Podstawy są ważne. Zawodowi sportowcy większość czasu spędzają pracując nad tym, czego nauczyli się na samym początku swojej przygody z tym sportem. Golfiści skupiają się na postawie, chwycie i wyrównaniu. Koszykarze ćwiczą dwutakty, podania i rzuty wolne. Nie poświęcają aż tyle czasu na te rzeczy, które potem ogląda się jako najważniejsze momenty meczu. Choć podstawy nie są ekscytujące, bez solidnego fundamentu nigdy nie osiągnęliby szczytów swojej profesji.

1 Nawiasem mówiąc, to są właśnie ci praktycy, którzy najprawdopodobniej zostaną zastąpieni przez tzw. sztuczną inteligencję (poprawnie: agentowe systemy kodujące – *agentic coding systems*).

Podobnie jak w każdej innej dziedzinie, w branży oprogramowania obowiązują pewne podstawowe zasady. Potrzeba czasu, aby je opanować. Większość swojej kariery spędzisz pracując z już istniejącym kodem. Nauka czytania kodu napisanego przez kogoś innego i szybkie rozumienie nowej bazy kodu jest kluczowe dla sukcesu. Nieznane bazy kodu mogą onieśmielać; musisz jednak nauczyć się czuć pewnie i komfortowo z niejednoznacznościami. Nawet po wielu miesiącach nie poznasz wszystkich niuansów tej bazy, ale też nie musisz rozumieć wszystkiego, zanim zaczniesz wprowadzać zmiany.

Zapewne znasz podstawy pisania kodu, ale niekoniecznie sposoby takiego pisania kodu, aby ułatwić życie osobie, która przyjdzie po tobie. Technika też nie stoi w miejscu; musisz być w stanie uczyć się nowych rzeczy. Wielu programistów ma obsesję na punkcie najnowszych technologii lub funkcji języka, jednocześnie ignorując ponadczasowe umiejętności interpersonalne, choć to one (zapewne) bardziej będą pomocne przy kolejnym awansie. Podstawy może nie generują takiego zainteresowania, jak najnowsze trendy, ale to one czynią różnicę pomiędzy stagnacją zawodową a możliwościami o szerszym zakresie i odpowiedzialności.

Wiele dróg, aby zostać inżynierem oprogramowania

Istnieje wiele ścieżek prowadzących do stania się inżynierem oprogramowania. Niektórzy wybierają przynajmniej początkowe studia informatyczne, podczas gdy inni uczestniczą w bootcampach, a jeszcze inni uczą się samodzielnie. Wielu z nich ma już dyplomy z pokrewnych dziedzin, takich jak elektronika lub elektrotechnika, po czym przechodzą do pisania kodu. Szczerze mówiąc, to, jak się nauczymy budować oprogramowanie, nie ma większego znaczenia.

Niektórzy mogą przekonywać, że wybrana przez nich ścieżka jest „lepsza", „szybsza" albo „tańsza". Każde podejście ma swoje zalety i wady. Samodzielna nauka nie wymaga płacenia czesnego i w zależności od talentu i predyspozycji może wymagać tygodni zamiast lat. W Internecie nie brakuje filmów i tutoriali, jednak przeszukiwanie ich może być żmudne i wymaga silnej motywacji wewnętrznej. Bycie samoukiem może również obejmować sporą dawkę prób i błędów, gdy trzeba przebrnąć przez materiały o (delikatnie mówiąc) różnej jakości.

Zasadniczo cele bootcampów i programów studiów się różnią. Programy licencjackich studiów informatycznych mają na celu przygotowanie do dalszych (magisterskich) studiów. Skupiają się na algorytmach, projektowaniu języków, teorii kompilatorów, systemach operacyjnych i innych powiązanych zagadnieniach. Jednak niekoniecznie przygotowują do codziennej pracy nad praktycznymi aplikacjami. Projekty są niewielkie i rzadko trwają dłużej niż kilka miesięcy, a ponadto niemal zawsze rozpoczynamy od zera, wolni od bagażu już istniejących aplikacji.

Bootcampy to miejsce wkuwania bardzo konkretnych informacji w krótkim czasie, często ograniczonym do tygodni. Zazwyczaj koncentrują się bardziej na określonych („modnych") językach, frameworkach itp. Można argumentować, że to podejście jest bardziej praktyczne, ale można też zauważyć, że zdobywana tak wiedza jest bardziej ulotna z natury, ponieważ popularność języków i frameworków ulega ciągłym wahaniom.

Bootcampy czasami odbywają się na uniwersytetach i może się okazać, że są prowadzone przez tych samych wykładowców, którzy uczą na regularnych studiach informatycznych. Może więc nawet nie być różnicy, jeśli chodzi o nauczycieli!

Każde z tych podejść wiąże się również z innym nakładem czasu. Bootcampy zwykle trwają około 600 godzin w ciągu kilku tygodni i masz niemal stały dostęp do kogoś, kto pomoże ci przetrwać trudny okres. Studia licencjackie mogą obejmować 1400 godzin zajęć rozłożonych na trzy lub cztery lata.

Niektórzy twierdzą, że dyplom licencjata daje pewną początkową przewagę, choć zmniejsza się ona z czasem. Oczywiście programy studiów informatycznych, jakie znamy dzisiaj, są stosunkowo nowe, ale jeszcze nie tak dawno większość programistów była z wykształcenia matematykami lub fizykami.



Wczesne programy studiów informatycznych często koncentrowały się na matematyce, a niektóre z nich były w zasadzie studiami matematycznymi z domieszką informatyki. W istocie wielu profesorów posiadało dyplomy z matematyki. Do dziś niektórzy uważają, że predyspozycje matematyczne są warunkiem koniecznym do odniesienia sukcesu jako inżynier oprogramowania. Tak nie jest. Kodowanie to komunikacja, a większość pracy polega na komunikowaniu się z innymi programistami, a nie z komputerem. Znajomość języków obcych jest o wiele ważniejsza od umiejętności matematycznych (*https://oreil.ly/qCWAx*). Nie jest zaskakujące, że pisanie kodu to kwestia języka, a nie liczb, o czym przeczytasz w rozdziale 2.

Ostatecznie sukces w branży oprogramowania polega na rozwiązywaniu problemów, kombinowaniu i kreatywności. Jeśli masz właściwe nastawienie, to je masz. Kropka. Nie ma znaczenia, jak rozwinąłeś swoje umiejętności i nie musisz się tłumaczyć z wybranej ścieżki.

Czego cię nauczono, a co potrzebujesz wiedzieć

Niezależnie od wybranej drogi nauki, zostałeś nauczony (albo nauczyłeś się sam), jak pisać kod. Opanowałeś język lub dwa, poznałeś `foo`, `bar` i inne generyczne nazwy zmiennych. Trochę poznałeś debugowanie, projektowanie wydajnych algorytmów i inne powiązane zagadnienia. Jednak wielka liczba ważnych rzeczy nie jest uwzględniana w typowym programie studiów ani wspominana przy nauce samodzielnej. Dlaczego? Zazwyczaj z powodu braku czasu, miejsca lub (fałszywego) założenia, że ta wiedza jest nam dana[2].

To, co wszystkie te kursy i materiały do nauki pozostawiają na boku, może mieć kluczowy wpływ na długoterminowy sukces. Te pomijane tematy obejmują to, jak pracować z innymi ludźmi, jak czytać kod, jak pisać *czytelny* kod i jak skutecznie pracować z kodem odziedziczonym. Te i wiele innych tematów stanowią treść pozostałej części tej książki.

2 Na przykład to, jak używać debugera czy znajomość nowoczesnych edytorów.

Większość deweloperów uczy się na projektach tworzonych od podstaw (tzw. projekty *greenfield*), które startują od pustej kartki i bez żadnego długu technicznego skrywającego się w istniejących bazach kodu. Biorąc pod uwagę czasowe ograniczenia narzucone przez długość semestru czy trwanie bootcampu, w większości sytuacji edukacyjnych zaczynamy od zera, aby ćwiczyć określoną umiejętność lub utrwalić jakąś koncepcję. Projekty są zwykle bardzo małe i odizolowane, a kod piszesz sam, bez udziału innych osób. Choć to się sprawdza w środowisku nauczania, nie jest to reprezentatywne dla realnego świata.

Jako praktykujący inżynier oprogramowania będziesz spędzać większość czasu na pracy z istniejącymi, przestarzałymi lub odziedziczonymi aplikacjami i całym mnóstwem decyzji, które zostały podjęte, zanim w ogóle się tu pojawiłeś[3]. I nawet gdy trafi się luksus pracy z nową bazą kodu, nawet ona będzie kumulować dług techniczny w nadchodzących miesiącach. Dzisiejsze projekty programistyczne składają się z setek tysięcy lub milionów wierszy kodu, tworzonego przez zespoły ludzi rozproszonych po całym globie. Będziesz stale pracować z innymi ludźmi, kolegami z zespołu lub klientami. Twój kod nie będzie jedynie spoczywać w repozytorium, czekając, aby ktoś go ocenił. Znajdzie się w środowisku produkcyjnym, gdzie prawdziwi ludzie będą oczekiwać, że będzie działać tak, jak zamierzono, dostarczając prawdziwą wartość biznesową. Łagodnie mówiąc, pierwsze zetknięcie z realiami tworzenia oprogramowania może być szokiem.

W szkole większość projektów trwa krótko – kilka miesięcy albo mniej. Jednak w rzeczywistym świecie projekty tak naprawdę nigdy się nie kończą, choć kiedyś mogą zostać porzucone, jak sztuka. Większość czasu życia kodu to faza utrzymania, co jest innym sposobem stwierdzenia, że oprogramowanie jest pełne produktów, które są ciągłą inwestycją, w przeciwieństwie do oddzielnych, dyskretnych projektów posiadających dobrze zdefiniowany początek i koniec. Tak długo, jak produkt programistyczny dostarcza wartość biznesową, inwestycje w niego będą trwały nadal.

Etos leniwego programisty

Etos leniwego programisty nie polega na spędzaniu całego dnia na oglądaniu filmów z kotkami w ulubionym portalu społecznościowym. Nie jest też metodologią uchylania się od pracy. *Etos leniwego programisty* to filozofia skoncentrowana na wydajności. Wielu początkujących inżynierów instynktownie rzuca się na poszukiwanie rozwiązań. Od razu zaczynają pisać kod, nie dając sobie czasu na zastanowienie się nad samą istotą problemu. Strategiczne lenistwo daje czas na przemyślenia, co zdecydowanie pomaga zwiększyć produktywność i tworzyć lepsze rozwiązanie.

> Poświęć tyle czasu, ile potrzeba, aby trwało to krócej.
>
> – przypisywane Patowi Parelli, twórcy szkoły jeździeckiej

3 Choć zawsze warto zadać sobie pytanie, dlaczego coś jest takie, jakie jest, nie należy się dziwić, gdy domysły kolegów będą równie dobre, jak twoje.

Unikaj podejścia siłowego. Możesz od tego rozpocząć, ale powinieneś powracać do koncepcji swojego projektu. Pomocne będzie zrozumienie takich rzeczy, jak notacja wielkiego O[4]. Dla przykładu, mogłeś zakodować algorytm sortowania bąbelkowego, ale dowiedziałeś się, że nie jest to najbardziej wydajny mechanizm sortujący. Jako inżynier powinieneś w projekcie rozważyć przypadek najlepszy, najgorszy i przeciętny i wybrać odpowiedni algorytm.

Są spore szanse, że nie jesteś w istocie pierwszą osobą, która próbuje rozwiązać konkretny problem. Poświęć trochę czasu na poszukanie istniejącego rozwiązania lub biblioteki. W wielu przypadkach 10 minut szukania może oszczędzić całych dni pracy. Jeśli czujesz, że powinien istnieć prostszy sposób, aby coś zrobić, najprawdopodobniej tak jest. Więcej na ten temat znajdziesz w podrozdziale „Nie wynajduj koła od nowa" na stronie 38.

Zadanie z kapitalizacji

Tu Nate. Na początku mojej kariery otrzymałem bardzo proste zadanie: kapitalizacji (zamiany wszystkich liter na wielkie) adresów w bazie danych, aby zapewnić spójność z działem pocztowym, w którym drukowano i wysyłano rachunki. Mój kierownik dokładnie objaśnił, co i gdzie trzeba zrobić, pokazał procedurę do zmodyfikowania, a także funkcję (napisaną przez kogoś, kogo już dawno tu nie było), którą powinienem wywołać w celu kapitalizacji adresu. Gdy badałem funkcję kapitalizacji, wydała mi się nadmiernie skomplikowana. Na zdrowy rozum powinna ona przyjmować jeden argument: ciąg znaków do kapitalizacji. Ale ta funkcja nie miała jednego argumentu. Było ich ponad tuzin. Potrzebowałbym wielu godzin, aby zrozumieć to coś.

Wydawało mi się, że tego rodzaju problem powinien rozwiązać sam język programowania. Przeprowadziłem małe poszukiwania i rzeczywiście, język miał wbudowaną funkcję kapitalizacji, której użyłem. Przeskakując do spotkania zespołu w następnym tygodniu: gdy krążyliśmy po pokoju omawiając to, nad czym pracujemy, wspomniałem o zadaniu z kapitalizacji. Jeden z kolegów spojrzał w moją stronę i spytał „O, użyłeś kapitalizacji Joe'ego?" Odpowiedziałem, że nie.

Teraz obróciły się na mnie wszystkie głowy. To był pamiętny moment. Pomimo że byłem najmniej doświadczonym programistą w tym pokoju, nikt inny nie wiedział, że język ma wbudowaną funkcję kapitalizacji. Być może została dodana po tym, jak poprzedni deweloper napisał swoją funkcję; nie wiem. Chodzi o to, że jeśli sądzisz, że powinien istnieć łatwiejszy sposób, poświęć trochę czasu, aby go poszukać. Nawet kilka godzin badania opcji i alternatywnych rozwiązań może oszczędzić ci dni, a może nawet tygodnie pracy.

4 W telegraficznym skrócie, notacja wielkiego O opisuje wydajność jednych algorytmów względem innych.

Wartość świeżego spojrzenia

> Najbardziej niebezpieczna fraza to „Zawsze tak robiliśmy”.
>
> – Grace Hopper, pionierka komputerów i admirał marynarki wojennej

Nawet jeśli nie masz tylu lat doświadczenia i wierszy napisanego kodu, co twoi starsi koledzy, pamiętaj, że jako początkujący masz jedną cenną zaletę: świeżą parę oczu. Nowi ludzie – czy to w bazie kodu, w zespole czy w ogóle w branży – często postrzegają wyzwania z obiektywnej perspektywy, nieobciążeni historycznymi decyzjami, które dawno straciły użyteczność. To świeże spojrzenie może prowadzić do innowacyjnych rozwiązań, identyfikować brak efektywności albo podważać utarte założenia. Nowicjusze mogą też wnieść entuzjazm do podupadającego projektu, odnawiając energię i zaangażowanie zespołu.

Zawsze tak robiliśmy

Tu Nate. Lata temu pomagałem klientowi w problemie wydajnościowym dotyczącym tego, jak aplikacja przetwarzała widgety w nocnym zadaniu wsadowym. Gdy firma zaczynała działalność, miała tylko kilku klientów z kilkoma pudełkami widgetów, co można było łatwo ukończyć przez noc. Wraz z rozwojem firmy przybywało klientów, a ci klienci mieli więcej widgetów. W chwili, gdy z nimi pracowałem, ich nocny wsad nie był już w stanie przetworzyć wszystkich na czas.

Usiadłem z nimi i zaczęliśmy przeglądać architekturę. Po kilku godzinach zadałem proste pytanie: „Czy przetwarzanie widgeta A ma cokolwiek wspólnego z przetwarzeniem widgetu B?” Nie, wprawdzie na końcu wykonywane były pewne agregacje, ale poszczególne widgety były niezależne. Zgłosiłem prostą propozycję: czy nie można po prostu przetwarzać tych widgetów równolegle? Wystarczy wdrożyć zestaw procesów roboczych, które będą pobierać kolejny widget, przetwarzać go, a wynik umieszczać w kolejce do późniejszej agreagacji.

Zapadła brzemienna cisza. Tak. Tak, to będzie działać. Choć z pewnością chciałbym, aby mój szef myślał, że miałem jakąś specjalną intuicję, tak naprawdę jej nie miałem. Po prostu nie wiązało mnie to, co jest najbardziej niebezpieczną frazą, jaką można usłyszeć w każdej organizacji: „Zawsze tak robiliśmy”. Samo to, że zawsze tak robiliśmy, nie musi oznaczać, że jest o właściwy sposób i na pewno nie oznacza, że jest to sposób najlepszy. Podejście, które ktoś wybrał lata temu, mogło być przypadkowe albo po prostu najbardziej oczywiste. Istnieją spore szanse, że istnieje lepszy sposób. Nigdy nie obawiaj się stawiania pytań ani szukania innego podejścia.

Nigdy nie lekceważ wartości innego punktu widzenia i nie obawiaj się pytać, dlaczego coś jest takie, jakie jest. Jeśli natrafisz na coś, co wydaje się dziwaczne, nie na miejscu albo po prostu nie ma sensu, poproś kolegę, aby ci to wyjaśnił. W niektórych przypadkach

niekonwencjonalne podejście będzie miało uzasadnione powody, ale znacznie częściej ludzie po prostu kopiują to, co zastali. Dopóki twoje pytania będą wynikać z chęci zdobycia wiedzy, a nie z próby zawstydzenia byłych (i obecnych) programistów, większość inżynierów oprogramowania pozytywnie zaangażuje się w dyskusję.

Nie rozwiązuj zbyt szybko

Kiedy natrafisz na problem, powstrzymaj się przed pokusą zbyt szybkiego przejścia do jego rozwiązywania. Zamiast tego poświęć trochę czasu na rozpoznanie i zrozumienie źródłowej przyczyny tego problemu. Daj sobie trochę miejsca na namysł. Nie mamy dość okazji, aby po prostu pomyśleć bez zakłóceń. Założę się, że twój kalendarz jest pełen spotkań, a korporacyjny system wiadomości przerywa ci pracę wiele razy w ciągu dnia. Pomiędzy codziennymi spotkaniami na stojąco z właścicielem produktu, menedżerem, liderem zespołu i architektem zapewne aktualizujesz swój status co parę godzin. Lista rzeczy do zrobienia nie chce się skurczyć, a testerzy znaleźli kolejną masę trudnych bugów. Ale pomimo tej całej presji czasowej kluczowe jest, aby poświęcić trochę czasu na przemyślenie rozwiązań, a nie zabierać się od razu za szybką poprawkę.

Choć takie poprawki rozwiązują problemy, przed którymi właśnie stoisz, na dłuższą metę powodują więcej kłopotów. Prowizoryczne rozwiązania („łatanie dziur") często stwarzają większe wyzwania. Aby sobie z tym poradzić, pojawił się ruch na rzecz testowania pod kątem oczekiwanych problemów we wcześniejszej fazie cyklu wytwarzania – innymi słowy, „przesunięcie w lewo", wcześniej na linii czasu projektu. Błędy odkryte w produkcji są zdecydowanie najbardziej kosztowne w naprawie. Klienci mogli ponieść straty, co może kosztować firmę dodatkowe pieniądze. Dane mogą być uszkodzone, co może być trudne do uzgodnienia. Naprawa podstawowego problemu często rozciąga się na wiele procedur, które wymagają skomplikowanej refaktoryzacji i – zapewne – koordynacji z innymi zespołami.

Przesunięcie w lewo nie dotyczy jedynie rozwiązywania problemów; w istocie chodzi tu o przeniesienie wszystkiego wartościowego do wcześniejszej fazy procesu. Programowanie w parach przekształca przeglądy kodu z czegoś, co następuje dni (albo i tygodnie) po ukończeniu pracy, w aktywność w czasie rzeczywistym. Uczynienie testów ciągłą praktyką upraszcza rozwiązywanie problemów, gdyż są identyfikowane bliżej miejsca, gdzie powstały. Ciągła integracja łagodzi kłopoty ze sprzecznymi zmianami kodu dzięki wczesnemu i częstemu scalaniu, a w połączeniu z ciągłym dostarczaniem gwarantuje, że wszelkie zmienności w środowiskach zostaną wykryte, zanim zaszkodzą klientowi. Wbudowanie zabezpieczeń w cały proces wytwarzania zmniejsza ryzyko katastroficznych podatności.

Znacznie tańsze jest odkrywanie takich problemów poprzez testy. Gdy błędy zostaną dostrzeżone bliżej chwili napisania kodu, członkowie zespołu nadal mają tę pracę w pamięci i nie muszą tracić czasu na rozszyfrowywanie tego, co robi podejrzany fragment. Testowe dane można odtworzyć bez zbędnych ceregieli. Nawet jeśli problem może być trudny do rozwiązania, mniejsza jest presja czasowa na jego szybkie naprawienie, gdyż nie

wpływa na żadnych klientów. Tak narodziły się testy gwarancji jakości (*quality assurance* – QA) i inżynierowie QA. Okazuje się, że jeszcze tańsze jest odkrywanie problemów w trakcie tworzenia kodu, co częściowo doprowadziło do koncepcji wytwarzania sterowanego testami (test-driven development – TDD), którą zajmiemy się w rozdziale 5.

Ale co by było, gdybyśmy w ogóle nie popełnili błędu? Gdyby został on rozwiązany w samym projekcie?

> Większość dużych problemów, z jakimi mamy do czynienia w oprogramowaniu, wynika z niezrozumienia. Nie mamy dobrego pojęcia o tym, co robimy, zanim to zrobimy. A potem tylko działamy, działamy, działamy i już się stało.
>
> – Rich Hickey, na Clojure Conj 2010

Nieporozumienia są powszechne w świecie oprogramowania i stanowią częste wyzwanie podczas tworzenia programów. Takie terminy jak „klient" i „odbiorca" mogą wydawać się wymienne, dopóki właściciel produktu nie stwierdzi, że pewne podejście sprawdzi się dla tego pierwszego, ale nie dla drugiego. Są to podobne, ale różne pojęcia i takie niuanse mają znaczenie. Żadna liczba testów ani najlepszy system typowania nie zdoła wykryć problemu z nieporozumieniem. Same techniczne rozwiązania nie wyeliminują tych problemów; wynikają one z fundamentalnych różnic w rozumieniu[5].

Zamiast szukać obejścia problemu powstałego wskutek nieporozumienia, spróbujmy go w ogóle uniknąć. Jak? Skupiając się na rozwiązywaniu problemów. Czy wiesz, jaki problem ma rozwiązać twoja aplikacja? Czy rozumiesz kontekst i ograniczenia? Jakie są powiązane problemy? Poświęć trochę czasu na przemyślenie przestrzeni problemu.

Deweloperzy często mają skłonność do nadmiernej komplikacji. Choć nie jest tak za każdym razem, zwykle jest wiele sposobów rozwiązania określonego problemu. Nigdy nie trać z oczu problemu, który próbujesz rozwiązań i spiesz się z wybieraniem rozwiązań. Czego oczekują klienci, jakie są ich rzeczywiste potrzeby? Lepiej nie zapędzać się samemu do narożnika, wyobrażając sobie, czego my moglibyśmy potrzebować. To zapewne nie jest to samo. Ale nie chcesz też tworzyć mnóstwa kodu, który później będziesz musiał wyrzucić albo, co jeszcze gorsze, utrzymywać, choć nie będzie używany.

Osiągnięcie tego może nie być łatwe, ale kieruj się w stronę „dlaczego", czyli pierwotnej przyczyny żądań klienta (patrz ramka „Pięć 'dlaczego'"). Zadawaj pytania. Rozmawiaj z klientami; rozmawiaj z właścicielem produktu. W wielu przypadkach może się okazać, że klient prosi o jedno, choć potrzebuje czegoś innego, ale tego nie widzi. Utknął na „Zawsze tak robiliśmy". Czasami klient przychodzi z prośbą o bardzo konkretne rozwiązanie – na przykład „Potrzebujemy możliwości wyeksportowania tej tabeli do arkusza kalkulacyjnego". Można poczuć pokusę, aby natychmiast zbudować taką funkcję, ale doświadczony inżynier najpierw spyta, jaka jest prawdziwa potrzeba (do czego potrzebny jest ten arkusz). Często istnieje rozbieżność pomiędzy tym, o co klient prosi, a tym, co rozwiąże jego biznesowy problem.

5 Słownik używanych pojęć może być jednym z najcenniejszych i najbardziej wpływowych artefaktów projektu. Warto go utworzyć, jeśli twój projekt jeszcze go nie ma.

Pięć „dlaczego”

„Pięć dlaczego” to stworzona jako część Toyota Production System metoda rozwiązywania problemów, koncentrująca się na szukaniu pierwotnej przyczyny problemu poprzez zadawanie pytania „Dlaczego” pięć razy, przy czym każde kolejne „dlaczego” ma uściślić odpowiedź na poprzednie pytanie. W ten sposób możemy przebić się przez powierzchowne symptomy problemu, aby odkryć jego podstawową istotę. Oto przykład:

- Problem: Robot przestał pracować.
  - Dlaczego? Obwód został przeciążony, co przepaliło bezpiecznik.
  - Dlaczego? Łożyska nie były dostatecznie smarowane i się zablokowały.
  - Dlaczego? Pompa olejowa nie podawała dość oleju.
  - Dlaczego? Wlot pompy jest zatkany opiłkami metalu.
  - Dlaczego? Pompa nie ma filtru.

Przejdźmy teraz do przykładu z programowania. Lata temu natknąłem się na stronę webową, w której kodzie wszystkie znaczniki zamykające znajdowały się w kolejnym wierszu. Interpreter się tym nie przejmował, ale dla mnie widok zamykających znaczników rozpoczynających każdy wiersz był co najmniej niepokojący. Spytałem kolegę, dlaczego ktoś w ogóle użył takiego podejścia. Odpowiedział, że z powodu buga w starej wersji serwera web, na którym działała ta strona. OK, ale dlaczego nadal obsługujemy tę starą wersję? O nie, już nie obsługujemy. OK, dlaczego ten bug nie został naprawiony? Ależ został. OK, więc dlaczego nadal widzę ten wzorzec? No cóż, to zapewne jakaś stara strona. Powiedziałem mu, że strona powstała kilka dni wcześniej. W tym momencie zgodziliśmy się, że powinniśmy poinformować deweloperów, aby przestali stosować ten wzorzec i dodaliśmy zadanie czyszczenia, aby przerobić te strony, które nadal używały przestarzałego i od dawna niepotrzebnego podejścia.

W wielu przypadkach pierwotnym źródłem jakiegoś wymagania może być „superużytkownik” starego systemu. Choć wiedza takich osób może być nieoceniona, są oni również uwarunkowani tym, co dotychczas mieli[6]. W niektórych przypadkach nie rozumieją podstawowej przyczyny (*dlaczego*) określonej funkcjonalności i zakładają, że jest to jedyny sposób wykonania jakiejś pracy (rozwiązania problemu). Innymi słowy, świadomie lub nie nakłaniają do skopiowania starego systemu w nowej technologii.

Kluczowe jest posiadanie szerszej perspektywy. Wybierz odpowiednie narzędzie dla zadania rozwiązania problemu, który klient naprawdę ma. Wielu inżynierów jest

6 Co wyjaśnia, dlaczego tak wiele aplikacji wydaje się preferować emaile i arkusze kalkulacyjne.

nadmiernymi optymistami. Bądź realistą; bądź bezwzględnie pragmatyczny. Jednak skalowalność ma znaczenie; nie każda aplikacja może obsłużyć jedną trzecią całego ruchu internetowego.

> Przewidywanie jest bardzo trudne, szczególnie gdy chodzi o przyszłość.
>
> – Duńskie przysłowie

Zastosuj złotą regułę do oprogramowania

Zapewne kiedyś we wczesnej fazie swojej ścieżki edukacyjnej poznałeś złotą regułę etyczną (*https://oreil.ly/L-bQw*), która głosi „traktuj innych tak, jak ty byś chciał być traktowany”[7]. Zasada ta rozciąga się na wszystkie kontynenty i kultury, od tysięcy lat stanowiąc podstawę funkcjonowania cywilizowanych społeczności. Ale jak się okazuje, nie jest to tylko zasada etyczna. Możesz (i powinieneś!) stosować taki sam standard do swojego kodu.

Pomyśl o tym deweloperze, który kiedyś będzie musiał podążać twoim śladem. Dodatkowo może się okazać, że tym kimś będziesz ty sam![8] Co *twoje* przyszłe ja chciałoby zobaczyć w kodzie źródłowym? Co ułatwi życie następnemu deweloperowi? Kod w swojej esencji jest mechanizmem komunikacji. Tak, jest to sposób instruowania komputera, aby wykonał pewne zadanie, ale to znaczenie blednie w porównaniu do zdolności komunikowania się z innymi deweloperami. Pisz kod, który ma być czytany. Optymalizuj dla człowieka, nie dla kompilatora[9].

Jeśli chcesz wyróżnić się z tłumu, stosuj się do złotej reguły. Pisz czysty kod, aby ci, którzy przyjdą później, mogli go zrozumieć i utrzymywać. Aktualizuj dokumentację, aby oszczędzić innym marnowania wysiłków na stosowanie przestarzałego podejścia. Twórz przejrzyste diagramy, które wyjaśniają, zamiast zaciemniać. Znajdź czas na odpowiadanie na pytania i pomaganie innym. Zbierz to wszystko razem, a staniesz się osobą pożądaną; ludzie będą chcieli cię w swoich zespołach, bo dzięki tobie wszyscy staną się lepsi.

Podsumowanie

Kariera inżyniera oprogramowania to dużo więcej, niż gapienie się w ekran i klepanie kodu. Bycie czymś więcej niż tylko koderem czy programistą wymaga dużo więcej, niż tylko tworzenia poprawnego składniowo kodu; wymaga wszechstronnego zestawu umiejętności. Nie ma znaczenia, gdzie ani jak nauczysz się zawodu – na studiach, bootcamppach czy z materiałów znalezionych online. Liczy się to, jak połączysz ze sobą te umiejętności.

To, czego cię nauczono, to zdecydowanie nie wszystko, co trzeba umieć, aby zapewnić sobie sukces w karierze. W tym rozdziale przedstawiliśmy porady, jak unikać najczęstszych

7 Albo w wersji negatywnej: „Nie czyń drugiemu, co tobie niemiłe”.

8 Każdemu deweloperowi zdarzyło się wpatrywać się w jakiś kod i zastanawiać, co za dureń go napisał, aby powoli sobie uświadomić, że to właśnie on jest tym durniem.

9 Z wyjątkiem pewnych, bardzo specjalistycznych przypadków. Rozpoznasz to, gdy się z tym zetkniesz.

pułapek, takich jak zbyt szybkie rozwiązywanie problemów, przekombinowanie i zagrożenia siłowego podejścia. Im wcześniej nauczysz się tych zasad, tym lepiej.

W praktyce

Znalezienie mentorów może być niezwykle cenne na początku kariery. Rozejrzyj się w swojej organizacji i grzecznie spytaj któregoś ze starszych inżynierów, czy pozwoli się zaprosić na pogawędkę przy kawie – to twój prezent. Porozmawiaj z nim o jego doświadczeniach, czego się nauczyć, co wolałby poznać wcześniej, co uczyniło go bardziej wartościowym dla zespołu, czego jeszcze powinien się nauczyć i tak dalej. Możesz także kontaktować się z prelegentami wydarzeń i autorami. Wielu z nich chętnie pomaga tym, którzy się do nich zwracają.

Następnym razem, gdy dostaniesz zadanie naprawienia błędu lub stworzenia nowej funkcji, zatrzymaj się i nie przechodź od razu do kodowania. Upewnij się, że rozumiesz problem, który próbujesz rozwiązać. Poświęć 30 minut na analizę. Czy istnieje gotowe rozwiązanie, z którego możesz skorzystać? Spróbuj naszkicować lub wymodelować koncepcje dziedziny problemu wraz z możliwymi rozwiązaniami. Rozważ omówienie swoich opcji z doświadczonym współpracownikiem.

Przejrzyj spis treści tej książki. W jakich tematach czujesz się najmocniejszy? Co mógłbyś poprawić? Wybierz obszar tematów, o których chcesz nauczyć się więcej, i zaplanuj ich poznawanie. Działaj zgodnie ze wskazówkami z punktu „W praktyce" i zapoznaj się z dodatkowymi materiałami w tych rozdziałach.

Twój framework, biblioteka lub dostawca chmurowy mogą usuwać (abstrahować) pewne rzeczy z codziennej pracy. Na przykład korzystanie ze Spring Data oszczędza pisania niezliczonych instrukcji SQL. Choć zdecydowanie trzeba wykorzystywać produktywność tych abstrakcji, trzeba też rozumieć warstwę abstrakcji *poniżej tej*, której używamy. Poświęć czas, aby zajrzeć pod maskę i się rozejrzeć; zdobyta tu wiedza będzie bardzo pomocna, szczególnie gdy coś nie chce działać zgodnie z planem.

Dodatkowe materiały

- *The Pragmatic Programmer: Your Journey to Mastery*, Andrew Hunt i David Thomas (Addison-Wesley Professional, 2019)[10]
- *The Mythical Man-Month: Essays on Software Engineering*, Fred Brooks (Addison-Wesley, 1995)[11]
- *Design Patterns: Elements of Reusable Object-Oriented Software*, Erich Gamma et al. (Addison-Wesley, 1994)[12]

10 Wyd. polskie *Pragmatyczny programista. Od czeladnika do mistrza* (Helion 2021).

11 Wyd. polskie *Legendarny osobomiesiąc. Opowieści o inżynierii oprogramowania* (Helion 2019).

12 Wyd. polskie *Wzorce projektowe. Elementy oprogramowania obiektowego wielokrotnego użytku* (Helion 2010).

- *Practices of an Agile Developer*, Venkat Subramaniam i Andy Hunt (The Pragmatic Bookshelf, 2006)
- *The Productive Programmer*, Neal Ford (O'Reilly Media, 2008)
- *Software Engineering at Google*, Hyrum Wright et al. (O'Reilly Media, 2020)[13]
- *The Staff Engineer's Path*, Tanya Reilly (O'Reilly Media, 2022)
- „No Silver Bullet: Essence and Accidents of Software Engineering", Frederick P. Brooks Jr. (The University of North Carolina at Chapel Hill Department of Computer Science, September 1986)
- *Code Complete*, 2 wydanie, Steve McConnell (Cisco Press, 2004)[14]

13 Wyd. polskie *Inżynieria oprogramowania według Google* (Helion 2023).
14 Wyd. polskie *Kod doskonały*, (Helion 2010).