Opinie o książce *Test-Driven Development: podstawy*

Nowoczesne i praktyczne podejście Roba Mayersa do TDD pozwala szybko zabrać się do pracy, zapewniając jednocześnie bezpieczeństwo dzięki jasnym regułom, przemyślanym sieciom zabezpieczeń oraz wiedzy wyniesionej z dziesiątek lat praktycznych doświadczeń. Wciągające, zabawne i przydatne dla każdego dewelopera.

– Jeffrey J. Langr, weteran tworzenia oprogramowania i autor książek

Po wielu dekadach stosowania TDD Rob daje nam to, czego potrzebowaliśmy – książkę mówiącą o kodzie testowym jak pierwszoplanowym elemencie. Jego spostrzeżenia dotyczące „zapachu testów" i programowania wspomaganego sztuczną inteligencją daje nam spojrzenie w przyszłość, w której wyraźne specyfikacje testów zastępują debaty o architekturze. Przykłady w wielu językach pokazują uniwersalne zasady TDD. Ta książka to niezbędne kompendium wiedzy.

– Shane Duan, Dyrektor ds. inżynierskich w Rockfish Data, Inc.

Książka ta to niezbędnik dla każdego, komu zależy na pisaniu wysokiej jakości sprawdzonego kodu. Widziałem podejście Roba w praktyce i byłem świadkiem niezwykłego wpływu, jaki może mieć skuteczne TDD. Jego pasja, jasność prezentacji i wskazówki praktyczne sprawiają, że ta książka jest bezcennym źródłem wiedzy dla każdego poważnego specjalisty w zakresie oprogramowania.

– Jorgen Hesselberg, autor książki of Unlocking Agility;
współzałożyciel Comparative Agility

Czy chcecie czuć się pewnie podczas rozwoju zawodowego? Czy chcecie zyskać zaufanie kolegów i udowodnić im, że jesteście godni zaufania? Osoba, która zajmuje się tym przez ćwierć wieku, pokaże wam, jak to osiągnąć.

– Kent Beck

Z pewnością jest to najlepsza książka na temat TDD, z jaką miałem do czynienia w ostatnim dziesięcioleciu. Myers unika pustych sloganów traktujących TDD jak uzupełnienie tradycyjnego podejścia, a jego uwagi na marginesie projektu wprowadzają czytelnika w samo sedno zagadnienia.

– GeePaw Hill, niezależny szkoleniowiec techniczny
w zakresie tworzenia oprogramowania

Rob przywraca TDD jego prawdziwy sens: rzemiosło łączące myślenie, jakość i radość. Ta książka inspiruje zarówno deweloperów, jak i zespoły, do dążenia do trwałej doskonałości technicznej jako podstawy sensownej, zwinnej współpracy twórczej. Gorąco polecam.

– Björn Jensen, CST, CTC, CEC & AKT

TDD było dla moich zespołów trudne do opanowania. Ale opanowanie go dawało nam magiczną moc. Bardzo podoba mi się to, jak ta książka ułatwia naukę. Można z łatwością pracować z nią, mając Roba jako swojego wirtualnego partnera.

– Lisa Crispin, autor i konsultant

Test-Driven Development: Podstawy

Rob Myers

APN Promise
Warszawa 2026

Test-Driven Development: Podstawy

Authorized translation from the English language edition, entitled „Essential Test-Driven Development" by Rob Myers, published by Pearson Education, Inc., publishing as Addison-Wesley Professional, ISBN: 978-0-13-449415-9

Polish language edition published by APN PROMISE SA

Autoryzowany przekład z wydania w języku angielskim, zatytułowanego „Essential Test-Driven Development" by Rob Myers, opublikowanego przez Pearson Education, Inc., publikującego jako Addison-Wesley Professional, ISBN: 978-0-13-449415-9

APN PROMISE SA, ul. Domaniewska 44a, 02-672 Warszawa
tel. +48 22 35 51 600
e-mail: wydawnictwo@promisegroup.com

ISBN: 978-83-7541-544-5 (druk), 978-83-7541-545-2 (ebook)

Fotografia na okładce: Tomasz Niesłuchowski

Przekład: Witold Sikorski
Redakcja: Marek Włodarz
Korekta: Ewa Swędrowska
Skład i łamanie: MAWart Marek Włodarz

Mojemu zmarłemu ojcu, Bobowi Myersowi, który dzielił ze mną zainteresowanie komputerami i fantastyką naukową. To on wprowadził mnie w tajniki pierwszego terminala telefonicznego, pozwalał grać przez wiele godzin w weekendy w gry tekstowe, takie jak „Adventure" i „Super Star Trek", a także pokazał mi, jak edytować kod i uruchamiać własne kopie tych gier.

A także mojej matce, Alice Myers, która cierpliwie znosiła dwóch introwertyków, gdy spędzali całe weekendy przy blacie kuchennym z ciągnącym się przez całe wejście do kuchni kablem telefonicznym.

Dedykuję im tę książkę z miłością, podziwem i wdzięcznością za ich wsparcie i zachęty oraz za to, że skierowali mnie na tę fascynującą ścieżkę kariery.

Spis treści

Część II: Praktyki pomocnicze

Przedmowa

Rozczarowanie

Rob Myers zaczyna książkę opisem swojego stanu psychicznego w 1997 roku, gdy zastanawiał się nad zmianą ścieżki kariery. Rozważał porzucenie programowania i ubieganie się o licencję na handel nieruchomościami. Jego kariera przestała mu się podobać, choć wcześniej myślał, że będzie się zajmował oprogramowaniem przez całe życie. Znam to uczucie. Jestem pewien, że wiele osób w branży programowania ma takie chwile. Rozglądamy się wtedy i dochodzimy do wniosku, że musi być jakaś lepsza droga.

Podobnie jak Roba, programowanie komputerów zafascynowało mnie, gdy byłem dzieckiem. Tę pasję rozwijałem przez całą szkołę średnią i studia, i odkryłem, że programowanie nie tylko sprawia mi przyjemność, ale jestem w tym też dobry. Później okazało się, że mam również talent do kierowania zespołami, które wspólnie piszą kod i tworzą bardziej złożone systemy. I wtedy pojawiły się prawdziwe problemy.

Zauważyłem pewne ważne wzorce, które później dały się we znaki zarówno mnie, jak i prowadzonym przeze mnie zespołom.

Pierwszy wzorzec pokazywał, że większość kodu mogła być utrzymywana tylko przez ich autora. Stało się to jednocześnie gwarancją zatrudnienia i więzieniem, z którego nie było ucieczki.

Drugi wzorzec mówił, że programiści są na ogół fatalnymi menedżerami projektów. Jeszcze gorzej im idzie organizowanie procesów, które mają na celu utrzymanie spójnej współpracy zespołów programistów przez dłuższy czas. Największym wyzwaniem przy zarządzaniu projektami była niemożność oszacowania, jak długi czas zajmie praca, do tego stopnia, że menedżerowie płacący za pracę zespołu musieli uruchamiać syrenę alarmową i wezwać do całodobowej pracy, aby zdążyć na czas. To prowadziło do największego koszmaru, z jakim mierzą się zespoły programistów: zespół ds. zapewniania jakości znajdował na końcowym etapie krytyczny błąd, który musiał być usunięty przez wypuszczeniem programu. Jednak często nie można go było zlokalizować nawet przez wiele miesięcy.

Mój zespół w firmie Interface Systems w Ann Arbor w stanie Michigan zetknął się z takim koszmarnym scenariuszem w połowie lat dziewięćdziesiątych XX wieku. Trwało to kilka tygodni, podczas których programiści gorączkowo próbowali znaleźć ten nieuchwytny błąd, którego w żaden sposób nie dało się obejść.

Jako kierownik zespołu byłem wciąż wzywany przez szefów, którzy byli sfrustrowani i domagali się rozwiązania. Wreszcie, gdy pewnego poranka przyszedłem do pracy, jeden z głównych deweloperów powiedział mi, że znalazł i rozwiązał ten problem. Eureka! Zapytałem go, jak to zrobił. Powiedział, że przeglądał katalogi plików zawierających kod i natknął się na element testujący, który napisał wiele miesięcy wcześniej. Odkurzył go i uruchomił. Powiedział, że od razu ujawniło to problem i doprowadziło go prosto do jego źródła.

Byłem pod wrażaniem, że mam ten koszmar za sobą, jednak wiedziałem, że jest to tylko stan tymczasowy, gdyż niewątpliwie kolejny koszmarny błąd czaił się ciemnych zakamarkach kodu czekając, aby uderzyć w najmniej korzystnym momencie.

Zacząłem więc planować ucieczkę. Moim pomysłem było zorganizowanie obozu kajakowego na granicznych wodach Minnesoty.

Pomyślałem: Musi być lepszy sposób.

To nic nowego

Rob pisze, że nawet ponad 25 lat po tym, jak Kent Besk i inni zaczęli pod koniec lat 90. XX wieku wprowadzać TDD, wykorzystując testowanie jednostkowe i automatyczne schematy testów jednostkowych, większość nadal tego nie praktykuje. Nawet ci, którzy to robią, wciąż potrzebują lepszego szkolenia i dyscypliny, aby włączyć je do standardowego procesu kodowania.

Doświadczenie mówi mi, że już znacznie dłużej wiemy, co należy robić. W latach 1978–1980 studiowałem na University of Michigan, uzyskując kolejne dyplomy z dziedziny informatyki i inżynierii. W lecie 1980 roku zadzwonił do mnie profesor John Saulor i poprosił, abym dołączył do grupy informatyków podczas letniego stażu w wysoko notowanej firmie noszącej nazwę Manufacturing Data Systems, Inc. (MDSI). Skorzystałem z tej okazji.

W ciągu czterech pierwszych tygodni stażu cała grupa chętnych studentów przeszła intensywne i ciekawe szkolenie, aby nauczyć się podejścia do pisania kodu w języku Pascal, stosowanego przez MDSI. Jednym z kluczowych elementów, którego uczył nas dr Saylor, było pisanie testów przed napisaniem kodu. Katalogowaliśmy testy, a potem pisaliśmy kod i uruchamialiśmy testy, aby stwierdzić, czy nasze kodowanie jest poprawne. Poświęcaliśmy tyle samo uwagi na nasz kod testowy, co na nasz kod produkcyjny. Naprawdę uważam, że otrzymałem klucz do nauki naprawdę nowoczesnego sposobu kodowania. Nie mogę uwierzyć, jakie miałem szczęście, mając możliwość spędzenia lata na doskonalenie tych umiejętności.

Po zakończeniu tej czterotygodniowej sesji zostaliśmy jako stażyści przypisani na lato do różnych wydziałów i szefów. Ja zostałem przydzielony do Vince'a B., aby pracować nad projektami CODE i CAPP. Gdy tylko usiadłem w moim kubiku, Vince wezwał mnie do biura, zamknął drzwi i powiedział, że mam nie używać żadnej z technik, których uczyłem się przez ostatnie cztery tygodnie. Miał taki styl przekazywania informacji, że oczywiście nie miałem zamiaru przekonywać go, że powinien zmienić zdanie. Jego podstawowy przekaz brzmiał: „Nie mamy czasu na bzdury. Mamy ustalone terminy".

Doświadczenia z tych czterech tygodni nigdy mnie nie opuściły – podobnie jak frustracja wynikająca z tego, że odmówiono mi możliwości wykorzystania tej wiedzy

Później, podczas mojej pracy w MDSI, miałem jedną okazję na zastosowanie nowoczesnych technik wraz z innym programistą. Dało to najlepszy kod, jaki kiedykolwiek napisałem w tym czasie. Była to jedyna szansa na zastosowanie podejścia dr. Saylora.

Fabryka Javy w Interface Systems

W 1997 roku awansowałem w Interface Systems na stanowisko wiceprezesa ds. badań i rozwoju (ang, research and development – R&D). Uważałem, że w tym miejscu mogę zacząć wprowadzać lepszy sposób, który zawsze uważałem za możliwy. Jednak przez pierwsze dwa lata sytuacja poprawiała się tylko w niektórych miejscach i to dzięki jednemu standardowemu podejściu: cięższej pracy. W 1999 roku miały miejsce ważne zbiegi okoliczności. Rozpocząłem współpracę z dwoma konsultantami, Jamesem Goebelem i Tomem Meloche, którzy pracowali dla firmy konsultingowej działającej u Chryslera. Mieli oni to szczęście, że natknęli się na innego konsultanta, który wnosił nowe radykalne pomysły do systemu kompleksowych odszkodowań Chryslera po tym, gdy system zaliczył katastrofalną historię porażek wdrożeniowych. Potrzebowali świeżego, nowego podejścia. Kent Beck potrafił przekonać zespół zarządzający, aby spróbowali czegoś nowego. W wyniku tego Tom i James zaczęli uczyć mnie o Extreme Programming, które obejmowało pojęcia pisania testów przed napisaniem kodu oraz automatyzację tych testów tak, aby mogły być wciąż uruchamiane w miarę tworzenia oprogramowania. Przypomniało mi to czas spędzony z dr. Saylorem i moje własne doświadczenia. Nie wracałem do poprzednich metod. Reszta jest dla mnie historią, którą dobrze opisałem w swojej książce: *Joy, Inc.: How We Built a Workplace People Love*.

Z czasem przekonałem się do wszystkich innych pomysłów Extreme Programming, zwłaszcza do programowania. Stało się to podstawą założenia mojej własnej firmy o nazwie Menlo Innovations, wraz z Tomem i Jamesem.

Co się kryje w nazwie?

Pęknięcie bańki internetowej na początku 2001 roku oznaczało koniec marzeń w Interface Systems, gdy kalifornijska firma, która przejęła nas we wrześniu 2000 roku, musiała zamknąć wszystkie swoje oddziały, w tym mój w Ann Arbor. Wiedzieliśmy z Tomem i Jamesem, jak zbudować świetną maszynownię i mogliśmy to powtórzyć. Niestety, kiedy zrobiliśmy to po raz pierwszy, zbudowaliśmy ją wewnątrz Titanica. Jak pamiętamy, los Titanica nie był zawiniony przez maszynownię. Z dzisiejszej perspektywy widać, że mieliśmy dużo szczęścia, mogąc zbudować prototyp Menlo Innovations w ramach Interface Systems.

Uruchomiliśmy naszą firmę podobnie jak zaczynali niemal wszyscy – w piwnicy mojego domu. Nazwa firmy powstała z inspiracji, jaką była dla mnie replika laboratorium Tomasza Edisona w Menlo Park w stanie New Jersey, znajdująca się w Greenfield Village w Dearborn, gdzie Henry Ford uczcił swego przyjaciela Edisona, rekonstruując jego laboratorium. James utworzył pierwotną witrynę internetową, ale nie umieścił na niej adresu firmy (moja żona nie uważała za właściwe, aby zachęcać ludzi do odwiedzania naszego domu w spokojnej dzielnicy Ann Arbor). Jednak umieściliśmy numery naszych telefonów komórkowych, aby ludzie mogli się z nami kontaktować. Kilka tygodni po uruchomieniu witryny internetowej otrzymałem telefon od kogoś, kto był bardzo zainteresowany tym, co robimy, zwłaszcza że nawiązywaliśmy do sposobu myślenia Kenta Becka i do Extreme Programming. Chciał z nami pracować. Mieszkał w północnej Kalifornii. Wyjaśniłem mu, że nie doszliśmy jeszcze do etapu zatrudniania ludzi, ale zapewniłem, że pozostaniemy w kontakcie. Nie wyobrażałem sobie, że ktoś z północnej Kalifornii może chcieć z nami pracować. (Pamiętajcie, że był to rok 2001). Jakiś miesiąc później zadzwonił do mnie znowu i pytał co nowego. Wyjaśniłem mu, że wszystko rozwija się powoli, ale obiecałem, że będę w kontakcie. Naciskał jeszcze bardziej i powiedział:

– Słuchaj, ja tylko wpadnę na lunch. Chciałbym się z wami spotkać.
Zakłopotany powiedziałem:
– Wpaść na lunch? Co masz na myśli?
– Menlo Park nie jest dla mnie aż tak daleko.
Nie mogłem się powstrzymać od śmiechu. Powiedziałem:
– Rob. My jesteśmy w Ann Arbor. W stanie Michigan!
Nie wahał się. Powiedział, że wsiądzie do samolotu. I tak zrobił!

Przez kolejne dwa lata Rob Myers raz na dwa tygodnie dojeżdżał z Kalifornii. Pomógł nam przebudować OTIS (Organ Transplant Information System) systemu opieki zdrowotnej Uniwersytetu Michigan. Wraz z nami przebudował krytyczny fragment infrastruktury technicznej w jednym z wiodących centrów przeszczepu

organów w USA. Gdy firma Menlo zakończyła swoją część pracy, po trzech latach istniało prawie 18000 testów jednostkowych. Przeszkoliliśmy zespół z Michigan w zakresie pisania testów i utrzymywania tych istniejących. W szczytowym momencie działalności przy każdej zmianie kodu wykonywano bezbłędnie prawie 30 tys. testów jednostkowych. System ten działał ponad 15 lat.

Gdybyśmy nie nazwali naszej firmy Menlo Innovations, być może nigdy nie spotkałbym Roba. W latach 2001–2003 scementowaliśmy naszą znajomość i od tego czasu jesteśmy w stałym kontakcie.

Efekt

Firma Menlo rozpoczęła działalność, mając już wdrożone wszystkie te procedury. Nigdy nie zawiedliśmy. Pracowaliśmy przy kilku systemach (jak OTIS), które dosłownie przechowywały w danych i algorytmach życie ludzi – na przykład nad cytometrem przepływowym, który ma certyfikat FDA na monitorowanie leczenia ludzi chorych na raka krwi.

Przez 25 lat naszej działalności mieliśmy do czynienia z dwoma awariami oprogramowania. Obie zostały rozwiązane w ciągu 24 godzin. Moje życie polegało na codziennym gaszeniu pożarów. Nadal nie widzę, aby te techniki były systematycznie nauczane w programach studiów. Porusza się je pobieżnie, ale nie uczy tak, jak w 1980 roku uczył mnie John Saylor.

Po raz pierwszy zetknąłem się z komputerem w celu programowania go w 1971 roku. Mogę powiedzieć, że bez wątpienia to, czego się nauczycie od jednego z najlepszych nauczycieli w tej dziedzinie, to najbardziej rewolucyjne pojęcia, jakie zostały odkryte w dziedzinie programowania komputerów. Bez dwóch zdań.

Na początku mojej kariery wierzyłem, że zawsze jest lepszy sposób. On istnieje i teraz się go nauczycie.

Można śmiało powiedzieć, że ocalił on moją karierę i przywrócił radość z wykonywania zawodu, o którym marzyłem od najmłodszych lat.

Dziękuję Ci Robie Myers za zaryzykowanie pracy z w firmie, która miała być tuż za rogiem, a okazała się być na drugim końcu kraju. Moje życie zostało dzięki temu, że wywarłeś tak ważny wpływ na nas w początkowym okresie działalności. I za to będę Ci dozgonnie wdzięczny.

– Richard Sheridan
CEO, współzałożyciel i Chief Storyteller, Menlo Innovations;
autor książek Joy, Inc.: How We Built a Workplace People Love (2013),
i Chief Joy Officer: How Great Leaders Elevate
Human Energy and Eliminate Fear (2018)

Wstęp

Test-Driven Development (TDD) to prosta praktyka, która daje deweloperom oprogramowania, zespołom i organizacjom wiele długo- i krótko terminowych korzyści. Polega ona na pisaniu automatycznego testu czyli „specyfikacji" zachowania oprogramowania tuż przed napisaniem kodu, który implementuje to zachowanie i sprawia, że test zakończy się powodzeniem.

TDD jest często opisywane jako sposób tworzenia oprogramowania zawierającego mniejszą liczbę błędów. Często jednak ludziom umyka fakt, że każdy istniejący test odgrywa istotną rolę przez cały cykl życia oprogramowania. TDD nie tylko natychmiast wyłapuje większość błędów implementacji, lecz także pozostawia za sobą reprezentatywnego strażnika dla każdego z zachowań. Ta siatka bezpieczeństwa w postaci istniejących testów pozwala deweloperom na zmianę wewnętrznej struktury kodu bez obaw o naruszenie innej części systemu, napisanej wiele miesięcy wcześniej. Daje to pewność potrzebną do wprowadzania nieoczekiwanych i innowacyjnych funkcji. Ta zdolność modyfikowania istniejącego kodu bez naruszania czegokolwiek stanowiła zawsze klucz do utrzymania i ulepszania oprogramowania.

TDD stanowi kombinację dwóch wcześniejszych praktyk Extreme Programming (XP). W 1996 roku, Kent Beck[1] sformułował XP, które zapewniło łatwiejsze do utrzymania i zdrowe podejście do tworzenia oprogramowania niż wiele wiodących ówcześnie metod. Dwie najwcześniejsze praktyki w XP to były (1) „najpierw testujemy" – pisanie testów jednostkowych przed napisaniem implementacji, która miała je przejść – oraz (2) „refaktoryzacja" – zmiana wewnętrznego projektu wynikowej implementacji. Beck dostrzegł wartość w połączeniu i opisaniu tych dwóch praktyk niezależnie od XP, a w 2002 roku opublikował książkę *Test-Driven Development by Example*[2*].

Minęło ponad 20 lat. Niestety większość zespołów tworzących oprogramowanie wciąż ma trudności z pełnym wdrożeniem TDD – i nadal ponosi konsekwencje unikania tego podejścia.

---

1 https://en.wikipedia.org/wiki/Kent_Beck

2 Polskie wydanie: *TDD. Sztuka tworzenia dobrego kodu*, Helion, 2016 (przyp. tłum.)

Po co jest ta książka?

W 1997 roku już od 12 lat zawodowo pisałem oprogramowanie i miałem tego serdecznie dość. Niezależnie od tego, jak dużo czasu poświęcaliśmy na projektowanie i testowanie na wstępie, klient w końcu wymyślał lub odkrywał scenariusz, którego nikt nie przewidział. Kod musiał się zmienić, często w istotny sposób, często rujnując nasze starannie przemyślane rozwiązania projektowe lub tworząc nowe ukryte błędy.

Więc w 1997 roku poważnie rozważałem zmianę ścieżki kariery, „Może zająć się nieruchomościami?".

Ale w roku 1998 po raz pierwszy zetknąłem się z większością praktyk opisanych w tej książce. Nagle programowanie zaczęło mieć znowu sens! Przez następnych sześć lat z ogromną przyjemnością napisałem większość najlepiej zaprojektowanych i najbardziej wolnych od błędów programów w mojej karierze, które pozostawały takimi nawet gdy – a zwłaszcza gdy – klienci prosili o coś nieoczekiwanego. Moja osobista misja związana z napisaniem tej książki jest prosta: pomóc innym tworzyć oprogramowanie dobrej jakości i cieszyć się z ich kariery. Napisałem tę książkę, aby współczesny deweloper oprogramowania mógł doświadczyć łatwości i radości, jakie ja znalazłem dzięki korzystaniu z TDD.

Celem tej książki jest przedstawienie programistom zestawu prostych – i niezbędnych! – praktyk, które mogą wykorzystać do tworzenia i utrzymania swojego kodu. TDD jest w istocie naprawdę proste. Daje nam przejrzysty zbiór kroków, dzięki którym możemy w pełni wykorzystać swoją kreatywność, talent, doświadczenie i radość z wykonywania tego rzemiosła. Za pomocą tych prostych technik możemy tworzyć, ulepszać i utrzymywać najbardziej złożone oprogramowanie, jakie tylko można sobie wyobrazić.

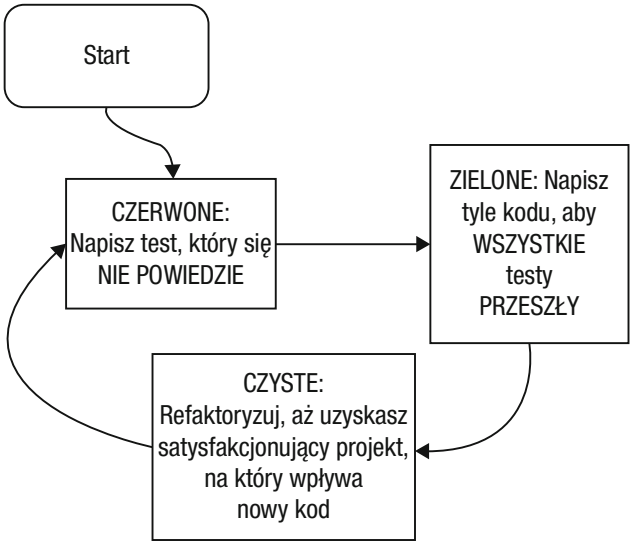
TDD w skrócie

Test-Driven Development (TDD) to praktyka, w której najpierw rozważamy, co oprogramowanie ma robić, zanim zdecydujemy, jak ma to robić. Każdy unikatowy opis danych na wejściu i oczekiwanych wyników zapisujemy w wykonywalnym i powtarzalnym formacie (test), zanim napiszemy kod, który zgodnie – jak zakładamy – przejdzie ten test. TDD wymaga od nas dostrzeżenia procesu myślowego, który ma nastąpić, zanim będziemy mogli napisać jakikolwiek kod, i utrwalenia tych przemyśleń w postaci czytelnego i powtarzalnego zestawu testów. Ten proces można podsumować w następujących krokach:

- **Krok 1. Napisz test, który zakończy się niepowodzeniem.** Napisz test dla pożądanego, ale nieistniejącego zachowania, uruchom go i sprawdź, czy zakończy się niepowodzeniem i czy zawiódł w przewidywalny sposób.
- **Krok 2. Napisz tylko tyle kodu, ile potrzeba, aby test zakończył się powodzeniem.** Napisz minimalną implementację, która przejdzie test bez pisania niczego ponad to, co jest potrzebne dla tego jednego testu i nie powodując niepowodzenia żadnych poprzednich testów.
- **Krok 3. Refaktoryzuj.** Poszukaj możliwości uporządkowania i uproszczenia struktury kodu i testów, a następnie wprowadź te zmiany w taki sposób, aby wszystkie testy nadal kończyły się powodzeniem. Ten krok nie jest opcjonalny. Często refaktoryzacja ułatwia napisanie kolejnego testu.

Można wtedy powrócić do pierwszego kroku, stopniowo rozbudowując zachowania, projekt strukturalny i zestaw testów chroniących te zachowania podczas przyszłych zmian.

Ten cykl często określa się skrótem „Czerwone – Zielone – Refaktoryzacja" lub „Czerwone – Zielone – Czyste", co pokazuje rysunek W1.



Rysunek W.1 *Najprostszy schemat blokowy „Czerwone – Zielone – Czyste"*

TDD można podsumować jako ten prosty zestaw mechanicznych kroków. Gdy jednak deweloperzy wykorzystują swoje umiejętności rozwiązywania problemów, kreatywność i doświadczenie do praktyk TDD, mogą utworzyć oprogramowanie wysokiej jakości z szybkością wynikającą z pewności siebie. Przy każdej iteracji cyklu TDD tworzą wysokiej jakości i dobrze zaprojektowane oprogramowanie, a ich rosnący zestaw szybkich i kompleksowych testów chroni wcześniejszą pracę włożoną w to rozwiązanie.

Dla zespołu wykorzystującego TDD testy w zestawie testów zespołu są wynikiem wszystkich wcześniej zrealizowanych zadań. Zestaw testów służy jako szczegółowe, aktualne i możliwe do uruchomienia specyfikacje, dokładnie opisujące to, co oprogramowanie w istocie robi. Dobre specyfikacje inżynierskie są czytelne, jasne i łatwo zrozumiałe zarówno dla obecnych, jak i przyszłych deweloperów.

Zamiast sytuacji, w której przyszłe zlecenia stają się coraz większym wyzwaniem, pochłaniają więcej czasu i powodują narastającą irytację wśród członków zespołu, dzięki stosowaniu TDD każde zlecenie może być realizowane z takim samym entuzjazmem, z jakim zespół zareagowałby na niego, gdyby pojawiło się pierwszego dnia projektu. Co więcej, ponieważ każde nowe zlecenie można zrealizować bez naruszania istniejącej funkcjonalności, czas wypuszczenia produktu na rynek znacznie się skraca.

Warunki, tematy i konwencje

Oto kilka definicji objaśniających terminy używane w tej książce.

Definicje

Implementacja: Kod, który dostarcza wartościowe zachowanie dostarczanego oprogramowania. W przeciwieństwie do kodu testowego, który sprawdza to zachowanie.

Refaktoryzacja: Każda zmiana w systemie oprogramowania, która zachowuje istniejące jawne zachowania i poprawia możliwości utrzymania.

- *Jawne zachowanie:* Zachowanie, które jest bezpośrednio opisane w kodzie za pomocą pętli, rozgałęzień, wywoływania metod i tak dalej. Przeciwnym przykładem jest szybkość wykonania: szybkość to niejawne zachowanie oparte na wielu skomplikowanych czynnikach. Żaden język programowania nie ma słowa kluczowego `faster`.
- *Możliwość utrzymania:* Miara tego, jak łatwo można rozszerzyć lub naprawić oprogramowanie. Zależy to zarówno od projektu kodu, jak i dokładności zestawu testów opracowanego przez zespół.

Projekt: Wewnętrzna struktura kodu oprogramowania, czyli sposób, w jaki deweloperzy tworzą strukturę implementacji i jak wybierają nazwy części tej struktury. Dobry projekt oprogramowania wyjaśnia I przekazuje informacje o sposobie implementacji zachowań. Dobry projekt pozwala, aby nowa funkcjonalność mogła zostać dodana bez nadmiernej pracy nad zmianami.

Unikam używania terminu „projekt", gdy definiuję „refaktoryzację", gdyż oba pojęcia są często mylnie postrzegane jako tematy dotyczące tylko deweloperów, jakby tylko oni musieli się tym interesować. Natomiast możliwość utrzymania jest rozumiana jako sposób, w jaki oprogramowanie zyskuje na jakości i wartości.

Zlecenie: Zadanie, o którego realizację proszony jest zespół deweloperów, które stworzy przyrostowy wzrost wartości dostarczanego oprogramowania Zwinne metody spopularyzowały pojęcie „historia użytkownika", oznaczające krótkie opisowe przedstawienie przykładu interakcji użytkownika z oprogramowaniem w celu osiągnięcia konkretnych celów użytkownika. Historia użytkownika jest zwykle znacznie krótsza niż „zlecenie na dodanie funkcji", ale używam tu skrótu „zlecenie" jako ogólnego terminu obejmującego oba pojęcia.

Test: Automatyczny skrypt definiujący i realizujący zachowanie oraz sprawdzający, czy wyniki zachowania są poprawne.

W tej książce czasami zamiast „test" piszę „specyfikacje" lub „scenariusz", aby podkreślić role testu jako specyfikacji (czyli dokumentacji) lub jako scenariusza (czyli przykładu).

Scenariusz: Pojedynczy przykład zachowania, który obejmuje to, co musi być prawdą, zanim to zachowanie się wydarzy oraz jakie wyniki powinno dać po zakończeniu. Wiele scenariuszy składa się na specyfikację.

Specyfikacja: Dokumentacja opisująca szczegółowo wszystkie żądania. W przypadku złożonego produktu istnieją często dwie specyfikacje: specyfikacja produktu i specyfikacja inżynierska. Mogą się one na siebie nakładać, ale specyfikacja inżynierska często obejmuje szczegóły techniczne, które nie muszą być widoczne dla użytkownika, analityka biznesowego czy menedżera produktu.

Test jednostkowy: Termin ten jest używany od dziesięcioleci i ludzie często zakładają, że każdy wie, co to znaczy. Wielu współczesnych deweloperów nie lubi tego określenia. Deweloperzy często słyszą wymaganie „musicie sprawdzić swój kod testem jednostkowym" i traktują to jak obowiązek. Wiele uczciwych prób testuje zgodność poprzez indywidualne metody testowania, gdyż inaczej wiersze kodu spowodują frustrację i zniechęcenie. Branża próbowała rozwiązać ten problem, zmieniając nazwę. „Mikrotest", „spec." albo „test deweloperski" są czasami wykorzystywane w tym celu i wszystkie one odzwierciedlają pewną prawdę dotyczącą tych testów. Na przykład „spec" przypomina nam, że piszemy fragment specyfikacji technicznych i sugeruje, że należy napisać go przed napisaniem implementacji. „Test deweloperski" przekazuje, że testy są pisane przez deweloperów, czytane przez nich i że istnieją przede wszystkim po to, aby wspierać trwający proces tworzenia.

Terminologia nie stanowi problemu – problemem jest raczej definicja „testu jednostkowego". Brakuje tu jasności: czym w istocie jest jednostka oprogramowania?[3] Klasą, funkcją czy metodą, gałęzią lub pętlą, wierszem kodu? Wszystko to stanowi zawartość i strukturę kodu, a nie to, co ten kod robi.

W całej tej książce[4] określenie „jednostkowy" rozumiem następująco: Jednostka oprogramowania to najmniejszy fragment zachowania w czasie wykonywania, który można niezależnie opisać. Implementacja jednostki zachowania, jak zobaczycie dalej, może obejmować metody lub być ukryta w logicznej klauzuli warunkowej.

W tej książce poruszono dwa istotne tematy: niezbędność i emergentność. Rzadko wymanienia się je z nazwy, ale stanowią one jej przewodnie zasady.

Polecane lektury

McKeown, Greg. Essentialism: *The Disciplined Pursuit of Less*. New York: Crown Business, 2014.

Niezbędność

Wybór słowa „essential"[5*] (niezbędny) w tytule tej książki jest celowy. Praktyki tu zawarte są niezbędne, co oznacza, że są konieczne. Niezbędne implikuje także, że nie ma nic ponadto. W swojej książce *Essentialism*, McKeown cytuje Dietera Ramsa, oryginalnego projektanta z firmy Braun: „Mniej, ale lepiej".

Jak można zbudować wartościowe oprogramowanie dobrej jakości, nie doprowadzając zespołu do wypalenia? Dzięki praktyce, która jest w istocie tak prosta, że nie przeszkadza w pracy, a jednocześnie tak skuteczna, że eliminuje niejasności, błędy i przeróbki, zastępując je elastycznością i pewnością siebie.

3 Termin „Mikrotest" jest tak samo rozmyty jak „jednostkowy": jak małe jest „mikro" i z czym je porównujemy?

4 Przeprowadziłem na LinkedIn nieformalną ankietę i znacząco wygrał „test jednostkowy", ale komentarze tłumaczące różne wybory moich kolegów były jeszcze bardziej pomocne.

5 Oryginalny tytuł książki to: *Essential Test-Driven Development* (przyp. tłum.)

Emergencja

> … proces, w którym większe jednostki, wzorce i prawidłowości pojawiają się dzięki interakcjom mniejszych lub prostszych jednostek, które same nie przejawiają takich właściwości[6].

Większość ćwiczeń i przykładów z tej książki pochodzi z kursów technicznych, które prowadzę, a każde z nich jest wykonywane za pomocą podejścia TDD i pracy w parach. Grupa mogła liczyć około 20 osób (czyli 10 par pracujących oddzielnie nad tym samym zagadnieniem), a ja mogłem zobaczyć 10 w pełni poprawnych, lecz różniących się od siebie rozwiązań.

Nigdy nie ma jednego „dobrego" projektu. Projekt oprogramowania jest mocno związany z kontekstem: składem zespołu, ich doświadczeniem i wykształceniem, używanymi technikami i wieloma innymi czynnikami – wszystko to składa się na wyjątkowość każdego projektu oprogramowania.

Metaforą używaną przeze mnie do opisania tego, co jest określane jako „projekt emergentny", jest przycinanie i kształtowanie klonu japońskiego. Nie da się z góry w pełni przewidzieć, jak będzie wyglądać drzewo. Zaczynamy od małej sadzonki, a zarówno ona, jak i gleba oraz pogoda mają wpływ na to, jak drzewo będzie rosło. Drzewo dostarcza nam informacje zwrotne, a my stopniowo pracujemy nad nim, dokonując w każdym sezonie niewielkich zmian, aby osiągnąć wzajemnie akceptowalny wynik estetyczny. Jest to kreatywna dziedzina i wymagająca wiedzy, która staje się coraz łatwiejsza wraz z nabieraniem doświadczenia.

Tak właśnie wygląda sterowane testami tworzenie oprogramowania: testy i implementacje ciągłe dostarczają sobie nawzajem informacji zwrotnej. Deweloperzy, którzy obserwują te informacje zwrotne, mają lepsze wyniki od tych, którzy próbują narzucić oprogramowaniu swoje z góry przyjęte założenia.

Dla kogo jest ta książka?

Książka ma na celu wyposażenie deweloperów w niezbędne umiejętności i techniki, które można wykorzystać w różnych dziedzinach biznesowych, językach programowania, środowiskach i narzędziach, z którymi będą mieli do czynienia w czasie swojej kariery jako twórcy oprogramowania, w tym – co jest być może najważniejsze – w nowoczesnych narzędziach opartych na sztucznej inteligencji.

Praktyki i techniki zwykle żyją dłużej niż określone technologie. TDD – a zwłaszcza podejście oparte na testach oraz praktyka utrzymywania zestawu przejrzystych

6 http://en.wikipedia.org/wiki/Emergence

i szybkich testów – ma zastosowanie wszędzie tam, gdzie będziecie pisać testy jednostkowe, scenariusze typu Gherkin lub polecenia dla agenta AI.

Współpracowałem z zespołami, które wykorzystywały praktyki opisane w tej książce do tworzenia i utrzymywania funkcji AWS Lambda, JavaScript z wykorzystaniem Promises, Angular lub React, aplikacji opartych na node.js, aplikacji Ruby on Rails, aplikacji ASP.Net/ADO.Net, PL-SQL, a nawet kodu asemblera.

Starałem się pisać przykłady zrozumiałe nawet wtedy, gdy są napisane w różnych językach programowania (przede wszystkim Java, C# i Ruby), aby pokazać (1) że TDD to efektywna praktyka niezależnie od stosowanego języka oraz (2) deweloperzy mogą zrozumieć kod w nieznanych językach, jeśli testy i implementacja są napisane w jasny sposób. Często wybieram język, który sprawia, że przykład jest nieco bardziej przejrzysty (np. słowo kluczowe override w C# sprawia, że nadpisanie jest oczywiste).

Organizacja książki

Książka jest podzielona na trzy części: „Podstawowe techniki", „Praktyki pomocnicze" oraz „Zwrot z inwestycji".

Część I: Podstawowe techniki

- Rozdział 1, „Myślenie na sposób TDD", zawiera przegląd TDD oraz analizę przydatności TDD w szybko zmieniającej się branży.
- Rozdział 2, „Podstawowe ruchy", prowadzi czytelnika przez przykład, pokazując zarówno mechanikę, jak i sposób myślenia tworzące TDD jako skuteczną dyscyplinę. Częścią tego sposobu myślenia jest traktowanie TDD jako kooperacyjnej gry typu win-win, w której komputer programisty jest jednym z graczy.
- Rozdział 3, „Wykorzystanie istniejących zachowań", stanowi kontynuację przykładu z rozdziału 2 i zawiera bardziej szczegółowe omówienie podejścia opartego na testach i sposobu podejścia do implementacji z perspektywy TDD.
- Rozdział 4, „Zachowania związane z wyjątkami", opisuje w skrócie, jak wykorzystać TDD do projektowania zachowań, które wyrzucają wyjątki lub zgłaszają błędy.
- Rozdział 5, „Utrzymanie praktyki opartej na testach", opisuje typowe pułapki i zapewnia dodatkowe techniki utrzymywania swojego zestawu testów i implementacji przez dłuższy czas. Dyscyplina, która z upływem czasu traci skuteczność, nie jest warta rozpoczynania, więc wyciągam wnioski

z doświadczeń zespołów TDD, które z powodzeniem współpracowały ze sobą przez co najmniej sześć miesięcy. Katalog popularnych „zapachów testów" został przygotowany, aby pomóc czytelnikom w utrzymywaniu ich własnych zestawów testów.

Część II: Praktyki pomocnicze

- Rozdział 6, „Dublery testowe", omawia różnorodne techniki, które pozwalają zastąpić kłopotliwe zewnętrzne zależności zewnętrzne i znacznie skrócić czas potrzebny na uruchomienia zestawów testów. Bez możliwości wyeliminowania zewnętrznych zależności w naszych testach zestaw testów będzie działał niezwykle wolno, stając się niepraktycznym obciążeniem, aż w końcu zostanie porzucony.
- Rozdział 7, „Testowanie starego kodu", obejmuje techniki testowania istniejącego, nieprzetestowanego kodu, gdy wymaga on utrzymania lub rozszerzeń. Testowanie starego kodu nie musi być strasznym lub daremnym wysiłkiem. Realizacja tych technik wzmocni również umiejętności czytelników w zakresie TDD i pozwoli zrozumieć, dlaczego TDD jest sprytniejszą „ścieżką najmniejszego oporu".

Część III: Zwrot z inwestycji

- Rozdział 8, „Czarne łabędzie", podaje biznesowy przypadek wykorzystywania TDD. Opisywane są trzy poziomy korzyści wartości: (1) mniej błędów i znacznie mniej czasu poświęcanego na usuwanie błędów, (2) skrócony „czas cyklu" (czyli czas potrzebny na utworzenie funkcji) oraz (3) większa elastyczność w reagowaniu na wartościowe, innowacyjne i nieprzewidziane prośby o dodanie nowych funkcji. Jest to dobry rozdział do przedstawienia sceptycznym członkom zespołu, menedżerom produktu, menedżerom funkcjonalności, architektom, testerom i deweloperom, niezależnie od ich wieloletniego doświadczenia.

Podziękowania

Ta książka jest znacznie lepsza dzięki wielu osobom, które ją przeczytały, oceniły i edytowały.

Moim pierwszym recenzentom:

- Lisie Crispin, której ogromne doświadczenie w zakresie zwinnego testowania pomogło mi wyjaśnić niejasne opisy i przykłady.
- Jamesowi Grenningowi, za bardzo przydatne uwagi oraz za zgodę na wykorzystanie jego akronimu ZOMBIESSS.
- Jeffowi Langrowi, za przebrnięcie przez ponad 75,000 słów pierwotnego szkicu. Jego dokładne uwagi zachęciły mnie do uproszczenia całych rozdziałów i usunięcia wielu „wątków pobocznych", co zmniejszyło rozmiary książki o co najmniej jedną trzecią. (Nie martwcie się! Nie usunąłem niczego niezbędnego!)

Mojej nieskończenie cierpliwej redaktorce naczelnej, Haze Humbert, która wyznaczył mi codzienny rytm pracy i cotygodniowe cele (to podejście wydaje mi się jakoś znajome). I mojemu poprzedniemu redaktorowi Chrisowi Guzikowskiemu, który dziesięć lat temu dał tej książce zielone światło. Zespołowi ekspertów z wydawnictwa Pearson, a są to między innymi: Chris Cleveland, Julie Nahil, Jill Hobbs, Vani Rana i Jayaprakash P (JP). Jeśli w książce zostały jakieś błędy, to wyłącznie moja wina.

Osobom, które pracowały ze mną przy przenoszeniu moich materiałów szkoleniowych do innego języka programowania lub przy aktualizacji przekładów z tej książki. Każde spotkanie potwierdzało, że najprzyjemniejszą metodą uczenia się nowego języka programowania jest programowanie w parach oparte na testach. Oto oni:

- Matt Hargett, który jako pierwszy przerobił moje ćwiczenia z Javy na C++;
- R. Jason Kerney, z którym pracowaliśmy w parach, używając języka F# i stosując zasady programowanie funkcyjnego;
- Pat Maddox, doskonały deweloper Ruby i niezwykle cierpliwy nauczyciel;

- William Pietri, który pomógł mi przenieść wszystko do Pythona w ciągu zalewie kilku dni;
- Lars Thorup, który współpracował przy tworzeniu i prowadzeniu kilku kursów TDD w językach C++ i JavaScript;
- Ted M. Young, którego głęboka wiedza i doświadczenie ocaliły moje archaiczne przykłady w Javie przed żenującym wycofaniem z użycia.

Moim mentorom w dziedzinie programowania z ostatnich 50 lat (chronologicznie): mojemu drogiemu Ojcu, nieżyjącemu już Larry'emu Allenowi, dr. Steve'owi Wamplerowi, nieżyjącemu już *wielkiemu* dr. Melvinowi K. Neville, Fredowi George oraz dr. Bobowi Kass.

Wielu przyjaciołom i kolegom, którzy pomogli mi rozwinąć się jako trenerowi, instruktorowi i autorowi. Są to: Scott Bain, Dale Emery, James Goebel, Peter Green, Bob Hartman, Elisabeth Hendrickson, Jorgen Hesselberg, Joshua Kerievsky, Richard Lawrence, Jeff McKenna, Alan Shalloway, Richard Sheridan, James Shore, Chris Sims oraz Roshi Jisho Warner.

Wreszcie ci wspaniali agenci z Serendipity, którzy sprawili, że jestem dziś tym, kim jestem:

- Wujek Pete, który podarował mi swoją starą kolekcję książek science fiction, gdy miałem 10 lat, ratując mnie z desperacji i rozpalając we mnie życiową pasję do wszystkiego, co nerdowskie.
- Drogi przyjaciel Ted Levie, mój pierwszy partner w „programowaniu w parach" około 1985 roku. Przesiadywaliśmy na trzecim piętrze w laboratorium graficznym NAU do 3 nad ranem, pijąc tanią, smakową kawę rozpuszczalną i próbując rozwiązać absurdalnie trudne projekty dr. Neville'a. Czterdzieści lat później nadal organizujemy nasze wędrówki, wycieczki na rowerach górskich i podróże samochodowe wypełnione głębokimi rozmowami (i lepszą kawą).
- Mój mąż Dave, bez którego ta książka by nie powstała. Naprawdę! Gdybym zgodnie z planem opuścił Minneapolis, zanim się spotkaliśmy, nigdy nie wziąłbym udziału w znakomitym szkoleniu w zakresie XP, które prowadził w 1998 roku Fred George. Dave nadal cierpliwie znosi moje entuzjastyczne żarty – do każdego, kto zechce słuchać – na temat TDD, XP oraz zalet i wad kodowania wspomaganego przez AI.

O autorze

Rob Myers odkrył swoją fascynację programowaniem jako nastolatek, podczas korzystania z przenośnego terminala swojego ojca do analizy i modyfikacji gier komputerowych na maszyny typu mainframe. Jego kariera w dziedzinie tworzenia oprogramowania zaczęła się w 1986 roku podczas stażu w Phoenix, jeszcze przed ukończeniem studiów na Uniwersytecie Północnej Arizony, gdzie uzyskał dyplom w zakresie informatyki i inżynierii. W 1998 roku zetknął się z Extreme Programming (XP) i uważa, że właśnie to zdarzenie uchroniło jego karierę przed typowymi wyzwaniami związanymi z tworzeniem produktów w tamtych czasach. W 2002 roku Rob zaczął prowadzić szkolenia i mentoring dla firm w zakresie XP, TDD i innych nowoczesnych praktyk tworzenia oprogramowania. W 2008 roku założył własną firmę szkoleniową, Agile Institute i od tego czasu z przyjemnością prowadzi zabawne, wciągające i bardzo nerdowskie kursy dla setek deweloperów, w tym swój flagowy kurs, Essential Test-Driven Development, który stał się podstawą tej książki.