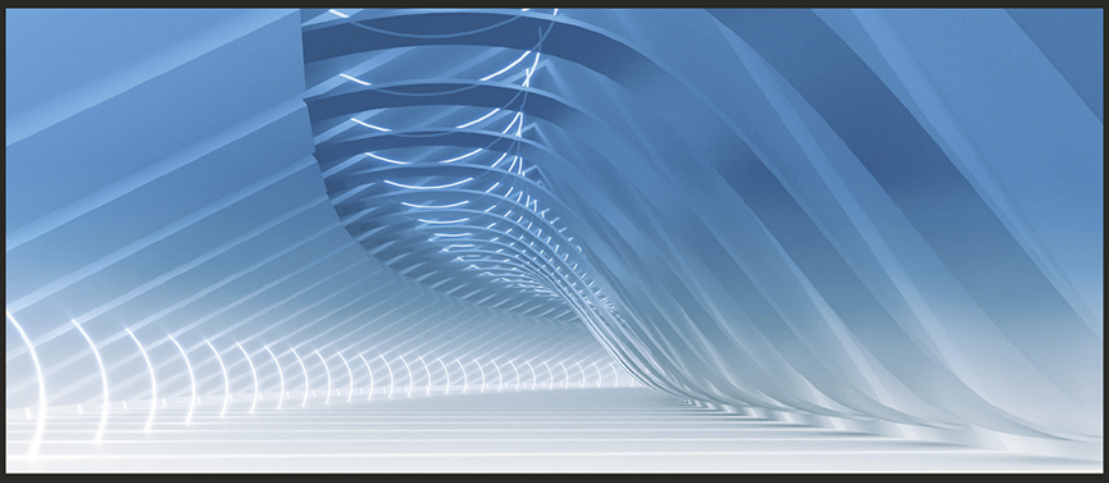
Helion

<packt>



Podejście TDD w Javie

Testowanie, SOLID i architektura heksagonalna jako fundamenty wysokiej jakości

ALAN MELLOR

Tytuł oryginału: Test-Driven Development with Java: Create higher-quality software by writing tests first with SOLID and hexagonal architecture

Tłumaczenie: Karolina Stangel

ISBN: 978-83-289-2209-9

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

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

Printed in Poland.

Spis treści

CZĘŚĆ 2. Techniki TDD

CZĘŚĆ 3. TDD w prawdziwym życiu

Obalamy mity na temat TDD

Rozdział 3

Programowanie sterowane testami (ang. *test-driven development*, TDD) przynosi wiele korzyści programistom i firmom. Nie jest jednak standardem w prawdziwych projektach. Często mnie to zaskakuje. Wykazano, że TDD poprawia jakość wewnętrznego i publicznie dostępnego kodu w różnych warunkach. Działa dla kodu frontendowego i backendowego, niezależnie od branży. Stosowałem to podejście w pracy nad systemami wbudowanymi, produktami do wideokonferencji, aplikacjami desktopowymi i flotami mikrousług.

Aby lepiej zrozumieć, w czym tkwi problem, poznamy częste uprzedzenia względem metodyki TDD, a następnie zobaczymy, jak możemy je przełamać. Jeżeli zrozumiemy, w czym ludzie dopatrują się trudności, możemy przygotować się do roli ambasadorów TDD i pomóc innym zmienić zdanie. Omówię tutaj sześć często powtarzanych mitów, które otaczają TDD, i postaram się sformułować na nie konstruktywną odpowiedź.

W tym rozdziale zajmiemy się następującymi mitami:

- „Pisanie testów mnie spowalnia".
- „Testy nie zapobiegają wszystkim błędom".
- „Skąd wiemy, że testy są poprawne?".
- „TDD to gwarancja dobrego kodu".
- „Kod jest zbyt skomplikowany, aby go testować".
- „Przed napisaniem kodu nie wiadomo, co testować".

„Pisanie testów mnie spowalnia"

To, że pisanie testów spowalnia rozwój oprogramowania, stanowi częsty zarzut przeciwko TDD. Taka krytyka nie jest zupełnie nieuzasadniona. Sam zawsze miałem poczucie, że dzięki TDD jestem szybszy, ale badania naukowe tego nie potwierdzają. Metaanaliza 18 badań pierwotnych przeprowadzona przez Association for Computing Machinery pokazała, że stosowanie TDD poprawiało produktywność w kontekście akademickim, ale spowalniało rozwój w warunkach komercyjnych. Nie jest to jednak koniec historii.

Zrozumienie korzyści ze spowolnienia

Wspomniane wcześniej badanie wskazywało, że dodatkowy czas poświęcony na TDD zwracał się w postaci mniejszej liczby defektów w gotowej wersji oprogramowania. Dzięki TDD błędy wykrywano i usuwano dużo szybciej niż w przypadku innych podejść.

Rozwiązując problemy przed etapem **zapewnienia jakości** (ang. *quality assurance*, QA), wdrożeniem, wydaniem czy zgłoszeniem defektu przez użytkownika, TDD pozwala wyeliminować dużą część niepotrzebnego wysiłku.

Różnicę w ilości pracy do wykonania możemy zobaczyć na rysunku 3.1.

Rysunek 3.1. Niekorzystanie z TDD spowalnia prace ze względu na przeróbki

Górny rząd reprezentuje rozwój funkcjonalności z użyciem TDD, w którym to przypadku mamy wystarczającą liczbę testów, aby zapobiec pojawianiu się defektów na produkcji. Dolny rząd przedstawia z kolei rozwój tej samej funkcjonalności w stylu **„koduj i naprawiaj"**, czyli bez użycia TDD, gdy o defekcie dowiadujemy się z produkcji. Bez TDD wykrywamy usterki bardzo późno, użytkownicy się denerwują, a przeróbki kosztują dużo czasu. Zauważ, że w przypadku podejścia „koduj i naprawiaj" docieramy do etapu zapewnienia jakości dużo szybciej, ale potem musimy uwzględnić wszystkie poprawki spowodowane przez niewykryte defekty. Poprawki są tym, czego ten mit nie bierze pod uwagę.

Stosując TDD, wszelkie projektowanie i testowanie realizujemy jawnie na samym początku. Przekuwamy to w wykonywalne testy i dokumentujemy w tej formie. Niezależnie od tego, czy piszemy testy, czy nie, spędzamy tyle samo czasu na zastanawianiu się nad tym, jakie specyficzne wymagania kod musi realizować. Okazuje się, że mechaniczne napisanie testu zajmuje bardzo niewiele czasu. Możesz go zmierzyć, gdy będziemy dodawać pierwszy test w rozdziale 5. „Piszemy pierwszy test". Całkowity czas spędzony nad fragmentem kodu obejmuje projektowanie, kodowanie i testowanie. Nawet gdy nie przygotowujemy testów automatycznych, czas projektowania oraz programowania pozostaje na stałym poziomie i stanowi czynnik dominujący.

Innym aspektem, który łatwo się pomija w rozważaniach, jest czas poświęcony na testy manualne. Bez cienia wątpliwości wszelki kod będzie testowany. Pytanie tylko kiedy i przez kogo. Jeżeli napiszemy test najpierw, przez nas — programistów. Stanie się to, zanim napiszemy pierwszą linijkę potencjalnie wadliwego kodu. Jeżeli zostawimy to zadanie testerom manualnym, spowolnimy cały proces rozwoju. Będziemy zmuszeni poświęcić czas na wyjaśnianie współpracownikom, jakie są kryteria sukcesu dla danego kodu, a oni będą musieli opracować plan testów manualnych, który wymaga napisania, sprawdzenia, zatwierdzenia i udokumentowania.

Wykonywanie testów manualnych jest bardzo czasochłonne. Ogólnie rzecz biorąc, cały system trzeba zbudować i uruchomić w środowisku testowym. Konieczne jest też ręczne przygotowanie baz danych, aby zawierały znane rekordy. Potem trzeba klikać po **interfejsie użytkownika**, aby przejść do odpowiedniego ekranu, na którym można wykonać

nowy kod. Wynik wymaga manualnej weryfikacji i podjęcia decyzji o jego poprawności. Te kroki trzeba wykonywać ręcznie za każdym razem, gdy pojawi się zmiana.

Co gorsza, im później decydujemy się na rozpoczęcie testowania, tym większa szansa, że będziemy się opierać na już istniejącym wadliwym kodzie. Nie możemy wiedzieć, że tak się dzieje, gdyż go jeszcze nie przetestowaliśmy. Często trudno to cofnąć. W niektórych projektach tak daleko odchodzi się od tego, co jest w głównej gałęzi kodu, że programiści zaczynają wysyłać sobie zmiany mailem. Fakt, że tworzymy dalszy kod na podstawie tego błędnego, sprawia, iż jeszcze trudniej go usunąć. To przykład złych praktyk, które zdarzają się jednak w prawdziwych projektach.

TDD to całkowite przeciwieństwo tego, co właśnie opisałem. Dzięki TDD konfiguracja, wykonanie poszczególnych kroków oraz sprawdzenie wyników przebiegają automatycznie. Mówimy o redukcji na skali czasu od minut w przypadku testów manualnych do milisekund w przypadku testów jednostkowych TDD. Ten czas oszczędzamy za każdym razem, gdy uruchamiamy dany test.

Chociaż testy manualne nie są tak wydajne jak TDD, istnieje jeszcze gorsza opcja — brak testów w ogóle. Gdy na produkcji dochodzi do awarii, oznacza to, że zrzuciliśmy na użytkowników zadanie testowania kodu. W takiej sytuacji mogą się pojawić konsekwencje finansowe i ryzyko utraty reputacji. W najlepszym przypadku to bardzo wolny sposób na wykrywanie defektów. Identyfikowanie wadliwych linii kodu na podstawie logów produkcyjnych i zawartości baz danych jest bardzo czasochłonne, a z doświadczenia wiem, że również niezwykle frustrujące.

To zabawne, że w projekcie, w ramach którego nigdy nie udaje się znaleźć czasu na napisanie testów jednostkowych, *zawsze* jest czas na ślęczenie nad logami produkcyjnymi, wycofywanie wypuszczonego kodu, publikowanie komunikatów marketingowych i porzucanie wszelkich innych zadań, aby ratować sytuację. Czasami mam wrażenie, że w niektórych podejściach do zarządzania oszczędza się minuty kosztem dni.

W TDD istnieje oczywiście początkowy koszt w postaci czasu potrzebnego na napisanie testów, ale w zamian narażamy się na mniej defektów do poprawy na produkcji, a to oznacza ogromną oszczędność pieniędzy, czasu i reputacji w porównaniu do wielu cyklów usuwania błędów w działającym już systemie.

Przełamywanie zarzutów dotyczących spowolnienia wynikającego z testów

Szukając dobrych argumentów, zwróć uwagę na czas stracony z powodu nieudanego wdrożenia lub błędów niewykrytych na etapie testowania manualnego. Znajdź zgrubne szacunki, ile czasu zajęło naprawienie ostatniego błędu wychwyconego dopiero na produkcji. Pomyśl, jaki brakujący test jednostkowy mógł uratować sytuację. Następnie oblicz, ile zajęłoby napisanie takiego testu. Przedstaw te liczby zainteresowanym stronom. Być może nawet skuteczniej byłoby oszacować całkowity koszt programowania i utracone dochody.

Ze świadomością, że testy przynoszą ogólną korzyść w postaci mniejszej liczby defektów, przeanalizujmy kolejne częste zastrzeżenie, zgodnie z którym testy są bezwartościowe, gdyż nie zapobiegają wszystkim błędom.

„Testy nie zapobiegają wszystkim błędom”

Te obiekcje do testowania wszelkiego rodzaju mają bardzo długą tradycję. Choć to oczywiście prawda, że testy nie wychwytują wszystkich błędów, oznacza to jednak raczej, iż potrzebujemy więcej skuteczniejszych testów, a nie mniej. Aby przygotować celną odpowiedź, musimy zrozumieć przyczyny tej postawy.

Dlaczego ludzie twierdzą, że testy nie wychwytują wszystkich błędów?

Możemy od razu zgodzić się z tym stwierdzeniem. Testy nie wychwytują wszystkich błędów. Dokładnie rzecz biorąc, udowodniono, że testowanie oprogramowania może wykryć tylko obecność defektów. Nie jest jednak w stanie udowodnić ich braku. Nawet w przypadku całej rzeszy przechodzących testów defekty nadal mogą się ukrywać w miejscach, których nie przetestowano.

Wydaje się to mieć zastosowanie również do innych dziedzin. Badania medyczne nie zawsze wykrywają zmiany, które nie są wystarczająco wyraźne. Testy samolotów w tunelu aerodynamicznym nie zawsze pozwalają zidentyfikować problemy, które pojawiają się w specyficznych warunkach lotu. Badanie próby partii w fabryce czekolady nie wychwyci wszystkich produktów niespełniających wymagań.

To, że nie można wykryć wszystkich błędów, nie oznacza jednak, iż testowanie jest bezwartościowe. Każdy napisany test, który wychwycił jeden defekt, to jeden defekt mniej w systemie. TDD zapewnia procedury, które pomagają myśleć o testowaniu podczas programowania, ale nadal istnieją obszary, w których testy nie będą skuteczne. Obejmują one:

- testy, o których napisaniu nie pomyśleliśmy;
- defekty, które wynikają z interakcji na poziomie systemów.

To testy, których nie napisaliśmy, stanowią prawdziwy problem. Nawet kiedy zaczynamy pracę od testów zgodnie z TDD, musimy wykazać się wystarczającą dyscypliną, aby dodać test do każdego scenariusza, który ma działać. Łatwo utworzyć test, a następnie napisać kod, dla którego ten test przechodzi. Kusi wtedy, aby programować dalej, ponieważ jesteśmy na fali. Łatwo przeoczyć jakiś przypadek brzegowy i nie napisać dla niego testu. Jeżeli brakuje nam testu, umożliwiamy pojawienie się defektu, który nie zostanie wykryty od razu.

Problem z interakcjami na poziomie systemu odnosi się do zachowań pojawiających się, gdy łączymy dwie przetestowane części oprogramowania. Interakcje między nimi mogą być bardziej skomplikowane, niż oczekiwano. Ściśle rzecz biorąc, jeżeli połączymy dwa dobrze przetestowane komponenty, nowe połączenie samo w sobie nie jest przetestowane. Niektóre interakcje mogą zawierać usterki, które są widoczne jedynie w ramach tych interakcji, nawet jeżeli części składowe oddzielnie przeszły wszystkie testy.

Oba wspomniane tutaj problemy są istotne i zasługują na uwagę. Testowanie nigdy nie wykryje wszystkich możliwych usterek, ale nie w tym tkwi jego główna wartość. Każdy test, który piszemy, zmniejsza liczbę potencjalnych błędów o jeden.

Nie testując nic, nigdy nie wykryjemy nieprawidłowości i nie zapobiegniemy żadnym defektom. Jeżeli będziemy utrzymywać testy, to niezależnie od ich liczby poprawimy jakość kodu. Zapobiegniemy wszystkim defektom, które te testy są w stanie wykryć. Możemy zobaczyć, jak płytki jest początkowy argument. To, że nie możemy sprawdzić wszystkiego, nie znaczy, iż nie powinniśmy dołożyć odpowiednich starań.

Przełamywanie zarzutów, że testy nie wychwytują wszystkich błędów

Sformułujmy problem inaczej. Możemy być pewni, że TDD zapobiega wielu rodzajom błędów. Nie wszystkim błędom — z pewnością, ale zestaw tysięcy testów przekłada się na istotną poprawę jakości aplikacji.

Aby wyjaśnić to współpracownikom, możemy wykorzystać znane analogie. Fakt, że silne hasło nie zatrzyma każdego hakera, nie znaczy, iż nie powinniśmy w ogóle używać haseł, wystawiając się na ataki *wszystkich* hakerów. Dbanie o formę nie zapobiegnie każdej chorobie, ale może uchronić nas przed wieloma poważnymi problemami zdrowotnymi.

Koniec końców jest to kwestia równowagi. Z jednej strony brak testów to zdecydowanie za mało, gdyż każdy defekt przedostanie się na produkcję. Z drugiej wiemy jednak, że testowanie nie może wyeliminować wszystkich błędów. Kiedy powinniśmy więc skończyć? Ile testów *wystarcza*? Powiedziałbym, że TDD pomaga podjąć decyzję na temat liczby testów w najlepszym możliwym momencie — wtedy, gdy myślimy o pisaniu kodu. Automatyczne testy TDD, które tworzymy, pozwalają zaoszczędzić czas na etapie testów manualnych realizowanych przez specjalistów QA. Tej ręcznej pracy nie trzeba będzie więcej wykonywać. Oszczędności czasu i pieniędzy stopniowo rosną, gdyż testy przynoszą korzyści przy każdej iteracji.

Teraz gdy już rozumiesz, dlaczego testowanie zawsze jest nieporównywalnie lepsze niż jego brak, możemy przyjrzeć się kolejnemu częstemu zastrzeżeniu. Dotyczy ono poprawności samych testów.

„Skąd wiemy, że testy są poprawne?”

To zastrzeżenie również ma swoją rację bytu, więc powinniśmy zrozumieć logikę, która się za nim kryje. Takie obiekcje często podnoszą osoby niezaznajomione z pisaniem testów automatycznych, gdyż nie rozumieją, w jaki sposób unikamy niepoprawnych testów. Jeżeli wyjaśnimy, jak zabezpieczamy się przed błędami, możemy im pomóc zmienić sposób myślenia.

Rozumienie obaw dotyczących pisania złych testów

Jednym z zastrzeżeń, z którym często można się spotkać, jest: *„Skąd wiemy, że testy są prawidłowe, skoro nie piszemy testów do testów?”*. Z takim problemem skonfrontowano mnie, gdy po raz pierwszy wprowadziłem testy jednostkowe w moim zespole. Ta kwestia

spowodowała rozłam. Część zespołu zrozumiała wartość testów od razu, innym było to obojętne, a jeszcze inni byli zdecydowanie wrogo nastawieni. Interpretowali tę nową praktykę jako sugestię, że robią coś źle. Traktowali ją jako zagrożenie. W tej sytuacji jeden z programistów wskazał na pewne braki w logice moich wyjaśnień.

Powiedziałem zespołowi, że nie możemy ufać wzrokowym przeglądom kodu produkcyjnego. Co prawda wszyscy jesteśmy biegli w czytaniu kodu, lecz jesteśmy tylko ludźmi, więc pewne rzeczy nam umykają. Testy jednostkowe pomagają tego uniknąć. Jeden bystry programista zadał wspaniałe pytanie: jeżeli inspekcja wzrokowa nie działa w przypadku kodu produkcyjnego, dlaczego twierdzimy, że sprawdzi się w przypadku testów? Jaka jest między nimi różnica?

Na dobre zobrazowanie tego problemu natrafiłem, gdy musiałem przetestować pewien wynik w formacie XML, co zdarzyło się — jak pamiętam — w 2005 roku. Kod weryfikujący ten wynik był naprawdę skomplikowany. Krytyka była uzasadniona. Nie było sposobu, abym mógł wzrokowo sprawdzić kod i z czystym sumieniem stwierdzić, że nie zawiera defektów.

Aby rozwiązać ten problem, użyłem TDD. Napisałem klasę pomocniczą, która porównywała dwa ciągi znaków XML, i informowała, czy były takie same oraz jaka była pierwsza różnica. Ponadto można było ją skonfigurować, aby ignorowała kolejność elementów XML. Wyciągnąłem ten skomplikowany kod z oryginalnego testu i zastąpiłem go wywołaniem klasy pomocniczej. Wiedziałem, że ta klasa nie zawierała defektów, gdyż przeszła każdy test TDD, który dla niej napisałem. Było tam wiele testów, które pokrywały wszystkie interesujące mnie scenariusze pozytywne i przypadki brzegowe. Oryginalny test, który był obiektem krytyki, stał się krótki i jasny.

O przegląd kodu poprosiłem kolegę, który wcześniej zgłosił zastrzeżenie. Na podstawie inspekcji wzrokowej zgodził się, że w tej nowej, prostszej formie test wygląda poprawnie. Dodał jednak, że pod warunkiem, iż *„klasa pomocnicza działa poprawnie”*. Tego byłem pewien, ponieważ przechodziła wszystkie testy TDD, które dla niej napisałem.

Dla pewności testujemy testy

U podstaw tego rozwiązania leży to, że prosty i krótki kod można poddać inspekcji wzrokowej. Powinniśmy się starać, aby większość testów jednostkowych była zwięzła i nieskomplikowana. Gdy testy stają się zbyt zawiłe, wyciągnij złożony kod do osobnego fragmentu i pracuj nad nim, stosując TDD. Oryginalny test będzie wystarczająco prosty, aby poddać go inspekcji wzrokowej, a pomocniczy kod testowy zostanie przetestowany osobnym zestawem testów. To klasyczny przykład zastosowania metody „dziel i zwyciężaj”.

W praktyce zachęcaj współpracowników do wskazywania miejsc, w których według ich oceny kod testowy jest zbyt skomplikowany, aby mu ufać. Przeprowadzaj refaktoryzację, aby móc używać prostych klas pomocniczych napisanych zgodnie z praktyką TDD. To podejście pomaga budować zaufanie, wyraża szacunek do zastrzeżeń współpracowników oraz pokazuje, jak tworzyć proste i nadające się do inspekcji wzrokowej testy TDD.

Teraz już wiesz, jak możesz upewnić się, że testy są poprawne. Kolejnym problemem związanym z TDD jest zbytnie poleganie na tej metodyce. Niektórzy wierzą, iż przestrzeganie procesu TDD gwarantuje dobry kod. Czy to prawda? Przeanalizujmy argumenty.

„TDD to gwarancja dobrego kodu”

Niektórzy podchodzą do TDD zbyt krytycznie, ale istnieją też niepoprawni optymiści, którzy wierzą, że TDD *gwarantuje* dobry kod. Ponieważ TDD definiuje pewien proces, który rzekomo poprawia jakość kodu, ktoś może założyć, że zastosowanie TDD to wszystko, czego potrzeba do zagwarantowania dobrego kodu. Niestety to nieprawda. TDD pomaga programistom pisać dobry kod oraz dostarcza informacji zwrotnych pokazujących, gdzie popełnili błędy w projektowaniu i logice. Nie stanowi jednak gwarancji dobrego kodu.

Zrozumienie problemu nadmiernych oczekiwań

W tym przypadku mamy do czynienia z nieporozumieniem. TDD nie jest zestawem technik, które bezpośrednio wpływają na decyzje projektowe. To zestaw technik, które pozwalają określić, czego oczekujemy od danego fragmentu kodu w określonych warunkach i przy pewnych założeniach projektowych. Zostawia nam swobodę wyboru pod względem projektowym i implementacyjnym.

TDD nie zawiera sugestii dotyczących wyboru dłuższej nazwy zmiennej. Nie podpowiada, czy wybrać interfejs, czy klasę abstrakcyjną. Czy funkcjonalność trzeba podzielić na dwie klasy, czy na pięć? TDD nie ma na to odpowiedzi. Czy powinniśmy wyeliminować powielony kod? Czy odwrócić zależność? Czy może połączyć się z bazą danych? To my decydujemy. TDD nie daje gotowych rozwiązań. To nie jest inteligentny byt. *Nie może zastąpić nas i naszej wiedzy.* To prosty proces, który ułatwia weryfikowanie pomysłów i założeń.

Zarządzanie oczekiwaniami względem TDD

W mojej ocenie TDD przynosi ogromne korzyści, ale musimy uwzględnić pewien kontekst. To podejście zapewnia natychmiastowe informacje zwrotne na temat podjętych decyzji, ale ważne decyzje projektowe zostawia programistom.

Stosując TDD, mamy swobodę pisania kodu zgodnie z zasadami **SOLID**, które omówię w rozdziale 7. „TDD i SOLID wspierają projektowanie”. Możemy programować w ramach podejścia proceduralnego, obiektowego lub funkcyjnego. TDD pozwala wybrać taki algorytm, jaki nam pasuje. Daje możliwość zmiany zdania na temat tego, jak chcemy zaimplementować rozwiązanie. Działa niezależnie od języka programowania i charakteryzuje się skutecznością w każdej branży.

Wspierając współpracowników w przełamywaniu oporów, pomagamy im zdać sobie sprawę z tego, że TDD to nie jest jakiś magiczny system, który zastępuje inteligencję i umiejętności programisty. Wspiera te umiejętności dzięki natychmiastowym informacjom zwrotnym na temat podjętych decyzji. Chociaż może to rozczarować tych, którzy mieli nadzieję, że ta technika umożliwi tworzenie idealnego kodu na podstawie wadliwych założeń, możemy też śmiało powiedzieć, iż TDD daje nam czas na przemyślenia. Korzyść jest taka, że stawiamy myślenie oraz projektowanie na pierwszym i honorowym miejscu. Pisząc test zakończony niepowodzeniem przed kodem produkcyjnym, który umożliwia przejście testu, upewniamy się, że przemyśleliśmy, co kod powinien robić i w jaki sposób można go użyć. To istotna korzyść.

Skoro rozumiesz już, że TDD nie zaprojektuje za Ciebie kodu, ale mimo to jest Twoim sojusznikiem, zastanówmy się, jak podejść do testowania skomplikowanego kodu.

„Kod jest zbyt skomplikowany, aby go testować"

Zawodowi programiści stale mają do czynienia z wysoce skomplikowanym kodem. Tak po prostu jest. Czasami aż ciśnie się na usta stwierdzenie, że dany kod jest zbyt skomplikowany, aby napisać do niego testy jednostkowe. To cenne spostrzeżenie. Możemy na przykład pracować z bardzo wartościowym i zaufanym **kodem odziedziczonym** (ang. *legacy code*), odpowiedzialnym za istotną część przychodów. Taki kod może być skomplikowany. Czy jest jednak *zbyt* skomplikowany, aby go testować? Czy można powiedzieć, że zawiłych fragmentów kodu po prostu nie da się testować?

Rozumienie przyczyn nietestowalnego kodu

Istnieją trzy przyczyny, dla których kod staje się skomplikowany i trudny do testowania:

- Niezamierzenie wybieramy trudniejszy sposób realizacji zadania, zwiększając zbędną złożoność.
- Nie możemy kontrolować systemów zewnętrznych i skonfigurować ich pod testy.
- Kod jest tak zagmatwany, że przestajemy go rozumieć.

Zbędna złożoność sprawia, że kod trudno się czyta i testuje. Zacznijmy od uświadomienia sobie, że każdy problem można rozwiązać na wiele sposobów. Powiedzmy, że chcemy dodać do siebie pięć liczb. Możemy napisać pętlę. Możemy też utworzyć pięć współbieżnych zadań, z których każde weźmie jedną z tych liczb, a następnie przekaże ją innemu współbieżnemu zadaniu, które wyliczy sumę. Przysięgam, że widziałem takie rzeczy na własne oczy, więc proszę Cię o jeszcze odrobinę cierpliwości. Moglibyśmy również zbudować skomplikowany system na bazie wzorców projektowych, w ramach którego każda liczba uruchamiałaby obserwatora, który umieszczałby ją w kolekcji, co uruchamiałoby kolejnego obserwatora dodającego tę liczbę do sumy, co z kolei uruchamiałoby jeszcze innego obserwatora co dziesięć sekund od wprowadzenia ostatnich danych wejściowych.

Wiem, niektóre z tych rozwiązań wyglądają mało poważnie. Część z nich rzeczywiście zmyśliłem. Odpowiedzmy sobie jednak szczerze na pytanie, iluż kiepskich decyzji projektowych byliśmy autorami. Sam na pewno pisałem kod, który był bardziej skomplikowany, niż wymagała tego sytuacja.

W przykładzie z dodawaniem pięciu liczb prawidłowym rozwiązaniem jest prosta pętla. Wszystko inne to zbędna złożoność — niepotrzebna i niezamierzona. Dlaczego ktoś miałby wybrać inne rozwiązanie? Może istnieć wiele powodów, w tym ograniczenia w ramach projektu, wytyczne kierownictwa lub po prostu osobiste preferencje. W każdym razie stało się: prostsze rozwiązanie było na wyciągnięcie ręki, ale go nie wybraliśmy.

Testowanie bardziej skomplikowanego kodu wymaga zazwyczaj bardziej skomplikowanych testów. Czasami zespół uważa, że nie ma sensu tracić czasu. Kod jest zawiły, trudno będzie napisać do niego testy, a wiemy, że działa. Czasami wydaje się, że najlepiej nic nie ruszać.

Systemy zewnętrzne są kolejną przyczyną utrudnień w testowaniu. Wyobraźmy sobie, że kod komunikuje się z zewnętrzną usługą sieciową. W takim przypadku ciężko napisać

powtarzalny test. Kod przetwarza odpowiedź usługi zewnętrznej, a ta przesyła za każdym razem inne dane. Nie możemy napisać testu i sprawdzić odpowiedzi otrzymanej z usługi, gdyż nie wiemy, co powinna zawierać. Gdybyśmy mogli zastąpić tę zewnętrzną usługę jakąś usługą testową, którą moglibyśmy kontrolować, łatwo rozwiązalibyśmy ten problem. Jeżeli kod na to nie pozwala, natrafiamy na mur.

Zagmatwany kod to następna odsłona problemu. Aby napisać test, musimy rozumieć, co kod robi z danymi wejściowymi. Czego oczekiwać na wyjściu? Jeżeli mamy fragment kodu, którego po prostu nie rozumiemy, nie możemy do niego utworzyć testu.

Chociaż to trzy rzeczywiste problemy, ich przyczyna jest jedna i ta sama — pozwoliliśmy, aby oprogramowanie popadło w taki stan. Mogliśmy zadbać o to, aby korzystało tylko z prostych algorytmów i struktur danych. Mogliśmy odizolować systemy zewnętrzne tak, aby przetestować resztę kodu bez nich. Mogliśmy dokonać modularyzacji kodu, żeby nie był tak zagmatwany.

Jak mamy rozmawiać z zespołem o tych kwestiach?

Nowe podejście do związku między dobrym projektem a prostymi testami

Wszystkie wcześniejsze problemy wynikają z tworzenia oprogramowania, które działa, ale nie przestrzega zasad dobrego projektowania. Z mojego doświadczenia najbardziej skutecznym sposobem na zmianę tego podejścia jest technika **programowania w parach** (ang. *pair programming*), czyli wspólna praca nad tym samym fragmentem kodu i pomaganie sobie nawzajem w znajdowaniu lepszych rozwiązań projektowych. Jeżeli programowanie w parach nie wchodzi w grę, przeglądy kodu również stanowią punkt kontrolny, który umożliwia pozytywne zmiany projektowe. Programowanie w parach jest lepsze, ponieważ w momencie, gdy następuje wreszcie przegląd kodu, czasami jest już zbyt późno na szeroko zakrojone zmiany. Taniej, lepiej i szybciej jest zapobiegać złym decyzjom projektowym, niż je poprawiać.

Radzenie sobie z odziedziczonym kodem pozbawionym testów

W prawdziwym życiu nie raz mamy do czynienia z odziedziczonym kodem, który musimy utrzymywać. Często jest tak, że rozrósł się on do monstrualnych rozmiarów i najlepiej byłoby go przepisać, ale nikt już nie wie, jak on działa. Czasami próżno szukać pisemnej dokumentacji lub specyfikacji, która pozwalałaby to zrozumieć. W innych przypadkach to, co napisano, jest już dawno nieaktualne i bardziej przeszkadza, niż pomaga. Autorzy starego kodu przeszli do innego zespołu lub odeszli z firmy.

Najlepszym rozwiązaniem jest w miarę możliwości zostawić taki kod w spokoju. Czasami musimy jednak dodawać nowe funkcjonalności, co wymaga wprowadzania zmian. W związku z tym, że nie mamy gotowych testów, dodanie nowego testu okaże się najpewniej niemożliwe. Kodu po prostu nie da się podzielić tak, aby uzyskać dostęp do punktów, o które możemy zaczepić kod testowy.

W takich przypadkach pomocna okazuje się technika **testów charakteryzacyjnych** (ang. *characterization test*). Możemy ją streścić w trzech punktach:

1. Uruchom kod odziedziczony z każdą możliwą kombinacją danych wejściowych.
2. Zarejestruj wyniki dla wszystkich zebranych danych wejściowych. Ten wynik nazywany jest zazwyczaj złotym wzorcem (ang. *golden master*).
3. Napisz test charakteryzacyjny, który uruchamia kod ze wszystkimi danymi wejściowymi. Porównaj każdy wynik ze złotym wzorcem. Jeżeli istnieje jakakolwiek różnica, test nie przejdzie.

Zautomatyzowany test porównuje wynik zmienionego kodu z wcześniejszym wynikiem. Umożliwia to bezpieczne przeprowadzenie refaktoryzacji odziedziczonego kodu. Możemy wtedy użyć standardowych technik refaktoryzacji w połączeniu z TDD. Zapisując wadliwe wyniki jako złoty wzorzec, mamy pewność, że w ramach tego etapu dokonaliśmy jedynie refaktoryzacji. Unikamy tym samym pułapki jednoczesnej refaktoryzacji kodu i poprawiania błędów. Jeżeli w oryginalnym kodzie obecne są defekty, prace realizujemy w dwóch etapach. Najpierw przeprowadzamy refaktoryzację bez zmieniania zachowania. Następnie naprawiamy błąd w ramach oddzielnego zadania. Nigdy nie naprawiamy błędów i nie refaktorujemy kodu w tym samym czasie. Dzięki testom charakteryzacyjnym mamy pewność, że nie mieszamy przypadkiem tych dwóch zadań.

W tym podrozdziale omówiłem, jak TDD pomaga uporać się ze zbędną złożonością, i pokazałem trudności wynikające ze zmieniania odziedziczonego kodu. Na pewno przyszła Ci w którymś momencie do głowy myśl, że napisanie testu przed kodem produkcyjnym oznacza, iż musimy wiedzieć, jak ten kod produkcyjny będzie wyglądał. Przyjrzyjmy się tej kwestii.

„Przed napisaniem kodu nie wiadomo, co testować”

Osoby uczące się TDD często są bardzo sfrustrowane, gdyż przed napisaniem kodu produkcyjnego nie są pewne, co testować. Nie jest to oczywiście zarzut wyssany z palca. Zacznę od wyjaśnienia, jakich trudności doświadczają programiści, a następnie przejdę do rozwiązania w postaci techniki, którą możemy z łatwością zastosować w pracy bez zmiany obranego podejścia.

Rozumienie trudności związanych z rozpoczynaniem pracy od pisania testów

W pewnym stopniu to naturalne, że myślimy o tym, jak zaimplementować kod. W końcu tak się uczymy programować. Piszemy `System.out.println("Witaj, Świecie!");` i nie zastanawiamy się nad jakąś specjalną strukturą dla tej kultowej instrukcji. Małe programy i narzędzia działają bez problemu, gdy tworzymy je liniowo, czyli tak, jak byśmy pisali listę zakupów.

Trudności pojawiają się, gdy programy się rozrastają. Potrzebujemy pomocy w organizowaniu kodu w łatwe do zrozumienia części. Chcemy, aby same się opisywały i były

wygodne w użyciu. Im większa baza kodu, tym mniej interesuje nas wnętrze tych części, a tym ważniejsze są *zewnętrzna* struktura i połączenia.

Powiedzmy, że piszemy klasę `TextEditorWidget`. Chcemy sprawdzać pisownię w czasie rzeczywistym. Znajdujemy bibliotekę zawierającą klasę `SpellCheck`. Nie obchodzi nas za bardzo, jak działa ta klasa. Ważne jest jedynie to, jak możemy jej użyć do sprawdzania pisowni. Musimy wiedzieć, jak utworzyć obiekt tej klasy, jakie metody wywołać, aby sprawdziła pisownię w tekście, oraz co zrobić z danymi wyjściowymi.

Tego rodzaju podejście to definicja projektowania oprogramowania — myślimy o tym, jak połączyć ze sobą różne komponenty. W miarę jak baza kodu rośnie, musimy skupić się na projektowaniu, jeżeli chcemy utrzymać kod w odpowiednim stanie. Stosujemy hermetyzację, aby ukryć szczegóły struktur danych oraz algorytmów w ramach funkcji i klas. Zapewniamy proste w użyciu interfejsy programowania.

Co zrobić, żeby nie musieć pisać kodu produkcyjnego na początku?

TDD przygotowuje grunt pod decyzje projektowe. Pisząc testy przed kodem produkcyjnym, definiujemy, w jaki sposób testowany kod ma być używany i wywoływany. TDD pomaga szybko stwierdzić, jakie są konsekwencje podejmowanych decyzji. Jeżeli test pokazuje, że utworzenie obiektu jest skomplikowane, mamy świadomość, iż krok tworzenia obiektu powinien być prostszy. To samo odnosi się do sytuacji, gdy obiekt jest trudny w użyciu: powinniśmy uprościć jego interfejs.

Co mamy jednak robić wtedy, gdy po prostu jeszcze nie wiemy, jak powinien wyglądać rozsądny projekt? Taka sytuacja ma często miejsce, gdy zaczynamy używać nowej biblioteki, integrujemy własny kod z kodem innej osoby lub zaczynamy pracować nad dużą historyjką użytkownika.

W takim przypadku możemy użyć **prototypu** (ang. *spike*), czyli krótkiej sekcji kodu, która wystarcza, aby zarysować projekt. Celem tego etapu nie jest pisanie możliwie najczystszego kodu. Nie uwzględniamy przypadków brzegowych i obsługi błędów. Mamy konkretny i ograniczony w zakresie cel w postaci eksploracji możliwych kombinacji obiektów oraz funkcji tworzących wiarygodny projekt. Gdy tylko osiągniemy cel, robimy notatki, a sam prototyp kasujemy. Teraz wiemy, jak ma wyglądać rozsądny projekt. Jesteśmy lepiej przygotowani do pisania testów. Możemy wrócić do stosowania TDD i cieszyć się wsparciem tej metody przy projektowaniu.

Co ciekawe, gdy zaczynamy pracę od początku, uzyskany projekt często jest o wiele lepszy niż prototyp. Pętla informacji zwrotnych w ramach TDD pomaga eksplorować nowe podejścia i usprawnienia.

Podsumowując: to naturalne, że kusi nas, by zaimplementować kod przed napisaniem testów. Dzięki TDD i prototypom możemy jednak usprawnić cały proces. Podejmujemy wiążące decyzje w **„ostatnim odpowiedzialnym momencie”** (ang. *last responsible moment*), czyli odkładamy to do ostatniej chwili. Dzięki temu nie jesteśmy skazani na suboptymalne rozwiązania wynikające z podejmowania przedwczesnych i nieodwracalnych decyzji. Gdy mamy wątpliwości, możemy eksplorować inne rozwiązania za pomocą **prototypu** — krótkiego, eksperymentalnego kodu, który służy do nauki i który możemy później wyrzucić.

Podsumowanie

W tym rozdziale przedstawiłem sześć częstych mitów, które przeszkadzają zespołom w przejściu na TDD. Wyjaśniłem też, w jaki sposób możemy na nie odpowiedzieć. Programowanie sterowane testami zasługuje na to, aby być dużo szerzej stosowane we współczesnym rozwoju oprogramowania niż obecnie. Nie chodzi o to, że to podejście nie działa. TDD ma po prostu problem z wizerunkiem — często wśród osób, które nigdy nie doświadczyły prawdziwej potęgi tego podejścia.

W drugiej części książki zastosujemy różne metody i techniki TDD w praktyce, pracując nad niewielką aplikacją internetową. W następnym rozdziale rozpoczniemy od poznania podstaw metodyk zwinnych i historyjek użytkownika.

Pytania i odpowiedzi

1. Dlaczego czasami uważa się, że TDD spowalnia programistów?

   Gdy nie tworzymy testów, oszczędzamy czas niezbędny do ich napisania. To podejście nie uwzględnia jednak dodatkowego czasu potrzebnego na znalezienie, odtworzenie i naprawienie defektu na produkcji.

2. Czy TDD eliminuje potrzebę ludzkiego wkładu w projektowanie?

   Nie. Wręcz przeciwnie. Nadal projektujemy kod, wykorzystując wszelkie techniki projektowe, które mamy do dyspozycji. TDD zapewnia jedynie szybkie informacje zwrotne na temat tego, czy dokonane wybory projektowe zaowocowały kodem poprawnym i prostym w użyciu.

3. Dlaczego mój zespół nie chce używać TDD?

   To fantastyczne pytanie, które należy im zadać! Naprawdę. Zobacz, czy któreś z ich zastrzeżeń nie zostały omówione w tym rozdziale. Jeżeli tak, możesz delikatnie naprowadzić rozmowę na zaprezentowane tam argumenty.

Polecane źródła

- *https://en.wikipedia.org/wiki/Characterization_test*

  Na tej stronie uzyskasz więcej informacji na temat techniki testów charakteryzacyjnych, w ramach której rejestrujemy wyniki działania istniejącego oprogramowania w aktualnej formie, co umożliwia refaktoryzację kodu bez zmieniania jego zachowania. Jest to szczególnie cenne w przypadku starszego kodu, dla którego oryginalne wymagania są niejasne lub który zmienił się z biegiem lat i zawiera części, od których uzależnione są inne systemy.

- *https://effectivesoftwaredesign.com/2014/03/27/lean-software-development-before-and-after-the-last-responsible-moment*

  Szczegółowa analiza dotycząca tego, co w projektowaniu oprogramowania oznacza wspomniany wcześniej termin „ostatni odpowiedzialny moment".

TDD: Twoja ścieżka do doskonałości w programowaniu!

Koncepcja programowania sterowanego testami oznacza tworzenie kodu wysokiej jakości. TDD (ang. *test-driven development*) uznaje testowanie za integralny element procesu tworzenia aplikacji. To proste i potężne narzędzie ułatwia także skuteczne zastosowanie wzorców projektowych. Jeśli planujesz zostać biegłym architektem oprogramowania, opanowanie TDD w praktyce jest koniecznością!

Dzięki tej książce zrozumiesz moc programowania sterowanego testami. Bazując na świetnie wyjaśnionym procesie budowy przykładowej aplikacji zgodnie z paradygmatem TDD, przyswoisz mechanizmy: cykl czerwone, zielone i refaktoryzacja, a także wzorzec przygotowania, działania i asercji. Dowiesz się też, jak za sprawą odwrócenia zależności i zamienników testowych uzyskać kontrolę nad systemami zewnętrznymi, takimi jak bazy danych. Poznasz ponadto zaawansowane techniki projektowania, w tym zasady SOLID, refaktoryzację i architekturę heksagonalną. Na podstawie piramidy testów nauczysz się znajdować równowagę między szybkimi i powtarzalnymi testami jednostkowymi a testami integracyjnymi. Używając Javy 17, opracujesz nowoczesną mikrousługę REST opartą na bazie danych PostgreSQL.

W książce między innymi:

- kodowanie przypadków testowych w Javie
- miejsce TDD w procesie tworzenia oprogramowania
- pisanie w Javie solidnego kodu wielokrotnego użytku
- rzeczywiste działanie TDD i jego skuteczność
- przebieg pracy w TDD
- refaktoryzacja a TDD

Alan Mellor pisze kod w C na potrzeby sterowania przemysłowego, w Javie i Go tworzy aplikacje internetowe dla handlu elektronicznego, branży gier i bankowości, a w C++ — oprogramowanie do magazynowania dokumentów. Współtworzył grę *Bounce* i symulator lotu dla Red Arrows, Zespołu Akrobacyjnego Królewskich Sił Powietrznych Wielkiej Brytanii.

Helion

helion.pl

HELION S.A.
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-289-2209-9

9 788328 922099

Cena: 79,00 zł

packt