Vladimir Khorikov

TESTY JEDNOSTKOWE

Zasady, praktyki i wzorce

Helion

Tytuł oryginału: Unit Testing Principles, Practices, and Patterns

Tłumaczenie: Katarzyna Bogusławska

Projekt okładki: Studio Gravite / Olsztyn; Obarek, Pokoński, Pazdrijowski, Zaprucki
Materiały graficzne na okładce zostały wykorzystane za zgodą Shutterstock Images LLC.

ISBN: 978-83-283-6871-2

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

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

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/tejeza.zip

Printed in Poland.

Spis treści

1 Cel testowania jednostkowego

Ten rozdział zawiera omówienie:

- obecnej kondycji testowania jednostkowego,
- celu testowania jednostkowego,
- konsekwencji posiadania złych zestawów testowych,
- zastosowania wskaźników pokrycia w celu mierzenia jakości zestawów testowych,
- właściwości dobrego zestawu testowego.

Nauka testowania jednostkowego nie kończy się na opanowaniu technikaliów, takich jak ulubiona struktura do testów czy biblioteka zaślepek i sterowników. Na ten aspekt zapewnienia jakości składa się znacznie więcej niż tylko pisanie testów. Musisz nieustająco dbać o jak najwyższy zwrot z inwestycji Twojego czasu, jaką jest testowanie jednostkowe, zmniejszać wysiłek niezbędny do wykonania tych testów i maksymalizować zyski z dostarczanej przez nie informacji. Sprostać tym wymaganiom nie jest łatwo!

Przyglądanie się projektom, które osiągnęły tę równowagę, jest fascynujące. Takie przedsięwzięcia rozwijają się przy niemal minimalnych nakładach pracy, ich utrzymanie nie wymaga dużo czasu i są w stanie szybko się zmieniać, by zaspokajać dynamiczne potrzeby klientów. Proporcjonalnie frustrujące jest obserwowanie projektów, którym tej równowagi brakuje. Pomimo wielkich starań i olbrzymiej liczby testów jednostkowych wloką się one pełne defektów i pochłaniają coraz większe sumy pieniędzy.

Różnice w technikach testowania jednostkowego sprowadzają się do faktu, że niektóre z tych podejść przynoszą świetne efekty i przysługują się jakości wytwarzanego oprogramowania, a wynikiem innych są testy, które niewiele wnoszą, często się psują i wymagają wiele pracy nad ich utrzymaniem.

To, czego nauczysz się z tej książki, pozwoli Ci odróżnić dobrą technikę testowania jednostkowego od złej. Dowiesz się, jak przeprowadzić analizę zysków ze swoich testów oraz ich kosztów i zastosować techniki testowania odpowiednie w danej sytuacji. Poznasz też sposoby unikania popularnych antywzorców, czyli wzorców rozwiązań, które początkowo wydają się sensowne, ale z biegiem czasu stają się utrapieniem.

Zacznijmy jednak od podstaw. Ten rozdział pokrótce przedstawia obecną kondycję testowania jednostkowego w branży programistycznej, wskazuje cel pisania i utrzymywania takich testów i dostarcza wskazówek co do skutecznych zestawów testowych.

1.1. Obecna kondycja testowania jednostkowego

W ciągu ostatnich dwóch dekad można było zaobserwować wzrost popularności testowania jednostkowego. W efekcie w licznych firmach jest to obecnie obowiązkowa część procesu wytwarzania oprogramowania. Wielu programistów stosuje testy jednostkowe i rozumie ich wagę. Dyskusja o tym, czy powinno się je przeprowadzać, została dawno i bezapelacyjnie rozstrzygnięta. O ile nie pracujesz nad projektem zabawką, werdykt brzmi: tak, powinno.

Gdy w grę wchodzi budowanie aplikacji typu *enterprise*, niemalże każdy projekt zakłada jakąś formę testowania jednostkowego. Duża część takich przedsięwzięć idzie znacznie dalej — osiąga wysokie pokrycie kodu testami dzięki wielkiej liczbie testów jednostkowych i integracyjnych. Proporcja między kodem produkcyjnym a kodem testowym może wynosić od 1:1 do nawet 1:3 (każdej linii kodu produkcyjnego odpowiada od jednej do trzech linii kodu testów). Czasami stosunek ten rośnie jeszcze bardziej, nawet do 1:10.

Jak wszystkie nowe technologie, testowanie jednostkowe się zmienia. Ciężar dyskusji przeniósł się z pytania: „Czy powinniśmy *stosować* testy jednostkowe?" na: „Co to znaczy, że piszę *dobre* testy jednostkowe?". To właśnie tutaj kryje się najwięcej wątpliwości.

Ich skutki można zaobserwować w wielu projektach związanych z oprogramowaniem. Część z nich ma testy automatyczne, być może nawet dużą ich liczbę. Mimo to istnienie takich testów nie przynosi rezultatów, na jakie liczyli programiści. Postęp w takich zespołach nadal może kosztować wiele czasu i wysiłku, funkcjonalność może być wytwarzana długo, defekty pojawiają się nieustająco w zaimplementowanych i uzgodnionych przypadkach użycia, a automatyczne testy jednostkowe w żaden sposób nie pozwalają opanować tej sytuacji. Co więcej, mogą ją pogorszyć.

Ta niekorzystna sytuacja jest skutkiem posiadania testów jednostkowych, które nie spełniają swojej funkcji. Różnica między dobrym a złym przypadkiem testowym nie jest jedynie kwestią wyczucia smaku czy preferencji. Tu waży się sukces lub porażka projektu, nad którym pracujesz.

Trudno przecenić wagę rozważań na temat tego, co stanowi o jakości testu jednostkowego. Mimo to takie rozmowy często nie są przeprowadzane w firmach wytwarzających oprogramowanie. W sieci można znaleźć kilka artykułów i nagrań z wystąpień podczas różnych konferencji, ale ze świecą szukać całościowego i wyczerpującego opracowania na ten temat.

Sytuacja na rynku książkowym nie jest znacznie lepsza. Wiele pozycji koncentruje się na podstawach testowania jednostkowego i nie wychodzi poza ten obszar. Takie źródła

mają oczywiście swoją wartość, zwłaszcza gdy to Twoje pierwsze kroki w tej dziedzinie, ale nauka nie kończy się na zapoznaniu się z nimi. Następny etap obejmuje bowiem nie tylko pisanie testów jednostkowych, ale też robienie tego w taki sposób, by uzyskać największą korzyść z włożonego w to czasu i wysiłku. Kiedy stajesz przed problemem dotyczącym efektywności i rozległości zbioru testowego, często jesteś pozostawiony sam sobie.

Niniejsza książka ma Ci przyjść z pomocą w takiej sytuacji. Przedstawia ona naukową, precyzyjną definicję idealnego testu jednostkowego. Zobaczysz, jak ją zastosować do praktycznych, rzeczywistych przykładów. Mam nadzieję, że pomogę Ci zrozumieć, dlaczego niektóre projekty zeszły na manowce pomimo sporej liczby testów, a także jak poprawić trajektorię takich przedsięwzięć.

Najbardziej skorzystasz na lekturze tej książki, jeśli zajmujesz się rozwijaniem aplikacji typu *enterprise*, ale podstawowe pomysły są wspólne dla wszystkich projektów programistycznych.

Czym jest aplikacja typu enterprise?

Aplikacja typu *enterprise* to aplikacja, której celem jest wsparcie lub automatyzacja wewnętrznych procesów organizacji. Może ona przyjąć różne formy, ale oprogramowanie tego typu często charakteryzuje się:

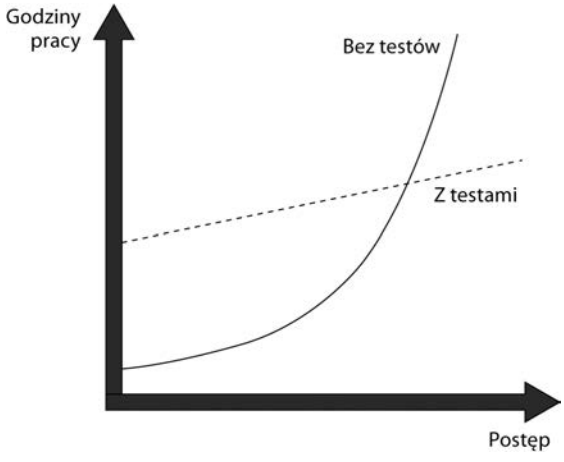
- wysokim poziomem skomplikowania logiki biznesowej,
- długim cyklem życia oprogramowania,
- umiarkowaną ilością danych,
- niskimi lub średnimi wymaganiami związanymi w wydajnością.

1.2. Cel testowania jednostkowego

Zanim zagłębię się w temat testowania jednostkowego, chciałbym zrobić krok wstecz i zastanowić się nad celem, jaki test jednostkowy ma pomóc osiągnąć. Często się mówi, że praktyki związane z testowaniem jednostkowym wpływają na poprawę projektu rozwiązania. To prawda: konieczność pisania testów jednostkowych do kodu w normalnych okolicznościach prowadzi do ulepszenia projektu oprogramowania. Nie jest to jednak główny cel tej aktywności, lecz jedynie korzystnie skorelowany efekt.

Jaki jest zatem cel testowania jednostkowego? Celem jest umożliwienie zrównoważonego rozwoju projektu opartego na wytwarzaniu oprogramowania. Kluczowe jest tutaj słowo „zrównoważony". Dość łatwo jest rozbudować projekt, zwłaszcza gdy zaczyna się go od samego początku. Znacznie trudniej jest utrzymać tempo i zakres rozbudowywania w czasie.

Rysunek 1.1 przedstawia dynamikę rozwoju typowego projektu pozbawionego testów jednostkowych. Początek jest żwawy, ponieważ nic Cię nie obciąża i nie spowalnia. Nie podjęto jeszcze żadnych złych decyzji co do architektury i nie ma kodu, o który należałoby się martwić. Jednak w miarę upływu czasu rośnie liczba godzin, jakie musisz poświęcić, by wykazywać to samo tempo postępu co początkowo. W końcu prędkość rozwijania oprogramowania znacząco spada, a w skrajnych przypadkach postęp w ogóle przestaje być możliwy.



Rysunek 1.1. Różnica dynamiki rozwoju pomiędzy projektami z testami jednostkowymi i bez. Projekt pozbawiony testów jednostkowych rozpoczyna się w wysokim tempie, ale wkrótce zwalnia tak, że jakikolwiek postęp przestaje być możliwy

Związek między testowaniem jednostkowym a projektowaniem kodu

Możliwość poddania kodu testom jednostkowym jest wygodnym papierkiem lakmusowym, ale sprawdza się tylko w jedną stronę — jej brak wskazuje ze względnie dużą dokładnością kod niskiej jakości. Jeśli znajdziesz fragment kodu trudny do przetestowania w ten sposób, będzie to dla Ciebie istotna wskazówka, że wymaga on ulepszenia. Niska jakość kodu najczęściej objawia się w tzw. wysokim sprzężeniu (ang. *tight coupling*), które oznacza, że różne fragmenty produkcyjnego kodu nie są wystarczająco uniezależnione od pozostałych, co utrudnia ich oddzielne testowanie.

Niestety, możliwość testowania jednostkowego nie jest rzetelnym wskaźnikiem poprawności kodu. To, że łatwo poddaje się go testom jednostkowym, nie oznacza, że jest wysokiej jakości. Projekt może być porażką, nawet jeśli rozwiązanie cechuje niskie sprzężenie.

Zjawisko szybko spadającego tempa pracy znane jest także jako **entropia oprogramowania**. Entropia (miara nieporządku w systemie) to pojęcie matematyczne i naukowe, stosowane także w technologii informatycznej (jeśli interesujesz się matematyką i nauką o entropii, zacznij od analizy drugiej zasady termodynamiki).

W wytwarzaniu oprogramowania entropia przejawia się w kodzie, który niszczeje. Za każdym razem, gdy zmieniasz coś w kodzie, ilość nieporządku (czyli entropia) rośnie. Jeśli pozostawisz system bez odpowiedniej opieki w postaci ciągłego czyszczenia i refaktoryzowania, będzie coraz bardziej złożony i coraz gorzej uporządkowany. Naprawienie jednego defektu doprowadzi do powstania dwóch innych, a zmiana w jednej części popsuje działanie trzech innych itd., zgodnie z efektem domina. Doprowadzi to do sytuacji, w której na kodzie nie będzie można polegać. A co najgorsze, niezwykle trudno będzie przywrócić jego stabilność.

Testy mogą pomóc zaradzić takiej sytuacji. Odgrywają rolę sieci bezpieczeństwa — narzędzia, które zapewnia ochronę przed większością defektów regresyjnych. Testy pozwalają się upewnić, że istniejące funkcjonalności są sprawne nawet po wprowadzeniu zmian czy refaktoryzacji kodu, która miała na celu lepsze dopasowanie do wymagań.

DEFINICJA Regresja ma miejsce, gdy funkcjonalność przestaje działać zgodnie z zamierzeniami po pewnym zdarzeniu (zwykle: po modyfikacji kodu). Pojęcia „regresja" i „defekt oprogramowania" w moich rozważaniach są synonimami i używam ich wymiennie.

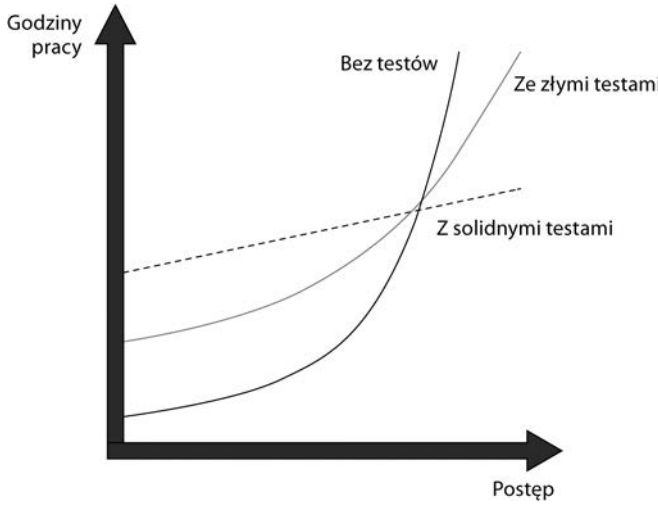
Testy wymagają jednak nieraz znacznego wysiłku przygotowawczego. W dłuższej perspektywie zwraca się on jednak, ponieważ pozwala projektowi stabilnie rosnąć na późniejszych etapach. Wytwarzanie oprogramowania bez wsparcia w postaci testów, które nieustająco weryfikują całość kodu, jest rozwiązaniem, które zupełnie się nie skaluje.

Zrównoważony i skalowalny rozwój jest kluczowy. To dzięki tym dwóm cechom będziesz w stanie utrzymać tempo pracy nad oprogramowaniem na dłuższą metę.

1.2.1. Co czyni test dobrym?

Chociaż testy jednostkowe pomagają utrzymać rozwój projektu, nie wystarczy po prostu pisać testy. Kiepsko napisane przypadki testowe doprowadzą do takiej samej ruiny.

Jak pokazuje rysunek 1.2, złe testy pozwalają początkowo zatrzymać pogorszenie jakości kodu. Spadek tempa pracy jest mniejszy niż w sytuacji, gdy testów nie ma wcale. Nic się jednak nie zmienia, gdy spojrzy się na to całościowo. Może minąć więcej czasu, zanim taki projekt wejdzie w fazę stagnacji, ale ten moment i tak jest nieunikniony.



Rysunek 1.2. Różnica dynamiki rozwoju pomiędzy projektami z dobrymi i złymi testami jednostkowymi. Projekt ze źle napisanymi przypadkami testowymi początkowo wykazuje te same właściwości co projekt o solidnych testach, ale stagnacja jest w nim nieunikniona

Pamiętaj, *nie wszystkie testy zostały stworzone równymi*. Niektóre są wartościowe i przysługują się jakości oprogramowania. Inne nie. Alarmują wynikami fałszywie negatywnymi, nie pomagają w wychwyceniu regresji, są wolne i trudne w utrzymaniu. Łatwo wpaść w pułapkę pisania testów jednostkowych po to, by po prostu mieć testy jednostkowe, bez refleksji nad tym, czy wnoszą one cokolwiek dobrego do projektu.

Nie uda Ci się osiągnąć celu testowania jednostkowego przez proste dodawanie przypadków testowych. Musisz mieć na uwadze zarówno wartość testu, jak i koszt jego utrzymania. Koszt ten określa ilość czasu potrzebną do:

- zaadaptowania kodu testu, gdy zmienia się pokryty przez niego produkcyjny kod,
- wykonania testu po każdej zmianie kodu,
- odniesienia się do fałszywych alarmów spowodowanych wykonaniem takiego testu,
- czytania i analizy testu w celu zrozumienia, jak działa kod, który podlega temu testowi.

Łatwo tworzyć testy, których wartość po odjęciu kosztów jest bliska zera albo wręcz ujemna z powodu wysokich nakładów na utrzymanie. Aby umożliwić zrównoważony rozwój projektu, musisz skupić się wyłącznie na testach wysokiej jakości. Tylko takie przypadki opłaca się mieć w zestawach testowych.

Kod produkcyjny a kod testowy

Ludziom często się wydaje, że kod produkcyjny i kod testowy to dwa różne światy. Panuje przekonanie, że testy stanowią dodatek do kodu produkcyjnego i nie pociągają za sobą żadnych kosztów własności. Stąd prosty wniosek, że im więcej testów, tym lepiej. To nieprawda. *Kod to obciążenie, a nie zasób!* Im więcej kodu tworzysz, tym więcej dajesz sobie miejsca do popełnienia błędu i wprowadzenia błędu w oprogramowaniu, tym bardziej rosną też koszty utrzymania. Zawsze lepiej rozwiązywać problemy jak najmniejszą ilością kodu.

Testy to też kod. Powinieneś traktować testy jako część swojego kodu, która odpowiada na konkretny problem, tj. konieczność zapewnienia poprawności funkcjonowania aplikacji. Testy jednostkowe, jak każdy inny kod, są podatne na defekty i wymagają utrzymania.

Stąd wynika, że kluczowe jest rozróżnianie pomiędzy dobrym a złym przypadkiem testowym, do czego wrócę w rozdziale 4.

1.3. Stosowanie wskaźników pokrycia do mierzenia jakości zestawu testowego

W tej sekcji zajmę się dwoma najbardziej popularnymi wskaźnikami pokrycia — pokryciem kodu i pokryciem gałęzi — oraz sposobami ich wyliczania i stosowania, a także możliwymi problemami. Wskażę, dlaczego ustalanie procentowego celu pokrycia może mieć zły wpływ na programistów, i przestrzegę przed poleganiem w ocenie jakości zestawu testowego wyłącznie na wskaźnikach pokrycia.

DEFINICJA Wskaźnik pokrycia przedstawia procentowo (od 0 do 100%), jaka część kodu źródłowego została wykonana w przebiegu testu.

Istnieją różne rodzaje wskaźników pokrycia i często wykorzystuje się je do oceny jakości zestawu testowego. Powszechnie uważa się, że im wyższy procent pokrycia, tym lepiej.

Niestety, sprawa nie jest tak prosta, a wskaźniki pokrycia, choć dostarczają cennych uwag, nie mogą wydajnie mierzyć *jakości* zbioru testowego. Tak samo jak w przypadku samej testowalności kodu za pomocą testów jednostkowych — pokrycie jest dobrą miarą tego, jak zły jest kod, ale kiepską miarą tego, jak jest on dobry.

Jeśli wskaźnik mówi, że pokrycie Twojego kodu jest niskie — załóżmy, że wynosi zaledwie 10% — to jest to dla Ciebie znak, że nie testujesz wystarczająco dużo. Niestety, odwrotność tego rozumowania nie jest poprawna — nawet 100% pokrycia nie daje gwarancji, że Twoje zestawy testowe są wysokiej jakości. Zestaw testowy zapewniający wysokie pokrycie nadal może być kiepskiej jakości.

Zwróciłem już na ten aspekt uwagę wcześniej — jeśli dodasz do swojego projektu przypadkowy test, nie możesz mieć nadziei, że poprawi on sytuację. Zajmijmy się jednak tym problemem szczegółowo w kontekście wskaźnika pokrycia kodu.

1.3.1. Interpretacja wskaźnika pokrycia kodu

Pierwszym i najczęściej używanym wskaźnikiem pokrycia jest **pokrycie kodu**, znane też jako **pokrycie testowe**, co widać na rysunku 1.3. Wskaźnik ten przedstawia stosunek linii kodu wykonanych przez co najmniej jeden test do łącznej liczby linii kodu produkcyjnego.

$$\text{Pokrycie kodu (pokrycie testowe)} = \frac{\text{Liczba wykonanych linii kodu}}{\text{Łączna liczba linii}}$$

Rysunek 1.3. Pokrycie kodu (pokrycie testowe) wylicza się jako stosunek linii kodu wykonanych w przebiegu zestawu testowego do łącznej liczby linii kodu produkcyjnego

Przyjrzyjmy się przykładowi, byś lepiej zrozumiał tę definicję. Listing 1.1 przedstawia metodę `IsStringLong` oraz test, który ją pokrywa. Metoda określa, czy przekazany do niej tekst jest długi (w tym przypadku za *długi* uznaje się tekst składający się z ponad pięciu znaków). Test ten wykonuje metodę z parametrem `"abc"` i sprawdza, czy taki tekst nie został uznany za długi.

Listing 1.1. Przykładowa metoda częściowo pokryta przez test

```
public static bool IsStringLong(string input)
{
    if (input.Length > 5)
        return true;
    return false;
}

public void Test()
{
    bool result = IsStringLong("abc");
    Assert.Equal(false, result);
}
```

Niepokryte testem

Pokryte testem

Łatwo tutaj obliczyć pokrycie. Łączna liczba linii w metodzie wynosi pięć (nawiasy klamrowe też się liczą). Liczba linii wykonanych przez test to cztery — test przechodzi przez wszystkie linie z wyjątkiem instrukcji `return true`. Daje nam to 4/5 = 0,8 = 80% pokrycia kodu.

Co by się stało, gdybym zastąpił niepotrzebną instrukcję warunkową `if` w ten sposób?

```
public static bool IsStringLong(string input)
{
    return input.Length > 5;
}

public void Test()
{
    bool result = IsStringLong("abc");
    Assert.Equal(false, result);
}
```

Czy zmieni się procent pokrycia? Tak, zmieni się, ponieważ test wykonuje teraz wszystkie trzy linie kodu (instrukcję `return` oraz dwa nawiasy klamrowe). Pokrycie kodu wzrośnie do 100%.

Ale czy ulepszyłem zestaw testowy dzięki tej modyfikacji? Oczywiście nie. Jedynie poprzestawiałem kod wewnątrz metody. Test nadal weryfikuje tę samą liczbę możliwych wyników.

Ten prosty przykład pokazuje, jak łatwo jest manipulować wartościami pokrycia. Im bardziej zwięzły jest Twój kod, tym lepiej wypadną wskaźniki pokrycia kodu, ponieważ weryfikacji podlega tylko liczba linii. Kondensowanie większej ilości kodu w mniejszej liczbie linii nie zmienia (i nie powinno zmieniać) wartości zestawu testowego czy utrzymywalności kodu produkcyjnego.

1.3.2. Interpretacja wskaźnika pokrycia gałęzi

Inny wskaźnik nosi nazwę **pokrycia gałęzi**. Pokrycie gałęzi dostarcza dokładniejszej informacji niż pokrycie kodu, ponieważ radzi sobie z niedostatkami wskaźnika pokrycia kodu. Zamiast brać pod uwagę wyłącznie liczbę linii, wskaźnik ten analizuje instrukcje warunkowe i przepływ kontroli, np. w instrukcji `if` lub `switch`. Przedstawia on, jak wiele z tych struktur trawersował choć jeden test w zestawie, co widać na rysunku 1.4.

$$\text{Pokrycie gałęzi} = \frac{\text{Trawersowane gałęzie}}{\text{Łączna liczba gałęzi}}$$

Rysunek 1.4. Pokrycie gałęzi wylicza się jako stosunek gałęzi kodu przetrawersowanych w przebiegu zestawu testowego do łącznej liczby gałęzi kodu produkcyjnego

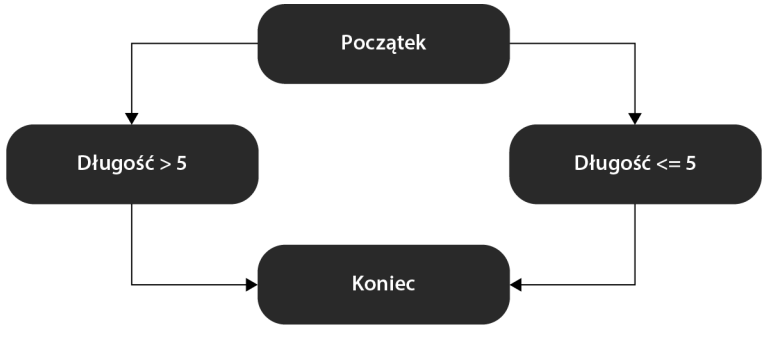
Aby wyliczyć wskaźnik pokrycia gałęzi, musisz zsumować wszystkie możliwe gałęzie w swoim kodzie i określić, przez ile z nich przeszły Twoje testy. Spójrzmy raz jeszcze na poprzedni przykład:

```
public static bool IsStringLong(string input)
{
    return input.Length > 5;
}

public void Test()
{
    bool result = IsStringLong("abc");
    Assert.Equal(false, result);
}
```

W metodzie są dwie gałęzie: jedna dotyczy sytuacji, gdy długość argumentu tekstowego jest większa niż pięć znaków, a druga — gdy długość nie przekracza tej wartości. Test pokrywa tylko jedną z nich, więc wskaźnik pokrycia gałęzi wynosi: 1/2 = 0,5 = 50%. Nie ma też znaczenia, jak ustrukturyzujesz kod poddawany testom — niezależnie od tego, czy zastosujesz instrukcję warunkową `if`, czy skorzystasz z krótszego zapisu, rezultat będzie taki sam. Wskaźnik pokrycia gałęzi bierze pod uwagę wyłącznie liczbę gałęzi, a zupełnie pomija liczbę linii, jaka była niezbędna, by je zaimplementować.

Rysunek 1.5 poniżej stanowi pomocne zobrazowanie tego wskaźnika. Możesz przedstawić możliwe ścieżki, jakie obierze kod poddawany testom, jako graf. Następnie będziesz w stanie określić, ile z nich zostało wykonanych w ramach testów. Metoda `IsStringLong` ma dwie takie ścieżki, ale test pokrywa tylko jedną z nich.



Rysunek 1.5. Metoda IsStringLong przedstawiona jako graf możliwych ścieżek kodu. Test pokrywa tylko jedną z nich, stąd pokrycie gałęzi wynosi 50%

1.3.3. Problemy z pokryciem gałęzi

Mimo że wskaźnik pokrycia gałęzi daje lepsze rezultaty niż wskaźnik pokrycia kodu, przy określaniu jakości zestawu testowego nie można polegać wyłącznie na nim. Są dwa powody:

- Nie można zagwarantować, że test weryfikuje wszystkie możliwe wartości wyjściowe z systemu poddawanego testom.
- Żaden wskaźnik pokrycia nie bierze pod uwagę kodu w zewnętrznych bibliotekach.

Przyjrzyjmy się bliżej każdej z tych kwestii.

BRAK GWARANCJI WERYFIKACJI WSZYSTKICH MOŻLIWYCH WARTOŚCI WYJŚCIOWYCH

Aby ścieżki kodu zostały przetestowane, a nie tylko wykonane, testy jednostkowe muszą dokonywać odpowiednich asercji. Innymi słowy, należy sprawdzić, czy wartość wynikowa z systemu poddawanego testom zgadza się z tą oczekiwaną na wyjściu z systemu. Co więcej, ta wartość wyjściowa może mieć kilka komponentów, a żeby wskaźnik pokrycia był rzetelny, musisz weryfikować wszystkie te elementy.

Następny przykład przedstawia inną wersję metody `IsStringLong`. Zapisuje ona ostatni wynik w publicznej właściwości `WasLastStringLong`.

Listing 1.2. Wersja metody IsStringLong, która zachowuje ostatni rezultat

```
public static bool WasStringLong { get; private set; }

public static bool IsStringLong(string input)
{
    bool result = input.Length > 5;
    WasStringLong = result;
    return result;
}

public void Test()
{
    bool result = IsStringLong("abc");
    Assert.Equal(false, result);
}
```

Pierwsza wartość wynikowa

Druga wartość wynikowa

Test weryfikuje tylko drugą wartość wynikową

Metoda `IsStringLong` ma w tej chwili dwie wartości wyjściowe: jawną, zakodowaną jako wartość zwracana, oraz niejawną, którą jest wartość właściwości. Mimo że test nie weryfikuje tej niejawnej wartości wyjściowej, wskaźniki pokrycia nadal przedstawiałyby te same rezultaty: 100% pokrycia kodu i 50% pokrycia gałęzi. Jak widać, wskaźniki pokrycia nie zapewniają, że kod został przetestowany, lecz jedynie to, że został w pewnym momencie wykonany.

Skrajnym przykładem takiej sytuacji z częściowo przetestowanymi wynikami jest testowanie pozbawione asercji (ang. *assertion-free testing*), które polega na pisaniu testów pozbawionych asercji. Poniżej przedstawiam próbkę takiego testowania.

Listing 1.3. Test pozbawiony asercji zawsze zwraca pozytywny wynik

```
public void Test()
{
    bool result1 = IsStringLong("abc");
    bool result2 = IsStringLong("abcdef");
}
```

Zwraca wartość prawdziwą: true

Zwraca wartość fałszywą: false

Powyższy test cechuje pokrycie kodu oraz gałęzi wynoszące 100%. Niestety, mimo to jest zupełnie bezużyteczny, ponieważ nie weryfikuje zupełnie niczego.

Załóżmy jednak, że szczegółowo weryfikujesz każdą wartość wynikową w swoich testach. Czy to w połączeniu ze wskaźnikiem pokrycia gałęzi stanowi rzetelny mechanizm do oceny jakości Twoich zestawów testowych? Niestety nie.

ŻADEN WSKAŹNIK POKRYCIA NIE WERYFIKUJE ŚCIEŻEK W ZEWNĘTRZNYCH BIBLIOTEKACH

Drugi problem ze wskaźnikami pokrycia polega na tym, że nie obejmują one ścieżek kodu w zewnętrznych bibliotekach, które są wykonywane, gdy testowany system wywołuje swoje metody. Przyjrzyjmy się poniższemu przykładowi:

Historia z frontu

Pomysł testowania pozbawionego asercji może wydawać się niemądry, występuje jednak w naturze.

Lata temu pracowałem w projekcie, na który zarząd nałożył bezapelacyjny wymóg 100% pokrycia kodu dla każdego realizowanego przedsięwzięcia. Inicjatywa ta wyrosła ze szlachetnych pobudek. Działo się to w czasach, gdy testowanie jednostkowe nie było tak popularne jak dziś. Niewiele osób w tamtej organizacji pisało testy jednostkowe, a jeszcze mniej robiło to w sposób spójny.

Zespół programistów uczestniczył w konferencji, podczas której wiele wystąpień było poświęconych testowaniu jednostkowemu. Po powrocie członkowie zespołu postanowili wcielić nowo zdobytą wiedzę w czyn. Kierownictwo wsparło ich w tych wysiłkach, i tak rozpoczęła się transformacja ku lepszym technikom programowania. Przygotowano wewnętrzne prezentacje. Zainstalowano nowe narzędzia. Co więcej, w całej firmie wdrożono nową zasadę: wszystkie zespoły programistyczne mają skupić się wyłącznie na pisaniu testów jednostkowych aż do czasu, gdy uzyskają 100% pokrycia kodu. Po osiągnięciu takiej wartości wskaźnika każde dodanie kodu, które obniżałoby jego wartość, miało zostać cofnięte przez automatyczny system budowania kodu.

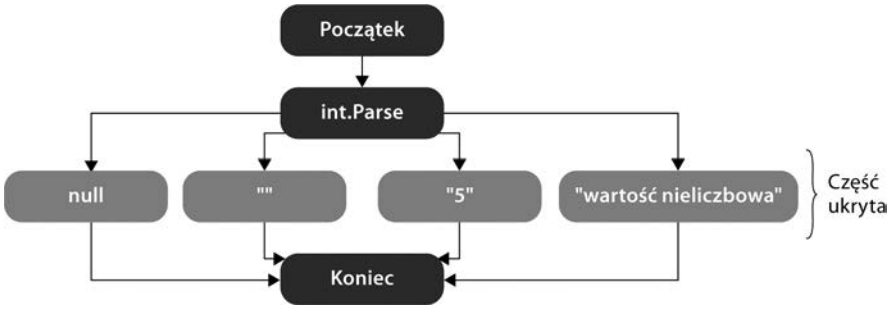
Jak możesz przypuszczać, nie wynikło z tego nic dobrego. Przytłoczeni takimi ograniczeniami programiści zaczęli szukać sposobów na przechytrzenie systemu. Wielu z nich szybko doszło do tego samego wniosku: jeśli obudujesz wszystkie swoje testy w bloki `try-catch` i nie wskażesz w nich żadnej asercji, takie testy zawsze będą zwracały pozytywny wynik! Zaczęto zatem bezmyślnie tworzyć testy tylko po to, by sprostać wymaganiu 100% pokrycia kodu. Nie muszę chyba mówić, że żaden z tych testów nie wniósł nic do projektu. Co więcej, pomysł ten szkodził projektowi, ponieważ deweloperzy marnowali swój czas i wysiłek na nieproduktywne działania, a koszty związane z utrzymaniem poszerzającej się bazy testów rosły.

W końcu wymaganą wartość pokrycia kodu obniżono do 90%, potem do 80%, a po pewnym czasie zupełnie zniesiono (i to z pożytkiem!).

```
public static int Parse(string input)
{
    return int.Parse(input);
}

public void Test()
{
    int result = Parse("5");
    Assert.Equal(5, result);
}
```

Wskaźnik pokrycia gałęzi wynosi 100%, a test weryfikuje wszystkie elementy wartości wyjściowej testowanej metody. Taki element jest zresztą tylko jeden — wartość zwracana. Mimo to test wcale nie jest wyczerpujący. Zupełnie pomija on ścieżki kodu w metodzie `int.Parse` z platformy .NET Framework, które mogą zostać wykonane w ramach wykonania powyższego przykładowego kodu. Takich ścieżek jest całkiem sporo nawet w tak prostej metodzie, o czym możesz się przekonać, spoglądając na rysunek 1.6 poniżej.



Rysunek 1.6. Ukryte ścieżki kodu zewnętrznej biblioteki. Wskaźniki pokrycia nie są w stanie ocenić, ile takich ścieżek jest i jak wiele z nich zostało wykonanych w wyniku Twoich testów

Typ wbudowany `integer` ma wiele gałęzi ukrytych przed testem, które mogą prowadzić do różnych wyników w zależności od parametru wejściowego. Oto kilka możliwych argumentów, które nie zostaną zamienione na liczbę całkowitą:

- wartość `null`,
- pusty łańcuch znaków,
- „wartość nieliczbowa”,
- zbyt długi łańcuch znaków.

Możesz się natknąć na wiele przypadków spornych, ale nie masz możliwości upewnienia się, że Twój test pokryje je wszystkie.

Nie znaczy to jednak, że wskaźniki pokrycia *powinny* brać pod uwagę ścieżki kodu w zewnętrznych bibliotekach (nie powinny!); chcę tylko zaznaczyć, że nie można polegać wyłącznie na takich miarach w ocenie tego, czy testy jednostkowe są dobre czy złe. Wskaźniki pokrycia nie powiedzą Ci, czy Twoje testy są wyczerpujące ani czy masz ich wystarczająco dużo.

1.3.4. Wymaganie procentowej wartości pokrycia

Mam nadzieję, że do tej pory już się zorientowałeś, że poleganie na wskaźnikach wydajności w celu określenia jakości zestawu testowego nie wystarczy. Na manowce może Cię wyprowadzić także wyznaczenie sobie celu uzyskania określonej procentowej wartości pokrycia, niezależnie od tego, czy będzie to 100, 90 czy zaledwie 70%. Wskaźniki pokrycia najlepiej traktować jako wskazówki, a nie cele same w sobie.

Weźmy za przykład pacjenta w szpitalu. Jego wysoka temperatura może wskazywać na gorączkę i stanowi pomocną obserwację. Szpital nie powinien jednak dążyć za wszelką cenę i wszystkimi możliwymi sposobami do ustabilizowania temperatury pacjenta na określonym poziomie. Gdyby było inaczej, szpital mógłby pokusić się o szybkie i „efektywne” rozwiązanie w postaci zamontowania klimatyzatora obok łóżka pacjenta i sterowanie jego temperaturą przez regulowanie ilości zimnego powietrza płynącego na jego ciało. Oczywiście takie podejście jest zgubne.

Podobnie dążenie do ustalonej wartości wskaźników pokrycia może być zachętą do działań sprzecznych z celem testowania jednostkowego. Zamiast skupić się na testowaniu istotnych części oprogramowania, zespół może zacząć szukać wyłącznie sposobów

na sprostanie sztucznemu wymaganiu. Właściwe testowanie jednostkowe jest już trudne samo w sobie. Narzucanie poziomu pokrycia odwraca jedynie uwagę programistów od tego, co testują, i dodatkowo utrudnia poprawne testowanie jednostkowe.

WSKAZÓWKA Dobrze jest osiągać wysoki wskaźnik pokrycia w kluczowych częściach systemu. Źle jest robić z tego poziomu bezwarunkowe wymaganie. Różnica jest tu subtelna, ale niezwykle ważna.

Pozwól, że jeszcze raz przypomnę: miara pokrycia jest dobrym wskaźnikiem kłopotów, ale kiepskim potwierdzeniem wysokiej jakości. Niska wartość pokrycia — na poziomie 60% — to pewny znak, że można się spodziewać problemów. Wynika z niej, że duża część Twojego kodu w ogóle nie jest testowana. Ale wysoka wartość pokrycia nie znaczy zupełnie nic. Mierzenie wartości pokrycia kodu powinno stanowić zaledwie pierwszy krok do uzyskania zestawu testowego wysokiej próby.

1.4. Właściwości dobrego zestawu testowego

Poświęciłem większość tego rozdziału na opisanie niewłaściwych sposobów mierzenia jakości zestawu testowego, czyli wykorzystania wyłącznie wskaźników pokrycia. A co z tą właściwą drogą? Jak powinieneś mierzyć jakość swojego zestawu testowego? Jedynym wartościowym sposobem jest analizowanie każdego przypadku testowego po kolei i pojedynczo. Oczywiście, nie musisz oceniać ich wszystkich na raz. Takie przedsięwzięcie mogłoby być ogromne i pochłonąć mnóstwo czasu i przygotowań. Ocenę możesz przeprowadzać stopniowo. Pragnę tutaj podkreślić, że nie ma zautomatyzowanych sposobów określania, jak dobry jest zestaw testowy. Musisz kierować się własnym rozsądkiem.

Spójrzmy jednak na szerszą perspektywę tego, co sprawia, że zestaw testowy jako całość jest efektywny (w rozdziale 4. zajmę się szczegółami rozróżnienia między dobrym a złym przypadkiem testowym). Dobry zestaw testowy:

- jest zintegrowany z cyklem wytwarzania oprogramowania,
- koncentruje się na najważniejszych częściach bazy kodu,
- dostarcza jak największej wartości przy jak najniższych kosztach utrzymania.

1.4.1. Integracja z cyklem wytwarzania oprogramowania

Cały sens posiadania testów automatycznych jest w tym, że cały czas z nich korzystasz. Wszystkie testy powinny łączyć się z cyklem wytwarzania oprogramowania. Byłoby idealnie, gdybyś uruchamiał je wszystkie przy każdej zmianie w kodzie, nawet tej najmniejszej.

1.4.2. Koncentracja na najważniejszych częściach kodu

Nie wszystkie testy zostały stworzone równymi i nie wszystkie części Twojego kodu produkcyjnego wymagają tej samej uwagi, jeśli chodzi o testowanie jednostkowe. Wartość testów płynie nie tylko z tego, jak są skonstruowane, ale także z tego, jaką część kodu weryfikują.

Ważna jest umiejętność wkładania największego wysiłku w testowanie jednostkowe najbardziej istotnych obszarów i sprawdzanie pozostałych powierzchownie lub pośrednio. W większości aplikacji najważniejszą część stanowi ta, które obejmuje logikę biznesową, czyli *model domeny*[1]. Testowanie logiki biznesowej daje Ci największy zwrot z zainwestowanego czasu.

Pozostałe obszary aplikacji można podzielić na trzy części:

- kod dotyczący infrastruktury,
- usługi zewnętrzne i zależności, takie jak bazy danych i zewnętrzne systemy do zintegrowania,
- kod, który to wszystko łączy.

Niektóre z tych części mogą jednak nadal wymagać solidnego testowania jednostkowego. Przykładowo kod dotyczący infrastruktury może obejmować złożone i ważne algorytmy, więc rozsądne wydaje się pokrycie go sporą liczbą testów. Mimo to, uogólniając, większość uwagi powinieneś kierować ku modelowi domeny.

Niektóre z Twoich testów, np. testy integracyjne, będą wykraczały poza model domeny i weryfikowały, jak system działa jako całość wraz z tymi częściami aplikacji, których waga nie jest krytyczna. Tak powinno być. Niemniej priorytetem pozostaje model domeny.

Miej na uwadze, że aby testować zgodnie z tą zasadą, musisz dobrze oddzielić model domeny od pozostałych, niekluczowych części kodu źródłowego. Należy wyizolować model tak, by precyzyjnie ująć go w swoich wyczerpujących testach. Szczegółowe wskazówki w tym zakresie podaję w rozdziale 2.

1.4.3. Maksymalna wartość przy minimalnych kosztach

Najtrudniejszą częścią testowania jednostkowego jest uzyskiwanie jak największej wartości przy jak najniższych kosztach utrzymania. Zagadnienie to stanowi główny temat tej książki.

Nie wystarczy włączyć testy do systemu budowania kodu. Nie wystarczy utrzymywać wysokie pokrycie modelu domeny. Równie istotne jest zachowywanie w zestawie testowym tylko tych przypadków, których wartość znacznie przewyższa koszt ich utrzymania.

Na tę drugą cechę składają się dwa elementy:

- identyfikacja wartościowego przypadku testowego (i, co z tego wynika — bezwartościowego),
- napisanie wartościowego przypadku testowego.

Chociaż umiejętności te wydają się podobne, różnią się w swojej naturze. Aby zidentyfikować wartościowy test, potrzebujesz punktu odniesienia. Z drugiej strony, aby *napisać* wartościowy test, musisz znać także techniki projektowania kodu. Testy jednost-

[1] Patrz Eric Evans, *Domain-Driven Design. Zapanuj nad złożonym systemem informatycznym*, Helion, Gliwice 2015.

kowe oraz poddawany im kod są ze sobą blisko związane. Nie sposób napisać wartościowego przypadku testowego, nie przyłożywszy się do napisania porządnego kodu źródłowego, który testy mają pokryć.

Tę różnicę można sobie wyobrazić jako różnicę między zdolnością rozpoznania, że piosenka jest dobra, a umiejętnością skomponowania dobrego utworu. Wysiłek, który trzeba włożyć, by zostać kompozytorem, jest nieporównanie większy od wysiłku niezbędnego do tego, by umieć rozróżnić dobrą i złą muzykę. Zasada ta sprawdza się też w testach jednostkowych. Napisanie dobrego testu wymaga więcej pracy niż analiza istniejącego, głównie dlatego, że przeważnie nie piszesz testów w próżni: musisz mieć na uwadze kod źródłowy. W związku z tym, choć skupiam się tutaj na testach jednostkowych, poświęcam w tej książce sporo miejsca rozważaniom dotyczącym projektowania kodu.

1.5. Czego nauczysz się z tej książki

Niniejsza książka daje punkt odniesienia, dzięki któremu będziesz w stanie analizować testy w swoich zestawach testowych. Ten punkt odniesienia ma być fundamentem, dzięki któremu spojrzysz na własne testy w nowym świetle i ocenisz, które z nich stanowią zysk dla projektu, a które należy przerobić lub wyrzucić.

Po zbudowaniu tych podstaw (w rozdziale 4.) przechodzę do analizy istniejących technik i praktyk testowania jednostkowego (w rozdziałach 4. – 6. oraz w części rozdziału 7.). Nie ma znaczenia, czy znasz te techniki i praktyki. Jeśli już o nich słyszałeś, spojrzysz na nie pod nowym kątem. Najprawdopodobniej intuicyjnie je *czujesz*. Ta książka pozwoli Ci dokładnie zrozumieć powody, dla których techniki i praktyki, które stosujesz od lat, są tak pomocne.

Nie ignoruj tej umiejętności! Zdolność precyzyjnego zakomunikowania swoich pomysłów współpracownikom jest nie do przecenienia. Programiście — nawet genialnemu — rzadko są przypisywane wszystkie należne zasługi związane z decyzją dotyczącą projektu oprogramowania, jeśli nie może jasno przedstawić, dlaczego ta decyzja została podjęta. W tej publikacji mam zamiar pomóc Ci przenieść Twoją argumentację z obszaru podświadomych przeczuć do tematów, o których możesz porozmawiać z każdym.

Jeśli nie masz doświadczenia z technikami i dobrymi praktykami testowania jednostkowego, wiele możesz się z tej książki nauczyć. Oprócz tego, że zyskasz punkt odniesienia do analizy swoich zestawów testowych, dowiesz się:

- jak zmieniać i poprawiać zestawy testowe wraz z kodem źródłowym, na którym się opierają,
- jak stosować różne techniki testowania jednostkowego,
- jak używać testów integracyjnych, by weryfikować funkcjonowanie systemu jako całości,
- jak zauważać antywzorce i ich unikać.

Poza testami jednostkowymi poruszam tu bardziej ogólny temat testów automatycznych, więc poznasz też testy integracyjne i systemowe.

W moich przykładach kodu używam C# i platformy .NET, ale nie musisz zawodowo posługiwać się tym językiem, by czytać tę książkę. C# jest po prostu językiem, w którym najwięcej pracuję. Żadne z zagadnień, które tu poruszam, nie ogranicza się do jednego języka, a wszystkie moje wskazówki można zastosować do każdego innego języka zorientowanego obiektowo, np. Javy czy C++.

Podsumowanie

- Kod niszczeje. Za każdym razem, gdy zmieniasz coś w kodzie, ilość nieporządku (czyli entropia) rośnie. Jeśli pozostawisz system bez odpowiedniej opieki w postaci ciągłego czyszczenia i refaktoryzowania, będzie coraz bardziej złożony i coraz gorzej uporządkowany. Testy mogą pomóc zaradzić takiej sytuacji. Odgrywają rolę sieci bezpieczeństwa — narzędzia, które zapewnia ochronę przed większością defektów regresyjnych.
- Pisanie testów jednostkowych jest ważne. Tak samo ważne jest pisanie *dobrych* testów jednostkowych. Końcowy wynik projektów bez testów i projektów z kiepskimi testami jest taki sam — albo stagnacja, albo duża liczba defektów regresyjnych przy każdym wydaniu.
- Celem testowania jednostkowego jest umożliwienie zrównoważonego rozwoju projektu opartego na wytwarzaniu oprogramowania. Dobry test jednostkowy pomaga unikać stagnacji i utrzymać tempo pracy deweloperskiej w dłuższym czasie. Dysponując dobrym zestawem testowym, możesz być pewny, że zmiany w kodzie nie popsują istniejących funkcjonalności. Dzięki temu z kolei będziesz w stanie łatwo zmieniać kod i dodawać nowe funkcjonalności.
- Nie wszystkie testy zostały stworzone równymi. Na każdy z nich składają się czynniki kosztów i zysków, które musisz uważnie wyważyć. W zestawie testowym zachowaj tylko te testy, których wartość po odjęciu kosztów jest dodatnia. Zarówno kod źródłowy, jak i kod testowy to ciężary, a nie kapitał.
- Możliwość poddania kodu testom jednostkowym jest wygodnym papierkiem lakmusowym, ale sprawdza się tylko w jedną stronę. Jeśli nie możesz napisać testów jednostkowych do swojego kodu, to prawdopodobnie jest on niskiej jakości, ale sama możliwość wykonania testów jednostkowych nie gwarantuje wysokiej jakości kodu.
- Tak samo działają wskaźniki pokrycia. Niski procent pokrycia zwiastuje kłopoty, ale wysoki nie oznacza, że Twój zestaw testowy jest dobry.
- Pokrycie gałęzi daje lepszy wgląd w całościowość zestawu testowego, ale nie może być używane jako wyznacznik tego, czy zestaw jest wystarczająco dobry. Nie uwzględnia obecności asercji w testach ani ścieżek kodu w bibliotekach, które wykorzystujesz jako zależności.
- Narzucanie docelowej wartości wskaźnika pokrycia prowadzi do wielu wypaczeń. Dobrze jest osiągać wysoki wskaźnik pokrycia w kluczowych częściach systemu. Źle jest robić z tego poziomu bezwarunkowe wymaganie.

- Dobry zestaw testowy:
  - jest zintegrowany z cyklem wytwarzania oprogramowania,
  - koncentruje się na najważniejszych częściach bazy kodu,
  - dostarcza jak największej wartości przy jak najniższych kosztach utrzymania.
- Jedynym sposobem, by osiągnąć cel testowania jednostkowego (tj. umożliwić zrównoważony rozwój projektu), jest:
  - nauczenie się rozróżniania między dobrym a złym przypadkiem testowym,
  - poprawianie testów tak, by stały się bardziej wartościowe.

Każdy inżynier oprogramowania zna teoretyczne podstawy testowania. O tym, że sumienne przeprowadzenie testów ma podstawowe znaczenie dla jakości gotowego produktu, nie trzeba nikogo przekonywać. A jednak wciąż zbyt często okazuje się, że zaplanowanie, napisanie i przeprowadzenie testów jednostkowych w praktyce nie jest łatwym zadaniem. Co gorsza, niewłaściwe testy psują kod, mnożą błędy i zabierają mnóstwo cennego czasu i pieniędzy. Okazuje się, że dla uzyskania maksymalnej jakości projektu, który trzeba dostarczyć w krótkim czasie, konieczne jest nauczenie się praktycznego stosowania zasad i wzorców testowania jednostkowego.

Jeśli znasz już podstawy testowania jednostkowego, dzięki tej książce nauczysz się projektowania i pisania testów, które obierają za cel model domeny i pozostałe kluczowe obszary kodu. Ten przejrzyście napisany przewodnik poprowadzi Cię przez proces tworzenia zestawów testowych o optymalnej wartości, bezpiecznej automatyzacji testowania i umiejętnego włączania go w cykl życia oprogramowania. W książce znalazły się uniwersalne wskazówki dotyczące analizy dowolnych testów jednostkowych oraz porady odnoszące się do zmian testów następujących wraz ze zmianami kodu produkcyjnego. Nie zabrakło również informacji, dzięki którym sprawnie zidentyfikujesz i wykluczysz ewentualne antywzorce testowania. Materiał został bogato zilustrowany przejrzystymi przykładami kodu napisanego w C#. Naturalnie, mają one zastosowanie także dla innych języków programowania.

W książce między innymi:

- **cel testowania jednostkowego i cechy dobrych testów**
- **fundamenty porządnego testowania jednostkowego**
- **wpływ zaślepek i sterowników na stabilność testów**
- **style testowania jednostkowego**
- **zalety i ograniczenia testowania integracyjnego**

Przed wdrożeniem dobrze przetestuj swój kod!

Vladimir Khorikov jest inżynierem oprogramowania od ponad 15 lat. Zdobył tytuł Microsoft Most Valuable Professional. To także ekspert w zakresie testowania jednostkowego. W ciągu ostatnich kilku lat opublikował kilka popularnych serii wpisów o testowaniu jednostkowym na różnych blogach, a także prowadził internetowe kursy na ten temat. Jest ceniony za bogatą wiedzę teoretyczną, którą potrafi znakomicie wyjaśnić i zastosować w praktyce.

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

ISBN 978-83-283-6871-2

Cena: 67,00 zł