

Kent Beck

TDD.

Sztuka tworzenia dobrego kodu



Twórz niezawodny kod!

Helion

Tytuł oryginału: Test Driven Development: By Example

Tłumaczenie: Andrzej Grażyński

Projekt okładki: Studio Gravite / Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

ISBN: 978-83-283-6572-8

Authorized translation from the English language edition, entitled:
TEST DRIVEN DEVELOPMENT: BY EXAMPLE; ISBN 0321146530;
by Kent Beck; published by Pearson Education, Inc, publishing as Addison Wesley.
Copyright © 2003 Pearson Education, Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Polish language edition published by Helion SA. Copyright © 2014, 2020.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Helion SA dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Helion SA nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Materiały graficzne na okładce zostały wykorzystane za zgodą Shutterstock Images LLC.

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/tddszv>
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

Przedmowa	7
Podziękowania	13
Wstęp	15
CZĘŚĆ I. Obliczenia finansowe	19
Rozdział 1. Portfel wielowalutowy	21
Rozdział 2. Zdegenerowane obiekty	29
Rozdział 3. Równość dla wszystkich	33
Rozdział 4. Prywatność	37
Rozdział 5. Franki, dolary... ..	39
Rozdział 6. Równość dla wszystkich — tak, ale... ..	43
Rozdział 7. Jabłka i pomarańcze	49
Rozdział 8. Tworzymy obiekty	51
Rozdział 9. Mnożenie rozdwojone	55
Rozdział 10. Mnożenie jednolite	61
Rozdział 11. Korzenie wszelkiego zła	67
Rozdział 12. Dodawanie — ostatecznie	71
Rozdział 13. Zróbmy to	77
Rozdział 14. Wymiana	83
Rozdział 15. Mieszany koszyk walutowy	87
Rozdział 16. Abstrakcja — ostatecznie	91
Rozdział 17. Obliczenia finansowe — retrospekcja	95

CZĘŚĆ II. Przykład — xUnit	103
Rozdział 18. Pierwsze kroki	105
Rozdział 19. Każdy sobie... ..	111
Rozdział 20. Sprzątanie po sobie	115
Rozdział 21. Zliczanie	119
Rozdział 22. Zaplanować nad awariami	123
Rozdział 23. W jedności siła	125
Rozdział 24. xUnit — retrospekcja	131
 CZĘŚĆ III. Wzorce dla programowania sterowanego testami	 133
Rozdział 25. O wzorcach TDD	135
Rozdział 26. Wzorce czerwonego paska	145
Rozdział 27. Wzorce testowania	153
Rozdział 28. Wzorce zielonego paska	161
Rozdział 29. Wzorce xUnit	167
Rozdział 30. Wzorce projektowe	177
Rozdział 31. Refaktoryzacja	193
Rozdział 32. Doskonaląc TDD	205
 Dodatek A. Diagramy oddziaływań	 219
Dodatek B. Fibonacci	223
 Posłowie	 227
Skorowidz	229

Rozdział 31

Refaktoryzacja

Opisywane w tym rozdziale wzorce związane są ze zmianami w projekcie systemu, nawet radykalnymi, ale przeprowadzanymi metodą małych kroków.

Na gruncie TDD refaktoryzacja¹ przeprowadzana jest w naprawdę interesujący sposób. Z zasady, refaktoryzacja musi zachowywać określone warunki związane z semantyką kodu; w programowaniu sterowanym testami przez owe warunki rozumiemy stan wykonanych już testów — jeżeli dotąd wszystkie były zaliczane, zabieg wykonywany na kodzie nie może tego zmienić, kiedy chcemy go uważać za refaktoryzację. Rozumując w ten sposób, możemy w kodzie modelowym zastąpić wybrane stałe zmiennymi i — z pełną świadomością — nazywać to „uzmiennienie” refaktoryzacją i to nawet w sytuacji, gdy mamy za sobą zaliczony tylko jeden przypadek testowy. Nawet jeśli przeczuwamy, że następne testy załamią się wobec aktualnej postaci kodu modelowego, nie przejmujemy się tym faktem z prostego powodu — obecnie testy te jeszcze nie istnieją.

Jest rozumiałe, iż ta swoista „równoważność obserwacyjna” nakłada na programistę szczególną odpowiedzialność w zakresie dostatecznego przetestowania tworzonego kodu — „dostatecznego” w tym sensie, że zaliczenie wszystkich przeprowadzonych testów pozwoli na wysnucie założenia, że zaliczone zostałyby *wszystkie możliwe* testy. Nie da się niczym usprawiedliwić postawa w rodzaju: „Wiem o istniejącym problemie, ale skoro wszystkie testy zostały zaliczone, mogę integrować swój kod z bazowym”.

¹ Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts *Refaktoryzacja. Ulepszanie struktury istniejącego kodu*, wyd. Helion 2011, <http://helion.pl/ksiazki/refuko.htm>. Oryginał: *Refactoring: Improving the Design of Existing Code*, wyd. Reading, MA: Addison-Wesley, 1999. ISBN 0201485672.

Uzgadnianie różnic

W jaki sposób ujednolicić dwa podobnie wyglądające fragmenty kodu? Stopniowo upodabniać je do siebie, aż staną się identyczne.

Refaktoryzacja to proces niekiedy denerwujący, czasem oczywisty, a czasem zdradliwy. Wyodrębniając metodę w sposób mechaniczny, mamy niewielką szansę popsuć zachowania systemu, jednak refaktoryzacje bardziej skomplikowane wymagają starannego przesłедzenia przepływu sterowania i przepływu danych. Skomplikowane wnioskowanie pozwala żywić nadzieję, że zmiany, które zamierzamy wprowadzić, nie zmieniają semantyki systemu; potem przychodzi czas na konfrontację z rzeczywistością — i włosy stają dęba.

Taki „skok wiary” — czyli przyjęcie czegoś za pewnik bez empirycznego dowodu — to zjawisko, którego w refaktoryzacji staramy się unikać, stosując metodę małych kroków i zyskując tym samym konkretne sprzężenie zwrotne. I choć uniknięcie go nie zawsze się udaje, to przynajmniej można w ten sposób wydatnie zmniejszyć możliwość jego występowania.

Refaktoryzacja może być wykonywana w różnej skali:

- dwie podobne pętle — po ich ujednoliceniu możemy połączyć je w jedną pętlę,
- dwie podobne gałęzie instrukcji warunkowej — po ich ujednoliceniu możemy usunąć badanie warunku,
- dwie podobne metody — po ich ujednoliceniu możemy jedną usunąć,
- dwie podobne klasy — po ich ujednoliceniu możemy jedną z nich usunąć.

Niekiedy proces ujednolicania może zabrnąć w przysłowiową ślepą uliczkę i wówczas trzeba wycofać się z ostatniego kroku; oczywiście, byłoby wspaniale, gdyby był to krok dość mały i samo wycofywanie również byłoby nieskomplikowane. Jeżeli przykładowo chcemy wyeliminować kilka podklas, ostatnim banalnym krokiem poprzedzającym eliminację podklasy jest usunięcie jej ostatnich metod (a dokładniej: przesunięcie tych metod do nadklasy). Po usunięciu takiej pustej klasy zastępujemy w kodzie wszystkie do niej odwołania odwołaniami do nadklasy — bez zmiany semantyki tego kodu. Żeby jednak można było „opróżnić” z metod daną klasę, najpierw trzeba ujednolicić te metody z ich odpowiednikami w nadklasie.

Izolowanie zmian

Jak zmienić jeden fragment wieloczęściowej metody? Rozpocząć od wyizolowania tego fragmentu.

W tym momencie nieodmiennie przychodzi mi na myśl analogia z salą operacyjną: sterylna zasłona przykrywa ciało pacjenta, pozostawiając odkryte jedynie operowane miejsce, by chirurg nie rozpraszał swej uwagi na nieistotne w tej chwili obiekty. I nawet

jeśli zredukowanie (chwilowo) jednostki ludzkiej do lewego dolnego kwadrantu podbrzusza wyda się komuś prostą analogią, to ja, będąc pacjentem, bardzo ceniłbym sobie skoncentrowanie uwagi operatora.

Wyizolowanie zmienianego fragmentu ma charakter tymczasowy: gdy już się żądaną zmianę urzeczywistni, można je zlikwidować. Czasami można w ogóle zlikwidować fragment będący przedmiotem zmiany: jeżeli na przykład jedyną rzeczą, jakiej oczekujemy od metody `findRate()`, jest zwrócenie odpowiedniego obiektu, możemy explicite rozwinąć (*inline*) ciało tej metody w każdym miejscu jej wywołania, a oryginalną metodę usunąć. Tego rodzaju zmian nie można jednak przeprowadzać lekkomyślnie, należy dokonać wyboru między kosztem wynikającym z posiadania dodatkowej metody a kosztem jawnego pojawienia się nowej koncepcji w kilku miejscach kodu.

Koncepcję izolowania zmian najczęściej uwidacznia się pod postacią trzech wzorców: *Metody wyodrębnionej* (przeważnie), *Ekstrakcji interfejsu* i *Obiektu-metody*.

Migracje danych

Jak zmienić znaczenie pewnych danych? Tymczasowo zastosować ich duplikowanie.

Jak?

Załóżmy dla uproszczenia, że pod pojęciem „danych” rozumiemy tu zmienną instancyjną. Jedną ze strategii zmiany jej znaczenia jest strategia „od środka na zewnątrz”: zmieniamy najpierw wewnętrzną implementację klasy (dokładniej — reprezentowanie wspomnianej zmiennej instancyjnej w ramach tej implementacji), po czym modyfikujemy publiczny interfejs tej klasy:

1. definiujemy nową zmienną instancyjną, zastępującą (w nowym znaczeniu) starą zmienną instancyjną;
2. wszędzie, gdzie przypisywana jest wartość starej zmiennej instancyjnej, dodajemy przypisanie identycznej wartości nowej zmiennej instancyjnej;
3. dla każdej instrukcji wykorzystującej starą zmienną instancyjną dodajemy bliźniaczą instrukcję wykorzystującą nową zmienną instancyjną;
4. usuwamy wszystkie instrukcje wykorzystujące starą zmienną instancyjną;
5. zmieniamy publiczny interfejs klasy tak, by odzwierciedlał nowe znaczenie zmiennej instancyjnej.

Konkurencyjna strategia — „z zewnątrz do środka” — przedstawia się następująco:

1. dodajemy do publicznego interfejsu klasy nowy parametr, odzwierciedlający nowe znaczenie rzeczonyj zmiennej instancyjnej;

2. w implementacji klasy zastępujemy wszystkie powiązania starej zmiennej instancyjnej ze starym parametrem przez powiązania starej zmiennej instancyjnej z nowym parametrem;
3. usuwamy z interfejsu stary parametr;
4. w implementacji klasy zamieniamy każde wystąpienie starej zmiennej instancyjnej na wystąpienie nowej zmiennej instancyjnej;
5. usuwamy starą zmienną instancyjną.

Dlaczego?

Z problemem migracji danych spotykamy się często przy refaktoryzacji typu „jeden na wiele”. Załóżmy, że chcemy zaimplementować w ten sposób klasę `TestSuite` reprezentującą zestaw testów. Oto początek.

```
def testSuite(self):
    suite= TestSuite()
    suite.add(WasRun("testMethod"))
    suite.run(self.result)
    assert("1 run, 0 failed" == self.result.summary())
```

Zaczynamy od wariantu „jeden”, czyli zestawu testowego złożonego z pojedynczego testu.

```
class TestSuite:
    def add(self, test):
        self.test= test
    def run(self, result):
        self.test.run(result)
```

Ten pojedynczy test reprezentowany jest przez zmienną instancyjną `test`, która (w opisywanym powyżej ujęciu) jest zmienną instancyjną „w starym znaczeniu”. Zmienna instancyjna „w nowym znaczeniu” reprezentować będzie kolekcję testów, więc nadamy jej nazwę `tests`. Zgodnie z pierwszą z opisanych strategii, wprowadzamy tymczasową duplikację danych, czyli współlistnienie w klasie `TestSuite` obu zmiennych instancyjnych — `test` i `tests`. Inicjujemy drugą (nową) zmienną instancyjną.

TestSuite

```
def __init__(self):
    self.tests= []
```

Wszędzie, gdzie występuje przypisanie wartości do starej zmiennej (`test`), dodajemy analogiczne przypisanie do nowej zmiennej (`tests`); tutaj „przypisanie” ma postać dodania nowego testu do kolekcji.

TestSuite

```
def add(self, test):
    self.test= test
    self.tests.append(test)
```

Z perspektywy aktualnie zaliczonych przypadków testowych właśnie wykonaliśmy refaktoryzację — semantyka klasy nie powinna się zmienić, ponieważ zestaw testowy w dalszym ciągu zawiera tylko jeden test.

```
TestSuite
def run(self, result):
    for test in self.tests:
        test.run(result)
```

Ponieważ w kodzie klasy nie ma już odwołań do zmiennej instancyjnej `test`, możemy ją usunąć (usuwając drugą instrukcję z metody `add()`).

```
TestSuite
def add(self, test):
    self.tests.append(test)
```

Opisane migrowanie danych krok po kroku okazuje się użyteczne także w przypadku konwersji danych między równoważnymi formatami różnych protokołów, na przykład w Javie między enumeratorem wektora a iteratorem kolekcji.

Wyodrębnianie metody

Jak uprościć treść długiej, skomplikowanej metody? Zastąpić jej fragment wywołaniem nowej metody, enkapsulując ten fragment.

Jak?

Metoda wyodrębniona to jeden z najbardziej skomplikowanych wzorców refaktoryzacji elementarnej, tu opiszę tylko jego typowy przypadek. Jednocześnie wyodrębnianie metody to jeden z najczęściej implementowanych mechanizmów automatycznej refaktoryzacji, więc rzadko konieczne jest jego „ręczne” realizowanie — które przeprowadza się według następującego scenariusza.

1. Znajdź fragment (region) metody, który kwalifikuje się do wyodrębnienia w postaci osobnej metody; najbardziej prawdopodobnymi do tego kandydaturami są ciała pętli, kompletne pętle i gałęzie instrukcji warunkowych.
2. Upewnij się, że we wspomnianym fragmencie nie istnieją przypisania wartości do zmiennych lokalnych (tymczasowych) używanych poza tym fragmentem.
3. Skopiuj wyodrębniany fragment jako ciało nowej metody.
4. Każdy parametr oryginalnej metody używany w wyodrębnianym fragmencie oraz każdą zmienną tymczasową używaną w tym fragmencie uczyn parametrami nowej metody.
5. Zastąp wyodrębniany fragment wywołaniem nowej metody (z odpowiednimi parametrami).

Dlaczego?

Z wyodrębniania metody korzystam zawsze wtedy, kiedy pomaga mi ono zrozumieć skomplikowany kod („w tym miejscu dzieje się coś konkretnego, jak mógłbym to nazwać?”). Po upływie pół godziny lepiej rozumiesz to, co robisz, a Twój partner widzi, że *naprawdę* jesteś pomocny.

Wyodrębnianie metody stosuję również wówczas, gdy w dwóch metodach dostrzegam podobne fragmenty (Refactoring Browser w środowisku Smalltalka oferuje nawet pomoc w postaci sprawdzenia, czy wyodrębniana właśnie metoda nie jest równoważna którejś z już istniejących i jeśli jest, proponuje użycie w zamian tej istniejącej).

Wyodrębnianie metody ma zawsze granice sensowności — nie należy tworzyć metod zbyt „drobnych”, bo te wcale czytelności kodu nie poprawiają. Jeśli czuję, że przekraczam tę granicę, często stosuję rozwijanie metody (*inlining*) — mając cały kod w jednym miejscu, przyglądam się, czy coś nadaje się do (sensownego) wyodrębnienia.

Rozwijanie metody

Jak uprościć przepływ sterowania w kodzie zbyt zagnieżdżonym lub zbyt rozrzuconym? Zastąpić wywołanie metody jej treścią.

Jak?

Opisana operacja nazywana jest rozwijaniem (*inlining*) metody i praktycznie wykonuje się ją następująco.

1. Skopiuj do schowka ciało metody.
2. Wklej ze schowka ciało metody w miejscu jej wywołania.
3. Zastąp we wklejonym kodzie parametry formalne metody aktualnymi parametrami jej wywołania. Jeśli wywoływana metoda związana jest z efektami ubocznymi² (na przykład `reader.getNext()`), koniecznie zapamiętaj w zmiennej lokalnej wynik jej wywołania.

Dlaczego?

Jednemu z recenzentów tej książki nie spodobała się sekwencja z części pierwszej, w której to sekwencji redukcja obiektu z typu `Expression` na typ `Money` jest zadaniem obiektu klasy `Bank`.

² Chodzi tu o tzw. funkcje zależne od czasu (*time-dependent functions*), czyli funkcje, które przy różnych wywołaniach zwracają różne wartości dla tych samych argumentów. Konceptyjnie zachowują się tak, jakby chwila ich wywołania była dodatkowym, niejawnym parametrem tego wywołania (stąd nazwa). Do takich metod należy między innymi cytowana metoda `getNext()` każdego iteratora — *przyjp. tłum.*

```
public void testSimpleAddition() {
    Money five= Money.dollar(5);
    Expression sum= five.plus(five);
    Bank bank= new Bank();
    Money reduced= bank.reduce(sum, "USD");
    assertEquals(Money.dollar(10), reduced);
}
```

„To jest zbyt skomplikowane. Czy nie można by tego zadania powierzyć samej klasie Money?”.

No to poeksperymentujmy. Rozwińmy implementację metody `Bank.reduce()`:

```
public void testSimpleAddition() {
    Money five= Money.dollar(5);
    Expression sum= five.plus(five);
    Bank bank= new Bank();
    Money reduced= sum.reduce(bank, "USD");
    assertEquals(Money.dollar(10), reduced);
}
```

Czy ta druga wersja jest czytelniejsza? Kwestia gustu, niewątpliwie jednak uproszczyła się nieco struktura sterowania. Zawsze wtedy, gdy wykonuję refaktoryzację, wyobrażam sobie system jako zbiór elementów logicznych i sterowania przepływającego między obiektami. I gdy dostrzegam w tym jakiś pomysł na refaktoryzację, wypróbuję go w praktyce.

Zdarza mi się czasem (nie pytaj, jak często) w ferworze dyskusji wpaść w pułapkę mojej własnej pomysłowości. Rozwijanie metody staje się wówczas sposobem na wydostanie się z tej pułapki. („To przesyłam tu, to tu... stop! Co dzieje się tutaj?”). Rozwijam wówczas kilka warstw abstrakcji, ogładam, co dzieje się naprawdę, i mogę ponownie abstrahować kod w oparciu o solidne podstawy — rzeczywistą potrzebę, a nie własne wyobrażenia.

Ekstrakcja interfejsu

Jak, w języku Java, zrealizować dodatkowe implementacje już zaimplementowanych operacji? Zdefiniować interfejs zawierający współdzielone operacje.

Jak?

1. Zadeklaruj interfejs; być może najbardziej odpowiednia nazwa dla tego interfejsu będzie już nazwą istniejącej klasy, wtedy nazwę tej klasy trzeba zmienić.
2. Zadeklaruj klasę, zawierającą dotychczasową implementację wspomnianych operacji, jako klasę implementującą utworzony interfejs; rzeczona implementacja operacji stanie się wówczas formalnie implementacją interfejsu.

3. Dodaj do interfejsu ewentualnie inne pożądane operacje, zwiększając — jeśli trzeba — ich widoczność w klasie, która je implementuje.
4. W wywołaniach wspomnianych operacji zastąp odwołania do klasy odwołaniami do interfejsu.

Dlaczego?

Często potrzeba ekstrakcji interfejsu wiąże się — w sposób mniej lub bardziej ewidentny — z wtórną implementacją operacji już zaimplementowanych w postaci metod jakiejś klasy. Jeśli przykładowo klasą tą jest `Rectangle` (prostokąt), a my chcemy zaimplementować analogiczne operacje w klasie `Oval` (reprezentującej elipsy), to wielce odpowiednią dla ekstrahowanego interfejsu wydaje się nazwa `Shape` („kształt”); nie zawsze jednak znalezienie odpowiedniej metafory przychodzi tak łatwo.

Z ekstrakcją interfejsu związane jest często użycie dwóch wzorców, które opisane zostały w rozdziale 27. — obiektu-atrapy i symulowanej katastrofy. Kwestię nazewnictwa należy wówczas przemyśleć szczególnie starannie, wszak mamy dopiero jedną rzeczywistą implementację. Niekiedy odczuwam pokusę, by nie przejmować się tym zbytnio i po prostu wybrać (na przykład) nazwę `IFile` dla interfejsu ekstrahującego operacje implementowane dotychczas w klasie `File`. Dogłębna analiza prowadzi jednak do wniosku bardziej konstruktywnego: `File` powinno być nazwą interfejsu, natomiast nazwę wspomnianej klasy należy zmienić na `DiskFile`, ponieważ dedykowana jest obsłudze plików dyskowych.

Przenoszenie metody

Jak przenieść metodę do miejsca, w którym faktycznie powinna się znajdować? Zdefiniować ją we właściwej klasie i wywołać.

Jak?

1. Skopiuj kod metody do schowka.
2. Wklej metodę ze schowka wewnątrz definicji klasy docelowej, ewentualnie zmień jej (metody) nazwę.
3. Jeśli w kodzie metody istnieją odwołania do oryginalnego obiektu, dodaj do utworzonej kopii parametr reprezentujący ten obiekt. Podobnie, jeżeli w kodzie metody wykorzystywane są wartości zmiennych instancyjnych oryginalnego obiektu, należy dodać parametr dla każdej z tych zmiennych. Jeżeli jednak w kodzie metody zmieniane są wartości zmiennych instancyjnych oryginalnego obiektu, musisz się — niestety — poddać.
4. Zastąp ciało oryginalnej metody wywołaniem nowej metody.

Dlaczego?

To jeden z moich ulubionych przykładów dydaktycznych z dziedziny refaktoryzacji, stanowi znakomitą ilustrację tego, jak złudne mogą okazywać się przekonania nieoparte faktami. Za obliczanie pola figury geometrycznej odpowiedzialny jest interfejs *Shape*.

Shape

```
...
int width= bounds.right() - bounds.left();
int height= bounds.bottom() - bounds.top();
int area= width * height;
...
```

Każdorazowo, gdy w danej metodzie widzę więcej niż jeden komunikat wysyłany do tego samego obiektu, staję się podejrzliwy. W powyższym przykładzie do obiektu *bounds* wysyłane są cztery komunikaty, co może sugerować, że powyższy fragment metody powinien znaleźć się w innym miejscu.

Rectangle

```
public int area() {
    int width= this.right() - this.left();
    int height= this.bottom() - this.top();
    return width * height;
}
```

Shape

```
...
int area= bounds.area();
...
```

Przykład ten ukazuje trzy ciekawe własności refaktoryzacji w wydaniu przenoszenia metody:

- łatwo dostrzec jej potrzebę, bez konieczności głębszego zrozumienia znaczenia kodu — wystarczy dostrzec wielokrotne odwołania do „obcego” obiektu;
- jest szybka i bezpieczna;
- jej rezultat bywa pouczający. („Ale przecież klasa *Rectangle* nie powinna wykonywać żadnych obliczeń...no tak, już widzę, tak *jest* lepiej”).

Niekiedy celowe okazuje się przeniesienie tylko fragmentu metody. Należy wówczas wyekstrahować ten fragment w postaci metody, przenieść tę metodę w wyznaczone miejsce i wstawić jej wywołanie w miejsce wyekstrahowanego fragmentu (wiele środowisk udostępnia opcję wykonania tej operacji w jednym kroku).

Obiekt-metoda

Jak optymalnie reprezentować skomplikowaną metodę wymagającą wielu parametrów i zmiennych lokalnych? Najlepiej w postaci obiektu.

Jak?

Obiekt reprezentujący metodę konstruuje się w następujący sposób.

1. Zdefiniuj klasę o konstruktorze z takim samym zestawem parametrów, jaki ma przedmiotowa metoda.
2. Dla każdej zmiennej lokalnej metody utwórz zmienną instancyjną o tej samej nazwie³.
3. Zdefiniuj metodę `run()`, o ciele takim samym jak ciało oryginalnej metody.
4. W oryginalnej metodzie zastąp ciało sekwencją obejmującą utworzenie obiektu wspomnianej klasy i wywołanie metody `run()` tego obiektu.

Dlaczego?

Obiekt-metoda jest pomocny w przygotowaniach do wprowadzenia nowego rodzaju logiki do systemu. Załóżmy na przykład, że dysponujemy szeregiem metod związanych z obliczaniem przepływów pieniężnych (*cash flow*), obliczających cząstkowe komponenty tego przepływu. Kiedy chcemy obliczyć sumaryczną wartość netto przepływu, rozpoczynamy od utworzenia Obiektu-metody zgodnego z dotychczasowym stylem obliczeń. Następnie programujemy obliczenia w nowym stylu, wraz z towarzyszącymi im testami, każdy — oczywiście — w małej skali. Przejście na obliczenia w nowym stylu będzie już tylko pojedynczym krokiem.

Obiekty-metody przydają się też do upraszczania kodu, który nie kwalifikuje się do refaktoryzacji w postaci metody wyodrębnionej ze względu na duże nasycenie parametrami i zmiennymi lokalnymi; w rezultacie wyodrębniona metoda wyglądałaby jeszcze gorzej niż oryginalny kod, ze względu na zbyt złożoną sygnaturę (każda zmienna lokalna oryginalnego kodu reprezentowana byłaby jako parametr nowej metody). W tej sytuacji utworzenie Obiektu-metody oznaczałoby utworzenie nowej przestrzeni nazw, wewnątrz której wyodrębnienie metody nie wymagałoby przekazywania żadnych parametrów.

³ W języku Java ten krok nie budzi wątpliwości, ale nie w każdym języku da się wykonać bezpośrednio: przykładowo w języku Ruby nazwa zmiennej instancyjnej musi rozpoczynać się od znaku `@`, a nazwa zmiennej lokalnej — od małej litery lub znaku podkreślenia; nie można więc utworzyć zmiennej instancyjnej o nazwie identycznej z nazwą zmiennej lokalnej. Uogólnienie powyższego scenariusza mogłoby więc wyglądać na przykład tak:

2. Dla każdej zmiennej lokalnej utwórz odpowiadającą jej zmienną instancyjną.
3. Zdefiniuj metodę `run()`, o ciele takim samym jak ciało oryginalnej metody, i zastąp każde wystąpienie zmiennej lokalnej odpowiadającą jej zmienną instancyjną — *przyp. tłum.*

Dodawanie parametru

W jaki sposób dodać parametr do metody?

Jak?

1. Jeśli metoda pochodzi z interfejsu, rozpocznij od dodania parametru do deklaracji tej metody w interfejsie.
2. Dodaj parametr do definicji metody w klasie.
3. Uruchom kompilację i kierując się błędami sygnalizowanymi przez kompilator, zmodyfikuj odpowiednio wywołania metody.

Dlaczego?

Poszerzanie sygnatury metody o kolejny parametr jest często naturalnym krokiem w rozwoju aplikacji: po zaliczeniu zestawu testów uświadamiamy sobie, że w nowych warunkach musimy uwzględnić w danej metodzie dodatkowe informacje, które przekazywane będą do niej właśnie za pośrednictwem dodatkowego parametru.

Dodawanie parametru może też być jednym z kroków migracji danych, opisywanej wcześniej w tym rozdziale: dodajemy parametr reprezentujący pewną informację w nowym znaczeniu, usuwamy wszystkie odwołania do starego parametru, a na końcu usuwamy z sygnatury stary parametr.

Parametr metody a parametr konstruktora

Jak przenieść parametr z metody (metod) do konstruktora?

Jak?

1. Dodaj parametr do konstruktora.
2. Dodaj zmienną instancyjną o takiej samej nazwie jak parametr⁴.
3. W konstruktorze nadaj nowej zmiennej instancyjnej wartość przekazywaną przez nowy parametr.
4. W treści metody zmień każde wystąpienie parametru (*parametr*) na odwołanie kwalifikowane (*this.parametr*).

⁴ Patrz mój poprzedni komentarz dotyczący nazewnictwa zmiennych — *przyp. tłum.*

5. Usuń parametr z sygnatury metody i wszystkich wywołań tej metody.
6. Usuń niepotrzebne kwalifikatory `this` z ciała metody.
7. Opcjonalnie — możesz zmienić nazwę nowej zmiennej instancyjnej, stosownie do potrzeb, konwencji, gustu i tak dalej.

Dlaczego?

Jeżeli do wielu metod przekazywany jest parametr w tym samym znaczeniu, można uprościć API klasy, wprowadzając określoną informację do tworzonego obiektu już na etapie jego kreowania, a nie dopiero w momencie wywoływania metod (warto zauważyć, że redukujemy w ten sposób duplikację).

I *vice versa*: jeśli dana zmienna instancyjna wykorzystywana jest tylko przez jedną metodę, być może bardziej logiczne będzie przekazywanie niezbędnej informacji do tej metody za pośrednictwem dodatkowego parametru jej wywołania (rzeczona zmienna instancyjna stanie się wówczas zbędna).

Skorowidz

A

Act, *Patrz:* obiekt akcja
akronim 3A, 111
aliasowanie, 33, 178, 180
aplikacja współbieżna, *Patrz:*
współbieżność
application test-driven
development, *Patrz:* ATDD
Arnoldi Massimo, 154
Arrange, *Patrz:* obiekt
aranżacja
arytmetyka wielowalutowa, 72
asercja, 132, 140, 141, 167
niespełniona, 132
Assert, *Patrz:* asercja
ATDD, 211, 212
atraktor, 216
awaria, 132

B

baza danych
atrapa, 154
obiektowa, 15
schematów, 218
bezpieczeństwo, 10
Binder Bob, 208
błąd, 132
bootstrap problem, *Patrz:*
problem ładowania
wstępnego

C

ciąg Fibonacciego, 223
code-critic, *Patrz:* kod analiza
krytyczna
committed, *Patrz:* transakcja
pamięciowa zatwierdzenie
Composite, *Patrz:* kompozyt
Coplien Jim, 57
CORBA/EJB, 218
Cunningham Ward, 11, 15, 72,
97, 189

D

dane
migracja, 195, 196
realistyczne, 181
realistyczne, 142
testowe, 142, 163
debugger, 168
debugowanie, 63
development, *Patrz:*
programowanie
diagram oddziaływań, 219
aktywność, 219
sprzężenie, 219, 220
driven, *Patrz:* sterowane
duplikacja, 25, 40, 101, 125,
128, 162, 183, 184, 186, 217
danych, 77
dziedziczenie, 53, 178, 183, 185
dziennik, 115, 156

E

edytor graficzny, 184
edytor vi, 210
Ennis Darach, 218
error, *Patrz:* błąd
expression, *Patrz:* wyrażenie
eXtreme Programming, 217

F

factory method, *Patrz:* metoda
fabrykująca
failure, *Patrz:* awaria
feedback, *Patrz:* sprzężenie
zwrotne
Fibonacciego ciąg, 223
fikstura, 92, 169, 211
zewnętrzna, 171
Fowler Martin, 131
framework, 11, 207
Freeman Steve, 25
funkcja, 108
sprintf, 120

G

Gabriel Richard, 158
Gamma Erich, 10, 182, 184
gniazdo, 141

H

Hansen Peter, 162
haszowanie, 85

I

IDE, 168, 174, 179, 206
 impostor, 178, 188, 189
 refaktoryzacja, 189
 inlining, *Patrz:* metoda
 rozwijanie
 instrukcja
 for, 126
 switch, 186
 interfejs, 74, 91
 CGI, 218
 ekstrakcja, 195, 199, 200
 GUI, 218
 java.io.Externalizable, 191
 Runnable, 179
 Swing, 218
 Test, 132
 interpreter, 211, 218
 iterator, 126

J

Java, 168, 183, 199
 JDBC, 218
 Jeffries Ron, 7, 29
 język
 Python, *Patrz:* Python
 Smalltalk, *Patrz:* Smalltalk
 środowisko, 131
 JHotDraw, 182
 JMS, 218
 JProbe, 100
 JUnit, 11, 22, 92, 97, 132, 172
 JXUnit, 168

K

klasa, 74, 186
 abstrakcyjna, 62, 91
 polimorfizm, *Patrz:*
 polimorfizm
 reprezentująca fiksturę, 172
 rzutowanie, *Patrz:*
 rzutowanie
 TestCase, 132
 testowanie, 30, 39
 TestSuite, 125, 132

kod

analiza krytyczna, 96
 inicjujący, 169
 integrowanie, 217
 kopiowanie, 40, 43
 metryki, 98
 obcy, 218
 odporność na obciążenie,
 100
 redundantny, 162
 semantyka, 193
 upraszczanie, 202
 użyteczność, 100
 wspólny fikstury, 169
 wydajność, 100
 kompilator, 218
 kompozyt, 88, 178, 189, 191
 sprzeczność wewnętrzna,
 189
 komunikat, 156, 179
 automatyczna zmiana
 nazw, 179
 wymiana, 218
 konstruktor, 33, 57, 67, 178, 187
 parametr, 203
 krzywa leptokurtozy, 99
 kurs wymiany, 83

L

Lange Manfred, 152
 LifeWare, 211
 Lisp, 179
 Logan Patrick, 173

Ł

łańcuch
 dziennik, 156
 znaków, 55

M

Mackinnon Tim, 152
 Martin Robert, 214
 Mean Time Between Failures,
 Patrz: MTBF
 menedżer zabezpieczeń, 182

metoda, 141

abstrakcyjna, 178, 183, 185
 dla celów debugowania, 63
 fabrykująca, 52, 57, 178, 187
 inline, 61, 65
 izolowanie fragmentu, 194
 parametr, 203, 204
 przenoszenie, 200
 redukcyjna, 83
 rejestrowanie wywołania,
 115
 rekurencyjna, 225
 rozwijanie, 198
 szablonowa, 178, 182
 tearDown, 171
 testowa, 105, 106, 172, 174
 writeExternal, 191
 wyodrębianie, 197, 198
 wyodrębniona, 195
 metodologia prysznic, 150
 mock object, *Patrz:* obiekt
 atrapa
 MTBF, 208, 218

N

nadklasa, 43, 47
 Newkirk Jim, 141, 149
 null, 35, 106

O

obiekt, 33, 72, 177
 akcja, 111
 aranżacja, 111
 asercja, 111, *Patrz:* asercja
 atrapa, 154, 155, 158
 implementacja, 72
 klasy abstrakcyjnej, 62
 kolekcja, 165, 191
 komunikacja, 155
 metoda, 195, 202, 203
 nieistniejący, 178
 o nieistotnej tożsamości, 179
 otwarty/zamknięty, 208
 podłączalny, 178, 184, 188
 porównywanie, 63
 protokół zewnętrzny, 72
 pusty, 178, 181, 189

rozproszony, 218
 selekcja, 184
 tworzenie, 178
 uruchomienie obliczeń, 178
 wartości, 33, 34, 178, 179,
 181
 wektor, 189
 wspólny dla wielu testów,
 169
 zerowy, *Patrz:* obiekt pusty
off-by-one error, 165
 operacja
 haszowania, 181
 porównania, 181
 oprogramowanie
 BIOS, 105
 UEFI, 105

P

parametr, 203, 204
 domyślny, 126
 kolekcjonujący, 178, 191
 podklasa, 43, 47, 178
 podmetoda, 183
 pole
 prywatne, 36, 38
 publiczne, 79
 polecenie, 178, 179
 polimorfizm, 81, 185
 preparacja, 161, 164
 problem ładowania wstępnego,
 105
 programowanie, 216
 parami, 152
 sekwencje typowe, 182
 sterowane architekturą, 29
 sterowane testami,
Patrz: TDD
 w parach, 217
 w pojedynkę, 158
 w warunkach stresu, 136
 w zespole, 159
 projektowanie
 organiczne, 8
 sterowane wzorcami, 214
 protokół, 178
 publiczny, 168

Python, 103, 129
 klasa, 108
 metoda, 108
 parametr domyślny, 126

R

Reeves Gareth, 218
 Refactoring Browser, 198, 206,
 210
 refaktoring, *Patrz:*
 refaktoryzacja
 refaktoryzacja, 8, 35, 36, 45, 80,
 108, 109, 116, 139, 177, 193,
 194, 212, 217
 automatyczna, 197
 automatyzacja, 206
 elementarna, 197
 impostor, 189
 jeden na wiele, 196
 lista, 139
 oprogramowania
 symulacyjnego, 142
 skala, 194
 rollback, *Patrz:* transakcja
 pamięciowa wycofanie
 rozkład
 grubego ogona, 99
 normalny, 99
 RPC, 218
 Ruby, 179
 rusztowanie, 169
 rzutowanie, 79

S

samopodobieństwo, 140
 samopodstawienie, 155, 156, 157
 scaffold, *Patrz:* rusztowanie
 selektor podłączalny, 178, 185,
 186, 187
 self-shunt, *Patrz:*
 samopodstawienie
 serwlet
 JSP, 218
 Struts, 218
 shower methodology, *Patrz:*
 metodologia prysznic
 Small-Lint, 96

Smalltalk, 96, 142, 168, 179,
 180, 183, 198, 206, 208, 210
 socket, *Patrz:* gniazdo
 sprzężenie zwrotne, 220
 negatywne, 221, 222
 pozytywne, 221, 222
 Stegner Wallace, 43
 sterowane, 216
 SUnit, 168
 system
 czasu rzeczywistego, 142
 element funkcjonalny, 140,
 141

Ś

średni czas międzyawaryjny,
Patrz: MTBF
 środowisko testowe, 131

T

tabela wymiany kursów, 84
 tablica haszowana, 84
 klucz, 34, 84
 TDD, 7, 29, 31, 45, 96, 101, 103,
 105, 178, 205, 207, 210, 212,
 213, 215, 217
 cykl, 11, 40, 99
 skalowalność, 210
 triangulacja, *Patrz:*
 triangulacja
 warunki początkowe, 213
 wzorzec, 133, *Patrz:*
 wzorzec projektowy
 test, 212, 216
 aplikacyjny, 211
 częstotliwość, 97
 dane, 143, 144
 realistyczne, 142
 testowe, 142
 długotrwałość, 207
 ewaluacja, 100
 implementacja, 161, 164
 spreparowana, 161, 162,
 164
 izolowanie, 111, 124, 137,
 138
 jednostkowy, 211, 218

- test
 - kod konfiguracyjny, 207
 - kolejność, 119, 145, 213
 - kolekcja, 191
 - kompilacja, 8, 22
 - liczba zmian, 99
 - lista, 138
 - nadwrażliwość, 207
 - obiekt wspólny, 169
 - objaśniający, 148
 - odpowiedź, 141, 146
 - patologia, 207
 - podkładanie defektów, 100
 - pokrycie instrukcji, 100
 - porównawczy, 142
 - pouczający, 148
 - prostota, 113
 - redundancja, 210
 - regresyjny, 149
 - równoległy, 142
 - sprawdzanie poprawności, 167
 - sprzężenie z kodem modelowym, 38
 - startowy, 146, 147
 - systemu czasu
 - rzeczywistego, 142
 - środowisko, 131
 - testAssertPosInfinityNotE
qualsNegInfinity, 172
 - tworzenie, 21, 140
 - uniwersalny, 155
 - usprawnianie, 38
 - wtórny, 153
 - wydajność, 111, 138
 - wyjątku, 174
 - załamanie, 111, 112, 121, 123
 - zliczanie, 123
 - zestaw, 174
 - test infected, *Patrz:* zarażenie testami
 - Test-Driven Development, *Patrz:* TDD
 - testowanie, 135
 - na poziomie aplikacji, *Patrz:* ATDD
 - równoległe, 142
 - strategia, 146
 - sytuacji wyjątkowej, 157
 - w skali makro, 211
 - zautomatyzowane, 136
 - transakcja pamięciowa, 136, 137
 - triangulacja, 31, 34, 101, 150, 163, 164
- ## U
- Ungar Dave, 150
 - unit tests, *Patrz:* test jednostkowy
- ## V
- Value Object, *Patrz:* Obiekt Wartości
- ## W
- Wake Bill, 111, 217
 - waluta referencyjna, 72
 - wartość
 - nil, *Patrz:* null
 - null, *Patrz:* null
 - pusta, *Patrz:* wartość null
 - Weinberg Gerald Marvin, 135, 219
 - wielowątkowość, *Patrz:* współbieżność
 - współbieżność, 10, 179
 - wyjątek, 121
 - ClassCastException, 78
 - generowanie, 157
 - przechwytywanie, 119
 - SubclassResponsibility, 183
 - testowanie, 174
 - wyrazenie, 73, 74, 96
 - wzorzec projektowy, 177, 214
 - ekstrakcji interfejsu, 195, 199, 200
 - impostor, 178, 188
 - kompozyt, 125, 126, 178, 189
 - metoda fabrykująca, 178, 187
 - metoda szablonowa, 178, 182
 - metoda wyodrębniona, 195
 - obiekt podłączalny, 178, 184, 188
 - obiekt pusty, 178, 189
 - obiekt wartości, 178, 179
 - obiekt Wartości, *Patrz:* Obiekt Wartości
 - obiekt-metody, 195
 - parametr
 - kolekcjonowania, 126
 - parametr kolekcjonujący, 178, 191
 - polecenie, 178, 179
 - pusty obiekt, 181
 - Selektor podłączalny, 178, 185
- ## X
- xUnit, 103, 131, 156, 174
 - wzorzec, 167
- ## Z
- zarażenie testami, 10
 - zasoby, 115
 - alokowanie, 115
 - atrapa, 154
 - niedostępne, 154
 - zwalnianie, 115, 171
 - zestaw testowy, 174
 - zmienna
 - globalna, 129, 154, 192
 - instancyjna, 62, 169, 195, 204
 - lokalna, 169
 - private, 44, 169
 - protected, 44
 - prywatna testowanie, 168
 - robocza, 44
 - tymczasowa, 46
 - znak %, 120

PROGRAM PARTNERSKI

— GRUPY HELION —



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA
Helion 

TDD.

Sztuka tworzenia dobrego kodu

Idealny kod, pokryty w pełni testami, to marzenie każdego dewelopera. Niestety, marzenia zazwyczaj rozmiągają się z rzeczywistością. Codziennosc większości programistów to nieczytelny kod i brak testów. Są to doskonałe warunki do powstawania błędów, często bardzo trudnych do wykrycia. Czy jest sposób, żeby wybrnąć z tego ślepego zaułka? Czy są techniki pozwalające tworzyć przejrzysty kod, którego zmiana nie będzie sprawiała trudności?

Oczywiście, że tak — wykorzystaj TDD (ang. Test Driven Development). Programowanie sterowane testami można sprowadzić do prostej zasady: w pierwszej kolejności napisz test, a następnie kod, który ma być testowany. Kent Beck w swojej książce zaprezentuje Ci w praktyce podejście TDD i pokaże, jak wdrożyć jego zasady w codziennej pracy. Zapoznaj się z licznymi przykładami zastosowania tej metody, przydatnymi poradami i najlepszymi wzorcami. To doskonały początek, żeby wdrożyć TDD w Twoim projekcie. Dowiedz się, jak tworzyć idealny kod!

Dzięki tej książce:

- poznasz technikę programowania sterowanego testami
- stworzysz czytelny kod, doskonale pokryty testami
- nie będziesz się bał dokonywać zmian w Twoim kodzie
- Twój kod osiągnie nowe standardy jakości

Tvoja przepustka do świata idealnego kodu!

Helion 	<i>Sprawdź nasze szkolenia!</i>	KOD KORZYŚCI Sięgnij po więcej! ▶ 
 helion.pl	SZKOLENIA 	ISBN 978-83-283-6572-8
 HELION SA ul. Kościuszki 1c 44-100 Gliwice tel.: 32 230 98 63 helion@helion.pl	AKADEMIA IT & BUSINESS WWW.SZKOLENIA.HELION.PL	 9 788328 365728
INFORMATYKA W NAJLEPSZYM WYDANIU		Cena: 59,00 zł