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.

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.

Spis treści

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 refaktoryzacja[1] 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 zrozumiałe, 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 popsucia zachowania systemu, jednak refaktoryzacje bardziej skomplikowane wymagają starannego prześledzenia przepływu sterowania i przepływu danych. Skomplikowane wnioskowanie pozwala żywić nadzieję, że zmiany, które zamierzamy wprowadzić, nie zmienią 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 prostacką 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 rzeczonej 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 współistnienie 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, enkapsulującej 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 uczyń 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 ubocznymi[2] (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. Koncepcyjnie 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 — *przyp. 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 uprościła 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, wypróbowuję 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, oglądam, 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 niepoparte 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 nazwie[3].
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 iż 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 parametr[4].
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

I

J

K

L

Ł

M

N

O

U

V

W

X

Z

TDD.

Sztuka tworzenia dobrego kodu

Idealny kod, pokryty w pełni testami, to marzenie każdego dewelopera. Niestety, marzenia zazwyczaj rozmijają się z rzeczywistością. Codzienność 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

Twoja przepustka do świata idealnego kodu!

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

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-283-6572-8
9 788328 365728

INFORMATYKA W NAJLEPSZYM WYDANIU
Cena: 59,00 zł