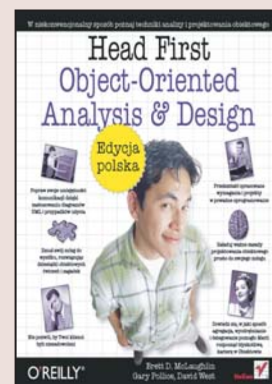
IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG
TWÓJ KOSZYK
DODAJ DO KOSZYKA
CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK
CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

Head First Object-Oriented Analysis and Design. Edycja polska



Autor: Brett D. McLaughlin, Gary Pollice, David West
Tłumaczenie: Piotr Rajca
ISBN: 978-83-246-0965-9
Tytuł oryginału: Head First Object-Oriented Analysis and Design
Format: 200x230, stron: 616
Przykłady na ftp: 165 kB

Poznaj techniki analizy i projektowania obiektowego

- Naucz się zbierać wymagania od użytkowników systemu
- Zarządzaj zmianami w specyfikacji
- Przeprowadź analizę i wykonaj projekt

Systemy informatyczne stają się coraz bardziej rozbudowane. Programowanie obiektowe znacznie ułatwia ich tworzenie i późniejsze modyfikacje, aby jednak system był sprawny i funkcjonalny, musi zostać zaprojektowany w oparciu o prawidłowo zebrane wymagania. Tu również z pomocą przychodzi metodologia obiektowa – wzorce projektowe, język UML i odpowiednie narzędzia niezwykle ułatwiają przygotowanie dobrego projektu.

Jeśli rozbudowane przykłady, skomplikowane diagramy i niezrozumiałe wywody teoretyczne wywołują w Tobie niechęć, koniecznie sięgnij po tę książkę! Dzięki niej poznasz metody analizy i projektowania obiektowego w nietypowy i ciekawy sposób, wykorzystujący najnowsze teorie skutecznego przekazywania wiedzy. Przeczytasz o tym, w jaki sposób warto gromadzić wymagania i oczekiwania użytkowników wobec projektowanego systemu, jak uwzględniać w projekcie postulowane zmiany i przeprowadzać proces analizy obiektowej. Nauczysz się stosować notację UML do przedstawiania struktury systemu i przetwarzanych przez niego danych. Dowiesz się także, jak testować projektowany system.

- Zasady i cele projektowania obiektowego
- Gromadzenie wymagań
- Przypadki użycia
- Analiza obiektowa
- Diagramy UML przedstawiające strukturę systemu
- Korzystanie ze wzorców projektowych
- Projektowanie architektury systemu
- Testowanie

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści (skrócony)

1	Dobrze zaprojektowane aplikacje są super. Tu zaczyna się wspaniałe oprogramowanie	31
2	Gromadzenie wymagań. Daj im to, czego chcą	83
3	Wymagania ulegają zmianom. Kocham cię, jesteś doskonały… A teraz — zmień się	137
4	Analiza. Zaczynamy używać naszych aplikacji w rzeczywistym świecie	169
5	Część 1. Dobry projekt = elastyczne oprogramowanie. Nic nie pozostaje wiecznie takie samo	221
	Przerywnik. Obiektowa katastrofa	245
	Część 2. Dobry projekt = elastyczne oprogramowanie. Zabierz swoje oprogramowanie na 30-minutowy trening	257
6	Rozwiązywanie naprawdę dużych problemów. „Nazywam się Art Vandelay... jestem Architektem”	301
7	Architektura. Porządkowanie chaosu	343
8	Zasady projektowania. Oryginalność jest przereklamowana	395
9	Iteracja i testowanie. Oprogramowanie jest wciąż przeznaczone dla klienta	441
10	Proces projektowania i analizy obiektowej. Scalając to wszystko w jedno	499
	Dodatek A Pozostałości	571
	Dodatek B Witamy w Obiektowie	589
	Skorowidz	603

Spis treści (szczegółowy)

Wprowadzenie

Twój mózg koncentruje się na analizie i projektowaniu obiektowym. Podczas gdy **Ty** starasz się czegoś **nauczyć**, Twój mózg robi Ci przysługę i dba o to, abyś przez przypadek **nie zapamiętał** zdobywanych informacji. Myśli sobie: „Lepiej zostawić trochę miejsca na bardziej istotne sprawy, na przykład jakich zwierząt unikać albo czy jazda na snowboardzie nago jest dobrym pomysłem”. W jaki zatem sposób możesz oszukać swój mózg i przekonać go, że Twoje życie zależy od znajomości analizy i projektowania obiektowego?

Dla kogo jest ta książka?	20
Wiemy, co sobie myślisz	21
Metapoznanie: myślenie o myśleniu	23
Zmuś swój mózg do posłuszeństwa	25
Ważne uwagi	26
Recenzenci techniczni	28
Podziękowania	29

Dobrze zaprojektowane aplikacje są super

1 Tu zaczyna się wspaniałe oprogramowanie

A zatem, w jaki sposób w praktyce pisze się wspaniałe oprogramowanie? Zawsze bardzo trudno jest określić, **od czego należy zacząć**. Czy aplikacja faktycznie **robi to, co powinna robić i czego od niej oczekujemy**? A co z takimi problemami jak powtarzający się kod — przecież to nie może być dobre ani właściwe rozwiązanie, prawda? Zazwyczaj trudno jest określić, które z wielu problemów należy rozwikłać w pierwszej kolejności, a jednocześnie mieć pewność, że podczas wprowadzania poprawek nie popsujemy innych fragmentów aplikacji. Bez obaw. Po zakończeniu lektury tego rozdziału będziesz już dokładnie **wiedział, jak pisać doskonałe oprogramowanie**, i pewnie podążał w kierunku trwałego poprawienia sposobu tworzenia programów. I w końcu zrozumiesz, dlaczego OOA&D to czteroliterowy skrót (pochodzący od angielskich słów: **Object-Oriented Analysis and Design**, analiza i projektowanie obiektowe), który Twoja matka **chciałaby**, byś poznał.

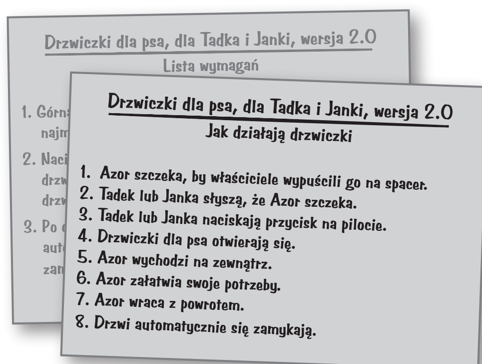
Rock-and-roll jest wieczny! 32
Nowa elegancka aplikacja Ryśka… 33
Co przede wszystkim zmieniłbyś w aplikacji Ryśka? 38
Doskonałe oprogramowanie… ma więcej niż jedną z wymienionych już cech 40
Wspaniałe oprogramowanie w trzech prostych krokach 43
W pierwszej kolejności skoncentruj się na funkcjonalności 48
Test 53
Szukamy problemów 55
Analiza metody search() 56
Stosuj proste zasady projektowania obiektowego 61
Projekt po raz pierwszy, projekt po raz drugi 66
Jak łatwo można wprowadzać zmiany w Twojej aplikacji? 68
Poddawaj hermetyzacji to, co się zmienia 71
Delegowanie 73
Nareszcie doskonałe oprogramowanie (jak na razie) 76
OOA&D ma na celu tworzenie wspaniałego oprogramowania, a nie dodanie Ci papierkowej roboty 79
Kluczowe zagadnienia 80

Gromadzenie wymagań

2 Daj im to, czego chcą

Każdy lubi zadowolonych klientów. Już wiesz, że pierwszy krok w pisaniu doskonałego oprogramowania polega na upewnieniu się, czego chce klient. Ale jak się dowiedzieć, **czego klient** oczekuje? Co więcej — skąd mieć pewność, że klient w ogóle **wie**, czego tak naprawdę chce? Właśnie wówczas na arenę wkraczają **„dobre wymagania"**. W tym rozdziale dowiesz się, w jaki sposób **zadowolić klientów**, upewniając się, że dostarczysz im właśnie to, czego chcą. Kiedy skończysz lekturę, wszystkie swoje projekty będziesz mógł opatrzyć etykietą „Satysfakcja gwarantowana" i posuniesz się o kolejny krok na drodze do tworzenia doskonałego oprogramowania... i to za każdym razem.

Nadszedł czas na kolejny pokaz Twych programistycznych umiejętności	84
Test programu	87
Nieprawidłowe zastosowanie (coś w tym stylu)	89
Czym jest wymaganie?	90
Tworzenie listy wymagań	92
Zaplanuj, co może się popsuć w systemie	96
Problemy w działaniu systemu są obsługiwane przez ścieżki alternatywne	98
(Ponowne) przedstawienie przypadku użycia	100
Jeden przypadek użycia, trzy części	102
Porównaj wymagania z przypadkami użycia	106
Twój system musi działać w praktyce	113
Poznajemy Szczęśliwą Ścieżkę	120
Przybornik projektanta	134



Drzwiczki dla psa oraz pilot stanowią elementy systemu, bądź też znajdują się wewnątrz niego.

Wymagania ulegają zmianom

3 Kocham cię, jesteś doskonały... A teraz — zmień się

Sądzisz, że dowiedziałeś się już wszystkiego o tym, czego chciał klient? Nie tak szybko... A zatem przeprowadziłeś rozmowy z klientem, zgromadziłeś wymagania, napisałeś przypadki użycia, napisałeś i dostarczyłeś klientowi odlotową aplikację. W końcu nadszedł czas na miłego, relaksującego drinka, nieprawdaż? Pewnie... aż do momentu gdy klient uzna, że tak naprawdę chce **czegoś innego niż to, co Ci powiedział**. Bardzo podoba mu się to, co zrobiłeś — poważnie! — jednak obecnie **nie jest już w pełni usatysfakcjonowany**. W rzeczywistym świecie **wymagania zawsze się zmieniają**; to Ty musisz sobie z tymi zmianami poradzić i pomimo nich zadbać o zadowolenie klienta.

Jesteś bohaterem! 138
Jesteś patałachem! 139
Jedyny pewnik analizy i projektowania obiektowego 141
Ścieżka oryginalna? Ścieżka alternatywna? Kto to wie? 146
Przypadki użycia muszą być zrozumiałe przede wszystkim dla Ciebie 148
Od startu do mety: jeden scenariusz 150
Wyznanie Ścieżki Alternatywnej 152
Uzupełnienie listy wymagań 156
Powielanie kodu jest bardzo złym pomysłem 164
Ostateczny test drzwiczek 166
Napisz swoją własną zasadę projektową! 167
Przybornik projektanta 168


```
public void pressButton() {
  System.out.println("Naciśnięto przycisk na pilocie...");
  if (door.isOpen()) {
    door.close();
  } else {
    door.open();

    final Timer timer = new Timer();
    timer.schedule(new TimerTask() {
      public void run() {
        door.close();
        timer.cancel();
      }
    }, 5000);
  }
```

class Remote { pressButton() }

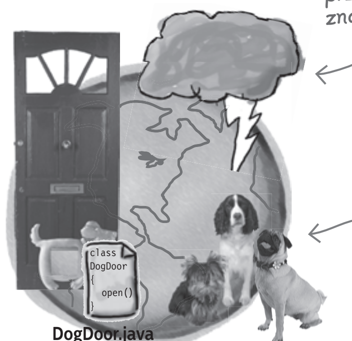
Remote.java

Analiza

4 Zaczynamy używać naszych aplikacji w rzeczywistym świecie

Czas zdać ostatnie egzaminy i zacząć stosować nasze aplikacje w rzeczywistym świecie. Twoje aplikacje muszą robić nieco więcej, niż jedynie działać prawidłowo na komputerze, którego używasz do ich tworzenia — komputerze o dużej mocy i doskonale skonfigurowanym; Twoje aplikacje muszą działać w takich warunkach, w jakich **rzeczywiści klienci będą ich używali**. W tym rozdziale zastanowimy się, jak zyskać pewność, że nasze aplikacje będą działać w **rzeczywistym kontekście**. Dowiesz się w nim, w jaki sposób analiza tekstowa może przekształcić stworzony wcześniej przypadek użycia w klasy i metody, które na pewno będą działać zgodnie z oczekiwaniami klienta. A kiedy skończysz lekturę tego rozdziału, także i Ty będziesz mógł powiedzieć: „Dokonałem tego! Moje oprogramowanie **jest gotowe do zastosowania w rzeczywistym świecie**!".

Jeden pies, dwa psy, trzy psy, cztery…	170
Twoje oprogramowanie ma kontekst	171
Określ przyczynę problemu	172
Zaplanuj rozwiązanie	173
Opowieść o dwóch programistach	180
Delegowanie w kodzie Szymka — analiza szczegółowa	184
Potęga aplikacji, których elementy są ze sobą luźno powiązane	186
Zwracaj uwagę na rzeczowniki występujące w przypadku użycia	191
Od dobrej analizy do dobrych klas…	204
Diagramy klas bez tajemnic	206
Diagramy klas to nie wszystko	211
Kluczowe zagadnienia	215



5 (część 1.)

Dobry projekt = elastyczne oprogramowanie

Nic nie pozostaje wiecznie takie samo

Zmiany są nieuniknione. Niezależnie od tego, jak bardzo podoba Ci się Twoje oprogramowanie w jego obecnej postaci, to najprawdopodobniej jutro zostanie ono **zmodyfikowane**. A im bardziej utrudnisz wprowadzanie modyfikacji w aplikacji, tym trudniej będzie Ci w przyszłości reagować na **zmiany potrzeb klienta**. W tym rozdziale mamy zamiar odwiedzić naszego starego znajomego oraz spróbować poprawić projekt istniejącego oprogramowania. Na tym przykładzie przekonamy się, jak **niewielkie zmiany mogą doprowadzić do poważnych problemów**. Prawdę mówiąc, jak się okaże, odkryte przez nas kłopoty będą tak poważne, że ich rozwiązanie będzie wymagało rozdziału składającego się aż z DWÓCH części!

Firma Gitary/Instrumenty Strunowe Ryśka rozwija się 222
Klasy abstrakcyjne 225
Diagramy klas bez tajemnic (ponownie) 230
Ściągawka z UML-a 231
Porady dotyczące problemów projektowych 237
Trzy kroki tworzenia wspaniałego oprogramowania (po raz kolejny) 239

5 (przerywnik)

OBIEKTOWA KATASTROFA!

Najbardziej popularny quiz w Obiektowie

Unikanie ryzyka	Sławni projektanci	Konstrukcje używane w kodzie	Utrzymanie i wielokrotne użycie	Nerwica oprogramowania
$100	$100	$100	$100	$100
$200	$200	$200	$200	$200
$300	$300	$300	$300	$300
$400	$400	$400	$400	$400

Dobry projekt = elastyczne oprogramowanie

5 (część 2.) Zabierz swoje oprogramowanie na 30-minutowy trening

Czy kiedykolwiek marzyłeś o tym, by być nieco bardziej elastycznym w działaniu? Jeśli kiedykolwiek wpadłeś w kłopoty podczas prób wprowadzania zmian w aplikacji, to zazwyczaj oznacza to, że Twoje oprogramowanie powinno być nieco **bardziej elastyczne i odporne**. Aby pomóc swojej aplikacji, będziesz musiał przeprowadzić odpowiednią analizę, zastanowić się nad niezbędnymi zmianami w projekcie i dowiedzieć się, w jaki sposób **rozluźnić zależności pomiędzy jej elementami**. I w końcu, w wielkim finale, przekonasz się, że **większa spójność może pomóc w rozwiązaniu problemu powiązań**. Brzmi interesująco? A zatem przewróć kartkę — przystępujemy do poprawiania nieelastycznej aplikacji.

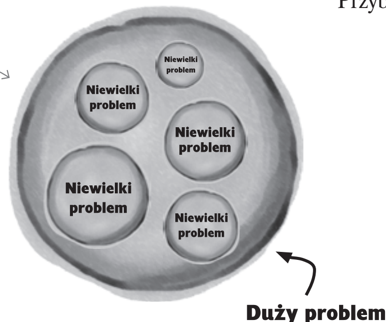
Wróćmy do aplikacji wyszukiwawczej Ryśka 258
Dokładniejsza analiza metody search() 261
Korzyści, jakie dała nam analiza 262
Dokładniejsza analiza klas instrumentów 265
Śmierć projektu (decyzja) 270
Zmieńmy złe decyzje projektowe na dobre 271
Zastosowanie „podwójnej hermetyzacji” w aplikacji Ryśka 273
Nigdy nie obawiaj się wprowadzania zmian 279
Elastyczna aplikacja Ryśka 282
Testowanie dobrze zaprojektowanej aplikacji Ryśka 285
Jak łatwo można zmodyfikować aplikację Ryśka? 289
Wielki konkurs łatwości modyfikacji 290
Spójna klasa realizuje jedną operację naprawdę dobrze 293
Przegląd zmian wprowadzanych w oprogramowaniu dla Ryśka 296
Doskonałe oprogramowanie to zazwyczaj takie, które jest „wystarczająco dobre” 298
Przybornik projektanta 300

Rozwiązywanie naprawdę dużych problemów

6 „Nazywam się Art Vandelay... jestem Architektem”

Nadszedł czas, by zbudować coś NAPRAWDĘ DUŻEGO. Czy jesteś gotów? Zdobyłeś już wiele narzędzi do swojego projektanckiego przybornika, jednak w jaki sposób z nich skorzystasz, kiedy będziesz musiał napisać coś **naprawdę dużego**? Cóż, może jeszcze nie zdajesz sobie z tego sprawy, ale **dysponujesz wszystkimi narzędziami, jakie mogą być potrzebne** do skutecznego rozwiązywania poważnych problemów. Niebawem poznasz kilka nowych narzędzi, takich jak **analiza dziedziny** oraz **diagramy przypadków użycia**, jednak nawet one bazują na wiadomościach, które już zdobyłeś, takich jak uważne słuchanie klienta oraz dokładne zrozumienie, co trzeba napisać, zanim jeszcze przystąpimy do faktycznego pisania kodu. Przygotuj się... nadszedł czas, byś sprawdził, jak sobie radzisz w roli architekta.

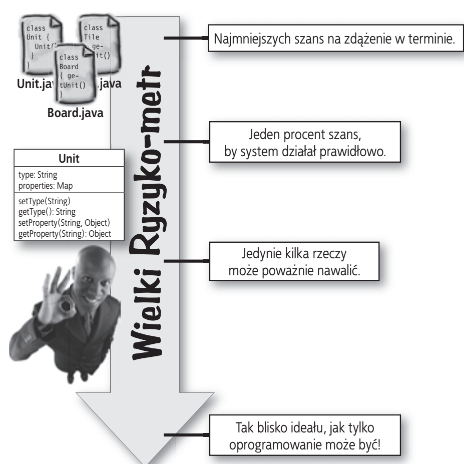
Rozwiązywanie dużych problemów	302
Wszystko zależy od sposobu spojrzenia na duży problem	303
Wymagania i przypadki użycia to dobry punkt wyjściowy…	308
Potrzebujemy znacznie więcej informacji	309
Określanie możliwości	312
Możliwość czy wymaganie	314
Przypadki użycia nie zawsze pomagają ujrzeć ogólny obraz tworzonego oprogramowania	316
Diagramy przypadków użycia	318
Mały aktor	323
Aktorzy to także ludzie (no dobrze… nie zawsze)	324
A zatem zabawmy się w analizę dziedziny!	329
Dziel i rządź	331
Nie zapominaj, kim tak naprawdę jest klient	335
Czym jest wzorzec projektowy?	337
Potęga OOA&D (i trochę zdrowego rozsądku)	340
Przybornik projektanta	342



Architektura

7 Porządkowanie chaosu

Gdzieś musisz zacząć, jednak uważaj, żeby wybrać właściwe „gdzieś”! Już wiesz, jak podzielić swoją aplikację na wiele małych problemów, jednak oznacza to tylko i wyłącznie tyle, iż obecnie nie masz jednego dużego, lecz **WIELE** małych problemów. W tym rozdziale spróbujemy pomóc Ci w określeniu, **gdzie należy zacząć**, i upewnimy się, że nie będziesz marnował czasu na zajmowanie się nie tym, co trzeba. Nadeszła pora, by pozbierać te wszystkie **drobne kawałki** na Twoim biurku i zastanowić się, jak można je przekształcić w **uporządkowaną i dobrze zaprojektowaną aplikację**. W tym czasie poznasz niesłychanie ważne „trzy P dotyczące architektury” i dowiesz się, że Risk to znacznie więcej niż jedynie słynna gra wojenna z lat 80.



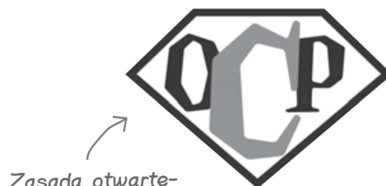
Czy czujesz się nieco przytłoczony? 344
Potrzebujemy architektury 346
Zacznijmy od funkcjonalności 349
Co ma znaczenie dla architektury 351
Trzy „P” dotyczące architektury 352
Wszystko sprowadza się do problemu ryzyka 358
Scenariusze pomagają zredukować ryzyko 361
Koncentruj się na jednej możliwości w danej chwili 369
Architektura jest strukturą Twojego projektu 371
Podobieństwa po raz kolejny 375
Analiza podobieństw: ścieżka do elastycznego oprogramowania 381
Co to znaczy? Zapytaj klienta 386
Zmniejszanie ryzyka pomaga pisać wspaniałe oprogramowanie 391
Kluczowe zagadnienia 392

Zasady projektowania

8 Oryginalność jest przereklamowana

Powielanie jest najlepszą formą unikania głupoty. Nic chyba nie daje większej satysfakcji niż opracowanie całkowicie nowego i oryginalnego rozwiązania problemu, który męczy nas od wielu dni... aż do czasu gdy okaże się, że ktoś **rozwikłał ten sam problem** już wcześniej, a co gorsza — zrobił to znacznie lepiej niż my. W tym rozdziale przyjrzymy się kilku **zasadom projektowania**, które udało się sformułować podczas tych wszystkich lat stosowania komputerów, i dowiemy się, w jaki sposób mogą one sprawić, że staniesz się lepszym programistą. Porzuć ambitne myśli o „zrobieniu tego lepiej" — lektura tego rozdziału udowodni Ci, jak pisać programy **sprytniej i szybciej**.

Zasada projektowania — w skrócie	396
Zasada otwarte-zamknięte	397
OCP, krok po kroku	399
Zasada nie powtarzaj się	402
Zasada DRY dotyczy obsługi jednego wymagania w jednym miejscu	404
Zasada jednej odpowiedzialności	410
Wykrywanie wielu odpowiedzialności	412
Przechodzenie od wielu do jednej odpowiedzialności	415
Zasada podstawienia Liskov	420
Studium błędnego sposobu korzystania z dziedziczenia	421
LSP ujawnia ukryte problemy związane ze strukturą dziedziczenia	422
Musi istnieć możliwość zastąpienia typu bazowego jego typem pochodnym	423
Naruszenia LSP sprawiają, że powstający kod staje się mylący	424
Deleguj funkcjonalność do innej klasy	426
Użyj kompozycji, by zebrać niezbędne zachowania z kilku innych klas	428
Agregacja — kompozycja bez nagłego zakończenia	432
Agregacja a kompozycja	433
Dziedziczenie jest jedynie jedną z możliwości	434
Kluczowe zagadnienia	437
Przybornik projektanta	438



Powtarzanie i testowanie

9 Oprogramowanie jest wciąż przeznaczone dla klienta

Czas pokazać klientowi, jak bardzo Ci na nim zależy. Nękają Cię szefowie? Klienci są zmartwieni? Udziałowcy wciąż zadają pytanie: „Czy wszystko będzie zrobione na czas?". Żadna ilość nawet wspaniale zaprojektowanego kodu nie zadowoli Twoich klientów; musisz **pokazać im coś działającego**. Teraz, kiedy dysponujesz już solidnym przybornikiem z narzędziami do programowania obiektowego, nadszedł czas, byś **udowodnił swoim klientom**, że pisane przez Ciebie oprogramowanie naprawdę działa. W tym rozdziale poznasz dwa sposoby pracy nad implementacją możliwości funkcjonalnych tworzonego oprogramowania — dzięki nim Twoi klienci poczują błogie ciepło, które sprawi, że powiedzą o Tobie: **„O tak, nie ma co do tego wątpliwości, jest właściwą osobą do napisania naszej aplikacji!".**

Twój przybornik narzędziowy powoli się wypełnia	442
Wspaniałe oprogramowanie tworzy się iteracyjnie	444
Schodzenie w głąb: dwie proste opcje	445
Programowanie w oparciu o możliwości	446
Programowanie w oparciu o przypadki użycia	447
Dwa podejścia do tworzenia oprogramowania	448
Analiza możliwości	452
Pisanie scenariuszy testowych	455
Programowanie w oparciu o testy	458
Podobieństwa po raz wtóry	460
Kładziemy nacisk na podobieństwa	464
Hermetyzujemy wszystko	466
Dopasuj testy do projektu	470
Testy bez tajemnic…	472
Udowodnij klientowi, że wszystko idzie dobrze	478
Jak dotąd używaliśmy programowania w oparciu o kontrakt	480
Tak naprawdę programowanie w oparciu o kontrakt dotyczy zaufania	481
Programowanie defensywne	482
Podziel swoją aplikację na mniejsze fragmenty funkcjonalności	491
Kluczowe zagadnienia	493
Przybornik projektanta	496

Unit
type: String
properties: Map
id: int
name: String
weapons: Weapon *
setType(String)
getType(): String
setProperty(String, Object)
getProperty(String): Object
getId(): int
setName(String)
getName(): String
addWeapon(Weapon)
getWeapons(): Weapons *

Wszystkie właściwości, które są wspólne dla wszystkich jednostek, zostały przedstawione w klasie Unit w formie odrębnych zmiennych.

Szymek doszedł do wniosku, że identyfikator jednostki będzie określany w konstruktorze klasy Unit, a zatem nie ma potrzeby definiowania odrębnej metody setId().

Dla każdej z nowych właściwości klasy Szymek zdefiniował odpowiednią parę metod.

Proces projektowania i analizy obiektowej

10 Scalając to wszystko w jedno

Czy dotarliśmy już do celu? Poświęciliśmy sporo czasu i wysiłku, by poznać wiele różnych sposobów pozwalających poprawić jakość tworzonego oprogramowania; teraz jednak nadeszła pora, by **połączyć i podsumować wszystkie zdobyte informacje**. Na to właśnie czekałeś: mamy zamiar zebrać **wszystko**, czego się nauczyłeś, i pokazać Ci, że wszystkie te informacje stanowią części jednego procesu, którego możesz wielokrotnie używać, by **tworzyć wspaniałe oprogramowanie**.

Tworzenie oprogramowania w stylu obiektowym 500
Trans-Obiektów 504
Mapa metra w Obiektowie 506
Lista możliwości 509
Przypadki użycia odpowiadają zastosowaniu, możliwości odpowiadają funkcjonalności 515
A teraz zacznij powtarzać te same czynności 519
Dokładniejsza analiza sposobu reprezentacji sieci metra 521
Używać klasy Line czy też nie używać... oto jest pytanie 530
Najważniejsze sprawy związane z klasą Subway 536
Ochrona własnych klas 539
Czas na przerwę 547
Wróćmy znowu do etapu określania wymagań 549
Koncentruj się na kodzie, a potem na klientach 551
Powtarzanie sprawia, że problemy stają się łatwiejsze 555
Jak wygląda trasa? 560
Samemu sprawdź Przewodnik Komunikacyjny po Obiektowie 564
Ktoś chętny na trzeci cykl prac? 567
Podróż jeszcze nie dobiegła końca… 569

Dodatek A Pozostałości

A

Dziesięć najważniejszych tematów (których nie poruszyliśmy)

Możesz nam wierzyć albo i nie, ale to jeszcze nie jest koniec. Owszem, wyobraź sobie, że nawet po przeczytaniu tych 600 stron wciąż możesz jeszcze znaleźć tematy, o których nawet nie wspomnieliśmy. Choć dziesięć zagadnień, jakie mamy zamiar przedstawić w tym dodatku, nie zasługuje na wiele więcej niż krótką wzmiankę, to jednak nie chcieliśmy, byś opuszczał Obiektów bez informacji na ich temat. Teraz będziesz miał nieco więcej tematów do rozmów podczas firmowej imprezy z okazji wygrania telewizyjnego quizu Obiektowa Katastrofa... poza tym któż, od czasu do czasu, nie kocha stymulujących rozmów o analizie i projektowaniu?

Kiedy już skończymy, pozostanie jeszcze... następny dodatek... no i oczywiście indeks, i może kilka reklam... ale później dotrzesz wreszcie do końca książki. Obiecujemy.

Nr 1. JEST i MA 572
Nr 2. Sposoby zapisu przypadków użycia 574
Nr 3. Antywzorce 577
Nr 4. Karty CRC 578
Nr 5. Metryki 580
Nr 6. Diagramy sekwencji 581
Nr 7. Diagramy stanu 582
Nr 8. Testowania jednostkowe 584
Nr 9. Standardy kodowania i czytelny kod 586
Nr 10. Refaktoryzacja 588

Antywzorce

Antywzorce są przeciwieństwem wzorców projektowych: stanowią one często stosowane ZŁE rozwiązania pewnych problemów. Powinniśmy być w stanie rozpoznawać te niebezpieczne pułapki i unikać ich.

Klasa: *DogDoor*

Opis: Reprezentuje faktyczne drzwiczki dla psa. Stanowi interfejs zapewniający możliwość korzystania z urządzeń sprzętowych kontrolujących działanie drzwiczek dla psa.

Odpowiedzialności:

Nazwa	Współpracownik
Otworzenie drzwiczek	
Zamknięcie drzwiczek	

Zwróć uwagę, by zapisać tu zarówno operacje, które dana klasa realizuje samodzielnie, jak i te, które wykonuje przy użyciu innych klas.

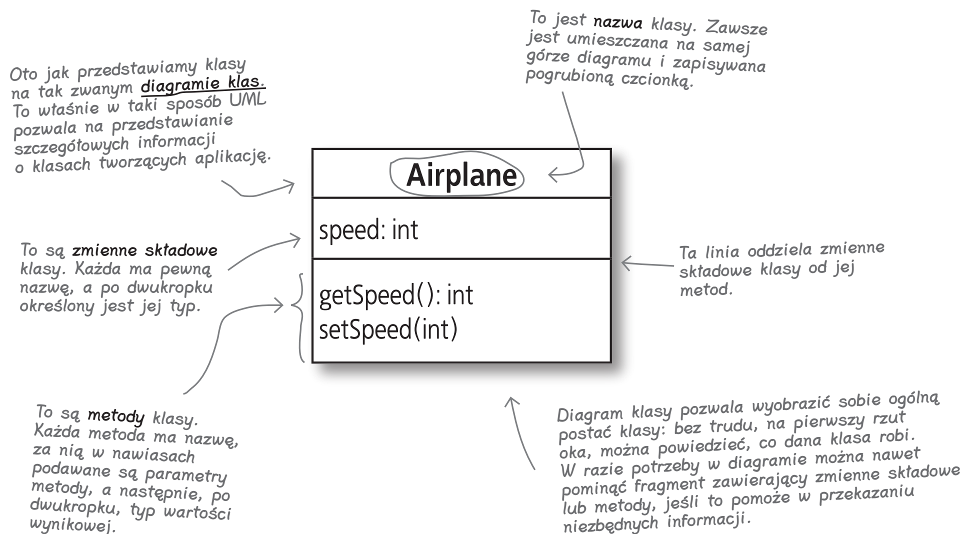
Do wykonania tych czynności nie są używane żadne inne obiekty.

Dodatek B Witamy w Obiektowie

B Stosowanie języka obiektowego

Przygotuj się na zagraniczną wycieczkę. Czas odwiedzić Obiektów — miejsce, gdzie **obiekty robią to, co powinny**, aplikacje są **dobrze hermetyzowane** (już wkrótce dowiesz się, co to znaczy), a projekty oprogramowania pozwalają na ich **wielokrotne stosowanie i rozbudowę**. Musisz jeszcze poznać kilka dodatkowych zagadnień i poszerzyć swoje **umiejętności językowe**. Nie przejmuj się jednak, nie zajmie Ci to wiele czasu i zanim się obejrzysz, już będziesz rozmawiał w języku obiektowym, jakbyś mieszkał w Obiektowie od wielu lat.

UML i diagramy klas	591
Dziedziczenie	593
Polimorfizm	595
Hermetyzacja	596
Kluczowe zagadnienia	600



S Skorowidz 603

1. Dobrze zaprojektowane aplikacje są super

Tu się zaczyna wspaniałe oprogramowanie

A zatem, w jaki sposób w praktyce pisze się wspaniałe oprogramowanie? Zawsze bardzo trudno jest określić, **od czego należy zacząć**. Czy aplikacja faktycznie **robi to, co powinna robić i czego od niej oczekujemy**? A co z takimi problemami jak powtarzający się kod — przecież to nie może być dobre ani właściwe rozwiązanie, prawda? Zazwyczaj trudno jest określić, które z wielu problemów należy rozwikłać w pierwszej kolejności, a jednocześnie mieć pewność, że podczas wprowadzania poprawek nie popsujemy innych fragmentów aplikacji. Bez obaw. Po zakończeniu lektury tego rozdziału będziesz już dokładnie **wiedział, jak pisać doskonałe oprogramowanie** i pewnie podążał w kierunku trwałego poprawienia sposobu tworzenia programów. I w końcu zrozumiesz, dlaczego **OOA&D** to czteroliterowy skrót (pochodzący od angielskich słów: **Object-Oriented Analysis and Design**, analiza i projektowanie obiektowe), który Twoja matka **chciałaby**, byś poznał.

Rock-and-roll jest wieczny!

Nie ma nic lepszego niż dźwięki doskonałej gitary w rękach świetnego muzyka, a firma Gitary Ryśka specjalizuje się w wyszukiwaniu doskonałych instrumentów dla wymagających i doświadczonych klientów.

Nie uwierzyłbyś, jaki mamy wybór gitar. Zapraszam, powiedz mi, jaka gitara Cię interesuje, a zapewniam, że znajdziemy instrument idealnie odpowiadający Twoim potrzebom i oczekiwaniom!

Poznaj Ryśka, pasjonata i miłośnika gitar, a jednocześnie właściciela ekskluzywnego sklepu z gitarami.

Właśnie kilka miesięcy temu Rysiek porzucił stosowany wcześniej „papierowy” system informacji o gitarach i podjął decyzję o przechowywaniu danych o stanie magazynu i transakcjach w systemie komputerowym. Wynajął w tym celu popularną firmę programistyczną SzybkoIKiepsko Sp. z o.o., która napisała mu odpowiednią aplikację. Rysiek poprosił ich nawet o napisanie nowego narzędzia — programu, który wspomagałby dobieranie gitar dla klientów.

Nowa elegancka aplikacja Ryśka...

Oto aplikacja, którą firma programistyczna napisała dla Ryśka… Jej zespół stworzył system, który całkowicie zastępuje papierowe notatki spisywane przez Ryśka wcześniej i który pomaga mu w odnajdywaniu gitar doskonale spełniających oczekiwania klientów. Oto diagram UML klas, który programiści firmy przedstawili Ryśkowi, by pokazać, co dla niego zrobili:

Każda gitara znajdująca się w magazynie sklepu Ryśka jest reprezentowana przez obiekt tej klasy.

Oto cały magazyn sklepu Ryśka, który jednocześnie zapewnia mu możliwość poszukiwania gitar.

Magazyn zawiera listę wszystkich gitar, jakimi Rysiek aktualnie dysponuje.

Guitar
serialNumber: String price: double builder: String model: String type: String backWood: String topWood: String
getSerialNumber(): String getPrice(): double setPrice(float) getBuilder(): String getModel(): String getType(): String getBackWood(): String getTopWood(): String

Inventory
guitars: List
addGuitar(String, double, String, String, String, String, String) getGuitar(String): Guitar search(Guitar): Guitar

Oto zmienne dostępne w klasie Guitar.

Oto metody klasy Guitar.

Ta metoda pobiera numer seryjny gitary i zwraca reprezentujący ją obiekt.

Ta metoda służy do wyszukiwania gitar; pobiera ona obiekt stanowiący reprezentację gitary idealnie spełniającej wymagania klienta i zwraca wyszukany w magazynie obiekt gitary, która pasuje do zadanych kryteriów.

Do tej metody przekazywane są szczegółowe informacje o gitarze; metoda ta tworzy odpowiedni obiekt Guitar i zapisuje go w magazynie Ryśka.

Rysiek zdecydował, że do informacji opisujących każdą gitarę należą: numer seryjny, cena, producent oraz model, typ (elektryczna bądź akustyczna) oraz gatunek drewna, z jakiego instrument został wykonany.

Przygotowaliśmy dla Ciebie kilka interesujących i ciekawych smakołyków, znajdziesz je w dodatku B. Jeśli po raz pierwszy stykasz się z zagadnieniami dotyczącymi projektowania obiektowego lub języka UML, to zajrzyj tam koniecznie.

Nowy w Obiektowie?

Jeśli nie spotkałeś się wcześniej z projektowaniem obiektowym, nie słyszałeś o diagramach UML bądź też nie jesteś pewny, czy dobrze rozumiesz znaczenie diagramów przedstawionych na powyższym rysunku, nie przejmuj się! Przygotowaliśmy specjalny pakiet ratunkowy „Witamy w Obiektowie”, który pomoże Ci wszystko zrozumieć. Zajrzyj na sam koniec książki i przeczytaj dodatek B — gwarantujemy Ci, że nie będziesz żałował. Kiedy skończysz, ponownie przeczytaj tę stronę, a na pewno wszystko nabierze zupełnie nowego sensu.

Oto jak wygląda kod Guitar.java

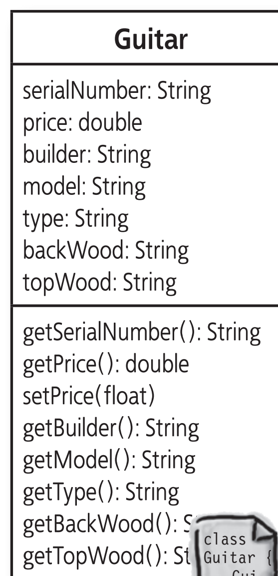
Na poprzedniej stronie przedstawiliśmy diagramy klas tworzących aplikację Ryśka; teraz nadszedł czas, abyś zobaczył, jak wygląda faktyczny kod źródłowy klas **`Guitar`** i **`Inventory`** (umieszczony odpowiednio w plikach **`Guitar.java`** oraz **`Inventory.java`**).

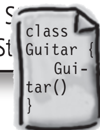
```
public class Guitar {
  private String serialNumber, builder, model, type, backWood, topWood;
  private double price;
  public Guitar(String serialNumber, double price,
                String builder, String model, String type,
                String backWood, String topWood) {
    this.serialNumber = serialNumber;
    this.price = price;
    this.builder = builder;
    this.model = model;
    this.type = type;
    this.backWood = backWood;
    this.topWood = topWood;
  }

  public String getSerialNumber() {
    return serialNumber;
  }

  public double getPrice() {
    return price;
  }
  public void setPrice(float newPrice) {
    this.price = newPrice;
  }
  public String getBuilder() {
    return builder;
  }
  public String getModel() {
    return model;
  }
  public String getType() {
    return type;
  }
  public String getBackWood() {
    return backWood;
  }
  public String getTopWood() {
    return topWood;
  }
}
```

To wszystko są właściwości, które widzieliśmy już wcześniej na diagramie klasy Guitar.

Na diagramach UML nie są umieszczane konstruktory klas; konstruktor klasy Guitar robi dokładnie to, czego można od niego oczekiwać: określa początkowe wartości właściwości nowego obiektu Guitar.

Łatwo zauważyć, jak diagram klas odpowiada metodom, które możemy znaleźć w kodzie źródłowym klasy Guitar





Guitar.java

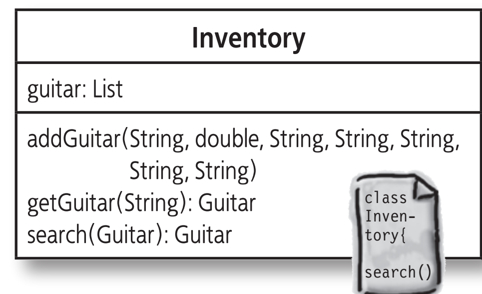
Plik Inventory.java...

```
public class Inventory {
  private List guitars;
  public Inventory() {
    guitars = new LinkedList();
  }
  public void addGuitar(String serialNumber, double price,
                        String builder, String model,
                        String type, String backWood, String topWood)
{
    Guitar guitar = new Guitar(serialNumber, price, builder,
                               model, type, backWood, topWood);
    guitars.add(guitar);
  }

  public Guitar getGuitar(String serialNumber) {
    for (Iterator i = guitars.iterator(); i.hasNext(); ) {
      Guitar guitar = (Guitar)i.next();
      if (guitar.getSerialNumber().equals(serialNumber)) {
        return guitar;
      }
    }
    return null;
  }
  public Guitar search(Guitar searchGuitar) {
    for (Iterator i = guitars.iterator(); i.hasNext(); ) {
      Guitar guitar = (Guitar)i.next();
      // Ignorujemy numer seryjny bo jest unikalny
      // Ignorujemy cenę gdyż jest unikalna
      String builder = searchGuitar.getBuilder();
      if ((builder != null) && (!builder.equals("")) &&
          (!builder.equals(guitar.getBuilder()))
        continue;
      String model = searchGuitar.getModel();
      if ((model != null) && (!model.equals("")) &&
          (!model.equals(guitar.getModel()))
        continue;
      String type = searchGuitar.getType();
      if ((type != null) && (!type.equals("")) &&
          (!type.equals(guitar.getType()))
        continue;
      String backWood = searchGuitar.getBackWood();
      if ((backWood != null) && (!backWood.equals("")) &&
          (!backWood.equals(guitar.getBackWood()))
        continue;
      String topWood = searchGuitar.getTopWood();
      if ((topWood != null) && (!topWood.equals("")) &&
          (!topWood.equals(guitar.getTopWood()))
        continue;
      return guitar;
    }
    return null;
  }
}
```

Pamiętaj, że usunęliśmy instrukcje import, by zaoszczędzić nieco miejsca.

Metoda addGuitar() pobiera wszystkie informacje konieczne do utworzenia nowego obiektu typu Guitar i dodaje go do magazynu.

Ta metoda jest nieco bardziej skomplikowana... porównuje ona wszystkie właściwości obiektu Guitar, przekazanego w jej wywołaniu, z właściwościami wszystkich obiektów tego typu, dostępnych w magazynie Ryśka.



Inventory.java

Wkrótce jednak okazało się, że Rysiek zaczął tracić klientów...

Wygląda na to, że niezależnie od tego, kim jest klient i jakiej gitary szuka, nowy program wyszukujący gitary prawie nigdy nie jest w stanie dopasować odpowiedniego instrumentu. Jednak Rysiek doskonale wie, że posiada gitarę, która na pewno spodobałaby się danemu klientowi… Zatem gdzie tkwi problem?

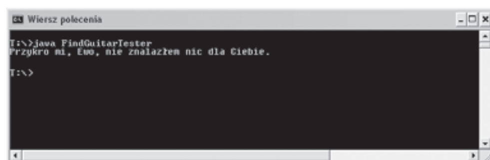
Program FindGuitarTester.java symuluje typowy dzień pracy Ryśka... Zjawia się klient, mówi Ryśkowi, jaka gitara by go interesowała, a Rysiek przeszukuje swój magazyn.

```
public class FindGuitarTester {

  public static void main(String[] args) {
    // Inicjalizacja zawartości magazynu gitar Ryśka
    Inventory inventory = new Inventory();
    initializeInventory(inventory);

    Guitar whatEveLikes = new Guitar("", 0, "fender", "Stratocastor",
                                   "elektryczna", "olcha", "olcha");
    Guitar guitar = inventory.search(whatEveLikes);
if (guitar != null) {
      System.out.println("Ewo, może spodoba Ci się gitara " +
        guitar.getBuilder() + " model " + guitar.getModel() +  " " +
        guitar.getType() + " :\n   " +
        guitar.getBackWood() + " - tył i boki,\n   " +
        guitar.getTopWood() + " - góra.\nMożesz ją mieć za " +
        guitar.getPrice() + "PLN!");
    } else {
      System.out.println("Przykro mi, Ewo, nie znalazłem nic dla Ciebie.");
    }
  }

  private static void initializeInventory(Inventory inventory) {
    ...
  }
}
```

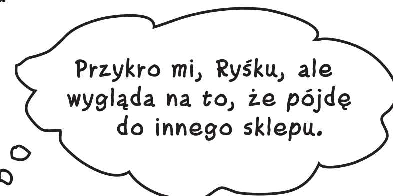
Ewa szuka gitary „Strat" firmy Fender, wykonanej w całości z drewna olchowego.

```
class
FingGui-
tar {
  main()
}
```

FindGuitarTester.java



Oto co się dzieje, gdy Ewa wejdzie do sklepu Ryśka, a ten spróbuje odnaleźć gitarę spełniającą jej oczekiwania.



Ale przecież ja wiem, że mam świetną gitarę Stratocastor firmy Fender. Jest właśnie tutaj:

```
inventory.addGuitar("V95693", 1499.95,
"Fender", "Stratocastor",
"elektryczna", "olcha", "olcha");
```

Wydaje się, że specyfikacja gitary dokładnie odpowiada oczekiwaniom Ewy... Zatem co się stało?

Oto fragment kodu inicjalizującego zawartość magazynu Ryśka. Wygląda na to, że Rysiek faktycznie ma gitarę, która spełnia oczekiwania Ewy.

Zaostrz ołówek

W jaki sposób zmieniłbyś projekt aplikacji Ryśka?

Przeanalizuj kody aplikacji Ryśka, przedstawione na trzech poprzednich stronach, oraz wyniki wykonanego testu. Jakie problemy udało Ci się zauważyć? Co byś zmienił? Poniżej zapisz **PIERWSZĄ** rzecz, jaką chciałbyś poprawić w aplikacji Ryśka.

Co przede wszystkim zmieniłbyś w aplikacji Ryśka?

Oczywiste jest, że w aplikacji Ryśka występuje jakiś problem; jednak znacznie trudniej jest określić, od czego należy zacząć wprowadzanie poprawek. I wygląda na to, że istnieje wiele różnych opinii na ten temat:

Spójrzcie na te wszystkie łańcuchy znaków! To jest po prostu okropne... czy zamiast nich nie możemy użyć stałych lub obiektów?

Hm... w swoich notatkach właściciel firmy wyraźnie pisze, że chciałby, żeby klienci mieli możliwość wyboru wielu instrumentów. Czy zatem metoda search() nie powinna zwracać listy obiektów?

Guitar
serialNumber: String price: double builder: String model: String type: String backWood: String topWood: String
getSerialNumber(): String getPrice(): double setPrice(float) getBuilder(): String getModel(): String getType(): String getBackWood(): String getTopWood(): String

Inventory
guitar: List
addGuitar(String, double, String, String, St String, String) getGuitar(String): Guitar search(Guitar): Guitar

Ten projektant musi być naprawdę kiepski. Klasy Guitar oraz Inventory w zbyt dużym stopniu zależą od siebie nawzajem. Nie potrafię sobie wyobrazić, by to była architektura, na której mógłby bazować rozwój aplikacji. Musimy ją jakoś zmienić.

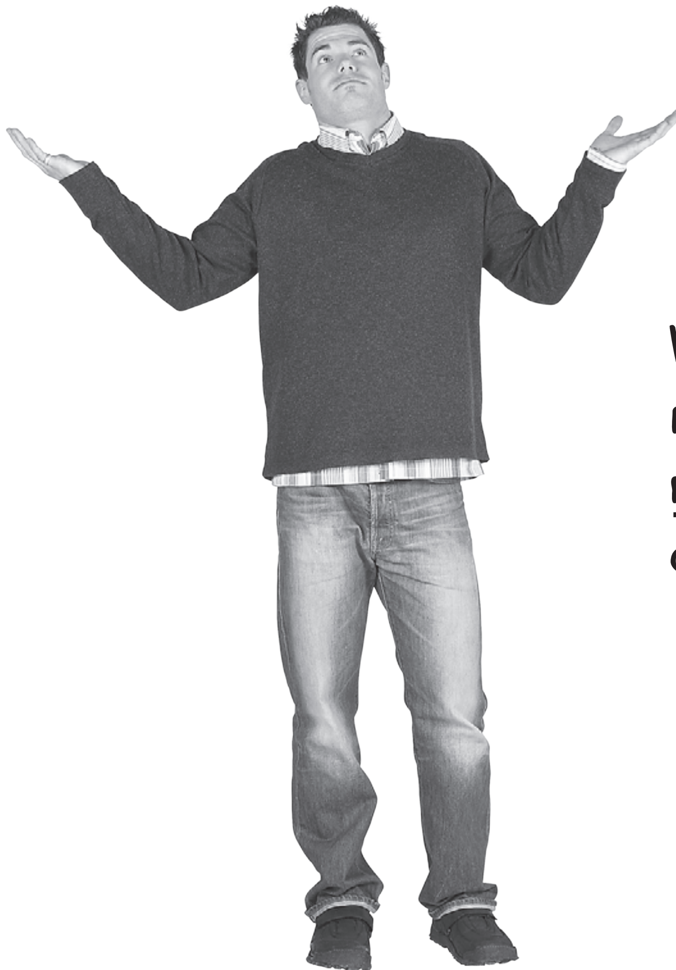
Jerzy zajmuje się programowaniem stosunkowo krótko, jednak szczerze i głęboko wierzy w słuszność pisania kodu obiektowego.

Franek zajmuje się programowaniem już od jakiegoś czasu i naprawdę dobrze zna zasady projektowania obiektowego i wzorce projektowe.

Julka zyskała reputację osoby, która zawsze oferuje klientom to, czego chcą.

Co zrobiłbyś w pierwszej kolejności?

Niby skąd mam wiedzieć, od czego należy zacząć? Mam wrażenie, że ilekroć zaczynam pracę nad nowym projektem, każdy ma inne zdanie odnośnie tego, co należy zrobić w pierwszej kolejności. Czasami zrobię coś dobrze, lecz czasami kończy się na tym, że muszę przerobić całą aplikację od początku, bo zacząłem w złym miejscu. **A ja chcę jedynie pisać świetne oprogramowanie!** A zatem, od czego powinienem zacząć pisanie aplikacji dla Ryśka?



W jaki sposób można za każdym razem pisać dobre oprogramowanie?

Chwileczkę... nie lubię się wtrącać, ale co to właściwie oznacza „wspaniałe oprogramowanie”? To jakieś tajemnicze hasło, które rzuca się w rozmowach, by zrobić odpowiednie wrażenie?

Dobre pytanie... na które można podać wiele różnych odpowiedzi:

Programista dbający o dobro klienta odpowie:

„Doskonałe oprogramowanie zawsze robi to, czego chce klient. A zatem, nawet jeśli klient wymyśli nowy sposób zastosowania takiego oprogramowania, to nie będzie ono działać niewłaściwie ani zwracać nieoczekiwanych wyników”.

Podstawą tego podejścia jest zwrócenie największej uwagi na to, by użytkownik był zadowolony z działania aplikacji.

Programista obiektowy odpowie:

„Doskonałe oprogramowanie to kod napisany obiektowo. Jest to zatem kod, w którym nie ma żadnych powtórzeń oraz w którym obiekty w bardzo dużym stopniu kontrolują swoje własne zachowanie. Takie oprogramowanie także łatwo rozbudować, gdyż jego projekt jest solidny i elastyczny”.

Dobrzy programiści obiektowi zawsze poszukują sposobów na to, by ich kod był jak najbardziej elastyczny.

To podejście, koncentrujące uwagę na zagadnieniach projektu, pozwala na tworzenie kodu zoptymalizowanego pod kątem przyszłego rozwoju aplikacji i wielokrotnego używania jej elementów. Wykorzystuje ono wzorce projektowe oraz wielokrotnie sprawdzone techniki projektowania i programowania obiektowego.

Nie jesteś całkiem pewny, co to wszystko znaczy? Nie martw się... wszystkiego dowiesz się w kolejnych rozdziałach.

Programista będący autorytetem w dziedzinie projektowania odpowie:

„Oprogramowanie będzie doskonałe, jeśli podczas jego tworzenia zastosujemy wypróbowane i sprawdzone wzorce projektowe i zasady projektowania. Zachowałeś luźne powiązania pomiędzy obiektami i sprawiłeś, by kod był otwarty na rozszerzanie, lecz zamknięty na modyfikacje. To także powoduje, że łatwiejsze będzie wielokrotne wykorzystanie kodu, a dzięki temu nie będziesz go musiał przerabiać, chcąc wykorzystać fragmenty aplikacji w innych projektach”.

Zaostrz ołówek

A co ***dla Ciebie*** znaczy termin „doskonałe oprogramowanie”?

Dowiedziałeś się już, co termin „doskonałe oprogramowanie” oznacza dla kilku różnych typów programistów. Zatem kto z nich ma rację? A może masz swoją własną definicję, która określa, co sprawia, że tworzona aplikacja będzie doskonała? Jeśli tak, to nadszedł czas, byś napisał własną definicję doskonałego oprogramowania:

Tu zapisz swoje personalia...

...a tu podaj co według Ciebie oznacza termin „doskonałe oprogramowanie”:

____________________ **uważa, że:**

„____

____

____

____

____”

Doskonałe oprogramowanie...
ma więcej niż jedną z wymienianych już cech

Nie sposób określić, czym jest „doskonałe oprogramowanie”, przy użyciu jednej prostej definicji. W rzeczywistości wszystkie stwierdzenia różnych programistów dotyczące dobrego oprogramowania, podane na stronie 40, określają cechy, dzięki którym oprogramowanie można uznać za „doskonałe”.

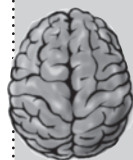
Przede wszystkim doskonałe oprogramowanie musi spełniać wymagania i oczekiwania klienta. Oprogramowanie musi robić to, czego klient od niego oczekuje.

Podbij swoich klientów.

Klienci uznają Twoje oprogramowanie za doskonałe, jeśli będzie ono robić to, czego od niego oczekują.

Tworzenie oprogramowania, które działa we właściwy sposób, jest czymś wspaniałym. Co się jednak stanie w sytuacjach, kiedy takie oprogramowanie trzeba będzie rozbudować lub zastosować jego część w innym projekcie? Napisanie kodu, który działa zgodnie z oczekiwaniami klienta, to za mało; równie ważne jest to, by przeszedł on próbę czasu.

Poza tym, doskonałe oprogramowanie powinno być dobrze zaprojektowane, poprawnie napisane oraz musi zapewniać łatwość utrzymania, wielokrotnego stosowania i rozszerzania.



Niech Twój kod będzie tak inteligentny jak Ty sam.

Zarówno Ty, jak i Twoi współpracownicy sami uznacie, że oprogramowanie jest doskonałe, jeśli jego utrzymanie, rozszerzanie i wielokrotne stosowanie nie będą przysparzać większych problemów.

Wspaniałe oprogramowanie w trzech prostych krokach

Obecnie może Ci się wydawać, że to wcale nie jest takie łatwe. Jednak pokażemy, że analiza i projektowanie obiektowe, wraz z kilkoma prostymi zasadami, mogą na zawsze zmienić postać tworzonego przez Ciebie oprogramowania.

1. Upewnij się, że oprogramowanie robi to, czego oczekuje klient.

W tym kroku koncentrujemy uwagę na kliencie. W PIERWSZEJ KOLEJNOŚCI powinieneś zapewnić, że aplikacja będzie robić to, czego klient od niej oczekuje. Na tym etapie prac dużą rolę odgrywa przygotowanie wymagań i przeprowadzenie odpowiedniej analizy.

2. Zastosuj proste zasady projektowania obiektowego, by poprawić elastyczność oprogramowania.

Kiedy oprogramowanie będzie już działać, możesz odszukać w nim i usunąć powtarzające się fragmenty kodu oraz upewnić się, że zastosowałeś dobre techniki projektowania obiektowego.

3. Staraj się, by projekt oprogramowania zapewniał łatwość jego utrzymania i pozwalał na jego wielokrotne stosowanie.

Czy uzyskałeś dobrą aplikację obiektową, która robi to, co powinna? Nadszedł czas, by zastosować wzorce i zasady, dzięki którym upewnisz się, że Twoje oprogramowanie jest odpowiednio przygotowane do wieloletniego użytkowania.

Pamiętasz Ryśka? Pamiętasz klientów, których stracił?

Wypróbujmy nasze pomysły i założenie dotyczące tworzenia doskonałego oprogramowania i przekonajmy się, czy nadają się one do zastosowania w realnym świecie. Rysiek dostał program do wyszukiwania gitar, który nie działa poprawnie, i to Twoim zadaniem będzie jego poprawienie i dołożenie wszelkich starań, by stworzyć naprawdę wspaniałą aplikację. Przyjrzyjmy się jeszcze raz programowi, jakim obecnie dysponuje Rysiek, i zobaczmy, w czym tkwi problem:

Oto nasz program testowy, który uwidocznił problemy w działaniu narzędzia do wyszukiwania gitar pasujących do kryteriów podanych przez użytkownika.

```
public class FindGuitarTester {
  public static void main(String[] args) {
    // Inicjalizacja magazynu gitar Ryśka
    Inventory inventory = new Inventory();
    initializeInventory(inventory);
    Guitar whatEveLikes = new Guitar("", 0, "fender", "Stratocastor",
                                    "elektryczna", "olcha", "olcha");
    Guitar guitar = inventory.search(whatEveLikes);
    if (guitar != null) {
```

Aplikacja Ryśka powinna dopasować te wymagania...

...do tej gitary znajdującej się w magazynie.

```
inventory.addGuitar("V95693",
1499.95, "Fender", "Stratocastor",
"elektryczna", "olcha", "olcha");
```

```
class
FingGui-
tar {
  main()
}
```

FindGuitarTester.java

A zatem wykonajmy nasze trzy podane wcześniej kroki:

1. Upewnij się, że oprogramowanie robi to, czego oczekuje klient.

Pamiętaj, że musimy zacząć od zapewnienia, by aplikacja robiła to, czego oczekuje Rysiek... a nie ma wątpliwości, że obecna aplikacja nie spełnia tego warunku.

2. Zastosuj proste zasady projektowania obiektowego, by poprawić elastyczność oprogramowania.

Na razie nie zawracaj sobie głowy stosowaniem w aplikacji wzorców ani obiektowych technik programowania... skoncentruj się na tym, by działała tak, jak powinna.

3. Staraj się, by projekt oprogramowania zapewniał łatwość jego utrzymania i pozwalał na jego wielokrotne stosowanie.

Skoro zaczynamy od funkcjonalności, to sprawdźmy, co się dzieje z tą niedziałającą metodą search(). Wygląda na to, że w magazynie Ryśka nazwa producenta jest zapisana małymi literami, a w wymaganiach klienta nazwa „Fender" zaczyna się z wielkiej litery. A zatem w metodzie search() musimy zastosować wyszukiwanie, które nie będzie uwzględniać wielkości liter.

Skorzystajmy z drobnej pomocy naszych znajomych programistów.

Franek: Oczywiście, to by rozwiązało bieżące problemy Ryśka, niemniej jednak uważam, że istnieje lepszy sposób zapewnienia poprawnego działania aplikacji niż wywoływanie metody `toLowerCase()` we wszystkich miejscach, gdzie są porównywane łańcuchy znaków.

Jerzy: Właśnie, o tym samym pomyślałem. Wydaje mi się, że to całe porównywanie łańcuchów znaków to nie jest najlepszy pomysł. Czy nie moglibyśmy zastosować jakichś stałych lub jakiegoś typu wyliczeniowego do określania producentów gitar oraz gatunków drewna?

Julka: Panowie, wybiegacie myślami *zdecydowanie* zbyt daleko. Krok 1. miał polegać na poprawieniu aplikacji w taki sposób, by robiła to, czego od niej oczekuje klient. Sądziłam, że na tym etapie prac nie będziemy zajmowali się zagadnieniami związanymi z projektem.

Franek: Cóż, to prawda; rozumiem, że mamy się skoncentrować na kliencie. Ale możemy przynajmniej zastanowić się, *jak* inteligentnie rozwiązać aktualnie występujące problemy, nieprawdaż? Chodzi mi o to, by nie tworzyć kolejnych problemów, które w przyszłości znowu będziemy musieli rozwiązywać.

Julka: Hmm… tak, całkiem słusznie. Na pewno nie chcemy, by zaproponowane przez nas rozwiązanie bieżących problemów powodowało pojawienie się nowych problemów projektowych. Jednak pomimo to na razie nie będziemy zajmowali się innymi fragmentami aplikacji, dobra?

Franek: Dobra. Możemy ograniczyć się do usunięcia tych wszystkich łańcuchów znaków, porównywania łańcuchów znaków i problemów związanych z wielkościami liter.

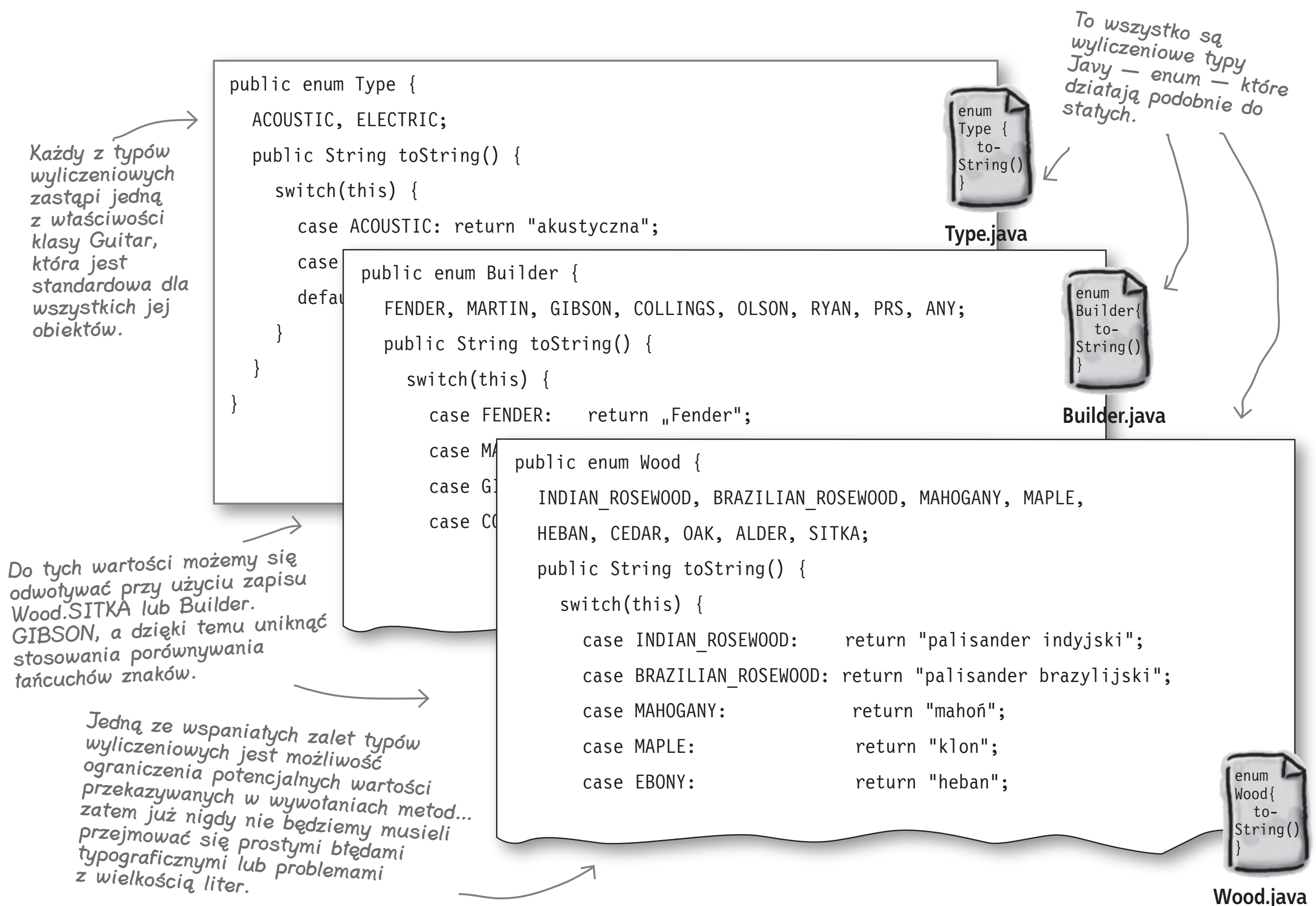
Jerzy: Właśnie. Jeśli zastosujemy typy wyliczeniowe, to zyskamy pewność, że podczas określania producenta gitary, typu drewna oraz rodzaju instrumentu będą używane wyłącznie prawidłowe wartości. W ten sposób zagwarantujemy, że Rysiek będzie mógł znajdować gitary pasujące do wymagań i oczekiwań klientów.

Julka: A jednocześnie poprawimy nieco projekt samej aplikacji… super! A zatem — do roboty, panowie.

Rozwiązując istniejące problemy, nie twórz nowych.

Eliminacja porównywania łańcuchów znaków

Pierwszą poprawką, jaką możemy wprowadzić w aplikacji Ryśka, jest usunięcie porównywania łańcuchów znaków. Choć można by zastosować metodę **`toLowerCase()`**, by rozwiązać problem z porównywaniem liter różnych wielkości, to jednak postaramy się w ogóle wyeliminować porównywanie łańcuchów znaków:



Nie ma niemądrych pytań

P: Nigdy wcześniej nie spotkałem się w języku Java ze słowem kluczowym enum. Co to takiego?

O: Pozwala ono na definiowanie *typów wyliczeniowych*. Ten typ danych jest dostępny także w języku C, C++ oraz w Javie w wersji 5.0 i kolejnych. Co więcej, pojawi się także w wersji 6.0 języka Perl.

Typy wyliczeniowe pozwalają na podanie nazwy typu, na przykład **`Wood`**, oraz listy wartości, jakie mogą być stosowane w ramach tego typu (na przykład: **`SITKA`**, **`ALDER`** bądź **`CEDAR`**). A zatem odwołanie do konkretnej wartości ma postać: **`Wood.SITKA`**.

P: A dlaczego typy wyliczeniowe przydadzą się nam w aplikacji Ryśka?

```
public class FindGuitarTester {
  public static void main(String[] args) {
    // Inicjalizacja zawartości magazynu gitar Ryśka
    Inventory inventory = new Inventory();
    initializeInventory(inventory);
    Guitar whatEveLikes = new Guitar("", 0, Builder.FENDER,
      "Stratocastor", Type.ELECTRIC, Wood.ALDER, Wood.ALDER);
    Guitar guitar = inventory.search(whatEveLikes);
    if (guitar != null) {
```

Te wszystkie łańcuchy znaków mogliśmy już zastąpić wartościami typów wyliczeniowych.

Jedyny łańcuch znaków, jaki nam pozostał i jaki sprawdzamy, określa model gitary; zostawiliśmy go, gdyż zbiór dostępnych modeli nie jest ograniczony, w odróżnieniu od producentów gitar oraz gatunków drewna używanych do ich wytwarzania.

FindGuitarTester.java

```
  public List search(Guitar searchGuitar) {
   for (Iterator i = guitars.iterator(); i.hasNext(); ) {
     Guitar guitar = (Guitar)i.next();
     // Ignorujemy numer seryjny, bo jest unikalny
     // Ignorujemy cenę, gdyż jest unikalna
     if (searchGuitar.getBuilder() != guitar.getBuilder())
       continue;
     String model = searchGuitar.getModel().toLowerCase();
     if ((model != null) && (!model.equals("")) &&
         (!model.equals(guitar.getModel().toLowerCase())))
       continue;
     if (searchGuitar.getType() != guitar.getType())
       continue;
     if (searchGuitar.getBackWood() != guitar.getBackWood())
       continue;
     if (searchGuitar.getTopWood() != guitar.getTopWood())
       continue;
     return guitar;
   }
   return null;
  }
```

Wygląda na to, że nic się nie zmieniło, jednak dzięki zastosowaniu typów wyliczeniowych nie musimy już martwić się, że metoda zwróci niewłaściwe wyniki ze względu na różnice w wielkości liter użytych w obu łańcuchach.

Jedyną właściwością, w jakiej musimy zadbać o wielkość liter, jest nazwa modelu gitary — ona wciąż jest zwyczajnym łańcuchem znaków.

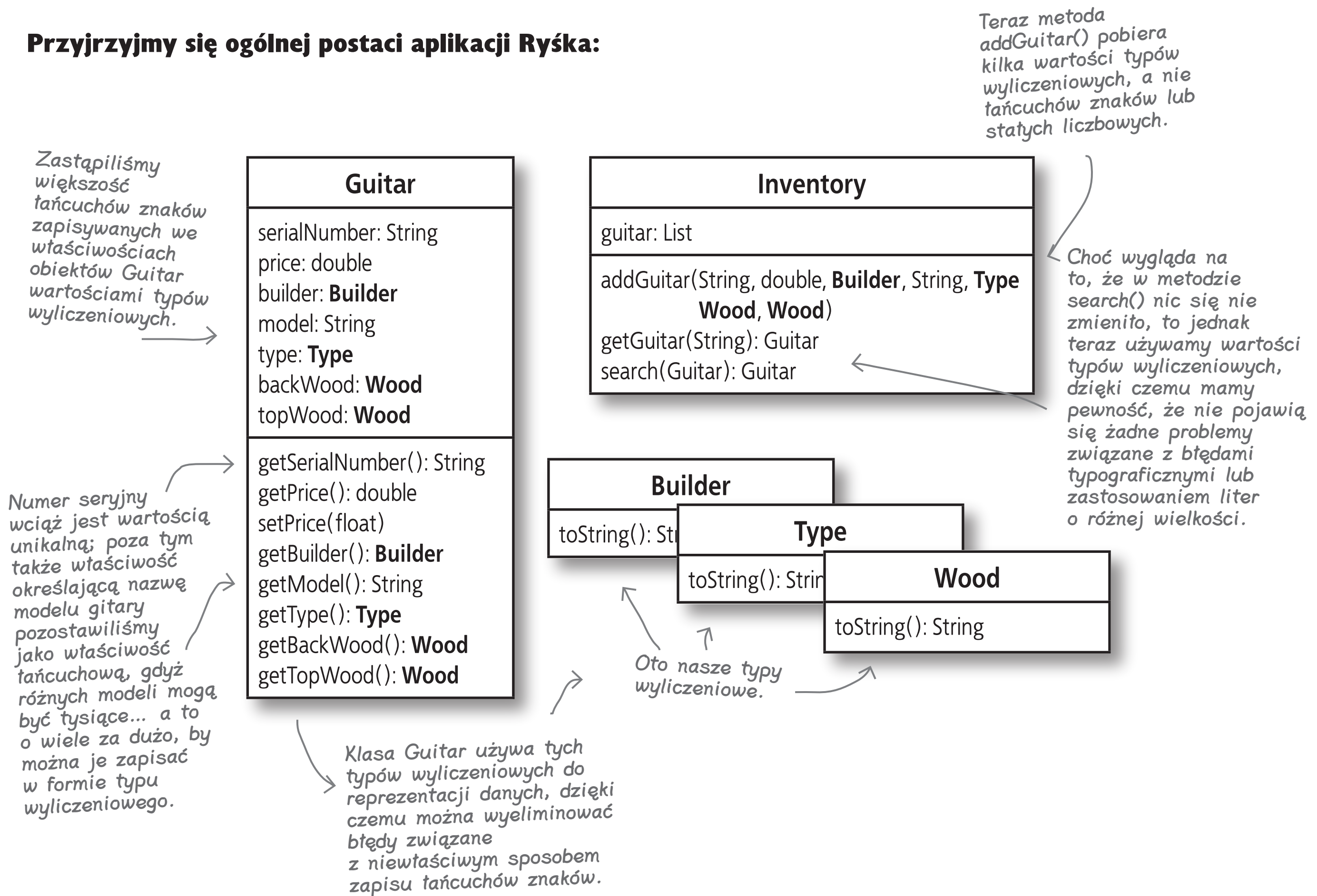
Inventory.java

O.: Bardzo dużą zaletą typów wyliczeniowych jest to, iż zabezpieczają one metody, w których są używane przed przekazaniem wartości niezdefiniowanych w danym typie. A zatem, jeśli tylko wartość typu wyliczeniowego zostanie błędnie zapisana, kompilator wygeneruje błąd. Jak widać, typy wyliczeniowe są doskonałym sposobem nie tylko na zapewnienie bezpieczeństwa typów, lecz także bezpieczeństwa wartości — nie sposób użyć niepoprawnych danych, jeśli można je wybrać tylko z ograniczonego zakresu lub zbioru ściśle określonych wartości.

P.: Używam starszej wersji języka Java. Czy to oznacza, że mam kolejny problem?

O.: Nie, nie będziesz mieć żadnych problemów. Zajrzyj do plików z serwera FTP Wydawnictwa Helion — zamieściliśmy tam specjalną wersję aplikacji Ryśka, w której nie są używane typy wyliczeniowe i która z powodzeniem będzie działać także w starszych wersjach języka Java.

Przyjrzyjmy się ogólnej postaci aplikacji Ryśka:



A zatem co tak naprawdę zrobiliśmy?

Znacznie zbliżyliśmy się do zakończenia pierwszego z trzech kroków prowadzących do tworzenia doskonałego oprogramowania. Problemy Ryśka z odnajdywaniem w magazynie gitar spełniających zadane kryteria to już przeszłość.

1. Upewnij się, że oprogramowanie robi to, czego oczekuje klient.

Co więcej, jednocześnie sprawiliśmy, że aplikacja Ryśka jest *bardziej solidna i mniej wrażliwa*. Nie będzie już tak łatwo przysparzać problemów, gdyż poprzez zastosowanie typów wyliczeniowych poprawiliśmy ją zarówno pod względem bezpieczeństwa typów, jak i bezpieczeństwa wartości. Z punktu widzenia Ryśka oznacza to mniej problemów, a z naszego — łatwiejsze utrzymanie aplikacji.

Kod, który nie jest wrażliwy i podatny na awarie, zazwyczaj określa się jako solidny.

Zaostrz ołówek

Wykonaj krok 1. w swoim własnym projekcie.

Czas przekonać się, czy będziesz w stanie sprostać wymaganiom swoich klientów. W pustych wierszach poniżej wpisz krótki opis jakiegoś projektu, nad którym aktualnie pracujesz (możesz także opisać jakiś projekt, który niedawno ukończyłeś):

____

____

____

Teraz, w kolejnych pustych wierszach, zapisz pierwszą rzecz, jaką zrobiłeś, rozpoczynając prace nad tym projektem. Czy miało to cokolwiek wspólnego z upewnieniem się, że aplikacja będzie działać zgodnie z oczekiwaniami i wymaganiami użytkownika?

____

____

____

____

____

Jeśli rozpoczynając prace nad projektem, skupisz się na czymś innym niż zaspokojenie potrzeb i oczekiwań użytkownika, to zastanów się, czym mogłoby się różnić Twoje podejście, gdybyś wiedział o trzech krokach pozwalających na tworzenie wspaniałego oprogramowania. Co by się w takim przypadku zmieniło? Czy sądzisz, że Twoja aplikacja byłaby dzięki temu lepsza, niż jest obecnie, a może gorsza?

____

____

____

____

____

Nie ma niemądrych pytań

P.: A zatem, pracując nad pierwszym krokiem do tworzenia doskonałego oprogramowania, można wprowadzać nieznaczne zmiany w projekcie aplikacji?

O.: Tak, o ile tylko cały czas będziesz się koncentrował głównie na potrzebach użytkownika. Chodzi o to, że chcesz, by wszystkie podstawowe cechy i możliwości aplikacji zostały poprawnie zaimplementowane, *zanim* zaczniesz wprowadzać poważne zmiany w jej projekcie. Niemniej jednak nic nie stoi na przeszkodzie, byś stosował dobre praktyki i techniki obiektowe także podczas pracy nad funkcjonalnością aplikacji, tak by mieć pewność, że od samego początku będzie ona dobrze zaprojektowana.

P.: Czy diagram przedstawiony na stronie 48 to diagram klas? Czy też jest to kilka diagramów — w końcu przedstawia więcej niż jedną klasę?

O.: Jest to diagram klas; na takim diagramie można bowiem zamieścić większą liczbę klas. Prawdę mówiąc, diagramy klas mogą przedstawiać znacznie więcej informacji na temat klasy, niż pokazaliśmy w diagramach zmieszczonych w tej książce. W kilku kolejnych rozdziałach będziemy dodawali do naszych diagramów kolejne informacje o klasach.

P.: A zatem jesteśmy gotowi, by przejść do kroku 2. i rozpocząć stosowanie zasad i technik projektowania obiektowego? Prawda?

O.: Niezupełnie. Można wskazać jeszcze kilka innych rzeczy, dzięki którym powinniśmy pomóc Ryśkowi, zanim będziemy gotowi do rozpoczęcia analizy programu i przystąpimy do jego poprawiania. Pamiętaj, że naszym podstawowym zadaniem jest zaspokojenie potrzeb użytkownika; dopiero *kiedy to zrobimy*, będziemy mogli zająć się poprawianiem projektu naszej aplikacji.

A ja myślałem, że moja nowa aplikacja jest doskonała... Dopiero potem uświadomiłem sobie, że w magazynie mam dwie gitary, które doskonale spełniały oczekiwania Ewy. Czy moglibyście zmienić wyszukiwanie w taki sposób, by zwracało oba odnalezione instrumenty?

Rysiek jest bardzo zadowolony z wprowadzonych modyfikacji, jednak naprawdę pilnie jest mu potrzebna możliwość, by aplikacja zwracała wszystkie odszukane gitary pasujące do wymagań użytkownika, a nie tylko jedną z nich.

Rysiek naprawdę chciałby, żeby Ewa mogła obejrzeć obie gitary.

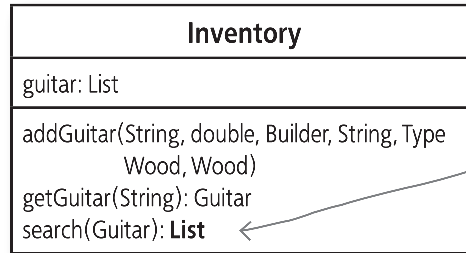
```
inventory.addGuitar("V95693",
1499.95, Builder.FENDER,
"Stratocastor", Type.ELECTRIC,
Wood.ALDER, Wood.ALDER);
```

Obie te gitary są niemal identyczne. Różnią się jedynie numerem seryjnym i ceną.

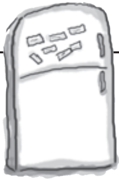
```
inventory.addGuitar("V95612",
1549.95, Builder.FENDER,
"Stratocastor", Type.ELECTRIC,
Wood.ALDER, Wood.ALDER);
```

Klienci Ryśka chcą mieć wybór

Rysiek wymyślił nowe wymaganie dla swojej aplikacji: chciałby, żeby narzędzie wyszukiwawcze zwracało *wszystkie* gitary znajdujące się w magazynie i spełniające wymagania klienta, a nie tylko pierwszą z nich.



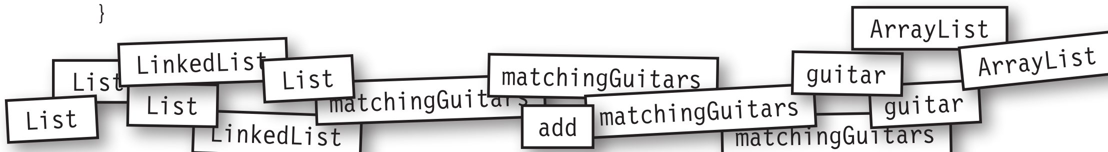
Chcemy, by w przypadku gdy Rysiek dysponuje większą liczbą instrumentów spełniających wymagania podane przez klienta, metoda search() mogła zwracać wiele obiektów Guitar.



Magnesiki z kodem

Kontynuujemy pracę nad pierwszym z trzech kroków do stworzenia wspaniałej aplikacji i skupiamy uwagę na tym, by nasza aplikacja zaczęła działać poprawnie. Poniżej został przedstawiony kod metody search() wyszukującej gitary w magazynie Ryśka; umieściliśmy w nim kilka pustych miejsc, które Ty musisz wypełnić. Użyj do tego celu magnesików z kodem pokazanych u dołu strony. Pamiętaj, że Twoim zadaniem jest takie zmodyfikowanie kodu metody search(), by zwracała ona wszystkie odnalezione w magazynie gitary spełniające kryteria podane przez użytkownika.

```
public ______ search(Guitar searchGuitar) {
  ______ _____________ = new _________();
  for (Iterator i = guitars.iterator(); i.hasNext(); ) {
    Guitar guitar = (Guitar)i.next();
    // Ignorujemy numer seryjny, bo jest unikalny
    // Ignorujemy cenę, gdyż jest unikalna
    if (searchGuitar.getBuilder() != guitar.getBuilder())
      continue;
    String model = searchGuitar.getModel().toLowerCase();
    if ((model != null) && (!model.equals("")) &&
        (!model.equals(guitar.getModel().toLowerCase())))
      continue;
    if (searchGuitar.getType() != guitar.getType())
      continue;
    if (searchGuitar.getBackWood() != guitar.getBackWood())
      continue;
    if (searchGuitar.getTopWood() != guitar.getTopWood())
      continue;
    ____________.____(_____);
  }
  return _____________;
}
```



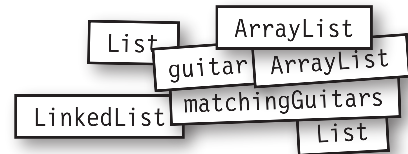
Magnesiki z kodem

Kontynuujemy pracę nad pierwszym z trzech kroków do stworzenia wspaniałej aplikacji i koncentrujemy uwagę na tym, by nasza aplikacja zaczęła działać poprawnie. Poniżej został przedstawiony kod metody **search()** wyszukującej gitary w magazynie Ryśka; umieściliśmy w nim kilka pustych miejsc, które Ty musisz wypełnić. Użyj do tego celu magnesików z kodem, pokazanych u dołu strony. Pamiętaj, że Twoim zadaniem jest takie zmodyfikowanie kodu metody **search()**, by zwracała ona wszystkie odnalezione w magazynie gitary spełniające kryteria podane przez użytkownika.

```
public List search(Guitar searchGuitar) {
  List matchingGuitars = new LinkedList
  for (Iterator i = guitars.iterator(); i.hasNext(); ) {
    Guitar guitar = (Guitar)i.next();
    // Ignorujemy numer seryjny, bo jest unikalny
    // Ignorujemy cenę, gdyż jest unikalna
    if (searchGuitar.getBuilder() != guitar.getBuilder())
      continue;
    String model = searchGuitar.getModel().toLowerCase();
    if ((model != null) && (!model.equals(„")) &&
        (!model.equals(guitar.getModel().toLowerCase())))
      continue;
    if (searchGuitar.getType() != guitar.getType())
      continue;
    if (searchGuitar.getBackWood() != guitar.getBackWood())
      continue;
    if (searchGuitar.getTopWood() != guitar.getTopWood())
      continue;
    matchingGuitars . add ( guitar );
  }
  return matchingGuitars
}
```

W praktyce w tym miejscu można zastosować zarówno klasę LinkedList, jak i ArrayList… obie spełniłyby swoje zadanie.

Gitary pasujące do podanych wymagań zostają dodane do listy wszystkich instrumentów, którymi klient może być zainteresowany.



Niewykorzystane magnesiki.

Nie ma niemądrych pytań

P: A zatem pierwszego etapu prac nie można uznać za zakończony aż do momentu, gdy aplikacja będzie działać tak, jak sobie tego życzy klient?

O: Właśnie. Powinieneś upewnić się, że aplikacja działa tak, jak powinna, zanim zajmiesz się zastosowaniem w niej wzorców projektowych bądź nim zaczniesz wprowadzać jakiekolwiek poważniejsze zmiany w jej strukturze.

P: A dlaczego zakończenie tego etapu prac przed rozpoczęciem kolejnego jest takie ważne?

O: Zapewnienie poprawnego działania oprogramowania wymaga zazwyczaj dokonania w nim bardzo wielu zmian. Wprowadzanie zbyt wielu zmian w strukturze aplikacji przed zagwarantowaniem poprawnego działania przynajmniej jej podstawowych możliwości funkcjonalnych może się okazać stratą czasu i wysiłku, gdyż struktura programu niekiedy ulega poważnym zmianom podczas implementacji jego kolejnych możliwości.

P: Wydaje mi się, że zwracacie nadmierną uwagę na ten „krok 1." i „krok 2.". A co, jeśli ja projektuję swoje aplikacje w inny sposób?

O: Nie musisz ściśle trzymać się podawanych przez nas informacji o każdym z kroków. Niemniej jednak stanowią one prostą sekwencję czynności, której wykonanie gwarantuje, że tworzone oprogramowanie będzie działać zgodnie z oczekiwaniami, będzie dobrze zaprojektowane i nie wystąpią problemy w jego wielokrotnym użytkowaniu. Jeśli w inny sposób zrealizujesz te same cele, to super!

Test

Wielokrotnie pisaliśmy o potrzebie uzyskania od klienta informacji o wymaganiach stawianych tworzonemu oprogramowaniu; teraz jednak nadszedł czas, by przekonać się, czy te wymagania są dobrze realizowane przez nasz kod. Sprawdźmy zatem, czy nasza aplikacja działa tak, jakby sobie tego życzył Rysiek:

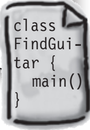
Oto program testowy, zaktualizowany tak, by mógł korzystać z nowej wersji narzędzia wyszukującego gitary w magazynie Ryśka.

Używamy w nim typów wyliczeniowych. Tym razem nie będzie więc żadnych problemów spowodowanych niewłaściwie zapisanymi łańcuchami znaków!

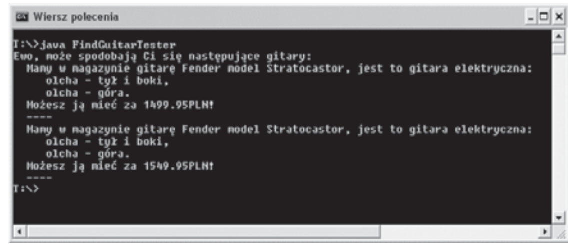
```
public class FindGuitarTester {

  public static void main(String[] args) {
    // Inicjalizacja zawartości magazynu gitar Ryśka
    Inventory inventory = new Inventory();
    initializeInventory(inventory);

    Guitar whatEveLikes = new Guitar("", 0, Builder.FENDER, "Stratocastor",
                                     Type.ELECTRIC, Wood.ALDER, Wood.ALDER);

    List matchingGuitars = inventory.search(whatEveLikes);
    if (!matchingGuitars.isEmpty()) {
      System.out.println("Ewo, może spodobają Ci się następujące gitary:");
      for (Iterator i = matchingGuitars.iterator(); i.hasNext(); ) {
        Guitar guitar = (Guitar)i.next();
        System.out.println("  Mamy w magazynie gitarę " +
          guitar.getBuilder() + " model " + guitar.getModel() + ", jest " +
          "to gitara" + guitar.getType() + " :\n      " +
          guitar.getBackWood() + " - tył i boki,\n      " +
          guitar.getTopWood() + " - góra.\n  Możesz ją mieć za " +
          guitar.getPrice() + "PLN!\n  ----");
      }
    } else {
      System.out.println("Przykro mi, Ewo, nie znalazłem nic dla Ciebie.");
    }
  }
```

W tej wersji programu testowego musimy przejrzeć całą listę instrumentów zwróconą przez narzędzie wyszukujące.

Teraz otrzymujemy całą listę gitar, które spełniają wymagania określone przez klienta.



FindGuitarTester.java



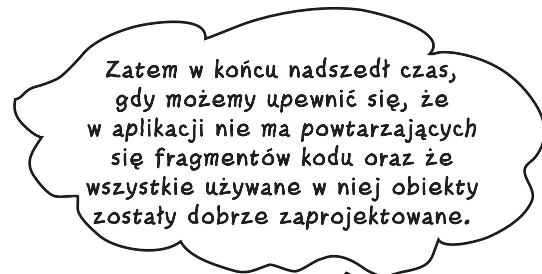
Wszystko zadziałało tak, jak powinno! Ewa mogła obejrzeć kilka poleconych jej gitar, a klienci na powrót zaczęli kupować instrumenty w sklepie Ryśka.

Wróćmy do naszych trzech kroków

Teraz, kiedy nasza aplikacja działa już tak, jak Rysiek sobie tego życzył, możemy zacząć stosować zasady projektowania obiektowego, by poprawić jej elastyczność i zapewnić, że będzie dobrze zaprojektowana.

Teraz, kiedy aplikacja działa już tak, jak chciał Rysiek, ten krok możemy uznać za zakończony.

✓1. Upewnij się, że oprogramowanie robi to, czego oczekuje klient.



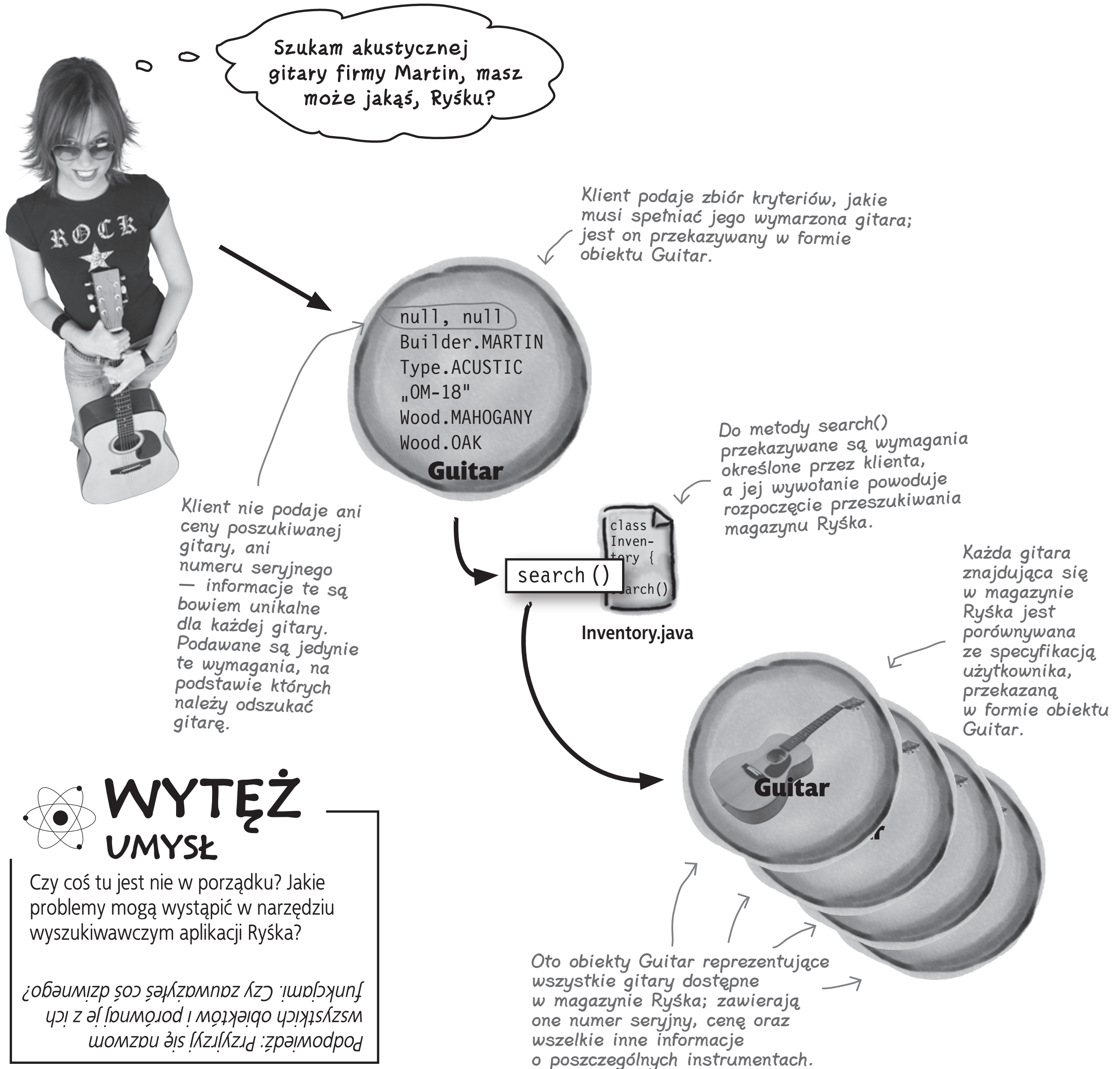
2. Zastosuj proste zasady projektowania obiektowego, by poprawić elastyczność oprogramowania.

W tym kroku analizujemy działający program i sprawdzamy, czy fragmenty, z jakich się składa, są scalone w sensowny sposób.

3. Staraj się, by projekt oprogramowania zapewniał łatwość jego utrzymania i pozwalał na jego wielokrotne stosowanie.

Szukamy problemów

Przyjrzyjmy się nieco dokładniej naszemu narzędziu do wyszukiwania gitar i sprawdźmy, czy uda się nam znaleźć jakieś problemy, które moglibyśmy rozwiązać przy wykorzystaniu prostych zasad projektowania obiektowego. Zacznijmy od przeanalizowania sposobu działania metody **`search()`** klasy **`Inventory`**.



WYTĘŻ UMYSŁ

Czy coś tu jest nie w porządku? Jakie problemy mogą wystąpić w narzędziu wyszukiwawczym aplikacji Ryśka?

Podpowiedź: Przyjrzyj się nazwom wszystkich obiektów i porównaj je z ich funkcjami. Czy zauważyłeś coś dziwnego?

Analiza metody search()

Poświęćmy nieco czasu na dokładniejsze przeanalizowanie tego, co się dzieje w metodzie `search()` klasy `Inventory`. Zanim przyjrzymy się samemu kodowi tej metody, zastanówmy się nad tym, co ona powinna robić.

1. **Klient podaje swoje wymagania dotyczące poszukiwanej gitary.**

   Każdy z klientów przychodzących do sklepu Ryśka chciałby, żeby poszukiwana gitara posiadała określone cechy: gatunek używanego drewna, typ gitary bądź też określony model, określonego producenta. Klienci podają te cechy Ryśkowi, który z kolei przekazuje je do narzędzia przeszukującego magazyn.

   Klient może wskazać jedynie ogólne cechy instrumentu. Dlatego też klienci nigdy nie podają ani ceny, ani numeru seryjnego poszukiwanej gitary.

2. **Narzędzie wyszukujące przegląda zawartość magazynu.**

   Kiedy narzędzie wyszukujące wie, czego chce klient, rozpoczyna wykonywanie pętli, w której sprawdzane są wszystkie gitary znajdujące się w magazynie.

3. **Każda gitara jest analizowana pod kątem zgodności z wymaganiami określonymi przez klienta.**

   Wszystkie ogólne właściwości, takie jak użyte gatunki drewna, producent czy też typ gitary, są porównywane z wymaganiami określonymi przez klienta.

   Dla każdej gitary znajdującej się w magazynie narzędzie wyszukujące sprawdza, czy spełnia ona wymagania określone przez klienta. Jeśli wymagania są spełnione, to gitara jest dodawana do listy pasujących instrumentów.

4. **Klientowi przedstawiana jest lista wszystkich instrumentów spełniających zadane kryteria.**

   W końcu lista pasujących instrumentów jest wyświetlana, a Rysiek może ją przedstawić klientowi. Klient może wybrać odpowiadający mu instrument, a Rysiek — zainkasować zapłatę.

Spróbuj słownie opisać rozwiązywany problem, aby upewnić się, że projekt rozwiązania odpowiada planowanym możliwościom funkcjonalnym aplikacji.

Tajemnica obiektów o niedopasowanych typach

W lepiej zaprojektowanych dzielnicach Obiektowa obiekty bardzo poważnie i precyzyjnie podchodzą do swoich zadań. Każdy z nich jest zainteresowany tylko i wyłącznie swoimi zadaniami i stara się je wykonywać jak najlepiej. Nie ma niczego, co dobrze zaprojektowane obiekty nie cierpiałyby bardziej niż wykonywanie zadań, do których tak naprawdę nie zostały przeznaczone.

Niestety, jak udało się nam zauważyć, właśnie taka sytuacja występuje w narzędziu służącym do przeszukiwania magazynu gitar Ryśka: w pewnym jego miejscu pewien obiekt jest używany do wykonywania operacji, których tak naprawdę nie powinien wykonywać. Twoim zadaniem jest rozwiązanie tej zagadki i określenie, w jaki sposób można poprawić aplikację Ryśka.

Aby ułatwić Ci zadanie, poniżej podaliśmy kilka przydatnych wskazówek, które pomogą Ci rozpocząć poszukiwania niepasującego typu obiektów:

1. Zadania wykonywane przez obiekty powinny pasować do nazwy tych obiektów.
Jeśli pewien obiekt należy do klasy Odrzutowiec, to prawdopodobnie powinien on mieć metody ląduj() oraz startuj(), jednak nie powinien udostępniać metody kontrolujBilety() — bo kontrola biletów jest zadaniem należącym do jakiegoś innego obiektu.

2. Każdy obiekt powinien reprezentować jedno pojęcie.
Nie chcesz używać obiektów realizujących dwa lub trzy różne obowiązki. Unikaj obiektów, które będą reprezentować „prawdziwą” kwaczącą kaczkę, żółtą, plastykową kaczuszkę do kąpieli oraz osobę, która schyla głowę, by uniknąć trafienia piłką na meczu w baseball.

3. Nieużywane właściwości obiektów są podejrzane.
Jeśli okaże się, że właściwości w obiekcie bardzo często mają wartości `null` lub w ogóle nie są używane, to może to oznaczać, że obiekt wykonuje więcej niż jedno zadanie. Skoro jakaś właściwość obiektu bardzo rzadko ma jakieś wartości, to dlaczego stanowi ona część tego obiektu? Czy nie lepiej byłoby ją umieścić w jakimś innym obiekcie, zawierającym tylko podzbiór właściwości oryginalnego obiektu?

STOP! Spróbuj rozwiązać tę zagadkę, zanim zaczniesz czytać następną stronę.

Jak myślisz, jaki typ obiektu jest nieprawidłowo używany w aplikacji Ryśka? Zapisz odpowiedź poniżej.

____

A jak myślisz, co należy zrobić, by rozwiązać ten problem? Jakie zmiany w aplikacji byś wprowadził?

____

____

____

No wiecie... Klienci Ryśka tak naprawdę nie przekazują mu obiektów klasy Guitar... Chodzi mi o to, że w istocie nie dają mu gitar, które on następnie porównuje z instrumentami w magazynie.

Franek: Fakt — masz rację. Nie pomyślałem o tym wcześniej.

Julka: No i co z tego? Zastosowanie obiektu Guitar znacznie ułatwia wykonywanie porównywania w metodzie search().

Jerzy: Nie bardziej niż zastosowanie jakiegokolwiek innego obiektu. Spójrzcie:

```
if (searchGuitar.getBuilder() !=
         guitar.getBuilder()) {
  continue;
}
```

To niewielki fragment metody search() klasy Inventory.

Jerzy: Tak naprawdę nie ma znaczenia, jakiego obiektu tu używamy, o ile tylko będziemy w stanie określić, na jakich cechach gitary zależy klientowi.

Franek: Tak... Myślę, że powinniśmy stworzyć nowy typ obiektów, który przechowywałby jedynie takie specyfikacje, jakie klient chce przekazać do metody **`search()`**. W takim przypadku do tej metody nie byłyby przekazywane obiekty **`Guitar`**, co, swoją drogą, nigdy mi się nie wydawało szczególnie sensowne.

Julka: Jednak czy takie rozwiązanie nie spowoduje powielania kodu w aplikacji? Jeśli stworzymy obiekt, w którym będzie można zapisać wszystkie specyfikacje podawane przez klienta, a oprócz tego mamy obiekt **`Guitar`** ze wszystkimi jego właściwościami i metodami, to w efekcie uzyskamy dwie metody **`getBuilder()`**, dwie metody **`getWood()`** i tak dalej... To chyba niezbyt dobrze...

Franek: W takim razie dlaczego nie hermetyzować tych właściwości i nie przenieść z klasy **`Guitar`** do jakiejś innej, nowej?

Jerzy: O rany... Rozumiałem wszystko do chwili, gdy powiedziałeś „hermetyzować". Myślałem, że hermetyzacja polega na zdefiniowaniu zmiennych jako prywatne, tak by nikt nie mógł ich używać w niewłaściwy sposób. Ale co to ma wspólnego z właściwościami gitary?

Franek: Hermetyzacja to także podział aplikacji na logiczne fragmenty oraz zachowanie ich separacji. A zatem, podobnie jak dane w klasie separujemy od działania pozostałych fragmentów aplikacji, tak i same właściwości gitary możemy oddzielić od samego obiektu **`Guitar`**.

Julka: Czy w takim przypadku w klasie **`Guitar`** pozostałaby jedynie zmienna zawierająca referencję do obiektu nowego typu, który gromadziłby wszystkie informacje o gitarze?

Franek: Dokładnie! W ten sposób udałoby się nam hermetyzować właściwości gitary od obiektu **`Guitar`** i umieścić je w osobnym obiekcie. Spójrzcie, moglibyśmy zrobić coś takiego...

Hermetyzacja pozwala podzielić aplikację na logiczne części.

Po raz pierwszy spotkałeś się z terminem **hermetyzacja**? Zajrzyj do dodatku B, przeczytaj skrócone informacje o Obiektowie i dopiero potem wróć do lektury tego rozdziału.

Zaostrz ołówek

Utworzenie obiektu GuitarSpec

Poniżej przedstawiliśmy diagram klasy Guitar oraz nowej klasy o nazwie GuitarSpec, o której przed chwilą rozmawiali Franek, Julka i Jerzy. Twoim zadaniem jest dodanie do klasy GuitarSpec wszystkich właściwości i metod, które według Ciebie będą w niej niezbędne. Następnie wykreśl z klasy Guitar wszystkie właściwości i metody, które nie będą już w niej potrzebne. W końcu w diagramie klasy Guitar pozostawimy Ci nieco wolnego miejsca, na wypadek gdybyś doszedł do wniosku, że będziesz musiał dodać do niej jakieś nowe właściwości lub metody.

Guitar
serialNumber: String price: double builder: Builder model: String type: Type backWood: Wood topWood: Wood ________ ________
getSerialNumber(): String getPrice(): double setPrice(float) getBuilder(): Builder getModel(): String getType(): Type getBackWood(): Wood getTopWood(): Wood ________ ________

Usuń z klasy Guitar wszystko, co według Ciebie powinno się znaleźć w klasie GuitarSpec.

Do klasy Guitar możesz dodać nowe właściwości i metody, jeśli uznasz to za właściwe.

GuitarSpec
________ ________ ________ ________ ________ ________ ________ ________ ________
________ ________ ________ ________ ________ ________ ________ ________ ________ ________

* Jeśli będziesz mieć problemy, to pomyśl o wspólnych elementach, które występują w obiekcie Guitar i które są przekazywane w wywołaniu metody search().

Zaostrz ołówek
Rozwiązanie

Utworzenie obiektu GuitarSpec

Poniżej przedstawiliśmy diagram klasy `Guitar` oraz nowej klasy o nazwie `GuitarSpec`, o której przed chwilą rozmawiali Franek, Julka i Jerzy. Twoim zadaniem jest dodanie do klasy `GuitarSpec` wszystkich właściwości i metod, które według Ciebie będą w niej niezbędne. Następnie wykreśl z klasy `Guitar` wszystkie właściwości i metody, które nie będą już w niej potrzebne. W końcu w diagramie klasy `Guitar` pozostawimy Ci nieco wolnego miejsca, na wypadek gdybyś doszedł do wniosku, że będziesz musiał dodać do niej jakieś nowe właściwości lub metody.

Te dwie właściwości wciąż są unikalne dla każdej gitary, a zatem muszą pozostać bez zmian.

To są właściwości, które klienci Ryśka przekazują do metody search(); a zatem możemy je przenieść do klasy GuitarSpec.

Będzie nam także potrzebna referencja do obiektu GuitarSpec opisującego daną gitarę.

Guitar

serialNumber: String
price: double
~~builder: Builder~~
~~model: String~~
~~type: Type~~
~~backWood: Wood~~
~~topWood: Wood~~
spec: GuitarSpec

getSerialNumber(): String
getPrice(): double
setPrice(float)
~~getBuilder(): Builder~~
~~getModel(): String~~
~~getType(): Type~~
~~getBackWood(): Wood~~
~~getTopWood(): Wood~~
getSpec(): GuitarSpec

GuitarSpec

builder: Builder
model: String
type: Type
backWood: Wood
topWood: Wood

getBuilder(): Builder
getModel(): String
getType(): Type
getBackWood(): Wood
getTopWood(): Wood

Powtarzanie się kodu eliminujemy, przenosząc wszystkie wspólne właściwości i związane z nimi metody do obiektu, którego będziemy używali zarówno do określania właściwości gitar, jak i kryteriów wyszukiwania.

Z metodami postępujemy dokładnie tak samo jak z właściwościami — z klasy Guitar usuwamy wszelkie metody, które powtarzają się w niej oraz w nowej klasie GuitarSpec.

A teraz uaktualnij swój kod

Dysponując powyższym diagramem, powinieneś być w stanie dodać do aplikacji nową klasę `GuitarSpec` i zaktualizować kod klasy `Guitar`. Od razu wprowadź niezbędne modyfikacje także w klasie `Inventory`, tak by całą aplikację można było poprawnie skompilować.

Nie ma niemądrych pytań

P: Rozumiem, dlaczego potrzebujemy obiektu, by przekazywać wymagania klienta do metody search()… ale dlaczego używamy go w obiekcie Guitar do przechowywania informacji o właściwościach gitary?

O: Załóżmy, że zastosowalibyśmy obiekt `GuitarSpec` jedynie do przechowywania wymagań klienta i przekazywania ich do metody `search()`, natomiast klasa `Guitar` pozostałaby niezmieniona. W takim przypadku, gdyby Rysiek zaczął handlować gitarami 12-strunowymi i chciał dodać właściwość `numStrings`, to musiałbyś dodać taką właściwość — oraz towarzyszący jej kod metody `getNumStrings()` — zarówno w klasie `GuitarSpec`, jak i `Guitar`. Rozumiesz zapewne, że to prowadziłoby do powtarzania się tego samego kodu.

Zamiast tego cały (potencjalnie) powtarzający się kod możemy umieścić w klasie `GuitarSpec`, a do klasy `Guitar` dodać referencję do obiektu `GuitarSpec`. W ten sposób unikniemy powtarzania się kodu.

Za każdym razem gdy zauważysz powtarzający się kod, poszukaj okazji do zastosowania hermetyzacji.

P: Wciąż nie do końca rozumiem, dlaczego takie rozwiązanie jest traktowane jako hermetyzacja. Czy możecie mi to jeszcze raz wyjaśnić?

O: Ideą hermetyzacji jest zabezpieczenie informacji gromadzonych w jednym miejscu aplikacji przed jej pozostałymi fragmentami. W najprostszym przypadku pewną informację przechowywaną w klasie można chronić przed pozostałym kodem aplikacji poprzez zdefiniowanie jej jako składowej prywatnej. Jednak czasami będziemy chcieli w taki sposób zabezpieczyć nie jedną, lecz całą grupę właściwości — takich jak szczegółowe informacje o cechach gitary — a nawet zachowania, na przykład sposób, w jaki latają poszczególne gatunki kaczek.

Jeśli usuniemy zachowanie poza klasę, będziemy mogli je zmieniać bez konieczności jednoczesnej modyfikacji samej klasy. A zatem, gdybyś zmienił sposób przechowywania właściwości, to nie musiałbyś wprowadzać jakichkolwiek modyfikacji w klasie `Guitar`, gdyż właściwości zostały z niej usunięte i przeniesione do innej klasy.

Właśnie na tym polega potęga hermetyzacji: poprzez podzielenie aplikacji na części uzyskujemy możliwość modyfikowania jednej z nich, bez konieczności wprowadzania zmian w innych. Ogólnie rzecz biorąc, powinieneś hermetyzować te części aplikacji, które mogą się zmieniać, oddzielając je od fragmentów aplikacji, które zmieniać się nie będą.

Zobaczmy, jak postępują prace nad naszymi trzema krokami, które mają nam pozwolić na stworzenie doskonałej aplikacji.

1. Upewnij się, że oprogramowanie robi to, czego oczekuje klient.

Zajmujemy się teraz krokiem 2. — czyli pracujemy nad projektem aplikacji.

2. Zastosuj proste zasady projektowania obiektowego, by poprawić elastyczność oprogramowania.

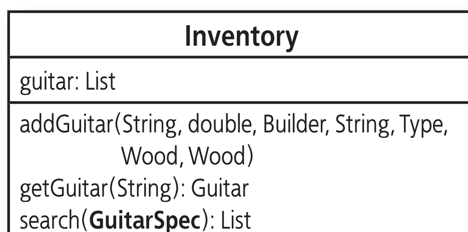
To właśnie w tym miejscu zaczynasz szukać poważnych problemów, zwłaszcza związanych z takimi zagadnieniami jak powtarzający się kod lub nieprawidłowo zaprojektowane klasy.

3. Staraj się, by projekt oprogramowania zapewniał łatwość jego utrzymania i pozwalał na jego wielokrotne stosowanie.

Pamiętaj, że w tym kroku też będziemy mieli sporo pracy związanej z projektem aplikacji; a zatem, zanim ukończysz prace, Twój kod będzie naprawdę łatwy do rozbudowy i wielokrotnego zastosowania.

Aktualizacja klasy Inventory

Teraz, kiedy już hermetyzowaliśmy specyfikację gitary, musimy wprowadzić w naszym kodzie kilka następnych zmian.



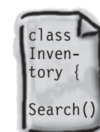
Teraz do metody search() przekazywany jest obiekt GuitarSpec, a nie Guitar.

Obecnie wszystkie informacje, jakich używamy podczas porównywania, pochodzą z obiektów GuitarSpec, a nie Guitar.

Ten kod jest niemal identyczny jak wcześniej, a jedyna różnica polega na tym, że porównujemy informacje przechowywane w obiekcie GuitarSpec.

```
public class Inventory {

  // właściwości, konstruktor, inne metody

  public List search(GuitarSpec searchSpec) {
    List matchingGuitars = new LinkedList();
    for (Iterator i = guitars.iterator(); i.hasNext(); ) {
      Guitar guitar = (Guitar)i.next();
      GuitarSpec guitarSpec = guitar.getSpec();
      if (searchSpec.getBuilder() != guitarSpec.getBuilder())
        continue;
      String model = searchSpec.getModel().toLowerCase();
      if ((model != null) && (!model.equals("")) &&
          (!model.equals(guitarSpec.getModel().toLowerCase())))
        continue;
      if (searchSpec.getType() != guitarSpec.getType())
        continue;
      if (searchSpec.getBackWood() != guitarSpec.getBackWood())
        continue;
      if (searchSpec.getTopWood() != guitarSpec.getTopWood())
        continue;
      matchingGuitars.add(guitar);
    }
    return matchingGuitars;
  }
}
```

Choć wprowadziliśmy nieznaczne modyfikacje w kodzie klasy, to ta metoda wciąż zwraca listę gitar spełniających zadane kryteria.



Inventory.java

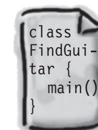
Przygotuj się na kolejny test

Aby przetestować te wszystkie modyfikacje, będziesz musiał wprowadzić kilka zmian w klasie **FindGuitarTester**:

```
public class FindGuitarTester {

  public static void main(String[] args) {
    // Inicjalizacja zawartości magazynu gitar Ryśka
    Inventory inventory = new Inventory();
    initializeInventory(inventory);

    GuitarSpec whatEveLikes =
      new GuitarSpec(Builder.FENDER, "Stratocastor",
                     Type.ELECTRIC, Wood.ALDER, Wood.ALDER);
    List matchingGuitars = inventory.search(whatEveLikes);
    if (!matchingGuitars.isEmpty()) {
      System.out.println("Ewo, może spodobają Ci się następujące gitary:";
      for (Iterator i = matchingGuitars.iterator(); i.hasNext(); ) {
        Guitar guitar = (Guitar)i.next();
        GuitarSpec spec = guitar.getSpec();
        System.out.println("  Mamy w magazynie gitarę " +
          spec.getBuilder() + " model " + spec.getModel() + ",
          jest " + "to gitara" + spec.getType() + " :\n     " +
          spec.getBackWood() + " - tył i boki,\n     " +
          spec.getTopWood() + " - góra.\n  Możesz ją mieć za " +
          spec.getPrice() + "PLN!\n  ----");
      }
    } else {
      System.out.println("Przykro mi, Ewo, nie znalazłem nic dla Ciebie.");
    }
  }

  private static void initializeInventory(Inventory inventory) {
    // dodanie gitar do magazynu
  }
}
```

Tym razem klient przekazuje do metody search() obiekt klasy GuitarSpec.

Także tutaj używamy nowej klasy GuitarSpec.



FindGuitarTester.java

DLACZEGO MAM NA TO ZWRACAĆ UWAGĘ ?

Dowiedziałeś się już całkiem dużo na temat pisania doskonałego oprogramowania, jednak wciąż jeszcze musisz się wiele nauczyć. Weź głęboki oddech i zastanów się nad niektórymi terminami i zasadami, które przedstawiliśmy w tym rozdziale. Połącz słowa umieszczone w lewej kolumnie z wyjaśnieniami celów stosowania danych technik lub zasad, znajdującymi się w kolumnie prawej.

Elastyczność	Beze mnie Twój klient nigdy nie będzie zadowolony. Bez względu na to, jak wspaniale byłaby zaprojektowana i napisana aplikacja, to właśnie ja sprawiam, że na twarzy klienta pojawia się uśmiech.
Hermetyzacja	Ja wkraczam do akcji tam, gdzie chodzi o możliwości wielokrotnego wykorzystania tego samego kodu i uzyskanie pewności, że nie trzeba będzie rozwiązywać problemów, które ktoś inny rozwiązał już wcześniej.
Funkcjonalność	Programiści używają mnie po to, by oddzielić od siebie fragmenty kodu, które ulegają zmianom, od tych, które pozostają takie same; dzięki temu łatwo można wprowadzać w kodzie modyfikacje — bez obawy, że cała aplikacja przestanie działać.
Wzorce projektowe	Programiści używają mnie po to, by oprogramowanie można było rozwijać i zmieniać bez konieczności ciągłego pisania go od samego początku. To ja sprawiam, że aplikacje stają się solidne i odporne.

→ Odpowiedzi znajdziesz na stronie 81

Nie ma niemądrych pytań

P: Hermetyzacja nie jest jedyną zasadą projektowania obiektowego, jakiej używamy na tym etapie prac nad aplikacją, prawda?

O: Nie. Kolejnymi z zasad, o których warto pamiętać, są dziedziczenie i polimorfizm. Jednak także i one łączą się z powtarzalnością kodu oraz hermetyzacją; dlatego rozpoczynanie pracy od poszukiwania miejsc, w jakich można zastosować hermetyzację, zawsze będzie dobrym rozwiązaniem.

W dalszej części książki przedstawimy znacznie więcej informacji o zasadach projektowania obiektowego (a w rozdziale 8. zaprezentujemy nawet kilka przykładów); zatem nie przejmuj się, jeśli teraz jeszcze nie wszystko dokładnie rozumiesz. Zanim skończysz lekturę tej książki, dowiesz się znacznie więcej o hermetyzacji, projektowaniu klas oraz o wielu innych zagadnieniach.

P: Ale nie do końca rozumiem, w jaki sposób ta cała hermetyzacja poprawia elastyczność mojego kodu. Możecie to jeszcze raz wyjaśnić?

O: Kiedy już poprawiłeś swój kod i zapewniłeś, że działa on tak, jak sobie tego życzył klient, największym problemem staje się elastyczność. Co się stanie, jeśli klient poprosi o dodanie do aplikacji kilku nowych właściwości lub możliwości funkcjonalnych? Jeśli w aplikacji będzie wiele powtarzających się fragmentów kodu bądź jeśli jej struktura dziedziczenia będzie skomplikowana i niejasna, to wprowadzanie modyfikacji w takim programie może stać się prawdziwym koszmarem.

Jednak dzięki zastosowaniu takich zasad jak hermetyzacja oraz poprawne projektowanie klas wprowadzanie zmian może być znacznie łatwiejsze, a sama aplikacja stanie się bardziej elastyczna.

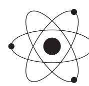
Wróćmy do aplikacji Ryśka...

Sprawdźmy teraz, czy wszystkie wprowadzone zmiany nie wpłynęły negatywnie na poprawność działania aplikacji Ryśka. Skompiluj wszystkie klasy i jeszcze raz uruchom program **`FindGuitarTester`**:

Wiersz polecenia

```
T:\>java FindGuitarTester
Ewo, może spodobają Ci się następujące gitary:
  Mamy w magazynie gitarę Fender model Stratocastor, jest to gitara elektryczna:
     olcha - tył i boki,
     olcha - góra.
  Możesz ją mieć za 1499.95PLN!
  ----
  Mamy w magazynie gitarę Fender model Stratocastor, jest to gitara elektryczna:
     olcha - tył i boki,
     olcha - góra.
  Możesz ją mieć za 1549.95PLN!
  ----
T:\>
```

Tym razem wyniki niczym się nie różnią od uzyskiwanych poprzednio, jednak sama aplikacja jest lepiej zaprojektowana i znacznie bardziej elastyczna.



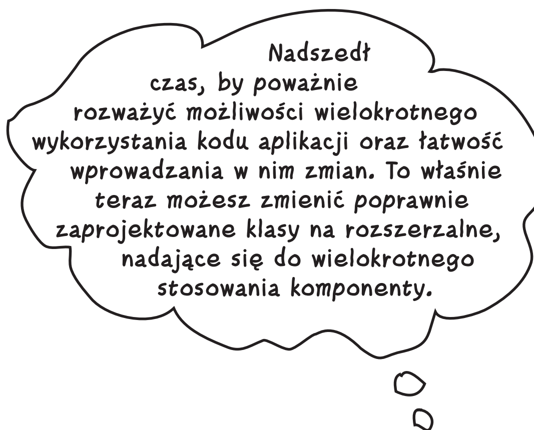
WYTĘŻ UMYSŁ

Czy jesteś w stanie podać trzy konkretne czynniki, które sprawiają, że w dobrze zaprojektowanym oprogramowaniu wprowadzanie zmian jest łatwiejsze niż w oprogramowaniu, w którym fragmenty kodu powielają się?

Projekt po raz pierwszy, projekt po raz drugi

Skoro już raz przeanalizowałeś aplikację i zastosowałeś w niej podstawowe techniki projektowania obiektowego, czas przyjrzeć się jej ponownie i upewnić, że jest ona nie tylko elastyczna, lecz także zapewnia łatwość wielokrotnego stosowania fragmentów kodu oraz ich rozszerzania.

✓1. Upewnij się, że oprogramowanie robi to, czego oczekuje klient.



✓2. Zastosuj proste zasady projektowania obiektowego, by poprawić elastyczność oprogramowania.

Skoro już zastosowałeś podstawowe zasady projektowania obiektowego, nadszedł czas, by wykorzystać wzorce projektowe i skoncentrować się na możliwościach wielokrotnego stosowania kodu.

3. Staraj się, by projekt oprogramowania zapewniał łatwość jego utrzymania i pozwalał na jego wielokrotne stosowanie.

Sprawdźmy, czy klasa Inventory.java jest (naprawdę) dobrze zaprojektowana

W poprzedniej części rozdziału zastosowaliśmy hermetyzację, by poprawić projekt narzędzia do wyszukiwania gitar w magazynie Ryśka, niemniej jednak w naszym kodzie wciąż można znaleźć miejsca, które należałoby poprawić w celu pozbycia się potencjalnych problemów. Dzięki temu nasz kod będzie można łatwiej rozszerzać, kiedy Rysiek wymyśli **kolejne** możliwości, które chciałby dodać do aplikacji, oraz łatwiej wykorzystać, jeśli zechcemy zastosować jego fragmenty w innej aplikacji.

Teraz, kiedy już oddałeś Ryśkowi sprawne narzędzie wyszukujące w magazynie gitary pasujące do wymagań określanych przez klientów, masz pewność, że Rysiek ponownie do Ciebie zadzwoni, jeśli zechce coś zmienić w swojej aplikacji.

Oto kod metody search() klasy Inventory. Przyjrzyj mu się dokładniej.

```
public List search(GuitarSpec searchSpec) {
    List matchingGuitars = new LinkedList();
    for (Iterator i = guitars.iterator(); i.hasNext(); ) {
      Guitar guitar = (Guitar)i.next();
      GuitarSpec guitarSpec = guitar.getSpec();
      if (searchSpec.getBuilder() != guitarSpec.getBuilder())
        continue;
      String model = searchSpec.getModel().toLowerCase();
      if ((model != null) && (!model.equals("")) &&
          (!model.equals(guitarSpec.getModel().toLowerCase())))
        continue;
      if (searchSpec.getType() != guitarSpec.getType())
        continue;
      if (searchSpec.getBackWood() != guitarSpec.getBackWood())
        continue;
      if (searchSpec.getTopWood() != guitarSpec.getTopWood())
        continue;
      matchingGuitars.add(guitar);
    }
    return matchingGuitars;
  }
```

```
class
Inven-
tory {

Search()
```

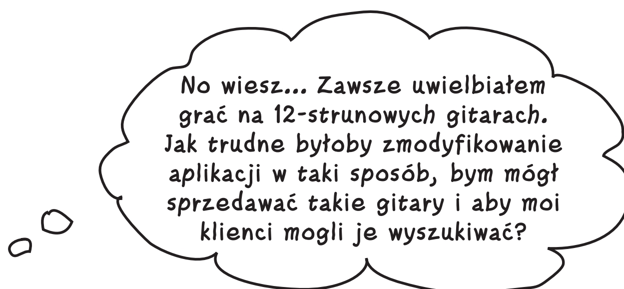
Inventory.java

Zaostrz ołówek

Co zmieniłbyś w tym fragmencie kodu?

W przedstawionym kodzie występuje poważny problem, Twoim zadaniem jest go odnaleźć. W poniższych pustych wierszach zapisz, na czym według Ciebie polega ten problem oraz w jaki sposób można by go rozwiązać.

Zobacz, co napisaliśmy na stronie 82



Jak łatwo będzie wprowadzić taką zmianę do aplikacji Ryśka?

Przeanalizuj diagram klas tworzących aplikację Ryśka i zastanów się, co należałoby zrobić, aby wyposażyć ją w możliwość obsługi gitar 12-strunowych. Jakie właściwości i metody musiałbyś dodać, w jakich klasach należałoby je umieścić? Jakie zmiany musiałbyś wprowadzić w kodzie aplikacji, by zapewnić klientom Ryśka możliwość wyszukiwania 12-strunowych gitar?

Ile klas musiałeś zmienić, by wprowadzić powyższą modyfikację? Czy dalej uważasz, że aplikacja Ryśka jest dobrze zaprojektowana?

Guitar
serialNumber: String price: double spec: GuitarSpec
getSerialNumber(): String getPrice(): double setPrice(float) getSpec(): GuitarSpec

GuitarSpec

builder: Builder
model: String
type: Type
backWood: Wood
topWood: Wood

getBuilder(): Builder
getModel(): String
getType(): Type
getBackWood(): Wood
getTopWood(): Wood

Przypiski do diagramu klas aplikacji Ryśka

Rysiek chciałby sprzedawać 12-strunowe gitary. Przygotuj zatem ołówek i umieść na diagramie klas notatki zawierające następujące informacje:

1. Do jakiej klasy dodałbyś nową właściwość o nazwie numStrings, która będzie służyć do przechowywania informacji o liczbie strun w danej gitarze?
2. Gdzie dodałabyś nową metodę o nazwie **`getNumStrings()`**, która zwraca liczbę strun w gitarze?
3. W jakich innych miejscach aplikacji powinieneś wprowadzić zmiany w kodzie, tak by podczas wyszukiwania gitar klienci Ryśka mogli określać, że interesują ich gitary 12-strunowe?

W końcu, w umieszczonych poniżej pustych wierszach zapisz wszelkie problemy związane z projektem aplikacji, które odnalazłeś podczas dodawania obsługi 12-strunowych gitar.

___

___

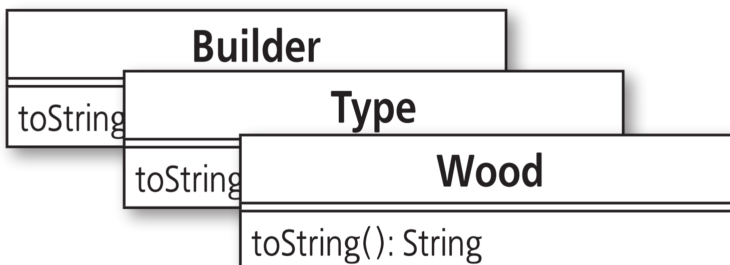
___

Oto podpowiedź: zapisana tutaj odpowiedź powinna być związana z informacjami, które zapisałeś w pustych wierszach na stronie 67.

Inventory

guitars: List

addGuitar(String, double, Builder, String, Type, Wood, Wood)
getGuitar(String): Guitar
search(GuitarSpec): List



Na czym polega przewaga utworzenia nowej właściwości o nazwie numStrings nad dodaniem właściwości logicznej określającej, czy dana gitara ma 12 strun?

Zaostrz ołówek Rozwiązanie

Przypiski do diagramu klas aplikacji Ryśka

Rysiek chciałby sprzedawać 12-strunowe gitary. Przygotuj zatem ołówek i umieść na diagramie klas notatki zawierające następujące informacje:

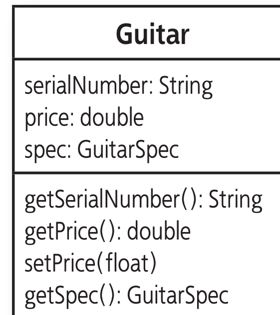
1. Do jakiej klasy dodałbyś nową właściwość o nazwie numStrings, która będzie służyć do przechowywania informacji o liczbie strun w danej gitarze?
2. Gdzie dodałabyś nową metodę o nazwie **`getNumStrings()`**, która zwraca liczbę strun w gitarze?
3. W jakich innych miejscach aplikacji powinieneś wprowadzić zmiany w kodzie, tak by podczas wyszukiwania gitar klienci Ryśka mogli określać, że interesują ich gitary 12-strunowe?

W końcu, w umieszczonych poniżej pustych wierszach zapisz wszelkie problemy związane z projektem aplikacji, które odnalazłeś podczas dodawania obsługi 12-strunowych gitar.

Dodajemy właściwość do klasy GuitarSpec, lecz jednocześnie musimy zmienić kod metody search() w klasie Inventory oraz konstruktor klasy Guitar.

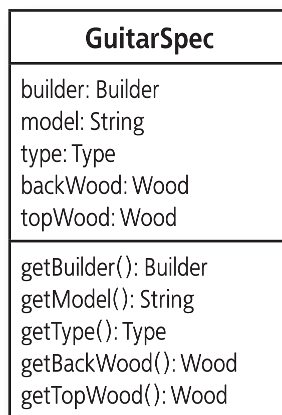
Oto co my wymyśliliśmy. Czy Ty zapisałeś coś podobnego?

Musimy zmienić konstruktor tej klasy, gdyż w jego wywołaniu są przekazywane wszystkie informacje, które następnie zapisujemy w obiekcie GuitarSpec.

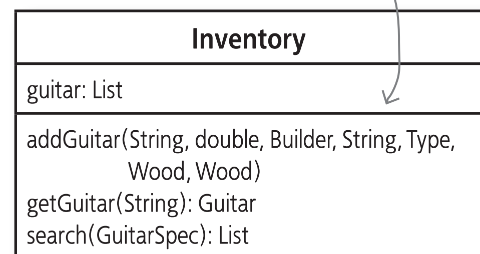


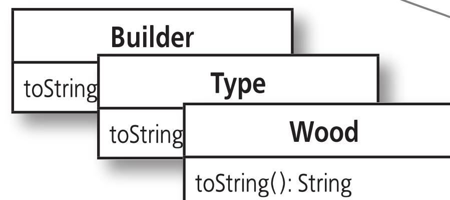
Metoda addGuitar() zdefiniowana w tej klasie także operuje na wszystkich właściwościach gitary. Dodanie nowej właściwości oznacza zatem konieczność wprowadzenia modyfikacji w tej metodzie — a to jest problem.

Do klasy GuitarSpec musimy dodać właściwość numStrings.



Do tej klasy musimy także dodać metodę getNumStrings(), która będzie zwracać liczbę strun, jaką posiada dana gitara.





Kolejny problem: musimy zmienić metodę search() klasy Inventory, aby uwzględnić nowe właściwości dodane do klasy GuitarSpec.

Czyli problem polega właśnie na tym? Dodawanie nowych właściwości do klasy GuitarSpec nie powinno zmuszać nas do wprowadzania modyfikacji w klasach Guitar oraz Inventory. Czy także ten problem możemy rozwiązać, wykorzystując hermetyzację?

Owszem — musimy hermetyzować specyfikacje gitary i lepiej izolować je od pozostałych fragmentów aplikacji Ryśka.

Choć nowe właściwości dodajemy jedynie do klasy **GuitarSpec**, to jednak przy tej okazji musimy zmodyfikować dwie inne klasy: **Guitar** oraz **Inventory**. W konstruktorze klasy **Guitar** należy przekazywać dodatkowy argument, a w metodzie **search()** klasy **Inventory** trzeba dodać jedno porównanie.

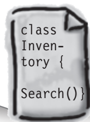
Ten konstruktor tworzy obiekt GuitarSpec, zatem każda zmiana specyfikacji gitary będzie powodować konieczność zmiany samego konstruktora.

Zastosowanie tego kodu w innym programie nie będzie prostym zadaniem. Wszystkie klasy aplikacji Ryśka są od siebie wzajemnie uzależnione i nie można zastosować jednej z nich bez jednoczesnego wykorzystania wszystkich pozostałych.

```
public Guitar (String serialNumber,
                double price,
                Builder builder,
                String model, Type type,
                Wood backWood, Wood topWood ) {
    this.serialNumber = serialNumber;
    this.price = price;
    this.spec = new GuitarSpec (builder, model,
                     type, backWood, topWood);
```

Guitar.java

```
public List search (GuitarSpec searchSpec) {
    List matchingGuitars = new LinkedList();
    for (Iterator i = guitars.iterator(); i.hasNext(); ) {
      Guitar guitar = (Guitar)i.next();
      GuitarSpec guitarSpec = guitar.getSpec();
      if (searchSpec.getBuilder() != guitarSpec.getBuilder())
        continue;
      String model = searchSpec.getModel().toLowerCase();
      if ((model != null) && (!model.equals(„")) &&
          (!model.equals(guitarSpec.getModel().toLowerCase())))
        continue;
      if (searchSpec.getType() != guitarSpec.getType())
        continue;
      if (searchSpec.getBackWood() != guitarSpec.getBackWood())
        continue;
      if (searchSpec.getTopWood() != guitarSpec.getTopWood())
        continue;
      matchingGuitars.add(guitar);
    }
    return matchingGuitars;
  }
```



Inventory.java

Zagadka projektowa

Sama wiedza, że w aplikacji Ryśka coś szwankuje, to zdecydowanie za mało. Nie wystarczy także świadomość, że konieczne jest ponowne zastosowanie hermetyzacji. Teraz *musisz* wymyślić, jak należy poprawić tę aplikację, by łatwiej można było wielokrotnie stosować jej fragmenty oraz by jej rozszerzanie nie przysparzało tylu problemów.

Problem:

Dodanie nowej właściwości do klasy **`GuitarSpec`** zmusza nas do wprowadzenia zmian także w kodzie klas **`Guitar`** oraz **`Inventory`**. Strukturę aplikacji należy zmienić w taki sposób, by dodawanie nowych właściwości do klasy **`GuitarSpec`** nie powodowało konieczności modyfikacji innych klas.

Twoje zadanie:

1. Dodaj właściwość **`numStrings`** oraz metodę **`getNumStrings()`** do klasy **`GuitarSpec`**.
2. Zmodyfikuj kod klasy **`Guitar`** w taki sposób, by właściwości obiektu **`GuitarSpec`** zostały usunięte z konstruktora tej klasy.
3. Zmień metodę **`search()`** w klasie **`Inventory`** w taki sposób, by porównywanie dwóch obiektów **`GuitarSpec`** zostało delegowane do klasy **`GuitarSpec`**, a nie było realizowane bezpośrednio w tej metodzie.
4. Zmodyfikuj kod klasy **`FindGuitarTester`**, by działała ona poprawnie ze zmienionymi klasami **`Guitar`**, **`GuitarSpec`** i **`Inventory`**, a następnie upewnij się, że wyszukiwanie gitar działa poprawnie.
5. Porównaj odpowiedzi, które podałeś, z naszymi odpowiedziami zamieszczonymi na stronie 74; następnie przygotuj się na kolejny test, który pozwoli Ci przekonać się, czy wreszcie zakończyłeś prace nad aplikacją Ryśka.

Nie wiesz, co — w tym kontekście — oznacza „delegowanie" — sprawdź...

Jedyna zmiana, jaką będziesz musiał wprowadzić w tym miejscu, dotyczy kodu tworzącego testową zawartość magazynu i polega na zastosowaniu nowego konstruktora klasy Guitar.

Nie ma niemądrych pytań

P.: Napisaliście, że powinienem „delegować” porównywanie do klasy GuitarSpec. Co to jest delegowanie?

O.: O delegowaniu mówimy w sytuacji, gdy obiekt, który musi wykonać pewną czynność, zamiast wykonać ją samemu, prosi o jej wykonanie (w całości lub częściowo) **inny** obiekt.

A zatem, w swojej zagadce projektowej nie chcesz, by metoda **`search()`** klasy **`Inventory`** porównywała dwie specyfikacje **`GuitarSpec`** bezpośrednio w swoim kodzie. Zamiast tego chcesz, by poprosiła obiekt **`GuitarSpec`** o określenie, czy specyfikacje te odpowiadają sobie. A zatem metoda **`search()`** *deleguje* porównanie do obiektu **`GuitarSpec`**.

P.: A dlaczego używamy takiego rozwiązania?

O.: Delegowanie ułatwia wielokrotne wykorzystywanie kodu aplikacji. Oprócz tego pozwala ono, by każdy obiekt koncentrował się wyłącznie na swoich możliwościach funkcjonalnych i chroni przed umieszczaniem zachowań związanych z jednym obiektem w różnych miejscach kodu aplikacji.

Jednym z najczęściej spotykanych przykładów delegowania w języku Java jest metoda **`equals()`**. Metoda ta nie stara się samodzielnie sprawdzać, czy dwa obiekty są sobie równe, lecz zamiast tego wywołuje metodę **`equals()`** jednego z porównywanych obiektów i przekazuje w jej wywołaniu drugi obiekt. Następnie pobiera jedynie wartość true lub false zwróconą przez wywołanie metody **`equals()`**.

P.: A niby w jaki sposób to delegowanie ma ułatwiać możliwości wielokrotnego wykonywania kodu?

O.: Dzięki delegowaniu każdy obiekt może sam zajmować się porównywaniem z innymi obiektami (lub wykonywaniem jakiejkolwiek innej operacji). To z kolei oznacza, że obiekty mogą być bardziej niezależne albo luźniej powiązane. Takie *luźniej powiązane* obiekty łatwiej jest zastosować w innej aplikacji, gdyż nie są one ściśle uzależnione od kodu innych obiektów.

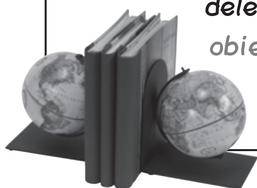
P.: Jeszcze raz — co oznacza „luźne powiązanie”?

O.: Luźne powiązanie oznacza, że każdy z obiektów używanych w aplikacji wykonuje tylko i wyłączenie swoje zadania. A zatem cała funkcjonalność aplikacji jest rozdzielona i zaimplementowana w dobrze zdefiniowanych obiektach, z których każdy w doskonały sposób realizuje przypisane mu zadania.

P.: A dlaczego to jest dobre rozwiązanie?

O.: Aplikacje tworzone przez luźno powiązane obiekty zazwyczaj są bardziej elastyczne i łatwiej można wprowadzać w nich modyfikacje. Ponieważ poszczególne obiekty nie są zwykle w żaden sposób zależne od innych obiektów, zatem możemy zmienić zachowanie jednego z obiektów bez konieczności wprowadzania modyfikacji w innych. Dzięki temu dodawanie nowych właściwości lub możliwości funkcjonalnych staje się *znacznie* prostsze.

Kącik naukowy



delegowanie — *proces polegający na tym, iż jeden obiekt zleca wykonanie pewnej operacji innemu, który wykonuje ją w imieniu pierwszego obiektu.*

Zagadka projektowa — Rozwiązanie

Sama wiedza, że w aplikacji Ryśka coś szwankuje, to zdecydowanie za mało. Nie wystarczy także świadomość, że konieczne jest ponowne zastosowanie hermetyzacji. Teraz *musisz* wymyślić, jak należy poprawić tę aplikację, by łatwiej można było wielokrotnie stosować jej fragmenty oraz by jej rozszerzanie nie przysparzało tylu problemów.

Problem:

Dodanie nowej właściwości do klasy **GuitarSpec** zmusza nas do wprowadzenia zmian także w kodzie klas **Guitar** oraz **Inventory**. Strukturę aplikacji należy zmienić w taki sposób, by dodawanie nowych właściwości do klasy **GuitarSpec** nie powodowało konieczności modyfikacji innych klas.

Twoje zadanie:

1. Dodaj właściwość **numStrings** oraz metodę **getNumStrings()** do klasy **GuitarSpec**.

```
public class GuitarSpec {

// inne właściwości
  private int numStrings;

  public GuitarSpec(Builder builder, String model, Type type,
                    int numStrings, Wood backWood, Wood topWood) {
    this.builder = builder;
    this.model = model;
    this.type = type;
    this.numStrings = numStrings;
    this.backWood = backWood;
    this.topWood = topWood;
  }

  // inne metody

  public int getNumStrings() {
    return numStrings;
  }
}
```

Nie zapomnij zmodyfikować konstruktora klasy GuitarSpec.

To było naprawdę łatwe...

class Guitar Spec { get- Num- Strings()

GuitarSpec.java

2 Zmodyfikuj kod klasy **Guitar** w taki sposób, by właściwości obiektu **GuitarSpec** zostały usunięte z konstruktora tej klasy.

```
public Guitar(String serialNumber, double price, GuitarSpec spec) {
    this.serialNumber = serialNumber;
    this.price = price;
    this.spec = spec;
}
```

Teraz nie tworzymy obiektu GuitarSpec w konstruktorze klasy Guitar, lecz przekazujemy go jako argument wywołania konstruktora.

```
class
Guitar {
  Gui-
tar()
}
```

Guitar.java

3 Zmień metodę **search()** w klasie **Inventory** w taki sposób, by porównywanie dwóch obiektów **GuitarSpec** zostało delegowane do klasy **GuitarSpec**, a nie było realizowane bezpośrednio w tej metodzie.

```
public List search(GuitarSpec searchSpec) {
  List matchingGuitars = new LinkedList();
  for (Iterator i = guitars.iterator(); i.hasNext(); ) {
    Guitar guitar = (Guitar)i.next();
    if (guitar.getSpec().matches(searchSpec))
      matchingGuitars.add(guitar);
  }
  return matchingGuitars;
}
```

Metoda search() stała się znacznie prostsza.

```
class
Inven-
tory {

search()
```

Inventory.java

Znaczna część kodu metody search() została z niej usunięta i przeniesiona do metody matches() klasy GuitarSpec.

```
public boolean matches(GuitarSpec otherSpec) {
  if (builder != otherSpec.builder)
    return false;
  if ((model != null) && (!model.equals("")) &&
      (!model.toLowerCase().equals(otherSpec.model.toLowerCase())))
    return false;
  if (type != otherSpec.type)
    return false;
  if (numStrings != otherSpec.numStrings)
    return false;
  if (backWood != otherSpec.backWood)
    return false;
  if (topWood != otherSpec.topWood)
    return false;
  return true;
}
```

Teraz dodawanie nowych właściwości do klasy GuitarSpec wymaga jedynie wprowadzania zmian w kodzie tej kasy — klasy Guitar oraz Inventory nie muszą być modyfikowane.

```
class
Guitar
Spec {
  get-
Num-
Strings()
```

GuitarSpec.java

Ostatni test aplikacji (przygotowanej do wielokrotnego używania kodu)

O rany… naprawdę wykonaliśmy kawał dobrej roboty od momentu, gdy Rysiek pokazał nam pierwszą wersję swojej aplikacji. Upewnijmy się, czy jej ostatnia wersja wciąż poprawnie działa — zarówno z punktu widzenia Ryśka, jak i jego klientów — i czy spełnia nasze wymagania dotyczące poprawnego projektu, prostego utrzymania oraz kodu, którego z łatwością będzie można wielokrotnie używać.

Oto co powinieneś zobaczyć po wykonaniu nowej wersji programu FindGuitarTester.

Wiersz polecenia

```
T:\>java FindGuitarTester
Ewo, może spodobają Ci się następujące gitary:
  Mamy w magazynie gitarę Fender model Stratocastor, jest to gitara elektryczna:
     olcha - tył i boki,
     olcha - góra.
  Możesz ją mieć za 1499.95PLN!
  ----
  Mamy w magazynie gitarę Fender model Stratocastor, jest to gitara elektryczna:
     olcha - tył i boki,
     olcha - góra.
  Możesz ją mieć za 1549.95PLN!
  ----
T:\>
```

Ewa może obejrzeć kilka wyszukanych dla niej gitar, a Rysiek — znów sprzedawać instrumenty swojej wyszukanej klienteli.

Gratulujemy!
Udało Ci się zmodyfikować niedziałającą aplikację przeszukującą magazyn Ryśka i zmienić ją w poprawnie zaprojektowane, doskonałe oprogramowanie.

Oto co zrobiliśmy

Przyjrzyjmy się pokrótce, w jaki sposób udało nam się sprawić, że obecnie aplikacja przeszukująca magazyn Ryśka działa tak dobrze:

Czy pamiętasz nasze trzy kroki? Wykonaliśmy je kolejno, by przekształcić niedziałające narzędzie przeszukujące magazyn Ryśka we w pełni funkcjonalną, dobrze zaprojektowaną aplikację.

Rozpoczęliśmy od poprawienia niektórych problemów związanych z działaniem aplikacji.

Następnie dodaliśmy do niej kilka nowych możliwości funkcjonalnych, dzięki czemu wyszukiwanie zwracało nie jedną, lecz całą listę gitar.

1. Upewnij się, że oprogramowanie robi to, czego oczekuje klient.

Rozbudowując możliwości funkcjonalne aplikacji, upewniliśmy się, że podejmowane decyzje związane z jej strukturą są właściwe i dobre.

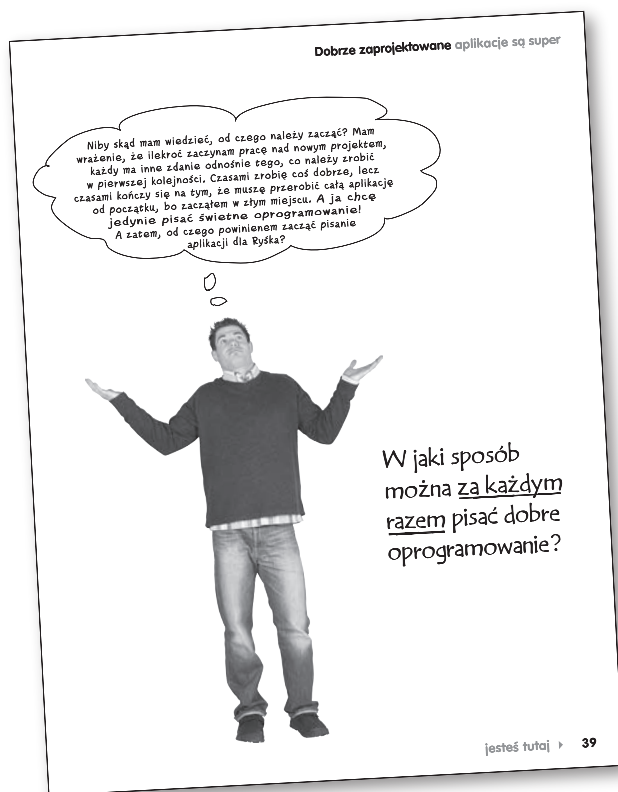
2. Zastosuj proste zasady projektowania obiektowego, by poprawić elastyczność oprogramowania.

Oprócz tego hermetyzowaliśmy właściwości gitary i zapewniliśmy, że dodawanie nowych nie będzie stanowić większego problemu.

3. Staraj się, by projekt oprogramowania zapewniał łatwość jego utrzymania i pozwalał na jego wielokrotne stosowanie.

Udało się nam nawet zastosować delegowanie, dzięki czemu obiekty używane w aplikacji stały się luźniej powiązane; to z kolei poprawiło możliwości wielokrotnego używania kodu aplikacji.

Czy pamiętasz tego biednego gościa?


Dobrze zaprojektowane aplikacje są super

Niby skąd mam wiedzieć, od czego należy zacząć? Mam wrażenie, że ilekroć zaczynam pracę nad nowym projektem, każdy ma inne zdanie odnośnie tego, co należy zrobić w pierwszej kolejności. Czasami zrobię coś dobrze, lecz czasami kończy się na tym, że muszę przerobić całą aplikację od początku, bo zacząłem w złym miejscu. A ja chcę jedynie pisać świetne oprogramowanie! A zatem, od czego powinienem zacząć pisanie aplikacji dla Ryśka?

W jaki sposób można za każdym razem pisać dobre oprogramowanie?

jesteś tutaj ▸ 39

On chciał jedynie pisać wspaniałe oprogramowanie. Jaka jest zatem odpowiedź? W jaki sposób można konsekwentnie pisać wspaniałe oprogramowanie?

Potrzebujesz w tym celu sekwencji czynności, które pozwolą Ci upewnić się, że Twoje oprogramowanie działa i jest dobrze zaprojektowane. Nie musi ona być ani długa, ani skomplikowana — wystarczy prosta, trójetapowa sekwencja, której użyliśmy do poprawienia aplikacji Ryśka i której z powodzeniem będziesz mógł używać we wszystkich swoich projektach.

Analiza i projektowanie obiektowe pomoże Ci pisać wspaniałe programy, i to za każdym razem.

Wyrażenie to będziemy określali skrótowo jako OOA&D, od angielskich słów Object-Oriented Analysis & Design.

Za każdym razem gdy w tym rozdziale wspominaliśmy o trzech krokach pozwalających na pisanie wspaniałego oprogramowania, tak naprawdę mieliśmy na myśli OOA&D — analizę i projektowanie obiektowe.

W istocie analiza i projektowanie obiektowe jest sposobem pisania oprogramowania. Jej celem jest to, by oprogramowanie robiło, co do niego należy, i było dobrze zaprojektowane. To z kolei oznacza, że kod jest elastyczny, łatwo można wprowadzać w nim zmiany, jest prosty w utrzymaniu i nadaje się do wielokrotnego używania.

OOA&D ma na celu tworzenie wspaniałego oprogramowania, a nie dodanie Ci papierkowej roboty!

Klienci są zadowoleni, kiedy ich aplikacje DZIAŁAJĄ.

Możemy uzyskać od klienta wymagania, które pozwolą nam upewnić się, że tworzona aplikacja będzie robić to, o co klient prosił. Z powodzeniem można do tego celu zastosować tak zwane przypadki użycia oraz diagramy, niemniej jednak wszystko sprowadza się do tego, by dowiedzieć się, jakich możliwości klient oczekuje od aplikacji.

O wymaganiach napiszemy w rozdziale 2.

Klienci są zadowoleni, kiedy ich aplikacje będą DZIAŁAĆ DŁUGO.

Nikt nie przepada za sytuacją, gdy aplikacja, która do tej pory działała dobrze, nagle zacznie szwankować. Jeśli dobrze zaprojektujemy nasze aplikacje, to będą one solidne i nie będą przysparzać problemów za każdym razem, gdy użytkownik zacznie ich używać w niezwykły bądź niespodziewany sposób. Klasy oraz diagramy sekwencji mogą pomóc w wykryciu usterek w projekcie aplikacji, jednak kluczowe znaczenie ma pisanie dobrze zaprojektowanego i elastycznego kodu.

Już nieco się dowiedziałeś o wrażliwych i kiepskich aplikacjach.

Klienci są zadowoleni, kiedy ich aplikacje można UAKTUALNIĆ.

Kiedy klient prosi o dodanie kilku nowych, prostych możliwości, to dla niego nie ma nic gorszego, niż usłyszeć, że wykonanie poprawek zajmie dwa tygodnie i będzie kosztowało kilkadziesiąt tysięcy złotych. Dzięki zastosowaniu technik projektowania obiektowego, takich jak hermetyzacja, składanie oraz delegowanie, tworzone oprogramowanie będzie proste w utrzymaniu i łatwe do rozszerzania oraz aktualizacji.

Chcesz poznać więcej informacji o delegowaniu, złożeniach i agregacji? Wszystkie te zagadnienia poruszymy szczegółowo najpierw w rozdziale 5., a następnie w rozdziale 8.

Programiści są zadowoleni, kiedy napisanego kodu można WIELOKROTNIE UŻYWAĆ.

Czy kiedyś napisałeś jakiś program dla jednego klienta, a następnie zauważyłeś, że po wprowadzeniu bardzo nieznacznych modyfikacji będzie on spełniał wymagania innego klienta? Wystarczy się nieco zastanowić nad tworzoną aplikacją, by uniknąć takich problemów jak wzajemne uzależnienie klas i niepotrzebne powiązania pomiędzy nimi, i w efekcie uzyskać kod, który z powodzeniem będzie można wielokrotnie stosować. Takie pojęcia jak Zasada Otwarte-Zamknięte (ang. *Open-Close Principle*, w skrócie OCP) oraz Zasada Pojedynczej Odpowiedzialności (ang. *Single Responsibility Principle*, w skrócie SRP) bardzo mogą w tym pomóc.

W rozdziale 8. przekonasz się, jak duże znaczenie i wpływ na powstający kod mają te zasady.

Programiści są zadowoleni, kiedy pisane przez nich aplikacje są ELASTYCZNE.

Czasami tylko nieznaczne zmiany i refaktoryzacja pozwalają zamienić dobrą aplikację we wspaniały framework, którego z powodzeniem będzie można używać do przeróżnych zadań. To właśnie umiejętność wprowadzania takich zmian sprawi, że powoli przestaniesz być zwyczajnym koderem i zaczniesz myśleć jak prawdziwy architekt oprogramowania (o tak… ci goście zarabiają znacznie więcej). Tu chodzi o ogarnięcie całości zagadnienia.

W rozdziałach 6. i 7. zajmiemy się umiejętnością ogólnego spojrzenia na rozwiązywany problem i tworzeniem dobrej architektury dla pisanych aplikacji.

To wszystko nazywamy właśnie analizą i projektowaniem obiektowym. Nie chodzi tu wcale o tworzenie głupkowatych diagramów… a o pisanie odlotowych aplikacji, które uszczęśliwią klientów, a Tobie zapewnią błogie poczucie wielkości i satysfakcji.

No i fantastycznie! Dzięki temu nowemu narzędziu do wyszukiwania nie mogę się opędzić od klientów. Ale swoją drogą... mam parę pomysłów na kilka nowych możliwości...

Widzisz? Już dostałeś propozycję dalszej pracy. Jednak Rysiek będzie musiał z tym poczekać aż do rozdziału 5... Mamy kilka ważniejszych problemów, którymi musimy się zająć w następnym rozdziale.

KLUCZOWE ZAGADNIENIA

- Jeśli aplikacja jest wrażliwa, byle co może doprowadzić do jej awarii.
- Dzięki zastosowaniu zasad projektowania obiektowego, takich jak hermetyzacja i delegowanie, można tworzyć znacznie bardziej elastyczne aplikacje.
- Hermetyzacja polega na dzieleniu aplikacji na logiczne części.
- O delegowaniu mówimy wtedy, gdy jeden obiekt przekazuje wykonanie pewnego zadania do innego obiektu.
- Zawsze zaczynaj prace nad projektem od określenia, czego chce klient.
- Kiedy uda Ci się już poprawnie zaimplementować podstawowe możliwości funkcjonalne aplikacji, możesz zająć się zmodyfikowaniem jej struktury i zapewnieniem jej elastyczności.
- Po zapewnieniu elastyczności projektu możesz zastosować wzorce projektowe, by dodatkowo go poprawić i ułatwić wielokrotne stosowanie kodu aplikacji.
- Odszukaj te fragmenty aplikacji, które często ulegają zmianom, i postaraj się oddzielić je od pozostałych fragmentów, które się nie zmieniają.
- Tworzenie aplikacji, które działają, lecz są nieprawidłowo zaprojektowane, zadowoli klienta, jednak Tobie przysporzy wielu zmartwień, problemów i nieprzespanych nocy, zmarnowanych na żmudne poprawianie błędów.
- Analiza i projektowanie obiektowe udostępnia metody pozwalające na tworzenie poprawnie zaprojektowanych aplikacji spełniających wymagania nie tylko klienta, lecz także programisty.

Rozwiązania ćwiczeń

DLACZEGO MAM NA TO ZWRACAĆ UWAGĘ ?

Dowiedziałeś się już całkiem dużo na temat pisania doskonałego oprogramowania, jednak wciąż jeszcze musisz się wiele nauczyć. Weź głęboki oddech i zastanów się nad niektórymi terminami i zasadami, które przedstawiliśmy w tym rozdziale. Połącz słowa umieszczone w lewej kolumnie z wyjaśnieniami celów stosowania danych technik lub zasad, znajdującymi się w kolumnie prawej.

Elastyczność

Hermetyzacja

Funkcjonalność

Wzorce projektowe

Beze mnie Twój klient nigdy nie będzie zadowolony. Bez względu na to, jak wspaniale byłaby zaprojektowana i napisana aplikacja, to właśnie ja sprawiam, że na twarzy klienta pojawia się uśmiech.

Ja wkraczam do akcji tam, gdzie chodzi o możliwości wielokrotnego wykorzystania tego samego kodu i uzyskanie pewności, że nie trzeba będzie rozwiązywać problemów, które ktoś inny rozwiązał już wcześniej.

Programiści używają mnie po to, by oddzielić od siebie fragmenty kodu, które ulegają zmianom, od tych, które pozostają takie same; dzięki temu łatwo można wprowadzać w kodzie modyfikacje — bez obawy, że cała aplikacja przestanie działać.

Programiści używają mnie po to, by oprogramowanie można było rozwijać i zmieniać bez konieczności ciągłego pisania go od samego początku. To ja sprawiam, że aplikacje stają się solidne i odporne.

Zaostrz ołówek Rozwiązanie

Co byś zmienił w tym kodzie?

W przedstawionym kodzie występuje poważny problem, Twoim zadaniem jest go odnaleźć. W poniższych pustych wierszach zapisz, na czym według Ciebie polega ten problem oraz w jaki sposób można by go rozwiązać.

Za każdym razem, gdy do klasy GuitarSpec zostanie dodana nowa właściwość bądź gdy zmieni się któraś z jej metod, konieczne będzie wprowadzenie zmian także w metodzie search() klasy Inventory. Porównywaniem powinna się zajmować klasa GuitarSpec; do niej powinniśmy także przenieść cały kod operujący na specyfikacji gitary, umieszczony obecnie w klasie Inventory.

```
public List search(GuitarSpec searchSpec) {
    List matchingGuitars = new LinkedList();
    for (Iterator i = guitars.iterator(); i.hasNext(); ) {
      Guitar guitar = (Guitar)i.next();
      GuitarSpec guitarSpec = guitar.getSpec();
      if (searchSpec.getBuilder() != guitarSpec.getBuilder())
        continue;
      String model = searchSpec.getModel().toLowerCase();
      if ((model != null) && (!model.equals("")) &&
          (!model.equals(guitarSpec.getModel().toLowerCase())))
        continue;
      if (searchSpec.getType() != guitarSpec.getType())
        continue;
      if (searchSpec.getBackWood() != guitarSpec.getBackWood())
        continue;
      if (searchSpec.getTopWood() != guitarSpec.getTopWood())
        continue;
      matchingGuitars.add(guitar);
    }
    return matchingGuitars;
  }
```

To nie jest przykład dobrego projektu. Za każdym razem, gdy do klasy GuitarSpec zostanie dodana nowa właściwość, konieczne będzie wprowadzenie zmian w tym kodzie.

```
class
Inven-
tory {

search()
```

Inventory.java

Zastanów się: czy klasa Inventory naprawdę koncentruje się na obsłudze magazynu gitar Ryśka? Czy też koncentruje się na tym, co sprawia, że dwie specyfikacje gitar — czyli dwa obiekty GuitarSpec — są sobie równe? Chcesz, by Twoje klasy skupiały się na swoich zadaniach. Porównywanie obiektów GuitarSpec jest zadaniem, którego wykonaniem powinna zajmować się klasa GuitarSpec, a nie klasa Inventory.