Poznaj zasady stosowania wzorców projektowych

Rusz głową!

Wzorce projektowe



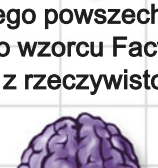
Jak unikać niebezpiecznych związków



Dlaczego powszechna opinia o wzorcu Factory mija się z rzeczywistością



Odkrywamy sekrety Mistrza Wzorców



Wzorce projektowe i nowoczesne metody nauki



Jak sieć Star Cafe podwoiła swoją wartość dzięki wzorcowi Decorator



Jak unikanie dziedziczenia zmieni Twoje życie



Eric Freeman, Elisabeth Freeman
Bert Bates, Kathy Sierra

O'REILLY®

Helion

helion kopia dla: Marek Stawarczyk mafikrpg@gmail.com

Tytuł oryginału: Head First Design Patterns

Tłumaczenie: Grzegorz Kowalczyk (wprowadzenie, rozdz. 1 – 8)
Paweł Koronkiewicz (rozdz. 9 – 14)

ISBN: 978-83-283-3359-8

© Helion S.A. 2005, 2010, 2017.

Authorized translation of the English edition © 2004 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.

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 właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte
w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej
odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne
naruszenie praw patentowych lub autorskich. Autor oraz 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. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

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

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

- Poleć książkę na Facebook.com
- Kup w wersji papierowej
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

helion kopia dla: Marek Stawarczyk mafikrpg@gmail.com

Zespołowi Gangu Czterech, którego głęboki wgląd
w opisywaną problematykę i fachowość w pracach
nad uchwyceniem i opisaniem wzorców projektowych
na zawsze zmieniły oblicze programowania
i życie programistów na całym świecie.

A tak poważniej, kiedy doczekamy się drugiego wydania?
Jak by nie patrzeć, minęło dopiero *dziesięć lat*!

Autorzy książki „Wzorce projektowe. Rusz głową!”



Elisabeth jest autorem, programistą i grafikiem komputerowym. Historia jej związków z internetem jest niemal tak długa, jak historia samej sieci. Jest współzałożycielem projektu Ada (TAP), znakomitej witryny WWW dla kobiet zajmujących się komputerami, obecnie przejętej przez ACM. W ostatnich latach pracowała jako specjalistka od mediów cyfrowych w dziale badań i rozwoju Walt Disney Company. Jednym z owoców jej pracy jest Motion, system zarządzania zawartością odpowiedzialny za rozsyłanie terabajtów filmów klientom firm Disney, ESPN i Moveis.com.

Elisabeth jest w głębi duszy informatykiem, co potwierdzają jej tytuły zdobyte na Yale University i Indiana University. Pracowała w wielu różnych dziedzinach. Zajmowała się między innymi językami programowania wizualnego, syndykacją RSS i systemami internetowymi. Jest też aktywnym uczestnikiem inicjatyw na rzecz zwiększenia udziału kobiet w informatyce i opracowuje programy mające ułatwić płci pięknej wejście w ten obszar nowoczesnego życia. Obecnie spędza wiele godzin przy swoim Maku, pracując z Javą lub Cocoa, choć jej marzeniem jest, aby pewnego dnia świat nawrócił się na korzystanie z systemu Scheme.

Elisabeth kocha wycieczki i świeże powietrze od czasów dzieciństwa, które spędzała w Szkocji. Gdy jest poza domem, jej nieodłącznym towarzyszem jest aparat fotograficzny. Jest też namiętną rowerzystką, wegetarianką i miłośniczką zwierząt.

Jej adres e-mail to beth@wickedlysmart.com.

Eric to informatyk z zacięciem do multimediów i projektowania architektury oprogramowania. Właśnie zakończył czteroletni etap wymarzonej pracy — kierowania pracami w zakresie internetowych transmisji szerokopasmowych i bezprzewodowych w Walt Disney Company — i powrócił do pisania, tworzenia ciekawego oprogramowania i poznawania największych tajemnic Javy i Maków.

Eric dużą część lat dziewięćdziesiątych spędził, pracując z Davidem Gelernterem nad projektowaniem alternatyw dla metafory pulpitu (obaj wciąż nie przestają zadawać sobie pytania: „Dlaczego plik musi mieć nazwę?”). Wyniki tych badań posłużyły jako podstawa do pracy doktorskiej na Yale University, złożonej w 1997 roku. Współuczestniczył też w zakładaniu firmy Mirror Worlds Technologies (obecnie przejętej), w której powstała komercyjna wersja omawianego w pracy narzędzia, Lifestreams.

Wcześniej Eric pisał oprogramowanie dla sieci i superkomputerów. Możesz go znać z książek takich, jak JavaSpaces, Principles, Patterns and Practice. Eric z nostalgią wspomina implementowanie systemów przestrzeni krotek na komputerach Thinking Machine CM-5 i budowanie pierwszych internetowych systemów informacyjnych dla NASA w okresie późnych lat osiemdziesiątych.

Eric mieszka obecnie na prawdziwej pustyni, niedaleko Santa Fe. Gdy nie pisze książek, artykułów czy programów, szybciej znajdziemy go dopracowującego swoje pomysły niż siedzącego przed telewizorem czy ożywiającego Dragon’s Lair (grę z początku lat osiemdziesiątych). Nie gardzi też długimi wieczorami w świetle księżyca, spędzanymi w roli didżeja techno.

Erica jest dostępny pod adresem *eric@wickedlysmart.com*. Prowadzi też blog — *http://www.ericfreeman.com*.

Twórcy serii „Rusz głową!"
(którzy pomogli również przy tej książce)



Kathy interesowała się teorią nauczania od czasu, gdy rozpoczęła pracę jako projektant gier (pisała gry dla firm Virgin, MGM i Amblin'). Gdy uczyła tworzyć prezentacje multimedialne i internetowe w ramach programu Entertainment Studies (prowadzonego przez UCLA Extension), opracowała podstawy koncepcji graficznej serii *Rusz głową!*. W ostatnich latach była pracownikiem szkoleniowym Sun Microsystems, gdzie przygotowywała wykładowców języka Java tej firmy do prowadzenia zajęć związanych z najnowszymi technologiami tego języka. Brała też udział w opracowywaniu egzaminów certyfikacyjnych. Razem z Bertem Batesem aktywnie wykorzystywała metodykę *Rusz głową!* w nauczaniu tysięcy programistów. Kathy jest założycielem witryny javaranch.com, która w 2003 i 2004 roku zdobyła nagrodę Jolt Cola Productivity Award magazynu Software Development. Obecnie można ją spotkać na kursach Java Jam Geek Cruise (geekcruises.com), gdzie uczy programowania w języku Java.

Niedawno przeprowadziła się z Kalifornii do stanu Kolorado, gdzie musi uczyć się nowych słów, takich jak „ice scraper" (drapak do lodu) i „fleece" („wełna" w języku górali). Najbardziej zachwyca się tamtejszymi błyskawicami.

Co lubi? Bieganie, jazdę na nartach, deskorolce i swoim islandzkim koniu, a także dziwne dziedziny nauki. Nie lubi entropii.

Bardzo aktywnie uczestniczy w dyskusjach w witrynie javaranch.com, od czasu do czasu dopisuje też co nieco do blogu java.net. Jej adres e-mail to *kathy@wickedlysmart.com*.

Bert ma za sobą wiele lat pracy w charakterze programisty i projektanta oprogramowania. Jego zainteresowanie teorią nauczania i technologiami wspomagającymi szkolenie wywodzi się z dziesięcioletniej praktyki w dziedzinie AI. Od dawna już pomaga swoim klientom doskonalić umiejętności w zakresie programowania. W ostatnim okresie kierował zespołem programistów, który przygotowywał część egzaminów certyfikacyjnych firmy Sun.

Pierwszą dekadę swojej kariery programisty poświęcił rozrzuconym po całym świecie sieciom telewizyjnym i radiowym, takim jak Radio New Zealand, Weather Channel i Arts & Entertainment Network (A & E). Jednym z jego ulubionych projektów była budowa kompletnej symulacji systemu kolejowego dla Union Pacific Railroad.

Bert to doświadczony, beznadziejnie uzależniony gracz w go. Programowi, który gra w tę grę, poświęcił ogromną ilość czasu. Należy też wspomnieć, że gra na gitarze, a ostatnio próbuje swoich sił z banjo.

Kontaktu z nim można szukać przez witrynę javaranch.com i serwer go IGS. Można też napisać bezpośrednio, pod adres terrapin@wickedlysmart.com.

Spis treści (skrócony)

	Wprowadzenie	21
1.	Witamy w krainie wzorców projektowych: *wprowadzenie*	33
2.	Jak sprawić, by Twoje obiekty były zawsze dobrze poinformowane: *Wzorzec Obserwator*	67
3.	Dekorowanie zachowania obiektów: *Wzorzec Dekorator*	109
4.	Pizzeria zorientowana obiektowo: *Wzorzec Fabryka*	139
5.	Obiekty jedyne w swoim rodzaju: *Wzorzec Singleton*	197
6.	Hermetyzacja wywołań: *Wzorzec Polecenie*	217
7.	Zdolność do adaptacji: *Wzorce Adapter oraz Fasada*	259
8.	Hermetyzacja algorytmów: *Wzorzec Metoda Szablonowa*	297
9.	Zarządzanie kolekcjami: *Wzorce Iterator i Kompozyt*	335
10.	Stan obiektu: *Wzorzec Stan*	403
11.	Kontrola dostępu do obiektu: *Wzorzec Proxy*	447
12.	Łączenie wzorców: *Wzorce złożone*	517
13.	Wzorce projektowe w praktyce: *Nowe życie z wzorcami*	595
14.	Dodatek: *inne wzorce*	629
	Skorowidz	649

Spis treści (z prawdziwego zdarzenia)

Wprowadzenie

Twój mózg jest skoncentrowany na wzorcach projektowych.

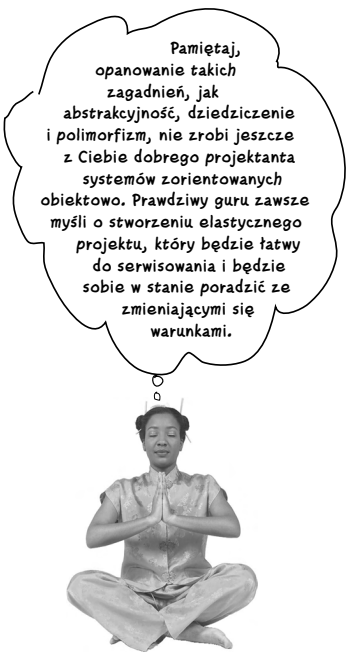
W tym rozdziale *Ty* starasz się czegoś dowiedzieć, a Twój *mózg* robi Ci przysługę i nie przykłada się do *zapamiętywania* zdobywanej wiedzy. Twój mózg myśli sobie: „Lepiej zostawię miejsce w pamięci na bardziej istotne informacje, na przykład: jakich dzikich zwierząt należy unikać bądź czy jeżdżenie nago na snowboardzie jest dobrym pomysłem". A zatem w jaki sposób możesz przekonać swój mózg, że Twoje życie zależy od poznania wzorców projektowych?

Dla kogo przeznaczona jest ta książka?	22
Wiemy także, co sobie myśli Twój mózg	23
Metapoznanie	25
Zmuś swój mózg do posłuszeństwa	27
Zespół recenzentów technicznych	30
Podziękowania	31

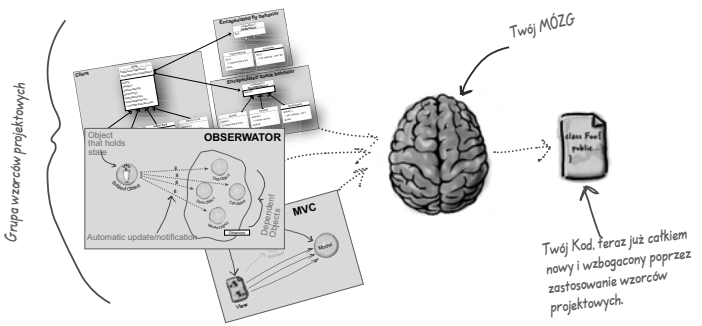
Wprowadzenie do wzorców projektowych

1 Witamy w krainie wzorców projektowych

Ktoś rozwiązał już Twoje problemy. W tym rozdziale dowiesz się, dlaczego (i w jaki sposób) możesz wykorzystać wiedzę i doświadczenia zdobyte przez innych projektantów i programistów, którzy podczas pracy nad różnymi projektami zmuszeni byli wstąpić na pełną zdradliwych pułapek ścieżkę i — co najważniejsze — udało im się przeżyć tę wyprawę. Zanim dobrniemy do końca rozdziału, rzucimy okiem na sposoby wykorzystywania wzorców projektowych i przedstawimy ich zalety, poznamy kilka podstawowych zasad projektowania zorientowanego obiektowo, a także omówimy sposób działania przykładowego wzorca. Najlepszą metodą zastosowania wzorca jest *załadowanie go bezpośrednio do Twojego mózgu*, a następnie *zlokalizowanie* obszarów w obrębie projektowanych rozwiązań oraz istniejących aplikacji, w których możesz go *zastosować*. Pracując z wzorcami projektowymi, zamiast wielokrotnego wykorzystywania tych samych fragmentów kodu wielokrotnie wykorzystujesz swoje *doświadczenia*.



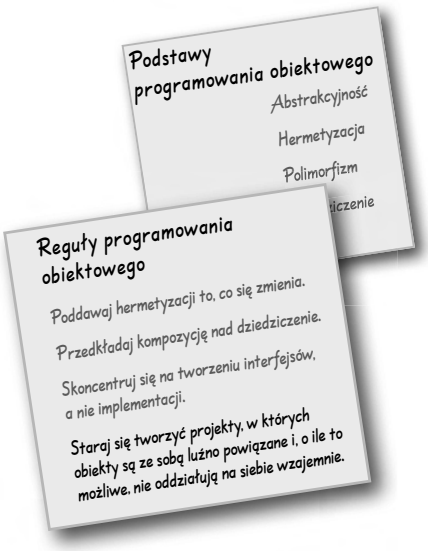
Prosta aplikacja o nazwie SymulatorKaczki	34
Jacek rozmyśla o dziedziczeniu…	37
A może by tak interfejs?	38
Jedyny pewny element w procesie tworzenia oprogramowania	40
Oddzielanie tego, co się zmienia, od tego, co pozostaje niezmienione	42
Projektowanie zachowania Kaczki	43
Testowanie kodu klasy Kaczka	50
Dynamiczne ustawianie zachowania	52
Wielki diagram „ukrytych” zachowań	54
Relacja MA może być lepsza niż JEST	55
Rozmawiając o wzorcach projektowania	56
Potęga wspólnego słownika wzorców	60
W jaki sposób mogę wykorzystywać wzorce projektowe?	61
Twoja skrzynka narzędziowa	64
Rozwiązania ćwiczeń	66



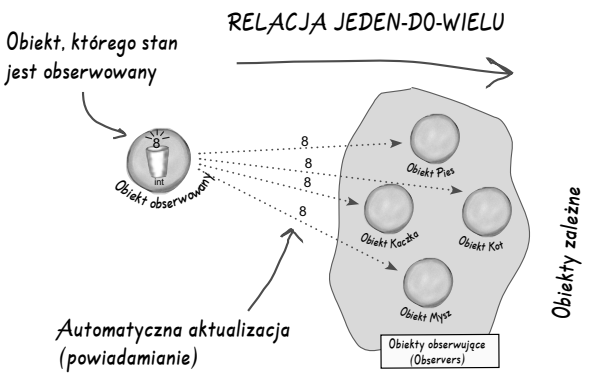
Wzorzec Obserwator

2 Jak sprawić, by Twoje obiekty były zawsze dobrze poinformowane

Nie przegap okazji, kiedy dzieje się coś naprawdę ciekawego! Przedstawimy Ci wzorzec, który potrafi poinformować inne obiekty o tym, że wydarzyło się coś, czym powinny się zająć. Co ciekawe, obiekty mogą nawet samodzielnie decydować w czasie działania programu o tym, czy chcą być informowane o takich wydarzeniach. Wzorzec Obserwator jest jednym z najczęściej wykorzystywanych wzorców w pakiecie JDK (ang. *Java Development Kit*), a co najważniejsze — jest wręcz niewiarygodnie użyteczny. W niniejszym rozdziale rzucimy również okiem na relacje typu jeden-do-wielu oraz tzw. luźne związki (tak, to prawda, napisaliśmy „luźne związki"). Korzystając z wzorca Obserwator, z pewnością odmienisz swoje życie.



Aplikacja sprawdzająca warunki pogodowe	69
Spotkanie z wzorcem Obserwator	74
Wydawca + Prenumerator = wzorzec Obserwator	75
Pięciominutowe przedstawienie — obserwowany kontra obserwujący	78
Definicja wzorca Obserwator	81
Siła luźnych zależności	83
Projektowanie stacji meteorologicznej	86
Implementacja stacji meteorologicznej	87
Java — zastosowanie wbudowanego wzorca Obserwator	94
Ciemna strona klasy java.util.Observable	101
Twoja skrzynka narzędziowa	104
Rozwiązania ćwiczeń	107



Wzorzec Dekorator

3 Dekorowanie zachowania obiektów

W zasadzie niniejszy rozdział możemy równie dobrze zatytułować „Otwieranie oczu programistom z nadmiernymi skłonnościami do używania dziedziczenia".

W tym rozdziale spróbujemy krytycznie przyjrzeć się zwyczajowym skłonnościom do nadużywania mechanizmu dziedziczenia oraz nauczymy Cię sposobów dekorowania zachowania klas w czasie działania programu przy użyciu pewnej formy kompozycji obiektów. Dlaczego? Po zapoznaniu się z technikami dekoracji zachowania klas będziesz mógł wyposażać swoje (i nie tylko) obiekty w nowe możliwości bez konieczności dokonywania jakichkolwiek modyfikacji w kodzie klas podstawowych.

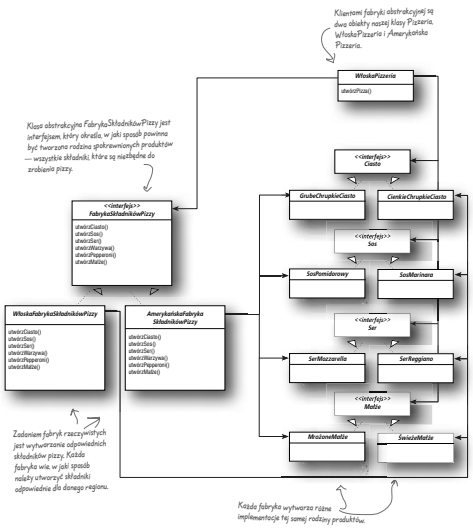
Witamy w „Star Cafè" 110
Reguła otwarte-zamknięte 116
Spotkanie z wzorcem Dekorator 118
Konstruowanie zamówienia przy użyciu Dekoratorów 119
Definicja wzorca Dekorator 121
Dekorujemy nasze Napoje 122
Tworzymy kod aplikacji „Star Cafè" 125
Dekoratory w świecie rzeczywistym: obsługa wejścia-wyjścia w języku Java 130
Tworzenie własnych dekoratorów obsługi wejścia-wyjścia 132
Twoja skrzynka narzędziowa 135
Rozwiązania ćwiczeń 136



Wzorzec Fabryka

4 Pizzeria zorientowana obiektowo

Przygotuj się do stworzenia kilku projektów, w których zastosujemy luźne powiązania pomiędzy poszczególnymi obiektami. Stworzenie nowego obiektu to dużo więcej niż tylko proste zastosowanie operatora new. Niebawem przekonasz się, że proces ten jest operacją, która nie zawsze powinna być publicznie dostępna, a co więcej, jest operacją, która często może prowadzić do poważnych problemów z powiązaniami międzyobiektowymi. A tego byś nie chciał, prawda? Przekonaj się, w jaki sposób wzorzec Factory może uratować Cię z takiej opresji.



Kiedy widzisz „nowy” obiekt, myśl o nim jako o „konkretnym”	140
Pizza w Obiektowie	142
Hermetyzacja procesu tworzenia obiektów	144
Budujemy prostą fabrykę pizzy	145
Tworzymy definicję „wzorca” Simple Factory	147
Nowa struktura Pizzerii	150
Zezwalamy klasom podrzędnym na podejmowanie decyzji	151
Tworzymy Pizzerię	153
Deklarowanie metody typu Fabryka	155
Spotkanie z wzorcem Metoda Fabrykująca	161
Równoległa hierarchia klas	162
Definicja wzorca Metoda Fabrykująca	164
Pizzeria mocno uzależniona	167
Sprawdzamy zależności pomiędzy obiektami	168
Zastosowanie reguły DIP	170
A w międzyczasie, na zapleczu Pizzerii…	174
Rodziny składników…	175
Budujemy fabryki składników pizzy	176
Fabryka Abstrakcyjna	183
Za kulisami	184
Definicja wzorca Fabryka Abstrakcyjna	186
Porównanie Metody Fabrykującej oraz Fabryki Abstrakcyjnej	190
Twoja skrzynka narzędziowa	192
Rozwiązania ćwiczeń	193



Wzorzec Singleton

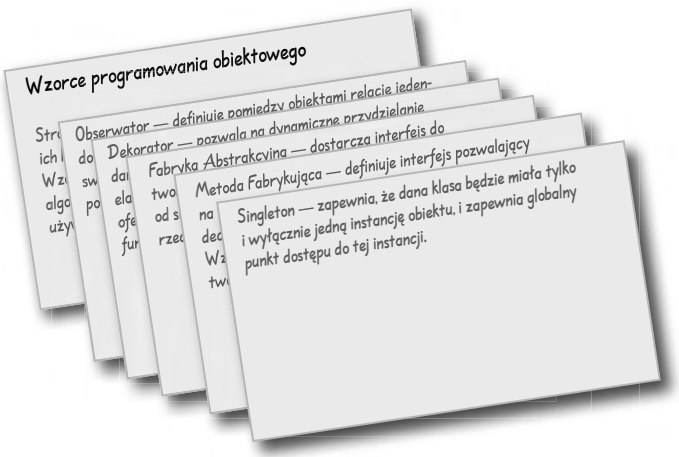
5 Obiekty jedyne w swoim rodzaju

Kolejnym przystankiem w naszej podróży jest wzorzec Singleton, czyli nasza przepustka do kreowania jedynych w swoim rodzaju obiektów, posiadających tylko jedną instancję. Być może ucieszysz się na wieść o tym, że Singleton jest najprostszym z istniejących wzorców projektowych (przynajmniej pod względem kategorii stopnia złożoności jego diagramu klas); jak by na to nie patrzeć, jego diagram składa się tylko z jednej klasy! Ale nie wpadaj w euforię; niezależnie od prostoty diagramu klas tego wzorca na drodze prowadzącej do jego implementacji napotkamy całkiem sporo wybojów i dziur. Lepiej zapnij mocno pasy — to nie będzie takie proste, jak mogłoby się wydawać.



Hershey, PA

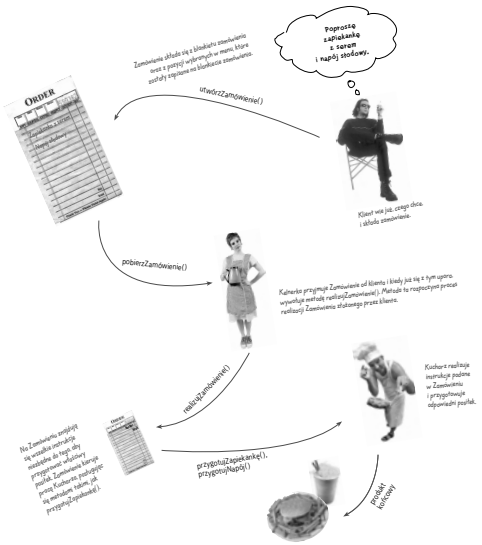
Jeden i tylko jeden	198
Mały Singleton	199
Analiza klasycznej implementacji wzorca Singleton	201
Wyznania obiektu Singleton	202
Fabryka czekolady	203
Definicja wzorca Singleton	205
Ups, mamy problem…	206
Zostań wirtualną maszyną Java	207
Jak sobie radzić z wielowątkowością?	208
Wzorzec Singleton — pytania i odpowiedzi	212
Twoja skrzynka narzędziowa	214
Rozwiązania ćwiczeń	216



Wzorzec Polecenie

6 Hermetyzacja wywołań

W niniejszym rozdziale przeniesiemy hermetyzację na zupełnie nowy poziom: mamy zamiar dokonać hermetyzacji wywołań metod. Zgadza się, dzięki hermetyzacji wywołań metod możemy wykrystalizować pewne fragmenty obliczeń tak, że obiekt wywołujący obliczenia nie musi się martwić, w jaki sposób je wykonać; po prostu wykorzystuje naszą metodę. Z takimi hermetyzowanymi wywołaniami metod możemy również dokonywać wielu zadziwiająco sprytnych operacji, takich jak na przykład zapisywanie ich do dzienników czy też ponowne wykorzystywanie w celu zaimplementowania mechanizmu Cofnij (ang. *Undo*) w naszej aplikacji.

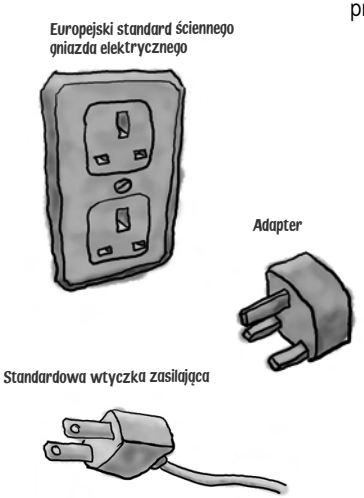


Automatyka w domu i zagrodzie	218
Mamy nową zabawkę! Sprawdzamy, jak działa SuperPilot…	219
Co zawiera otrzymany dysk CD-R?	220
A w międzyczasie w naszym barze szybkiej obsługi…	223
Przyjrzyjmy się nieco dokładniej wzajemnym interakcjom…	224
Zadania i zakresy odpowiedzialności	225
Od Baru do wzorca Polecenie	227
Nasze pierwsze POLECENIE	229
Definicja wzorca Polecenie	232
Wzorzec Command i SuperPilot	234
Implementujemy SuperPilota	236
Sprawdzamy możliwości naszego SuperPilota	238
Nadszedł wreszcie czas, aby utworzyć trochę dokumentacji…	241
Implementacja mechanizmu wycofywania przy użyciu stanów	246
Każdy pilot powinien posiadać tryb Impreza!	250
Zastosowanie makropoleceń	251
Kolejne zastosowania wzorca Polecenie — kolejkowanie żądań	254
Kolejne zastosowania wzorca Polecenie — żądania rejestracji	255
Twoja skrzynka narzędziowa	256
Rozwiązania ćwiczeń	258

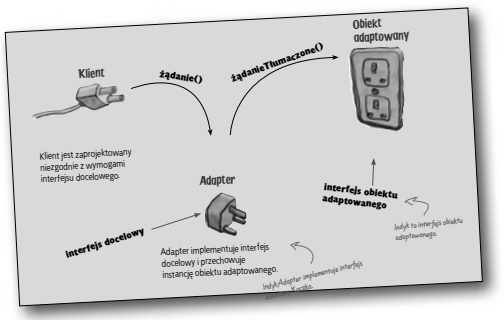
Wzorce Adapter oraz Fasada

7 Zdolność do adaptacji

W niniejszym rozdziale mamy zamiar dokonać paru niesamowitych wyczynów z dziedziny rzeczy niemożliwych, takich jak na przykład włożenie kwadratowego kołka do okrągłego otworu. Brzmi nierealnie? Nie wtedy, kiedy mamy pod ręką odpowiednie wzorce projektowe. Pamiętasz wzorzec Dekorator? Podczas pracy z nim owijaliśmy obiekty innymi obiektami tak, aby nadać im nowe zachowania. Teraz mamy zamiar postępować tak samo, ale w nieco innym celu: chcemy sprawić, by ich interfejsy wyglądały jak coś, czym nie są. Dlaczego jednak mielibyśmy to robić? Na przykład po to, aby zaadaptować projekt oczekujący danego interfejsu do klasy, która implementuje zupełnie inny interfejs. To jeszcze nie wszystko; skoro już jesteśmy przy tym temacie, przyjrzymy się również innemu wzorcowi, który owija obiekty w celu uproszczenia ich interfejsów.



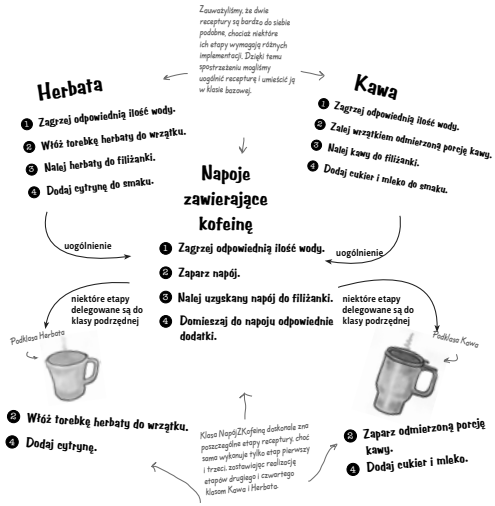
Adaptery są wśród nas	260
Adaptery zorientowane obiektowo	261
Wzorzec Adapter bez tajemnic	265
Definicja wzorca Adapter	267
Adaptery obiektów i klas	268
Temat dzisiejszej wieczornej pogawędki: Adapter obiektów i Adapter klas	271
Adaptery w świecie rzeczywistym	272
Adaptujemy interfejs Enumeration do wymagań interfejsu Iterator	273
Temat dzisiejszej wieczornej pogawędki: wzorce Dekorator i Adapter	276
Nie ma to jak kino domowe	279
Światła, kamera, fasada!	282
Konstruujemy fasadę naszego systemu kina domowego	285
Definicja wzorca Fasada	288
Reguła ograniczania interakcji	289
Twoja skrzynka narzędziowa	294
Rozwiązania ćwiczeń	296



Wzorzec Metoda Szablonowa

8 Hermetyzacja algorytmów

Jesteśmy jak w transie: hermetyzowaliśmy już proces tworzenia obiektów, wywołania metod, złożone interfejsy, kaczki, indyki, pizze… ciekawe, co będzie następne? Otóż teraz mamy zamiar zająć się hermetyzacją fragmentów algorytmów, tak aby klasy podrzędne mogły „podczepiać się" w różnych miejscach wykonywanych obliczeń. Co więcej, zajmiemy się również regułą projektowania, której korzenie wywodzą się w prostej linii z... Hollywood.

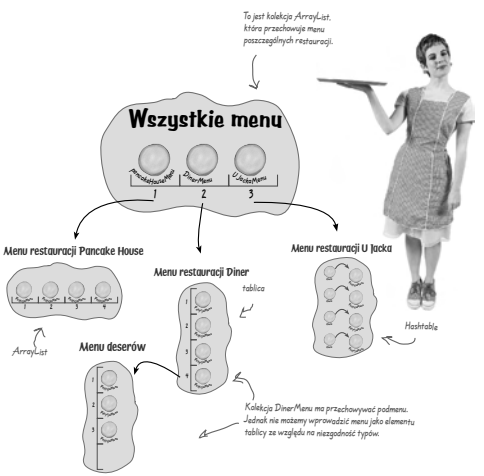


Tworzymy klasy reprezentujące kawę i herbatę (w języku Java)	299
Kawa i herbata, czyli klasy abstrakcyjne	302
Ciągniemy nasz projekt o krok dalej…	303
Wydobywanie metody recepturaParzenia()	304
Czego już dokonaliśmy?	307
Spotkanie z wzorcem Metoda Szablonowa	308
Zróbmy sobie herbatę…	309
Co nam daje zastosowanie metody szablonowej?	310
Definicja wzorca Metoda Szablonowa	311
Bliskie spotkania z kodem aplikacji	312
Haczyk na wzorzec Metoda Szablonowa…	314
Zastosowanie haczyka	315
Testujemy naszą aplikację	316
Reguła Hollywood	318
Reguła Hollywood a wzorzec Metoda Szablonowa	319
Wzorzec Metoda Szablonowa w głębokiej kniei…	321
Sortowanie przy użyciu wzorca Metoda Szablonowa	322
A teraz musimy posortować trochę kaczek…	323
Porównywanie kaczek z innymi kaczkami	324
Robimy maszynę do sortowania kaczek	326
Zabawy z ramkami	328
Aplety Java	329
Temat dzisiejszej wieczornej pogawędki: wzorce Metoda Szablonowa oraz Strategia	330
Twoja skrzynka narzędziowa	332
Rozwiązania ćwiczeń	333

Wzorce Iterator i Kompozyt

9 Zarządzanie kolekcjami

Jest wiele sposobów grupowania obiektów w kolekcje. Można utworzyć obiekty Array, Stack, List, Hashtable. Każdy z nich ma swoje zalety i wady. Jednak w pewnym momencie klient rozpocznie iteracyjne przetwarzanie elementów kolekcji. Czy wtedy ujawnisz mu swoją implementację? Mam nadzieję, że nie. To nie byłoby profesjonalne. Nie musisz się jednak obawiać, Twoja kariera zawodowa nie jest zagrożona. W tym rozdziale przedstawimy metodę, która umożliwia klientom przetwarzanie iteracyjne bez wiedzy o tym, jak obiekty są przechowywane. Przedstawimy też technikę tworzenia *superkolekcji* (ang. *super collections*) obiektów, które pozwalają na obsługę bardzo rozbudowanych struktur danych. Będziemy także pisać o odpowiedzialności obiektów.



Fuzja restauracji Diner i Pancake House	336
Implementacje menu Łukasza i Miłosza	338
Czy można hermetyzować iteracje?	343
Wzorzec Iterator	345
Wiązanie iteratora z obiektem menu	347
Co już mamy… Szersze spojrzenie na kod naszego projektu	351
Uproszczenia po wprowadzeniu interfejsu java.util.Iterator	353
Jaki jest efekt końcowy?	355
Definicja wzorca Iterator	356
Jeden zakres odpowiedzialności	359
Iteratory i kolekcje	368
Iteratory i kolekcje w języku Java 5	369
I gdy już miało być tak dobrze…	373
Definicja wzorca Kompozyt	376
Projektujemy menu oparte na wzorcu Kompozyt	379
Implementacja klasy Menu	382
Powracamy do iteratora	388
IteratorPusty	392
Wzorce Iterator i Kompozyt razem…	394
Twoja skrzynka narzędziowa	399
Rozwiązania ćwiczeń	400

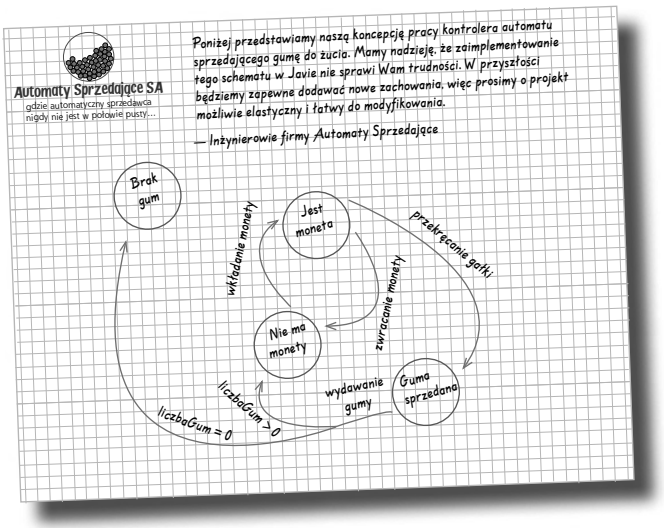
Wzorzec Stan

10 Stan obiektu

Mało znany fakt: wzorce Strategy i State to bliźniaki, rozdzielone zaraz po narodzinach. Jak już wiemy, wzorzec Strategy umożliwił przeprowadzenie wielu niezwykle udanych przedsięwzięć opartych na zamiennie stosowanych algorytmach. Wzorzec State ma inną rolę. Jest nią wspomaganie obiektów w kontrolowaniu ich własnych zachowań poprzez wewnętrzną zmianę stanu. Łatwo usłyszeć, jak mówi swoim podopiecznym: „Powtarzaj za mną: jestem wystarczająco zdolny, jestem wystarczająco dobry, dam radę to zrobić...".



Krótka narada	405
Maszyny stanowe 101	406
Piszemy kod	408
Wiedziałeś, że to jest blisko… zmiana!	412
Kłopotliwy STAN rzeczy…	414
Definiowanie interfejsów i klas reprezentacji stanu	417
Implementowanie klas Stan	419
Nowa wersja automatu sprzedającego	420
Definicja wzorca Stan	428
Wzorzec Stan kontra wzorzec Strategia	429
Wzorzec Stan, weryfikacja projektu	435
Niemal zapomnieliśmy!	438
Twoja skrzynka narzędziowa	441
Rozwiązania ćwiczeń	442



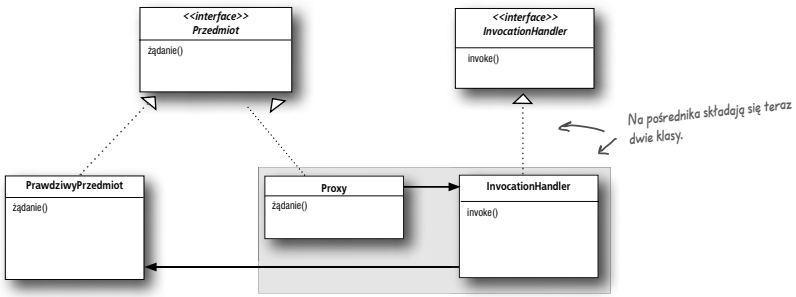
Wzorzec Proxy

11 Kontrola dostępu do obiektu

Próbowałeś kiedyś stosować metodę „dobrego i złego"? Ty jesteś tym dobrym, który zrobi wszystko, o co się go poprosi, który jest zawsze miły i uprzejmy. Nie chcesz jednak, żeby *każdy* mógł prosić o Twoje usługi. To jest miejsce dla „złego", który będzie *kontrolował dostęp* do Ciebie. Takie jest właśnie zadanie pośredników (ang. *proxy*) w modelu obiektowym — kontrolowanie dostępu i zarządzanie nim. Jak się przekonamy, istnieje *bardzo wiele* schematów takiego pośrednictwa. Obiekty Proxy mogą przekazywać wywoływanie metody obiektowi w innym węźle internetu; bywa też, że zastępują wyjątkowo leniwe obiekty.



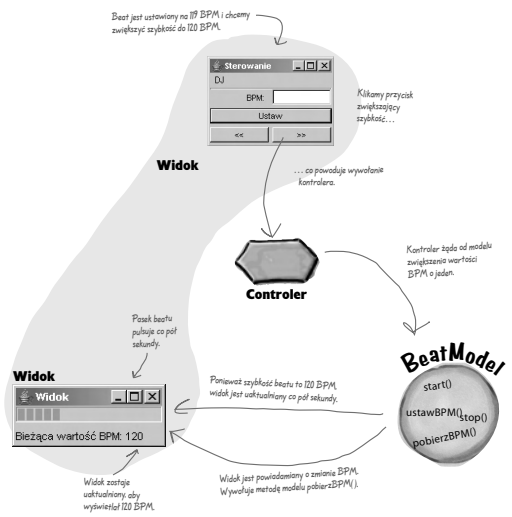
Kontrolowanie stanu automatów sprzedających	448
Rola „zdalnego pośrednika"	452
RMI — wycieczka z przewodnikiem	455
Zdalny pośrednik automatu sprzedającego	468
Pośrednik zdalny, za kulisami	476
Definicja wzorca Proxy	478
Pośrednik wirtualny	480
Projektowanie wirtualnego pośrednika do wyświetlania okładek	482
Pośrednik wirtualny, za kulisami	488
Wykorzystanie mechanizmów Java API	492
Teatrzyk — ochrona przedmiotów	496
Budowanie dynamicznego pośrednika	497
ZOO pośredników	506
Twoja skrzynka narzędziowa	508
Rozwiązania ćwiczeń	509



Wzorce złożone

12 Łączenie wzorców

Przyszłoby Ci do głowy, że wzorce mogą pracować razem? Byliśmy już świadkami wielu niespokojnych „Pogawędek przy kominku" (a ominął Cię „Death Match" wzorców, który wydawca kazał wyrzucić) — czy wsłuchując się w ich ton, można jeszcze liczyć na to, że wzorce będą ze sobą współpracować? Możesz wierzyć lub nie, ale najbardziej wyszukane projekty obiektowe wykorzystują wiele wzorców jednocześnie. Przygotuj się na kolejny poziom wiedzy o wzorcach projektowych. Czas na wzorce złożone.



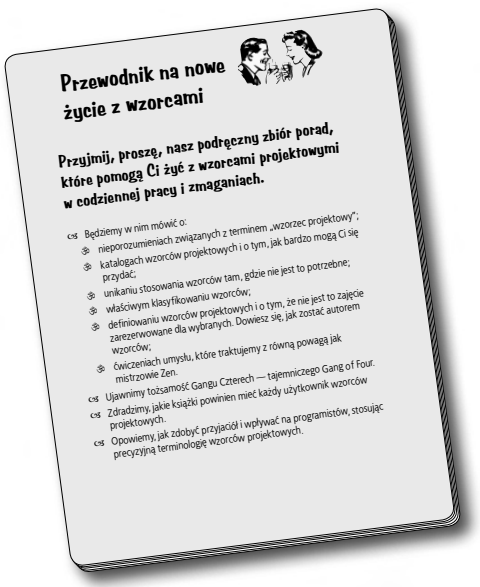
Wzorzec złożony	518
Powrót kaczek	519
Potrzebujemy adaptera gęsi	522
Wprowadzamy zliczanie kwaknięć	524
Fabryka produkująca kaczki	526
Tworzymy stado kaczek	531
Przygotowanie interfejsu Observable	534
Co zrobiliśmy?	541
Widok z lotu kaczki — diagram klas	542
Model-Widok-Kontroler — piosenka	544
Kluczem do schematu MVC będą wzorce projektowe	546
Spojrzenie na schemat Model-Widok-Kontroler przez pryzmat wzorców	550
Wykorzystujemy MVC do sterowania beatem...	552
Piszemy kod elementów	555
Widok	557
A teraz kontroler	560
Eksplorujemy możliwości wzorca Strategia	563
Adaptowanie modelu	564
Nowy kontroler — SerceKontroler	565
Wzorzec MVC i sieć WWW	567
Model 2 a wzorce projektowe	575
Twoja skrzynka narzędziowa	578
Rozwiązania ćwiczeń	579



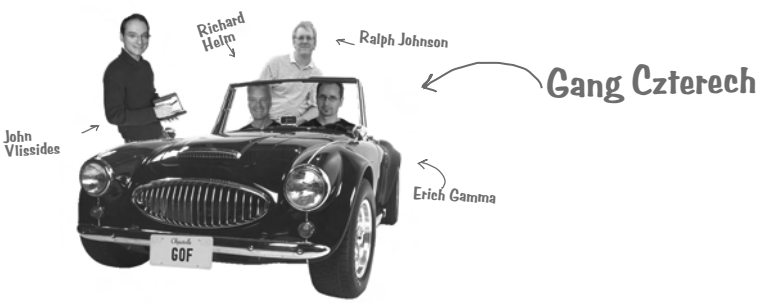
Nowe życie z wzorcami

13 Wzorce projektowe w praktyce

Ach, jesteś już gotowy na spotkanie z nowym wspaniałym światem pełnym wzorców projektowych… Ale zanim rozpoczniesz wędrówkę ku nowym horyzontom, poświęć chwilę na przeczytanie rozdziału poświęconego pewnym szczególnym kwestiom, które pojawiają się, gdy rozpoczynasz stosowanie wzorców w codziennej pracy. Nie wszędzie jest tak pięknie, jak w Obiektowie. Przygotowaliśmy więc mały przewodnik, który pomoże Ci odnaleźć się w twardej rzeczywistości...



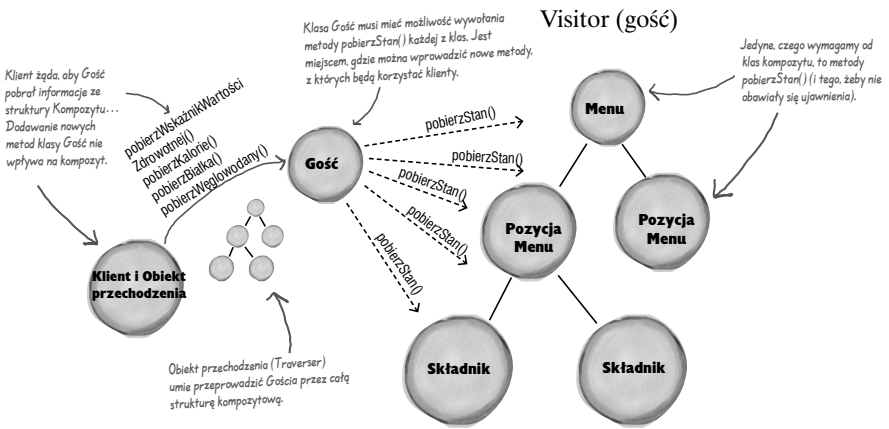
Przewodnik na nowe życie z wzorcami	596
Definicja wzorca projektowego	597
Drugie spojrzenie na definicję wzorca	599
Niech moc będzie z Tobą	600
Katalog wzorców	601
Jak tworzyć wzorce	604
Zostać autorem wzorców projektowych	605
Porządkowanie wzorców projektowych	607
Myślenie wzorcami	612
Głowa pełna wzorców	615
Nie zapominaj o potędze jednolitego słownictwa	617
Pięć podstawowych sposobów promowania Twojego słownictwa	618
Gang Czterech w Obiektowie	619
Podróż dopiero się zaczyna…	620
Inne źródła informacji o wzorcach	621
ZOO pełne wzorców	622
Walka ze złem przy użyciu antywzorców	624
Twoja skrzynka narzędziowa	626
Opuszczamy Obiektowo…	627



14 Dodatek — inne wzorce

Nie wszyscy mogą być sławni. Przez ostatnie dziesięć lat wiele się w świecie wzorców zmieniło. Od czasu pierwszego wydania książki *Design Patterns: Elements of Reusable Object-Oriented Software (Wzorce projektowe)* programiści wykorzystali opisane w niej schematy tysiące razy. Wzorce, które zebraliśmy w tym dodatku, to dopracowane, kompletne, „oficjalne" wzorce grupy GoF. Różnią się od wcześniej opisanych tylko tym, że nie spotkamy ich tak często jak tych, którym poświęciliśmy całe rozdziały. Nie umniejsza to ich zalet i nie powinno zniechęcać do ich stosowania tam, gdzie wymaga tego sytuacja. Celem niniejszego dodatku jest zapewnienie Ci szerszej orientacji w najłatwiej dostępnych zasobach zgromadzonej przez lata wiedzy.

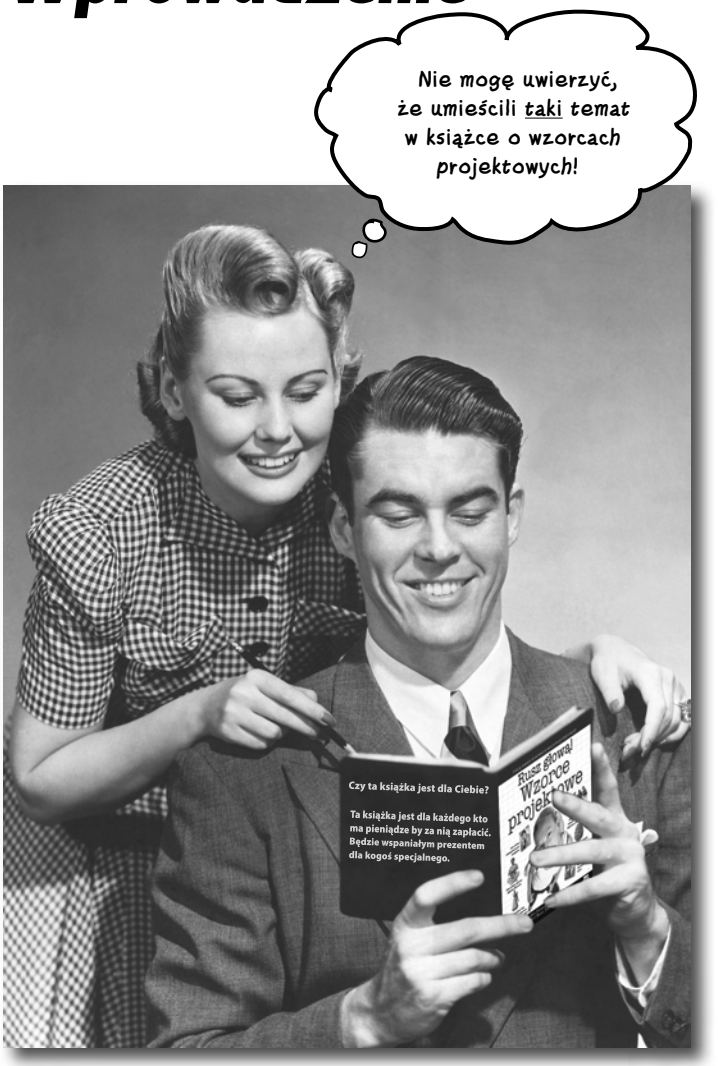
Bridge (most) 630
Builder (budowniczy) 632
Chain of Responsibility (łańcuch odpowiedzialności) 634
Flyweight (waga piórkowa) 636
Interpreter (interpreter) 638
Mediator (mediator) 640
Memento (memento) 642
Prototype (prototyp) 644
Visitor (gość) 646



S Skorowidz

649

Wprowadzenie



W tym rozdziale odpowiemy na palące pytanie: „DLACZEGO to znalazło się w książce o wzorcach projektowych?”.

Dla kogo przeznaczona jest ta książka?

Jeżeli możesz odpowiedzieć twierdząco na wszystkie poniższe pytania:

1. **Czy znasz język Java? (Nie musisz być guru...)**

Jeżeli nie znasz języka Java, ale za to świetnie posługujesz się C#, to również powinieneś spokojnie dać sobie radę.

2. Czy chcesz **poznać, zrozumieć, zapamiętać oraz stosować w praktyce** wzorce projektowe, włączając w to zasady projektowania zorientowanego obiektowo, na których opiera się cała koncepcja wzorców projektowych?

3. **Czy preferujesz stymulującą pogawędkę przy obiedzie zamiast suchego, nudnego, akademickiego wykładu?**

Jeśli tak, to jest to książka dla Ciebie.

Kto prawdopodobnie nie powinien sięgać po tę książkę?

Jeśli możesz odpowiedzieć twierdząco na przynajmniej jedno z poniższych pytań:

1. **Czy programowanie w języku Java to dla Ciebie zupełnie nowy temat?**

   (Pamiętaj, że zaawansowana znajomość języka Java nie jest wymagana; co więcej, nawet jeżeli zupełnie nie znasz języka Java, ale za to znasz C#, powinieneś bez większych problemów zrozumieć przynajmniej 80% przykładowych kodów programów. W praktyce *może* się okazać, że znajomość języka C++ również będzie wystarczająca).

2. Czy jesteś zaprawionym w bojach projektantem bądź deweloperem aplikacji zorientowanych obiektowo, szukającym **podręcznika Javy**?

3. Czy jesteś architektem systemów, poszukującym rozwiązań i wzorców do zastosowania na skalę **korporacyjną**?

4. Czy **boisz się spróbować czegoś nowego**? Czy wolałbyś raczej leczenie kanałowe niż szal w szkocką kratkę? Czy uważasz, że dana książka o programowaniu nie może być uznana za poważną, skoro poszczególne obiekty i komponenty Java zostały nieco „uczłowieczone"?

Jeśli tak, ta książka nie jest dla Ciebie.



[dopisek z działu marketingu: niniejsza książka jest przeznaczona dla każdego Czytelnika, który dysponuje gotówką bądź kartą kredytową].

Wiemy, co sobie myślisz

„Jakim cudem może to być poważna książka o programowaniu w Javie?”

„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, wygląda i *oczekuje* czegoś niezwykłego. W taki sposób został stworzony i to pomaga mu przetrwać.

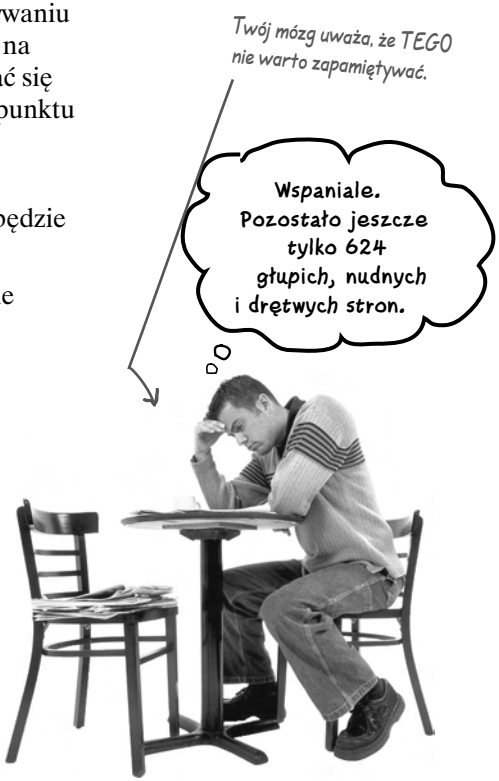
W dzisiejszych czasach jest raczej mało prawdopodobne, abyś stał się przekąską dla tygrysa. Ale Twój mózg wciąż obserwuje. W końcu nigdy nic nie wiadomo.

Zatem, co Twój mózg robi z tymi wszystkimi rutynowymi, zwyczajnymi, normalnymi informacjami, jakie do niego codziennie docierają? Otóż robi wszystko, co tylko może, aby nie przeszkadzały w jego *najważniejszym* zadaniu — w zapamiętywaniu rzeczy, które mają *prawdziwe znaczenie*. Twój mózg nie traci czasu i energii na zapamiętywanie nudnych informacji; one nigdy nie będą w stanie przedostać się przez ten swego rodzaju filtr odrzucający wszystkie zdarzenia, które z jego punktu widzenia są „w oczywisty sposób całkowicie nieważne”.

W jaki sposób Twój mózg *wie*, co jest istotne? Jeżeli założymy, że podczas codziennej przechadzki nagle stanie przed Tobą wygłodniały tygrys, co się będzie działo wewnątrz Twojej głowy?

Neurony zapłoną. Emocje się rozszaleją. Potężna dawka adrenaliny zostanie wpompowana do Twojego krwiobiegu.

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. Jesteś w bezpiecznym miejscu — ciepłym, przytulnym i zupełnie pozbawionym tygrysów. Uczysz się. Przygotowujesz się do egzaminu. Albo rozgryzasz jakiś trudny problem techniczny, którego rozwiązanie, według Twojego szefa, nie powinno zająć Ci więcej niż tydzień, a najwyżej dziesięć dni.

Jest tylko jeden, drobny problem. Twój mózg stara Ci się oddać wielką przysługę, próbując działać tak, aby te w *oczywisty* sposób nieistotne informacje nie zajmowały cennych zasobów w Twojej głowie. Zasobów, które powinny zostać wykorzystane do zapamiętania 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.

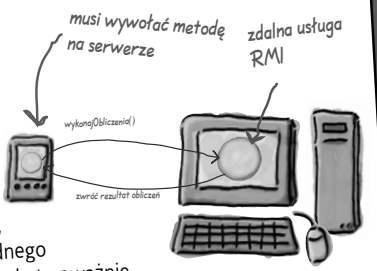
Co gorsza, nie ma żadnego sposobu, aby powiedzieć mózgowi: „Hej, mój mózgu, dziękuję ci bardzo, ale niezależnie od tego, jak nudna jest ta książka i jak nisko na emocjonalnej skali Richtera zostaną zarejestrowane przeżycia, których aktualnie doznaję, *naprawdę* chciałbym zapamiętać wszystkie te informacje”.

Wyobrażamy sobie, że Czytelnik tej książki jest uczniem

A więc, co trzeba zrobić, aby się czegoś nauczyć? Otóż, po pierwsze, powinieneś to coś poznać, a następnie upewnić się, że o tym nie zapomnisz. Pamiętaj jednak, że uczenie się nie polega jedynie na wtłaczaniu do biednej głowy suchych faktów. Najnowsze badania z dziedziny nauki o przyswajaniu informacji, neurobiologii oraz psychologii nauczania niezbicie dowodzą, że proces uczenia się wymaga o wiele więcej niż tylko czytania tekstu wydrukowanego w takim czy innym podręczniku. My wiemy, co potrafi pobudzić Twój mózg do działania.

Oto niektóre z głównych założeń niniejszej książki

Spróbuj przedstawić problem w postaci graficznej. Rysunki są znacznie łatwiejsze do zapamiętania niż same słowa i sprawiają, że proces uczenia się przebiega zdecydowanie bardziej efektywnie (przeprowadzone badania niezbicie dowodzą, że zastosowanie rysunków w procesie nauczania poprawia efektywność zapamiętywania informacji nawet o 89%). Dodatkowo zastosowanie rysunków sprawia, że informacje stają się znacznie bardziej zrozumiałe. **Wystarczy umieścić dany opis bezpośrednio na lub też w okolicach rysunku**, do którego się on odnosi (zamiast na następnej stronie), a prawdopodobieństwo, że osoby uczące się będą w stanie rozwiązać omawiany problem, wzrośnie niemal *dwukrotnie*.



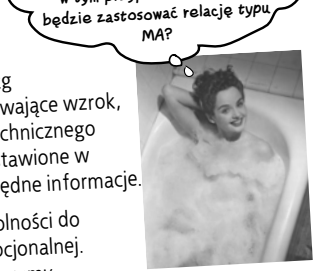
Stosuj konwersacje i personifikacje. Według najnowszych badań w testach końcowych studenci uzyskiwali wyniki o 40% lepsze, jeśli treść była przekazywana w sposób bezpośredni, w pierwszej osobie, i w konwencji rozmowy, a nie bardzo formalnie. Zamiast prowadzenia nudnego wykładu, staraj się opowiadać historyjki. Używaj zwyczajnego języka. Nie traktuj swojej osoby zbyt poważnie. Zastanów się: kiedy Ty zwracałbyś większą uwagę na to, co się dookoła Ciebie dzieje — podczas stymulującej rozmowy przy obiedzie czy podczas nużącego, przydługiego wykładu?





Metoda nie posiada ciała. Jej definicja kończy się średnikiem.

Zmuś Czytelnika (słuchacza) do głębszego zastanowienia. Innymi słowy, jeśli nie zmusisz neuronów do aktywnego wysiłku, w Twojej głowie nie zdarzy się nic wielkiego. Czytelnik musi mieć dostateczną motywację, być szczerze zaangażowany, zaciekawiony i podekscytowany rozwiązywaniem problemów, wyciąganiem wniosków i zdobywaniem nowej wiedzy. Osiągnięcie takiego wspaniałego rezultatu jest możliwe poprzez stawianie kolejnych wyzwań, zapraszanie do rozwiązywania ćwiczeń, zadawanie pytań zmuszających do zastanowienia oraz nakłanianie do działań, które wymagają zaangażowania zarówno obu półkul mózgowych, jak i wszystkich zmysłów.



Zdobądź — przyciągnij na dłużej — uwagę i zainteresowanie Czytelnika. Zapewne niemal każdy znalazł się kiedyś w sytuacji, gdy mimo najszczerszych chęci przyswojenia sobie jakiejś partii materiału zasypiał po przeczytaniu pierwszej strony podręcznika. Twój mózg zwraca uwagę na rzeczy niezwykłe, interesujące, dziwne, przykuwające wzrok, nieoczekiwane. Pamiętaj, że poznawanie nowego zagadnienia technicznego wcale nie musi być nudne. Jeśli dane zagadnienie zostanie przedstawione w atrakcyjny sposób, Twój mózg znacznie szybciej przyswoi sobie niezbędne informacje.

Zagraj na emocjach Czytelnika (słuchacza). Teraz już wiemy, że zdolności do zapamiętywania informacji są w znacznej mierze zależne od ich zawartości emocjonalnej. Łatwiej zapamiętujemy to, na czym nam *zależy*. Zapamiętujemy w sytuacjach, w których coś *odczuwamy*. Oczywiście, nie mamy tu na myśli wzruszających historii o małym chłopcu i jego psie. Chodzi nam o emocje takie, jak zaskoczenie, ciekawość, radosne podekscytowanie, wywołujące nam na usta słowa : „A niech to..." i uczucie satysfakcji —„Jestem wielki!" — jakie odczuwamy po poprawnym rozwiązaniu zagadki, zrozumieniu zasad funkcjonowania czegoś, co powszechnie uchodzi za bardzo trudne lub po prostu po zdaniu sobie sprawy z tego, że teraz znamy więcej szczegółów technicznych niż Robert z działu technicznego.



Metapoznawanie — myślenie o myśleniu

Jeśli naprawdę chcesz się czegoś nauczyć i jeśli chcesz się tego nauczyć szybciej i dokładniej, zwracaj uwagę na fakt, jak bardzo Ci na tym zależy. Myśl o tym, jak myślisz. Poznawaj sposoby własnego uczenia się.



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

Możemy chyba jednak bezpiecznie założyć, że jeżeli trzymasz w ręku tę książkę, chcesz poznać wzorce projektowe, a co więcej, prawdopodobnie nie chcesz na to tracić zbyt wiele czasu. Zapewne chciałbyś również *zapamiętać* to, co tutaj przeczytasz, oraz być w stanie wykorzystać zdobytą wiedzę w praktyce. To oznacza jedno: musisz zdobywane informacje po prostu zrozumieć. Aby w jak największym stopniu wykorzystać niniejszą książkę, *dowolną* inną książkę lub jakikolwiek inny sposób uczenia się, należy wziąć odpowiedzialność za swój mózg. 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 wspomniany wcześniej tygrys stojący naprzeciw Ciebie. W przeciwnym razie będziesz zmuszony prowadzić nieustającą wojnę ze swoim własnym mózgiem, który ze wszystkich sił i poniekąd w dobrej wierze będzie się starał, aby te nowe informacje nie zostały utrwalone.

A zatem, w jaki sposób można ZMUSIĆ mózg, aby wzorce projektowe były dla niego tak samo ważne, jak głodny tygrys?

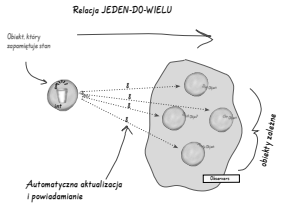
Można to zrobić w sposób powolny i męczący lub szybki i bardziej efektywny. Powolny sposób polega na wielokrotnym powtarzaniu. Oczywiście, wiesz, że *jesteś w stanie* nauczyć się i zapamiętać nawet najnudniejsze zagadnienia, jeżeli tylko będziesz odpowiednio długo je „wkuwał". Po odpowiedniej ilości 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 wyzwoli kilka różnych *typów* aktywności. Wszystkie zagadnienia, o jakich pisaliśmy na poprzedniej stronie, są kluczowymi elementami rozwiązania. Udowodniono, że wszystkie potrafią pomóc w zmuszeniu mózgu do wydajniejszej pracy na Twoją korzyść. Przykładowo, niezależne badania wykazują, że umieszczenie słów *na* opisywanych rysunkach (a nie w innych miejscach na stronie, na przykład w nagłówku lub wewnątrz akapitu) sprawia, że mózg stara się zrozumieć relację pomiędzy słowami a rysunkiem, a to z kolei zwiększa aktywność neuronów. Większa aktywność neuronów = większa szansa, że mózg *uzna* informacje za warte zainteresowania i — ewentualnie — zapamiętania.

Prezentowanie informacji w formie konwersacji również pomaga; dzieje się tak dlatego, iż ludzie wykazują zdecydowanie większe zainteresowanie w sytuacjach, kiedy wydaje im się, że biorą udział w rozmowie, i kiedy oczekuje się od nich, że będą śledzić jej przebieg i brać w niej czynny udział. Zdumiewający jest fakt, iż mózg zdaje się *nie zważać* na to, że tak naprawdę 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 biernie uczestniczysz w wykładzie na sali pełnej sennych studentów. Nie ma potrzeby wykazywania jakiejkolwiek aktywności.

Warto jednak tutaj zauważyć, że rysunki i przedstawianie informacji w formie rozmowy to dopiero początek.

Oto, co zrobiliśmy



Używaliśmy ***rysunków***, ponieważ Twój mózg zwraca większą uwagę na obrazki niż na tekst. Z punktu widzenia Twojego mózgu można śmiało założyć, że jeden obrazek jest wart tyle, co 1024 słowa. W sytuacjach, kiedy tekst i rysunek były ze sobą powiązane, staraliśmy się umieszczać tekst bezpośrednio *na* rysunku, gdyż mózg działa bardziej efektywnie, gdy tekst jest zlokalizowany bezpośrednio *wewnątrz* elementu, który opisuje, niż kiedy jest umieszczony w innym miejscu i stanowi część większego fragmentu tekstu.

Stosowaliśmy ***powtórzenia***, wielokrotnie podając tę samą informację *na różne sposoby*, wykorzystując różne środki przekazu oraz odwołując się do *różnych zmysłów*. Wszystko po to, aby zwiększyć szansę, że informacja zostanie poprawnie i trwale zakodowana w większej ilości obszarów Twojego mózgu.



Wzorcowy Guru

Używaliśmy pomysłów i rysunków w ***nieoczekiwany*** sposób, ponieważ Twój mózg oczekuje i pragnie nowości; poza tym staraliśmy się zawrzeć w nich *chociaż trochę* ***emocji***, gdyż Twój mózg jest skonstruowany w taki sposób, że zwraca baczniejszą uwagę na biochemię związaną z emocjami. Prawdopodobieństwo zapamiętania czegoś, co budzi w nas jakieś emocje, co powoduje, że *czujesz*, jest dużo większe, nawet jeżeli to uczucie nie jest niczym więcej jak tylko lekkim ***rozbawieniem***, ***zaskoczeniem*** lub ***cieniem zainteresowania***.

Używaliśmy bezpośrednich zwrotów i przekazywaliśmy treści ***w formie konwersacji***, gdyż Twój mózg zwraca na nie większą uwagę, jeśli sądzi, że prowadzisz rozmowę, niż gdy jesteś jedynie biernym słuchaczem prezentacji. Mózg działa w ten sposób nawet wtedy, gdy *czytasz* zapis rozmowy.

W książce zamieściliśmy ponad 40 ***ćwiczeń***, ponieważ mózg uczy się i pamięta więcej, gdy coś ***robi***, niż gdy o czymś tylko *czyta*. W dodatku zamieszczone w niniejszej książce ćwiczenia stanowią swego rodzaju wyzwanie, ale nie są przesadnie trudne, gdyż właśnie takie preferuje większość *ludzi*.



CELNE SPOSTRZEŻENIA

Zastosowaliśmy ***wiele stylów nauczania***, gdyż — przykładowo — *TY* możesz preferować instrukcje opisujące krok po kroku właściwy sposób postępowania, ktoś inny woli ogólną analizę omawianego zagadnienia, a jeszcze inny preferuje przeglądanie przykładowego fragmentu kodu programu. Jednak niezależnie od ulubionego sposobu nauki z pewnością *każdy* skorzysta na tym, że te same informacje będą przedstawiane kilkakrotnie na różne sposoby.

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

Łamigłówka



Zamieściliśmy tu też ***historyjki*** i ćwiczenia prezentujące ***więcej niż jeden punkt widzenia***, ponieważ Twój mózg uczy się łatwiej, gdy jest zmuszony do przetwarzania, analizowania i oceniania podawanych opinii.

Stawialiśmy przed Tobą ***wyzwania***, zarówno poprzez zamieszczanie ćwiczeń, jak i stawianie ***pytań***, na które nie zawsze można odpowiedzieć w prosty sposób. To wszystko dlatego, że mózg uczy się i pamięta, gdy musi nad czymś *popracować* (analogicznie, nie można zdobyć dobrej *kondycji*, wyłącznie *obserwując* innych ludzi ćwiczących na siłowni). Dołożyliśmy jednak wszelkich starań, aby upewnić się, że gdy pracujesz, robisz dokładnie to, co trzeba, dokładnie tak, aby zrozumienie trudnego przykładu nie wymagało wykorzystania ani jednego dendrytu za dużo, oraz aby Twoje przepracowane dendryty nie musiały analizować tekstu zbyt lapidarnego lub napisanego niezrozumiałym żargonem.



Gdzie tylko to było możliwe, staraliśmy się dokonać ***antropomorfizacji*** omawianych zagadnień. W opowiadaniach, przykładach, rysunkach i wszelkich innych możliwych miejscach tekstu staraliśmy się personifikować tekst, gdyż *TY* jesteś człowiekiem, a Twój mózg zwraca większą uwagę na *ludzi* niż na *rzeczy*.

Zastosowaliśmy metodę ***80/20***. Zakładamy bowiem, że nie jest to książka dla osób, które mają zamiar pisać doktorat na inżynierii oprogramowania; stąd nie będziemy się w niej zajmować *wszelkimi* możliwymi zagadnieniami, a jedynie tymi, które faktycznie będą dla Ciebie *przydatne i interesujące*.



Wytnij poniższe porady i przyklej na lodówce.

Oto, co możesz zrobić, aby zmusić swój mózg do posłuszeństwa

A zatem zrobiliśmy, co było w naszej mocy. Reszta zależy od Ciebie. Możesz zacząć od przedstawionych poniżej porad; posłuchaj swojego mózgu i zorientuj się, które sprawdzają się w Twoim przypadku, a które nie dają pozytywnych rezultatów. Zawsze warto wypróbowywać nowinki.

1 Zwolnij. Im więcej zrozumiesz, tym mniej będziesz musiał zapamiętać.

Nie ograniczaj się jedynie do *czytania*. Przerwij na chwilę lekturę i pomyśl. Kiedy znajdziesz w tekście pytanie, nie zaglądaj od razu na stronę z odpowiedzią. Wyobraź sobie, że ktoś faktycznie *zadaje* Ci pytanie. Im bardziej zmusisz swój mózg do myślenia, tym większa szansa, że przyswoisz i zapamiętasz dany materiał.

2 Wykonuj ćwiczenia. Rób swoje własne notatki.

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

3 Czytaj sekcje oznaczone jako „Nie istnieją głupie pytania".

Oznacza to po prostu, że powinieneś zapoznać się z treścią *wszystkich* takich sekcji. Nie są to elementy opcjonalne czy dodatkowe — wręcz przeciwnie, ***stanowią one integralną część podstawowej zawartości książki!*** Nie pomijaj ich.

4 Rób sobie przerwy.

Wstań, przeciągnij się, pochodź trochę, zmień fryzurę, przejdź do innego pokoju. Dzięki temu Twój mózg będzie mógł coś poczuć, a nauka nie będzie kojarzyć mu się z konkretnym miejscem.

5 Niech lektura tej książki będzie ostatnią rzeczą, jaką robisz przed pójściem spać. A przynajmniej ostatnią stanowiącą wyzwanie intelektualne rzeczą danego dnia.

Pewne elementy procesu uczenia się (w szczególności przenoszenie informacji do pamięci długoterminowej) mają miejsce już *po* odłożeniu książki na półkę. Twój mózg potrzebuje trochę czasu dla siebie, przeznacza go na dodatkowe przetwarzanie dostarczonych wcześniej informacji. Jeśli podczas tego czasu zmusisz go do innej działalności, część z przyswojonych informacji może zostać utracona.

6 Pij wodę. Dużo wody.

Twój mózg pracuje najlepiej, gdy dostarcza mu się dużo płynów. Odwodnienie (które może następować nawet, zanim poczujesz pragnienie) zdecydowanie obniża Twoją zdolność percepcji.

7 Rozmawiaj o zdobywanych informacjach. Na głos.

Mówienie aktywuje odmienne fragmenty mózgu. Jeśli próbujesz coś zrozumieć lub zwiększyć szansę na zapamiętanie informacji na dłużej, powtarzaj je na głos. Jeszcze lepiej — staraj się je na głos komuś wytłumaczyć. W ten sposób nauczysz się szybciej, a oprócz tego będziesz mógł odkryć kwestie, z których nie zdawałeś sobie sprawy podczas czytania książki.

8 Posłuchaj swojego mózgu.

Staraj się nie dopuścić do sytuacji, w której Twój mózg staje się przeciążony. Jeśli zauważysz, że zaczynasz czytać pobieżnie lub niemal natychmiast zapominasz to, o czym przeczytałeś przed chwilą, to najwyższy czas, żeby sobie zrobić przerwę. Po przekroczeniu pewnego punktu wcale nie będziesz się uczył szybciej, nieustannie wpychając na siłę do głowy nowe informacje; co gorsza, taki sposób postępowania może mieć zdecydowanie negatywny wpływ na cały proces przyswajania wiedzy.

9 Poczuj coś!

Twój mózg musi wiedzieć, że to, czego się uczysz, jest *ważne*. Z zaangażowaniem staraj się śledzić zamieszczane w tekście historyjki i opowiadania. Nadawaj własne tytuły zdjęciom. Płakanie ze śmiechu po przeczytaniu głupiego dowcipu jest *ciągle* lepsze od braku jakiejkolwiek reakcji.

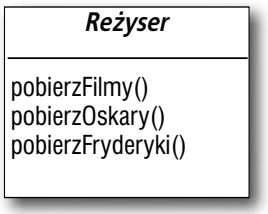
10 Zaprojektuj coś!

Zastosuj poznane zasady w praktyce podczas tworzenia nowego projektu bądź modernizacji jakiegoś projektu, który już „odleżał" swoje w archiwum. Po prostu zrób *coś*, co pozwoli Ci zdobyć doświadczenia nieco inne niż te, których nabrałeś podczas wykonywania ćwiczeń i zadań opisywanych w naszej książce. Wszystko, czego tak naprawdę potrzebujesz, to ołówek i problem do rozwiązania... problem, którego rozwiązanie może korzystać z jednego lub więcej poznanych wzorców projektowych.

Kilka ostatnich spraw, o których musisz wiedzieć

To książka dla osób dopiero rozpoczynających przygodę z wzorcami projektowymi, a nie podręcznik akademicki. Celowo usunęliśmy wszystko, co mogłoby Ci przeszkadzać w nauce, niezależnie od tego, nad czym pracujesz w danym miejscu książki. Podczas pierwszej lektury należy zaczynać od jej samego początku, gdyż kolejne rozdziały opierają się na tym, co widziałeś i czego dowiedziałeś się wcześniej.

Wykorzystaliśmy zmodyfikowany, uproszczony sposób zapisu przypominający UML



Używamy prostych diagramów wzorowanych na UML

Chociaż z dużym prawdopodobieństwem można założyć, że już gdzieś wcześniej miałeś okazję spotkać się z językiem UML, informujemy, że nie będzie on omawiany w niniejszej książce, a co ważniejsze, do zrozumienia opisywanych tutaj zagadnień jego znajomość nie jest wymagana. Jeżeli nigdy wcześniej nie zetknąłeś się z językiem UML, nie masz powodów do zmartwień — w kolejnych rozdziałach niniejszej książki będziemy udzielać niezbędnych wskazówek i wyjaśnień. Krótko mówiąc, nie musisz przejmować się zagadnieniami związanymi z wzorcami projektowymi oraz językiem UML jednocześnie. Nasze diagramy są bardzo blisko spokrewnione z językiem UML — na pierwszy rzut oka są do niego bardzo podobne czy nawet dokładnie odpowiadają jego składni, aczkolwiek w niektórych przypadkach zaczynają od niego odbiegać; zazwyczaj jedynym wytłumaczeniem takiego odstępstwa była nieco samolubna chęć pofolgowania naszym artystycznym wizjom i wyobrażeniom.

Nie staraliśmy się omawiać KAŻDEGO wzorca projektowego, jaki kiedykolwiek został opracowany

Istnieje *bardzo dużo* różnego rodzaju wzorców projektowych. Wystarczy wymienić fundamentalne wzorce, stanowiące podwaliny programowania zorientowanego obiektowo, wzorce J2EE firmy Sun, wzorce JSP, wzorce strukturalne, wzorce do projektowania i tworzenia gier i wiele, wiele innych. Z drugiej strony, naszym założeniem było stworzenie książki, która będzie ważyła mniej niż sam Czytelnik będący jej użytkownikiem, stąd omówienie wszystkich (a przynajmniej zdecydowanej większości) istniejących wzorców projektowych wykraczałoby daleko poza ramy tak pomyślanej publikacji. Zamiast tego skoncentrowaliśmy się na omówieniu wybranych spośród wzorców GoF wzorców projektowych, które będą dla Ciebie *ważne* i przydatne. Staraliśmy się przygotować całość tak, żebyś był w stanie dokładnie, rzetelnie i w pełni zrozumieć, dlaczego, w jaki sposób i kiedy należy z nich korzystać. Dodatkowo na końcu książki (nomen omen w dodatku...) znajdziesz nieco skrócony opis wybranych innych wzorców (takich, które według wszelkiego prawdopodobieństwa będą wykorzystywane o wiele rzadziej). W każdym razie z całą pewnością po uważnej lekturze niniejszej książki będziesz w stanie wziąć do ręki dowolny katalog wzorców projektowych i szybko ruszyć z robotą do przodu.

Ćwiczenia NIE SĄ elementem opcjonalnym

Ćwiczenia i inne elementy interakcyjne zamieszczone w naszej książce nie są tylko i wyłącznie miłymi dodatkami; są one integralną częścią jej podstawowej zawartości. Niektóre z nich mają na celu wspomaganie procesu zapamiętywania, inne pomagają w zrozumieniu tekstu, a jeszcze inne ułatwiają praktyczne zastosowanie wiedzy nabytej podczas pracy z książką. Nie pomijaj takich ćwiczeń.

Czasownika „składa się" używamy tu w ogólnym znaczeniu przyjętym dla programowania zorientowanego obiektowo — jest ono bardziej elastyczne niż to wynikające z reguł zapisu UML

Kiedy mówimy, że „dany obiekt składa się z innego obiektu", to przyjmujemy, że takie dwa obiekty łączy relacja typu MA. Takie podejście do zagadnienia odzwierciedla nasz tradycyjny sposób używania czasownika „składa się" i jest tożsamy z jego zastosowaniem we wzorcach GoF (o czym będziemy pisali w dalszej części niniejszej książki). Ostatnimi czasy standard UML zaprezentował nieco zmodyfikowane podejście do tego czasownika, wyraźnie wyodrębniając kilka różnych rodzajów jego zastosowań. Jeżeli jesteś ekspertem w dziedzinie UML, i tak nadal będziesz w stanie bez żadnych problemów pracować z naszą książką i samodzielnie przekładać „nasze" zastosowania tego czasownika na bardziej szczegółowe i zgodne z UML-owym sposobem jego pojmowania.

Redundancja zagadnień jest zamierzona i bardzo istotna

Jedną z najważniejszych różnic pomiędzy innymi książkami a *Rusz głową!* jest to, że my chcemy, abyś *naprawdę* zrozumiał omawiane zagadnienia. Co więcej, bardzo chcemy, abyś po zakończeniu pracy z książką pamiętał, czego się z niej nauczyłeś. Z punktu widzenia autorów większości podręczników wspomaganie procesu zapamiętywania i przypominania nie jest najważniejszym celem, natomiast nasza książka sama w sobie uwzględnia przebieg procesu *uczenia się* i z tego powodu podczas jej lektury z wieloma zagadnieniami będziesz spotykał się więcej niż tylko raz.

Przykładowe kody programów są tak zwięzłe, jak to tylko było możliwe

Nasi Czytelnicy niejednokrotnie sygnalizowali, że przeglądanie 200 wierszy kodu w poszukiwaniu tych istotnych 2 linijek, które należy dokładnie zrozumieć, może być niezwykle frustrujące. Większość przykładów zamieszczonych w tej książce została zilustrowana najmniejszą możliwą ilością wierszy kodu, tak że fragmenty, które chcesz zrozumieć, stały się proste i klarowne. Nie powinieneś jednak za to oczekiwać, że całość zamieszczonego kodu będzie odporna na błędy czy też będzie stanowić kompletną całość — wręcz przeciwnie, wszystkie przykłady zostały napisane specjalnie pod kątem zrozumienia zasady działania danego elementu i nie zawsze są w pełni funkcjonalne z technicznego punktu widzenia.

W niektórych przypadkach nie dołączaliśmy nawet do przykładowego kodu wszystkich niezbędnych poleceń i wyrażeń; robiliśmy to jednak z założeniem, że pisałeś już programy w języku Java i wiesz, że np.definicja klasy `ArrayList` znajduje się w pakiecie `java.util`. Jeżeli importowane definicje nie są częścią normalnego pakietu J2SE API, staraliśmy się to wyraźnie zaznaczyć. Przygotowaliśmy również szereg kodów źródłowych, z którymi będziesz się spotykał w trakcie lektury książki. Spakowany plik zawierający kody przykładów możesz pobrać z naszego serwera FTP*: *ftp://ftp.helion.pl/przyklady/wzorrg.zip.*

Dodatkowo, cały czas mając na względzie edukacyjne zadanie analizowanych przykładów kodu, nie umieszczaliśmy naszych definicji klas w poszczególnych pakietach (innymi słowy, z założenia wszystkie nasze klasy znajdują się w domyślnym pakiecie definicji klas Javy). Zdecydowanie nie polecamy takiego rozwiązania podczas tworzenia rzeczywistych projektów, stąd, jeżeli zdecydujesz się na pobranie pakietów kodów przykładowych z internetu, szybko zorientujesz się, że *zostały one* pogrupowane w odpowiednie pakiety.

Do ćwiczeń oznaczonych jako „Wysil szare komórki” nie przygotowaliśmy odpowiedzi

W przypadku niektórych z nich odpowiedzi nie zostały w ogóle zamieszczone w niniejszej książce, w przypadku innych doświadczenia wyniesione z rozwiązywania zadań zamieszczonych w innych częściach książki pozwolą Ci określić, czy podane odpowiedzi są dobre, czy nie. W jeszcze innych ćwiczeniach zamieściliśmy szereg podpowiedzi, które ułatwią Ci poruszanie się we właściwym kierunku.

---

* Uwaga! Kody źródłowe zamieszczone w niniejszej książce mają charakter poglądowy i zostały odpowiednio do tego celu przygotowane (m.in. nazwy klas, metod, zmiennych itp. zostały przetłumaczone na język polski). Czytelników zainteresowanych kompletnymi, w pełni funkcjonalnymi wersjami poszczególnych programów prosimy o pobranie odpowiednich plików z serwera FTP naszego wydawnictwa: *ftp://ftp.helion.pl/przyklady/wzorrg.zip — przyp. tłum.*

Zespół recenzentów technicznych

Valentin Crettaz



Jef Cumps



Barney Marispini





Nieustraszony lider ekstremalnego zespołu recenzentów niniejszej książki

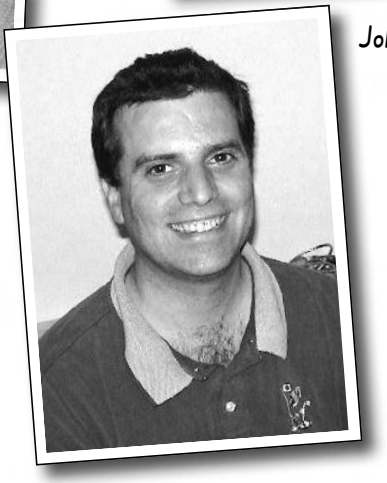
Ike Van Atta



Johannes deJong

Jason Menard





Mark Spritzler



Dirk Schreckmann



Philippe Maquet

Pamięci Philippe'a Maqueta

1960-2004

Twoje niesamowite doświadczenie techniczne, nieustający entuzjazm oraz głęboka troska o Czytelnika zawsze będą nas inspirowały.

Nigdy Cię nie zapomnimy.

Podziękowania

Wydawnictwo O'Reilly:

Nasze serdeczne podziękowania kierujemy do **Mike'a Loukidesa** z wydawnictwa O'Reilly, za rozpoczęcie całego dzieła i ogromny wkład w nadanie pomysłowi serii *Rusz głową!* ostatecznego kształtu. Wielkie podziękowania należą się również osobie, która, stojąc za całą serią *Rusz głową!*, stanowiła jej siłę napędową — **Timowi O'Reilly**. Dziękujemy pomysłowej „mamusi" serii *Rusz głową!*, **Kyle Hart**, a także gwieździe rock and rolla, **Ellie Volkhausen**, za jej inspirujący projekt okładek oraz **Colleen Gorman**, za jej ciężką pracę przy edycji i korekcie niniejszej książki. Wreszcie składamy podziękowania **Mike'owi Hendricksonowi**, za podjęcie wyzwania, jakim była praca nad książką i zbudowanie wspaniałego zespołu redakcyjnego.

Nasz nieustraszony zespół korekty:

Jesteśmy niezwykle wdzięczni naszemu szefowi zespołu korekty, **Johannesowi deJongowi**. Johannes, jesteś naszym bohaterem. Chcielibyśmy również powiedzieć, że bardzo doceniamy wkład pracy menedżera grupy korektorów z zespołu **Javaranch**, **Phillippe'a Maqueta**. To właśnie Ty rozjaśniłeś życie tysięcy projektantów i deweloperów, a wpływ, jaki wywarłeś na ich (i nasze) życie, będzie trwał wiecznie.

Składamy podziękowania **Jefowi Cumpsowi**, który był przerażająco skuteczny w wyszukiwaniu problemów oraz słabych stron rękopisów kolejnych rozdziałów i w efekcie miał ogromny wpływ na końcową postać naszej książki. Dzięki, Jef! **Valentin Crettaz** (członek zespołu AOP), który był z nami od momentu ukazania się pierwszej książki w serii *Rusz głową!*, udowodnił nam niezbicie (jak zwykle zresztą), jak bardzo potrzebujemy jego wiedzy technicznej oraz doświadczenia. Jesteś naprawdę dobry, Valentin (ale wyrzuć ten krawat!).

Dwóch nowicjuszy w zespole korektorów, Barney Marispini oraz Ike Van Atta, odwaliło kawał naprawdę dobrej roboty podczas sprawdzania naszej książki — chłopcy, dzięki Waszej pracy uzyskaliśmy *ogromną ilość* cennych wskazówek i sugestii. Dzięki za dołączenie do zespołu.

Autorzy otrzymali również wspaniałe wsparcie techniczne ze strony moderatorów i ekspertów zespołu Javaranch: **Marka Spritzlera**, **Jasona Menarda**, **Dirka Schreckmanna**, **Thomasa Paula** oraz **Margarity Isaeva**. Jak zawsze, chcielibyśmy złożyć specjalne podziękowania przewodnikowi stada autorów witryny *javaranch.com*, **Paulowi Wheatonowi**.

Dziękujemy finalistom konkursu „Wybierz okładkę książki *Wzorce projektowe. Rusz głową!*", przeprowadzonego przez zespół Javaranch. Jego zwycięzca, Si Brewster, przedstawił opracowanie, którego efektem jest zdjęcie dziewczyny widoczne na okładce. Pozostałymi finalistami konkursu byli m.in. Andrew Esse, Gian Franco Casula, Helen Crosbie, Pho Tek, Helen Thomas, Sateesh Kommineni oraz Jeff Fisher.

Jeszcze więcej podziękowań*

Od Erica oraz Elisabeth

Pisanie książek z serii *Rusz głową!* przypomina nieco szaleńczą jazdę z dwójką wspaniałych przewodników: **Kathy Sierra** oraz **Bertem Batesem**. Pracując z Kathy i Bertem porzucasz wszelkie klasyczne konwencje i zasady pisania książek i wkraczasz do świata pełnego opowieści, teorii omawiających procesy nauczania, poznawania i percepcji oraz świata popkultury, gdzie właśnie Czytelnik zawsze ustala reguły gry. Serdecznie dziękujemy Wam obojgu za wprowadzenie nas do Waszego niesamowitego świata; mamy nadzieję, że seria *Rusz głową!* może być tego wyrazem. Z całą powagą musimy przyznać, że była to fascynująca przygoda. Dziękujemy za wszystkie Wasze pełne troski wskazówki, za delikatne, aczkolwiek stanowcze mobilizowanie nas do podążania naprzód, a przede wszystkim za okazywane nam zaufanie (w końcu nie każdemu można powierzyć opiekę nad swoim dzieckiem). Obydwoje jesteście wręcz niebezpiecznie zdolni i sprytni, a dodatkowo jesteście również najbardziej zakręconą parą dwudziestodziewięciolatków, jakich znamy. Więc… co będzie dalej?

Wielkie podziękowania dla **Mike'a Loukidesa** oraz **Mike'a Hendricksona**. Mike L. był zawsze przy nas, na każdym etapie pracy nad książką. Mike, Twoje bezcenne rady wsparte niesamowitym doświadczeniem ogromnie przyczyniły się do nadania niniejszej książce ostatecznego kształtu, a Twoje nieustanne namowy i zachęty pozwalały nam w trudnych chwilach mimo wszystko przeć naprzód. Mike H., dziękujemy za Twoją wytrwałość, upór i pięć lat cierpliwego nalegania, abyśmy napisali książkę o wzorcach projektowych; w końcu zrobiliśmy to i jesteśmy szczęśliwi, że ukazała się ona właśnie w serii *Rusz głową!*.

Specjalne podziękowania składamy **Erichowi Gamma**, który podczas dokonywania korekt naszej książki często wykraczał daleko poza nominalny zakres swoich obowiązków (doszło nawet do tego, że zabrał ze sobą na wakacje pierwszą, jeszcze surową wersję naszego dzieła). Erich, Twoje zaangażowanie w pracę nad tą książką było dla nas ciągłą inspiracją, a przeprowadzona przez Ciebie drobiazgowa korekta techniczna znakomicie przyczyniła się do podniesienia jej poziomu. Oczywiście, nie może się tutaj obejść bez serdecznych podziękowań dla całego **Gangu Czterech** za wsparcie i zaangażowanie oraz za „występ specjalny" w naszym Obiektowie. Podziękowania jesteśmy również winni **Wardowi Cunninghamowi** oraz całej rzeszy ludzi, którzy przyczynili się do powstania Portland Pattern Repository — podczas pracy nad książką będącego dla nas niezastąpionym źródłem informacji.

Napisanie książki technicznej wymaga współpracy dużego zespołu ludzi, mogącego swobodnie stanowić całą populację niewielkiej wioski: **Bill Pugh** oraz **Ken Arnold** jako eksperci dali nam cały szereg znakomitych rad dotyczących wzorca *Singleton*. **Joshua Marinacci** również dostarczył nam wiele cennych uwag i wskazówek. Artykuł **Johna Brewera**, zatytułowany „Why a Duck?" („Dlaczego Kaczka?"), stał się dla nas inspiracją do utworzenia aplikacji *SymulacjaKaczki* (i bardzo się cieszymy, że John również lubi kaczki). **Daniel Steinberg** był naszym „technicznym łącznikiem" oraz funkcjonował jako pełna sieć wsparcia emocjonalnego. Chcielibyśmy podziękować również **Jamesowi Dempsey'owi** z firmy Apple, za udzielenie zgody na wykorzystanie w naszej książce jego wspaniałego utworu, zatytułowanego „Model, View, Controller" .

Wreszcie, na samym końcu, chcielibyśmy złożyć nasze bardzo osobiste podziękowania całemu zespołowi korektorów z rancza **Javaranch**, za ich wnikliwe poprawki i gorące wsparcie. W tej książce jest więcej Was, niż moglibyście się tego spodziewać.

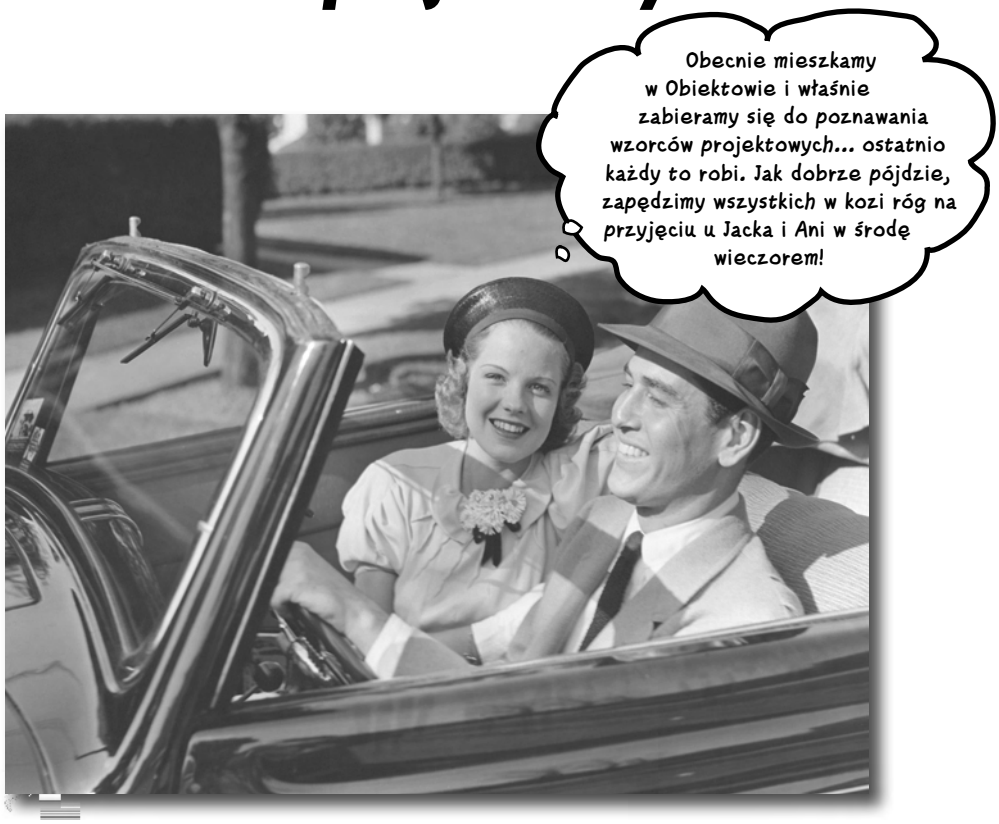
Od Kathy i Berta

Chcielibyśmy podziękować Mike'owi Hendersonowi za odnalezienie kogoś takiego, jak Eric i Elisabeth… ale nie możemy tego zrobić. Z powodu tych dwojga odkryliśmy (ku naszemu przerażeniu), że nie jesteśmy *jedynymi* ludźmi, którzy mogą napisać książkę z serii *Rusz głową!*. ;) Z drugiej jednak strony, jeżeli Czytelnicy chcą *wierzyć*, że te wszystkie fajne rzeczy w książce napisali właśnie Kathy i Bert, to kim *my* u licha ciężkiego jesteśmy, abyśmy mieli wyprowadzać ich z błędu?

---

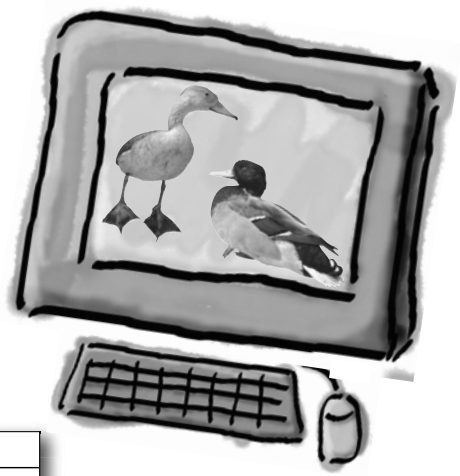
* Duża liczba zamieszczonych podziękowań wynika z faktu, iż testujemy prawdziwość teorii, według której każda osoba wymieniona w podziękowaniach kupuje przynajmniej jeden egzemplarz książki, a może i więcej — dla rodziny, znajomych itp. Jeśli zatem chciałbyś zostać wymieniony w podziękowaniach w naszej kolejnej książce i jeśli masz dużą rodzinę, to napisz do nas.

Witamy w krainie wzorców projektowych

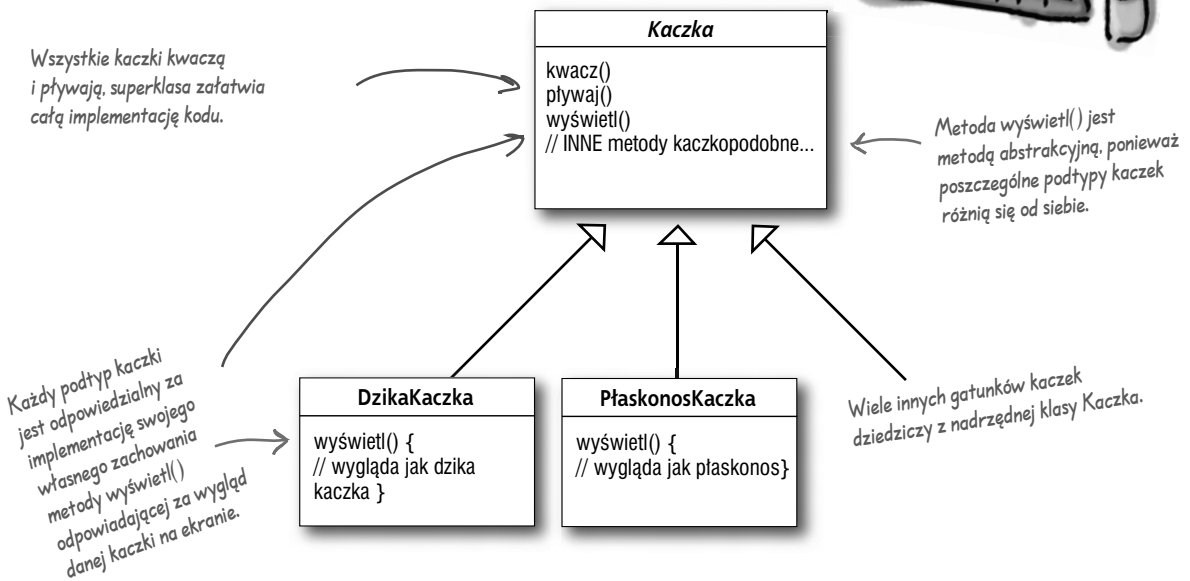


Ktoś rozwiązał już Twoje problemy. W tym rozdziale dowiesz się, dlaczego (i w jaki sposób) możesz wykorzystać wiedzę i doświadczenia zdobyte przez innych projektantów i programistów, którzy podczas pracy nad różnymi projektami zmuszeni byli wstąpić na pełną zdradliwych pułapek ścieżkę i — co najważniejsze — udało im się przeżyć tę wyprawę. Zanim dobrniemy do końca rozdziału, rzucimy okiem na sposoby wykorzystywania wzorców projektowych i przedstawimy ich zalety, poznamy kilka podstawowych zasad projektowania zorientowanego obiektowo, a także omówimy sposób działania przykładowego wzorca.
Najlepszą metodą zastosowania wzorca jest *załadowanie go bezpośrednio do Twojego mózgu*, a następnie *zlokalizowanie* obszarów w obrębie projektowanych rozwiązań oraz istniejących aplikacji, w których możesz go *zastosować*. Pracując z wzorcami projektowymi, zamiast wielokrotnego wykorzystywania tych samych fragmentów kodu wielokrotnie wykorzystujesz swoje *doświadczenia*.

Wszystko rozpoczęło się od prostej aplikacji o nazwie SymulatorKaczki



Jacek pracuje w firmie, która jest producentem odnoszącej ogromne sukcesy na rynku gry, której akcja toczy się w wirtualnym stawie z kaczkami. Gra *SymulatorKaczki* może wyświetlić wiele różnych gatunków kaczek, pływających oraz wydających odgłosy kwakania. Pierwszy zespół projektantów, który pracował nad powstaniem tej gry, wykorzystywał standardowe techniki programowania zorientowanego obiektowo i utworzył jedną klasę nadrzędną, czy też, inaczej mówiąc, superklasę o nazwie *Kaczka* (nomen omen…), z której wszystkie inne rodzaje kaczek dziedziczą właściwości.



W zeszłym roku firma Jacka była poddawana nieustannym naciskom ze strony firm konkurencyjnych. Po długiej, wyjazdowej, mile spędzonej na polu golfowym sesji burzy mózgów zarząd firmy doszedł do wniosku, że czas wdrożyć nowe, całościowe, innowacyjne rozwiązania. Potrzebowali czegoś *naprawdę* robiącego wrażenie, czegoś, co wywarłoby odpowiednie wrażenie na nadchodzącym Walnym Zgromadzeniu Akcjonariuszy, które miało się odbyć w *przyszłym tygodniu* w Maui.

Ale teraz chcemy, aby nasze kaczki umiały LATAĆ

Zarząd firmy po długiej i burzliwej dyskusji zdecydował, że kaczki, które potrafią latać, to jest właśnie to, dzięki czemu ten właśnie symulator rozniesie na strzępy symulatory konkurencji.
No i oczywiście szef Jacka powiedział im, że Jacek nie będzie miał żadnych problemów z przygotowaniem i zaprezentowaniem mniej lub bardziej działającej wersji symulatora w ciągu najbliższego tygodnia. „A poza tym" — powiedział szef Jacka — „Jacek jest programistą zorientowanym obiektowo… Do licha ciężkiego, *czy to naprawdę jest takie trudne*?".

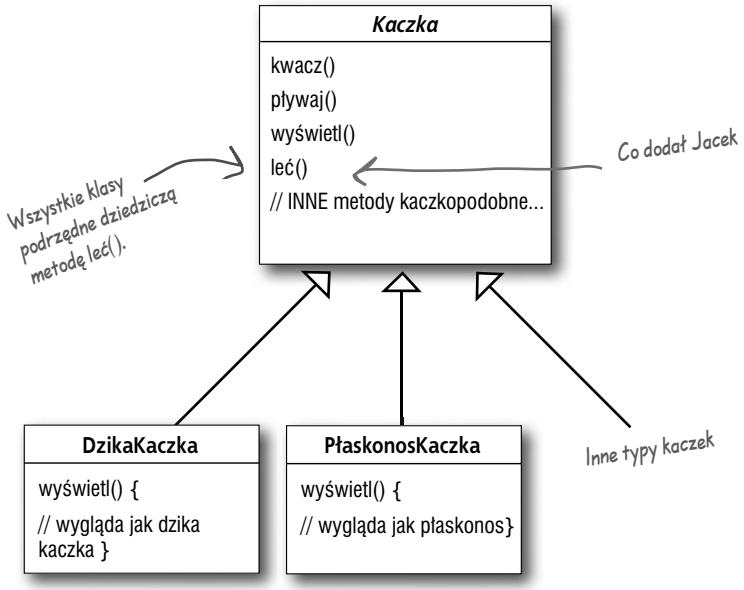


Czego chcemy



Jacek

Wszystko, czego mi trzeba, to po prostu dodać metodę leć() do klasy nadrzędnej Kaczka. Wtedy wszystkie inne kaczki będą ją dziedziczyły. Nadszedł w końcu czas, kiedy będę mógł ujawnić swój prawdziwy geniusz programisty zorientowanego obiektowo.



Ale coś poszło nie tak, jak powinno...

Słuchaj, Jacek, jestem właśnie na spotkaniu akcjonariuszy. Właśnie uruchomiliśmy wersję demonstracyjną tego nowego symulatora i na ekranie zaczęły latać gumowe kaczuszki. Czy to miał być przejaw Twojego poczucia humoru? A może po prostu ostatnio zacząłeś się rozglądać za nowym pracodawcą?





Co się stało?

Jacek nie zwrócił uwagi na — zdawałoby się, oczywisty — fakt, że nie *wszystkie* podklasy klasy nadrzędnej Kaczka powinny *umieć latać*. W momencie, kiedy Jacek dodawał nowe zachowanie do superklasy Kaczka, dodał je również do kilku podklas, dla których takie zachowanie *nie jest* właściwe. W efekcie w programie SymulatorKaczki pojawiły się niepożądane (a można by rzec, nieożywione) obiekty, które mimo to latają.

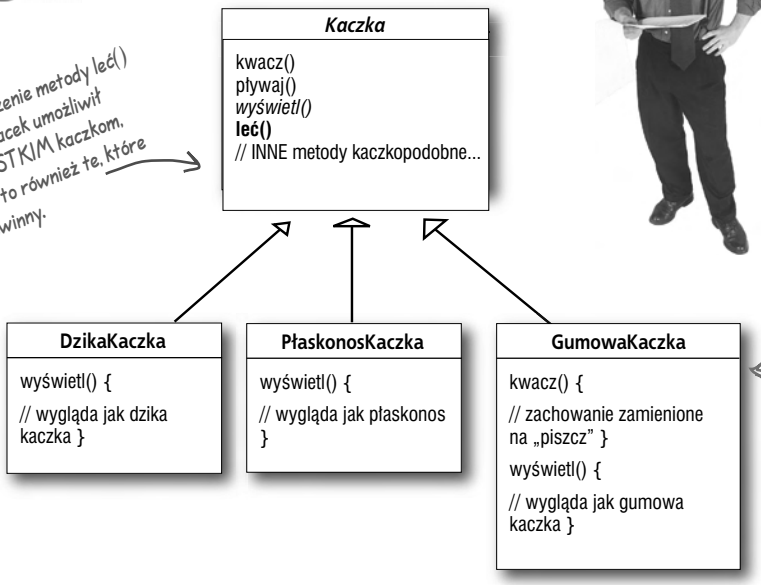
Lokalna aktualizacja kodu programu spowodowała niepożądany globalny efekt uboczny (pojawienie się latających gumowych kaczek)!

No dobrze, więc wychodzi na to, że w moim programie jest niewielka usterka. Ale nie bardzo rozumiem, dlaczego, u licha, nie nazwali tego po prostu „dodatkowymi możliwościami". Przecież wygląda to całkiem niczego sobie...



To, o czym Jacek myślał, że jest świetną okazją do zastosowania dziedziczenia i do ponownego wykorzystania istniejącego kodu, okazało się niezbyt fortunnym rozwiązaniem w sytuacji, kiedy należy dokonać poprawek i aktualizacji.

Poprzez umieszczenie metody leć() w superklasie Jacek umożliwił latanie WSZYSTKIM kaczkom, włączając w to również te, które latać nie powinny.



Gumowe kaczki nie kwaczą, stąd metoda kwacz() zmieniła zachowanie na „piszcz".

Jacek rozmyśla o dziedziczeniu...



GumowaKaczka

```
kwacz() {//piszcz}
wyświetl() {// gumowa kaczka }
leć() {
//zmień zachowanie na „nie rób nic” }
```



WabikKaczka

```
kwacz() {
// zmień zachowanie na „nie rób nic” }
wyświetl() { // kaczka wabik
}
leć() {
// zmień zachowanie na „nie rób nic” }
```

Oto definicja kolejnej klasy w naszej hierarchii; zwróć uwagę, że ta kaczka, podobnie jak GumowaKaczka, nie lata, ale dodatkowo również nie kwacze.

Zaostrz ołówek

Które z poniżej wymienionych negatywnych zjawisk będzie rezultatem zastosowania mechanizmu dziedziczenia do tworzenia poszczególnych zachowań Kaczki? (Zaznacz wszystkie pasujące).

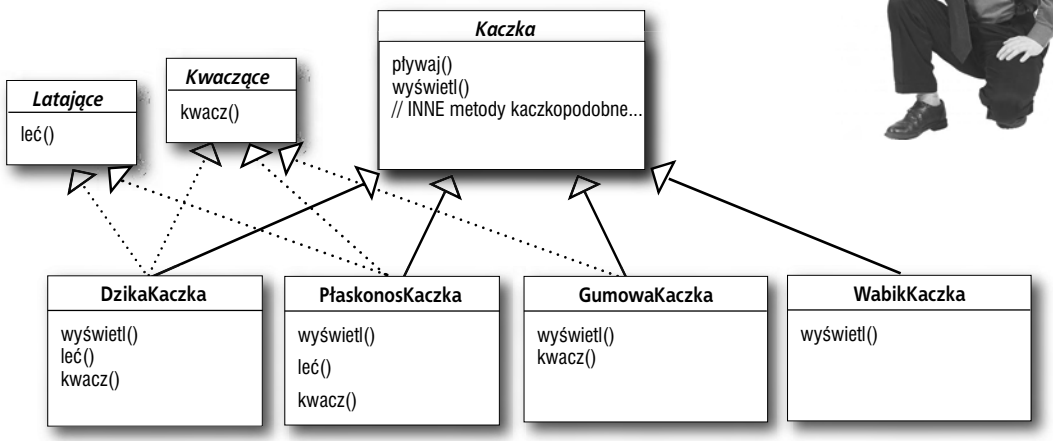
- ☐ **A.** **Kod programu jest duplikowany w kolejnych podklasach.**
- ☐ **B.** **Trudno zmienić zachowania metod podczas działania programu.**
- ☐ **C.** **Nie możemy spowodować, że kaczki zaczną tańczyć.**
- ☐ **D.** **Trudno zebrać informacje o zachowaniach wszystkich kaczek.**
- ☐ **E.** **Kaczki nie mogą latać i kwakać w tym samym czasie.**
- ☐ **F.** **Wprowadzane zmiany mogą mieć niezamierzony wpływ na inne kaczki.**

A może by tak interfejs?

Jacek kilka minut temu uzmysłowił sobie, że dziedziczenie prawdopodobnie nie jest najlepszym rozwiązaniem problemu. Nastąpiło to w momencie, gdy otrzymał dokument stwierdzający, że zarząd firmy postanowił wprowadzać modyfikacje produktu co sześć miesięcy (co gorsza, nie wiadomo z góry, na czym owe modyfikacje mogłyby polegać). Jacek nagle zdał sobie sprawę, że specyfikacja będzie się ciągle zmieniać i z tego powodu będzie zmuszony szczegółowo analizować i prawdopodobnie zmieniać zachowania metod leć() oraz kwacz() dla każdej nowej podklasy klasy nadrzędnej Kaczka, która zostanie dodana do programu… *za każdym razem*.

W takiej sytuacji Jacek potrzebuje lepszego rozwiązania, po prostu takiego, które pozwoli tylko *wybranym* gatunkom kaczek (ale nie *wszystkim*) latać lub kwakać.





Co TY sądzisz o takim projekcie?

Jeśli mam być szczera, to jest najbardziej idiotyczny z Twoich pomysłów. Czy wiesz, co to jest „duplikacja kodu"? Jeżeli uważasz, że zmiana zachowania zaledwie kilku metod była złym pomysłem, to jak będziesz się czuł, kiedy będziesz musiał dokonać drobnych modyfikacji metody opisującej latanie... dla wszystkich 48 podklas latających Kaczek?!



Co byś zrobił na miejscu Jacka?

Wiemy, że nie *wszystkie* podklasy powinny mieć zachowania opisujące latanie czy kwakanie, stąd możemy śmiało założyć, że dziedziczenie nie jest tutaj dobrym rozwiązaniem. Z drugiej jednak strony, choć implementacja interfejsów takich jak Latające i (lub) Kwaczące *częściowo* rozwiązuje nasz problem (gumowe kaczki nie będą już latać wbrew swojej naturze), takie podejście kompletnie niszczy możliwość ponownego wykorzystywania tych samych fragmentów kodu opisującego takie zachowania, czyli, krótko mówiąc, w razie potrzeby wprowadzania nowych klas, poprawek czy modyfikacji otrzymujemy kolejny koszmar. Oczywiście, nie warto już tutaj wspominać o różnych sposobach latania, spotykanych przecież wśród tych kaczek, które *potrafią* latać…

W tym momencie możesz już chyba w zasadzie po cichutku oczekiwać, że odpowiedni *wzorzec projektowy* nadjedzie z fasonem na białym rumaku i załatwi sprawę. Ale czy to byłoby naprawdę zabawne? Zapewne nie, więc mamy zamiar poszukać rozwiązania w dobry, staroświecki sposób — *wykorzystując zasady projektowania oprogramowania zorientowanego obiektowo*.

Czyż nie byłoby cudownie, gdyby istniała możliwość napisania oprogramowania w taki sposób, że jeżeli zaistnieje potrzeba dokonania kolejnych zmian, można będzie ich dokonywać bez konieczności gruntownej modyfikacji istniejącego kodu? Moglibyśmy spędzać znacznie mniej czasu na poprawianiu kodu, a za to poświęcić go więcej na implementację w programach nowych, lepszych funkcji...



Jedyny pewny element w procesie tworzenia oprogramowania

Zgadnij, jak nazywa się jedyny pewny element, na który możesz niezawodnie liczyć podczas tworzenia oprogramowania?

Niezależnie od tego, gdzie pracujesz, co tworzysz i jakiego języka programowania używasz, jak będzie się nazywał jedyny pewny element, który zawsze będzie Ci towarzyszył?

YNAIMS

(użyj lusterka, aby zobaczyć odpowiedź)

Niezależnie od tego, jak dobrze zaprojektujesz aplikację, z upływem czasu musi ona rosnąć i zmieniać się — w przeciwnym wypadku po prostu *umrze* śmiercią naturalną.



Zaostrz ołówek

Zmiany mogą być wywoływane przez szereg różnych czynników. Wypisz kilka powodów, dla których musiałeś dokonywać modyfikacji kodu w Twoich aplikacjach (na dobry początek umieściliśmy kilka własnych przykładów).

Klienci lub użytkownicy zdecydowali, że chcą otrzymać coś zupełnie nowego bądź też rozszerzyć funkcjonalność istniejącego oprogramowania.

Zarząd firmy podjął decyzję, że będziemy korzystać z zupełnie nowej bazy danych innego producenta, a same dane zostaną dostarczone przez dostawcę, który oczywiście wykorzystuje zupełnie inny ich format. Auuć!

Zerowanie problemów

Zdążyliśmy się już chyba oswoić z myślą, że ponieważ zachowania kaczek w poszczególnych podklasach zmieniają się, a co więcej, *różne podklasy* kaczek mogą mieć różne zachowania, mechanizm dziedziczenia w naszym przypadku nie będzie się sprawował zbyt dobrze. Na pierwszy rzut oka pomysł z implementacją interfejsów Latające i Kwaczące wygląda obiecująco — tylko kaczki, które potrafią latać, będą posiadały interfejs Latające itd. — z wyjątkiem tego, że interfejsy Javy nie posiadają własnego kodu implementującego, więc nie można wielokrotnie wykorzystywać fragmentów takiego kodu. Oznacza to, że jeżeli przyjdzie Ci modyfikować jakieś określone zachowanie, będziesz zmuszony prześledzić wszystkie podklasy danej superklasy, wyszukać w nich poszczególne implementacje danego zachowania, a następnie dokonać odpowiednich modyfikacji — o tym, że przy okazji zapewne „dorzucisz" do modyfikowanego kodu *nowe* błędy, nie warto już nawet wspominać!

Na szczęście istnieje odpowiednia na taką okazję *reguła projektowania*:



Reguła projektowania

Zidentyfikuje fragmenty aplikacji, które się zmieniają i oddziel je od tych, które pozostają stałe.

Jest to nasza pierwsza i na pewno nie ostatnia z reguł projektowania. W dalszych częściach niniejszej książki poświęcimy im zapewne jeszcze nieco miejsca.

Przykładowo, jeżeli zidentyfikujesz jakiś dany fragment kodu, który ulega zmianie, dajmy na to, za każdym razem, kiedy otrzymujesz nowe wymagania dla aplikacji, wiesz, że znalazłeś zachowanie, które powinno zostać „wyciągnięte" z kodu i oddzielone od całej reszty, która nie ulega zmianom.

Przedstawioną powyżej zasadę możesz interpretować również w inny sposób: weź elementy, które się zmieniają, i dokonaj ich hermetyzacji, tak aby ich modyfikacja i aktualizacja w przyszłości nie pociągała za sobą konieczności dokonywania jakichkolwiek zmian w elementach stałych.

Pomimo swojej uderzającej prostoty powyższa zasada stanowi praktycznie podwaliny pod niemal każdy wzorzec projektowy. Wszystkie wzorce zapewniają w taki czy inny sposób to, aby pewne fragmenty systemu mogły być modyfikowane niezależnie od innych.

OK, a zatem do dzieła — czas wyciągnąć definicje zachowań kaczek z poszczególnych podklas klasy nadrzędnej Kaczka.

Weź to, co się zmienia, i dokonaj odpowiedniej hermetyzacji, tak aby nie miało to żadnego wpływu na pozostałą część Twojego kodu.

Rezultat? Mniej niezamierzonych efektów ubocznych będących konsekwencją wprowadzania nowych zmian w kodzie aplikacji oraz większa elastyczność całego systemu.

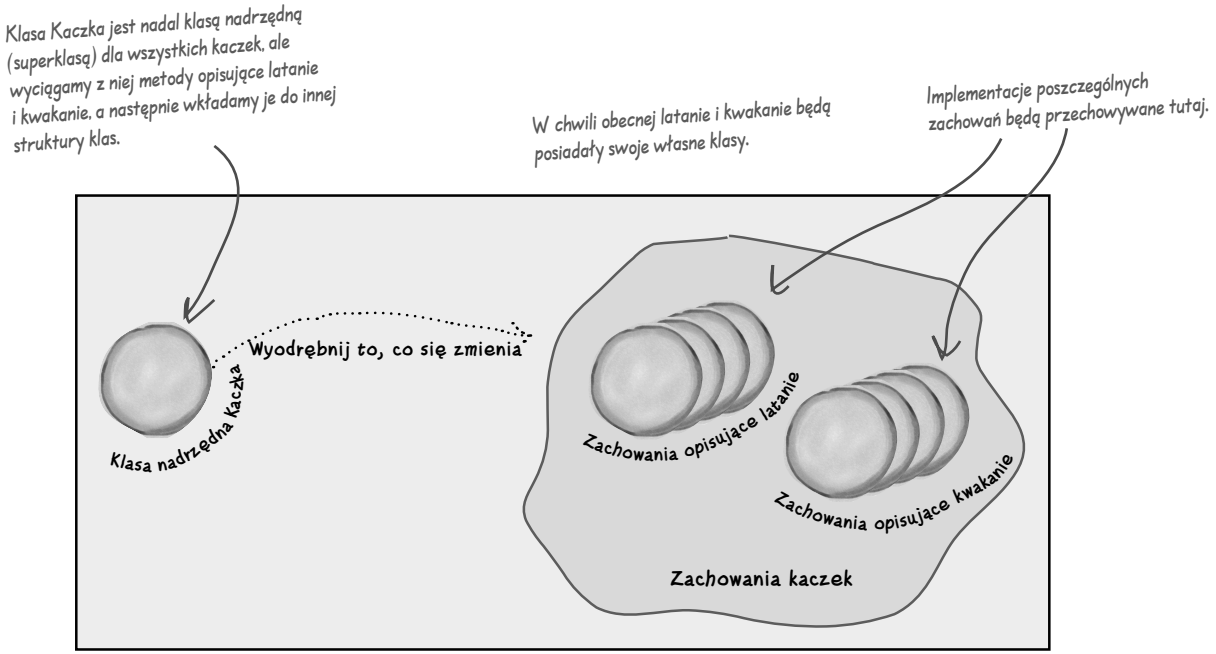
Oddzielanie tego, co się zmienia, od tego, co pozostaje niezmienione

Gdzie mamy rozpocząć? Jak zapewne pamiętasz, poza „drobnymi" problemami z metodami leć() i kwacz() klasa Kaczka radziła sobie całkiem dobrze i wyglądało na to, że inne elementy tej klasy nie będą się zbyt często zmieniały. Z tego względu poza wprowadzeniem kilku niezbyt dużych modyfikacji będziemy się starali zostawić klasę Kaczka we względnym spokoju.

Teraz, aby oddzielić „elementy, które się zmieniają, od tych, które pozostają niezmienione", będziemy chcieli utworzyć dwa *zestawy* klas (całkowicie niezależnych od superklasy Kaczka), jeden dla metody *leć()* i drugi dla metody *kwacz()*. Każdy z zestawów klas będzie posiadał swoją niezależną implementację odpowiednich zachowań. Przykładowo, może się zdarzyć tak, że będziesz chciał miał mieć osobną klasę odpowiadającą za *kwakanie*, inną klasę odpowiadającą za *piszczenie* i wreszcie *jeszcze jedną*, która będzie odpowiadała za *niewydawanie żadnych dźwięków*.

Wiemy, że metody leć() oraz kwacz() są tymi częściami superklasy Kaczka, które podlegają zmianom „od kaczki do kaczki".

Aby wydzielić omawiane zachowania z superklasy Kaczka, musimy wyciągnąć obydwie metody z wnętrza tej klasy, a następnie utworzyć zestaw nowych klas reprezentujących poszczególne zachowania.



Projektowanie zachowania Kaczki

W jaki sposób zaprojektujemy zestaw klas, które będą odpowiedzialne za implementację latania i kwakania?

Przede wszystkim chcielibyśmy, aby nasz projekt był elastyczny — w końcu, jak by na to nie patrzeć, to właśnie sztywność zachowania naszych poprzednich kaczek wpędziła Jacka w problemy. Wiemy również, że chcielibyśmy *przypisywać* odpowiednie zachowania poszczególnym obiektom klasy Kaczka. Przykładowo, moglibyśmy utworzyć nowy egzemplarz obiektu DzikaKaczka, a następnie zainicjować go konkretnym *rodzajem* latania. A jeżeli już przy tym jesteśmy, to dlaczego nie upewnić się wcześniej, że będziemy w stanie dynamicznie zmieniać zachowania danej kaczki? Innymi słowy, powinniśmy do definicji klasy Kaczka dołączyć metody pozwalające na dynamiczne przypisywanie zachowań, tak abyśmy mogli *zmieniać* zachowania obiektu DzikaKaczka *w czasie działania* programu.

Mając na uwadze takie cele, rzućmy okiem na naszą drugą *regułę projektowania*:



Reguła projektowania

Skoncentruj się na tworzeniu interfejsów, a nie implementacji.

Do reprezentacji poszczególnych zachowań będziemy wykorzystywali odpowiednie interfejsy — przykładowo LatanieInterfejs i KwakanieInterfejs — a każda implementacja danego zachowania będzie posiadała jeden z tych interfejsów.

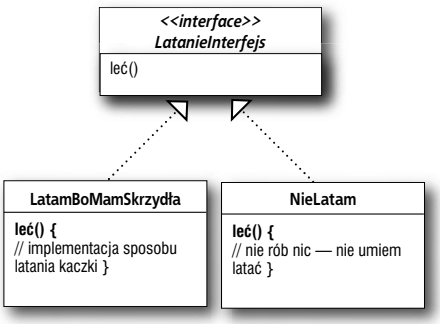
Tym razem jednak to nie klasa Kaczka będzie przechowywała implementacje wspomnianych interfejsów. Zamiast tego utworzymy szereg klas, których jedynym przeznaczeniem będzie reprezentowanie poszczególnych zachowań (np. „piszczenie") — i to właśnie one będą przechowywały te implementacje.

Takie podejście jest zupełnie różne od naszego dotychczasowego postępowania, gdzie zachowanie było determinowane określoną implementacją w klasie nadrzędnej Kaczka lub indywidualną implementacją w samej podklasie. W obydwu przypadkach polegaliśmy na *implementacji*. W efekcie nie było innego wyjścia, jak tylko korzystać z istniejącej implementacji zachowania, a co gorsza, praktycznie nie było sposobów na zmianę danego zachowania (innych niż bezpośrednia modyfikacja kodu programu).

W naszym nowym projekcie klasa nadrzędna Kaczka będzie wykorzystywała zachowania reprezentowane przez *interfejsy* (LatanieInterfejs i KwakanieInterfejs), dzięki czemu właściwa *implementacja* poszczególnych zachowań nie będzie zablokowana w superklasie Kaczka (innymi słowy, rzeczywista implementacja danego zachowania zostanie umieszczona w odpowiedniej klasie będącej częścią składową danego interfejsu).

Od tej chwili zachowania klasy Kaczka będą umieszczone w osobnej klasie — czyli klasie, która przechowuje implementację interfejsu reprezentującego dane zachowanie.

W ten sposób superklasa Kaczka nie musi znać żadnych szczegółów implementacji swoich własnych zachowań.

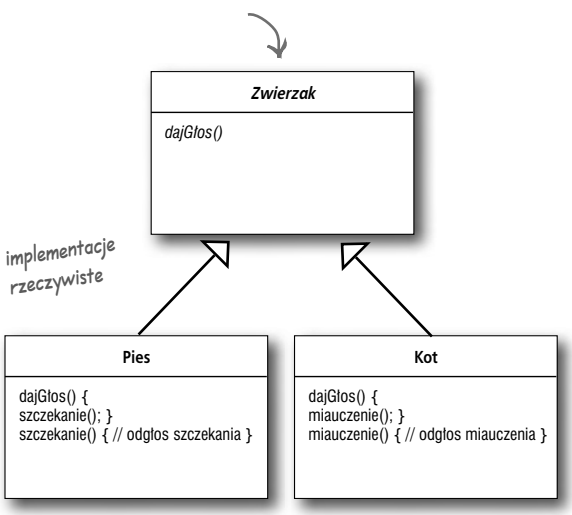




„Skoncentruj się na tworzeniu interfejsów" oznacza „skoncentruj się na tworzeniu supertypów"

Słowo *interfejs* jest tutaj zdecydowanie zbyt przeładowane. Z jednej strony, mamy samą jego *ideę*, a z drugiej strony `interfejs` związany bezpośrednio ze słowem kluczowym interface języka Java. Warto zatem uświadomić sobie, że przedstawioną wcześniej regułę „skoncentruj się na tworzeniu `interfejsów`" można z powodzeniem wcielać w życie bez konieczności użycia słowa kluczowego interface i tworzenia interfejsu w ścisłym rozumieniu składni języka Java. Cała sztuka polega na tym, aby odpowiednio wykorzystywać polimorfizm i tworzyć odpowiednie supertypy tak, żeby zachowanie poszczególnych obiektów nie było ściśle związane z ich własnym kodem. W zasadzie moglibyśmy więc zamiast zalecać „skoncentruj się na tworzeniu supertypów" powiedzieć, że „deklarowanym typem zmiennych powinien być supertyp, czyli zazwyczaj klasa abstrakcyjna bądź interfejs — chodzi o to, aby obiekty przypisane do tych zmiennych mogły przyjmować postać dowolnej, rzeczywistej implementacji supertypu, co w praktyce oznacza, że klasa, która je deklaruje, nie musi *znać* prawdziwych typów poszczególnych obiektów!".

Prawdopodobnie to, co teraz powiemy, nie będzie dla Ciebie niczym nowym, ale aby mieć pewność, że myślimy tak samo jak Ty, przedstawimy prosty przykład zastosowania typu polimorficznego. Wyobraź sobie klasę abstrakcyjną o nazwie Zwierzak, posiadającą dwie rzeczywiste implementacje, Pies i Kot.



Bezpośrednia implementacja kodu może wyglądać następująco:

```
Pies p = new Pies();
p.szczekaj();
```

Deklaracja zmiennej „p" jako zmiennej typu Pies (rzeczywista implementacja Zwierzaka) wymusza tworzenie kodu odpowiadającego danej implementacji.

Ale za to **utworzenie odpowiedniego interfejsu (lub supertypu)** może wyglądać następująco:

```
Zwierzak zwierzak = new Pies();
zwierzak.dajGłos();
```

Wiemy, że to jest Pies, ale możemy tutaj użyć polimorficznego odwołania do tego zwierzaka.

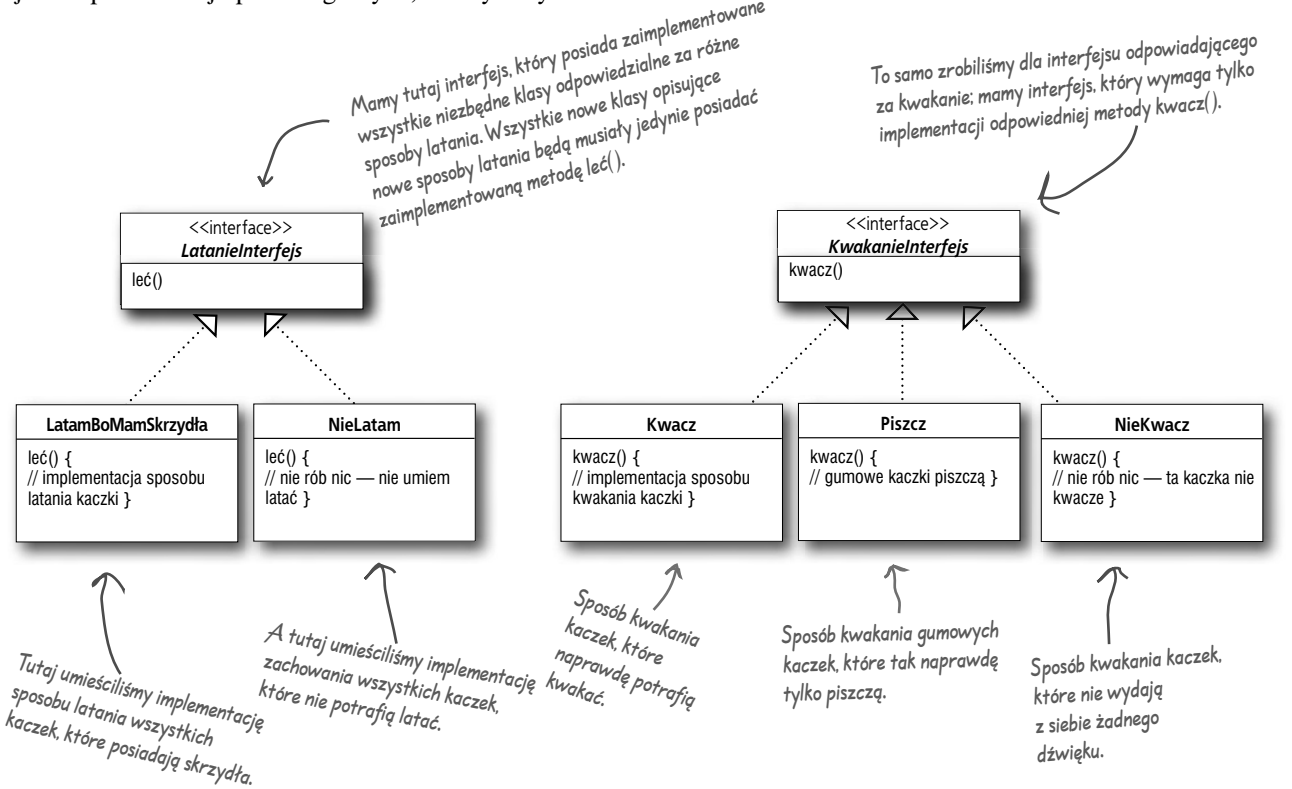
Sztywne umieszczenie w kodzie tworzenia nowego obiektu (jak np. new Pies()) jest niezłym rozwiązaniem, ale jeszcze lepszym będzie tworzenie rzeczywistej implementacji obiektu bezpośrednio w czasie działania programu:

```
a = getZwierzak();
a.dajGłos();
```

Nie wiemy, jaki jest FAKTYCZNY podtyp Zwierzaka w danej chwili... jednak wszystko, o co musimy zadbać, to żeby program „wiedział", jak odpowiedzieć na polecenie a.dajGłos().

Implementacja zachowań Kaczki

Poniżej przedstawiamy dwa interfejsy, LatanieInterfejs oraz KwakanieInterfejs, łącznie z odpowiednimi klasami, których zadaniem jest implementacja poszczególnych, rzeczywistych zachowań:



Dzięki takiemu projektowi inne typy obiektów mogą wykorzystywać odpowiednie zachowania (latanie i kwakanie), ponieważ nie są one już dłużej ukryte w klasie nadrzędnej Kaczka!

W razie potrzeby możemy dodawać nowe zachowania bez konieczności modyfikacji jakichkolwiek istniejących klas opisujących dotychczasowe zachowania i bez modyfikacji tych klas opisujących Kaczki, które *wykorzystują* zachowania związane z lataniem.

Jak widać, możemy swobodnie korzystać z zalet, jakie daje możliwość WIELOKROTNEGO WYKORZYSTYWANIA kodu i to bez tego całego niepotrzebnego bagażu, jaki niesie ze sobą mechanizm dziedziczenia.

Nie istnieją głupie pytania

P: Czy zawsze powinienem najpierw utworzyć aplikację, zidentyfikować elementy, które się zmieniają, i dopiero wtedy oddzielić je i odpowiednio opakować?

O: Nie zawsze. Często zdarza się tak, że już na etapie projektowania aplikacji można przewidzieć obszary, które będą podlegały zmianom, i dzięki temu niejako z wyprzedzeniem zaprojektować i zaimplementować odpowiednio elastyczny kod. Z lektury naszej książki dowiesz się, że odpowiednie reguły i wzorce projektowe mogą być wykorzystywane na niemal każdym etapie tworzenia aplikacji.

P: Czy powinniśmy utworzyć również interfejs Kaczka?

O: Nie w tym przypadku. Jak się niebawem przekonasz, po złożeniu wszystkiego w jedną całość bardzo ułatwi nam życie fakt, że będziemy mieli Kaczkę jako rzeczywistą implementację klasy oraz poszczególne rodzaje kaczek, jak np. DzikaKaczka, które będą z niej dziedziczyły wspólne właściwości i metody. Dzięki temu, że teraz usunęliśmy elementy specyficzne dla dziedziczenia, możemy korzystać ze struktury, która posiada wszystkie zalety dziedziczenia, ale nie powiela jego wad i problemów.

P: Posiadanie klasy, która opisuje tylko zachowanie, wydaje się nieco dziwne. Czy klasy nie powinny przypadkiem reprezentować *czegoś bardziej materialnego*? *Czy klasy nie powinny posiadać stanu ORAZ zachowania*?

O: W systemach zorientowanych obiektowo — tak, klasy reprezentują elementy, które — ogólnie rzecz biorąc — posiadają zarówno stan (zapisany w zmiennych obiektowych), jak i odpowiednie metody. W takim przypadku można powiedzieć, że to właśnie takie elementy charakteryzują się określonymi zachowaniami. Należy jednak pamiętać o tym, że nawet dane zachowanie może posiadać stan i swoje metody; przykładowo, zachowanie związane z lataniem może posiadać swoje zmienne obiektowe reprezentujące odpowiednie atrybuty latania (takie jak ilość uderzeń skrzydłami na minutę, maksymalna szybkość i wysokość lotu itp.).



Zaostrz ołówek

1. W jaki sposób przy wykorzystaniu naszego nowego projektu możesz dodać możliwość latania z napędem rakietowym do naszej aplikacji SymulatorKaczki?

2. Czy myślałeś kiedyś o klasie, która mogłaby używać zachowania Kwacz, a która nie opisywałaby kaczki?

Odpowiedzi:

1) Wystarczy utworzyć klasę o nazwie np. LotZNapędemRakietowym, która będzie miała zaimplementowaną odpowiednią metodę lec().

2) Przykładem takiej klasy może być klasa opisująca wabik (czyli swego rodzaju gwizdek służący do przywabiania kaczek, wydający z siebie dźwięki naśladujące odgłosy wydawane przez kaczki).

Integracja zachowania Kaczki

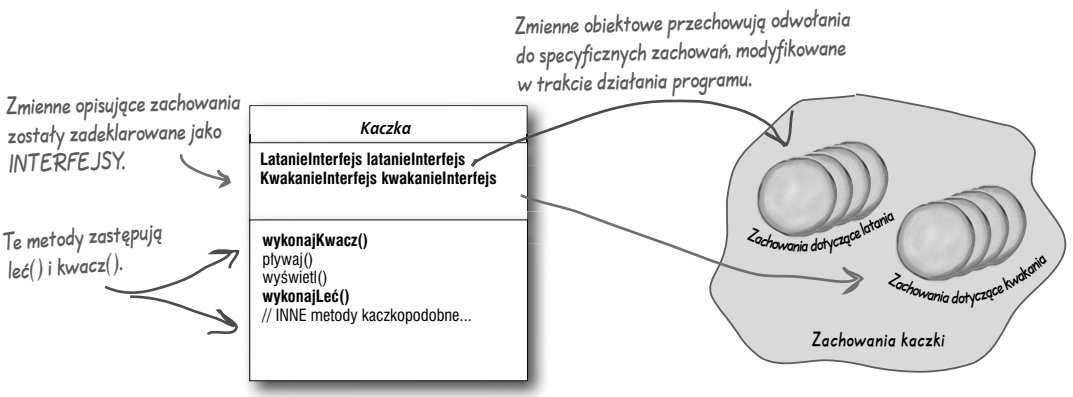
Elementem kluczowym jest teraz fakt, że obiekt Kaczka zamiast wykorzystywać metody opisujące latanie i kwakanie zdefiniowane w superklasie Kaczka (bądź w klasach podrzędnych), będzie *delegować* te zachowania do innych obiektów.

Zaraz dowiesz się, jak tego dokonać:

❶ **Po pierwsze, w definicji klasy Kaczka umieścimy dwie zmienne obiektowe**, nazywane *latanieInterfejs* i *kwakanieInterfejs*, które zostaną zdefiniowane jako zmienne typu *interfejs* (a nie jako implementacja rzeczywistego typu zmiennej). Każdy obiekt reprezentujący kaczkę będzie polimorficznie ustawiał te zmienne na odwołania do *określonego* typu zachowania — i to wszystko w zależności od potrzeb — w czasie działania programu (LatamBoMamSkrzydła, Piszcz itp.).

Z klasy nadrzędnej Kaczka (jak również ze wszystkich podklas) usuniemy metody leć() oraz kwacz(), co wynika z faktu, że odpowiadające im zachowania zostały przeniesione do klas LatanieInterfejs oraz KwakanieInterfejs.

Zamienimy definicje metod leć() i kwacz() w klasie Kaczka na dwie podobne metody, o nazwach wykonajLeć() oraz wykonajKwacz() — niebawem dowiesz się, jakie jest ich przeznaczenie.



❷ **Teraz zajmiemy się implementacją zmiennej wykonajKwacz():**

```
public class Kaczka {
   KwakanieInterfejs kwakanieInterfejs;
   // więcej

   public void wykonajKwacz() {
   kwakanieInterfejs.kwacz();
   }
}
```

Każda Kaczka posiada odwołania do czegoś, co ma zaimplementowany interfejs KwakanieInterfejs.

Zamiast samodzielnie obsługiwać różne sposoby kwakania, obiekt Kaczka deleguje obsługę tego zachowania do obiektu wskazywanego przez zmienną kwakanieInterfejs.

Całkiem proste, prawda? Aby zakwakać, Kaczka po prostu pozwala obiektowi wskazywanemu przez zmienną kwakanieInterfejs na wydawanie odgłosów kwakania.

Na tym etapie tworzenia kodu zupełnie nie dbamy o to, jaki to jest typ obiektu. ***Jedyne, na co musimy zwracać uwagę, to aby taki obiekt wiedział, jak wykonać metodę kwacz()!***

Więcej integracji...

❸ No dobrze, teraz nadszedł czas, aby zacząć myśleć o tym, **w jaki sposób będą ustawiane wartości zmiennych obiektowych latanieInterfejs oraz kwakanieInterfejs**. Rzućmy okiem na definicję klasy DzikaKaczka:

```
public class DzikaKaczka extends Kaczka {

    public DzikaKaczka() {
        kwakanieInterfejs = new Kwacz();
        latanieInterfejs = new LatamBoMamSkrzydła();
    }

   public void wyświetl() {
        System.out.println("Jestem prawdziwą Dziką Kaczką");
    }
}
```

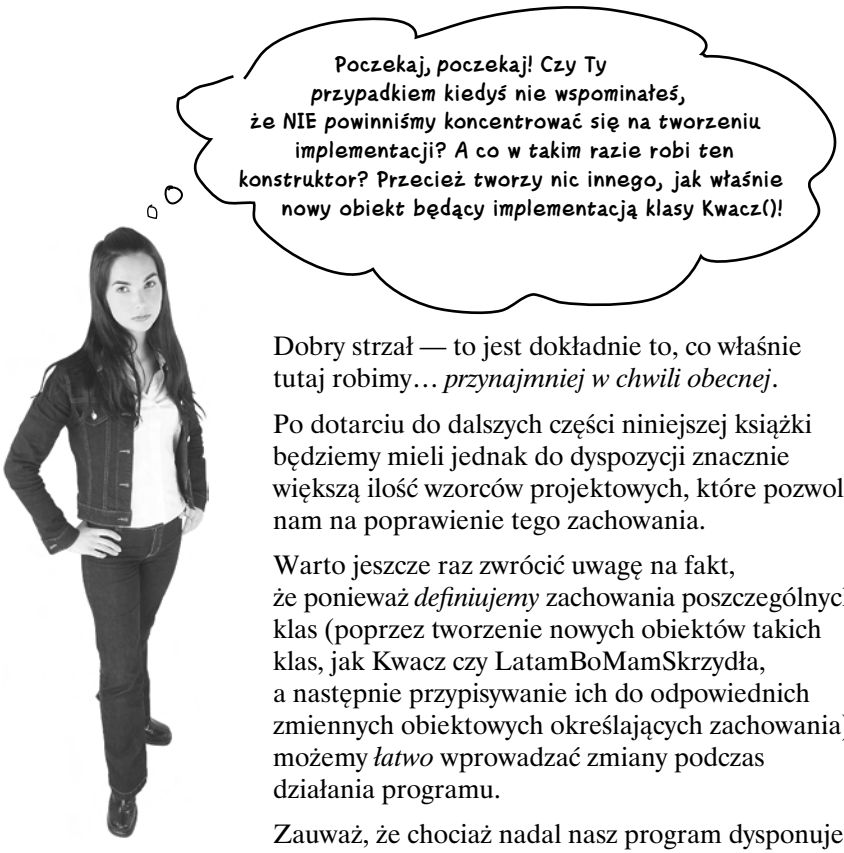
DzikaKaczka wykorzystuje klasę Kwacz do obsługi procesu kwakania, stąd w momencie, kiedy wywoływana jest metoda wykonajKwacz(), odpowiedzialność za przebieg kwakania jest delegowana do obiektu Kwacz. Dzięki temu otrzymujemy poprawne kwakanie dla danego typu obiektu.

Pamiętaj, DzikaKaczka dziedziczy zmienne obiektowe kwakanieInterfejs oraz latanieInterfejs z klasy Kaczka.

Klasa ta wykorzystuje klasę LatamBoMamSkrzydła jako definicję sposobu latania.

Jak wiemy, w realnym świecie dzika kaczka kwacze dokładnie tak, jak się tego możemy po kaczkach spodziewać — jest to prawdziwe **kwakanie**, a nie **piszczenie** czy też **milczenie**. Zatem, co się dzieje w naszym programie? Kiedy nowy obiekt DzikaKaczka zostaje utworzony, jego konstruktor przypisuje do dziedziczonej zmiennej obiektowej kwakanieInterfejs nowy obiekt typu Kwacz() (jest on rzeczywistą implementacją klasy KwakanieInterfejs).

Opisany powyżej proces przebiega identycznie dla zachowania opisującego sposób latania — konstruktor klasy DzikaKaczka przypisuje do dziedziczonej zmiennej obiektowej latanieInterfejs nowy obiekt typu LatamBoMamSkrzydła() (jest on rzeczywistą implementacją klasy LatanieInterfejs).



Dobry strzał — to jest dokładnie to, co właśnie tutaj robimy… *przynajmniej w chwili obecnej*.

Po dotarciu do dalszych części niniejszej książki będziemy mieli jednak do dyspozycji znacznie większą ilość wzorców projektowych, które pozwolą nam na poprawienie tego zachowania.

Warto jeszcze raz zwrócić uwagę na fakt, że ponieważ *definiujemy* zachowania poszczególnych klas (poprzez tworzenie nowych obiektów takich klas, jak Kwacz czy LatamBoMamSkrzydła, a następnie przypisywanie ich do odpowiednich zmiennych obiektowych określających zachowania), możemy *łatwo* wprowadzać zmiany podczas działania programu.

Zauważ, że chociaż nadal nasz program dysponuje dużą elastycznością, to nie radzimy sobie zbyt dobrze z inicjowaniem zmiennych obiektowych w taki elastyczny sposób. Pomyśl jednak o tym, ponieważ zmienna obiektowa kwakanieInterfejs jest typu interfejsu, a więc (wykorzystując magię polimorfizmu) możemy do niej dynamicznie w czasie działania programu przypisywać wybrane klasy implementujące poszczególne zachowania.

Znajdź chwilę na zastanowienie się, w jaki sposób można by zaimplementować kaczkę, której zachowania będą mogły zmieniać się dynamicznie w trakcie działania programu (przykładowy kod takiego programu znajdziesz kilka stron dalej).

Testowanie kodu klasy Kaczka

❶ Wpisz i skompiluj kod klasy Kaczka podany poniżej (plik Kaczka.java) oraz kod klasy DzikaKaczka (plik DzikaKaczka.java), który znajdziesz dwie strony wcześniej.

```
public abstract class Kaczka {

    LatanieInterfejs latanieInterfejs;
    KwakanieInterfejs kwakanieInterfejs;
    public Kaczka() {
    }

    public abstract void wyświetl();

    public void wykonajLeć() {
        latanieInterfejs.leć();
    }

    public void wykonajKwacz() {
        kwakanieInterfejs.kwacz();
    }

    public void pływaj() {
        System.out.println("Wszystkie kaczki pływają, nawet te sztuczne!");
    }
}
```

Deklaruje dwie zmienne typu interfejsu dla poszczególnych zachowań. Wszystkie klasy podrzędne superklasy Kaczka (w tym samym pakiecie) będą dziedziczyły te zmienne.

Delegacja zachowania do odpowiedniej klasy.

❷ Wpisz i skompiluj kod interfejsu LatanieInterfejs (plik LatanieInterfejs.java) oraz kody dwóch klas implementujących zachowania (pliki LatamBoMamSkrzydła.java oraz NieLatam.java).

```
public interface LatanieInterfejs {
    public void leć();
}
```

Interfejs, który posiadają wszystkie klasy opisujące to zachowanie (latanie).

```
public class LatamBoMamSkrzydła implements LatanieInterfejs {
    public void leć() {
        System.out.println("O rany! Latam!");
    }
}
```

Implementacja sposobu latania kaczek, które NAPRAWDĘ potrafią latać...

```
public class NieLatam implements LatanieInterfejs {
    public void leć() {
        System.out.println("Nie umiem latać!");
    }
}
```

Implementacja sposobu latania kaczek, które tak naprawdę nie potrafią latać (jak np. kaczki gumowe czy kaczki wabiki).

Testowania kodu klasy Kaczka ciąg dalszy...

❸ Wpisz i skompiluj kod interfejsu KwakanieInterfejs (plik KwakanieInterfejs.java) oraz kody trzech klas implementujących zachowania (pliki Kwacz.java, NieKwacz.java oraz Piszcz.java).

```
public interface KwakanieInterfejs {
    public void kwacz();
}
```

```
public class Kwacz implements KwakanieInterfejs {
    public void kwacz() {
        System.out.println("Kwa! Kwa!");
    }
}
```

```
public class NieKwacz implements KwakanieInterfejs {
    public void kwacz() {
        System.out.println("<<<CISZA>>>");
    }
}
```

```
public class Piszcz implements KwakanieInterfejs {
    public void kwacz() {
        System.out.println("Piszczę!");
    }
}
```

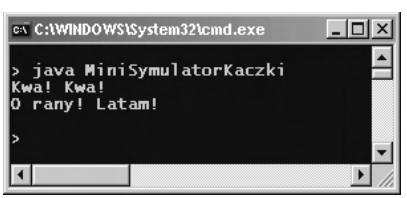
❹ Wpisz i skompiluj klasę testową (MiniSymulatorKaczki.java).

```
public class MiniSymulatorKaczki {
    public static void main(String[] args) {
        Kaczka dzika = new DzikaKaczka();
        dzika.wykonajKwacz();
        dzika.wykonajLeć();
    }
}
```

To polecenie powoduje wywołanie dziedziczonej metody wykonajKwacz(), która następnie deleguje obsługę zachowania do właściwej metody interfejsu KwakanieInterfejs (przykładowo, wywołuje metodę kwacz() właściwą dla rodzaju obiektu wskazywanego przez dziedziczoną zmienną kwakanieInterfejs).

Następnie robimy dokładnie to samo z dziedziczoną przez klasę DzikaKaczka metodą wykonajLeć().

❺ Uruchom kod programu!



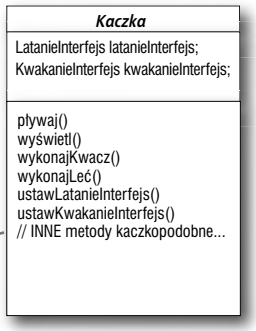
Dynamiczne ustawianie zachowania

Co za wstyd! Mamy wszystkie niezbędne mechanizmy dynamiczne wbudowane naszym kaczkom, a nie korzystamy z nich! Wyobraź sobie, że chcesz ustawić zachowanie kaczki poprzez zastosowanie specjalnej metody pozwalającej wprost na zmianę zachowania odpowiedniej klasy podrzędnej, zamiast poprzez wprowadzanie odpowiednich zmian w konstruktorze kaczki.

❶ Dodaj dwie nowe metody do klasy Kaczka:

```
public void ustawLatanieInterfejs (LatanieInterfejs li) {
    latanieInterfejs = li;
}

public void ustawKwakanieInterfejs(KwakanieInterfejs ki) {
    kwakanieInterfejs = ki;
}
```



Metody te możemy wywoływać w dowolnym momencie, kiedy tylko będziemy chcieli zmienić zachowanie kaczek „w locie".

dopisek redaktora: niepotrzebna dwuznaczność — poprawić

❷ Utwórz nowy typ Kaczki (plik ModelKaczki.java)

```
public class ModelKaczki extends Kaczka {
    public ModelKaczki() {
        latanieInterfejs = new NieLatam();
        kwakanieInterfejs = new Kwacz();
    }

    public void wyświetl() {
        System.out.println("Jestem modelem kaczki!");
    }
}
```

Nasz model kaczki rozpoczyna swoją egzystencję nieco „uziemiony" … bez możliwości latania.

❸ Utwórz nowy typ klasy LatanieInterfejs (plik LotZNapędemRakietowym.java).

Tak właśnie ma być, tworzymy przecież zachowanie opisujące latanie przy użyciu napędu rakietowego.

```
public class LotZNapędemRakietowym implements LatanieInterfejs {
    public void leć() {
        System.out.println("Uuuuaaa! Lot z napędem rakietowym!");
    }
}
```



❹ **Zmień klasę testową (plik MiniSymulatorKaczki.java), dodaj do niego ModelKaczki i spraw, aby ModelKaczki mógł korzystać w czasie lotu z napędu rakietowego.**

```
public class MiniSymulatorKaczki {
    public static void main(String[] args) {
        Kaczka dzika = new DzikaKaczka();
        dzika.wykonajKwacz();
        dzika.wykonajLeć();

        Kaczka model = new ModelKaczki();
        model.wykonajLeć();
        model.ustawLatanieInterfejs(new LotZNapędemRakietowym());
        model.wykonajLeć();
    }
}
```

przed



Pierwsze wywołanie metody wykonajLeć() deleguje obsługę zachowania do odpowiedniej metody obiektu, na który wskazuje zmienna latanieInterfejs ustawiona w konstruktorze klasy ModelKaczki, czyli w tym przypadku obiektu typu NieLatam.

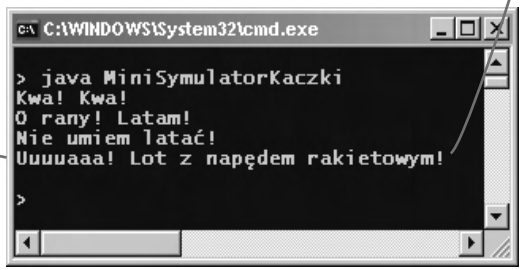
To polecenie wywołuje dziedziczoną przez model metodę, która pozwala na dynamiczną zmianę zachowania i... proszę bardzo! Nasz model nagle nie dość, że lata, to jeszcze posiada napęd rakietowy!

Jeżeli to zadziałało, oznacza to ni mniej, ni więcej, że nasz model kaczki dynamicznie zmienił swoje zachowanie! Nie mógłbyś TEGO dokonać, jeżeli cała implementacja byłaby umieszczona wewnątrz klasy Kaczka.



po

❺ **Uruchom ten program!**



Aby zmienić zachowanie kaczki w czasie działania programu, wystarczy wywołać odpowiednią metodę (ang. *behavior setter method*), która pozwala na dynamiczną zmianę zachowania obiektu.

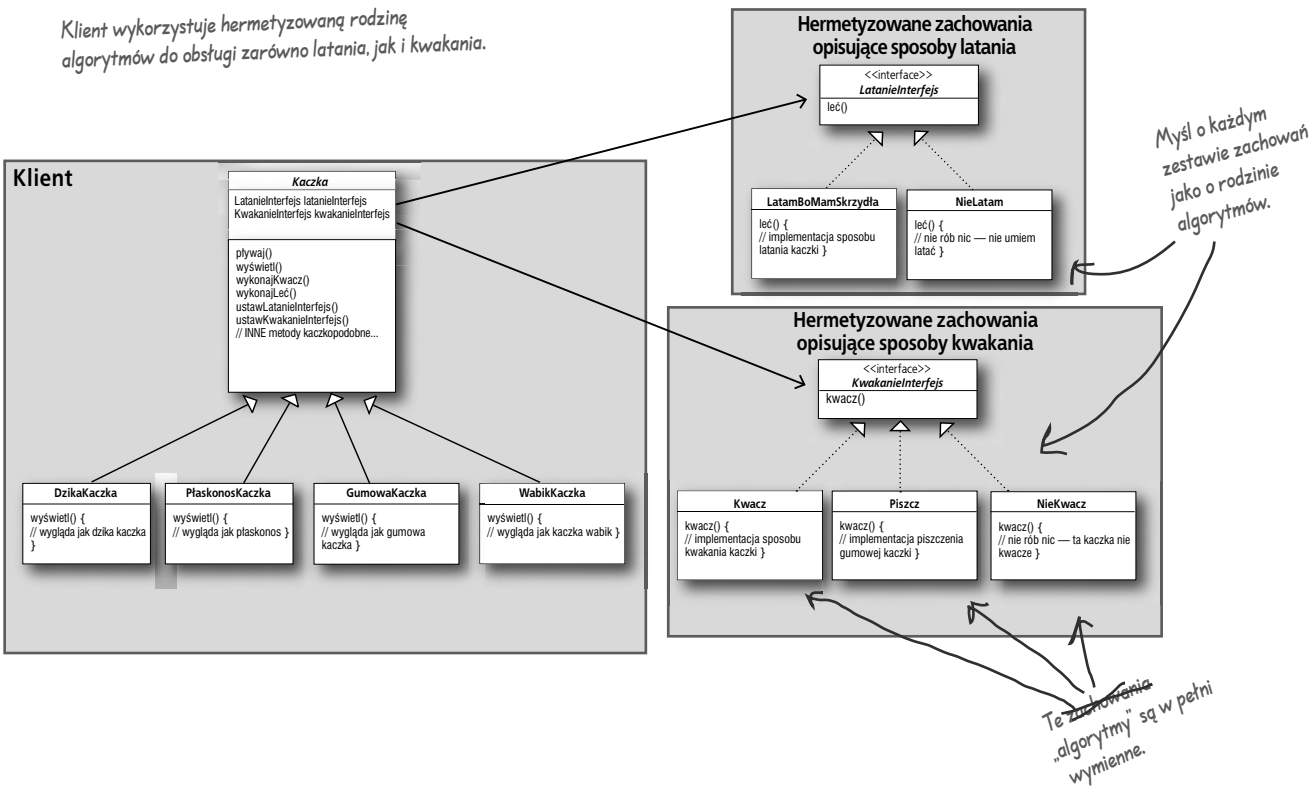
Wielki diagram „ukrytych" zachowań

Jak by na to nie patrzeć, dokonaliśmy już całkiem głębokiego zanurzenia w szczegółowe zagadnienia związane z projektowaniem symulatora kaczki. Nadszedł zatem czas na wynurzenie, zaczerpnięcie powietrza i rzut oka na całość zagadnienia z tzw. szerokiej perspektywy.

Na poniższym diagramie przedstawiliśmy całą przeprojektowaną strukturę klas. Mamy tu wszystko, czego mógłbyś oczekiwać: poszczególne kaczki rozszerzają funkcjonalność klasy Kaczka, sposoby latania zostały zaimplementowane w podklasach interfejsu LatanieInterfejs, a sposoby kwakania zostały zaimplementowane w podklasach interfejsu KwakanieInterfejs.

Zwróć także uwagę, że zaczynamy nieco inaczej opisywać pewne elementy. Zamiast myśleć o zachowaniach kaczek jako o *zbiorze zachowań*, będziemy o nich myśleć jako o *rodzinie algorytmów*. Pomyśl o tym, że w projekcie SymulatoraKaczki algorytmy reprezentują co prawda poszczególne czynności, jakie kaczka może wykonać (różne sposoby latania lub kwakania), ale przecież równie dobrze moglibyśmy użyć takich samych technik do utworzenia zestawu klas, które implementują zasady obliczania podatku dochodowego przy uwzględnieniu rodzaju płatnika, różnych stawek podatkowych oraz obowiązujących ulg i zwolnień.

Zwróć szczególną uwagę na *relacje* pomiędzy poszczególnymi klasami. Najlepiej będzie, jeśli po prostu weźmiesz pióro i zapiszesz nazwy odpowiednich relacji (JEST, MA oraz IMPLEMENTUJE) na poszczególnych strzałkach diagramu klas.



Relacja MA może być lepsza niż JEST

Relacja MA jest bardzo ciekawa: każda kaczka ma interfejsy LatanieInterfejs oraz KwakanieInterfejs, do których deleguje zachowania „latanie" i „kwakanie".

Kiedy w ten sposób łączysz dwie klasy, używasz kompozycji. Zamiast *dziedziczyć* swoje zachowania, kaczki otrzymują je w ten sposób, że są *komponowane* z określonych obiektów implementujących poszczególne zachowania.

Jest to bardzo ważna technika, która znajduje swoje odzwierciedlenie w naszej trzeciej *regule projektowania*:



Reguła projektowania

Przedkładaj kompozycję nad dziedziczenie.

Jak zapewne zdążyłeś zauważyć, tworzenie systemów metodą kompozycji daje w efekcie o wiele więcej elastyczności. Nie tylko pozwala na pakowanie całych rodzin algorytmów do ich własnych zestawów klas, ale również na zmianę *zachowania w czasie działania programu* tak długo, jak długo obiekty, których używasz do kompozycji, implementują odpowiednie interfejsy zachowań.

Kompozycja jest wykorzystywana w wielu różnych wzorcach projektowych; w czasie pracy z niniejszą książką poznasz jeszcze wiele jej zalet, ale również dowiesz się o jej wadach.



WYTĘŻ UMYSŁ

Wabik to rodzaj specjalnego gwizdka, którego myśliwi używają do naśladowania odgłosów wydawanych przez kaczki (odgłosów kwakania). Zastanów się, w jaki sposób mógłbyś zaimplementować swój własny wabik, który nie dziedziczyłby zachowań z klasy Kaczka?



Mistrz i uczeń...

Mistrz: *Młody człowieku, powiedz mi, czego nauczyłeś się o filozofii zorientowanej obiektowo.*

Uczeń: *Mistrzu, nauczyłem się, że najbardziej obiecująca jest możliwość wielokrotnego wykorzystywania tych samych elementów.*

Mistrz: *Kontynuuj, młody człowieku...*

Uczeń: *Mistrzu, dzięki możliwościom, jakie daje dziedziczenie, wszystkie dobre rzeczy mogą być wykorzystywane po wielokroć, dzięki czemu czas, który ma być poświęcony na projektowanie, kurczy się tak samo szybko, jak zagajnik sprawnie wycinanego bambusa...*

Mistrz: *Młody człowieku, czy więcej czasu poświęcamy Kodowi **przed** czy **po** zakończeniu projektowania?*

Uczeń: *Odpowiedź brzmi: **po**, Mistrzu. Zawsze spędzamy więcej czasu na konserwacji i wprowadzaniu zmian do Kodu niż na jego początkowym projektowaniu.*

Mistrz: *Więc, młody człowieku, czy nasze wysiłki powinny iść w stronę zapewnienia **jak największych** możliwości wielokrotnego wykorzystywania rzeczy nawet kosztem pogorszenia możliwości obsługi i rozbudowy?*

Uczeń: *Mistrzu, wierzę, że tak właśnie wygląda prawda.*

Mistrz: *Widzę, że jeszcze musisz się sporo nauczyć, młody człowieku. Chciałbym, abyś teraz odszedł i oddał się dalszym medytacjom na temat właściwości dziedziczenia. Jak zdążyłeś zauważyć, dziedziczenie ma swoje wady, ale oprócz niego istnieje jeszcze wiele sposobów na zapewnienie możliwości ponownego wykorzystywania rzeczy...*

Rozmawiając o wzorcach projektowania



Gratulujemy poznania pierwszego wzorca projektowego!

Właśnie zastosowałeś swój pierwszy wzorzec projektowy — wzorzec o nazwie Strategia (ang. *Strategy*). Tak, wszystko się zgadza! Użyłeś tego wzorca do przeprojektowania aplikacji SymulatorKaczki. Dzięki jego zastosowaniu nasz symulator jest gotowy na niemal dowolne zmiany, jakie zechce wprowadzić zarząd firmy po kolejnym spotkaniu z klientami w Vegas.

Dopiero teraz, pomimo iż wcześniej przeprowadziliśmy Cię przez długą drogę implementacji wzorca, przedstawimy jego formalną definicję:

Wzorzec Strategia definiuje rodzinę algorytmów, pakuje je jako osobne klasy i powoduje, że są one w pełni wymienne. Zastosowanie tego wzorca pozwala na to, aby zmiany w implementacji algorytmów przetwarzania były całkowicie niezależne od strony klienta, który z nich korzysta.

Jeżeli chcesz zaimponować kolegom bądź zrobić dobre wrażenie na swoich przełożonych, użyj właśnie tej definicji.



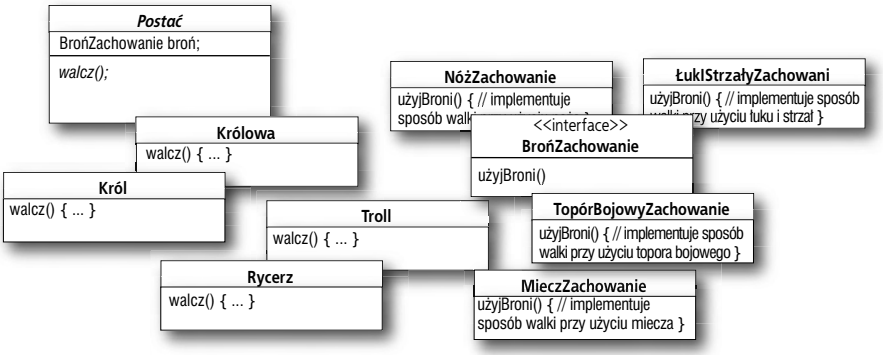
Łamigłówka projektowa

Poniżej znajdziesz całe mnóstwo przypadkowo porozkładanych klas oraz interfejsów odpowiedzialnych za implementację akcji w grze przygodowej. Znajdziesz tam zarówno klasy opisujące bohaterów gry, jak i klasy opisujące zachowania broni, jakich poszczególne postaci mogą używać. Każda postać może używać tylko jednej broni w danej chwili, ale za to może zmieniać broń w dowolnym momencie gry. Twoim zadaniem jest uporządkowanie tego całego kramu….

(Odpowiedzi zostały podane na końcu rozdziału).

Twoje zadanie:

1. Poukładaj poszczególne klasy.
2. Poprawnie zidentyfikuj jedną klasę abstrakcyjną, jeden interfejs oraz osiem klas.
3. Narysuj odpowiednie strzałki pomiędzy klasami.
   a. Takimi strzałkami oznacz dziedziczenie (słowo kluczowe „extends”).
   b. Takimi strzałkami oznacz interfejsy (słowo kluczowe „implements”).
   c. Takimi strzałkami oznacz relacje „MA”.
4. Umieść metodę ustawBroń() we właściwej klasie.




```
ustawBroń(BrońZachowanie w)
{ this.broń = w; }
}
```

Zasłyszane w lokalnym barze szybkiej obsługi...



Jaka jest różnica pomiędzy tymi dwoma zamówieniami? Żadna! Obie dziewczyny zamówiły to samo, z tym tylko, że Alicja użyła przynajmniej dwa razy tyle słów oraz (świadomie bądź też nie) zdawała się testować wytrzymałość gderliwego kelnera z baru szybkiej obsługi.

Co takiego zatem posiada Florentyna, a czego nie ma Alicja? Otóż zarówno ona, jak i kelner z baru szybkiej obsługi używają **wspólnego słownika**. Dzięki temu nie tylko łatwiej jest się jej dogadać z kelnerem, ale również i kelnerowi łatwiej zapamiętać całe zamówienie, ponieważ zapewne ma on wszystkie zestawy obiadowe (czy też, patrząc na to nieco inaczej, wzorce zestawów obiadowych) dobrze ugruntowane w pamięci.

Wzorce projektowe pozwalają Ci używać swego rodzaju wspólnego słownika z innymi projektantami (programistami). Kiedy dobrze poznasz to słownictwo, z pewnością będziesz mógł się łatwiej z nimi komunikować; co więcej, jest również wysoce prawdopodobne, że będziesz zachęcał innych, nieznających wzorców projektowych, do zapoznania się z nimi. Podniesie to również Twoje umiejętności tworzenia architektury aplikacji, ponieważ będziesz w stanie **myśleć o nich bardziej abstrakcyjnie na poziomie *wzorca***, a nie tylko trzeźwo i konkretnie na poziomie *obiektu*.

Zasłyszane w sąsiednim boksie

Tak więc stworzyłem w końcu tę klasę rozgłoszeniową. Śledzi ona wszystkie obiekty, które w danej chwili nasłuchują, i za każdym razem, kiedy otrzymuje nowy zestaw danych, wysyła odpowiedni komunikat do wszystkich odbiorców. Fajną właściwością jest to, że obiekty nasłuchujące mogą w każdej chwili dołączyć do grona odbiorców nadawanych komunikatów lub nawet mogą się same stamtąd usunąć. Cały proces jest naprawdę dynamiczny i elastyczny!

Rysiek





WYTĘŻ UMYSŁ

Czy potrafisz sobie wyobrazić inne wspólne słowniki, które będą wykorzystywane w sytuacjach odmiennych niż tylko projektowanie zorientowane obiektowo czy bar szybkiej obsługi? (Podpowiedź: pomyśl o mechanikach samochodowych, stolarzach, pracownikach kontroli lotów). Jakimi cechami szczególnymi charakteryzuje się przekazywanie informacji w takim żargonie?

Czy myślałeś o tych aspektach projektowania zorientowanego obiektowo, które są komunikowane poprzez nazwę danego wzorca projektowego? Jakie cechy szczególne niesie ze sobą nazwa „wzorzec Strategia"?

Rysiek, czy Ty przypadkiem nie mówiłeś, że używałeś wzorca Obserwator?



Właśnie. Jeżeli komunikujesz się przy użyciu wzorców, inni projektanci będą od razu dokładnie i precyzyjnie orientowali się w szczegółach projektu, który opisujesz. Z drugiej strony, uważaj, aby nie dostać „gorączki wzorcowej"... Łatwo się zorientujesz, że już ją masz, jeżeli zaczniesz używać wzorców do tworzenia aplikacji typu „Hello World!"...

Potęga wspólnego słownika wzorców

Kiedy komunikujesz się przy użyciu wzorców, jest to znacznie więcej niż tylko posługiwanie się ŻARGONEM.

Wspólne słowniki wzorców są naprawdę potężnym narzędziem. Kiedy porozumiewasz się z innym projektantem czy całym zespołem przy użyciu wzorców, tak naprawdę przekazujesz nie tylko samą nazwę wzorca, ale jednocześnie cały zestaw jego cech szczególnych, jego charakterystyki i ograniczenia.

„Do implementacji zachowania kaczek używamy wzorca Strategii". Taka informacja mówi Ci, że zachowania kaczek zostały „zapakowane" (poddane hermetyzacji) w postaci osobnego zestawu klas, który może być łatwo rozbudowywany i modyfikowany w razie potrzeby, a jeżeli to konieczne, nawet w trakcie działania programu.

Wzorce pozwalają Ci powiedzieć więcej przy użyciu mniejszej ilości słów. Jeżeli do opisywania swoich projektów używasz wzorców, inni projektanci szybko i dokładnie zorientują się, o co Ci chodzi.

Na ilu byłeś takich spotkaniach, na których dyskusja bardzo szybko podryfowała w stronę szczegółów implementacji takiego czy innego elementu?

Rozmowa na poziomie wzorców pozwoli Ci poświęcić większą ilość czasu na przemyślenie samego projektu niż na jego implementację. Rozmowa o rozwiązaniach aplikacji przy wykorzystaniu wzorców pozwala na utrzymywanie dyskusji na poziomie projektowania. Dzięki temu unikniesz konieczności zagłębiania się w szczegółowe rozważania na temat implementacji poszczególnych obiektów i klas.

W miarę, jak cały zespół zaczyna wymieniać pomysły i doświadczenia w kategoriach wzorców, zaczynasz tworzyć całą społeczność użytkowników, która czerpie korzyści z posługiwania się wzorcami.

Wspólne słowniki mogą być rodzajem turbodoładowania dla całego Twojego zespołu projektowego. Zespół projektowy, który ma duże doświadczenie w korzystaniu z wzorców, może iść naprzód o wiele szybciej i z mniejszymi szansami na nieporozumienia.

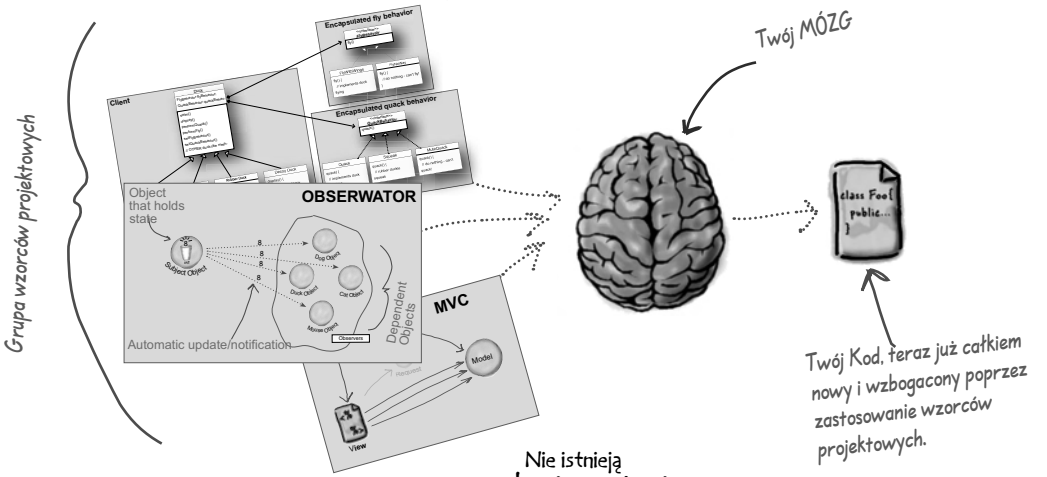
Pomyśl o stworzeniu w swojej firmie grupy, która będzie zajmowała się studiowaniem, analizą i wdrażaniem wzorców — może nawet uda Ci się połączyć przyjemne z pożytecznym i za trudy nauki ktoś Ci jeszcze co nieco zapłaci. . . ;)

Wspólne słowniki zachęcają coraz więcej młodych projektantów do większego wysiłku. Młodzi projektanci często starają się naśladować starszych, bardziej doświadczonych kolegów. Jeżeli starszy projektant korzysta z wzorców projektowych, to zazwyczaj młodsi projektanci są bardziej zmotywowani, aby również się z nimi zapoznać. Spróbuj stworzyć w swojej firmie całe środowisko projektantów posługujących się wzorcami.

W jaki sposób mogę wykorzystywać wzorce projektowe?

Praktycznie wszyscy bez wyjątku korzystamy z różnego rodzaju gotowych produktów, takich jak gotowe biblioteki procedur, szkielety aplikacji i wiele innych. Po prostu, nie zastanawiając się, bierzemy taki produkt, dopisujemy trochę dodatkowego kodu, którego nie znajdziemy w API (skrót od ang. *Application Programming Interface*), kompilujemy do postaci gotowego, naszego programu, a następnie pozostaje już tylko czerpać korzyści z dużej ilości kodu, który tak naprawdę napisał przecież ktoś inny. Przykładowo, pomyśl o API języka Java i tej całej funkcjonalności, jaką Ci ona daje: obsługa sieci, graficznego interfejsu użytkownika, obsługa operacji wejścia-wyjścia itd. Biblioteki procedur i szkielety aplikacji przeszły długą drogę, zanim wyewoluowały do obecnego modelu, w którym możesz wybrać odpowiednie komponenty i po prostu dołączyć je do aplikacji. Ale… mimo całej swojej doskonałości API nie pomoże nam w utworzeniu takiej struktury naszych własnych aplikacji, która będzie przejrzysta, łatwa w serwisowaniu i elastyczna. I właśnie w tym miejscu przychodzą nam z pomocą wzorce projektowe.

Wzorce projektowe nie są od razu implementowane w kodzie aplikacji — najpierw muszą odpowiednio zagnieździć się w Twoim MÓZGU. Dopiero po tym, jak odpowiednia wiedza na temat praktycznego zastosowania wzorców znajdzie się tam na swoim miejscu, możesz spróbować użyć jej podczas tworzenia nowych projektów bądź reorganizacji kodu istniejących aplikacji (za co warto się zabrać w sytuacji, gdy dojdziesz do wniosku, że schemat relacji w istniejącym kodzie zaczyna nieubłaganie przypominać solidną porcję spaghetti).



Nie istnieją głupie pytania

P: Jeżeli wzorce projektowe są tak wspaniałe, to dlaczego ktoś po prostu nie stworzy ich biblioteki, tak aby inni już nie musieli tego robić?

O: Wzorce projektowe znajdują się na nieco wyższym poziomie abstrakcji niż zwykłe biblioteki procedur. Mówią one nam, w jaki sposób możemy stworzyć strukturę klas i obiektów, która umożliwi nam rozwiązanie określonych problemów. Natomiast naszym zadaniem jest adaptacja takiego projektu do potrzeb konkretnej aplikacji.

P: Czy biblioteki i szkielety aplikacji też są wzorcami projektowymi?

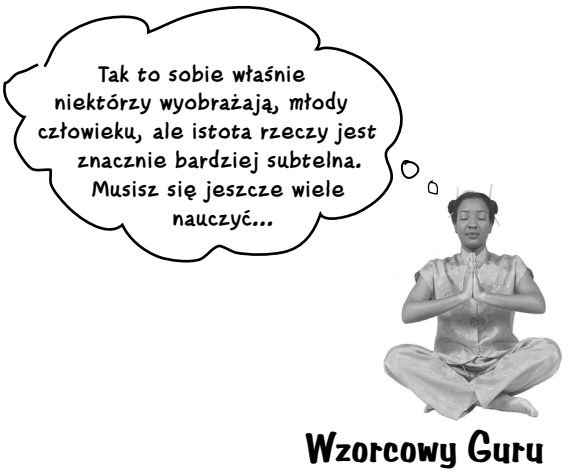
O: Biblioteki i szkielety aplikacji nie są wzorcami projektowymi; one zapewniają tylko specyficzne rozwiązania implementacji określonych elementów, które możemy dołączyć bądź w inny sposób wykorzystać w naszym kodzie. Z drugiej jednak strony, trzeba przyznać, że niektóre biblioteki itp. wykorzystują wzorce projektowe w swoich własnych implementacjach. To znakomite rozwiązanie, ponieważ jeżeli dobrze rozumiesz zasady funkcjonowania wzorców projektowych, tym szybciej przyswoisz sobie znajomość danego API, które w oparciu o takie wzorce powstało.

P: Czyli, krótko mówiąc, nie istnieją żadne biblioteki wzorców projektowych?

O: Nie, ale nieco później omawiać będziemy katalogi wzorców projektowych obejmujących zestawienia wzorców, których możesz użyć podczas tworzenia własnych aplikacji.



Projektant-sceptyk



Wzorcowy Guru

Projektant: No dobrze, ale czy… hmmm, czy to wszystko nie jest jednak po prostu programowaniem zorientowanym obiektowo; to znaczy, chodzi mi o to, że jeżeli postępuję zgodnie z regułami hermetyzacji klas i znam zagadnienia dotyczące abstrakcyjności oraz mechanizmów dziedziczenia i polimorfizmu, to czy naprawdę muszę myśleć o wzorcach projektowych? Czy to nie dlatego byłem na tych wszystkich szkoleniach i kursach programowania zorientowanego obiektowo? Myślę, że wzorce projektowe są użyteczne dla tych, którzy nie znają dobrze zasad programowania zorientowanego obiektowo.

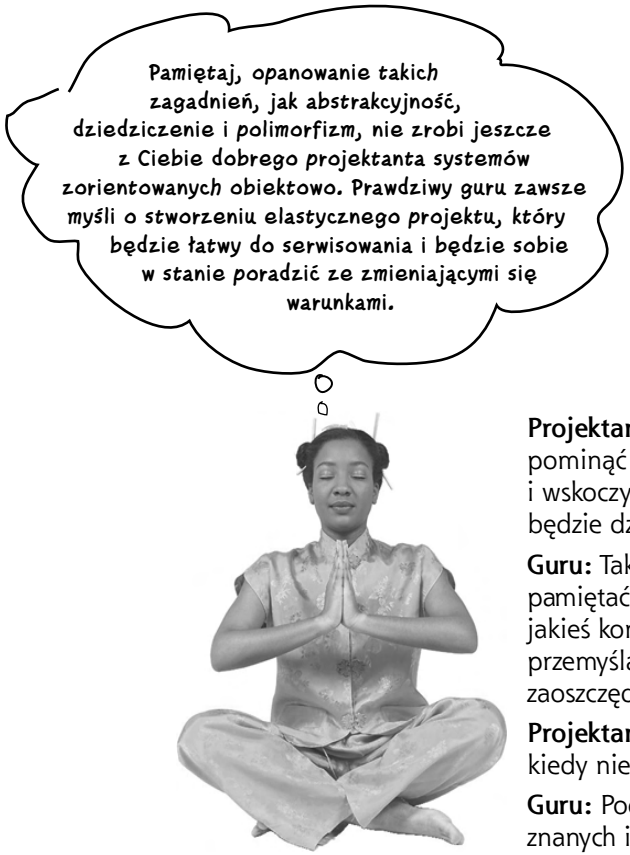
Guru: Taaak, to jest właśnie jeden z mitów o projektowaniu zorientowanym obiektowo: że dzięki dobrej znajomości podstaw takiego programowania będziemy automatycznie specjalistami w tworzeniu systemów elastycznych, nadających się do wielokrotnego wykorzystania oraz łatwych w serwisowaniu i modyfikacji.

Projektant: A nie?

Guru: Nie. Jak się okazuje, zasady tworzenia systemów zorientowanych obiektowo, które posiadają wymienione wcześniej cechy, nie zawsze są takie proste i oczywiste i często ich odkrycie bywało okupione ciężką i wytężoną pracą.

Projektant: Myślę, że zaczynam łapać, o co chodzi. Po pewnym czasie te wszystkie, czasami niezbyt oczywiste, zasady tworzenia systemów zorientowanych obiektowo zostały zebrane…

Guru: … tak, w postaci zestawu wzorców nazywanych powszechnie wzorcami projektowymi.



Projektant: Czy zatem, znając odpowiednie wzorce, mogę pominąć tę całą ciężką pracę nad początkowym projektem i wskoczyć od razu do wdrażania projektu, który zawsze będzie działał?

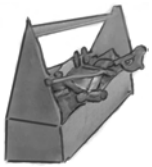
Guru: Tak, do pewnego stopnia, ale powinieneś jednak pamiętać, że projektowanie jest sztuką. Zawsze będą istniały jakieś kompromisy. Jeżeli jednak będziesz korzystał z dobrze przemyślanych i przetestowanych wzorców projektowych, zaoszczędzisz mnóstwo czasu.

Projektant: A co powinienem zrobić w sytuacji, kiedy nie uda mi się znaleźć odpowiedniego wzorca?

Guru: Pod „przykryciem" wzorców znajdziesz szereg dobrze znanych i rozumianych reguł projektowania zorientowanego obiektowo, a znając je, będziesz na pewno w stanie poradzić sobie w takiej sytuacji, kiedy gotowy wzorzec nie jest dostępny.

Projektant: Reguły? Masz na myśli te wszystkie abstrakcyjności, hermetyzacje klas oraz…

Guru: Tak! Jednym z sekretów, które pozwalają na stworzenie łatwych w serwisowaniu systemów zorientowanych obiektowo, jest myślenie z wyprzedzeniem, jak mogą one zmieniać się w czasie — właśnie reguły są dopasowane do takich sytuacji.



Twoja skrzynka narzędziowa

Prawie skończyłeś już lekturę pierwszego rozdziału naszej książki! Do tej pory zdążyłeś umieścić kilka narzędzi w swojej skrzynce „zorientowanej obiektowo"; zanim wyruszymy na podbój rozdziału 2., zróbmy dla przypomnienia ich listę.

Podstawy programowania obiektowego

Abstrakcyjność

Hermetyzacja

Polimorfizm

Dziedziczenie

Zakładamy, że już znasz podstawy projektowania zorientowanego obiektowo i zasady działania polimorfizmu klas, dziedziczenia oraz hermetyzacji. Jeżeli jednak Twoja wiedza wymaga pewnego odświeżenia, mimo wszystko powinieneś teraz wyciągnąć z półki książkę „Java. Rusz głową! Wydanie II", przypomnieć sobie co nieco i dopiero wtedy ponownie wrócić do tego rozdziału.

Reguły programowania obiektowego

Poddawaj hermetyzacji to, co się zmienia.

Przedkładaj kompozycję nad dziedziczenie.

Skoncentruj się na tworzeniu interfejsów, a nie implementacji.

W najbliższym czasie przyjrzymy się tym zagadnieniom nieco dokładniej; dołożymy również do tej listy kilka nowych pozycji.

Wzorce programowania obiektowego

Strategia - definiuje rodziny algorytmów, dokonuje ich hermetyzacji i powoduje, że stają się one wymienne. Wzorzec Strategy pozwala na modyfikację danego algorytmu niezależnie od klienta, który tego algorytmu używa.

W czasie pracy z książką myśl o tym, w jaki sposób wzorce są powiązane z podstawami i regułami projektowania zorientowanego obiektowo.

Jeden z głowy, ale jeszcze dużo przed nami!



CELNE SPOSTRZEŻENIA

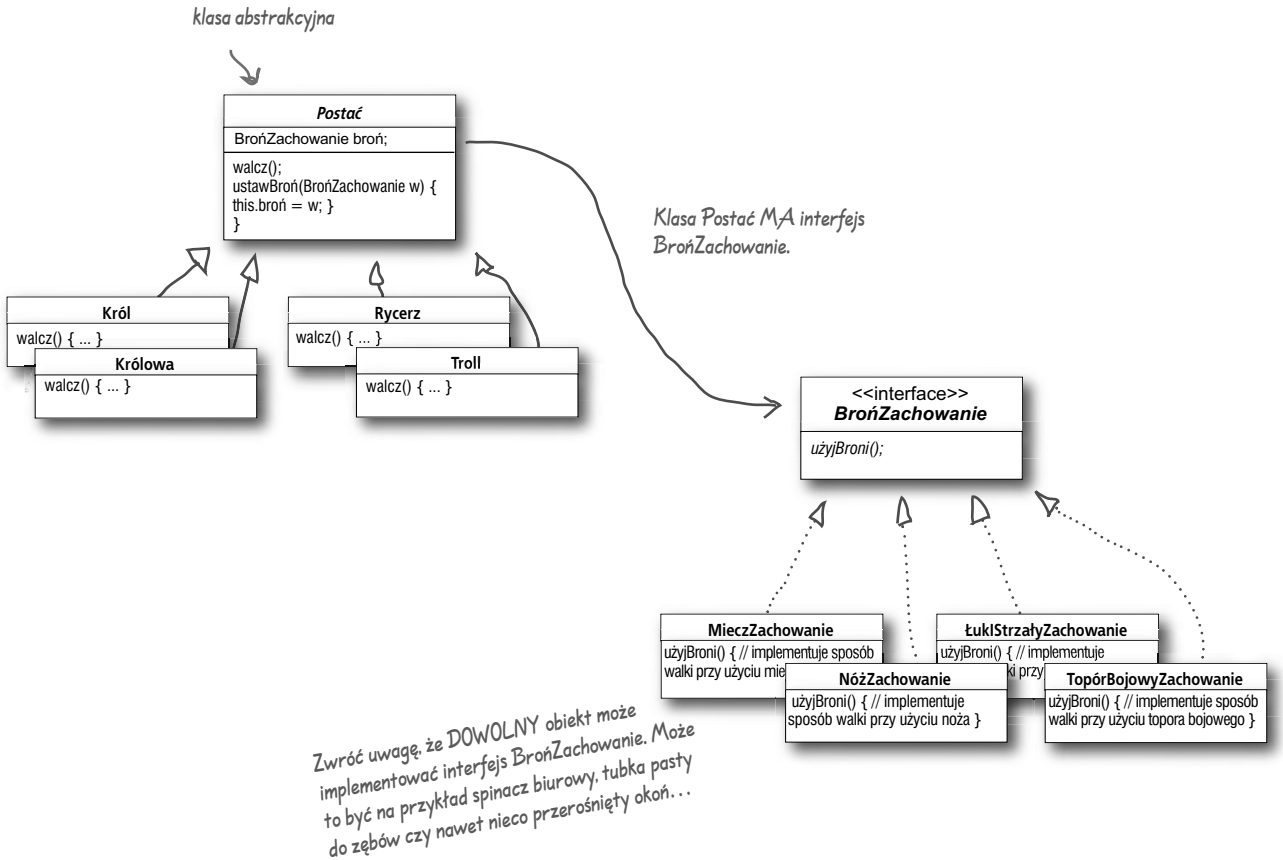
- Znajomość podstaw projektowania zorientowanego obiektowo nie zrobi automatycznie z Ciebie dobrego projektanta.
- Dobre projekty zorientowane obiektowo powinny nadawać się do wielokrotnego użytku, być proste do rozbudowy oraz łatwe w serwisowaniu i modyfikacji.
- Wzorce wskazują sposoby tworzenia całych systemów posiadających cechy charakterystyczne dobrych projektów zorientowanych obiektowo.
- Wzorce projektowe znakomicie sprawdziły się w wielu rzeczywistych aplikacjach systemów zorientowanych obiektowo.
- Wzorce nie udostępniają Ci gotowego kodu, a jedynie ogólne sposoby rozwiązywania problemów pojawiających się w fazie projektowania. Sam musisz zaadaptować je do konkretnych aplikacji.
- Wzorce nie są *wynalazkami*, wzorce się po prostu *odkrywa*.
- Większość wzorców i reguł odnosi się do sytuacji, w których w danym oprogramowaniu muszą zostać dokonane określone *zmiany*.
- Większość wzorców umożliwia modyfikowanie pewnych fragmentów systemu całkowicie niezależnie od pozostałych elementów systemu.
- Bardzo często próbujemy zidentyfikować elementy, które się w danym systemie zmieniają, a następnie próbujemy dokonać ich hermetyzacji.
- Wzorce zapewniają rodzaj wspólnego, jednolitego języka, który może maksymalizować efektywność komunikacji pomiędzy poszczególnymi członkami zespołu projektowego.



Rozwiązanie łamigłówki projektowej

Klasa Postać jest klasą abstrakcyjną dla wszystkich pozostałych klas (postaci), takich jak Król, Królowa, Rycerz oraz Troll. Z kolei Broń jest interfejsem zaimplementowanym dla wszystkich klas opisujących poszczególne rodzaje broni. Jak łatwo się zorientować, wszystkie postaci oraz poszczególne rodzaje broni zostały zaimplementowane jako osobne klasy.

Aby zmienić rodzaj używanej broni, każda z postaci musi wywołać metodę ustawBroń() zdefiniowaną w klasie nadrzędnej (w superklasie) Postać. Podczas walki poszczególne postaci wywołują metodę użyjBroni() i dzięki temu mogą wykorzystywać aktualnie wybraną broń do zadawania śmiertelnych ciosów innym postaciom.





Rozwiązania



Zaostrz ołówek

Które z poniżej wymienionych negatywnych zjawisk będzie rezultatem zastosowania mechanizmu dziedziczenia do tworzenia poszczególnych zachowań Kaczki? (Zaznacz wszystkie, które pasują).

- ☑ A. Kod programu jest duplikowany w kolejnych podklasach.
- ☑ B. Trudno zmienić zachowania metod podczas działania programu.
- ☐ C. Nie możemy spowodować, że kaczki zaczną tańczyć.
- ☑ D. Trudno zebrać informacje o zachowaniach wszystkich kaczek.
- ☐ E. Kaczki nie mogą latać i kwakać w tym samym czasie.
- ☑ F. Wprowadzane zmiany mogą mieć niezamierzony wpływ na inne kaczki.



Zaostrz ołówek

Wypisz kilka powodów, dla których musiałeś dokonywać modyfikacji kodu w Twoich aplikacjach. Twoja lista może być zupełnie inna, ale dla porównania umieściliśmy kilka naszych przykładów. Wyglądają znajomo?

Klienci lub użytkownicy zdecydowali, że chcą otrzymać coś zupełnie nowego bądź też rozszerzyć funkcjonalność istniejącego oprogramowania.

Zarząd firmy podjął decyzję, że będziemy korzystać z zupełnie nowej bazy danych innego producenta, a same dane zostaną dostarczone przez dostawcę, który oczywiście wykorzystuje zupełnie inny ich format. Auuć!

Taaak... nastąpiły zmiany w technologii i, niestety, musimy dokonać aktualizacji kodu tak, aby można było skorzystać z nowych protokołów.

Podczas budowania naszego systemu zdobyliśmy tak wiele cennych doświadczeń, że chcielibyśmy teraz cofnąć się do początku, rozpocząć wszystko od nowa i zrobić to trochę lepiej.

Jak sprawić, by Twoje obiekty były zawsze dobrze poinformowane



Nie przegap okazji, kiedy dzieje się coś naprawdę ciekawego! Przedstawimy Ci wzorzec, który potrafi poinformować inne obiekty o tym, że wydarzyło się coś, czym powinny się zająć. Co ciekawe, obiekty mogą nawet samodzielnie decydować w czasie działania programu o tym, czy chcą być informowane o takich wydarzeniach. Wzorzec Obserwator jest jednym z najczęściej wykorzystywanych wzorców w pakiecie JDK (ang. *Java Development Kit*), a co najważniejsze — jest wręcz niewiarygodnie użyteczny. W niniejszym rozdziale rzucimy również okiem na relacje typu jeden-do-wielu oraz tzw. luźne związki (tak, to prawda, napisaliśmy „luźne związki"). Korzystając z wzorca Obserwator, z pewnością odmienisz swoje życie.

Gratulacje!

Wasz zespół właśnie wygrał kontrakt na budowę Pogodynki 2, najnowszej generacji internetowej stacji sprawdzającej i podającej aktualną pogodę.



Pogodynka sp. z o.o.
ul. Deszczowa 25
00-000 Śniegowo

Oświadczenie o współpracy

Chcielibyśmy złożyć serdeczne gratulacje z okazji wygrania przez Państwa firmę kontraktu na budowę naszej najnowszej generacji internetowej stacji meteorologicznej!

Stacja meteorologiczna oparta będzie na naszym znakomitym, opatentowanym obiekcie DanePogodowe, który automatycznie śledzi bieżące warunki pogodowe (temperatura otoczenia, wilgotność oraz ciśnienie atmosferyczne). Chcielibyśmy, aby Wasza firma utworzyła aplikację, która początkowo będzie miała za zadanie wyświetlać na ekranie trzy główne elementy: informację o bieżących warunkach pogodowych, dane statystyczne o pogodzie oraz prostą prognozę pogody, wszystkie aktualizowane w czasie rzeczywistym w chwili, gdy obiekt DanePogodowe otrzyma najnowsze odczyty z urządzeń pomiarowych.

Musimy jednak pamiętać o tym, że jest to perspektywiczna wersja stacji meteorologicznej. Nasza firma chce w najbliższej przyszłości wypuścić na rynek wersję API do stacji Pogodynka 2, która umożliwi innym projektantom tworzenie własnych sposobów wyświetlania informacji o pogodzie i łatwe dołączanie takich modułów do głównego oprogramowania stacji. Chcielibyśmy, aby Wasza firma dostarczyła nam pierwszą wersję takiego API.

Uważamy, że nasza firma realizuje znakomity biznesplan: po podpisaniu umowy z klientem mamy zamiar obciążać go dodatkowymi kosztami za każdy panel wyświetlający informacje z naszej stacji. A teraz najlepsza część: oczekujemy, że akceptowaną przez Was formą płatności będą opcje na zakup akcji naszej wspaniałej firmy!

Z niecierpliwością oczekujemy na projekt stacji oraz wersję alfa aplikacji sterującej.

Z poważaniem,

Janusz Monsun

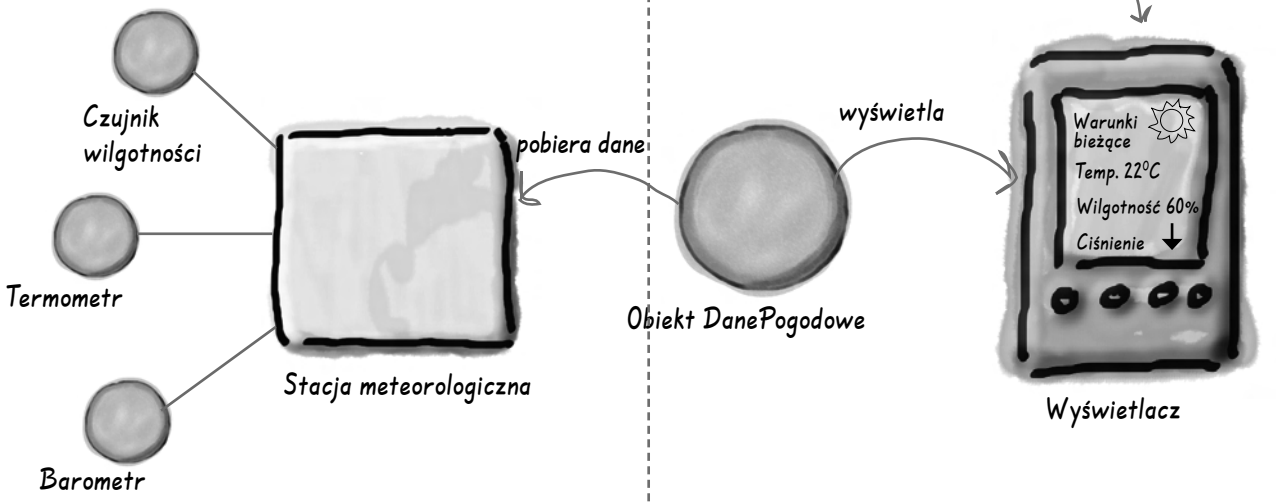
Janusz Monsun, Prezes

P.S. Pliki źródłowe zawierające kod naszego opatentowanego obiektu DanePogodowe wyślemy do Państwa kurierem jeszcze dziś w nocy.

Ogólne spojrzenie na aplikację sprawdzającą warunki pogodowe

Trzema głównymi graczami w naszym systemie są: stacja meteorologiczna (fizyczne urządzenie zbierające dane z czujników temperatury, ciśnienia i wilgotności), obiekt DanePogodowe (który zapewnia śledzenie danych nadchodzących z tej stacji oraz aktualizuje informacje wyświetlane na ekranie) oraz sam wyświetlacz, którego zadaniem jest zobrazowanie w postaci czytelnej dla użytkownika informacji o bieżących warunkach pogodowych.

„Warunki bieżące" to jeden z dostępnych trybów wyświetlania informacji. Użytkownik może również zażyczyć sobie wyświetlenia danych statystycznych o pogodzie dla naszego regionu (tryb „Statystyka") lub też prostej prognozy pogody (tryb „Prognoza").



Elementy dostarczane przez firmę Pogodynka sp. z o.o.

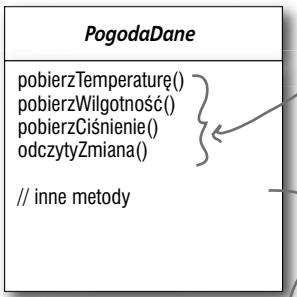
Elementy, które musimy zaimplementować

Obiekt DanePogodowe „wie", w jaki sposób należy komunikować się z fizyczną stacją meteorologiczną tak, aby otrzymywać aktualne odczyty danych. Po otrzymaniu danych obiekt aktualizuje wyświetlane informacje dla trzech głównych trybów pracy: „Warunki bieżące" (tryb ten pokazuje informacje na temat temperatury otoczenia, wilgotności oraz ciśnienia atmosferycznego), „Statystyka" oraz „Prognoza".

Jeżeli zdecydujemy się na zaakceptowanie tego kontraktu, naszym zadaniem będzie stworzenie aplikacji, która będzie wykorzystywała obiekt DanePogodowe do aktualizacji informacji wyświetlanych w trzech głównych trybach pracy.

Rozpakowywanie klasy DanePogodowe

Jak zostało obiecane, następnego dnia rano kurier dostarczył pliki źródłowe klasy DanePogodowe. Kiedy rzuciliśmy okiem na jej kod, wszystko stało się jasne:



Te trzy metody zwracają aktualne odczyty z czujników, odpowiednio: temperatury, wilgotności oraz ciśnienia atmosferycznego.

Nie dbamy o to, JAK ustawiane są wartości tych zmiennych; obiekt PogodaDane „wie", w jaki sposób otrzymać zaktualizowane informacje ze stacji meteorologicznej.

Jak widać, projektanci klasy DanePogodowe zostawili nam wskazówkę na temat tego, co musimy tutaj dołożyć...

```
/*
* Ta metoda jest wywoływana
* za każdym razem, kiedy zostaną
 zaktualizowane
* odczyty z czujników pogody
*
*/
 public void odczytyZmiana() {
     // tutaj umieść odpowiedni kod
 }
```

plik DanePogodowe.java

Pamiętaj, że tryb „Warunki bieżące" jest tylko JEDNYM z trzech dostępnych trybów wyświetlania informacji.



Wyświetlacz

Nasze zadanie polega na implementacji metody odczytyZmiana(), tak aby była ona w stanie automatycznie aktualizować informacje wyświetlane w trybach „Warunki bieżące", „Statystyka" oraz „Prognoza".

O czym wiemy do tej pory?

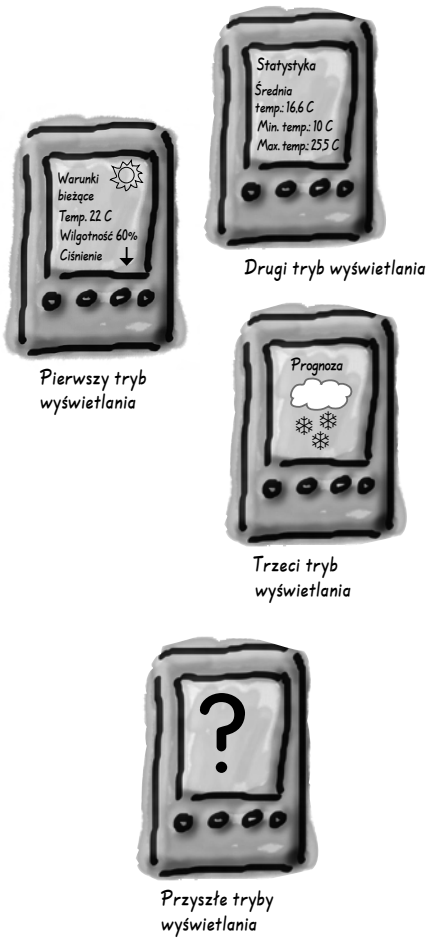


Specyfikacja nadesłana z firmy Pogodynka sp. z o.o. nie była do końca jasna, więc musieliśmy się zorientować, co tak naprawdę mamy zrobić. Podsumujmy zatem, czego dowiedzieliśmy się dotychczas:

- Klasa DanePogodowe posiada odpowiednie metody, umożliwiające pobieranie trzech mierzonych wartości (temperatury otoczenia, wilgotności oraz ciśnienia atmosferycznego) z czujników fizycznych.
- Metoda odczytyZmiana() jest wywoływana za każdym razem, kiedy pojawiają się nowe dane z czujników fizycznych (nie wiemy, a w zasadzie nawet nie dbamy o to, w jaki sposób ta metoda jest wywołana; wiemy tylko, że tak *po prostu działa*).
- Musimy zaimplementować obsługę trzech trybów wyświetlania wykorzystujących dane o pogodzie: trybu *Warunki bieżące*, trybu *Statystyka* oraz trybu *Prognoza*. Dane wyświetlane w poszczególnych trybach muszą być na bieżąco aktualizowane za każdym razem, kiedy obiekt DanePogodowe zyskuje odczyty nowych pomiarów.
- Cały system musi być łatwy do rozbudowy — inni projektanci mogą tworzyć swoje własne, nowe tryby wyświetlania danych, a użytkownicy mogą dodawać do aplikacji bądź usuwać z niej dowolną ilość trybów wyświetlania. W chwili obecnej znamy tylko *trzy* początkowe tryby wyświetlania danych („Warunki bieżące", „Statystyka" oraz „Prognoza").

```
pobierzTemperaturę()
pobierzWilgotność()
pobierzCiśnienie()


odczytyZmiana()
```



Pierwszy, pozorny sukces w walce ze stacją meteorologiczną

Poniżej przedstawiamy pierwszy wariant implementacji naszego systemu — skorzystamy tu z porady, jaką nam przekazali programiści z firmy Pogodynka sp. z o.o., i umieścimy nasz kod wewnątrz metody odczytyZmiana():

```
public class DanePogodowe {
    // deklaracje zmiennych obiektowych
    public void odczytyZmiana() {

        float temp = pobierzTemperaturę();
        float wilgotność = pobierzWilgotność();
        float pressure = pobierzCiśnienie();

        warunkiBieżąceWyświetl.aktualizacja(temp, wilgotność, ciśnienie);
        statystykaWyświetl.aktualizacja(temp, wilgotność, ciśnienie);
        prognozaWyświetl.aktualizacja(temp, wilgotność, ciśnienie);
    }
    // w tym miejscu można wstawić inne metody obiektu PogodaDane
}
```

Pobierz najbardziej aktualne odczyty, wywołując odpowiednie metody umożliwiające pobieranie trzech mierzonych wartości z czujników fizycznych (metody zostały już wcześniej zaimplementowane).

Aktualizacja wyświetlanych informacji…

Wywołuje każdy wyświetlany element i przekazuje mu do wyświetlania najbardziej aktualne wartości odczytów z odpowiedniego czujnika.



Zaostrz ołówek

Opierając się na pierwszej implementacji naszego systemu, określ, które z wymienionych zdarzeń są prawdziwe. (Zaznacz wszystkie poprawne odpowiedzi).

A. ❑ Tworzymy poszczególne implementacje, a nie interfejsy.

B. ❑ Dodanie nowego trybu wyświetlania będzie każdorazowo wymuszało modyfikację kodu programu.

C. ❑ Nie mamy żadnych możliwości dodawania (lub usuwania) wybranych trybów wyświetlania podczas działania programu.

D. ❑ Poszczególne wyświetlane elementy nie posiadają wspólnego interfejsu.

E. ❑ Nie dokonaliśmy hermetyzacji tych elementów aplikacji, które się zmieniają.

F. ❑ Naruszyliśmy hermetyzację klasy DanePogodowe.

Co jest nie tak z naszą implementacją?

Powróć na chwilę myślami do tych wszystkich pojęć i reguł, o których wspominaliśmy w rozdziale 1...

```
public class PogodaDane {
    // deklaracje zmiennych obiektowych
    public void odczytyZmiana() {
        float temp = pobierzTemperaturę();
        float wilgotność = pobierzWilgotność();
        float pressure = pobierzCiśnienie();

        warunkiBieżąceWyświetl.aktualizacja(temp, wilgotność, ciśnienie);
        statystykaWyświetl.aktualizacja(temp, wilgotność, ciśnienie);
        prognozaWyświetl.aktualizacja(temp, wilgotność, ciśnienie);
    }
```

Obszar zmian, powinniśmy zatem dokonać jego hermetyzacji.

W przypadku tworzenia poszczególnych implementacji nie mamy żadnych możliwości dodawania ani usuwania wyświetlanych elementów bez dokonywania modyfikacji w kodzie programu.

Wygląda na to, że przynajmniej staramy się używać wspólnego, jednolitego interfejsu dla wyświetlanych elementów... Wszystkie posiadają metodę aktualizacja(), której argumentami są wartości temperatury otoczenia, wilgotności oraz ciśnienia atmosferycznego.



Rzucimy teraz okiem na wzorzec Obserwator, a następnie powrócimy do naszej aplikacji i pokażemy, w jaki sposób można zastosować go do aplikacji obsługującej stację meteorologiczną.

Spotkanie z wzorcem Obserwator

Zapewne wiesz, w jaki sposób działa prenumerata gazet i czasopism:

1. Wydawca nowej gazety lub czasopisma rozpoczyna swoją działalność i na rynku pojawia się nowy tytuł prasowy.

2. Zamawiasz prenumeratę u danego wydawcy i od tego momentu za każdym razem, kiedy pojawia się nowe wydanie, zostaje Ci ono dostarczone. Całość działa tak długo, jak długo pozostajesz prenumeratorem danego tytułu prasowego.

3. Jeżeli nie chcesz więcej otrzymywać danej gazety lub czasopisma, po prostu wypowiadasz prenumeratę i od tego momentu nowe wydania przestają do Ciebie docierać.

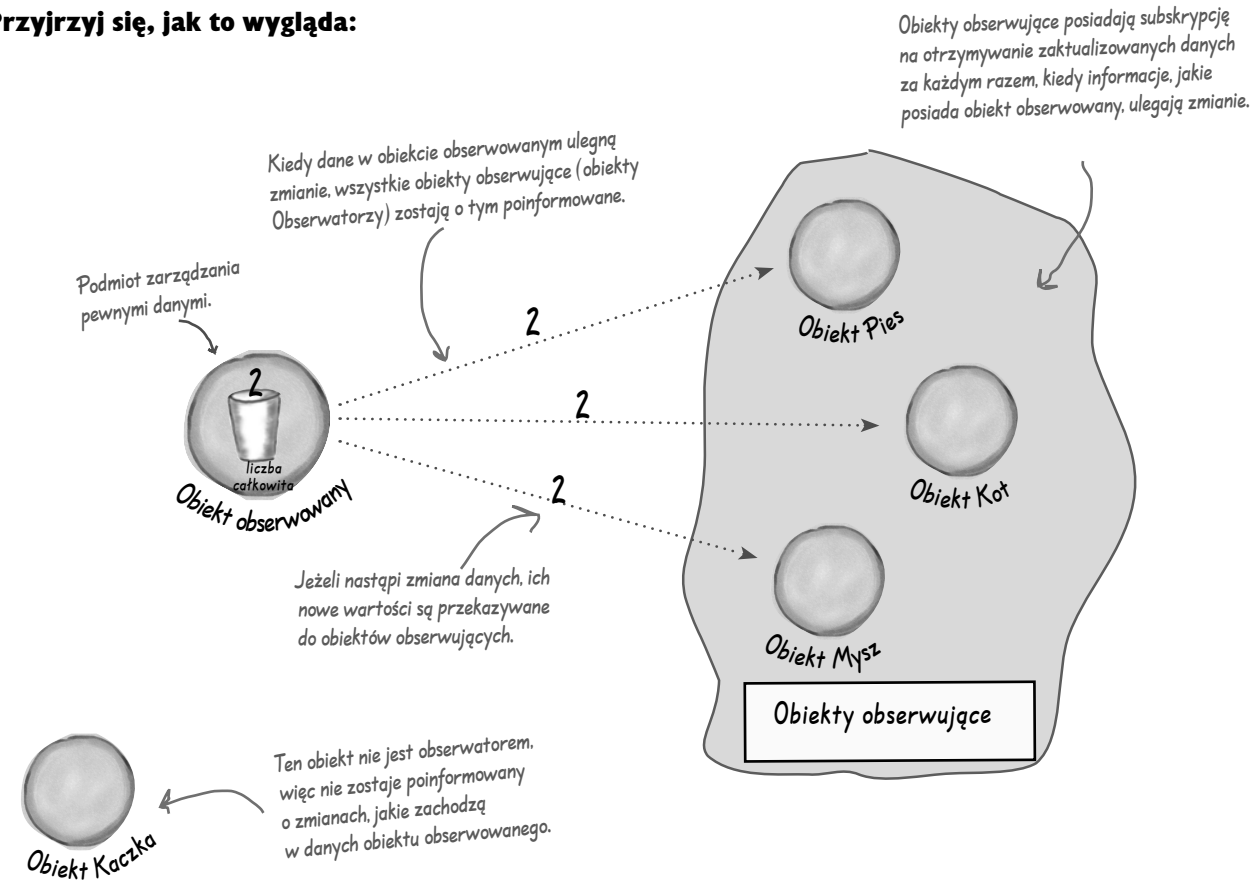
4. Jak długo dany wydawca (czy też dany tytuł prasowy) istnieje na rynku, tak długo różni ludzie, firmy, hotele, linie lotnicze itp. będą ustawicznie dokonywały nowych prenumerat i wypowiadały stare prenumeraty.



Wydawca + Prenumerator = wzorzec Obserwator

Jeżeli rozumiesz zasady prenumerowania gazet i czasopism, oznacza to, że całkiem dobrze rozumiesz zasady funkcjonowania wzorca Obserwator, z tym tylko, że we wzorcu zamiast wydawcy występuje PODMIOT (obiekt obserwowany), a zamiast prenumeratorów występują OBIEKTY OBSERWATORZY (obiekty obserwujące).

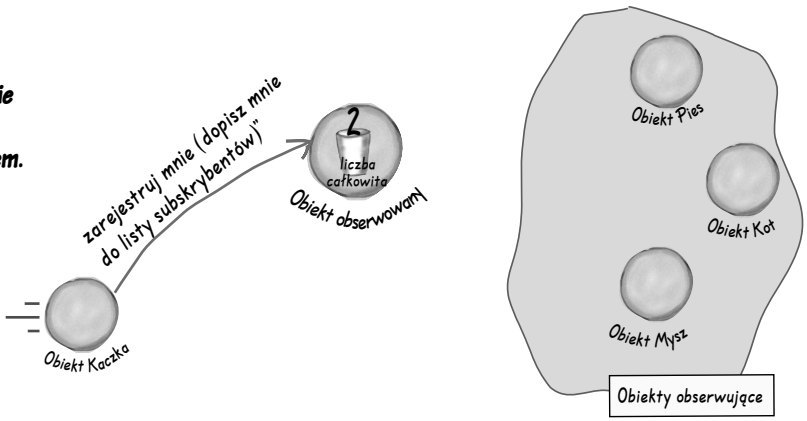
Przyjrzyj się, jak to wygląda:



Dzień z życia wzorca Obserwator

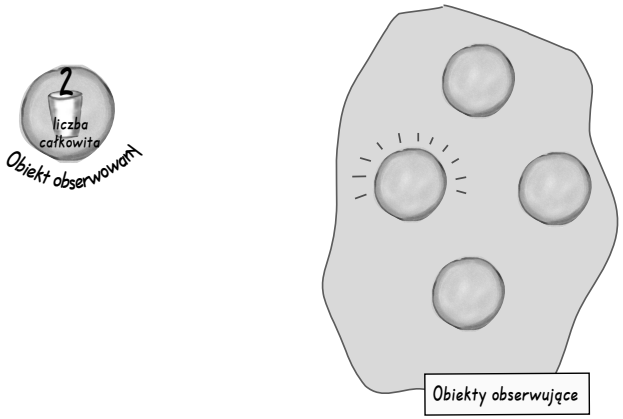
Obiekt Kaczka pojawia się i następnie informuje obiekt obserwowany, że chciałby zostać jego obserwatorem.

Obiekt Kaczka tak naprawdę chce tylko jednego: danych, jakie obiekt obserwowany wysyła za każdym razem, kiedy zmienia się jego stan.



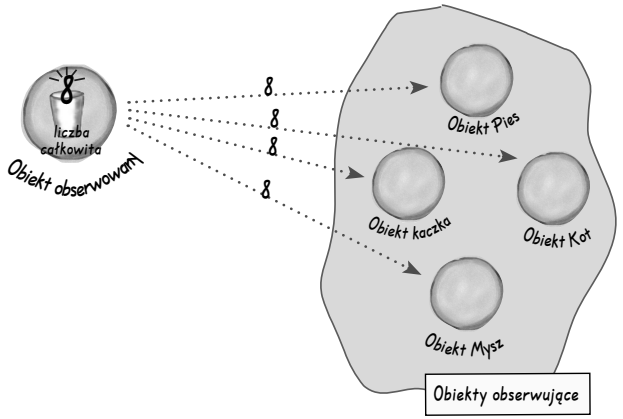
Obiekt Kaczka jest teraz oficjalnie uznanym obserwatorem.

Kaczka jest podekscytowana... znajduje się już na liście i z niecierpliwością oczekuje na nadejście pierwszych informacji od obiektu obserwowanego.



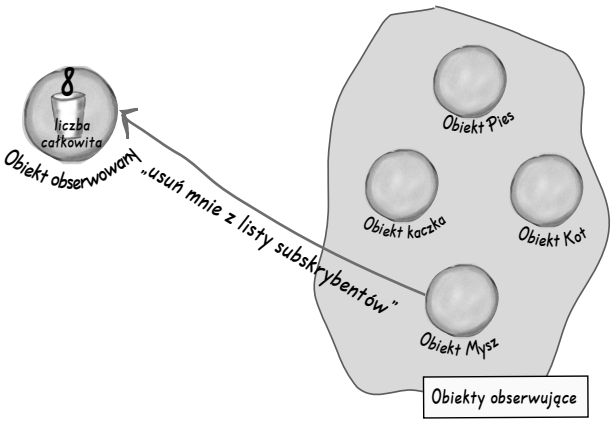
Obiekt obserwowany otrzymuje nowe dane!

W takiej sytuacji obiekt Kaczka oraz pozostałe obiekty obserwujące otrzymują powiadomienie, że obiekt obserwowany zmienił stan.



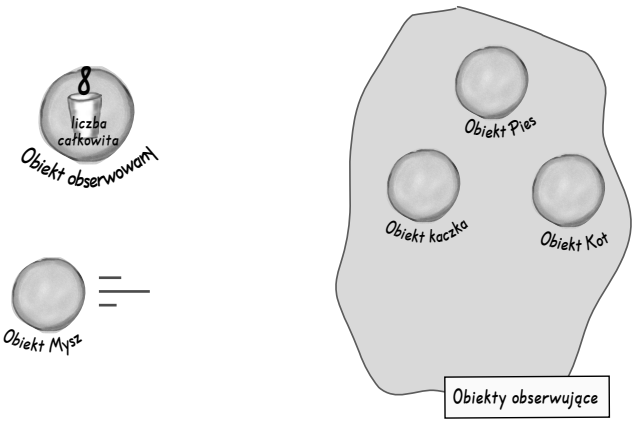
Obiekt Mysz prosi o usunięcie z listy obserwatorów.

Mysz otrzymywała aktualizacje danych z obiektu obserwowanego od wieków — nic dziwnego zatem, że w końcu jej się znudziło i postanowiła zrezygnować z bycia obserwatorem.



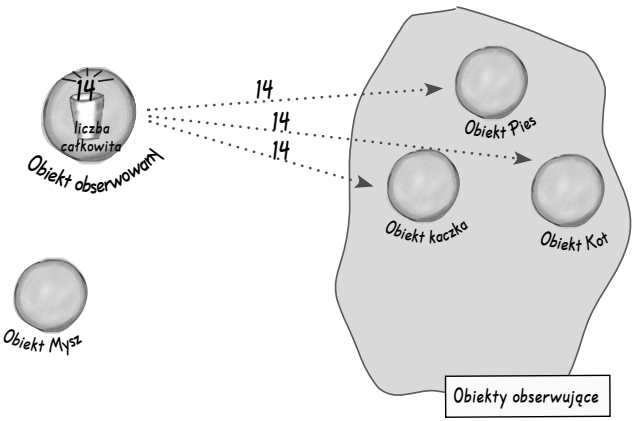
Myszy już tu nie ma!

Obiekt obserwowany potwierdza przyjęcia żądania Myszy i usuwa ją z listy obserwatorów.



Obiekt obserwowany ponownie zmienił stan.

Wszystkie obiekty obserwujące zostają o tym fakcie powiadomione; wyjątkiem jest tutaj Mysz, która już nie znajduje się na liście subskrybentów. Nie mów tego nikomu, ale tak naprawdę nasza Mysz w skrytości ducha tęskni za bieżącymi komunikatami… być może pewnego dnia wystąpi ona z prośbą o ponowne zaliczenie jej w poczet obserwatorów.





Pięciominutowe przedstawienie — obserwowany kontra obserwujący

W dzisiejszym skeczu dwóch zakręconych projektantów oprogramowania spotyka prawdziwego „łowcę talentów”...

Mówi Robert. Szukam nowego miejsca pracy w charakterze projektanta systemów opartych na Javie. Mam pięcioletnie doświadczenie oraz...



1

Projektant nr 1

Hmmm, oczywiście, Ty i kto tylko jeszcze będzie chciał, mój chłopcze. Właśnie umieszczam Cię na mojej liście projektantów systemów Java. Nie dzwoń do mnie, to ja do Ciebie oddzwonię!



2

Łowca talentów (Podmiot)

Cześć, mam na imię Joanna, napisałam już wiele systemów wykorzystujących EJB. Jestem zainteresowana dowolnym stanowiskiem pracy w dziale zajmującym się projektowaniem systemów Java.



3

Projektant nr 2

Umieszczę Twoje nazwisko na mojej liście. Dzięki temu będziesz informowana na bieżąco, razem z pozostałymi obserwatorami.



4

Obiekt obserwowany (Podmiot)

5 *W międzyczasie życie Roberta i Joanny biegło swoim zwykłym torem; jeżeli pojawiała się jakaś oferta pracy dla projektantów Javy, otrzymywali informacje — krótko mówiąc, zostali obserwatorami.*



6 **Obiekt obserwowany**



7 **Obserwator**



Obserwator

Joanna znajduje nowego pracodawcę.



8 **Obserwator**



9 **Obiekt obserwowany**

Dwa tygodnie później



Joanna wielce sobie ceni swoje obecne życie i nie musi już odgrywać roli obserwatora. Jest również bardzo zadowolona, że nowa firma zaproponowała jej bardzo dogodne warunki finansowe, tym lepsze, że nie musiała dodatkowo opłacić usług łowcy talentów.

Ale co się stało z naszym drogim Robertem? Słyszeliśmy, że pobił łowcę talentów na jego własnym podwórku. Jest teraz nie tylko obserwatorem, ale również posiada swoją własną listę subskrybentów (obserwatorów), których powiadamia o pojawieniu się nowych ofert na rynku pracy. Krótko mówiąc, Robert jest teraz zarówno obserwującym, jak i obserwowanym.



Definicja wzorca Obserwator

Jeżeli będziesz próbował w jakiś sposób zobrazować sobie wzorzec Obserwator, to zapewne szybko dojdziesz do wniosku, że przedstawiony wcześniej przykład z wydawnictwem i prenumeratą periodyków prasowych stanowi jego znakomitą wizualizację.

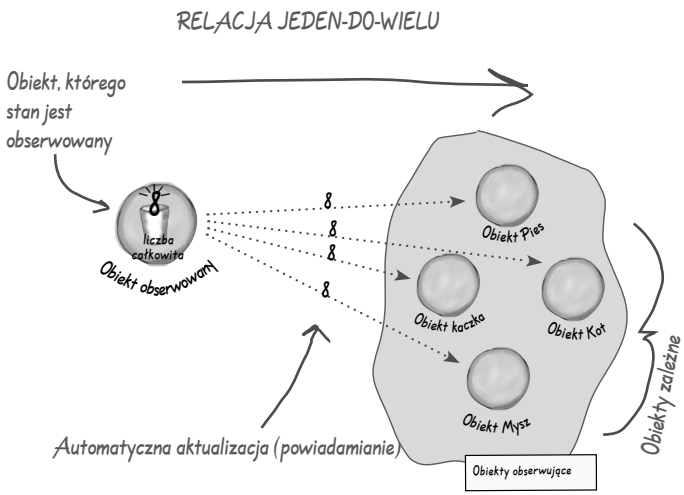
Jednak w świecie rzeczywistym wzorzec Obserwator jest zazwyczaj definiowany następująco:

> **Wzorzec Obserwator** definiuje pomiędzy obiektami relację jeden-do-wielu w taki sposób, że kiedy wybrany obiekt zmienia swój stan, to wszystkie jego obiekty zależne zostają o tym powiadomione i automatycznie zaktualizowane.

Wzorzec Obserwator definiuje relację jeden–do–wielu pomiędzy obiektami danego zbioru obiektów.

Kiedy stan jednego z obiektów zmienia się, wszystkie jego obiekty zależne zostają o tym powiadomione.

Porównajmy zatem powyższą definicję z tym, o czym do tej pory mówiliśmy na temat wzorca:

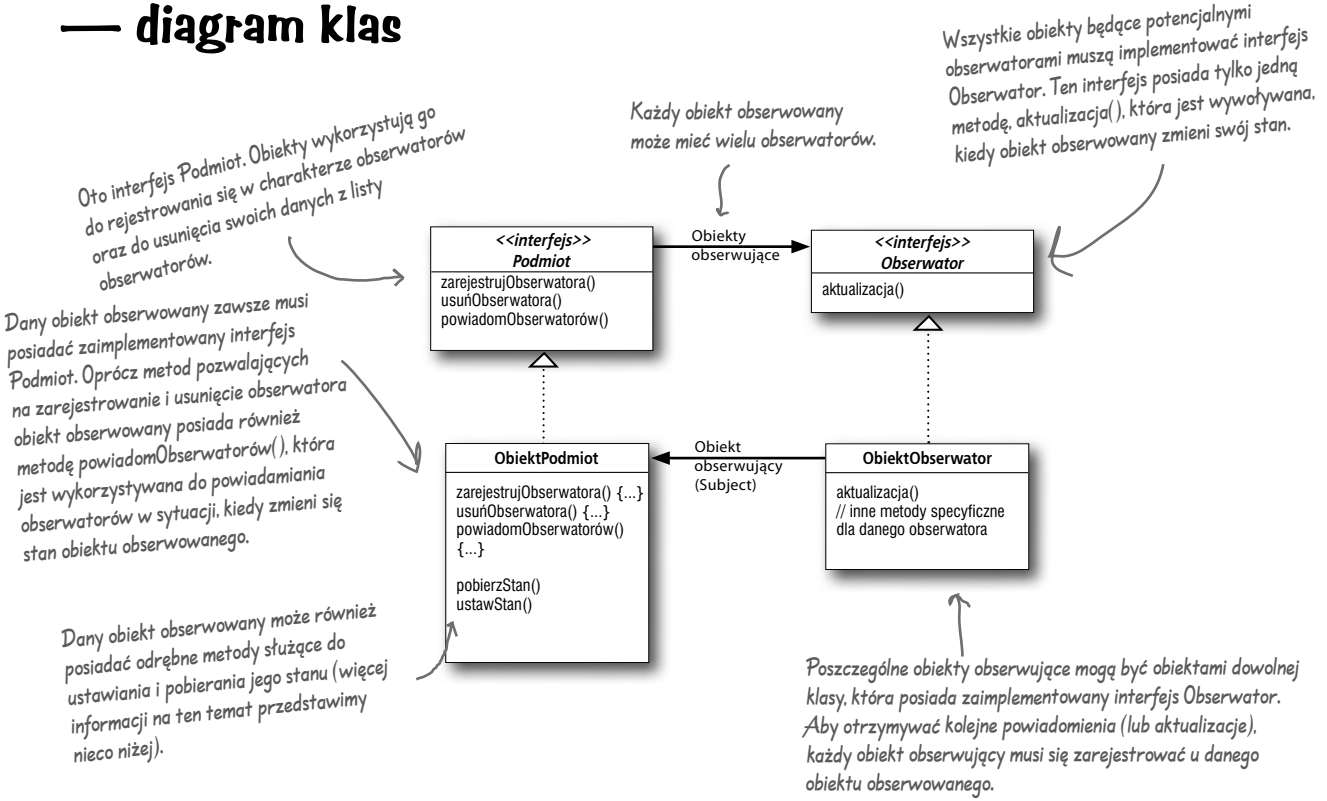


Obiekt obserwowany oraz obiekty obserwujące są ze sobą powiązane relacją jeden-do-wielu. Obiekty obserwujące są zależne od obiektu obserwowanego, co przejawia się w ten sposób, że jeżeli obiekt obserwowany zmienia swój stan, to wszystkie obiekty obserwujące zostają o tym powiadomione. W zależności od sposobu, w jaki realizowane jest powiadamianie, obiekty zależne mogą być również automatycznie aktualizowane (mogą otrzymywać nowe dane od obiektu obserwowanego).

Jak się niebawem sam przekonasz, wzorzec Obserwator może być implementowany na kilka różnych sposobów, aczkolwiek większość z nich obraca się dookoła tworzenia osobnych klas, które posiadają interfejsy Podmiot oraz Obserwator.

Rzućmy na to okiem...

Definicja wzorca Obserwator — diagram klas



Nie istnieją głupie pytania

P: Co to wszystko ma wspólnego z relacją jeden-do-wielu?

O: Zgodnie z założeniami wzorca Obserwaror to obiekt obserwowany jest obiektem, który posiada określony stan i kontroluje go. Zatem wynika stąd, że mamy tylko JEDEN obiekt, który posiada stan. Z kolei obiekty obserwujące wykorzystują ten stan, nawet jeżeli nie są jego właścicielami. Krótko mówiąc, istnieje wielu obserwatorów, którzy polegają na informacjach docierających z obiektu obserwowanego, przekazywanych w momencie, kiedy jego stan ulega zmianie. Możemy zatem powiedzieć, że jest to relacja pomiędzy JEDNYM obiektem obserwowanym, a WIELOMA obiektami obserwującymi.

P: Gdzie zatem w tym całym układzie znajduje się zależność?

O: Ponieważ to obiekt obserwowany jest jedynym i wyłącznym właścicielem danych, aktualizacja obiektów obserwujących jest zależna od zmiany stanu obiektu obserwowanego. Takie podejście pozwala na stosowanie bardziej przejrzystego procesu projektowania (zorientowanego obiektowo), niż miałoby to miejsce w sytuacji, kiedy wiele obiektów może sterować tym samym zestawem danych.

Siła luźnych zależności

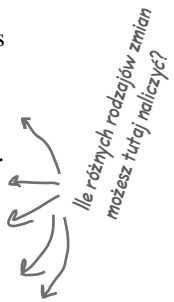
Kiedy dwa obiekty są ze sobą luźno powiązane, mogą ze sobą współpracować, ale z drugiej strony wzajemnie nie wiedzą o sobie zbyt wiele.

Wzorzec Obserwator zapewnia utworzenie takiej struktury obiektów, w której obiekty obserwowane są luźno powiązane ze swoimi obiektami obserwującymi.

Dlaczego?

Jedyną informacją, jaką obiekt obserwowany posiada o obiekcie obserwującym, jest to, że obiekt obserwujący posiada pewien określony interfejs (interfejs Obserwator). Obiekt obserwowany nie musi nawet wiedzieć, do jakiej klasy należy obiekt obserwujący i czym się on zajmuje. Nie musi też znać żadnej tego typu informacji.

Nowych obserwatorów możemy dodawać w dowolnym momencie. Ponieważ jedynym elementem, na którym opiera się działanie obiektu obserwowanego, jest lista zarejestrowanych obiektów implementujących interfejs Obserwator, nowe obiekty obserwujące mogą być dodawane w dowolnie wybranej chwili. W praktyce w czasie działania programu możemy wymienić dowolny, zarejestrowany obiekt obserwujący na inny — a obiekt obserwowany niezależnie od tego będzie sobie funkcjonował tak samo, jak do tej pory. Jak łatwo się domyślić, w analogiczny sposób możemy usunąć w dowolnej chwili dowolny obiekt obserwujący.



Dodawanie nowych typów obserwatorów nigdy nie pociąga za sobą konieczności modyfikacji obiektu obserwowanego. Powiedzmy, że pojawiła nam się (nie wnikamy, skąd) nowa klasa, której obiekty muszą być obserwatorami. Aby dodać je do listy subskrybentów, nie musimy modyfikować kodu obiektu obserwowanego — wszystko, czym musimy się zająć, to implementacja w nowym obiekcie interfejsu Obserwator i zarejestrowanie nowego obserwatora na liście obiektu obserwowanego. Ten ostatni zupełnie nie będzie się tym przejmował; jego zadaniem jest po prostu dostarczanie określonych informacji do wszystkich obiektów, które zostały zarejestrowane i implementują interfejs Obserwator.

Zarówno obiekty obserwowane, jak i obiekty obserwujące mogą być niezależnie od siebie wielokrotnie wykorzystywane. Jeżeli dla któregoś z obiektów obserwowanych lub obserwujących znajdziesz inne zastosowanie, możesz je łatwo wykorzystać, ponieważ są one ze sobą bardzo luźno powiązane.

Zarówno zmiany wprowadzane do obiektu obserwowanego, jak do obiektów obserwujących nie mają wzajemnie na siebie żadnego wpływu. Ponieważ wspomniane dwa rodzaje obiektów są ze sobą luźno powiązane, możemy je niemal dowolnie modyfikować tak długo, jak długo poszczególne obiekty będą spełniały założenie posiadania poprawnie zaimplementowanego interfejsu Podmiot bądź Obserwator.



Reguła projektowania

Staraj się tworzyć projekty, w których obiekty są ze sobą luźno powiązane i, o ile to możliwe, nie oddziałują na siebie wzajemnie.

Projekty charakteryzujące się luźnymi powiązaniami pomiędzy obiektami pozwalają na tworzenie bardzo elastycznych systemów, w łatwy sposób adaptujących się do wprowadzanych zmian, co zawdzięczają minimalizacji wzajemnych zależności wewnętrznych.



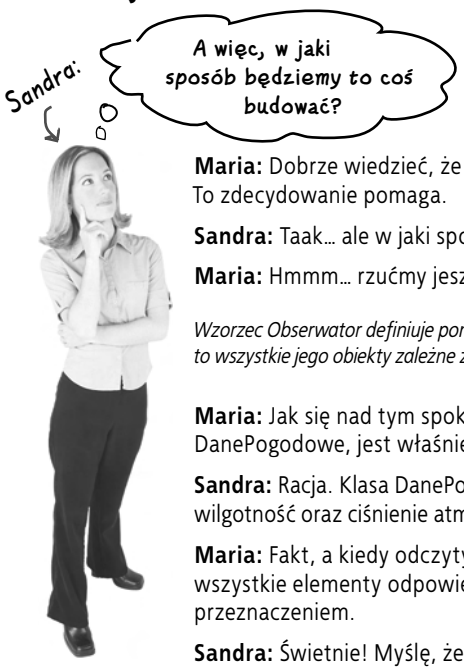
Zaostrz ołówek

Zanim przejdziesz do kolejnej sekcji, spróbuj naszkicować strukturę klas, jakiej będziesz potrzebował do zaimplementowania oprogramowania stacji meteorologicznej, włączając w to klasę DanePogodowe oraz jej elementy odpowiadające za wyświetlanie. Upewnij się, że rysowany diagram pokazuje, w jaki sposób poszczególne elementy są ze sobą połączone, a także wyjaśnia, w jaki sposób inny programista lub projektant może dołączyć swój własny, nowy tryb wyświetlania.

Jeżeli potrzebujesz małej pomocy, przeczytaj to, co zamieściliśmy na następnej stronie; wygląda na to, że Twoje koleżanki z boksu obok już rozpoczęły dysputę na temat tego projektu.

Zasłyszane w sąsiednim boksie...

Wracając do tematu projektu stacji meteorologicznej... wygląda na to, że Twoje koleżanki z boksu obok zaczęły już myśleć o rozwiązaniu tego zadania...



Maria: Dobrze wiedzieć, że będziemy się posługiwać wzorcem Obserwator. To zdecydowanie pomaga.

Sandra: Taak... ale w jaki sposób go zaimplementujemy?

Maria: Hmmm... rzućmy jeszcze raz okiem na definicję tego wzorca:

Wzorzec Obserwator definiuje pomiędzy obiektami relację jeden-do-wielu w taki sposób, że kiedy wybrany obiekt zmienia swój stan, to wszystkie jego obiekty zależne zostają o tym powiadomione i automatycznie zaktualizowane.

Maria: Jak się nad tym spokojnie zastanowić, wychodzi na to, że taka definicja ma sens. Nasza klasa, DanePogodowe, jest właśnie tym „jednym", a „wieloma" są poszczególne tryby wyświetlania informacji o pogodzie.

Sandra: Racja. Klasa DanePogodowe posiada również swoje stany... to znaczy, mam na myśli temperaturę otoczenia, wilgotność oraz ciśnienie atmosferyczne, a te wartości definitywnie będą się zmieniać na przestrzeni czasu.

Maria: Fakt, a kiedy odczyty tych parametrów będą się zmieniały, będziemy musieli poinformować o tym wszystkie elementy odpowiedzialne za wyświetlanie, tak aby mogły przetwarzać nowe wartości zgodnie ze swoim przeznaczeniem.

Sandra: Świetnie! Myślę, że już wiem, w jaki sposób możemy zastosować wzorzec Obserwator do naszego systemu śledzenia pogody według wskazań czujników stacji meteorologicznej.

Maria: Ale ciągle pozostaje kilka spraw, które musimy rozważyć, a co do których nie jestem całkowicie pewna, że je dobrze rozumiem.

Sandra: Na przykład?

Maria: Na przykład to, w jaki sposób dostarczymy wyniki pomiarów poszczególnych parametrów do elementów odpowiedzialnych za wyświetlanie danych.

Sandra: No cóż, przypomnij sobie, jak wygląda diagram wzorca Obserwator. Jeżeli naszym obiektem obserwowanym zostanie obiekt DanePogodowe, a obiektami obserwującymi będą elementy (obiekty) odpowiedzialne za wyświetlanie poszczególnych informacji, to aby otrzymywać aktualne informacje, obiekty takie będą musiały najpierw zostać zarejestrowane przez obiekt obserwowany, prawda?

Maria: Tak... a kiedy obiekt obserwowany będzie znał listę swoich subskrybentów, wystarczy, że będzie wywoływał odpowiednie metody, których zadaniem będzie przekazywanie im zaktualizowanych danych z pomiarów.

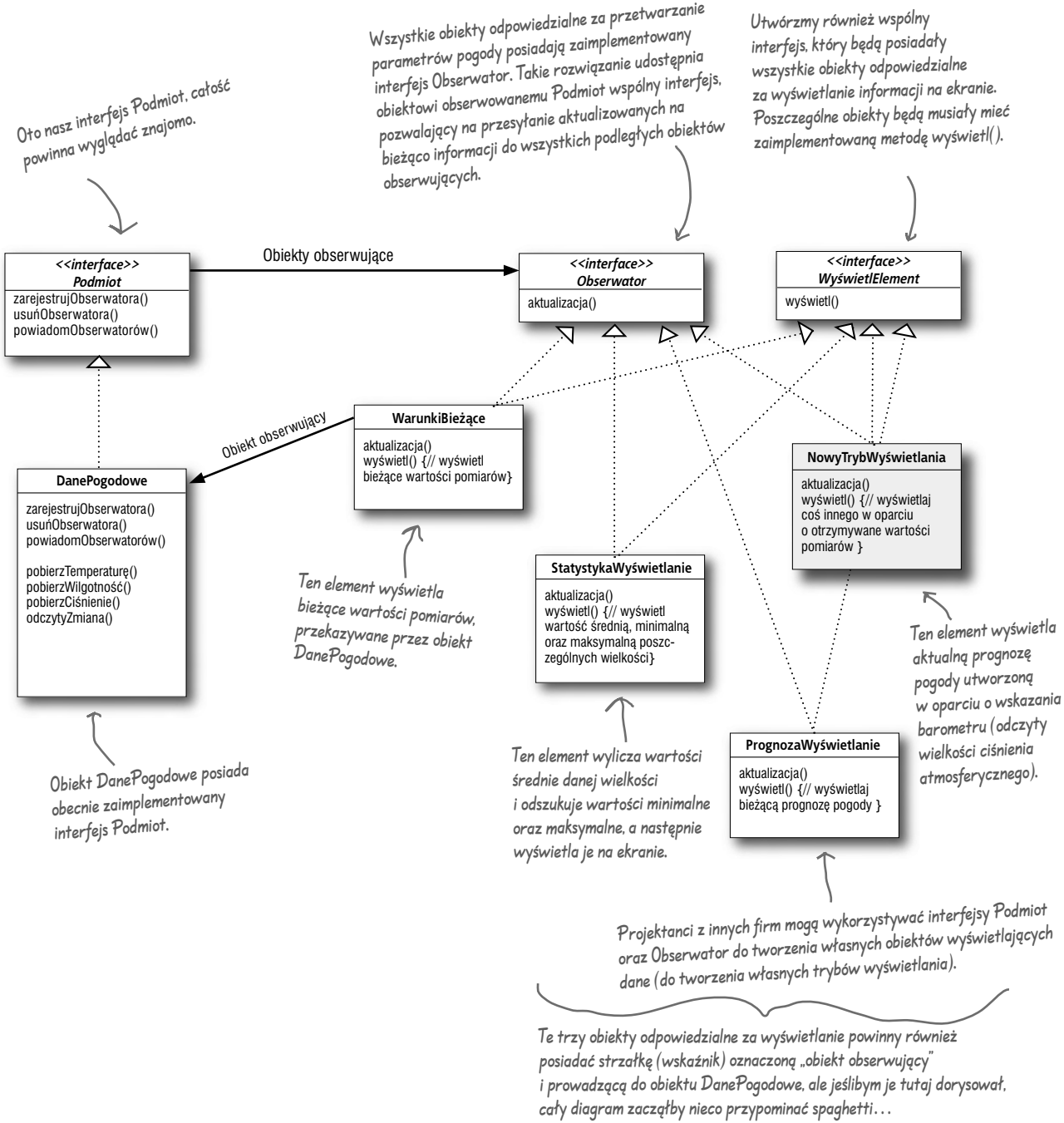
Sandra: Musimy jednak pamiętać, że każdy wyświetlany element może być zupełnie inny... Więc tak sobie myślę, gdzie jest miejsce na wprowadzenie wspólnego interfejsu? Pomimo, że poszczególne elementy mogą być zupełnie odmiennych typów, wszystkie powinny posiadać taki sam interfejs, tak aby obiekt DanePogodowe wiedział, w jaki sposób wysyłać im wyniki poszczególnych pomiarów.

Maria: Wiem, o co Ci chodzi. Wynika z tego, że każdy obiekt odpowiedzialny za wyświetlanie będzie posiadał metodę, powiedzmy, aktualizacja(), która będzie wywoływana przez obiekt DanePogodowe.

Sandra: ... i ta metoda aktualizacja() będzie zdefiniowana w wspólnym interfejsie, zaimplementowanym we wszystkich obiektach.

Projektowanie stacji meteorologicznej

Jak poniższy diagram wygląda w porównaniu z Twoim?



Implementacja stacji meteorologicznej

Rozpoczniemy teraz implementację aplikacji obsługującej stację meteorologiczną, wykorzystując do tego celu przedstawiony diagram klas oraz biorąc pod uwagę wnioski płynące z rozmowy Marii i Sandry (zapis rozmowy zamieściliśmy kilka stron wcześniej). W nieco dalszej części niniejszego rozdziału przekonasz się, że Java posiada pewne wbudowane mechanizmy obsługi wzorca Obserwator, aczkolwiek w chwili obecnej mamy szczery zamiar nieco pobrudzić sobie rączki, samemu zabrać się do roboty i stworzyć niezbędny kod. Co prawda, wbudowane mechanizmy Javy pozwalają na skorzystanie z nich w niektórych wypadkach, ale samodzielne stworzenie odpowiednich klas gwarantuje o wiele większą elastyczność całego systemu (a zaręczam, że nie jest to takie trudne). Dobrze, zatem zacznijmy od stworzenia odpowiednich interfejsów (pliki Podmiot.java, Obserwator.java oraz WyświetlElement.java):

```
public interface Podmiot {
    public void zarejestrujObserwatora (Obserwator o);
    public void usuńObserwatora(Obserwator o);
    public void powiadomObserwatorów();
}
```

Metody pobierają jako argument obiekt typu Obserwator, czyli konkretnie obiekt, który ma zostać zarejestrowany na liście zarejestrowanych bądź usunięty z tej listy.

Ta metoda jest wywoływana w celu powiadomienia wszystkich obserwatorów o tym, że stan obiektu obserwowanego zmienił się.

```
public interface Obserwator {
    public void aktualizacja(float temp, float wilgotność, float ciśnienie);
}
```

Te zmienne odpowiadają wartościom stanu, jakie obiekty obserwujące otrzymują od obiektu obserwowanego, kiedy zmieniają się odczyty parametrów pogody.

Wszystkie obiekty obserwujące posiadają zaimplementowany interfejs Obserwator, stąd wszystkie muszą mieć także zaimplementowaną metodę aktualizacja(). Właśnie w tym miejscu wykorzystamy sugestie, jakie padły podczas rozmowy Marii z Sandrą, i tutaj będziemy przekazywać wyniki kolejnych pomiarów do poszczególnych obserwatorów.

```
public interface WyświetlElement{
    public void wyświetl();
}
```

Interfejs WyświetlElement posiada tylko jedną metodę, wyświetl(), którą będziemy wywoływać w sytuacji, kiedy niezbędne będzie wyświetlanie danego elementu na ekranie.



WYTĘŻ UMYSŁ

Maria i Sandra myślały, że przekazywanie wyników pomiarów bezpośrednio do obiektów obserwujących jest najprostszym sposobem na ich uaktualnianie. Czy zgadzasz się z ich zdaniem? Wskazówka: czy w naszej aplikacji jest jakiś obszar, który może w przyszłości ulec zmianie? Jeżeli taka zmiana nastąpi, to czy będzie ona wystarczająco dobrze hermetyzowana, czy też będzie jednak wymagała dokonania modyfikacji wielu innych części kodu aplikacji?

Czy możesz pomyśleć nad opracowaniem nowych sposobów przekazywania uaktualnionych odczytów do obiektów obserwujących?

Nie musisz się przejmować — powrócimy jeszcze do problemu zatwierdzenia projektu aplikacji po zakończeniu wstępnej fazy jej implementacji.

Implementacja interfejsu Podmiot w klasie DanePogodowe

PAMIĘTAJ: w prezentowanych przykładach listingów nie umieszczamy poleceń „import" ani „package". Komplet kodów źródłowych poszczególnych przykładów znajdziesz na serwerze FTP wydawnictwa HELION: ftp://ftp.helion.pl/przyklady/wzorrg.zip

Czy pamiętasz naszą pierwszą próbę implementacji klasy DanePogodowe, którą przeprowadziliśmy na początku tego rozdziału? Być może będziesz chciał nieco odświeżyć sobie pamięć… Teraz właśnie nastał czas, aby do niej powrócić i trochę ją przeprojektować — mając na względzie wymagania, jakie narzuca nam wzorzec Obserwator…

Tutaj dokonujemy implementacji interfejsu Subject.

```
public class DanePogodowe implements Podmiot {
    private ArrayList obserwatorzy;
    private float temperatura;
    private float wilgotność;
    private float ciśnienie;

    public DanePogodowe() {
        obserwatorzy = new ArrayList();
    }

    public void zarejestrujObserwatora(Obserwator o) {
        obserwatorzy.add(o);
    }

    public void usuńObserwatora(Obserwator o) {
        int i = obserwatorzy.indexOf(o);
        if (i >= 0) {
            obserwatorzy.remove(i);
        }
    }

    public void powiadomObserwatorów() {
        for (int i = 0; i < obserwatorzy.size(); i++) {
            Obserwator Obs = (Obserwator)obserwatorzy.get(i);
            Obs.aktualizacja(temperatura, wilgotność, ciśnienie);
        }
    }

    public void odczytyZmiana() {
        powiadomObserwatorów();
    }

    public void ustawOdczyty(float temperatura, float wilgotność, float ciśnienie) {
        this.temperatura = temperatura;
        this.wilgotność = wilgotność;
        this.ciśnienie = ciśnienie;
        odczytyZmiana();
    }

    // tutaj zamieść inne metody klasy DanePogodowe
}
```

Klasa DanePogodowe implementuje obecnie interfejs Podmiot.

Dodaliśmy zmienną typu ArrayList, której zadaniem jest przechowywanie listy obserwatorów, a tworzenie tej zmiennej umieściliśmy w definicji konstruktora obiektów klasy DanePogodowe.

Kiedy dany obserwator się rejestruje, dopisujemy go po prostu na końcu listy.

Analogicznie, jeżeli dany obserwator chce się wyrejestrować, po prostu usuwamy go z listy.

A tutaj mamy coś fajnego: to jest właśnie miejsce, w którym mówimy obiektom obserwującym (obserwatorom), że zmienił się stan obiektu obserwowanego. Ponieważ wszystkie obiekty obserwujące posiadają interfejs Obserwator, wiemy, że posiadają także zaimplementowaną metodę aktualizacja() i w związku z tym wiemy, jak możemy je powiadomić.

Obserwatorów będziemy powiadamiali po otrzymaniu nowych wartości pomiarów ze stacji meteorologicznej.

No dobra… musimy się przyznać, że ze swojej strony chcieliśmy sprzedawać kopię naszej wspaniałej aplikacji do obsługi stacji meteorologicznej razem z każdym egzemplarzem niniejszej książki, ale, niestety, nasz wydawca nie chciał się zgodzić na takie rozwiązanie. Z tego zatem względu, zamiast odczytywać wyniki pomiarów z jakichś rzeczywistych urządzeń na stacji meteo, będziemy wykorzystywali tę metodę do testowania naszych obiektów odpowiedzialnych za poszczególne tryby wyświetlania. Jeżeli chcesz, dla rozrywki możesz napisać odpowiedni kod, który będzie pobierał odpowiednie informacje z jakiejś strony sieci WWW.

A teraz pora na zbudowanie elementów wyświetlających dane

Mamy już naszą klasę DanePogodowe zapiętą niemal na ostatni guzik, nadszedł więc czas na stworzenie elementów, które będą wyświetlały odpowiednie informacje w poszczególnych trybach pracy na ekranie. Zleceniodawca, firma Pogodynka sp. z o.o., poprosiła o przygotowanie trzech trybów wyświetlania: „Warunki bieżące", „Statystyka" oraz „Prognoza". Rzućmy okiem na element odpowiedzialny za wyświetlanie bieżących warunków pogodowych; kiedy dokładnie się z nim zapoznasz i zrozumiesz jego zasady działania, powinieneś sprawdzić kody pozostałych dwóch elementów, odpowiedzialnych za statystykę i prognozy (odpowiednie kody źródłowe znajdziesz na serwerze FTP wydawnictwa HELION: *ftp://ftp.helion.pl/przyklady/wzorrg.zip*). Szybko się zorientujesz, że ich struktura jest bardzo zbliżona.

Ta klasa, odpowiadająca za wyświetlanie warunków bieżących, posiada zaimplementowany interfejs Obserwator, dzięki czemu może otrzymywać informacje o zmieniających się danych od obiektu klasy DanePogodowe.

Ta klasa posiada również zaimplementowany interfejs WyświetlElement, ponieważ nasze API będzie wymagało, aby każdy element odpowiedzialny za wyświetlanie informacji na ekranie posiadał taki interfejs.

```
public class WarunkiBieżąceWyświetl implements Obserwator, WyświetlElement {
    private float temperatura;
    private float wilgotność;
    private Podmiot DanePogodowe;

    public WarunkiBieżąceWyświetl(Podmiot DanePogodowe) {
        this.DanePogodowe = DanePogodowe;
        DanePogodowe.zarejestrujObserwatora(this);
    }

    public void aktualizacja(float temperatura, float wilgotność, float ciśnienie) {
        this.temperatura = temperatura;
        this.wilgotność = wilgotność;
        wyświetl();
    }

    public void wyświetl() {
        System.out.println("Warunki bieżące " + temperatura
            + " stopni C oraz " + wilgotność + "% wilgotność");
    }
}
```

Konstruktor obiektów klasy WarunkiBieżąceWyświetl otrzymuje jako argument wywołania obiekt DanePogodowe (czyli obiekt obserwowany, Podmiot), a następnie wykorzystuje go do zarejestrowania swojego obiektu jako obserwatora.

Kiedy wywoływana jest metoda aktualizacja(), zachowujemy wartości temperatury otoczenia oraz wilgotności, a następnie wywołujemy metodę wyświetl().

Metoda wyświetl() powoduje po prostu wyświetlenie najbardziej aktualnych odczytów temperatury i wilgotności.

Nie istnieją głupie pytania

P: Czy metoda aktualizacja() to najlepsze miejsce do umieszczenia wywołania metody realizującej wyświetlanie?

O: W naszym prostym przykładzie wywoływanie metody wyświetl(), kiedy następowała aktualizacja danych, miało sens. Nie zmienia to jednak faktu, że masz rację, twierdząc, że istnieją o wiele lepsze sposoby wyświetlania przychodzących danych. Przekonasz się o tym przy okazji omawiania wzorca Model-Widok-Kontroler.

P: Dlaczego odwołanie do obiektu obserwowanego (Podmiot) jest zapamiętywane? Przecież nie wygląda na to, aby było ono używane gdziekolwiek poza konstruktorem tej klasy?

O: Prawda, aczkolwiek w przyszłości może zaistnieć potrzeba usunięcia tego obiektu z listy zarejestrowanych obserwatorów i wtedy posiadanie takiego gotowego, istniejącego odwołania może okazać się bardzo użyteczne.

Włączamy zasilanie naszej stacji meteorologicznej



❶ Po pierwsze, przygotujmy środowisko testowe.

Nasz aplikacja do obsługi stacji meteorologicznej jest w zasadzie gotowa do pracy. Jedyne, czego jeszcze będziemy potrzebować, to odrobina kodu, który połączy wszystkie stworzone do tej pory moduły w jedną, zgrabną i sprawnie działającą całość. Poniżej znajdziesz naszą pierwszą przymiarkę do tego zadania. W nieco dalszej części książki powrócimy jeszcze do tego zagadnienia i upewnimy się, że poszczególne elementy składowe całej aplikacji mogą być łatwo do niej dołączane za pośrednictwem odpowiedniego pliku konfiguracyjnego. W chwili obecnej nasz kod wygląda następująco:

```
public class StacjaMeteo {

    public static void main(String[] args) {
        DanePogodowe danePogodowe = new DanePogodowe();

        WarunkiBieżąceWyświetl warunkiBieżąceWyświetl =
            new WarunkiBieżąceWyświetl(danePogodowe);
        StatystykaWyświetl statystykaWyświetl = new StatystykaWyświetl(danePogodowe);
        PrognozaWyświetl prognozaWyświetl = new PrognozaWyświetl(danePogodowe);

        danePogodowe.ustawOdczyty(26.6, 65, 1013.1f);
        danePogodowe.ustawOdczyty(27.7, 70, 997.0f);
        danePogodowe.ustawOdczyty(25.5, 90, 997.0f);
    }
}
```

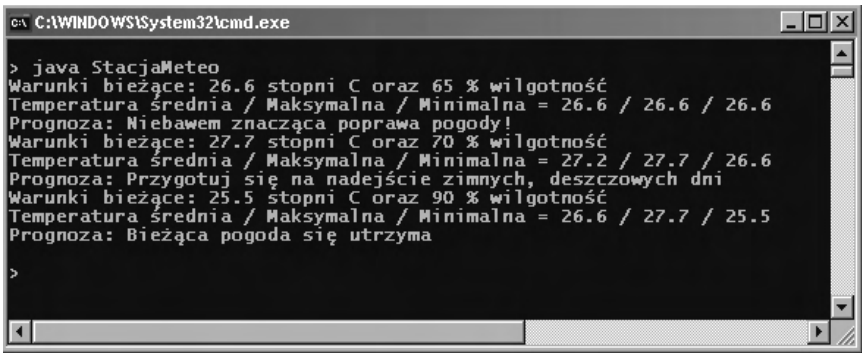
Najpierw utworzymy obiekt klasy DanePogodowe.

Jeżeli nie chcesz ściągać gotowych przykładów z serwera FTP wydawnictwa Helion, możesz po prostu wykomentować te dwa wiersze i uruchomić program bez nich.

Tworzymy trzy obiekty odpowiedzialne za poszczególne tryby wyświetlania i przekazujemy im stworzony obiekt klasy DanePogodowe.

Symulacja kolejnych wartości odczytów z czujników pogody.

❷ Uruchom program i pozwól, aby zaczęła działać magia wzorca Obserwator.





Zaostrz ołówek

Właśnie przed chwilą zadzwonił p. Janusz Monsun, prezes rady nadzorczej firmy Pogodynka sp. z o.o., i stwierdził, że sprzedaż całego systemu nie będzie możliwa, jeżeli nie zostanie zaimplementowany nowy tryb wyświetlania, prezentujący tzw. indeks ciepła. Poniżej załączamy definicję tego indeksu:

Indeks ciepła to wskaźnik, który łączy ze sobą wskazania temperatury i wilgotności powietrza i określa rzeczywistą temperaturę otoczenia (tzn. *odczuwaną temperaturę*). Aby obliczyć wartość indeksu ciepła, należy wziąć wartość temperatury `T*)` oraz wilgotności względnej `RH`, a następnie podstawić do poniższego wzoru:

indeksCiepła =

$$16.923 + 1.85212 * 10^{-1} * T + 5.37941 * RH - 1.00254 * 10^{-1} * T * RH + 9.41695 * 10^{-3} * T2 + 7.28898 * 10^{-3} * RH2 + 3.45372 * 10^{-4} * T2 * RH - 8.14971 * 10^{-4} * T * RH2 + 1.02102 * 10^{-5} * T2 * RH2 - 3.8646 * 10^{-5} * T3 + 2.91583 * 10^{-5} * RH3 + 1.42721 * 10^{-6} * T3 * RH + 1.97483 * 10^{-7} * T * RH3 - 2.18429 * 10^{-8} * T3 * RH2 + 8.43296 * 10^{-10} * T2 * RH3 - 4.81975 * 10^{-11} * T3 * RH3$$

Słowem, nie pozostało Ci nic innego, jak tylko rozpocząć „wklepywanie"!

Oczywiście, żartowaliśmy! Nie martw się, nie będziesz musiał pracowicie przepisywać całego wzoru; wystarczy, że stworzysz swój własny plik *IndeksCiepła.java* i skopiujesz do niego gotową formułę, którą znajdziesz w pliku *IndeksCiepła.txt*.

Plik IndeksCiepła.txt możesz odnaleźć na serwerze FTP wydawnictwa Helion: ftp://ftp.helion.pl/przyklady/wzorrg.zip

Jak działa ta magiczna formuła? Aby się tego dowiedzieć, powinieneś zajrzeć do książki *Meteorologia. Rusz głową!* (o ile kiedyś takowa się ukaże), zapytać kogoś pracującego w Instytucie Meteorologii i Gospodarki Wodnej lub po prostu zapytać Wielkiego Google'a (*http://www.google.pl*).

Po wprowadzeniu odpowiednich zmian i uruchomieniu programu wyniki jego działania powinny wyglądać następująco:

Oto zmiany, jakie powinieneś zauważyć w wynikach działania programu.

```
C:\WINDOWS\System32\cmd.exe
> java StacjaMeteo
Warunki bieżące: 26.6 stopni C oraz 65 % wilgotność
Temperatura średnia / Maksymalna / Minimalna = 26.6 / 26.6 / 26.6
Prognoza: Niebawem znacząca poprawa pogody!
Obliczony indek ciepła ma wartość 28.30852
Warunki bieżące: 27.7 stopni C oraz 70 % wilgotność
Temperatura średnia / Maksymalna / Minimalna = 27.2 / 27.7 / 26.6
Prognoza: Przygotuj się na nadejście zimnych, deszczowych dni
Obliczony indek ciepła ma wartość 30.50068
Warunki bieżące: 25.5 stopni C oraz 90 % wilgotność
Temperatura średnia / Maksymalna / Minimalna = 26.6 / 27.7 / 25.5
Prognoza: Bieżąca pogoda się utrzyma
Obliczony indek ciepła ma wartość 28.69426

>
```



Temat dzisiejszej wieczornej pogawędki: **Obserwowany (Podmiot) oraz Obserwujący (Obserwator) przekomarzają się na temat najlepszego sposobu przekazywania informacji o zmianach stanu do obiektów obserwujących.**

Obiekt obserwowany

Cieszę się, że w końcu mamy okazję porozmawiać osobiście.

Niby tak, ale przecież robię to, co do mnie należy. Zawsze mówię Wam, co w trawie piszczy… To, że nie wiem, kim tak naprawdę jesteście, nie oznacza wcale, że o Was nie dbam. A poza tym wiem o Was jedną, najważniejszą rzecz — macie zaimplementowany interfejs Obserwator.

Tak? Ciekawe, co?

Przepraszam BARDZO. Wysyłam powiadomienia wraz z informacjami o moim stanie tak, abyście Wy, leniwi obserwatorzy, wiedzieli, co tu się w końcu dzieje!

No dobrze…. myślę, że to mogłoby zadziałać. Musiałbym tylko bardziej otworzyć się na Wasze zakusy i pozwolić wszystkim Obserwatorom dostać się do środka tak, abyście mogli samodzielnie pobierać potrzebne Wam informacje o stanie. Niemniej jednak, takie rozwiązanie może być nieco niebezpieczne. Nie mogę przecież pozwolić na to, aby Obserwatorzy wchodzili na mój teren bez zaproszenia i przeglądali wszystko, co mam do zaoferowania.

Obiekt obserwujący

Naprawdę?? A ja myślałem, że Ty zupełnie nie dbasz o nas, Obserwatorów.

Tak, pewnie, ale to tylko malutka część tego, co posiadamy. Tak czy inaczej, z pewnością wiemy o Tobie znacznie więcej…

Na przykład to, że zawsze przekazujesz informacje o swoim stanie nam, Obserwatorom, abyśmy wiedzieli, co się wewnątrz Ciebie dzieje. A to od czasu do czasu może być nieco irytujące…

OK, poczekaj moment i nie gorączkuj się tak; po pierwsze, nie jesteśmy leniwi, mamy po prostu bardzo wiele innych zadań do wykonania pomiędzy tymi Twoimi jakże-bardzo-ważnymi-powiadomieniami, panie Obserwowany; po drugie, dlaczego nie pozwolisz nam po prostu wejść na Twój teren i wziąć potrzebnych informacji, zamiast wypychać je uparcie na siłę wszystkim Obserwatorom?

Obiekt obserwowany

Tak, mógłbym w zasadzie pozwolić Ci na **pobieranie** mojego stanu. Ale czy takie rozwiązanie nie będzie dla Ciebie mniej wygodne? Jeżeli musiałbyś przychodzić do mnie za każdym razem, kiedy będziesz czegoś potrzebował, to może się zdarzyć, że aby otrzymać niezbędne informacje, będziesz zmuszony do wykonywania wielokrotnych wywołań metod.
Z tej prostej przyczyny wolę **wysyłać** Wam mój stan… dzięki temu otrzymujecie to wszystko, co niezbędne, w jednym, prostym powiadomieniu.

Hmmm, jak by na to nie patrzeć, faktycznie oba sposoby mają swoje zalety. Zauważyłem, że Java posiada wbudowaną obsługę wzorca Obserwator, który pozwala na przesyłanie informacji zarówno w postaci wysyłania, jak i pobierania danych.

Świetnie… być może będę miał zatem okazję zobaczyć dobry przykład takiego rozwiązania, który pozwoli mi zmienić zdanie…

Obiekt obserwujący

To dlaczego po prostu nie napiszesz jakiejś publicznie dostępnej metody pozwalającej na pobieranie danych i nie udostępnisz nam jej, abyśmy sami mogli pobierać niezbędne informacje o Twoim stanie?

Nie bądź taki natarczywy! Istnieje tak wiele różnych typów Obserwatorów, że z pewnością nie możesz zawsze przewidzieć wszystkich naszych potrzeb. Po prostu pozwól nam wejść na Twój teren i pobrać potrzebne informacje o Twoim stanie. W ten sposób, jeżeli któryś z nas będzie potrzebował tylko pewnej niewielkiej części informacji, nie będzie zmuszony do przyjmowania ich kompletu. Dzięki temu łatwiej później będzie dokonywać modyfikacji. Powiedzmy, na przykład, że trochę się rozwiniesz i Twój stan będzie posiadał kilka nowych informacji. Wtedy, hmmm... jeżeli korzystasz z metody pobierania danych, nie będziesz musiał biegać dookoła i modyfikować metody aktualizacja() każdego zarejestrowanego Obserwatora. Wystarczy, że zmienisz się sam, udostępniając nowe metody, dające dostęp do dodatkowych elementów Twojego stanu.

Och, naprawdę? Myślę zatem, że powinniśmy mu się przyjrzeć…

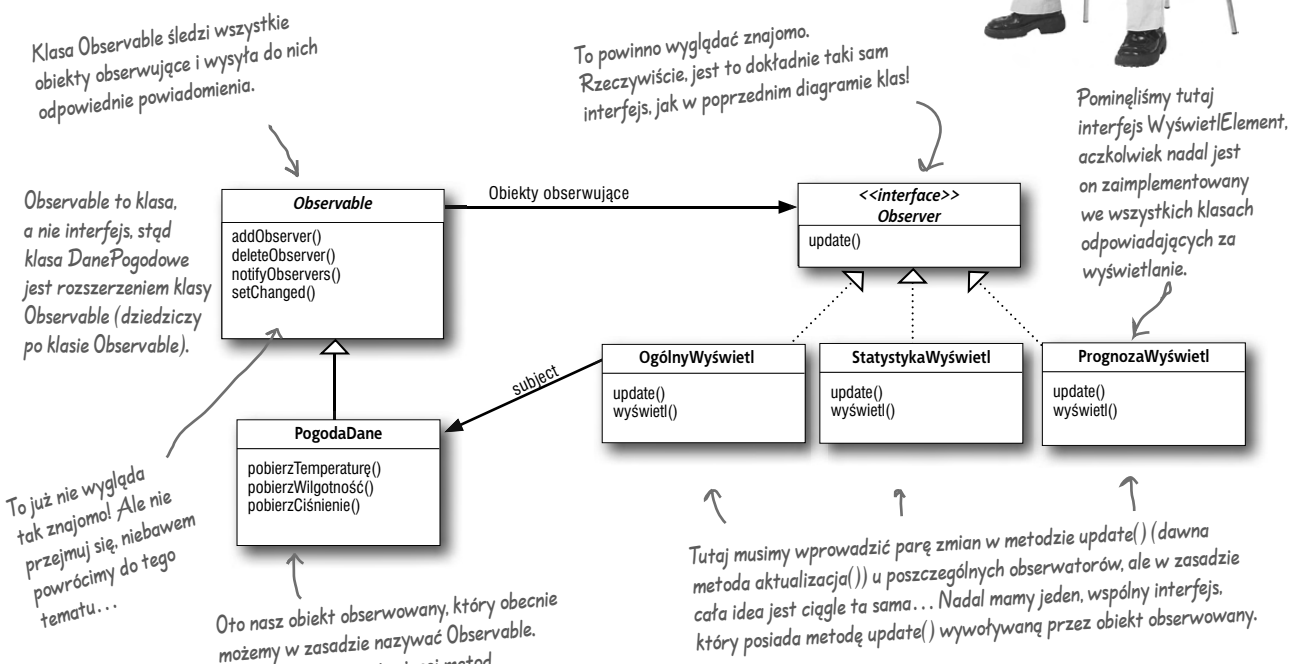
Niemożliwe, czyżbyśmy jednak doszliśmy do jakiegoś porozumienia? Czyli jednak zawsze jest jakaś nadzieja…?

Java — zastosowanie wbudowanego wzorca Obserwator

Jak pamiętasz, do tej pory tworzyliśmy swój kod, zgodny z wymogami wzorca Obserwator. Warto jednak wspomnieć, że język Java posiada także własną obsługę tego wzorca, wbudowaną w kilka różnych API. Najbardziej ogólną implementacją jest interfejs Observer i klasa Observable (oba elementy zlokalizowane w pakiecie java.util). Elementy te są dosyć podobne do naszych interfejsów Obserwator i Podmiot, ale już gotowe, i praktycznie natychmiast udostępniają Ci swoją dosyć rozbudowaną funkcjonalność. Dzięki temu pakietowi możesz skorzystać zarówno z metody wysyłania powiadomień przez obiekt obserwowany, jak i z metody pobierania stanu obiektu obserwowanego przez obiekty obserwujące, co zresztą postaramy się zademonstrować.

Aby lepiej zrozumieć interfejs java.util.Observer i klasę java.util.Observable, przeanalizuj przedstawiony poniżej nowy, zorientowany obiektowo projekt naszej stacji meteorologicznej:





Java — jak funkcjonuje wbudowany wzorzec Obserwator

Wbudowane mechanizmy obsługi wzorca Obserwator działają nieco inaczej niż implementacja, której używaliśmy w początkowej wersji projektu stacji meteorologicznej. Najbardziej oczywistą różnicę stanowi fakt, że klasa DanePogodowe (czyli nasz obiekt obserwowany) jest teraz rozszerzeniem klasy Observable i dziedziczy po niej metody addObserver(), deleteObserver() oraz notifyObservers() (a oprócz tego jeszcze kilka innych). Poniżej dowiesz się, w jaki sposób możemy skorzystać w języku Java z wbudowanego mechanizmu obsługi wzorca Obserwator.

Aby dowolny obiekt został obserwatorem...

Jak zwykle, musisz zaimplementować interfejs Obserwatora (teraz jest to interfejs java.util.Observer), a następnie wywołać metodę addObserver() dla dowolnego, wybranego obiektu klasy Observable. Analogicznie, aby usunąć dany obiekt z listy obserwatorów, musisz wywołać metodę deleteObserver().

Aby wybrany obiekt klasy Observable wysyłał powiadomienia o zmianie stanu...

Po pierwsze i najważniejsze, dany obiekt musi dać się obserwować, czyli musi dziedziczyć po klasie nadrzędnej java.util.Observable. Jeżeli ten warunek jest spełniony, trzeba wykonać jeszcze dwa kroki:

1. **Musisz wywołać metodę setChanged(), aby zasygnalizować, że zmienił się stan obiektu obserwowanego.**

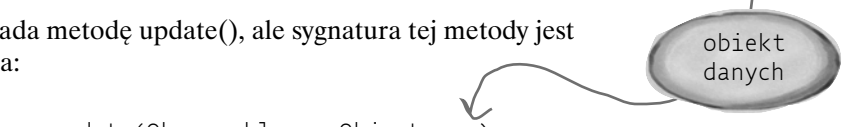
2. **Następnie musisz wywołać jedną z dwóch dostępnych metod notifyObservers():**

   `notifyObservers()` lub `notifyObservers(Object arg)`.

Ta wersja pozwala na pobranie dowolnego obiektu danych, który następnie zostanie przekazany podczas powiadamiania do wszystkich zarejestrowanych obiektów obserwujących.

Aby obserwator otrzymywał powiadomienia...

Obserwator posiada metodę update(), ale sygnatura tej metody jest jednak nieco inna:



Jako argument jest tutaj przekazywany obiekt, który wysłał powiadomienie

Tutaj jako argument jest przekazywany obiekt danych, który wcześniej został przekazany do metody notifyObservers(), lub też wartość null, jeżeli obiekt danych nie został wcześniej określony.

Jeżeli chcesz wysyłać dane do obserwatorów, możesz przekazać odpowiednie dane jako obiekt danych do metody notifyObserver(arg). Jeżeli nie, obserwator musi „pobierać" potrzebne dane z obiektu (obserwowanego) klasy Observable, który zostaje do niego przekazany. Jak? Zmienimy teraz po raz kolejny projekt naszej stacji meteorologicznej i wtedy przekonasz się sam.



Fragment pseudokodu klasy Observable.

Metoda setChanged() jest wykorzystywana do zasygnalizowania, że stan obiektu obserwowanego zmienił się i wywołanie metody notifyObservers() powinno przynieść aktualizację danych dla obserwatorów. Jeżeli metoda notifyObservers() zostanie wywołana bez uprzedniego wywołania metody setChanged(), obserwatorzy NIE ZOSTANĄ powiadomieni. Rzućmy teraz okiem za kulisy (czyli zajrzyjmy do kodu źródłowego klasy Observable) i zobaczmy, jak działa metoda setChanged().

Za kulisami

```
setChanged() {
    changed = true
}

notifyObservers(Object arg){
    if (changed) {
        dla każdego obserwatora na liście {
            call update(this, arg)
        }
        changed = false
    }
}

notifyObservers(){
    notifyObservers(null)
}
```

Metoda setChanged() ustawia flagę changed na wartość „prawda" (true).

Metoda notifyObservers() będzie wysyłała powiadomienia wyłącznie wtedy, jeśli wcześniej flaga changed zostanie ustawiona na wartość „prawda" (true).

Po rozesłaniu powiadomień metoda ponownie ustawia flagę changed na wartość „fałsz" (false).

Dlaczego takie rozwiązanie jest konieczne? Metoda setChanged() została pomyślana jako sposób na zapewnienie większej elastyczności w sposobie powiadamiania obserwatorów, sposób pozwalający na wprowadzenie optymalizacji powiadomień. Przykładowo, załóżmy, że na naszej stacji meteorologicznej znajdują się tak bardzo czułe przyrządy, że odczyty temperatury ciągle oscylują wokół pewnej temperatury bazowej z dokładnością do kilku setnych części stopnia. Taka sytuacja może spowodować, że obiekt klasy DanePogodowe będzie nieustannie wysyłał nowe powiadomienia. Zamiast do tego dopuścić, lepiej będzie wysyłać powiadomienia wyłącznie wtedy, kiedy różnica temperatury pomiędzy dwoma kolejnymi pomiarami wyniesie, dajmy na to, co najmniej pół stopnia (inaczej mówiąc, tylko w takiej sytuacji wywołana zostanie metoda setChanged()).

Być może nie będziesz zbyt często korzystał z opisanego przed chwilą mechanizmu, niemniej jednak, jeżeli już zaistnieje taka konieczność, warto pamiętać, że rozwiązanie masz gotowe. Pamiętaj również, że jeżeli chcesz, aby powiadomienia w ogóle działały, musisz najpierw wywołać metodę setChanged(). Jeżeli zdecydujesz się na korzystanie z opisanego mechanizmu, warto również wspomnieć o metodach takich, jak clearChanged() (ustawia flagę changed na wartość „fałsz" (false)) czy hasChange() (zwraca aktualny stan flagi changed).

Kolejna wersja kodu Stacji Meteo — zastosowanie wbudowanych mechanizmów Javy do obsługi wzorca Obserwator

Po pierwsze, dopasujemy kod klasy DanePogodowe tak, aby można było skorzystać z klasy java.util.Observable

1 *Upewnij się, że importujemy właściwą klasę i właściwy interfejs.*

2 *Teraz dla klasy Observable tworzymy nową klasę podrzędną.*

3 *Nie musimy już dłużej śledzić naszych obserwatorów ani zarządzać procesami ich rejestracji oraz usuwania z listy (tym zajmuje się już klasa nadrzędna), więc usunęliśmy kod przeznaczony uprzednio do rejestracji, usuwania i wysyłania powiadomień do obserwatorów.*

```
import java.util.Observable;
import java.util.Observer;

public class DanePogodowe extends Observable {
    private float temperatura;
    private float wilgotność;
    private float ciśnienie;

    public DanePogodowe() { }

    public void odczytyZmiania() {
        setChanged();
        notifyObservers();*
    }

    public void ustawOdczyty(float temperatura, float wilgotność, float ciśnienie) {
        this.temperatura = temperatura;
        this.wilgotność = wilgotność;
        this.ciśnienie = ciśnienie;
        odczytyZmiana();
    }

    public float pobierzTemperaturę() {
        return temperatura;
    }

    public float pobierzWilgotność() {
        return wilgotność;
    }

    public float pobierzCiśnienie() {
        return ciśnienie;
    }
}
```

4 *Nasz konstruktor już nie musi tworzyć osobnej struktury danych przeznaczonej do przechowywania danych obserwatorów.*

* *Zwróć uwagę, że nie wysyłamy już obiektu danych do metody notifyObservers(). Oznacza to, że właśnie korzystamy z metody POBIERANIA stanu obiektu obserwowanego.*

5 *Zanim wywołamy metodę notifyObservers(), musimy wywołać metodę setChanged(), która ustawi flagę wskazującą, że stan obiektu obserwowanego zmienił się.*

6 *Te metody nie są nowe, ale ponieważ chcemy skorzystać z metody „pobierania", pomyśleliśmy sobie, że przypomnimy Ci o ich istnieniu. Obserwatorzy będą ich używać do pobierania informacji o stanie obiektu klasy DanePogodowe.*

A teraz przyszedł czas na modyfikację klasy WarunkiBieżąceWyświetl

1. Ponownie upewnij się, że importujemy właściwą klasę i właściwy interfejs.
2. Obecnie importujemy interfejs Observer z pakietu java.util

```
import java.util.Observable;
import java.util.Observer;

public class WarunkiBieżąceWyświetl implements Observer, WyświetlElement {
    Observable observable;
    private float temperatura;
    private float wilgotność;

    public WarunkiBieżąceWyświetl(Observable observable) {
        this.observable = observable;
        observable.addObserver(this);
    }

    public void update(Observable obs, Object arg) {
        if (obs instanceof DanePogodowe) {
            DanePogodowe danePogodowe = (DanePogodowe)obs;
            this.temperatura = danePogodowe.pobierzTemperaturę();
            this.wilgotność = danePogodowe. pobierzWilgotność();
            wyświetl();
        }
    }

        public void wyświetl() {
        System.out.println("Warunki bieżące " + temperatura
            + "stopni C oraz " + wilgotność + % wilgotność");
    }
}
```

3. Konstruktor naszej klasy pobiera obecnie obiekt klasy Observable, a my wykorzystujemy to do wpisania obiektu opisującego warunki bieżące na listę obserwatorów.
4. Zmodyfikowaliśmy metodę update() tak, aby mogła pobierać jako argumenty zarówno obiekt klasy Observable, jak i opcjonalny obiekt danych.
5. W przypadku metody update() najpierw musimy przekonać się, że obiekt obserwowany jest typu DanePogodowe, a następnie możemy wykorzystać jego metody służące do pobierania danych do uzyskania wartości pomiarów temperatury otoczenia i wilgotności. Na zakończenie wywołujemy metodę wyświetl().





Magnesiki z kodem

Jak widać, definicja klasy PrognozaWyświetl znajduje się w kompletnej rozsypce. Czy możesz poukładać poszczególne fragmenty kodu w odpowiedniej kolejności tak, aby całość znów mogła poprawnie działać? A przy okazji, niektóre z nawiasów klamrowych pospadały nam przy okazji na ziemię, a są zdecydowanie za małe, aby podejmować wysiłek ich podnoszenia, więc możesz swobodnie dopisać tyle nawiasów klamrowych, ile będzie potrzeba!

```
public PrognozaWyświetl(Observable
observable) {
```

```
wyświetl();
```

```
observable.addObserver(this);
```

```
if (observable instanceof DanePogodowe) {
```

```
public class PrognozaWyświetl implements
Observer, WyświetlElement {
```

```
public void wyświetl() {
// kod metody wyświetl()
```

```
ostatnieCiśnienie = bieżąceCiśnienie;
bieżąceCiśnienie = danePogodowe.pobierzCiśnienie();
```

```
private float bieżąceCiśnienie = 1010.1f;
private float ostatnieCiśnienie;
```

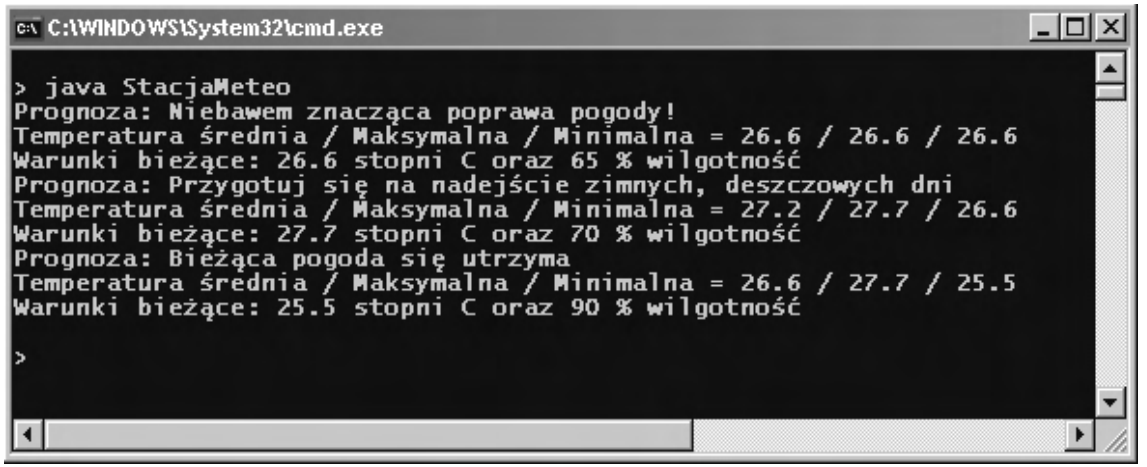
```
DanePogodowe danePogodowe =
(DanePogodowe) observable;
```

```
public void update(Observable observable,
        Object arg) {
```

```
import java.util.Observable;
import java.util.Observer;
```

Uruchamianie nowego kodu

Aby się upewnić, że wszystko działa, jak należy, spróbujmy uruchomić nową wersję naszej aplikacji...



Hmmm, czy zauważyłeś tutaj jakąś różnicę? Nie? Spójrz zatem jeszcze raz...

Zobaczysz, że o ile wyniki obliczeń niczym się od siebie nie różnią, o tyle (i nieco tajemniczo) kolejność wyświetlania poszczególnych wierszy jest zupełnie inna. Dlaczego tak się stało? Pomyśl przez chwilę, zanim przeczytasz odpowiedź poniżej…

Nigdy nie polegaj na oszacowaniu kolejności wykonywania poszczególnych powiadomień

Klasa java.util.Observable posiada zaimplementowaną metodę notifyObservers(), w której poszczególni obserwatorzy są powiadamiani w *zupełnie innej* kolejności, niż to miało miejsce w naszej własnej implementacji. Kto zatem ma rację? Otóż nikt; po prostu, my wybraliśmy inny sposób implementacji.

Problem mógłby się jednak pojawić w sytuacji, kiedy kod programu *zależałby* w istotny sposób od określonej kolejności powiadomień. Dlaczego? Ano dlatego, że jeżeli musiałbyś zmienić implementację klasy Observable i (lub) interfejsu Observer, kolejność powiadomień mogłaby się również zmienić, a Twoja aplikacja najzwyczajniej w świecie zaczęłaby generować nieprawidłowe rezultaty. Oczywiście, taka implementacja jest jak najbardziej *daleka* od tego, co określamy mianem „luźnych powiązań”.



Ciemna strona klasy java.util.Observable

Tak, to bardzo dobre pytanie. Jak zauważyłeś, Observable jest *klasą*, a nie *interfejsem*. Co gorsza, klasa ta nawet nie posiada *zaimplementowanego* interfejsu. Niestety, tak się jednak złożyło, że implementacja klasy java.util.Observable jest obciążona szeregiem problemów, które znacznie ograniczają jej użyteczność i możliwości zastosowania. Nie mówimy tutaj, że jest ona całkowicie bezużyteczna, stwierdzamy jedynie, że posiada kilka poważnych dziur i wybojów, na które po prostu trzeba uważać.

Observable jest klasą

Dzięki przedstawionym wcześniej regułom projektowania już wiesz, że to nie jest dobre rozwiązanie, ale tak właściwie, jakie zagrożenia to ze sobą niesie?

Po pierwsze, ponieważ Observable jest *klasą*, to aby z niej skorzystać, musisz stworzyć jej *klasy podrzędne*. Oznacza to w praktyce, że nie możesz dodać zachowań klasy Observable do już istniejącej klasy, która dziedziczy po innej klasie nadrzędnej. Taka cecha zdecydowanie ogranicza możliwości wielokrotnego wykorzystywania tej klasy (i czy przypadkiem nie dlatego zdecydowaliśmy się na korzystanie z wzorców projektowych?).

Po drugie, ponieważ nie istnieje interfejs Observable, nie możesz nawet zbudować swojej własnej implementacji, która będzie dobrze współpracowała z wbudowanym API interfejsu Observer języka Java. Nie masz również możliwości wymiany pakietu java.util na inny (powiedzmy, na taki, który potrafi obsługiwać wielowątkowość).

Klasa Observable chroni swoje kluczowe metody

Jeżeli przyjrzysz się API wzorca Observer, przekonasz się, że metoda setChanged() jest chroniona. I co z tego? No cóż, oznacza to mniej więcej tyle, że nie możesz wywołać metody setChanged() dopóty, dopóki nie dokonasz dziedziczenia z superklasy Observable. Oznacza to również, że nie możesz nawet stworzyć obiektu klasy Observable i wyposażyć go w swoje własne obiekty, po prostu *jesteś skazany* na dziedziczenie. Struktura klasy Observable stanowi zatem pogwałcenie również drugiej reguły projektowania… *Przedkładaj kompozycję nad dziedziczenie.*

Co zatem robić?

Klasa Observable może, mimo wszystko, odpowiadać na Twoje potrzeby, jeżeli możesz dziedziczyć z klasy java.util.Observable. Z drugiej jednak strony, zawsze możesz zdecydować się na stworzenie swojej własnej implementacji, tak jak pokazaliśmy na początku niniejszego rozdziału. Niezależnie od tego, na jakie rozwiązanie się zdecydujesz, znasz już dobrze wzorzec Obserwator i dzięki temu masz dobre podstawy do pracy z dowolnym API, które go wykorzystuje.

Inne miejsca w JDK, w których możesz znaleźć wzorzec Obserwator

Jeżeli jesteś ciekawy, jak wygląda implementacja wzorca Obserwator w JavaBeans, zainteresuj się bliżej interfejsem PropertyChangeListener.

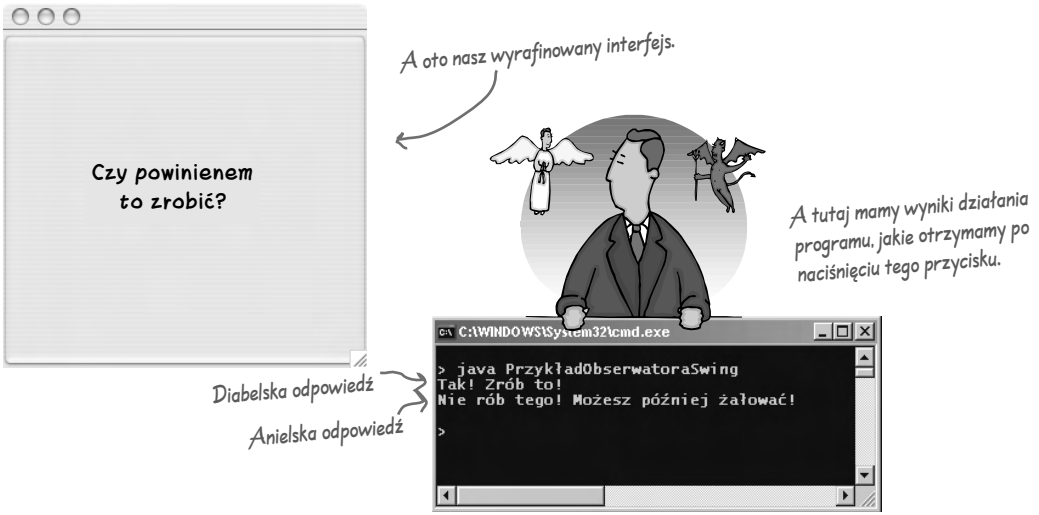
java.util.Observer oraz java.util.Observable nie są jedynymi miejscami w JDK (ang. Java Development Kit), w których możesz odnaleźć implementację wzorca Observer; zarówno komponenty JavaBean, jak i Swing mogą się pochwalić swoimi własnymi implementacjami tego wzorca. Na tym etapie wiedzy, na jakim obecnie się znajdujesz, możesz spokojnie pokusić się o samodzielną analizę wymienionych API; ale teraz, mimo wszystko, szybko zaprezentujemy przykład zastosowania biblioteki Swing, ot, tak po prostu, dla przyjemności.

Tytułem wprowadzenia...

Rzućmy okiem na jeden z prostych elementów Swing API, JButton. Jeżeli zajrzysz „pod maskę" klasy nadrzędnej w stosunku do JButton, czyli superklasy AbstractButton, zobaczysz, że posiada ona wiele metod pozwalających na dodawanie i usuwanie metod obiektów nasłuchujących (inaczej słuchaczy). Wspomniane metody pozwalają na dodawanie i usuwanie obserwatorów — lub, jak je nazywają w pakiecie Swing, obiektów nasłuchujących — z listy obiektów, które „nasłuchują" różnego rodzaju zdarzeń mogących nastąpić w obiekcie będącym częścią biblioteki Swing (obiekt nasłuchiwany). Przykładowo, ActionListener pozwala na „nasłuchiwanie" dowolnego typu zdarzeń, jakie mogą wystąpić w związku z przyciskiem (np. naciśnięcie przycisku). W Swing API znajdziesz wiele różnych rodzajów obiektów nasłuchujących.

Mała aplikacja, która odmieni nasze życie

No dobrze, nasza aplikacja jest całkiem prosta. Posiada ona jeden przycisk, na którym figuruje pytanie „Czy powinienem to zrobić?"; kiedy klikniesz ten przycisk, słuchacze (obserwatorzy) muszą odpowiedzieć na to pytanie w taki sposób, w jaki będą chcieli. Zaimplementujemy dwa takie obiekty nasłuchujące, nazywane AniołSłuchacz oraz DiabełSłuchacz. Poniżej przedstawiono zachowanie naszej aplikacji:



A oto i sam kod...

Nasza zmieniająca-życie-na-lepsze aplikacja nie wymaga zbyt dużej ilości kodu. Wszystko, co musimy zrobić, to stworzyć obiekt JButton, dołożyć go do obiektu JFrame, a następnie skonfigurować nasze obiekty nasłuchujące. W przypadku obiektów nasłuchujących użyjemy klas wewnętrznych (ang. *inner classes*), co w bibliotece Swing jest dosyć rozpowszechnioną techniką programowania. Jeżeli nie masz doświadczenia w programowaniu klas wewnętrznych lub nie znasz biblioteki Swing, powinieneś zajrzeć do rozdziału „Historia bardzo graficzna" książki *Java. Rusz głową! Wydanie II*.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

public class PrzykładObserwatoraSwing {
    JFrame frame;

    public static void main(String[] args) {
        PrzykładObserwatoraSwing przykład  = new PrzykładObserwatoraSwing ();
        przykład.uruchom();
    }

    public void uruchom() {
        frame = new JFrame();
        JButton button = new JButton("Czy powinienem to zrobić?");
        button.addActionListener(new AniołSłuchacz());
        button.addActionListener(new DiabełSłuchacz());
        frame.getContentPane().add(BorderLayout.CENTER, button);

        // Parametry okienka
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(BorderLayout.CENTER, button);
        frame.setSize(300,300);
        frame.setVisible(true);
    }

    class AniołSłuchacz implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            System.out.println("Nie rób tego! Możesz później żałować!");
        }
    }

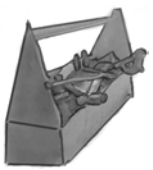
    class DiabełSłuchacz implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            System.out.println("Tak! Zrób to!");
        }
    }
}
```

Prosta aplikacja wykorzystująca bibliotekę Swing. Tworzy ona na ekranie ramkę i umieszcza wewnątrz niej przycisk.

Powoduje, że zarówno anielski, jak i diabelski obiekt staje się obiektem nasłuchującym (obserwatorem) zdarzeń związanych z przyciskiem.

Tutaj umieszczone zostały klasy obserwatorów, zdefiniowane jako klasy wewnętrzne (ale wcale tak być nie musi).

Kiedy zmienia się stan obiektu obserwowanego (w naszym przypadku jest to oczywiście przycisk), zamiast metody update() wywoływana jest metoda actionPerformed().



Twoja skrzynka narzędziowa

Witamy na końcu rozdziału 2. Możesz teraz z czystym sumieniem dodać do swojej skrzynki narzędziowej kilka nowych elementów...



CELNE SPOSTRZEŻENIA

- Wzorzec Obserwator definiuje pomiędzy obiektami relację jeden--do-wielu.
- Obiekty obserwowane (obiekty Podmiot) przesyłają powiadomienia i aktualizacje do obiektów obserwujących przy użyciu jednego, wspólnego interfejsu.
- Obiekty obserwujące są luźno powiązane z obiektem obserwowanym, co oznacza, że obiekt obserwowany wie o nich tylko tyle, że posiadają zaimplementowany interfejs Obserwator.
- Zgodnie z definicją wzorca informacje o zmianach stanu obiektu obserwowanego mogą być wysyłane lub pobierane (metoda „pobierania" jest uważana za bardziej politycznie poprawną).
- Nie można uzależniać poprawnego funkcjonowania aplikacji od określonej kolejności powiadamiania przesyłanego do obiektów obserwujących.
- Java posiada kilka różnych implementacji wzorca obserwator, włączając w to implementację klasy ogólnego przeznaczenia w pakiecie java.util.Observable.
- Należy zwracać baczną uwagę na ograniczenia związane z implementacją klasy java.util.Observable.
- Nie należy się obawiać tworzenia własnych implementacji klasy Observable.
- Biblioteka Swing bardzo mocno wykorzystuje implementację wzorca Obserwator; jest to cecha charakterystyczna niemal wszystkich pakietów GUI.
- Wzorzec Obserwator jest wykorzystywany w wielu innych miejscach, jak np. JavaBeans czy RMI.





Reguły projektowania — wyzwanie

Napisz, w jaki sposób wzorzec Obserwator wykorzystuje poszczególne reguły projektowania.

Reguła projektowania

Dokonaj identyfikacji wszystkich zmiennych elementów Twojej aplikacji, a następnie oddziel je od elementów, które przez cały czas pozostają stałe.

Reguła projektowania

Skoncentruj się na tworzeniu interfejsów, a nie implementacji

Reguła projektowania

Przedkładaj kompozycję nad dziedziczenie.

Ten przypadek może sprawić nieco problemów, zatem wskazówka: zastanów się, w jaki sposób obiekty obserwujące współpracują z obiektem obserwowanym.



Rozwiązania



Zaostrz ołówek

Opierając się na pierwszej implementacji naszego systemu, określ, które z wymienionych zdarzeń są prawdziwe. (Zaznacz wszystkie poprawne odpowiedzi).

- ✓ A. Tworzymy poszczególne implementacje, a nie interfejsy.
- ✓ B. Dodanie nowego trybu wyświetlania będzie każdorazowo wymuszało modyfikację kodu programu.
- ✓ C. Nie mamy żadnych możliwości dodawania (lub usuwania) wybranych trybów wyświetlania podczas działania programu.
- D. Poszczególne wyświetlane elementy nie posiadają wspólnego interfejsu.
- ✓ E. Nie dokonaliśmy hermetyzacji tych elementów aplikacji, które się zmieniają.
- F. Naruszyliśmy hermetyzację klasy PogodaDane.



Reguły projektowania — wyzwanie

Reguła projektowania

Dokonaj identyfikacji wszystkich zmiennych elementów Twojej aplikacji, a następnie oddziel je od elementów, które przez cały czas pozostają stałe.

Elementami, które we wzorcu Obserwator podlegają zmianom, są: stan obiektu obserwowanego oraz ilość i rodzaj obiektów obserwujących. Zgodnie z założeniami wzorca możesz modyfikować obiekty, które są zależne od stanu obiektu obserwowanego bez konieczności modyfikacji samego obiektu obserwowanego. To jest to, co nazywamy planowaniem z wyprzedzeniem!

Reguła projektowania

Skoncentruj się na tworzeniu interfejsów, a nie implementacji

Zarówno obiekt obserwowany Podmiot, jak i obiekty obserwujące wykorzystują interfejsy. Obiekt obserwowany śledzi wszystkie obiekty, które mają zaimplementowany interfejs Obserwator, podczas gdy obiekty obserwujące do rejestracji, usuwania z listy zarejestrowanych obiektów oraz do otrzymywania powiadomień wykorzystują interfejs Podmiot. Jak już zauważyliśmy, takie rozwiązanie pozwala na utrzymanie „zgrabnej", luźno związanej struktury obiektów.

Reguła projektowania

Przedkładaj kompozycję nad dziedziczenie.

Wzorzec Obserwator wykorzystuje zasadę kompozycji do tworzenia dowolnej liczby obiektów obserwujących powiązanych z danym obiektem obserwowanym. Takie relacje nie są w żaden sposób tworzone poprzez mechanizm dziedziczenia — są za to tworzone całkowicie poprzez kompozycję i to w trakcie działania programu!



Rozwiązania



Magnesiki z kodem

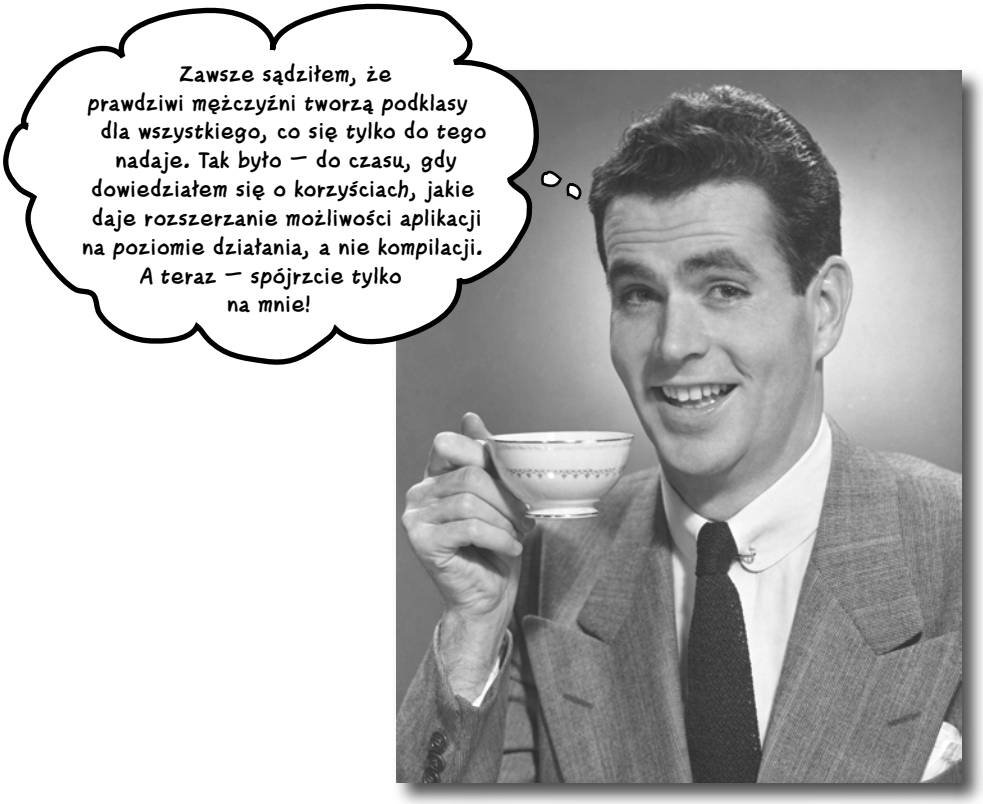
```
import java.util.Observable;
import java.util.Observer;

public class PrognozaWyświetl implements
Observer, WyświetlElement {

    private float bieżąceCiśnienie = 1010.1f;
    private float ostatnieCiśnienie;

    public PrognozaWyświetl(Observable
    observable) {
        DanePogodowe danePogodowe =
        (DanePogodowe) observable;
        observable.addObserver(this);
    }
    public void update(Observable observable,
            Object arg) {
        if (observable instanceof PogodaDane) {
            ostatnieCiśnienie = bieżąceCiśnienie;
            bieżąceCiśnienie = pogodaDane.pobierzCiśnienie();
            wyświetl();
        }
    }
    public void wyświetl() {
    // kod metody wyświetl()
    }
}
```

Dekorowanie zachowania obiektów



W zasadzie niniejszy rozdział możemy równie dobrze zatytułować „Otwieranie oczu programistom z nadmiernymi skłonnościami do używania dziedziczenia".
W tym rozdziale spróbujemy krytycznie przyjrzeć się zwyczajowym skłonnościom do nadużywania mechanizmu dziedziczenia oraz nauczymy Cię sposobów dekorowania zachowania klas w czasie działania programu przy użyciu pewnej formy kompozycji obiektów. Dlaczego? Po zapoznaniu się z technikami dekoracji zachowania klas będziesz mógł wyposażać swoje (i nie tylko) obiekty w nowe możliwości *bez konieczności dokonywania jakichkolwiek modyfikacji w kodzie klas podstawowych.*

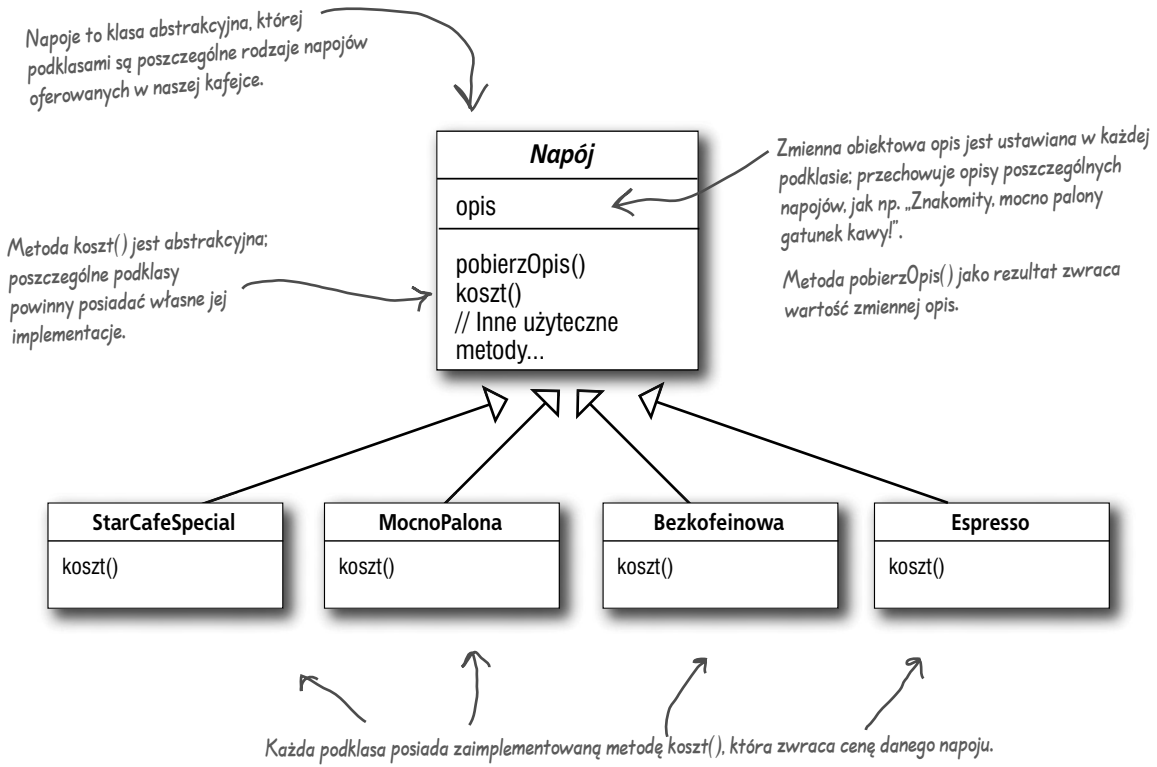
Witamy w „Star Café"



„Star Café" jest jedną z najszybciej rozwijających się sieci kawiarni*. Jeżeli przez okno widzisz jedną z nich, możesz być całkowicie pewny, że gdy wyjrzysz za róg najbliższego skrzyżowania, zobaczysz kolejną.

Ponieważ rozwijają się tak szybko, ostatnio niemal rozpaczliwie walczą z tym, aby kolejne aktualizacje ich systemu zamówień nadążały za ciągle rozszerzającą się paletą oferowanych napojów.

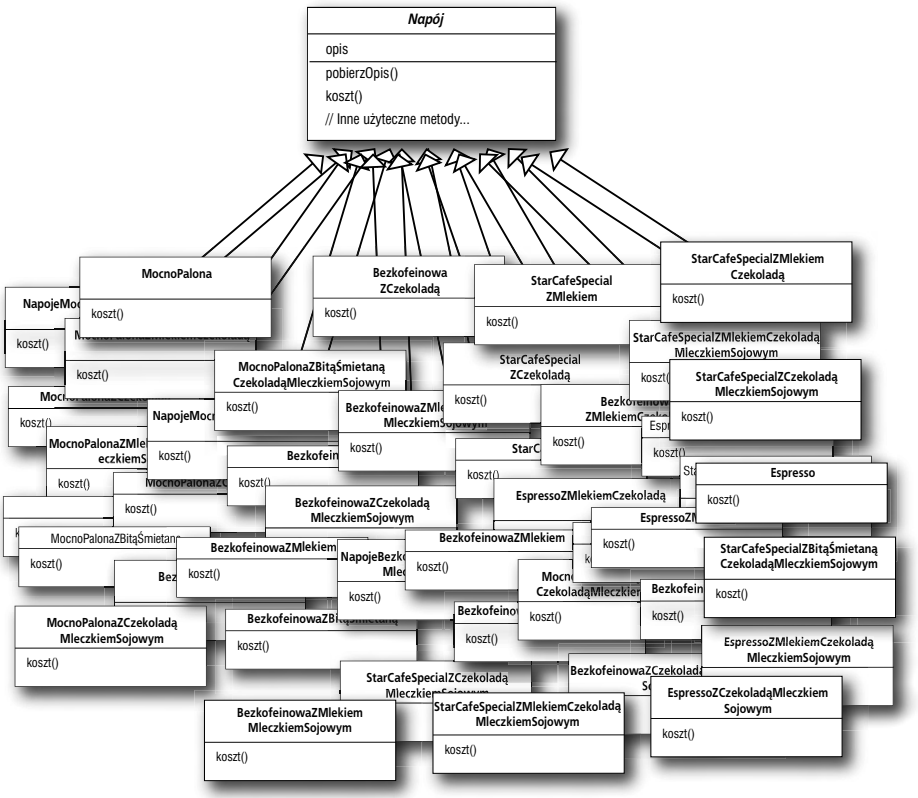
Kiedy pierwsza kawiarenka „Star Café" otwarła swe podwoje dla klientów, diagram klas ich systemu zamówień wyglądał mniej więcej tak:



* Oczywiście, jest to sieć fikcyjna, wszelkie podobieństwo do jakiejkolwiek rzeczywistej instytucji jest całkowicie przypadkowe i niezamierzone — *przyp. tłum.*

Do każdej kawy możesz również zamówić szereg dodatków, takich jak mleko, mleczko sojowe, mocha* (znana również pod nazwą „czekolada"). Możesz pokryć to wszystko bitą śmietaną. „Star Café" za każdy z tych dodatków dolicza niewielką opłatę, więc, chcąc nie chcąc, właściciele sieci musieli informacje o dodatkach (i ich możliwych kombinacjach) wbudować w swój system zamówień.

Ich pierwsza próba wyglądała mniej więcej tak...



Poszczególne metody koszt() obliczają cenę danego napoju przy uwzględnieniu odpowiedniej kawy oraz wszystkich zamówionych dodatków.



* Pod nazwą Mocha (lub Arabica Mocha) kryje się również znakomity gatunek kawy arabskiej, odznaczający się wybitnie czekoladowym aromatem i średnią mocą napoju; polecam! — *przyp. tłum.*



WYTĘŻ UMYSŁ

Chyba dla wszystkich jest oczywiste, że projektanci ze „Star Café" sami sobie narobili kłopotów, tworząc strukturę aplikacji, która z punktu widzenia serwisowania, modyfikacji i rozbudowy jest po prostu koszmarnym snem szalonego programisty. Wyobraź sobie tylko, co oni powinni zrobić, jeśli cena mleka pójdzie w górę? Albo co mają zrobić, jeśli firma wprowadzi nową polewę karmelową?

Spróbuj pomyśleć nieco szerzej, nie tylko o problemie serwisowania aplikacji — które spośród omawianych do tej pory reguł projektowania zostały pogwałcone podczas tworzenia tego projektu?

Wskazówka: dwie z tych reguł zostały tutaj kompletnie podeptane!



Przecież to głupie — po co nam te wszystkie klasy? Czy nie możemy po prostu użyć zmiennych obiektowych oraz mechanizmu dziedziczenia z klasy nadrzędnej do obsługi wszystkich dodatków?

No dobrze, zatem spróbujmy tak zrobić. Projektowanie rozpoczniemy z naszej superklasy bazowej Napój, do której dodamy zmienne obiektowe odpowiadające temu, czy dany napój będzie z mlekiem, czy też nie; czy będzie z czekoladą, czy nie itd.



Nowe zmienne typu boolean dla każdego z dodatków.

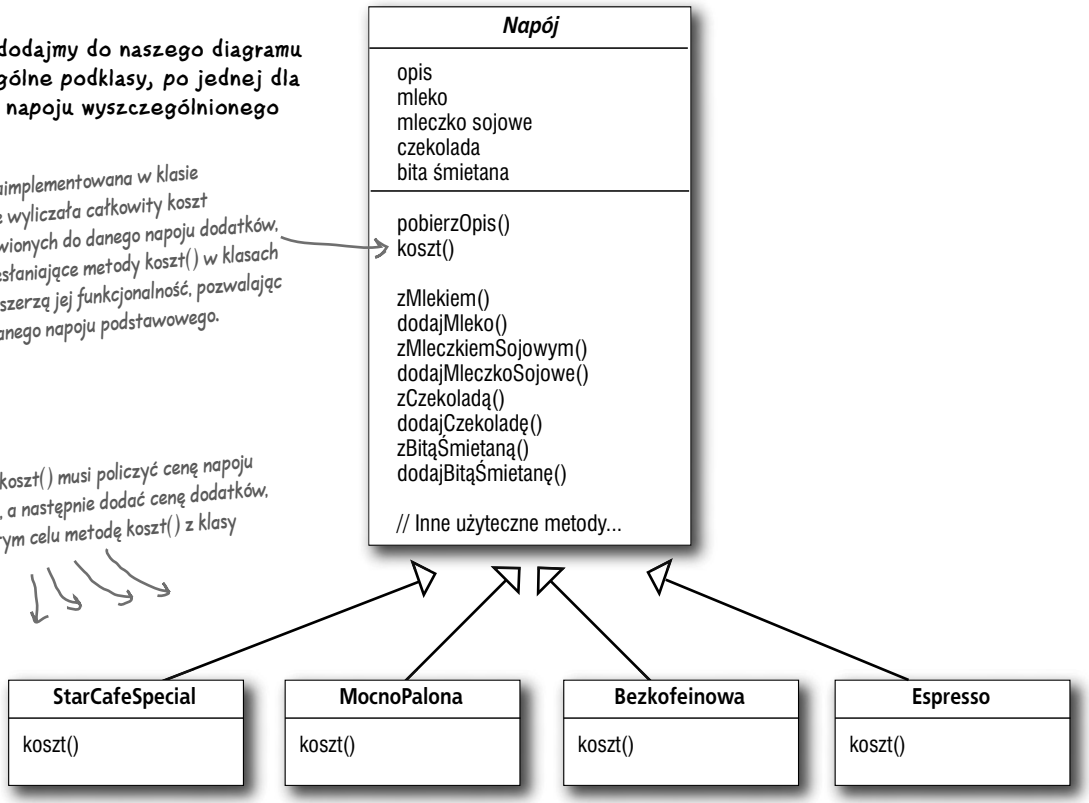
Teraz zaimplementujemy metodę koszt() bezpośrednio w klasie Napój (do tej pory była to metoda abstrakcyjna), tak że będzie mogła policzyć cenę powiązaną z dodatkami dla danego obiektu klasy Napój. Klasy podrzędne będą nadal miały własne wersje metody koszt() (czyli będą przesłaniały metodę koszt()), aczkolwiek będą również wywoływały tę metodę bezpośrednio z klasy nadrzędnej (czyli superwersję tej metody) tak, aby mogły policzyć cenę napoju podstawowego plus koszty wszystkich zamówionych do niego dodatków.

Te metody pobierają oraz ustawiają wartości zmiennych logicznych dla poszczególnych dodatków.

A teraz dodajmy do naszego diagramu poszczególne podklasy, po jednej dla każdego napoju wyszczególnionego w menu:

Metoda koszt() zaimplementowana w klasie nadrzędnej będzie wyliczała całkowity koszt wszystkich zamówionych do danego napoju dodatków, podczas gdy przesłaniające metody koszt() w klasach podrzędnych rozszerzą jej funkcjonalność, pozwalając dołączyć cenę danego napoju podstawowego.

Każda metoda koszt() musi policzyć cenę napoju podstawowego, a następnie dodać cenę dodatków, wywołując w tym celu metodę koszt() z klasy nadrzędnej.



Zaostrz ołówek

Spróbuj napisać kod implementujący metodę koszt() dla następujących klas (możesz użyć w zapisie pseudokodu Java):

```
public class Napój {
  public double koszt() {




  }
}
```

```
public class MocnoPalona extends Napój {
  public MocnoPalona() {
    opis = "Znakomity, mocno palony gatunek
kawy!"
  }
  public double koszt() {



  }
}
```



Zobaczmy, w sumie wyszło z tego pięć klas. Czyli definitywnie jest to kierunek, w którym powinniśmy się dalej poruszać.

Nie jestem tego taki pewny; jeżeli sobie pomyślę, jakie zmiany może przechodzić ten projekt w przyszłości, przewiduję kilka potencjalnych problemów.





Zaostrz ołówek

Jakie wymagania (bądź inne czynniki) mogą się zmieniać tak, że będą miały wpływ na nasz projekt?

Zmiana cen poszczególnych dodatków będzie nas zmuszała do modyfikacji istniejącego kodu.

Wprowadzanie do oferty nowych dodatków będzie nas zmuszało do dodawania nowych metod oraz modyfikacji kodu metody koszt() w klasie nadrzędnej.

Może się okazać, że w ofercie pojawią się nowe napoje. Dla niektórych napojów (na przykład dla mrożonej herbaty) pewne dodatki mogą być nieodpowiednie, aczkolwiek mimo wszystko podklasa Herbata powinna dziedziczyć takie metody, jak np. zBitąŚmietaną().

A co w sytuacji, kiedy klient zażyczy sobie podwójną porcję czekolady?

Jak już przekonaliśmy się w rozdziale 1., to naprawdę nie jest dobry pomysł!

Twoja kolej:



Mistrz i uczeń...

Mistrz: *Od naszego ostatniego spotkania upłynęło już całkiem sporo czasu, młody człowieku... Czy spędzałeś dużo czasu pogrążony głęboko w rozważaniach nad istotą dziedziczenia?*

Uczeń: *Tak, Mistrzu. Dziedziczenie jest zaiste potężną bronią, nauczyłem się jednak, że nie zawsze jej użycie prowadzi do wystarczająco elastycznych i łatwych w modyfikacji projektów.*

Mistrz: *O, co za niespodzianka! Poczyniłeś, widzę, pewne postępy! A zatem powiedz mi, mój młody adepcie, w jaki sposób możemy uzyskać kod, którego da się używać wielokrotnie, inaczej niż poprzez dziedziczenie?*

Uczeń: *Mistrzu, nauczyłem się, że istnieją sposoby, które podczas działania programu dają takie same efekty, jakie normalnie daje dziedziczenie; należy zatem skorzystać z odpowiedniej kompozycji i delegacji.*

Mistrz: *Proszę, kontynuuj...*

Uczeń: *Kiedy dziedziczę dane zachowania poprzez tworzenie klasy podrzędnej, takie zachowanie jest tworzone statycznie podczas kompilacji programu. Co więcej, wszystkie klasy podrzędne muszą dziedziczyć te same zachowania. Jeżeli jednak mogę rozszerzyć zakres zachowań obiektu poprzez kompozycję, takiej operacji mogę dokonać dynamicznie podczas działania programu.*

Mistrz: *Bardzo dobrze, mój młody adepcie, jesteś już bliski ujrzenia potęgi i mocy kompozycji.*

Uczeń: *Tak, Mistrzu. Obecnie mogę już przy użyciu tej techniki dodawać nowe zadania do poszczególnych obiektów, włączając w to zadania, o których nawet się nie śniło projektantom klasy nadrzędnej. Co więcej, nie muszę w żaden sposób modyfikować ich kodu!*

Mistrz: *A czego nauczyłeś się o wpływie, jaki zastosowanie kompozycji wywiera na możliwość zarządzania i modyfikacji kodu programu?*

Uczeń: *To jest właśnie to, do czego zmierzałem, Mistrzu. Wykorzystując dynamiczną kompozycję obiektów, mogę im nadawać nowe cechy poprzez proste tworzenie nowego kodu, a nie modyfikację kodu istniejącego. Zatem, ponieważ nie będę modyfikował istniejącego kodu, szanse na wprowadzenie nowego błędu czy też zaistnienie nieoczekiwanych efektów ubocznych zdecydowanie maleją.*

Mistrz: *Bardzo dobrze. Wystarczy na dzisiaj, młody człowieku. Teraz chciałbym, abyś się oddalił i oddał dalszym medytacjom i rozważaniom na ten temat... Pamiętaj, kod powinien być tak zamknięty (na zmiany), jak kwiat lotosu o zmierzchu, ale jednocześnie tak otwarty (na rozbudowę), jak kwiat lotosu o świcie.*

Reguła otwarte-zamknięte

Nasz młody adept sztuki projektowania jest teraz na najlepszej drodze do poznania jednej z najważniejszych reguł projektowania:



Reguła projektowania

Klasy powinny być otwarte na rozbudowę, ale zamknięte na modyfikacje.



Wejdź proszę, jest *otwarte*. Możesz bez skrępowania rozbudowywać nasze klasy dowolnymi nowymi zachowaniami. Jeżeli Twoje potrzeby lub wymagania ulegną zmianie (a my wiemy, że wcześniej czy później tak właśnie się stanie), po prostu idź do przodu i dokonuj rozbudowy wedle własnego uznania.



Przepraszamy, *zamknięte*. To prawda, spędzamy masę czasu, pracując nad tym, aby kod aplikacji był stabilny i wolny od błędów, więc nie możemy pozwolić Ci go — ot, tak sobie — modyfikować. Kod musi, niestety, pozostać zamknięty na wszelkie modyfikacje. Jeżeli Ci się to nie podoba, możesz porozmawiać z przełożonym.

Naszym celem jest umożliwienie łatwej rozbudowy poszczególnych klas poprzez dodawanie im nowych zachowań, ale bez konieczności modyfikacji ich istniejącego kodu. Co otrzymamy w zamian za nasze wysiłki po osiągnięciu tego celu? Struktury, które będą się łatwo adaptowały do zmian i będą wystarczająco elastyczne, aby przyjąć nowe zachowania spełniające nowe wymagania.

Nie istnieją głupie pytania

P: Otwarte na rozbudowę i zamknięte na modyfikacje? To wygląda na swego rodzaju sprzeczność. W jaki sposób można osiągnąć taką strukturę?

O: To jest bardzo dobre pytanie. Faktycznie, na pierwszy rzut oka wygląda to na sprzeczność. W końcu, jak by na to nie patrzeć, im mniejsze są możliwości modyfikacji danego czegoś, tym trudniej jest to coś rozbudować, prawda?

Jak się jednak okazuje, w programowaniu zorientowanym obiektowo istnieje kilka pomysłowych technik pozwalających na rozbudowę istniejących systemów nawet w sytuacji, kiedy nie możemy modyfikować zasadniczego kodu aplikacji. Przypomnij sobie wzorzec Obserwator (o którym mówiliśmy w rozdziale 2.)... Dodając nowe obiekty obserwujące, możemy rozbudować obiekt obserwowany bez konieczności dodawania nowego kodu do tego ostatniego. Niebawem poznasz kilka nowych sposobów rozbudowy czy też dodawania kolejnych zachowań przy użyciu innych technik programowania zorientowanego obiektowo.

P: No dobrze, rozumiem zasady funkcjonowania wzorca Obserwator, ale w jaki sposób, ogólnie mówiąc, można zaprojektować coś, co będzie podatne na rozbudowę, ale jednocześnie zamknięte na modyfikacje?

O: Wiele wzorców projektowych daje nam już wielokrotnie sprawdzone w praktyce rozwiązania, które pozwalają na ochronę kodu aplikacji przed modyfikacją przy jednoczesnej rozbudowie jej funkcjonalności. W niniejszym rozdziale znajdziesz dobry przykład zastosowania wzorca Dekorator przy jednoczesnym postępowaniu zgodnie z regułą otwarte-zamknięte.

P: W jaki sposób mogę stworzyć strukturę (aplikacji), w której każdy z elementów będzie zgodny z regułą otwarte-zamknięte?

O: Zazwyczaj nie da się tego zrobić. Tworzenie elastycznych i otwartych na rozbudowę aplikacji zorientowanych obiektowo bez modyfikacji istniejącego kodu wymaga poświęcenia dodatkowego czasu i zasobów. Ogólnie rzecz biorąc, zazwyczaj nie możemy sobie pozwolić na takie zaprojektowanie każdego elementu naszej aplikacji (co więcej, prawdopodobnie byłaby to zwykła strata czasu, a więc i pieniędzy...). Projektowanie zgodnie z regułą otwarte-zamknięte zazwyczaj wymaga wprowadzenia nowych poziomów abstrakcji, co nieuchronnie prowadzi do powiększenia stopnia skomplikowania kodu naszej aplikacji. Salomonowym rozwiązaniem jest więc koncentrowanie się raczej na tych obszarach systemu, dla których prawdopodobieństwo wystąpienia znaczących zmian jest największe, i stosowanie reguł projektowania właśnie tam.

P: Będę wiedział, które obszary zmian są najważniejsze?

O: Częściowo jest to kwestia doświadczenia w projektowaniu systemów zorientowanych obiektowo, jak również — oczywiście — dogłębna znajomość dziedziny, dla której dany system powstaje. Szczegółowa analiza innych przykładów i rozwiązań z pewnością pomoże Ci w procesie nauki identyfikacji obszarów zmiennych, które będą występowały w Twoich aplikacjach.

Pomimo, iż może to pozornie wyglądać na sprzeczność, istnieją techniki programowania zorientowanego obiektowo pozwalające na rozbudowę funkcjonalności kodu bez konieczności jego bezpośredniej modyfikacji.

Wybierając obszary kodu aplikacji, które powinny zostać rozbudowane, powinieneś zachować umiar i zdrowy rozsądek; postępowanie zgodnie z regułą otwarte–zamknięte WSZĘDZIE jest niepotrzebną stratą czasu, zasobów, zazwyczaj też pieniędzy, a poza tym może prowadzić do powstania bardzo skomplikowanego i trudnego do zrozumienia kodu programu.

Spotkanie z wzorcem Dekorator

OK, mam dość tego klubu „Miłośników projektowania zorientowanego obiektowo". Mamy tutaj poważny, rzeczywisty problem! Pamiętacie o nas? „Star Cafè"? Czy Wy naprawdę sądzicie, że któraś z tych Waszych reguł projektowania może nam w czymś pomóc?



Jak się niedawno przekonaliśmy, próba odwzorowania systemu zamówień napojów wraz z różnymi dodatkami przy użyciu mechanizmów dziedziczenia nie dała zbyt dobrych rezultatów — występowała „eksplozja klas", otrzymywaliśmy sztywny projekt, efektem były też próby rozbudowy klasy bazowej mechanizmami nieodpowiednimi dla niektórych klas podrzędnych.

Zamiast tego w chwili obecnej zrobimy coś zupełnie innego: rozpoczniemy od przygotowania napoju podstawowego. Dodatkami „udekorujemy" go dopiero podczas działania programu. Przykładowo, jeżeli klient zażyczy sobie czarną, mocno paloną kawę z czekoladą i bitą śmietaną, będziemy postępować następująco:

1. **Weź obiekt MocnoPalona.**
2. **„Udekoruj" go obiektem Czekolada.**
3. **„Udekoruj" go obiektem „BitaŚmietana".**
4. **Wywołaj metodę koszt() i przy użyciu mechanizmu delegacji dodaj koszty dodatków.**

No dobrze, ale w jaki sposób mogę „dekorować" poszczególne obiekty i w jaki sposób zastosować tutaj delegację? Zobaczmy zatem, jak to działa.

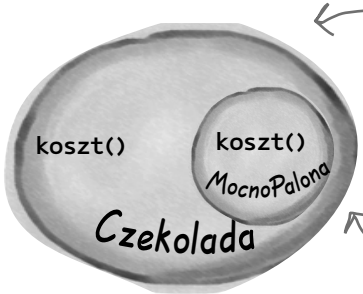
Konstruowanie zamówienia przy użyciu Dekoratorów

❶ **Rozpoczynamy od naszego obiektu MocnoPalona.**



Pamiętaj, że obiekt MocnoPalona dziedziczy zachowania z klasy Napój i posiada zaimplementowaną metodę koszt(), która wylicza koszt napoju.

❷ **Klient zażyczył sobie czekolady, więc tworzymy obiekt Czekolada i „zawijamy" go dookoła obiektu MocnoPalona.**



Obiekt Czekolada jest dekoratorem. Jego typ odzwierciedla typ obiektu dekorowanego, czyli w naszym przypadku obiektu klasy Napój (przez słowo „odzwierciedla" rozumiemy tutaj, że po prostu jest tego samego typu…).

Jak widać, Czekolada również posiada swoją metodę koszt() i dzięki własnościom polimorfizmu możemy traktować każdy napój, do którego dodamy Czekoladę, jako kolejny napój (ponieważ Czekolada jest podtypem klasy Napoje).

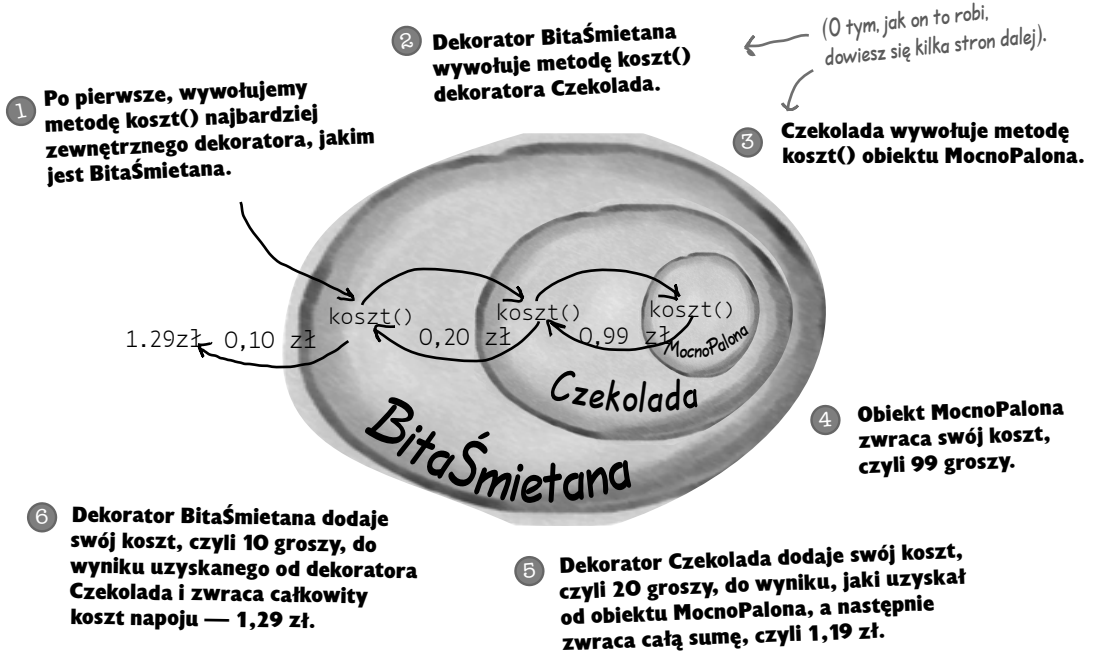
❸ **Klient życzy sobie również bitej śmietany, więc tworzymy oczywiście dekorator BitaŚmietana i „zawijamy" go dookoła obiektu Czekolada.**



BitaŚmietana jest dekoratorem, więc również odzwierciedla typ obiektu MocnoPalona i posiada zaimplementowaną metodę kosz().

Czyli obiekt MocnoPalona „zawinięty" w obiekty Czekolada i BitaŚmietana jest nadal napojem (obiektem klasy Napoje), więc możemy z nim zrobić dokładnie to samo, co z „czystym" obiektem MocnoPalona, włączając w to wywoływanie jego metody koszt().

4 A teraz nadszedł wreszcie czas na obliczenie kwoty, jaką powinien zapłacić klient za swoje zamówienie. Dokonamy tego poprzez wywołanie metody koszt() najbardziej zewnętrznego dekoratora, BitaŚmietana, który z kolei będzie delegował obliczanie kosztów do obiektów, które dekoruje. Po uzyskaniu wyników ich obliczeń dekorator ten dodaje swój koszt (bitej śmietany) i dzięki temu otrzymujemy całkowity koszt napoju.



Podsumujmy zatem, czego dowiedzieliśmy się do tej pory...

- Obiekty dekorujące są tego samego typu, co obiekty dekorowane.
- Jeden obiekt podstawowy może zostać „zawinięty" zarówno w jeden, jak i w większą ilość dekoratorów.
- Przy założeniu, że dekorator jest tego samego typu, co obiekt dekorowany, możemy przekazywać obiekt „owinięty" dekoratorem zamiast obiektu oryginalnego.
- Dekorator dodaje swoje własne zachowania przed delegowaniem do obiektu dekorowanego właściwego zadania i (lub) po nim.

Kluczowe zagadnienie!

- Obiekty mogą być dekorowane w dowolnym momencie, czyli możemy je również dekorować dynamicznie w czasie działania programu, używając do tego takiej liczby dekoratorów, jaka nam będzie potrzebna.

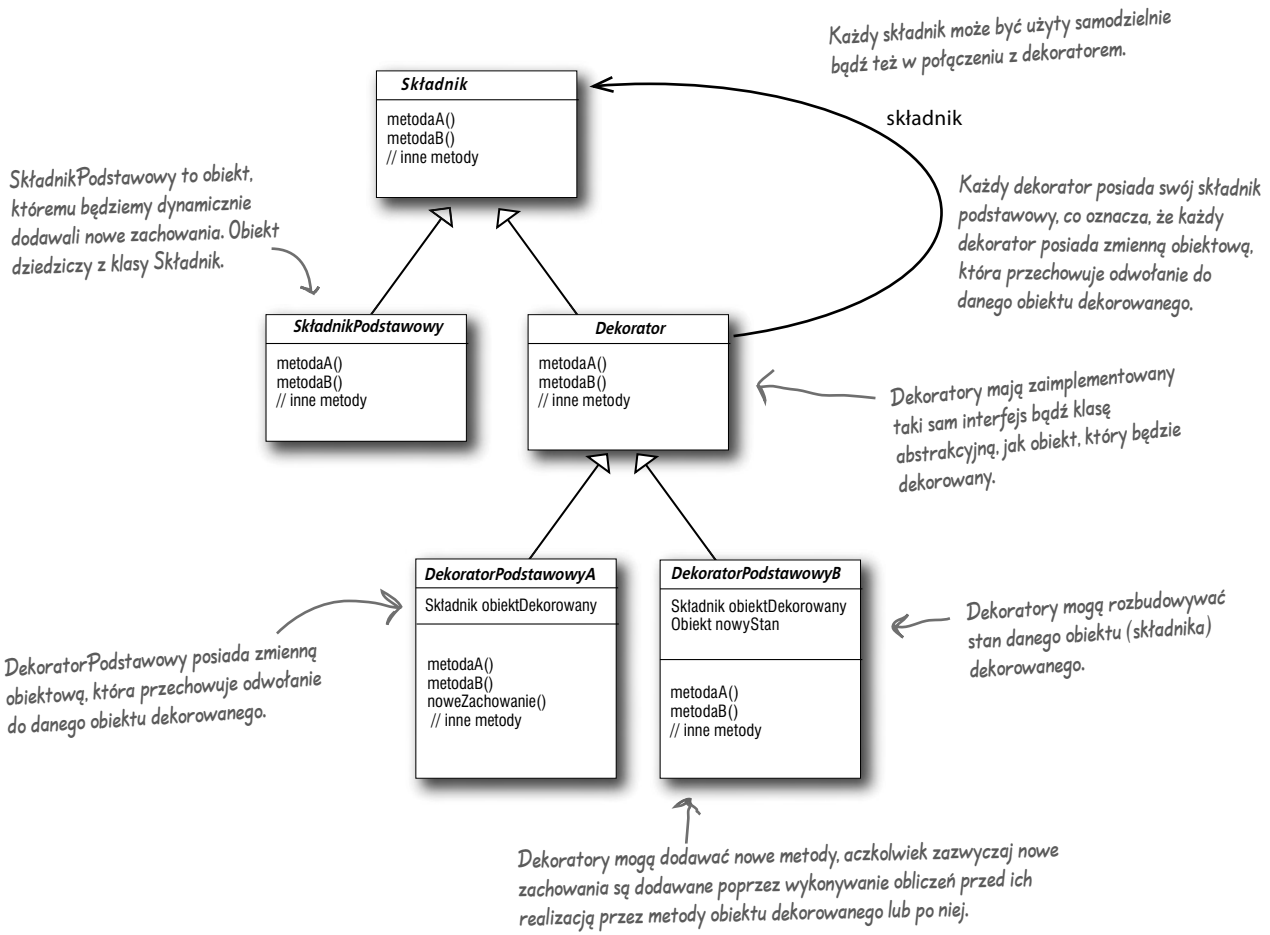
A teraz zobaczymy, jak to naprawdę działa — przyjrzymy się definicji wzorca Dekorator, a także napiszemy nieco kodu.

Definicja wzorca Dekorator

Przede wszystkim przyjrzyjmy się definicji wzorca Dekorator:

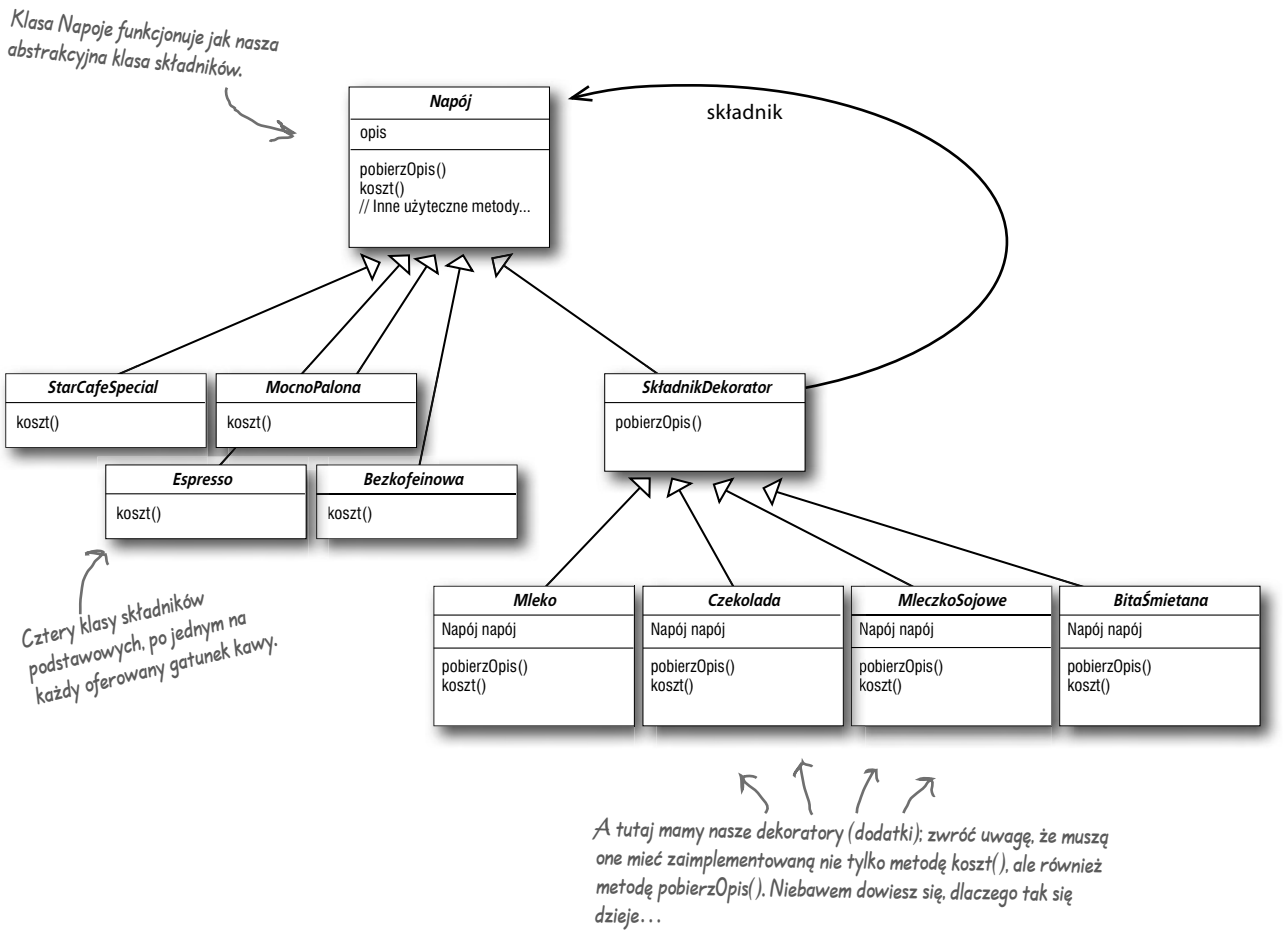
> **Wzorzec Dekorator** pozwala na dynamiczne przydzielanie danemu obiektowi nowych zachowań. Dekoratory dają elastyczność podobną do tej, jaką daje dziedziczenie, oferując jednak w zamian znacznie rozszerzoną funkcjonalność.

Powyższa definicja opisuje nam jedynie *przeznaczenie* wzorca Dekorator, ale nie daje nam niemal żadnych wskazówek, w jaki sposób moglibyśmy *zaimplementować* ten wzorzec w naszej aplikacji. Rzućmy okiem na diagram klas, który nieco rozjaśni nam całą sytuację (na następnej stronie zobaczymy tę samą strukturę, wykorzystaną do rozwiązania naszego problemu z napojami).



Dekorujemy nasze Napoje

Spróbujemy teraz wtłoczyć napoje firmy „Star Café” w ramy diagramu klas wzorca Dekorator...



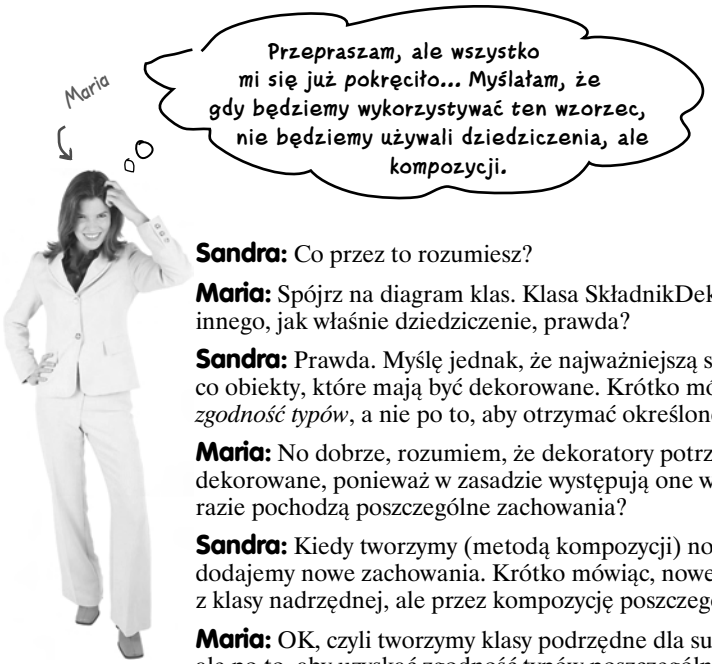


WYTĘŻ UMYSŁ

Zanim przejdziesz dalej, pomyśl, w jaki sposób zaimplementowałbyś metodę koszt() dla poszczególnych gatunków kawy oraz dodatków. Zastanów się również, jak zaimplementowałbyś metodę pobierzOpis() dla poszczególnych dodatków.

Zasłyszane w sąsiednim boksie...

Zamieszanie z Dziedziczeniem i Kompozycją



Sandra: Co przez to rozumiesz?

Maria: Spójrz na diagram klas. Klasa SkładnikDekorator jest rozszerzeniem klasy Napój. Przecież to nic innego, jak właśnie dziedziczenie, prawda?

Sandra: Prawda. Myślę jednak, że najważniejszą sprawą jest fakt, że dekoratory są tego samego typu co obiekty, które mają być dekorowane. Krótko mówiąc, używamy tutaj dziedziczenia po to, aby osiągnąć *zgodność typów*, a nie po to, aby otrzymać określone *zachowania*.

Maria: No dobrze, rozumiem, że dekoratory potrzebują tego samego „interfejsu" co składniki dekorowane, ponieważ w zasadzie występują one w miejscu rzeczywistych składników. Ale skąd w takim razie pochodzą poszczególne zachowania?

Sandra: Kiedy tworzymy (metodą kompozycji) nowy dekorator dla danego składnika, wtedy właśnie dodajemy nowe zachowania. Krótko mówiąc, nowe zachowania tworzone są nie poprzez dziedziczenie z klasy nadrzędnej, ale przez kompozycję poszczególnych obiektów.

Maria: OK, czyli tworzymy klasy podrzędne dla superklasy Napój nie po to, aby dziedziczyć zachowania, ale po to, aby uzyskać zgodność typów poszczególnych obiektów. Dane zachowania biorą się z kompozycji dekoratorów z poszczególnymi obiektami, jak również i z innymi dekoratorami.

Sandra: Zgadza się.

Maria: Ooooo, teraz to rozumiem. A ponieważ używamy kompozycji obiektów, dysponujemy o wiele większą elastycznością w łączeniu i dobieraniu dodatków z napojami. Sprytne.

Sandra: Tak to już jest, że jeśli polegamy tylko na mechanizmie dziedziczenia, wszystkie zachowania mogą być ustalane statycznie w czasie kompilacji programu. Innymi słowy, otrzymujemy wtedy wyłącznie zachowania, które daje klasa nadrzędna lub które zostają przesłonięte przez metody w klasach podrzędnych. Natomiast przy użyciu kompozycji możemy mieszać i dobierać dekoratory w niemal dowolny sposób… *i to dynamicznie, w trakcie działania programu.*

Maria: I jeżeli dobrze rozumiem, w dowolnym momencie możemy zaimplementować nowe dekoratory, które zapewnią nam nowe zachowania. Jeżeli polegalibyśmy tylko na mechanizmie dziedziczenia, musielibyśmy modyfikować istniejący kod za każdym razem, kiedy chcielibyśmy dołożyć nowe zachowania.

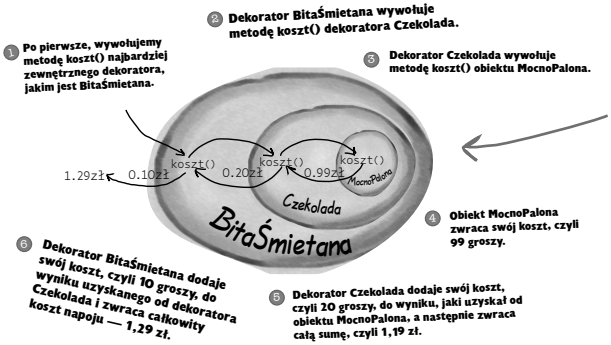
Sandra: Dokładnie tak.

Maria: Mam jeszcze tylko jedno pytanie. Jeżeli wszystko, czego potrzebujemy od dziedziczenia, to tylko prawidłowy typ składnika, dlaczego nie użyć zamiast abstrakcyjnej klasy Napój po prostu interfejsu?

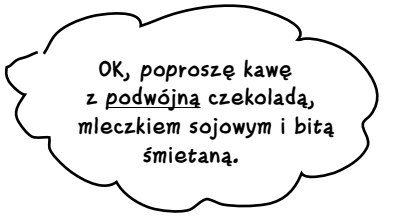
Sandra: Niby można, ale pamiętaj, że kiedy otrzymaliśmy ten kod od firmy „Star Café", abstrakcyjna klasa Napoje już była tam zdefiniowana. Tradycyjnie wzorzec Dekorator nie wymienia składnika abstrakcyjnego, ale w języku Java oczywiście moglibyśmy użyć interfejsu. Pamiętaj jednak, że zawsze staramy się uniknąć modyfikacji istniejącego kodu, więc nie starajmy się tego „naprawić" tak długo, jak klasa abstrakcyjna spełnia swoje zadanie.

Szkolenie nowego barmana

Narysuj schemat tego, co się dzieje, kiedy bar otrzymuje zamówienie na „kawę z podwójną czekoladą, mleczkiem sojowym i bitą śmietaną". Skorzystaj z cen podanych w menu na rysunku obok i narysuj schemat w takim samym formacie, jakiego używaliśmy kilka stron wcześniej:



Ten schemat przygotowaliśmy dla zamówienia na „mocno paloną kawę z czekoladą i bitą śmietaną".





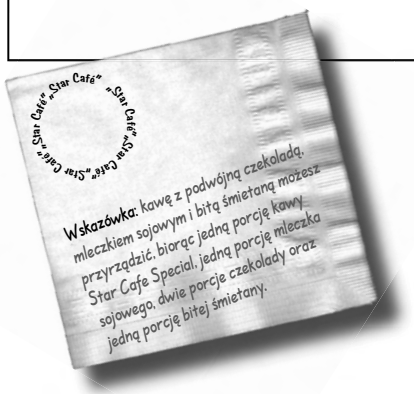
„Star Café"

Kawy	
Star Cafe Special	0,89 zł
Mocno Palona	0,99 zł
Bezkofeinowa	1,05 zł
Espresso	1,99 zł

Dodatki	
Mleko	10 gr
Czekolada	20 gr
Mleczko sojowe	15 gr
Bita śmietana	10 gr

Zaostrz ołówek

Schemat narysuj tutaj.



Tworzymy kod aplikacji „Star Café"



Nadszedł już czas, abyśmy zamienili nasz projekt aplikacji w rzeczywisty kod programu.

Rozpoczniemy od definicji klasy Napój, której kod nie będzie się w niczym różnił od oryginału otrzymanego od firmy „Star Café". Przypomnij sobie, jak on wygląda:

Klasa Napój jest klasą abstrakcyjną, która posiada dwie metody: pobierzOpis() oraz koszt()

```
public abstract class Napój {
    String opis = "Napój nieznany";

    public String pobierzOpis() {
        return opis;
    }

    public abstract double koszt();
}
```

Metoda pobierzOpis() została już wcześniej zaimplementowana, ale musimy jeszcze zaimplementować metodę koszt() w poszczególnych klasach podrzędnych.

Kod klasy Napój jest wystarczająco prosty. Teraz utworzymy kod klasy abstrakcyjnej SkładnikDekorator:

Przede wszystkim musimy zapewnić zgodność typów z klasą Napój, stąd nasza klasa będzie rozszerzeniem klasy Napój (klasa SkładnikDekorator dziedziczy po klasie Napój).

```
public abstract class SkładnikDekorator extends Napój {
    public abstract String pobierzOpis();
}
```

Będziemy również wymagali, aby poszczególne składniki dodatków (dekoratory) posiadały zaimplementowaną metodę pobierzOpis(). Dlaczego tak powinno być, przekonasz się już za chwilę...

Tworzenie kodu klas opisujących napoje

Mamy już gotowy kod naszych klas bazowych, więc nadszedł czas na implementację poszczególnych napojów. Rozpoczniemy od znanego wszystkim Espresso. Pamiętaj, że dla każdego napoju musimy utworzyć opis (czyli ustawić wartość zmiennej opis) oraz zaimplementować metodę koszt().

```
public class Espresso extends Napój {

    public Espresso() {
        opis = "Kawa Espresso";
    }

    public double koszt() {
        return 1.99;
    }
}
```

Ponieważ klasa Espresso dziedziczy po klasie Napój, sama jest niewątpliwie napojem.

Aby dołożyć opis do napoju, ustawiamy wartość zmiennej opis w konstruktorze tej klasy. Pamiętaj, że zmienna obiektowa opis jest dziedziczona z klasy nadrzędnej Napój.

W końcu musimy wyliczyć koszt kawy Espresso. Nie musimy się martwić o dodatki i ich koszty, więc po prostu zwracamy bezpośrednią cenę kawy Espresso, czyli w naszym przypadku 1,99 zł.

```
public class StarCafeSpecial extends Napój {
    public StarCafeSpecial() {
        opis = "Kawa Star Cafe Special";
    }

    public double koszt() {
        return 0.89;
    }
}
```

Tutaj mamy definicję kolejnego Napoju. Cała nasza praca sprowadza się do ustawienia wartości zmiennej opis („Kawa Star Cafe Special") oraz zwracania prawidłowej ceny napoju, czyli 0,89 zł.

Definicje pozostałych dwóch klas napojów (MocnoPalona oraz Bezkofeinowa) możesz utworzyć dokładnie w taki sam sposób.



Tworzenie kodu klas opisujących dodatki

Jeżeli przypomnisz sobie, jak wyglądał diagram klas wzorca Dekorator, zorientujesz się, że utworzyliśmy już nasz składnik abstrakcyjny (klasa Napój), mamy gotowe poszczególne składniki (klasy StarCafeSpecial i inne) oraz gotowy dekorator abstrakcyjny (klasa SkładnikDekorator). Nadszedł zatem czas na implementację poszczególnych dekoratorów. Oto definicja klasy Czekolada:

Czekolada jest dekoratorem, więc dziedziczy po klasie SkładnikDekorator.

Pamiętaj, klasa SkładnikiDekorator dziedziczy po klasie Napoje.

Mamy zamiar utworzyć obiekt klasy Czekolada, który będzie się odwoływał do obiektu klasy Napój, wykorzystując:

(1) Zmienną obiektową przechowującą dekorowany obiekt.

(2) Specyficzny sposób ustawiania tej zmiennej obiektowej na dekorowany obiekt. Tutaj mamy zamiar przekazywać obiekt reprezentujący dekorowany napój do konstruktora dekoratora.

```
public class Czekolada extends SkładnikDekorator {
    Napój napój;

    public Czekolada(Napój napój) {
        this.napój = napój;
    }

    public String pobierzOpis() {
        return napój.pobierzOpis() + ", Czekolada";
    }

    public double koszt() {
        return napój.koszt() + 0,20;
    }
}
```

Chcemy, aby opis obejmował nie tylko nazwę napoju — przykładowo „Mocno Palona" — ale również nazwy każdego dodatku użytego do dekorowania, przykładowo „Mocno Palona, Czekolada". Z tego względu najpierw delegujemy wykonanie metody pobierzOpis() do dekorowanego obiektu (co pozwala na pobranie jego opisu), a następnie dołączamy opis naszego dekoratora, „Czekolada", do uzyskanego wyniku.

Teraz musimy wyliczyć cenę napoju z dodatkiem czekolady. Najpierw delegujemy wykonanie metody koszt() do dekorowanego obiektu, tak aby uzyskać jego koszt, a następnie dodajemy koszt czekolady do uzyskanego rezultatu.

Na następnej stronie utworzymy wreszcie nasz napój i wzbogacimy go różnymi dodatkami (dekoratorami), ale najpierw...



Zaostrz ołówek

Napisz i skompiluj odpowiedni kod programu definiujący dodatki MleczkoSojowe oraz BitaŚmietana. Będziesz go potrzebował do uruchomienia i testowania całej aplikacji.

Podajemy kawę

Gratulacje. Nadszedł czas, aby spokojnie usiąść, zamówić kilka filiżanek aromatycznej kawy i w całej rozciągłości podziwiać elastyczność projektu, który stworzyliśmy przy wykorzystaniu wzorca Dekorator.

Oto prosty kod, który umożliwia przeprowadzenie testów* naszej aplikacji:

```
public class StarCafe {

    public static void main(String args[]) {
        Napój napój = new Espresso();
        System.out.println(napój.pobierzOpis() + "   " +
                + napój.koszt() + " zł");

        Napój napój2 = new MocnoPalona();
        napój2 = new Czekolada(napój2);
        napój2 = new Czekolada(napój2);
        napój2 = new BitaŚmietana(napój2);
        System.out.println(napój2.pobierzOpis() + "   " +
                + napój2.koszt() + " zł");

        Napój napój3 = new StarCafeSpecial();
        napój3 = new MleczkoSojowe(napój3);
        napój3 = new Czekolada(napój3);
        napój3 = new BitaŚmietana(napój3);
        System.out.println(napój3.pobierzOpis() + "   " +
                + napój3.koszt()+ " zł");
    }
}
```

Zamawia kawę Espresso (bez dodatków) i drukuje jej opis oraz cenę.

Utwórz obiekt MocnoPalona.

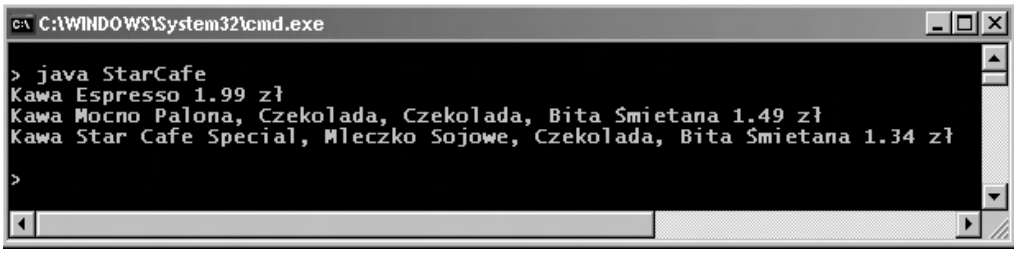
Udekoruj go Czekoladą.

Udekoruj drugą porcją Czekolady.

Udekoruj BitąŚmietaną.

Wreszcie poprosimy o kawę StarCafeSpecial, z MleczkiemSojowym, Czekoladą oraz BitąŚmietaną.

*Znacznie lepszy sposób tworzenia obiektów dekorowanych poznasz przy okazji omawiania wzorców Fabryka oraz Builder. Uwaga — opis wzorca Builder znajdziesz w dodatku.



Nie istnieją głupie pytania

P: Zastanawiam się nad funkcjonowaniem kodu, który testowałby tylko wybrane składniki — powiedzmy, kawę StarCafeSpecial — i wykonywał jakąś operację, np. udzielał rabatu na ten gatunek kawy. W sytuacji, kiedy taki kod zostałby „zawinięty" w jeden lub więcej dekoratorów, przestałby przecież działać.

O: Masz całkowitą rację. Jeżeli przygotujesz kod, który jest uzależniony od konkretnego typu składnika, dekorator uniemożliwi jego działanie. Dopóty, dopóki tworzysz kod aplikacji w oparciu o definicję klasy abstrakcyjnej, zastosowanie dekoratorów będzie dla niego całkowicie przezroczyste. Jeżeli jednak zaczniesz opierać swój kod o konkretne implementacje poszczególnych składników, z pewnością będziesz musiał przemyśleć projekt aplikacji i zastosowanie dekoratorów.

P: Czy przypadkiem w pewnych sytuacjach nie może się zdarzyć tak, że cały proces wyliczania kosztów zakończy się nie na zewnętrznym, ale na którymś z wewnętrznych dekoratorów? Przykładowo, jeżeli zamówimy kawę MocnoPalona z Czekoladą, MleczkiemSojowym i BitąŚmietaną, w zasadzie można dosyć łatwo popełnić błąd i napisać kod, którego realizacja zakończy się, dajmy na to, na odwołaniu do MleczkaSojowego zamiast do BitejŚmietany. W takiej sytuacji BitaŚmietana nie zostanie w ogóle uwzględniona w zamówieniu.

O: Oczywiście, możesz tutaj słusznie argumentować, że korzystając z wzorca Dekorator, musisz zarządzać większą liczbą obiektów, stąd istnieje zwiększone prawdopodobieństwo popełnienia błędu w kodzie programu i w konsekwencji wystąpienia np. opisywanych w pytaniu błędów. Należy jednak pamiętać o tym, że zazwyczaj do tworzenia dekoratorów używa się innych wzorców projektowych, takich jak np. Fabryka czy Builder. Kiedy omówimy te wzorce, przekonasz się, że tworzenie rzeczywistej implementacji składnika łącznie z jego dekoratorem jest dobrze hermetyzowane i nie będzie powodowało takich problemów.

P: Czy dany dekorator posiada jakąś wiedzę o innych dekoratorach w danym łańcuchu przetwarzania? Przykładowo, załóżmy, że chcę, aby moja metoda pobierzOpis() drukowała taki opis: „Bita Śmietana, podwójna Czekolada" zamiast „Czekolada, Bita Śmietana, Czekolada"? Takie rozwiązanie wymagałoby tego, aby najbardziej zewnętrzny dekorator znał wszystkie wewnętrzne dekoratory.

O: Dekoratory zostały pomyślane w ten sposób, aby dodawały nowe zachowania do obiektów dekorowanych. Jeżeli chcesz zajrzeć w głąb, do dekoratorów znajdujących się na poszczególnych poziomach łańcucha przetwarzania, niejako zaczynasz wypychać dekoratory daleko poza granicę tego, do czego tak naprawdę zostały stworzone. Jednak takie rozwiązania są możliwe. Wyobraź sobie dekorator DodatkiFormatujWydruk, który dokonuje finalnej analizy otrzymanego opisu i potrafi wydrukować opis w postaci „Bita Śmietana, podwójna Czekolada" zamiast „Czekolada, Bita Śmietana, Czekolada". Zauważ, że aby ułatwić realizację takiego zadania, metoda pobierzOpis() mogłaby zwracać wynik w postaci typu ArrayList.



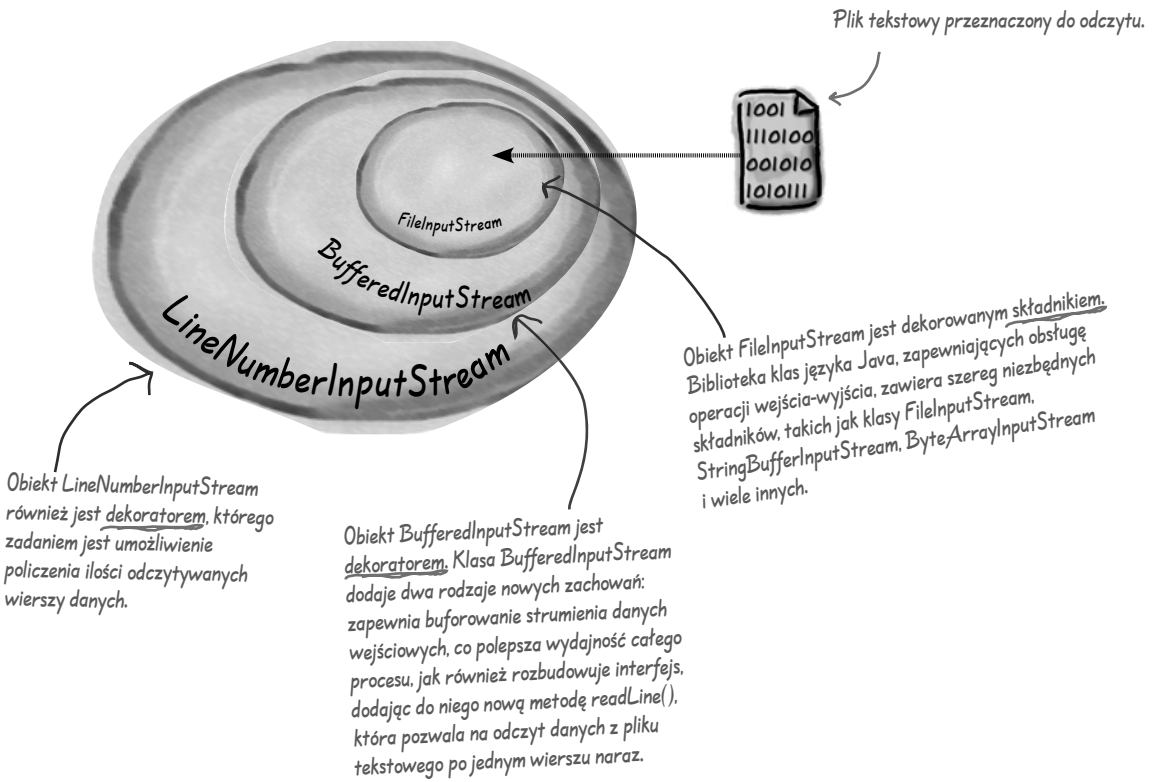
Zaostrz ołówek

Nasi przyjaciele ze „Star Café" wprowadzili do menu „wielkości" kawy. Obecnie możesz kupić kawę w trzech rozmiarach: małą, średnią i dużą. Specjaliści ze „Star Café" stwierdzili, że takie elementy powinny być nieodłączną częścią klasy opisującej napoje, stąd dodali do klasy Napój dwie metody: ustawWielkość() oraz pobierzWielkość(). Chcieli również, aby poszczególne dodatki były wyceniane w zależności od wielkości napoju, przykładowo, MleczkoSojowe będzie kosztowało 10 gr, 15 gr oraz 20 gr odpowiednio dla małej, średniej i dużej kawy.

W jaki sposób należałoby zmodyfikować klasy opisujące dekoratory, aby mogły one sobie poradzić z powyższym zadaniem?

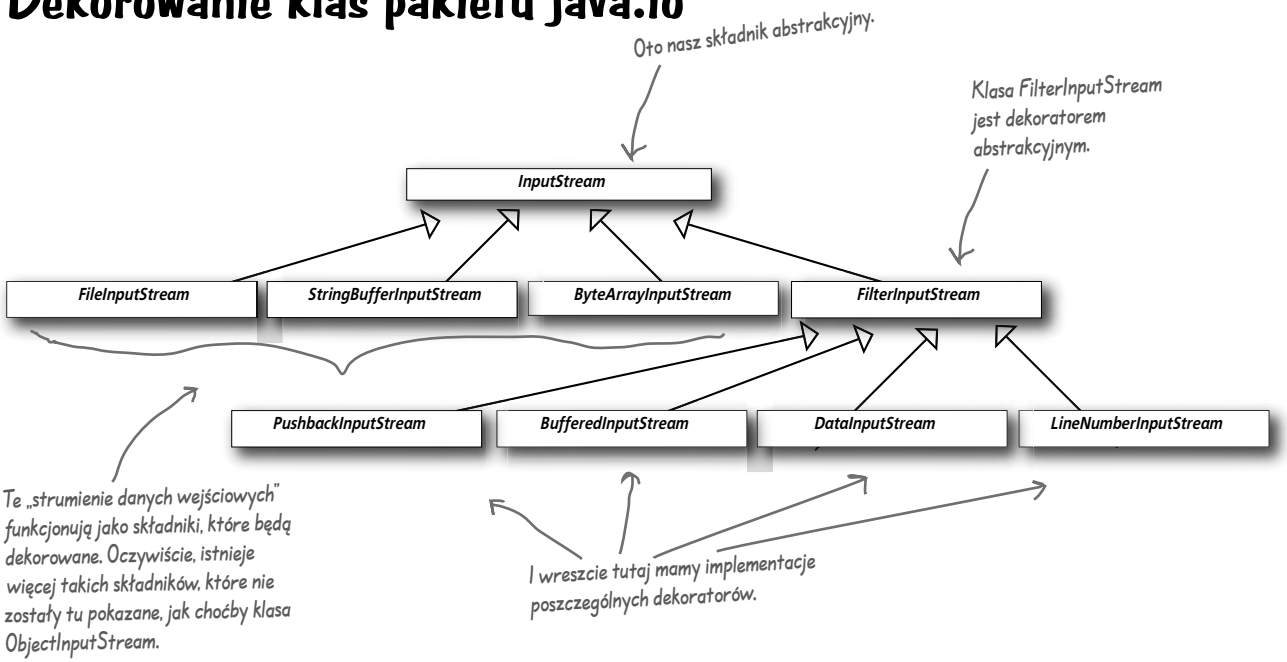
Dekoratory w świecie rzeczywistym: obsługa wejścia-wyjścia w języku Java

Ogromna ilość klas dostępnych w pakiecie java.io jest po prostu… *przytłaczająca*. Na pocieszenie powiemy Ci jednak, że jeżeli widząc po raz pierwszy (a nawet po raz drugi i trzeci…) listę klas tego pakietu, wrzasnąłeś „Uuuuaaa!", z pewnością nie byłeś w tym odosobniony. Co więcej, teraz, kiedy już poznałeś wzorzec Dekorator, te wszystkie klasy z pakietu java.io powinny być dla Ciebie o wiele bardziej zrozumiałe, jako że pakiet ten powstał w dużej mierze w oparciu właśnie o wzorzec Dekorator. Poniżej przedstawiliśmy typowy zestaw obiektów, które wykorzystują dekoratory do modyfikacji zachowań przy odczycie danych z pliku:



Klasy BufferedInputStream oraz ***LineNumberInputStream*** są rozszerzeniem klasy ***FilterInputStream***, która odgrywa rolę abstrakcyjnej klasy dekoratora.

Dekorowanie klas pakietu java.io



Jak sam zapewne zauważyłeś, powyższy diagram klas wcale tak bardzo nie różni się od naszego projektu aplikacji dla „Star Café". Obecnie nie powinieneś mieć już większych trudności ze zrozumieniem podręczników użytkownika pakietu java.io API, jak również z samym tworzeniem dekoratorów dla różnych strumieni danych *wejściowych*.

Niebawem przekonasz się, że strumienie danych *wyjściowych* mają taką samą strukturę. Zapewne zorientowałeś się już, że strumienie Reader oraz Writer (odpowiednio dla wejścia i wyjścia danych znakowych) są niemal dokładnym odzwierciedleniem struktury klas strumieni (z kilkoma wprawdzie drobnymi różnicami i niekonsekwencjami, ale na tyle małymi, że można bez trudu zrozumieć zasady ich funkcjonowania).

Z drugiej strony, pakiet java.io odsłania również jedną z *najciemniejszych stron* wzorca Dekorator: projekty budowane z wykorzystaniem tego wzorca często charakteryzują się powstawaniem dużej ilości niewielkich klas, które później mogą stanowić poważne utrudnienie dla projektanta chcącego skorzystać z opartego na tym wzorcu API. Teraz jednak, kiedy już doskonale znasz zasady funkcjonowania wzorca Dekorator, możesz na takie zjawisko zwracać większą uwagę; kiedy zaś będziesz korzystał z czyjegoś API przepełnionego różnymi dekoratorami, z pewnością łatwiej będzie Ci się zorientować w całej strukturze klas i wykorzystywać poszczególne dekoratory w sposób zapewniający osiągnięcie optymalnych, żądanych rezultatów.

Tworzenie własnych dekoratorów obsługi wejścia-wyjścia

OK, znasz już zasady funkcjonowania wzorca Dekorator, widziałeś też diagram klas zapewniających obsługę operacji wejścia-wyjścia. Krótko mówiąc, powinieneś być gotowy do napisania swojego własnego dekoratora obsługującego strumień danych wejściowych.

Co sądzisz o tym: napisz dekorator, którego zadaniem będzie konwersja wszystkich liter dużych na litery małe w strumieniu danych wejściowych. Innymi słowy, jeżeli w strumieniu danych wejściowych pojawi się następujący łańcuch alfanumeryczny: „Znam wzorzec dekorator, zatem to Ja tutaj USTALAM ZASADY!", Twój dekorator dokona konwersji tego łańcucha do następującej postaci: „znam wzorzec dekorator, zatem to ja tutaj ustalam zasady!".

Żaden kłopot. Po prostu muszę utworzyć klasę podrzędną do klasy FilterInputStream (rozszerzającą funkcjonalność klasy FilterInputStream) i przesłonić w niej metodę read().



Nie zapomnij zaimportować pakietu java.io (odpowiedni kod nie został tutaj pokazany).

Przede wszystkim musimy utworzyć nową podklasę, która będzie dziedziczyć po klasie FileInputStream, abstrakcyjnym dekoratorze wszystkich klas strumieni wejściowych (xxxxxInputStream).

```
public class LowerCaseInputStream extends FilterInputStream {
    public LowerCaseInputStream(InputStream in) {
        super(in);
    }

    public int read() throws IOException {
        int c = super.read();
        return (c == -1 ? c : Character.toLowerCase((char)c));
    }

    public int read(byte[] b, int offset, int len) throws IOException {
        int result = super.read(b, offset, len);
        for (int i = offset; i < offset+result; i++) {
            b[i] = (byte)Character.toLowerCase((char)b[i]);
        }
        return result;
    }
}
```

Teraz musimy zaimplementować dwie metody read(), których zadaniem jest odczytywanie danych. Pobierają one pojedynczy znak (bajt) lub tablicę znaków (tablicę bajtów), a następnie dokonują konwersji każdego znaku (każdy bajt reprezentuje jeden znak) do postaci małych liter.

PAMIĘTAJ: w naszych przykładach nie zamieszczamy fragmentów kodu odpowiedzialnych za import i obsługę poszczególnych pakietów bibliotek języka Java. Kompletny zestaw kodów źródłowych przykładów zamieszczonych w naszej książce możesz pobrać z serwera FTP wydawnictwa Helion: ftp://ftp.helion.pl/przyklady/wzorrg.zip

Testowanie nowego dekoratora obsługującego wejście-wyjście

Napiszemy teraz szybko nieco kodu, którego zadaniem będzie testowanie naszego nowego dekoratora:

```
public class InputTest {
    public static void main(String[] args) throws IOException {
        int c;

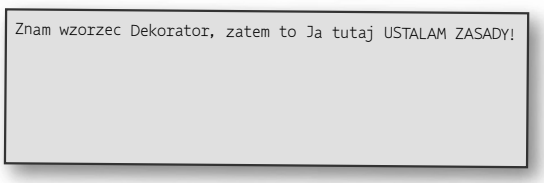
        try {
            InputStream in =
                new LowerCaseInputStream(
                    new BufferedInputStream(
                        new FileInputStream("test.txt")));

            while((c = in.read()) >= 0) {
                System.out.print((char)c);
            }

            in.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Tworzy obiekt klasy FileInputStream i następnie dekoruje go obiektami BufferedInputStream oraz naszym nowym filtrem LowerCaseInputStream.

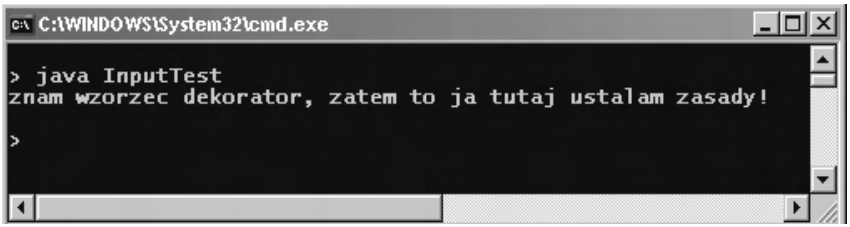
Korzystając ze strumienia danych, odczytujemy dane z pliku i na bieżąco drukujemy je na ekranie.



plik test.txt

Musisz utworzyć ten plik samodzielnie.

Odpalamy program:





Wzorce bez tajemnic

Wywiad tygodnia: Wyznania Dekoratora

Rusz głową!: Spotykamy się dzisiaj z wzorcem Dekorator. Słyszeliśmy, że ostatnio miałeś ze sobą pewne kłopoty?

Dekorator: Tak. Wiem, że wszyscy naokoło widzą mnie jako wspaniały wzorzec projektowy, podczas gdy — sam wiesz, jak to jest — muszę sobie radzić z różnymi problemami, tak jak każdy z nas.

Rusz głową!: To może podzielisz się swoimi problemami również z nami?

Dekorator: Oczywiście, czemu nie. Jak sam wiesz, posiadam moc dodawania projektom elastyczności i to jest naprawdę coś — ale, niestety, również mam swoje *ciemne strony*. Widzisz, czasami zbytnio się rozpędzam i powoduję, że w projekcie pojawia się ogromna ilość małych klas, co może spowodować, że staje się on nieco zbyt zagmatwany i trudny do zrozumienia dla innych programistów.

Rusz głową!: Czy możesz nam zaprezentować jakiś konkretny przykład?

Dekorator: Weźmy pakiet bibliotek java.io. Wśród programistów cieszą się one, niestety, zasłużoną opinią mało przyjaznych i trudnych do zrozumienia na pierwszy rzut oka. Jeżeli jednak oni zrozumieliby, że cały ten zestaw różnych klas to po prostu zestaw dekoratorów zbudowanych dookoła klas InputStream, ich życie stałoby się o wiele łatwiejsze.

Rusz głową!: Ale to przecież nie wygląda źle. W niczym nie zmienia to faktu, że nadal jesteś wspaniałym wzorcem, a cała reszta to po prostu kwestia odpowiedniej edukacji i szkolenia, prawda?

Dekorator: No... niestety, obawiam się, że jest coś jeszcze. Mam pewne problemy z kodem: widzisz, niektórzy ludzie po prostu biorą odpowiedni fragment kodu klienta, który operuje na określonych typach obiektów, i wprowadzają dekoratory, nie myśląc już o niczym innym. Jedną z moich naprawdę wspaniałych cech jest to, że w większości przypadków dekoratory możesz wstawić w sposób całkowicie przezroczysty i klient nigdy nie musi być poinformowany, że je wykorzystuje. Ale, niestety — jak już wspomniałem — niektóre partie kodu są uzależnione od określonych typów obiektów i kiedy zaczniesz wprowadzać do takiego układu dekoratory, robi się małe „Bum!”. I zaczyna być naprawdę źle.

Rusz głową!: Wiesz co, myślę jednak, że wszyscy doskonale zdają sobie sprawę, że przy wprowadzaniu dekoratorów do kodu należy zachować pewną ostrożność. Nie sądzę, aby był to wystarczający powód do robienia sobie poważnych wyrzutów.

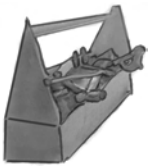
Dekorator: No wiem, wiem, i mimo wszystko staram się tego nie robić. Innym problemem, z którym się muszę borykać, jest to, że wprowadzenie dekoratorów do aplikacji może spowodować wzrost stopnia złożoności kodu niezbędnego do prawidłowego tworzenia obiektów poszczególnych klas składników. Kiedy korzystasz z dekoratorów, musisz nie tylko utworzyć sam obiekt danej klasy, ale jeszcze dodatkowo „opakować” go Bóg wie iloma dekoratorami.

Rusz głową!: Na przyszły tydzień mam już zaplanowany wywiad z wzorcami Fabryka oraz Builder — słyszałem, że one mogą być wielce pomocne przy rozwiązywaniu tego problemu?

Dekorator: To prawda; w zasadzie powinienem częściej się z nimi spotykać.

Rusz głową!: Mimo to i tak wszyscy wiemy, że jesteś wspaniałym wzorcem projektowym, umożliwiającym tworzenie elastycznych projektów pozostających w zgodzie z regułą otwarte-zamknięte, więc nie przejmuj się niczym, głowa do góry i myśl pozytywnie!

Dekorator: Postaram się, dziękuję bardzo.



Twoja skrzynka narzędziowa

Masz już za sobą kolejny rozdział i nowe reguły i wzorce znalazły się w Twojej narzędziowni.



CELNE SPOSTRZEŻENIA

- Dziedziczenie jest jedną z form rozszerzania funkcjonalności klasy, ale niekoniecznie musi być najlepszym sposobem na osiągnięcie w pełni elastycznych projektów aplikacji.
- Tworząc projekt aplikacji, powinieneś tak go skonstruować, aby możliwe było rozszerzanie zachowań poszczególnych elementów bez konieczności modyfikowania istniejącego kodu.
- Wykorzystując kompozycję oraz delegację, można dodawać nowe zachowania podczas działania programu.
- Wzorzec Dekorator stanowi pod względem funkcjonalności (dodawanie nowych zachowań) poważną alternatywę dla dziedziczenia.
- Wzorzec Dekorator posługuje się zbiorem klas dekorujących (dekoratorów), które są wykorzystywane do dekorowania poszczególnych obiektów (składników).
- Klasy dekorujące odzwierciedlają typy obiektów dekorowanych (w rzeczywistości dekoratory są tego samego typu co obiekty dekorowane, niezależnie, czy zostało to osiągnięte metodą dziedziczenia, czy implementacji odpowiednich interfejsów).
- Dekoratory zmieniają zachowania obiektów dekorowanych (składników), dodając nowe zachowania przed wywołaniami metod danego składnika i (lub) po nich (lub nawet pomiędzy nimi).
- Każdy składnik może być „otoczony" dowolną ilością dekoratorów.
- Dekoratory są zazwyczaj całkowicie przezroczyste dla klientów danego składnika — tak długo, jak długo funkcjonowanie klienta nie jest uzależnione od rzeczywistej implementacji danego składnika.
- Zastosowanie dekoratorów może być przyczyną pojawienia się w projekcie dużej ilości małych obiektów; nadużywanie dekoratorów może doprowadzić do zdecydowanego wzrostu złożoności kodu.

Rozwiązania ćwiczeń

```
public class Napój {

// deklaracje zmiennych obiektowych mlekoKoszt,
// mleczkoSojoweKoszt, czekoladaKoszt, bitaŚmietanaKoszt
// oraz odpowiednich metod dla poszczególnych dodatków

public float koszt() {

    float dodatekKoszt = 0.0;
    if (zMlekiem()) {
      dodatekKoszt += mlekoKoszt;
    }
    if (zMleczkiemSojowym()) {
      dodatekKoszt += mleczkoSojoweKoszt;
    }
    if (zCzekoladą()) {
      dodatekKoszt += czekoladaKoszt;
    }
    if (zBitąŚmietaną()) {
      dodatekKoszt += bitaŚmietanaKoszt;
    }
    return dodatekKoszt;
  }
}
```

```
public class MocnoPalona extends Napój {

  public MocnoPalona() {
    opis = „Znakomity, mocno palony gatunek kawy!"
  }

  public float koszt() {

    return 1.99 + super.koszt();

  }
}
```

Szkolenie nowego barmana

kawa z podwójną czekoladą, mleczkiem sojowym i bitą śmietaną





1. **Po pierwsze, wywołujemy metodę koszt() najbardziej zewnętrznego dekoratora, jakim jest BitaŚmietana.**
2. **Dekorator BitaŚmietana wywołuje metodę koszt() dekoratora Czekolada.**
3. **Dekorator Czekolada wywołuje metodę koszt() kolejnego dekoratora Czekolada.**
4. **Drugi dekorator Czekolada wywołuje metodę koszt() dekoratora MleczkoSojowe.**
5. **Ostatni dodatek! Dekorator MleczkoSojowe wywołuje metodę koszt() obiektu StarCafeSpecial.**
6. **Metoda koszt() obiektu StarCafeSpecial zwraca swój koszt (czyli 89 gr) i schodzi nam ze stosu.**
7. **Metoda koszt() dekoratora MleczkoSojowe dodaje do kosztu swoje 15 gr, zwraca wynik tej operacji i schodzi ze stosu.**
8. **Metoda koszt() drugiego dekoratora Czekolada dodaje do kosztu swoje 20 gr, zwraca wynik tej operacji i schodzi ze stosu.**
9. **Metoda koszt() pierwszego dekoratora Czekolada dodaje do kosztu swoje 20 gr, zwraca wynik tej operacji i schodzi ze stosu.**
10. **Wreszcie rezultat tego szeregu operacji powraca do metody koszt() dekoratora BitaŚmietana, który dodaje swój koszt, czyli 10 groszy, do wyniku uzyskanego od dekoratora Czekolada, i zwraca całkowity koszt napoju, czyli 1,54 zł.**

Rozwiązania ćwiczeń

Nasi przyjaciele ze „Star Café” wprowadzili do menu „wielkości” kawy. Obecnie możesz kupić kawę w trzech rozmiarach: małą, średnią i dużą. Specjaliści ze „Star Café” stwierdzili, że takie elementy powinny być nieodłączną częścią klasy opisującej napoje, stąd dodali do klasy Napoje dwie metody: ustawWielkość() oraz pobierzWielkość(). Chcieli również, aby poszczególne dodatki były wyceniane w zależności od wielkości napoju, przykładowo, MleczkoSojowe będzie kosztowało 10 gr, 15 gr oraz 20 gr odpowiednio dla małej, średniej i dużej kawy.

W jaki sposób należałoby zmodyfikować klasy opisujące dekoratory, aby mogły one sobie poradzić z powyższym zadaniem?

```
public class MleczkoSojowe extends SkładnikDekorator{
    Napój napój;

    public MleczkoSojowe (Napój napój) {
        this.napój = napój;
    }

    public intpobierzWielkość() {
        return napój.pobierzWielkość();
    }

    public String pobierzOpis() {
        return napój.pobierzOpis() + ", Mleczko Sojowe";
    }

    public double koszt() {
        double koszt = napój.koszt();
        if (pobierzWielkość() == Napój.MAŁY) {
            koszt += 0,10;
        } else if (pobierzWielkość() == Napój.ŚREDNI) {
            koszt += 0,15;
        } else if (pobierzWielkość() == Napój.DUŻY) {
            koszt += 0,20;
        }
        return.koszt;
        }
}
```

Teraz musimy zaimplementować metodę pobierzWielkość() do dekorowanego napoju. Musimy również zaimplementować tę metodę w klasie abstrakcyjnej, ponieważ będzie używana przez wszystkie dekoratory.

Tutaj pobieramy wielkość napoju, a następnie dodajemy do ceny napoju odpowiedni koszt dodatku.

4. Wzorzec Fabryka

Pizzeria zorientowana obiektowo



Przygotuj się do stworzenia kilku projektów, w których zastosujemy luźne powiązania pomiędzy poszczególnymi obiektami. Stworzenie nowego obiektu to dużo więcej niż tylko proste zastosowanie operatora **new**. Niebawem przekonasz się, że proces ten jest operacją, która nie zawsze powinna być publicznie dostępna, a co więcej, jest operacją, która często może prowadzić do poważnych problemów z powiązaniami międzyobiektowymi. A *tego* byś nie chciał, prawda? Przekonaj się, w jaki sposób wzorzec Fabryka może uratować Cię z takiej opresji.

No dobrze, mamy już za sobą trzy rozdziały, a do tej pory nie uzyskałam odpowiedzi na moje pytanie dotyczące operatora new. Zgodnie z regułami projektowania nie powinniśmy tworzyć implementacji, ale przecież za każdym razem, kiedy używam operatora new, właśnie dokładnie to robię, prawda?



Kiedy widzisz „nowy” obiekt, myśl o nim jako o „konkretnym”

Rzeczywiście, za każdym razem, kiedy korzystasz z operatora **new**, tworzysz obiekt będący reprezentacją rzeczywistej klasy, czyli bezdyskusyjnie jest to tworzenie implementacji, a nie interfejsu. I to jest rzeczywiście dobre pytanie; wiesz już przecież, że „dowiązanie” Twojego kodu programu do definicji konkretnej klasy powoduje, że cały kod staje się bardziej wrażliwy na zmiany i mniej elastyczny.

```
Kaczka kaczka = new DzikaKaczka();
```

Aby utrzymać elastyczność kodu, chcemy skorzystać z interfejsów.

Ale musimy utworzyć obiekt konkretnej, rzeczywistej klasy.

Kiedy posiadasz już cały zestaw powiązanych ze sobą, rzeczywistych klas, często jesteś zmuszony do tworzenia kodu takiego, jaki przedstawiliśmy na poniższym przykładzie:

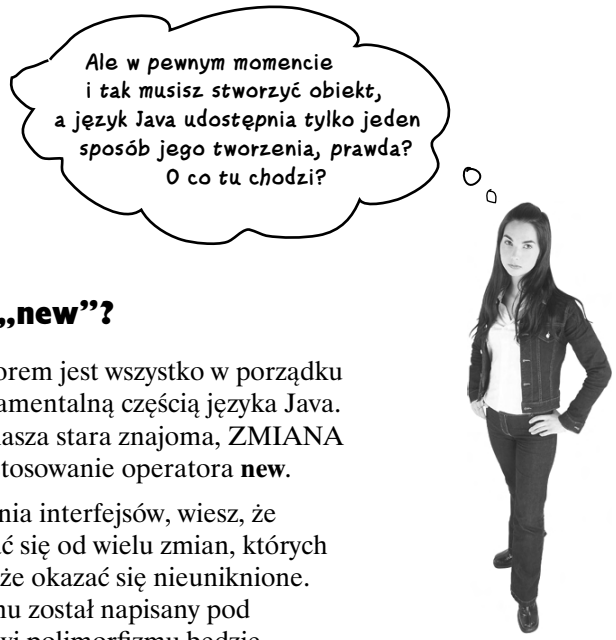
```
Kaczka kaczka;

if (piknik) {
    kaczka = new DzikaKaczka();
} else if (polowanie) {
    kaczka = new WabikKaczka();
} else if (wanna) {
    kaczka = new GumowaKaczka();
}
```

Mamy całe mnóstwo klas opisujących poszczególne gatunki kaczek, a do momentu uruchomienia programu nie będziemy wiedzieli, której z nich trzeba użyć do utworzenia nowej instancji obiektu Kaczka.

Mamy tutaj kilka różnych klas, których obiekty mogą zostać stworzone, ale decyzja o tym, która klasa zostanie wybrana do stworzenia takiego obiektu, zapadnie dopiero podczas działania programu, po analizie pewnych określonych warunków.

Kiedy zaczynasz analizować taki kod, jak ten przedstawiony powyżej, szybko dochodzisz do wniosku, że kiedy nastąpi zmiana wymagań bądź konieczność rozbudowy danej aplikacji, nie obejdzie się, niestety, bez modyfikacji istniejącego kodu i dodawania bądź usuwania z niego całych fragmentów. Często taki kod posiada powiązania z wieloma innymi elementami programu, co w efekcie powoduje, że jego konserwacja i aktualizacja jest niezwykle uciążliwa, a całość podatna na błędy.



Co jest nie tak z operatorem „new”?

Technicznie rzecz biorąc, z tym operatorem jest wszystko w porządku — jak by na to nie patrzeć, jest on fundamentalną częścią języka Java. Prawdziwym sprawcą problemów jest nasza stara znajoma, ZMIANA — i to, w jaki sposób wpływa ona na zastosowanie operatora **new**.

Kiedy programujesz pod kątem tworzenia interfejsów, wiesz, że będziesz w stanie skutecznie odizolować się od wielu zmian, których wprowadzenie na przestrzeni czasu może okazać się nieuniknione. Dlaczego? Jeżeli kod Twojego programu został napisany pod kątem interfejsów, dzięki mechanizmowi polimorfizmu będzie współpracował z dowolną nową klasą posiadającą taki interfejs. Z drugiej zaś strony, jeżeli Twój kod wykorzystuje cały szereg klas rzeczywistych, musisz sobie zdawać sprawę z tego, że sam szukasz problemów, ponieważ w sytuacji, kiedy trzeba będzie dodać nowe klasy rzeczywiste, będzie on musiał z pewnością zostać solidnie zmodyfikowany. Innymi słowy, taki kod nie będzie „zamknięty na modyfikacje”. Aby rozszerzyć go o nowe klasy, będziesz musiał go „otworzyć” na zmiany.

Pamiętaj, że Twój projekt powinien być „otwarty na rozbudowę, ale zamknięty na modyfikacje” — patrz rozdział 3.

Zatem, co możesz w takiej sytuacji zrobić? Warto sięgnąć pamięcią wstecz i przypomnieć sobie wszystkie znane Ci reguły projektowania — z pewnością znajdziesz tam odpowiednie wskazówki. Pamiętaj, nasza pierwsza reguła odnosi się do zmian i wyraźnie mówi, że powinno się *dokonać identyfikacji wszystkich zmiennych elementów Twojej aplikacji, a następnie oddzielić je od elementów, które przez cały czas pozostają niezmienione*.



WYTĘŻ UMYSŁ

W jaki sposób mógłbyś oddzielić i poddać hermetyzacji wszystkie fragmenty kodu odpowiedzialne za tworzenie nowych instancji obiektów poszczególnych klas?

Identyfikacja zmiennych elementów aplikacji



Załóżmy, że posiadasz świetną pizzerię i jako szczęśliwy właściciel dobrze prosperującego biznesu w Obiektowie możesz sobie pozwolić na napisanie na przykład takiego kodu:

```
Pizza zamówPizza() {
    Pizza pizza = new Pizza();

    pizza.przygotowanie();
    pizza.pieczenie();
    pizza.krojenie();
    pizza.pakowanie();
    return pizza;
}
```

Ze względu na pożądaną elastyczność bardzo chcielibyśmy, aby była to klasa abstrakcyjna lub interfejs, ale, niestety, nie możemy stworzyć takiego obiektu ani w jednym, ani w drugim przypadku.

Ale potrzebny Ci więcej niż jeden gatunek pizzy...

Więc dodałeś nieco kodu, który jest odpowiedzialny za *określenie* odpowiedniego gatunku pizzy i jego przygotowanie:

```
Pizza zamówPizza(String type) {
    Pizza pizza;

    if (type.equals("serowa")) {
        pizza = new SerowaPizza();
    } else if (type.equals("grecka")) {
        pizza = new GreckaPizza();
    } else if (type.equals("pepperoni")) {
        pizza = new PepperoniPizza();
    }

    pizza.przygotowanie();
    pizza.pieczenie();
    pizza.krojenie();
    pizza.pakowanie();
    return pizza;
}
```

Teraz jako argument przekazujemy gatunek pizzy.

Opierając się na informacji o żądanym gatunku pizzy, tworzymy obiekt odpowiedniej klasy i przypisujemy go do zmiennej obiektowej pizza. Zwróć uwagę, że każdy gatunek pizzy musi posiadać zaimplementowany interfejs Pizza.

Kiedy nasze zamówienie na Pizzę jest już gotowe, musimy ją przygotować (wiesz, o co chodzi — wyrobić ciasto, nałożyć sos, nałożyć farsz, dodatki i ser), następnie musimy ją upiec, pokroić i zapakować!

Każdy gatunek Pizzy (SerowaPizza, WegetariańskaPizza itd.) sam „wie", jak ma być przygotowywany.

Ale klienci naciskają, aby dodawać ciągle nowe gatunki pizzy

Właśnie zdałeś sobie sprawę z tego, że wszystkie konkurencyjne pizzerie w Obiektowie już umieściły w swojej ofercie dwa nowe, bardzo popularne wśród klientów gatunki pizzy: „owoce morza" oraz „wegetariańska". Oczywiście, nie możesz sobie pozwolić na to, aby konkurencja Cię zdeklasowała, więc również umieszczasz w menu swojej pizzerii wspomniane dwa nowe gatunki pizzy; ponieważ zaś ostatnimi czasy pizza grecka słabo się sprzedawała, postanowiłeś wycofać ją ze swojej oferty:

Ten kod NIE JEST zamknięty na modyfikacje. Jeżeli nasza pizzeria zmieni swoją ofertę, będziemy musieli powracać do tego miejsca kodu i modyfikować go.

```
Pizza zamówPizza(String type) {
    Pizza pizza;

    if (type.equals("serowa")) {
        pizza = new SerowaPizza();
    } else if (type.equals("grecka")) {
        pizza = new GreckaPizza();
    } else if (type.equals("pepperoni")) {
        pizza = new PepperoniPizza();
    } else if (type.equals("owocemorza")) {
        pizza = new OwoceMorzaPizza();
    } else if (type.equals("wegetariańska")) {
        pizza = new WegetarianskaPizza();
    }

    pizza.przygotowanie();
    pizza.pieczenie();
    pizza.krojenie();
    pizza.pakowanie();
    return pizza;
}
```

To jest fragment kodu, który ulega zmianom. W miarę, jak zmianom ulegać będzie oferta pizzerii, będziemy musieli modyfikować ten fragment kodu.

Możemy zakładać, że ten fragment kodu pozostanie stały, niezależnie od zmian w ofercie pizzerii. Na szczęście, cały proces przygotowania, pieczenia i pakowania pizzy pozostaje prawie niezmieniony już od wielu lat. Nie przewidujemy zatem żadnych zmian w tym kodzie, a tylko w gatunkach pizzy, które on obsługuje.

Nikt chyba nie ma wątpliwości, że to właśnie cała zabawa z wyborem klasy, *która* zostanie użyta do tworzenia obiektu, powoduje zamieszanie w naszej metodzie zamówPizza() i skutecznie blokuje jej możliwość zamknięcia na modyfikacje. Niemniej jednak w chwili obecnej wiemy już, które elementy podlegają zmianom, a które pozostają stałe — nadszedł zatem czas, aby skorzystać z hermetyzacji.

Hermetyzacja procesu tworzenia obiektów

Teraz już wiemy, że najlepszym rozwiązaniem będzie przeniesienie procesu tworzenia obiektów poza ramy metody zamówPizza(). Ale jak tego dokonać? To, co mamy zamiar zrobić, polega na zabraniu całego kodu odpowiedzialnego za tworzenie obiektów i przełożeniu go do innego obiektu, którego zadaniem będzie tylko i wyłącznie tworzenie pizzy.

```
if (type.equals("serowa")) {
        pizza = new SerowaPizza();
    } else if (type.equals("pepperoni")) {
        pizza = new PepperoniPizza();
    } else if (type.equals("owocemorza")) {
        pizza = new OwoceMorzaPizza();
    } else if (type.equals("wegetariańska")) {
        pizza = new WegetarianskaPizza();
    }
```

```
Pizza zamówPizza(String type) {
    Pizza pizza;

    pizza.przygotowanie();
    pizza.pieczenie();
    pizza.krojenie();
    pizza.pakowanie();
    return pizza;
}
```

Najpierw wyjmujemy cały kod odpowiedzialny za tworzenie obiektów z metody zamówPizza().

Co tu się będzie działo?

A następnie umieszczamy ten kod w obiekcie, którego jedynym i wyłącznym zadaniem będzie tworzenie pizzy. Jeżeli jakikolwiek inny obiekt będzie potrzebował utworzenia nowej pizzy, będzie się musiał z tym zwrócić do tego właśnie obiektu.



Mamy świetną nazwę dla naszego nowego obiektu: nazwiemy go Fabryka (ang. *Factory*).

Fabryki będą się zajmowały szczegółami tworzenia poszczególnych obiektów. Kiedy stworzymy obiekt ProstaFabrykaPizzy, nasza metoda zamówPizza() stanie się po prostu klientem tego obiektu. Za każdym razem, kiedy metoda ta będzie potrzebowała nowej pizzy, będzie musiała poprosić Fabrykę o jej przygotowanie. Czasy, kiedy metoda zamówPizza() musiała znać różnice pomiędzy pizzą grecką a pizzą z owocami morza, odeszły już w niepamięć. Obecnie metoda zamówPizza() dba tylko o to, by po prostu otrzymać obiekt pizza posiadający odpowiedni interfejs Pizza, dzięki któremu może wywołać metody przygotowanie(), pieczenie(), krojenie() oraz pakowanie().

Ciągle jednak pozostaje nam trochę szczegółów do dopracowania. Spróbujmy zatem zaimplementować prostą fabrykę pizzy, dzięki czemu wiele się wyjaśni…

Budujemy prostą fabrykę pizzy

Rozpoczniemy od stworzenia samej fabryki. Mamy zamiar zdefiniować obiekt, który pozwoli na całkowitą hermetyzację procesu tworzenia obiektów reprezentujących pizze dowolnego gatunku. Oto i on…

Oto nasza nowa klasa, ProstaFabrykaPizzy. Ma ona tylko jedno zadanie: tworzenie pizzy dla swoich klientów.

Najpierw w fabryce zdefiniujemy metodę utwórzPizza(). To jest metoda, której wszyscy klienci będą używali do tworzenia nowych obiektów.

```
public class ProstaFabrykaPizzy {
    public Pizza utwórzPizza(String type) {
        Pizza pizza = null;

    if (type.equals("serowa")) {
        pizza = new SerowaPizza();
    } else if (type.equals("pepperoni")) {
        pizza = new PepperoniPizza();
    } else if (type.equals("owocemorza")) {
        pizza = new OwoceMorzaPizza();
    } else if (type.equals("wegetariańska")) {
        pizza = new WegetariańskaPizza();
    }
        return pizza;
    }
}
```

Oto kod, który wyciągnęliśmy z metody zamówPizza().

Ten kod jest nadal silnie uzależniony od poszczególnych parametrów (czyli poszczególnych gatunków pizzy), dokładnie tak, jak to miało miejsce w oryginalnej metodzie zamówPizza().

P: Jakie są zalety takiego postępowania? Przecież wygląda na to, że po prostu wypychamy cały problem do innego obiektu.

O: Jednym z ważnych czynników, o których musimy pamiętać, jest to, że ProstaFabrykaPizzy może mieć wielu klientów. Do tej pory widzieliśmy tylko metodę zamówPizza(); nie zmienia to jednak w niczym faktu, że gdzieś może istnieć klasa PizzeriaMenu, która, powiedzmy, wykorzystuje fabrykę do pobierania od niej aktualnych opisów i cennika poszczególnych gatunków pizzy.

Nie istnieją
głupie pytania

Możemy również gdzieś posiadać klasę DostawaDoDomuPizza, która, co prawda, będzie traktowała fabrykę w nieco inny sposób niż klasa PizzeriaMenu, ale mimo to nadal będzie klientem tej fabryki.

Tak więc dzięki hermetyzacji procesu tworzenia pizzy w jednej klasie mamy tylko jeden element, który będzie podlegał modyfikacjom podczas implementacji zmian w systemie.

Nie zapominaj również, że niebawem spróbujemy całkowicie usunąć proces tworzenia obiektów z naszego kodu klienta.

P: Widziałem kiedyś podobny projekt, w którym taka fabryka była zdefiniowana jako metoda statyczna. Czym to się różni?

O: Definiowanie prostej fabryki w postaci metody statycznej jest często stosowaną techniką, znaną pod nazwą „fabryki statycznej" (ang. *static factory*). Dlaczego korzystamy z metody statycznej? Ponieważ w takim przypadku nie jest potrzebne tworzenie obiektów za pomocą konstruktora danej klasy. Należy również pamiętać, że takie postępowanie ma jednak poważną wadę, ponieważ nie można w takim przypadku użyć dziedziczenia i zmieniać działania konstruktora klasy.

Aktualizujemy kod klasy Pizzeria

Nadszedł już czas, aby poprawić kod naszego klienta. Jak wspominaliśmy wcześniej, chcemy, aby tworzenie nowej pizzy było wyłącznie zadaniem fabryki. Poniżej przedstawiamy zmiany, jakie wprowadziliśmy do kodu aplikacji:

Klasa Pizzeria posiada teraz odwołanie do obiektu klasy ProstaFabrykaPizza.

Obiekt Fabryka jest przekazywany jako argument konstruktora klasy Pizzeria.

```
public class Pizzeria {
    ProstaFabrykaPizzy fabryka;

    public Pizzeria(ProstaFabrykaPizzy fabryka) {
        this.fabryka = fabryka;
    }

    public Pizza zamówPizza(String type) {
        Pizza pizza;

        pizza = fabryka.utwórzPizza(type);

        pizza.przygotowanie();
        pizza.pieczenie();
        pizza.krojenie();
        pizza.pakowanie();
        return pizza;
    }
// tutaj umieść inne metody
}
```

Metoda zamówPizza() wykorzystuje fabrykę do tworzenia nowej pizzy, przekazując jej tylko gatunek żądanej pizzy.

*Zwróć uwagę, że zamieniliśmy operator **new** na **metodę** utwórzPizza() obiektu fabryka. Dzięki temu pozbywamy się tutaj tworzenia nowych obiektów!*



WYTĘŻ UMYSŁ

Wiemy już, że kompozycja obiektów pozwala (między innymi) na dynamiczną zmianę ich zachowań w trakcie działania programu, ponieważ możliwe jest dynamiczne zamienianie ich implementacji. Jak możemy wykorzystać takie rozwiązanie w naszej aplikacji obsługującej pizzerię? Jakie implementacje zachowań fabryki mogą być zamieniane?

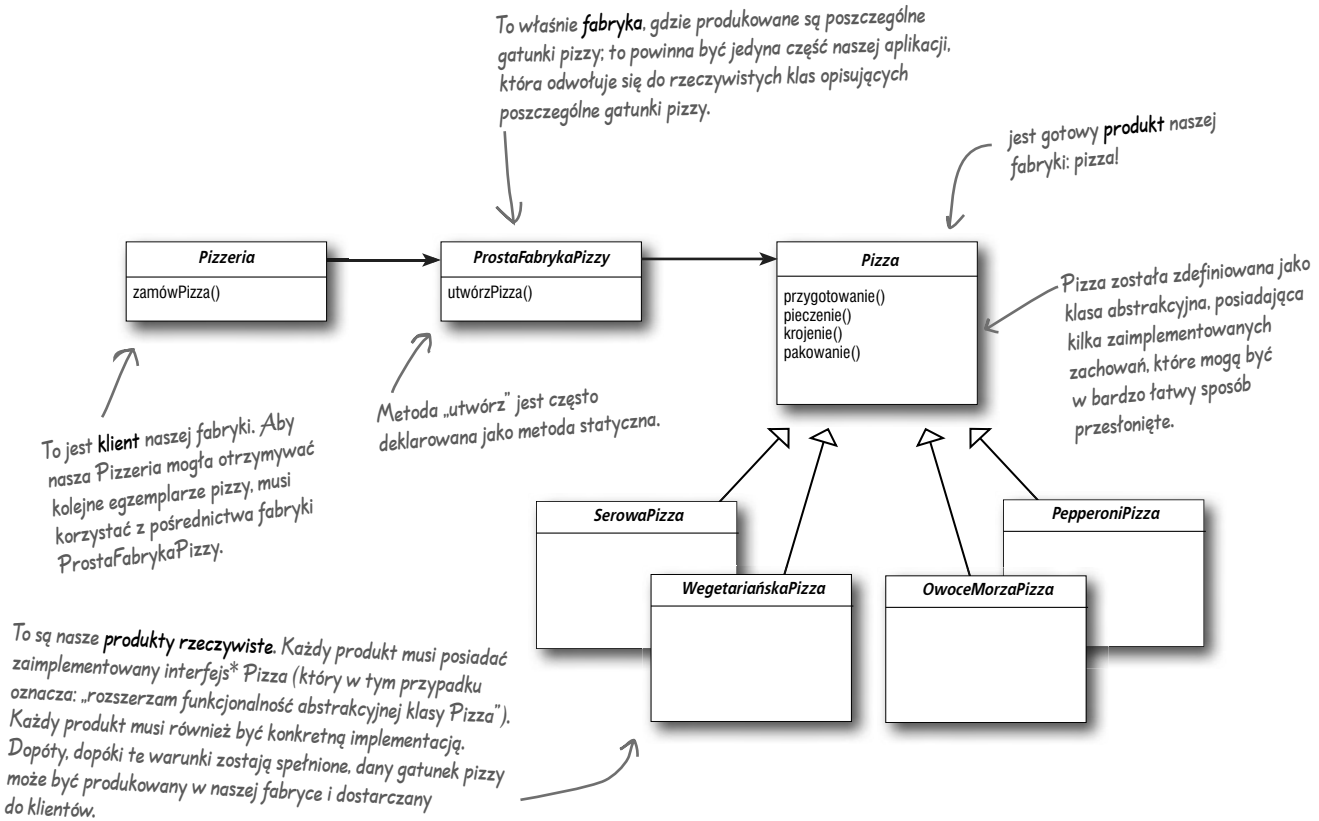
Nie wiemy, w czym się specjalizujesz, ale możemy przyjąć, że obsługujemy fabryki produkujące pizzę włoską, amerykańską i meksykańską (nie zapominajmy również o naszej rodzimej pizzy polskiej).

Tworzymy definicję „wzorca" Simple Factory



Sam fakt, że Prosta Fabryka nie jest PRAWDZIWYM wzorcem projektowym, nie oznacza wcale, że nie powinniśmy sprawdzić, w jaki sposób można z niego skorzystać. Rzućmy okiem na diagram klas naszej nowej Pizzerii:

W zasadzie Prosta Fabryka nie jest pełnoprawnym wzorcem projektowym; jest to raczej idiom, którym chętnie posługują się programiści. Ponieważ jest on bardzo często spotykany, nadamy mu tytuł Honorowego Członka Loży Wzorców Projektowych Rusz głową! Niektórzy projektanci mylą jednak ten idiom z rzeczywistym wzorcem Fabryka, więc jeżeli kiedyś w rozmowie z innym programistą przydarzy Ci się niezręczna chwila ciszy, będziesz miał świetny temat, pozwalający na przełamanie lodów.



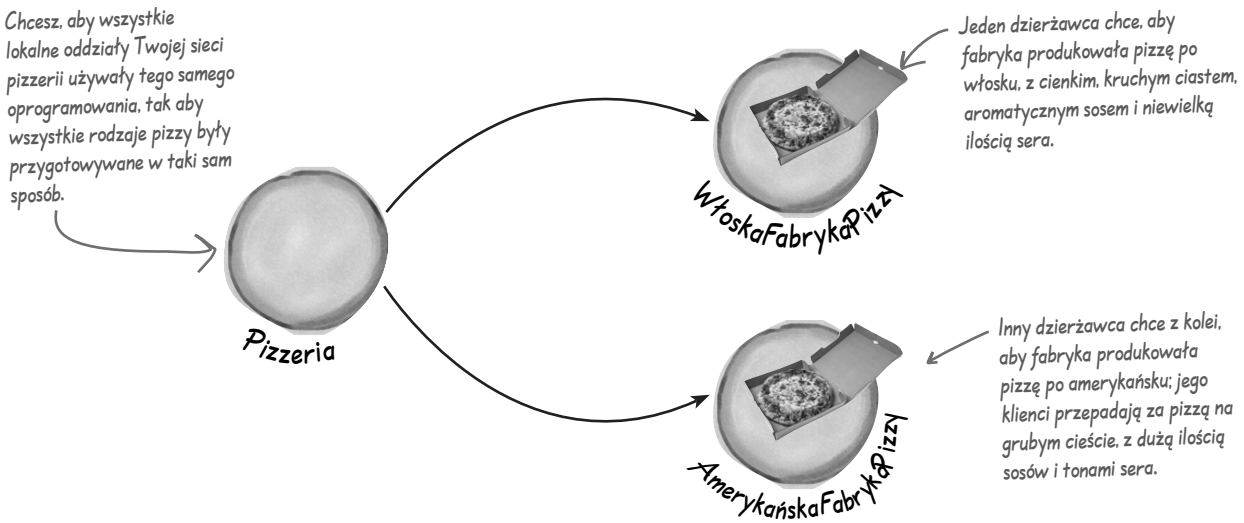
Potraktuj nasze dotychczasowe rozważania o Simple Factory jako rozgrzewkę. Teraz nadszedł czas, aby zająć się dwoma poważnymi wzorcami, które spełniają rolę prawdziwych „fabryk". Ale nie martw się, kolejne dostawy pizzy już są w drodze!

* *Pamiętaj: w terminologii wzorców projektowych fraza „zaimplementować interfejs" NIE ZAWSZE oznacza „utworzyć klasę, która implementuje interfejs Java przy użyciu w deklaracji klasy słowa kluczowego* **implements***". Ogólnie rzecz ujmując, jeżeli klasa rzeczywista implementuje daną metodę ze swojego supertypu (którym może być klasa LUB interfejs), nadal traktujemy ją jako klasę „posiadającą zaimplementowany interfejs" tego supertypu.*

Jak zbudować sieć pizzerii*

Twoja wspaniała Pizzeria w Obiektowie radziła sobie tak rewelacyjnie, że po prostu zrujnowała wszelką konkurencję i teraz wszyscy wokoło chcą mieć lokal Pizzerii w swoim najbliższym sąsiedztwie. Jako właściciel sieci z pewnością będziesz chciał zapewnić identyczną, wysoką jakość usług w każdym oddziale, więc zdecydujesz, że każdy lokal ma używać Twojego wspaniałego, sprawdzonego oprogramowania.

Ale co począć z oddziałami zlokalizowanymi w innych regionach kraju (czy nawet w innych krajach)? Zapewne każdy z nich będzie chciał oferować swoje własne, lokalne rodzaje pizzy (włoska, amerykańska, meksykańska, polska — aby wymienić tylko kilka z nich), w zależności od gustów i upodobań lokalnych koneserów pizzy.



Widzieliśmy już jeden sposób rozwiązania takiego problemu...

Jeżeli weźmiemy naszą SimplePizzaFactory i stworzymy dla niej trzy różne fabryki, WłoskaFabrykaPizzy, AmerykańskaFabrykaPizzy oraz MeksykańskaFabrykaPizzy, możemy uruchomić daną Pizzerię i korzystając z kompozycji, dobrać dla niej odpowiednią fabrykę — w ten sposób otrzymamy gotowy, działający lokalny oddział pizzerii. To jest pierwsze podejście do rozwiązania problemu.

Zobaczmy zatem, jak to mogłoby wyglądać w praktyce…

* Wszelkie przepisy, procedury i inne szczegóły dotyczące przygotowywania pizzy na sposób włoski, amerykański i jakikolwiek inny, zostały tutaj umieszczone tylko i wyłącznie gwoli ilustracji zasad funkcjonowania wzorca Fabryka i kodu Java i niekoniecznie muszą mieć cokolwiek wspólnego z rzeczywistymi charakterystykami poszczególnych gatunków pizzy; za ewentualne pomieszanie z poplątaniem tłumacz składa serdeczne przeprosiny koneserom tej niezbyt wyrafinowanej, ale jakże smacznej i popularnej potrawy (do których zresztą i sam też się zalicza…) — *przyp. tłum.*

```
WłoskaFabrykaPizzy włoskaFabryka = new WłoskaFabrykaPizzy();
Pizzeria włoskaPizzeria = new Pizzeria(włoskaFabryka);
włoskaPizzeria.zamówPizza("wegetariańska");
```

tworzymy fabrykę produkującą pizzę w stylu włoskim.

Następnie tworzymy Pizzerię i przekazujemy jej odwołanie do odpowiedniej fabryki (WłoskaFabrykaPizzy).

i kiedy zamawiamy pizzę, otrzymujemy ją przygotowaną w stylu włoskim.

```
AmerykańskaFabrykaPizzy amerykańskaFabryka = new AmerykańskaFabrykaPizzy();
Pizzeria amerykańskaPizzeria = new Pizzeria(amerykańskaFabryka);
amerykańskaPizzeria.zamówPizza("wegetariańska");
```

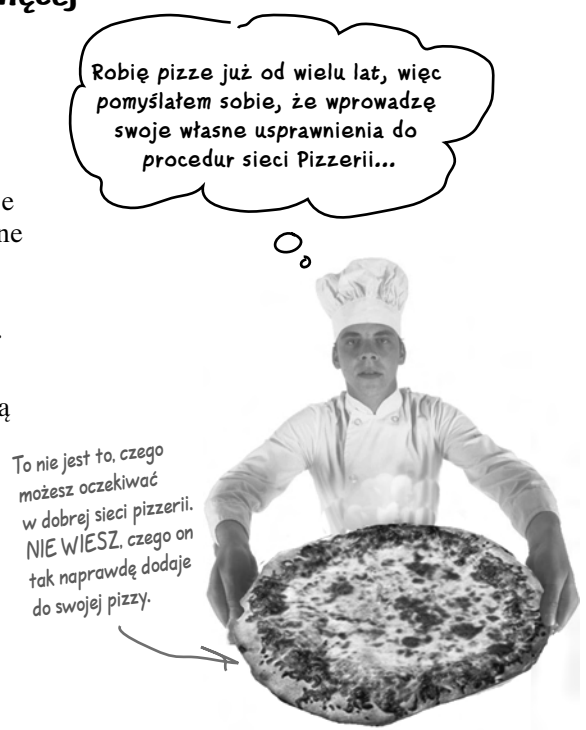
Z amerykańską pizzą postępujemy w analogiczny sposób: tworzymy fabrykę produkującą pizzę w stylu amerykańskim, a następnie tworzymy nową Pizzerię i metodą kompozycji przypisujemy jej odpowiednią fabrykę. W ten sposób, kiedy zamawiamy pizzę, otrzymujemy ją przygotowaną w stylu amerykańskim.

Ale wygląda na to, że chciałbyś mieć nieco więcej kontroli nad jakością produktów...

Po pewnym czasie funkcjonowania takiego systemu przekonałeś się, że co prawda poszczególni dzierżawcy wykorzystywali Twoje fabryki do tworzenia pizzy, ale do reszty procesu przygotowywania pizzy zaczęli stosować swoje własne rozwiązania: poszczególne rodzaje pizzy były pieczone na różne sposoby, dzierżawcy zapominali, że pizzę należy przed dostarczeniem pokroić, a poza tym wykorzystywali opakowania pochodzące od zupełnie różnych producentów.

W czasie analizowania tej sytuacji dochodzisz do wniosku, że tak naprawdę chciałbyś jednak utworzyć pewną określoną strukturę produkcji, która mocno wiąże ze sobą pizzerię i proces produkcji pizzy, ale jednocześnie zapewnia dużą elastyczność całego procesu.

W jednej z wczesnych wersji kodu naszej aplikacji, jeszcze przed wprowadzeniem klasy ProstaFabrykaPizzy, kod odpowiedzialny za tworzenie pizzy był powiązany z kodem Pizzerii, ale takie rozwiązanie nie było elastyczne. Krótko mówiąc, w jaki sposób możemy mieć pizzę i jednocześnie ją zjeść?



Nowa struktura Pizzerii

A jednak *istnieje* sposób, aby umieścić wszystkie operacje związane z produkcją pizzy w klasie Pizzeria, a mimo to zapewnić dzierżawcom możliwość tworzenia regionalnych odmian tego dania.

Aby tego dokonać, mamy zamiar umieścić metodę utwórzPizza() z powrotem w kodzie klasy Pizzeria, ale tym razem zadeklarować ją jako **metodę abstrakcyjną**, a następnie dla każdej regionalnej odmiany pizzy tworzyć klasy podrzędne do klasy Pizzeria.

Aby to lepiej zilustrować, rzućmy okiem na zmiany w definicji klasy Pizzeria:

Klasa Pizzeria jest teraz klasą abstrakcyjną (dlaczego tak jest, przekonasz się poniżej).

```
public abstract class Pizzeria {

    public Pizza zamówPizza(String type) {
        Pizza pizza;

        pizza = utwórzPizza(type);

        pizza.przygotowanie();
        pizza.pieczenie();
        pizza.krojenie();
        pizza.pakowanie();

        return pizza;

    }

    abstract Pizza utwórzPizza(String type);
}
```

Obecnie metoda utwórzPizza() jest ponownie wywoływana w klasie Pizzeria, a nie w fabryce.

Tutaj cała reszta wygląda tak samo...

Teraz nasz obiekt reprezentujący fabrykę został przesunięty do tej metody.

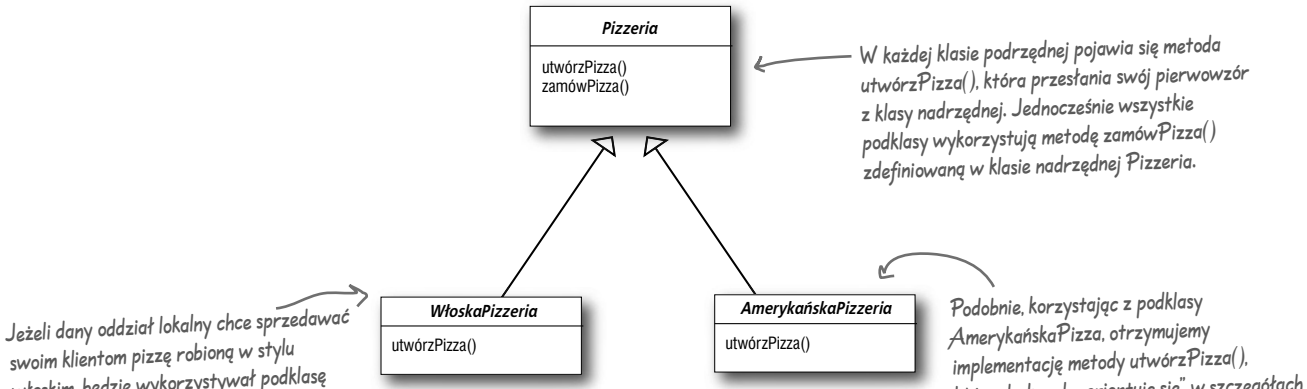
Nasza „metoda fabryczna" ma teraz postać metody abstrakcyjnej w klasie Pizzeria.

Teraz nasze poszczególne oddziały pizzerii czekają na utworzenie odpowiednich klas podrzędnych; dla każdego lokalnego oddziału naszej sieci stworzymy osobną podklasę (WłoskaFabrykaPizzy, AmerykańskaFabrykaPizzy oraz MeksykańskaFabrykaPizzy). Każda podklasa będzie musiała sama zadecydować, co będzie wchodziło w skład danej pizzy. Zobaczmy, jak takie rozwiązanie sprawdzi się w praktyce.

Zezwalamy klasom podrzędnym na podejmowanie decyzji

Pamiętaj, klasa Pizzeria ma już całkiem niezły system zamówień zrealizowany w metodzie zamówPizza(), a Ty chcesz się przekonać, że jest on spójny we wszystkich oddziałach pizzerii Twojej sieci.

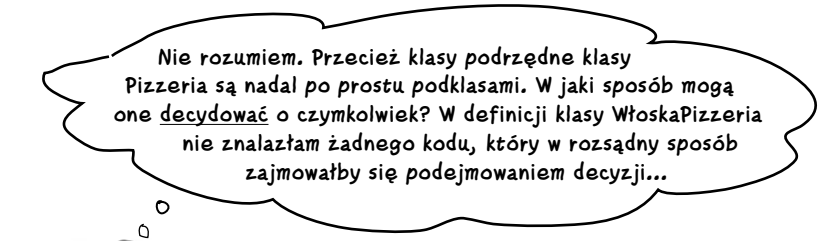
Główna różnica pomiędzy poszczególnymi lokalnymi oddziałami polega na tym, że każdy z nich tworzy swoje odmiany pizzy — pizza włoska jest robiona na cienkim cieście, pizza amerykańska ma ciasto grube i tak dalej — a my mamy teraz zamiar „wrzucić" te wszystkie wariacje na temat pizzy do jednej metody, utwórzPizza(), której zadaniem będzie tworzenie odpowiednich gatunków tego — skądinąd znakomitego — produktu. Przeprowadzimy to w ten sposób, że pozwolimy poszczególnym podklasom klasy Pizzeria decydować, jak będzie wyglądała metoda utwórzPizza(). Docelowo będziemy mieli zatem pewną ilość rzeczywistych klas podrzędnych klasy Pizzeria, z których każda będzie posiadała swoje własne warianty pizzy i z których każda będzie się mieściła w ogólnej strukturze Pizzerii, a mimo to będzie korzystała z dobrze dopasowanej do swoich potrzeb metody zamówPizza().




```
public Pizza utwórzPizza(type) {
if (type.equals("serowa")) {
        pizza = new WłoskaSerowaPizza();
    } else if (type.equals("pepperoni")) {
        pizza = new WłoskaPepperoniPizza();
    } else if (type.equals("owoce morza")) {
        pizza = new WłoskaOwoceMorzaPizza();
    } else if (type.equals("wegetariańska")) {
        pizza = new WłoskaWegetariańskaPizza();
    }
}
```

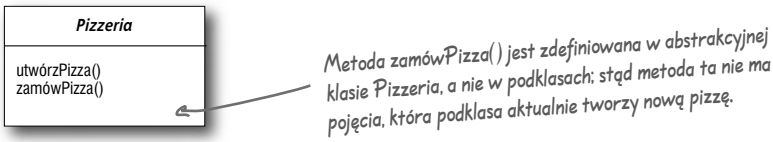
```
public Pizza utwórzPizza(type) {
if (type.equals("serowa")) {
        pizza = new AmerykańskaSerowaPizza();
    } else if (type.equals("pepperoni")) {
        pizza = new AmerykańskaPepperoniPizza();
    } else if (type.equals("owoce morza")) {
        pizza = new AmerykańskaOwoceMorzaPizza();
    } else if (type.equals("wegetariańska")) {
        pizza = new AmerykańskaWegetariańskaPizza();
    }
}
```

W jaki sposób podklasy mogą podejmować decyzje?

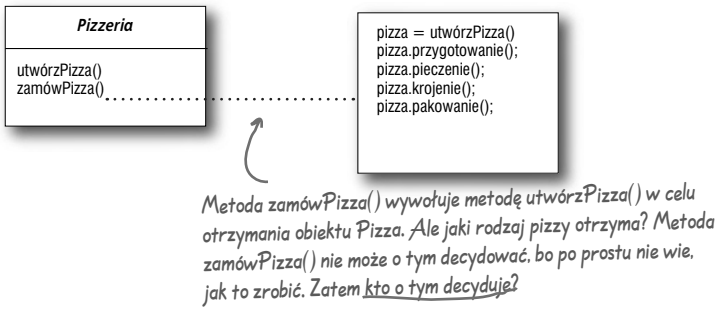




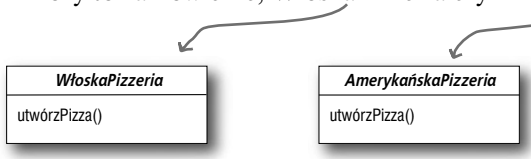
Spróbuj jeszcze raz przemyśleć to wszystko z punktu widzenia metody zamówPizza() zadeklarowanej w klasie Pizzeria: metoda ta jest zdefiniowana w klasie abstrakcyjnej, ale wszystkie typy rzeczywiste są tworzone wyłącznie w podklasach.



A teraz zwróć uwagę, że metoda zamówPizza() wykonuje wiele różnych operacji na obiekcie Pizza (takich jak przygotowanie, pieczenie, krojenie, pakowanie). Ponieważ jednak Pizza jest klasą abstrakcyjną, metoda zamówPizza() nie wie, jakie klasy rzeczywiste są zaangażowane w takie operacje. Innymi słowy, nie ma tutaj żadnego sprzężenia.



Kiedy metoda zamówPizza() wywołuje metodę utwórzPizza(), do realizacji tego zamówienia (czyli do tworzenia pizzy) zostanie wywołana jedna z klas podrzędnych. Jaki rodzaj pizzy zostanie utworzony? No cóż, zależy to od tego, który lokalny oddział pizzerii złoży to zamówienie, WłoskaPizzeria czy AmerykańskaPizzeria.



Zatem, czy decyzja o wyborze rodzaju tworzonej pizzy jest naprawdę podejmowana w czasie rzeczywistym? Oczywiście nie, ale patrząc na to z perspektywy metody zamówPizza(), jeżeli korzystamy z oddziału WłoskaPizzeria, to właśnie ta podklasa determinuje rodzaj tworzonej pizzy. Wynika stąd jasno, że poszczególne podklasy tak naprawdę nie „podejmują decyzji" — decydujesz *TY*, wybierając lokalny oddział pizzerii — niemniej jednak formalnie właśnie lokalny oddział pizzerii ma ostateczny wpływ na to, jaką pizzę otrzymasz.

Tworzymy Pizzerię

Bycie właścicielem sieci pizzerii ma swoje zalety — możesz narzucać własne standardy i rozwiązania. Poszczególne oddziały lokalne muszą utworzyć klasę podrzędną dziedziczącą po klasie Pizzeria i dostarczyć metodę utwórzPizza(), która będzie posiadała zaimplementowany sposób tworzenia danego rodzaju pizzy. Zajmiemy się teraz przygotowaniem odpowiednich procedur dla trzech rodzajów pizzy oferowanych przez trzy główne ośrodki regionalne.

Poniżej zamieściliśmy kod dla oddziału przygotowującego pizzę w stylu włoskim:

Metoda utwórzPizza() zwraca obiekt Pizza, a sama podklasa jest w pełni odpowiedzialna za to, obiekt której klasy rzeczywistej zostanie utworzony.

Klasa WłoskaPizzeria jest rozszerzeniem klasy Pizzeria(), stąd dziedziczy po niej (między innymi) metodę zamówPizza().

```
public class WłoskaPizzeria extends Pizzeria {

    Pizza utwórzPizza(String item) {
        if (item.equals("serowa")) {
            return new WłoskaSerowaPizza();
        } else if (item.equals("wegetariańska")) {
            return new WłoskaWegetariańskaPizza();
        } else if (item.equals("owoce morza")) {
            return new WłoskaOwoceMorzaPizza();
        } else if (item.equals("pepperoni")) {
            return new WłoskaPepperoniPizza();
        } else return null;
    }
}
```

Musimy tutaj zaimplementować metodę utwórzPizza(), ponieważ w klasie Pizzeria została ona zadeklarowana jako metoda abstrakcyjna.

W tym miejscu tworzone są obiekty naszych klas rzeczywistych. Zwróć uwagę, że dla każdego rodzaju pizzy jest tworzony jej włoski wariant.

Zauważ, że metoda zamówPizza() z klasy nadrzędnej nie ma pojęcia, jaka pizza zostanie utworzona; metoda ta „wie" tylko, że otrzymaną pizzę należy odpowiednio przygotować, upiec, pokroić i wreszcie zapakować!

Po zakończeniu procesu tworzenia kodu odpowiednich podklas będziemy mogli przekonać się w praktyce, jak wygląda zamówienie kilku pizz. Zanim jednak będzie to możliwe, spróbuj na następnej stronie samodzielnie napisać odpowiedni kod dla klas AmerykańskaPizzeria oraz MeksykańskaPizzeria.



Zaostrz ołówek

Kod klasy WłoskaPizzeria jest już gotowy; pozostały do zrobienia jeszcze tylko dwie klasy i będziemy mogli uruchomić pierwsze oddziały naszej sieci pizzerii! Poniżej napisz odpowiedni kod dla klas AmerykańskaPizzeria oraz MeksykańskaPizzeria:

Deklarowanie metody typu Fabryka

Dokonując zaledwie kilku stosunkowo prostych transformacji klasy Pizzeria, przeszliśmy od etapu, na którym obiekt Pizza był odpowiedzialny za tworzenie wybranych instancji klas rzeczywistych (czyli za tworzenie poszczególnych gatunków pizzy), do sytuacji, w której odpowiedzialność za to spada na zbiór klas podrzędnych, reprezentujących poszczególne lokalne oddziały pizzerii. Przyjrzyjmy się temu bliżej:

```
public abstract class Pizzeria {

    public Pizza zamówPizza(String type) {
        Pizza pizza;

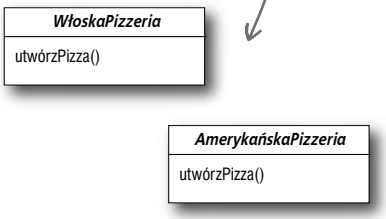
        pizza = utwórzPizza(type);

        pizza.przygotowanie();
        pizza.pieczenie();
        pizza.krojenie();
        pizza.pakowanie();

        return pizza;
    }
    protected abstract Pizza utwórzPizza(String type);

    // tutaj umieść inne metody
}
```

Tworzeniem instancji obiektów klas rzeczywistych zajmują się tutaj (poprzez metodę utwórzPizza()) poszczególne klasy podrzędne.



Cała odpowiedzialność za tworzenie obiektów Pizza została przesunięta do osobnej **metody**, która działa jako swego rodzaju **fabryka**.



Bliskie spotkania z kodem aplikacji

Metoda typu Fabryka obsługuje tworzenie obiektów i powoduje hermetyzację tego procesu w klasie podrzędnej. Takie rozwiązanie powoduje usunięcie sprzężenia pomiędzy kodem klienta w klasie nadrzędnej a kodem odpowiedzialnym za tworzenie obiektów, zlokalizowanym w klasie podrzędnej.

```
abstract Produkt metodaFabryka(String type)
```

Metoda typu Fabryka może być sparametryzowana (ale nie musi), co pozwala na dokonywanie wyboru spośród całej gamy produktów.

Metoda typu Fabryka jest metodą abstrakcyjną, dzięki czemu za obsługę tworzenia obiektów odpowiedzialne są klasy podrzędne.

Metoda typu Fabryka zwraca Produkt, który jest zazwyczaj wykorzystywany przez metody zdefiniowane w superklasie.

Metoda typu Fabryka powoduje, że kod klienta (czyli kod zlokalizowany w superklasie — np. w naszym przypadku metoda zamówPizza()) „nie wie", jaki rodzaj Produktu będzie aktualnie tworzony.

Zobaczmy, jak to działa: zamawianie pizzy przy pomocy metody typu Fabryka



A więc w jaki sposób oni składają zamówienia?

1. Przede wszystkim, Jacek i Eryk potrzebują instancji obiektu klasy Pizzeria. Jacek musi utworzyć obiekt klasy AmerykańskaPizzeria, a z kolei Eryk musi utworzyć obiekt klasy WłoskaPizzeria.
2. Mając odpowiednie obiekty klasy Pizzeria w ręku, zarówno Jacek, jak i Eryk mogą wywołać metodę zamówPizza() i przekazać do niej typ pizzy, jaką chcą otrzymać (serowa, wegetariańska, z owocami morza itd.).
3. W celu utworzenia pizzy wywoływana jest metoda utwórzPizza(),zdefiniowana w dwóch podklasach, WłoskaPizzeria i AmerykańskaPizzeria. Zgodnie z tym, co zdefiniowaliśmy wcześniej, metoda z klasy WłoskaPizzeria powoduje utworzenie obiektu (pizzy) włoskiej, a metoda z klasy AmerykańskaPizzeria powoduje utworzenie obiektu (pizzy) amerykańskiej. Zarówno w jednym, jak i w drugim przypadku do metody zamówPizza() zwracany jest obiekt typu Pizza.
4. Metoda zamówPizza() nie ma pojęcia, jaki rodzaj pizzy został utworzony, ale „wie”, że jest to pizza, którą trzeba teraz przygotować, upiec, pokroić i zapakować dla Eryka i Jacka.

Sprawdźmy, jak naprawdę przebiega proces tworzenia pizzy po otrzymaniu zamówienia...



Za kulisami

1 **Spróbujemy prześledzić zamówienie Eryka: najpierw musimy utworzyć obiekt klasy WłoskaPizzeria:**

```
Pizzeria włoskaPizzeria = new WłoskaPizzeria();
```

Tworzy nową instancję obiektu klasy WłoskaPizzeria.



włoskaPizzeria

2 **Mamy już odpowiedni oddział pizzerii, więc możemy złożyć zamówienie:**

```
włoskaPizzeria.zamówPizza("serowa");
```

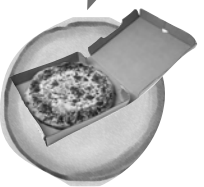
Wywoływana metoda zamówPizza() jest metodą instancji obiektu włoskaPizzeria (dziedziczoną z klasy bazowej Pizzeria).

utwórzPizza("serowa")

3 **Następnie metoda zamówPizza() wywołuje metodę utwórzPizza():**

```
Pizza pizza = utwórzPizza("serowa");
```

Pamiętaj, że metoda utwórzPizza() (czyli nasza fabryka) jest zaimplementowana w klasie podrzędnej. W tym przypadku jako rezultat jej działania zostanie zwrócona serowa pizza włoska.



Pizza

4 **Wreszcie mamy gotowy, utworzony półprodukt serowej pizzy włoskiej w ręku, więc metoda zamówPizza() przystępuje do finalizacji procesu dostarczania pizzy:**

```
pizza.przygotowanie();
pizza.pieczenie();
pizza.krojenie();
pizza.pakowanie();
```

Wszystkie te metody są zdefiniowane wewnątrz obiektu Pizza. Obiekt ten jest wynikiem działania metody-fabryki utwórzPizza(), zdefiniowanej w klasie WłoskaPizzeria.

Metoda zamówPizza() otrzymuje jako rezultat obiekt Pizza, aczkolwiek nie wie dokładnie, z jakiej klasy rzeczywistej on pochodzi.

Brakuje nam już tylko jednego elementu układanki: PIZZY!

Nasza Pizzeria nie będzie zbyt popularna wśród klientów, jeżeli nie będzie można kupić tam pizzy, więc najwyższy czas je zaimplementować



Rozpoczniemy od utworzenia klasy abstrakcyjnej Pizza, po której będą dziedziczyły wszystkie klasy rzeczywiste reprezentujące poszczególne gatunki pizzy.

Każdy gatunek pizzy posiada zdefiniowaną nazwę, rodzaj ciasta, rodzaj sosu oraz odpowiedni zestaw dodatków.

```
public abstract class Pizza {
    String nazwa;
    String ciasto;
    String sos;
    ArrayList dodatki = new ArrayList();

    void przygotowanie() {
        System.out.println("Przygotowywanie: " + nazwa);
        System.out.println("Wyrabianie ciasta...");
        System.out.println("Dodawanie sosu...");
        System.out.println("Dodatki: ");
        for (int i = 0; i < dodatki.size(); i++) {
            System.out.println("   " + dodatki.get(i));
        }
    }

    void pieczenie() {
        System.out.println("Pieczenie: 25 minut
              w temperaturze 350 stopni Celsjusza");
    }

    void krojenie() {
        System.out.println("Krojenie pizzy na 8 kawałków");
    }

    void pakowanie() {
        System.out.println("Pakowanie pizzy
              w oficjalne pudełko naszej sieci Pizzerii.");
    }

    public String pobierzNazwa() {
        return nazwa;
    }

}
```

Klasa abstrakcyjna zapewnia zestaw podstawowych, domyślnych procedur opisujących proces pieczenia, krojenia oraz pakowania pizzy.

Przygotowywanie pizzy wymaga realizacji kilku operacji w ściśle określonej kolejności.

PAMIĘTAJ: w naszych przykładach nie zamieszczamy fragmentów kodu odpowiedzialnych za import i obsługę poszczególnych pakietów bibliotek języka Java. Kompletny zestaw kodów źródłowych przykładów zamieszczonych w naszej książce możesz pobrać z serwera FTP wydawnictwa Helion: ftp://ftp.helion.pl/przyklady/wzorrg.zip

Teraz będziemy potrzebować kilku rzeczywistych klas podrzędnych... Co powiesz na to, abyśmy zdefiniowali klasy reprezentujące włoską i amerykańską odmianę pizzy serowej?

```
public class WłoskaSerowaPizza extends Pizza {

    public WłoskaSerowaPizza() {
        nazwa = "Włoska pizza serowa z sosem Marinara";
        ciasto = "Cienkie kruche ciasto";
        sos = "Sos Marinara";

        dodatki.add("Tarty ser Reggiano");
    }
}
```

Pizzeria włoska przygotowuje pizzę serową z sosem Marinara na cienkim, kruchym cieście.

Z jednym tylko dodatkiem: tartym serem Reggiano!

```
public class AmerykańskaSerowaPizza extends Pizza {

    public AmerykańskaSerowaPizza() {
        nazwa = "Amerykańska Pizza Serowa";
        ciasto = "Extra grube, chrupkie ciasto";
        sos = "Sos z pomidorów śliwkowych";

        dodatki.add("Grubo tarty ser Mozzarella");
    }

    void krojenie() {
        System.out.println("Krojenie pizzy na kwadratowe kawałki");
    }
}
```

Pizzeria amerykańska przygotowuje pizzę serową na bazie ekstragrubego, chrupkiego ciasta i sosu z pomidorów śliwkowych.

Do amerykańskiej pizzy serowej należy dodawać ogromne ilości grubo tartego sera mozzarella!

Pizzeria amerykańska wymaga krojenia pizzy na kwadratowe kawałki, stąd podklasa AmerykańskaSerowaPizza implementuje metodę krojenie(), która przesłania swój odpowiednik z superklasy.

Czekałeś już wystarczająco długo, zatem najwyższy czas na pizzę!

```
public class PizzaTest {

    public static void main(String[] args) {
        Pizzeria włoska = new WłoskaPizzeria();
        Pizzeria amerykańska = new AmerykańskaPizzeria();

        Pizza pizza = włoska.zamówPizza("serowa");
        System.out.println("Eryk zamówił: " + pizza.pobierzNazwa() + "\n");

        pizza = amerykańska.zamówPizza("serowa");
        System.out.println("Jacek zamówił: " + pizza.pobierzNazwa() + "\n);
    }
}
```

Najpierw tworzymy dwa różne lokalne oddziały pizzerii.

Następnie używamy jednego z nich do realizacji zamówienia złożonego przez Eryka.

A drugiego do realizacji zamówienia złożonego przez Jacka.

```
C:\WINDOWS\System32\cmd.exe

> java PizzaTest
Przygotowywanie: Włoska pizza serowa z sosem Marinara
Wyrabianie ciasta...
Dodawanie sosu...
Dodatki:
   Tarty ser Reggiano
Pieczenie: 25 minut w temperaturze 350 stopni Celsjusza
Krojenie pizzy na 8 kawałków
Pakowanie pizzy w oficjalne pudełko naszej sieci Pizzerii.
Eryk zamówił: Włoska pizza serowa z sosem Marinara

Przygotowywanie: Amerykańska pizza serowa
Wyrabianie ciasta...
Dodawanie sosu...
Dodatki:
   Gruby tarty ser Mozzarella
Pieczenie: 25 minut w temperaturze 350 stopni Celsjusza
Krojenie pizzy na kwadratowe kawałki
Pakowanie pizzy w oficjalne pudełko naszej sieci Pizzerii.
Jacek zamówił: Amerykańska pizza serowa

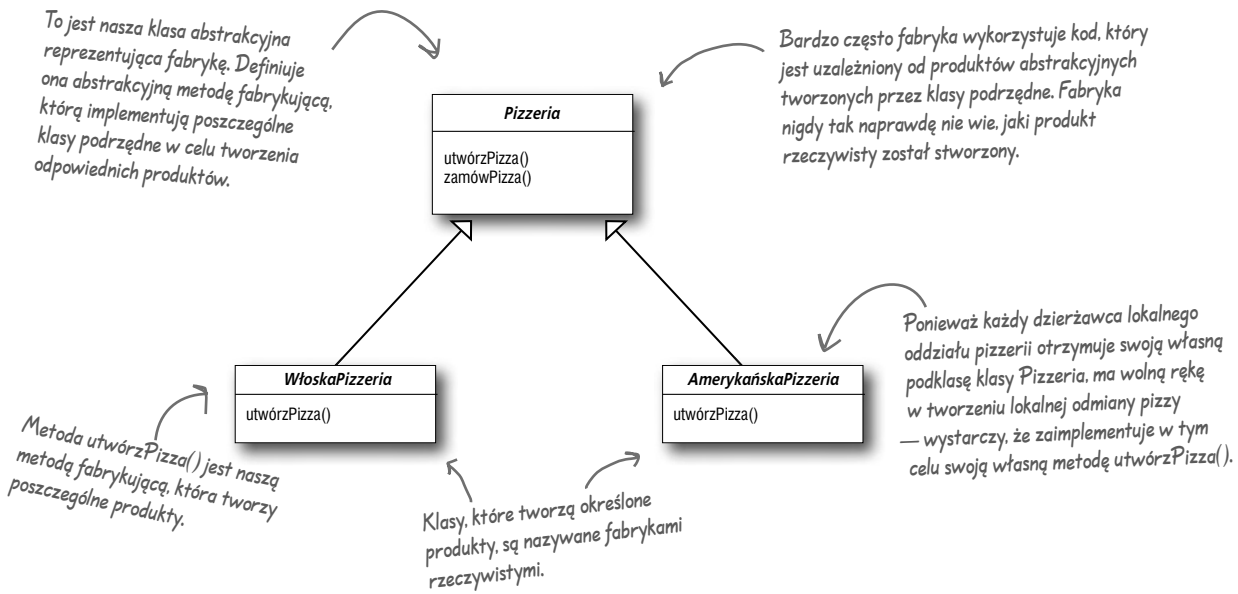
>
```

Obydwie pizze zostają przygotowane, wzbogacone dodatkami, upieczone, pocięte i zapakowane. Nasza klasa bazowa wcale nie musi znać szczegółów, wszystkim zajmuje się odpowiednia klasa podrzędna poprzez utworzenie obiektu reprezentującego daną pizzę.

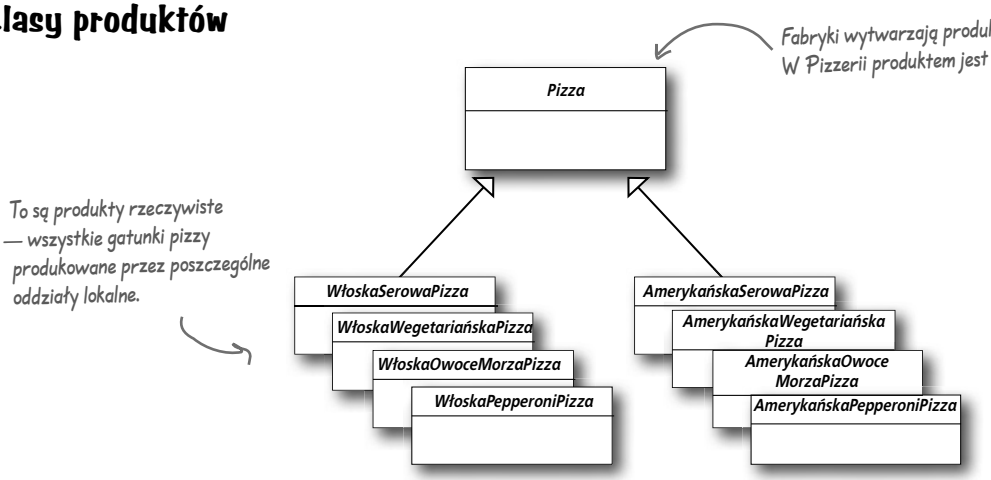
Wreszcie nadszedł czas na spotkanie z wzorcem Metoda Fabrykująca (ang. *Factory Method*)

Wszystkie wzorce typu Fabryka wykorzystują hermetyzację procesu tworzenia obiektów. Wzorzec metoda Fabrykująca hermetyzuje ten proces poprzez pozwolenie klasom podrzędnym na decydowanie, jaki obiekt zostanie utworzony. Przeanalizujmy teraz diagram klas, aby poznać głównych graczy ligi tego wzorca:

Klasy fabryki



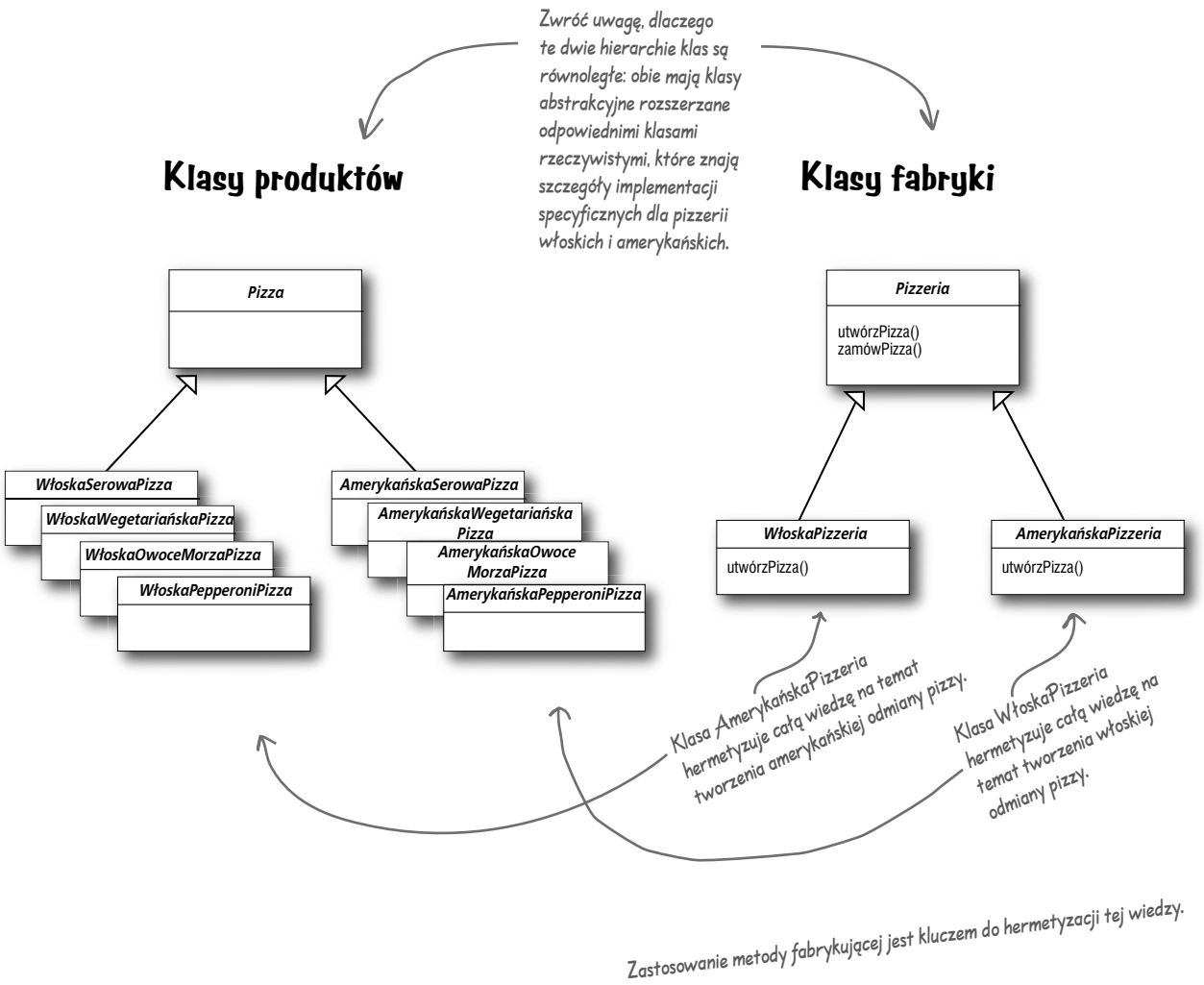
Klasy produktów



Z innej perspektywy: równoległa hierarchia klas

Jak zauważyłeś, Metoda Fabrykująca zapewnia stworzenie odpowiedniej struktury klas poprzez ustanowienie metody zamówPizza() połączonej z metodą fabrykującą. Warto również zwrócić uwagę, w jaki sposób ten wzorzec hermetyzuje „wiedzę" o sposobie tworzenia produktów w poszczególnych konstruktorach klas.

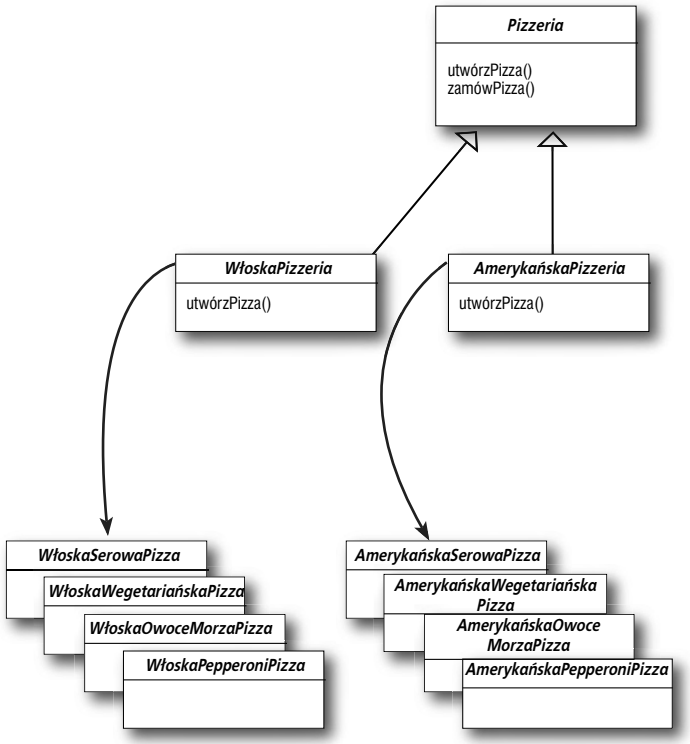
Rzućmy okiem na dwie równoległe hierarchie klas i zobaczmy, w jaki sposób są one ze sobą powiązane:





Łamigłówka projektowa

Będziemy potrzebować kolejnej odmiany dla szalonych koneserów pizzy meksykańskiej (oczywiście, „szalonych” w dobrym tego słowa znaczeniu). Narysuj zatem kolejną, równoległą hierarchię klas, która będzie niezbędna, aby rozszerzyć działalność Twojej sieci pizzerii na obszar Meksyku.



Tutaj umieść swój rysunek...

OK, a teraz wypisz poniżej pięć *najbardziej dziwacznych* dodatków do pizzy, jakie tylko jesteś sobie w stanie wyobrazić. Po zakończeniu będziesz naprawdę gotowy, aby uruchomić swój biznes w oddziale meksykańskim.

Definicja wzorca Metoda Fabrykująca

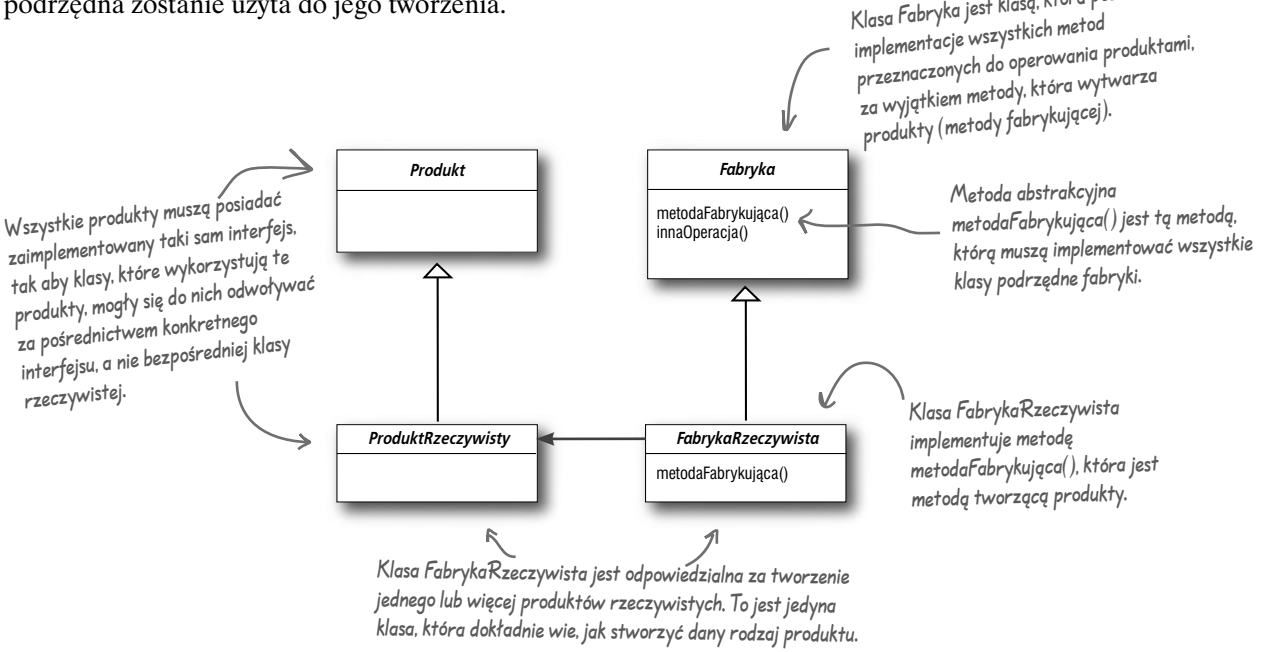
Nadszedł czas, abyśmy zaprezentowali oficjalną definicję wzorca Metoda Fabrykująca:

> **Wzorzec Metoda Fabrykująca** definiuje interfejs pozwalający na tworzenie obiektów, ale pozwala klasom podrzędnym decydować, jakiej klasy obiekt zostanie utworzony. Wzorzec Metoda Fabrykująca przekazuje odpowiedzialność za tworzenie obiektów do klas podrzędnych.

Tak jak w przypadku każdej innej fabryki, wzorzec Metoda Fabrykująca daje możliwość hermetyzacji tworzenia obiektów typów rzeczywistych. Patrząc na zamieszczony poniżej diagram klas, możesz zauważyć, że abstrakcyjna klasa fabryki daje Ci interfejs wyposażony w metodę umożliwiającą tworzenie obiektów, znaną również pod nazwą „metody fabrykującej". Wszystkie inne metody zaimplementowane w abstrakcyjnej klasie fabryki mają za zadanie operować na produktach tworzonych przez metodę fabrykującą. Tylko klasy podrzędne implementują metodę fabrykującą i dzięki temu mają możliwość tworzenia określonych produktów.

W zasadzie mógłbyś zapytać ich, co rozumieją przez słowo „decyduje", ale możemy się założyć, że w chwili obecnej rozumiesz to zagadnienie znacznie lepiej od nich!

Nie dziw się, jeśli bardzo często będziesz słyszał, że programiści w rozmowie między sobą mówią, że wzorzec Metoda Fabrykująca pozwala klasom podrzędnym decydować, jakiej klasy obiekt zostanie utworzony. Mówią oni tak nie dlatego, że wzorzec pozwala na faktyczne podejmowanie decyzji przez klasy podrzędne w czasie działania programu, ale dlatego, że klasa fabryki jest napisana w ten sposób, że nie ma pojęcia, jakie produkty zostaną stworzone; wybór produktu jest determinowany tym, jaka klasa podrzędna zostanie użyta do jego tworzenia.



Nie istnieją głupie pytania

P: Jakie są zalety stosowania wzorca Metoda Fabrykująca w sytuacji, kiedy masz tylko jedną klasę typu FabrykaRzeczywista?

O: Wzorzec Metoda Fabrykująca jest użyteczny nawet, kiedy masz tylko jedną fabrykę rzeczywistą, ponieważ korzystając z niego, usuwasz sprzężenie pomiędzy implementacją produktu a jego zastosowaniem. Jeżeli będziesz dodawał nowe produkty bądź zmieniał implementacje wybranych z nich, nie będzie to miało żadnego wpływu na fabrykę (ponieważ klasa Fabryka nie jest w żaden sposób mocno związana z żadnym produktem rzeczywistym).

P: Czy poprawne będzie stwierdzenie, że nasze pizzerie, włoska i amerykańska, zostały zaimplementowane przy użyciu „wzorca" ProstaFabryka? Przecież tak właśnie wyglądają...

O: Wyglądają faktycznie bardzo podobnie, aczkolwiek są wykorzystywane w inny sposób. Pomimo, iż implementacje poszczególnych oddziałów lokalnych wyglądają bardzo podobnie do tych z przykładu ProstaFabrykaPizzy, pamiętaj o tym, że stanowią one rozszerzenie klasy, w której metoda utwórzPizza() została zadeklarowana jako abstrakcyjna. Zdefiniowanie zachowania metody utwórzPizza() jest zadaniem dla poszczególnych oddziałów lokalnych. W przykładzie ProstaFabryka, fabryka jest po prostu kolejnym obiektem, który jest komponowany z klasą Pizzeria.

P: Czy metoda fabrykująca i sama fabryka są zawsze abstrakcyjne?

O: Nie, metoda fabrykująca może zostać tak zdefiniowana, aby tworzyła pewne wybrane produkty rzeczywiste. Dzięki takiemu rozwiązaniu zawsze będziesz miał możliwość kreowania pewnych produktów, nawet jeżeli nie zostaną stworzone żadne podklasy fabryki.

P: Każdy oddział pizzerii może serwować cztery różne rodzaje pizzy, opierając się na argumencie przekazywanym podczas zamówienia. Czy wszystkie fabryki rzeczywiste mogą tworzyć różne rodzaje produktów, czy też czasami mogą być przeznaczone do przygotowania tylko jednego produktu?

O: Rozwiązanie, które zaimplementowaliśmy w naszym przykładzie, jest znane jako parametryzowana metoda fabrykująca. W zależności od wartości przekazywanego do niej parametru może ona tworzyć więcej niż jeden rodzaj obiektów. Nie zmienia to jednak w niczym faktu, że bardzo często takie fabryki produkują tylko jeden rodzaj obiektów i nie są w żaden sposób parametryzowane. Oba rodzaje fabryk są pełnoprawnymi formami wzorca Metoda Fabrykująca.

P: Ale wygląda na to, że parametryzacja naszej metody nie posiada zabezpieczeń na zgodność przekazywanych typów. Przecież przekazujemy po prostu zmienną typu String! Co się stanie, jeżeli w zamówieniu poproszę o pizzę salami?

O: Masz oczywiście rację, takie postępowanie spowodowałoby wygenerowanie czegoś, co w naszym środowisku nosi nazwę „błędu czasu wykonania" (ang. *runtime error*). Istnieje co najmniej kilka różnych, dosyć wyrafinowanych technik, które mogą być użyte do zabezpieczenia zgodności przekazywanych typów, lub też, innymi słowy, do zapewnienia, że błędy w przekazywanych parametrach zostaną wyłapane już na etapie kompilowania kodu. Przykładowo, możesz stworzyć obiekt, który będzie reprezentował dozwolone typy parametrów, możesz skorzystać ze statycznych zmiennych finalnych lub też, jeżeli używasz Java 2 Platform Standard Edition 5.0, możesz skorzystać z typów wyliczeniowych *enum*.

P: Nadal nie do końca jednak rozumiem, jakie są różnice pomiędzy prostą fabryką a metodą fabrykującą. Wyglądają bardzo podobnie, z wyjątkiem tego, że w metodzie fabrykującej pizza jest zwracana (produkowana) przez klasę podrzędną. Czy możesz to szerzej wyjaśnić?

O: Masz zupełną rację, mówiąc, że rozwiązanie z podklasami wygląda całkiem podobnie do prostej fabryki, aczkolwiek powinieneś traktować prostą fabrykę jako rozwiązanie jednorazowe, podczas gdy metoda fabrykująca daje Ci pełną strukturę, która pozwala klasom podrzędnym decydować, jaka implementacja zostanie użyta. Przykładowo, metoda zamówPizza() tworzy ogólną konstrukcję pozwalającą na powstawanie pizzy w ten sposób, że zgodnie z wymaganiami metody fabrykującej tworzone są odrębne klasy rzeczywiste, bezpośrednio zaangażowane w przygotowywanie pizzy. Poprzez tworzenie podklas klasy Pizzeria możesz decydować, jakie produkty rzeczywiste będą zwracane do metody zamówPizza(). Porównaj to z rozwiązaniem, jakie oferuje prosta fabryka. Daje ona, co prawda, możliwość hermetyzacji procesu tworzenia obiektów, ale nie udostępnia takiej elastyczności, jaką zapewnia metoda fabrykująca, ponieważ nie będziesz miał możliwości wprowadzenia różnorodności tworzonych obiektów.



Mistrz i uczeń...

Mistrz: Powiedz, jak przebiega Twoja nauka, mój młody adepcie?

Uczeń: Pogłębiałem wiedzę, którą zamknąłeś kiedyś w maksymie: „Poddawaj hermetyzacji to, co się zmienia", mój Mistrzu.

Mistrz: Mów dalej...

Uczeń: Nauczyłem się już, że można poddawać hermetyzacji kod odpowiedzialny za tworzenie obiektów. Kiedy korzystamy z kodu, którego zadaniem jest przygotowanie obiektów konkretnych klas, zazwyczaj jest to obszar kodu, w którym często mogą zachodzić różne zmiany. Poznałem również technikę zwaną „fabryką", która pozwala właśnie na taką hermetyzację procesu tworzenia obiektów.

Mistrz: No dobrze, a nawiązując do tych „fabryk": jakie są zalety ich używania?

Uczeń: Jest ich wiele. Umieszczając kod odpowiedzialny za tworzenie obiektów w jednym obiekcie lub metodzie, mogę uniknąć duplikowania kodu i zapewnić, że całe jego serwisowanie będzie się odbywało w jednym miejscu. Oznacza to również, że kod klienta musi współpracować tylko z odpowiednimi interfejsami i nie jest zależny od implementacji klas rzeczywistych niezbędnych do utworzenia danego obiektu. Jak już zdążyłem się wcześniej nauczyć, takie rozwiązanie pozwala mi skoncentrować się na tworzeniu interfejsu, a nie implementacji, co powoduje, że mój kod staje się coraz bardziej elastyczny i łatwy do rozbudowania w przyszłości.

Mistrz: Taaak, widzę, że wykształcasz w sobie coraz lepsze instynkty obiektowe. Czy masz jeszcze na dzisiaj jakieś pytania do mnie?

Uczeń: Mistrzu, wiem, że poprzez hermetyzację procesu tworzenia obiektów produkuję kod, który wykorzystuje klasy abstrakcyjne, i dzięki temu usuwam sprzężenia pomiędzy kodem klienta a implementacjami klas rzeczywistych. Ale jednak mimo to kod mojej fabryki musi ciągle używać klas rzeczywistych do tworzenia instancji prawdziwych obiektów. Czy ja przypadkiem nie próbuję sam sobie zamydlić oczu?

Mistrz: Mój młody człowieku, tworzenie obiektów jest nieodłącznie związane z szarą rzeczywistością naszego codziennego życia; musimy tworzyć obiekty albo nigdy nie bylibyśmy w stanie napisać najprostszego programu w języku Java. Ale dogłębna znajomość naszej rzeczywistości pozwala nam na takie zaprojektowanie kodu, że proces kreacji obiektów nie będzie miał już nic wspólnego z mydleniem oczu — zostanie po prostu odpowiednio ujarzmiony i zabezpieczony, a my będziemy mogli odpowiednio o niego dbać. Jeżeli jednak tworzenie kodu wymknie się spod kontroli i zostanie puszczone na żywioł, wówczas, niestety, zazwyczaj nie będziemy mogli zebrać owoców naszej pracy.

Uczeń: Mistrzu, widzę głęboką prawdę w Twoich słowach.

Mistrz: Miałem nadzieję, że tak właśnie będzie. A teraz idź już, proszę, i oddaj się medytacjom na temat zależności pomiędzy obiektami.

Pizzeria mocno uzależniona



Zaostrz ołówek

Załóżmy, że nigdy w życiu nie słyszałeś o fabrykach obiektów i metodach fabrykujących. Poniżej zamieściliśmy wersję klasy Pizzeria, która nie korzysta z fabryki; postaraj się policzyć, od ilu rzeczywistych obiektów klasy Pizza ta klasa jest uzależniona. Jeżeli teraz dodamy do oferty pizzę meksykańską, to o ile zmieni się ilość klas?

```
public class PizzeriaMocnoUzależniona{

    public Pizza utwórzPizza(String style, String type) {
        Pizza pizza = null;
        if (style.equals("Włoska")) {
            if (type.equals("serowa")) {
                pizza = new WłoskaSerowaPizza();
            } else if (type.equals("wegetariańska")) {
                pizza = new WłoskaWegetariańskaPizza();
            } else if (type.equals("owoce morza")) {
                pizza = new WłoskaOwoceMorzaPizza();
            } else if (type.equals("pepperoni")) {
                pizza = new WłoskaPepperoniPizza();
            }
        } else if (style.equals("Amerykańska")) {
            if (type.equals("serowa")) {
                pizza = new AmerykańskaSerowaPizza();
            } else if (type.equals("wegetariańska")) {
                pizza = new AmerykańskaWegetariańskaPizza();
            } else if (type.equals("owoce morza")) {
                pizza = new AmerykańskaOwoceMorzaPizza();
            } else if (type.equals("pepperoni")) {
                pizza = new AmerykańskaPepperoniPizza();
            }
        } else {
            System.out.println("Błąd: zamówiono niepoprawny rodzaj pizzy.");
            return null;
        }
        pizza.przygotowanie();
        pizza.pieczenie();
        pizza.krojenie();
        pizza.pakowanie();
        return pizza;
    }
}
```

Obsługuje wszystkie gatunki pizzy przyrządzane na sposób włoski.

Obsługuje wszystkie gatunki pizzy przyrządzane na sposób amerykański.

Swoje odpowiedzi możesz wpisać tutaj:

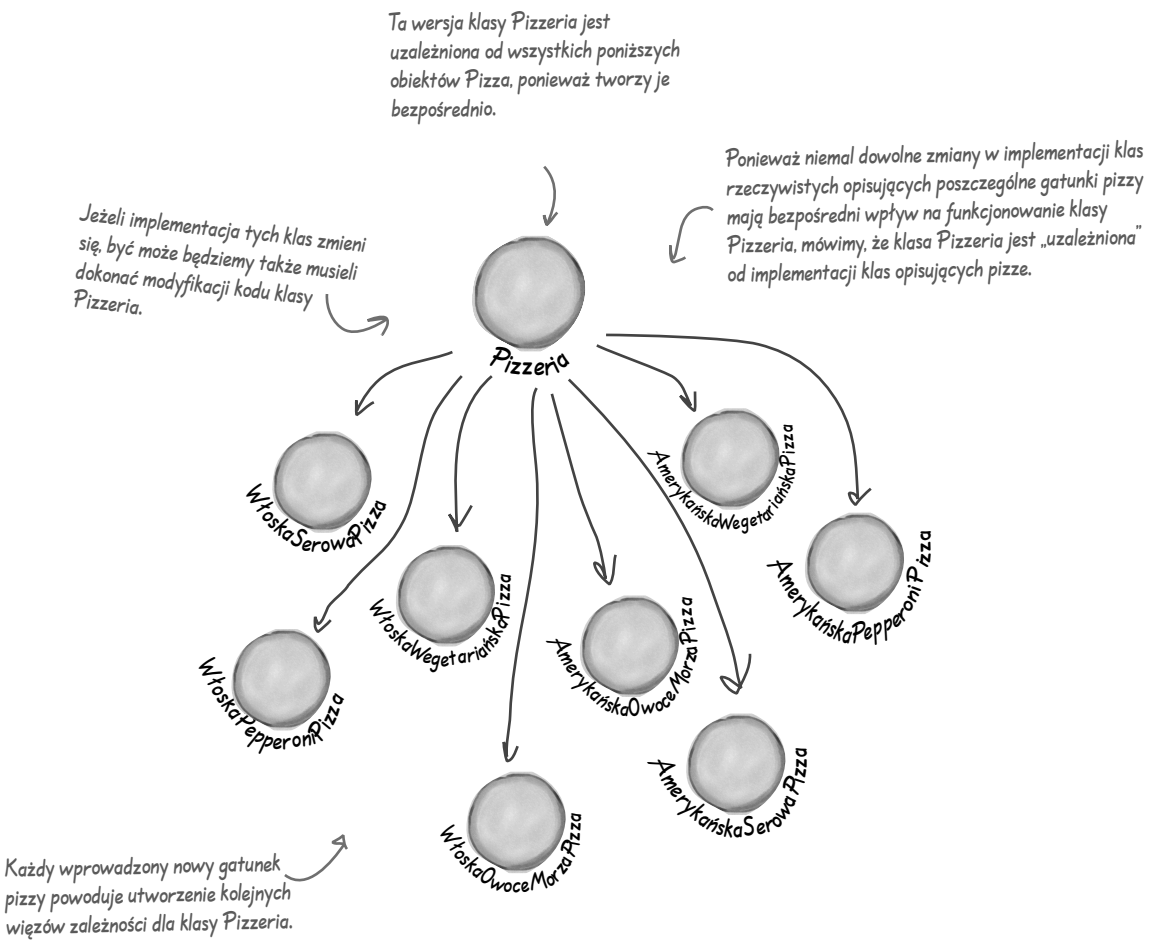
ilość _____

ilość po wprowadzeniu pizzy meksykańskiej _____

Sprawdzamy zależności pomiędzy obiektami

Kiedy bezpośrednio tworzysz obiekt danej klasy, w pewien sposób uzależniasz się od definicji tej klasy rzeczywistej. Rzućmy jeszcze raz okiem na znajdujący się na poprzedniej stronie kod klasy PizzeriaMocnoUzależniona. Wszystkie obiekty Pizza są tworzone wewnątrz tej klasy, zamiast zostać oddelegowane do odpowiedniej fabryki.

Jeżeli narysujemy diagram reprezentujący taką wersję klasy Pizzeria wraz ze wszystkimi obiektami, od których jest ona uzależniona, będzie on wyglądał mniej więcej tak:



Reguła odwracania zależności

Oczywiście, w chwili obecnej powinno być już dla wszystkich zupełnie jasne, że redukcja zależności kodu od implementacji klas rzeczywistych jest dla każdego programu „rzeczą dobrą i pożądaną". I to do tego stopnia, że w programowaniu zorientowanym obiektowo istnieje osobna reguła, która formalizuje ten zamiar; co więcej, reguła ta ma również swoją dużą, formalną nazwę — jest to r*eguła odwracania zależności* (ang. DIP — *Dependency Inversion Principle*) lub po prostu reguła DIP.

Oto kolejna sentencja, dzięki której możesz błysnąć na spotkaniu ze swoimi przełożonymi! Podwyżka, jaką w związku z tym otrzymasz, będzie z pewnością dużo większa niż koszt niniejszej książki, ale pomyśl również o tym, jaki podziw i szacunek wzbudzisz wśród swoich kolegów.

Zatem oto formalna, ogólna definicja naszej reguły:



Reguła projektowania

Uzależniaj kod od abstrakcji, a nie od klas rzeczywistych.

Na pierwszy rzut oka ta reguła brzmi dosyć podobnie do reguły: „skoncentruj się na tworzeniu interfejsów, a nie implementacji", prawda? Faktycznie jest podobna, aczkolwiek reguła DIP kładzie jeszcze większy nacisk na tworzenie składników abstrakcyjnych. Sugeruje, że składniki wysokiego poziomu nie powinny być uzależnione od składników niskopoziomowych; zamiast tego składniki na *obu* poziomach powinny zależeć wyłącznie od elementów abstrakcyjnych.

Składnikiem „wysokiego poziomu" jest klasa, której zachowania zostały zdefiniowane przy użyciu zachowań innych składników (składników „niskiego poziomu").

Przykładowo, klasa Pizzeria jest składnikiem wysokiego poziomu, ponieważ jej zachowania są determinowane przez poszczególne gatunki pizzy — tworzy różne obiekty klasy Pizza, zajmuje się ich przygotowaniem, pieczeniem, krojeniem i pakowaniem; jak łatwo się zorientować, składnikami niskiego poziomu są klasy opisujące poszczególne gatunki pizzy.

Ale co, u licha ciężkiego, to wszystko znaczy w praktyce?

No cóż, rozpocznijmy od ponownego spojrzenia na diagram reprezentujący mocno uzależnioną wersję klasy Pizzeria (znajdujący się na poprzedniej stronie). Klasa Pizzeria jest naszym „składnikiem wysokiego poziomu", a implementacje klas opisujących poszczególne gatunki pizzy są naszymi „składnikami niskiego poziomu" — i jak to jasno wynika z diagramu, klasa Pizzeria jest mocno uzależniona od implementacji klas opisujących gatunki pizzy.

Z kolei reguła projektowania mówi nam, że powinniśmy tworzyć nasz kod w taki sposób, aby był uzależniony od klas abstrakcyjnych, a nie rzeczywistych. Reguła taka obowiązuje zarówno dla modułów wysokiego, jak i niskiego poziomu.

Ale pozostaje pytanie, jak mamy tego dokonać? Pomyśl, w jaki sposób możemy zaaplikować regułę DIP do implementacji klasy PizzeriaMocnoUzależniona…

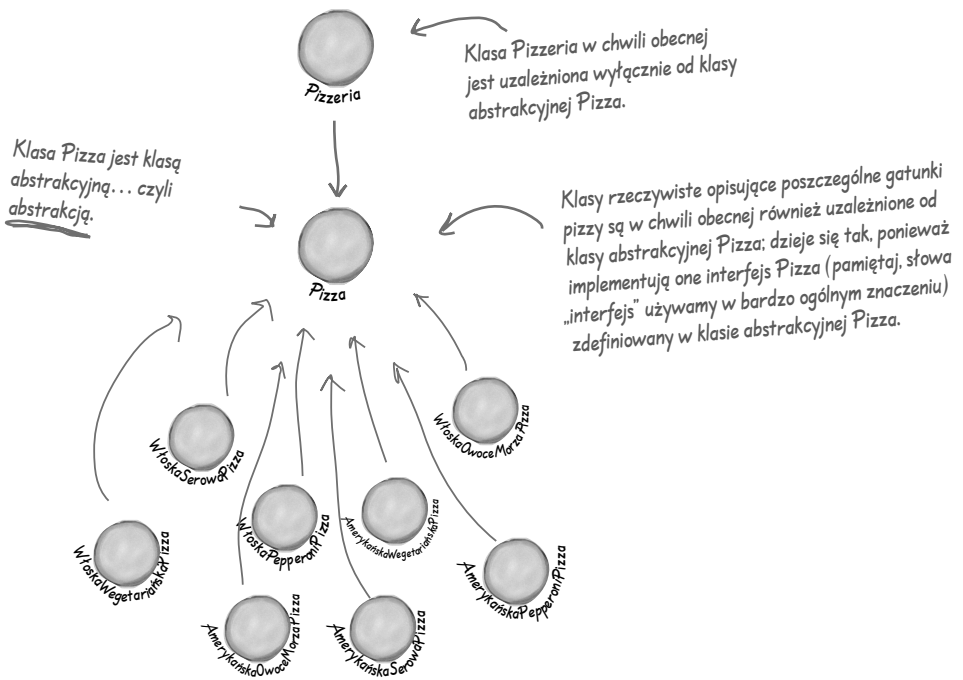
Zastosowanie reguły DIP

Głównym problemem, jaki trapi naszą mocno uzależnioną klasę Pizzeria, jest fakt, że jest uzależniona od każdego bez wyjątku rodzaju pizzy, ponieważ jej metoda zamówPizza() faktycznie tworzy kolejne instancje obiektów poszczególnych klas opisujących gatunki pizzy.

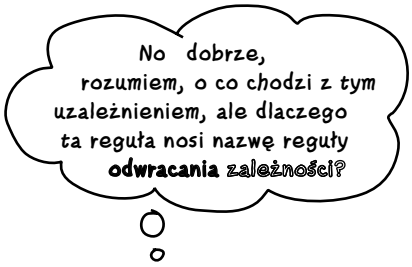
Pomimo, iż utworzyliśmy teraz klasę abstrakcyjną Pizza, nadal tworzymy instancje obiektów Pizza w jej kodzie, więc takie rozwiązanie nie przynosi nam zbyt wielu korzyści.

W jaki sposób możemy zatem wyprowadzić tworzenie instancji obiektów klas opisujących poszczególne gatunki pizzy poza kod metody zamówPizza()? No cóż, jak już dobrze wiemy, właśnie zastosowanie metody fabrykującej pozwala nam na osiągnięcie tego celu.

Zatem po zastosowaniu metody fabrykującej nasz diagram będzie wyglądał następująco:



Zauważ, że po zastosowaniu metody fabrykującej zarówno nasz składnik wysokiego poziomu (czyli klasa Pizzeria), jak i składniki niskiego poziomu (czyli klasy rzeczywiste opisujące poszczególne gatunki pizzy) są uzależnione od abstrakcji (czyli klasy abstrakcyjnej Pizza). Zastosowanie metody fabrykującej stanowi nie tylko technikę pozwalającą pozostawać w zgodzie z wymogami reguły DIP, ale jest również jedną z najbardziej efektywnych technik programowania.



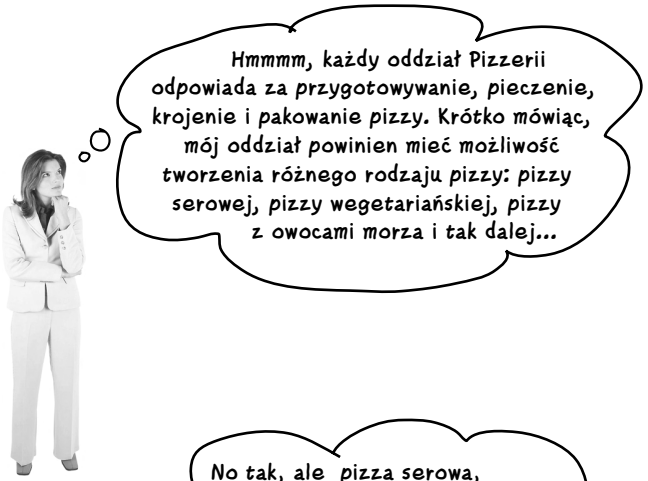


Gdzie jest miejsce na „odwracanie” w regule odwracania zależności?

Owo tytułowe „odwracanie” w nazwie reguły DIP bierze się stąd, że powoduje ona „odwrócenie” typowego sposobu myślenia o programowaniu zorientowanym obiektowo. Jeśli wrócisz na chwilę do diagramu zamieszczonego na poprzedniej stronie, z pewnością zauważysz, że składniki niskiego poziomu w chwili obecnej są uzależnione od elementu abstrakcyjnego znajdującego się na wyższym poziomie hierarchii; podobnie składnik wysokiego poziomu jest „przywiązany” do tego samego elementu abstrakcyjnego. Krótko mówiąc, diagram zależności od-góry-do-dołu, który narysowaliśmy kilka stron temu, odwrócił się tak, że w chwili obecnej zarówno składnik wysokiego poziomu, jak i składniki niskiego poziomu są uzależnione od tego samego elementu abstrakcyjnego.

Spróbujemy teraz spojrzeć na proces projektowania w zupełnie inny, niezbyt typowy sposób, i zobaczymy, jak zastosowanie reguły DIP może odwrócić tok naszego myślenia o całym projekcie…

Odwróć swój tok myślenia...



OK, zatem chcesz zaimplementować Pizzerię. Jaka myśl pojawia się najpierw w Twojej głowie?

Dobrze, rozpoczynasz na samej górze i schodzisz kolejno w dół ku poszczególnym klasom rzeczywistym. Ale — jak już to pokazaliśmy wcześniej — nie chcesz, aby Twoja Pizzeria znała szczegóły tworzenia poszczególnych gatunków pizzy, ponieważ będzie wówczas zbyt silnie uzależniona od implementacji klas rzeczywistych opisujących te gatunki!



A teraz spróbuj „odwrócić” swój tok myślenia… zamiast rozpocząć proces projektowania od góry, rozpocznij od poziomu poszczególnych gatunków pizzy i zastanów się, gdzie można umieścić element abstrakcyjny.

Świetnie! Myślisz, że do spełniania roli elementu abstrakcyjnego świetnie nadaje się *Pizza*. A teraz powróć na początek i jeszcze raz przemyśl cały projekt naszej Pizzerii.



No i temat zamknięty. Ale pamiętaj, że aby wydostać klasy rzeczywiste opisujące poszczególne gatunki pizzy z Pizzerii, będziesz musiał polegać na odpowiedniej fabryce. Kiedy już tego dokonasz, klasy rzeczywiste opisujące poszczególne gatunki pizzy będą uzależnione wyłącznie od elementu abstrakcyjnego, a takoż i sama klasa Pizzeria będzie wykazywała identyczną zależność. Krótko mówiąc, rozpoczęliśmy od klasycznego projektu, w którym nasza Pizzeria była uzależniona od wielu klas rzeczywistych, a następnie dokonaliśmy „odwrócenia” tych zależności (a przy okazji i naszego toku myślenia).

Kilka wskazówek, które mogą Ci pomóc w postępowaniu zgodnie z regułą...

Poniżej zamieściliśmy kilka wskazówek, które pomogą Ci unikać tworzenia projektów, które stoją w sprzeczności z regułą odwracania zależności:

- Żadna zmienna nie powinna przechowywać odwołania do klasy rzeczywistej.

*Jeżeli skorzystasz z operatora **new**, otrzymasz w zamian odwołanie do klasy rzeczywistej. Aby tego uniknąć, powinieneś skorzystać z fabryki!*

- Żadna klasa nie powinna dziedziczyć z klasy rzeczywistej.

Jeżeli dziedziczysz po klasie rzeczywistej, siłą rzeczy jesteś od niej uzależniony. Staraj się dziedziczyć po elementach abstrakcyjnych, takich jak interfejsy czy klasy abstrakcyjne.

- Żadna metoda nie powinna przesłaniać metody zaimplementowanej w dowolnej z jej klas bazowych.

Jeżeli przesłaniasz zaimplementowaną metodę, to znaczy, że Twoja klasa bazowa nie była odpowiednim elementem abstrakcyjnym, od którego powinieneś zaczynać. Metody zaimplementowane w klasie bazowej są tak pomyślane, aby mogły być używane przez wszystkie klasy podrzędne.

Momencik, ale czy przypadkiem postępowanie w całkowitej zgodności z tymi wskazówkami nie należy do gatunku rzeczy niemożliwych? Jeżeli będę chciała zawsze się ich trzymać, nie będę w stanie napisać nawet najprostszego programiku!



Masz całkowitą rację! Podobnie, jak ma to miejsce w przypadku wielu innych reguł, podane wskazówki należy traktować jako coś, do czego trzeba dążyć, a nie jako coś, czego trzeba zawsze bezwzględnie przestrzegać. Przecież jest zupełnie oczywiste, że każdy program napisany w języku Java będzie zawsze łamał takie wytyczne!

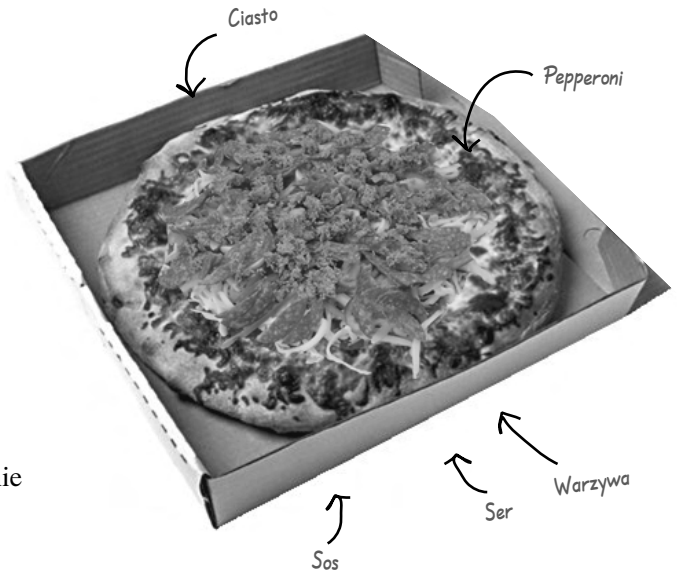
Jeżeli jednak będziesz w stanie przyswoić sobie nasze wskazówki i podczas tworzenia programów będziesz je zawsze mieć na uwadze, po prostu będziesz wiedzieć, że w danym miejscu nie postępujesz zgodnie z regułami. Zapewne będą istniały dobre powody, aby właśnie tak postępować. Przykładowo, jeżeli masz daną klasę, która raczej nie powinna się w przyszłości zmieniać (a Ty dobrze o tym wiesz), stworzenie wprost instancji obiektu tej konkretnej klasy naprawdę nie spowoduje, że świat się zawali. Pomyśl o tym; obiekty typu String tworzyliśmy zawsze i wszędzie, nawet się nad tym specjalnie nie zastanawiając. Czy takie postępowanie stanowi pogwałcenie reguły? Oczywiście! A czy takie postępowanie jest w porządku? Oczywiście! Dlaczego? Ponieważ jest bardzo mało prawdopodobne, aby definicja klasy String kiedykolwiek uległa zmianie.

Z drugiej strony, jeżeli masz daną klasę i wiesz, że prawie na pewno (bądź na pewno) będzie ona podlegała zmianom, masz do dyspozycji kilka dobrych, sprawdzonych technik, takich jak metoda fabrykująca, które pozwolą Ci dokonać hermetyzacji takich zmian.

A w międzyczasie, na zapleczu Pizzerii...

Projekt naszej Pizzerii zaczyna coraz szybciej nabierać ostatecznego kształtu: posiada elastyczną strukturę jak również jest w dużej mierze zgodny z różnym regułami projektowania.

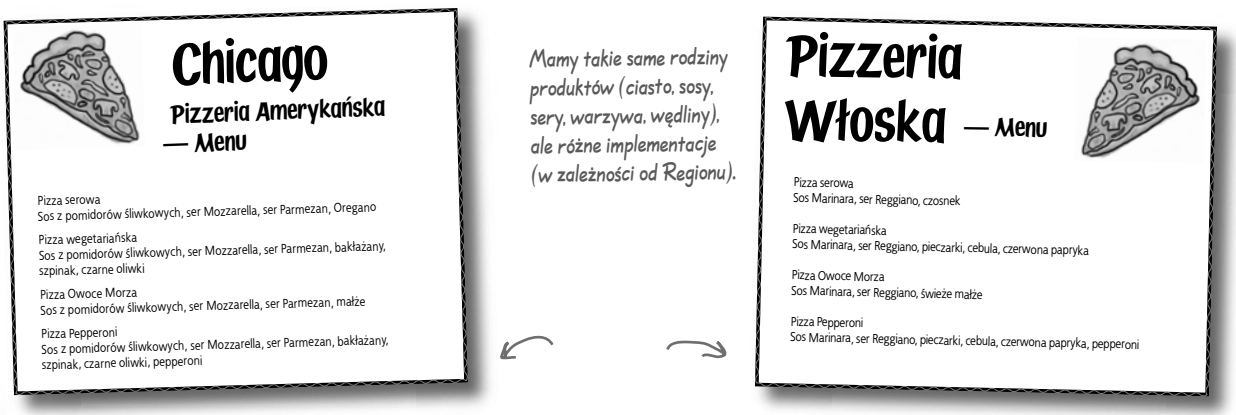
Kluczem do sukcesu sieci Twoich Pizzerii w Obiektowie (i nie tylko…) były zawsze świeże, znakomitej jakości składniki pizzy. Jednak podczas kontroli, jaką ostatnio przeprowadziłeś (w końcu bycie właścicielem sieci pizzerii do czegoś również zobowiązuje), odkryłeś, że co prawda dzięki odpowiedniemu projektowi struktury aplikacji wszyscy dzierżawcy przestrzegają Twoich *procedur*, ale niestety kilku z nich, aby obniżyć koszty produkcji i powiększyć zyski, zaczęło stosować tańsze składniki, o wiele gorszej jakości. Wiesz, że musisz na to jakoś zareagować, ponieważ na dłuższą metę takie postępowanie niekorzystnie odbije się na wizerunku Twojej sieci!



Zapewnienie jednakowej jakości wszystkich składników

A więc, w jaki sposób masz zamiar wymusić to, aby wszyscy dzierżawcy używali jednakowych, wysokiej jakości składników? Po prostu wybudujesz fabrykę, która będzie produkowała poszczególne składniki i dostarczała je do wszystkich dzierżawców!

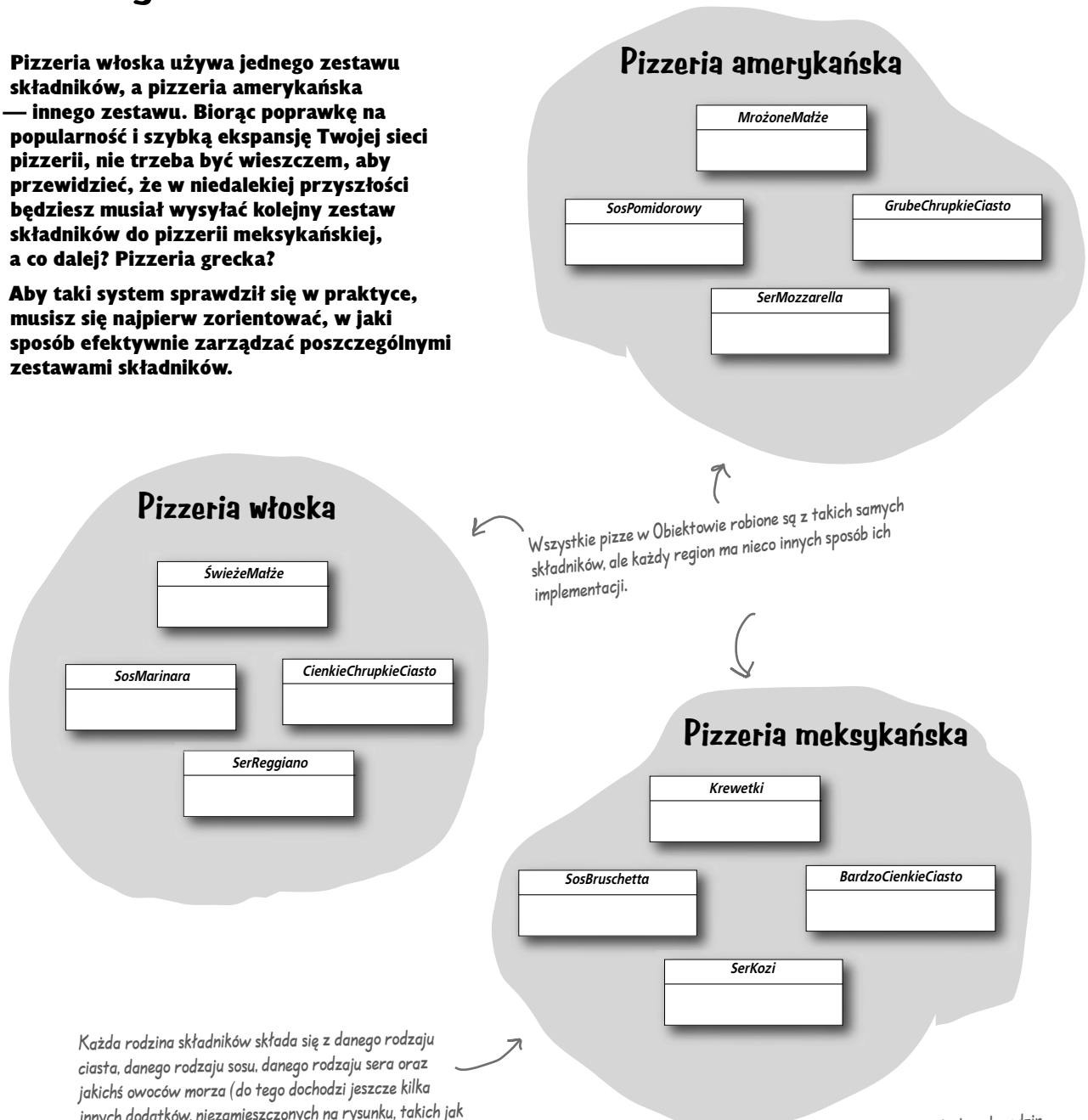
Z takim planem mamy tylko jeden problem: poszczególne oddziały pizzerii są ulokowane w różnych regionach i to, co jest „sosem pomidorowym" z punktu widzenia pizzerii włoskiej, niekoniecznie musi nim być w pizzerii amerykańskiej. Krótko mówiąc, jeden zestaw składników musi być dostarczany do pizzerii włoskiej, a zupełnie *inny* zestaw składników będzie zaopatrywał pizzerię amerykańską. Przyjrzyjmy się temu bliżej:



Rodziny składników...

Pizzeria włoska używa jednego zestawu składników, a pizzeria amerykańska — innego zestawu. Biorąc poprawkę na popularność i szybką ekspansję Twojej sieci pizzerii, nie trzeba być wieszczem, aby przewidzieć, że w niedalekiej przyszłości będziesz musiał wysyłać kolejny zestaw składników do pizzerii meksykańskiej, a co dalej? Pizzeria grecka?

Aby taki system sprawdził się w praktyce, musisz się najpierw zorientować, w jaki sposób efektywnie zarządzać poszczególnymi zestawami składników.



Budujemy fabryki składników pizzy

Mamy teraz zamiar zbudować fabrykę, która będzie wytwarzała składniki niezbędne dla naszych pizzerii; fabryka będzie odpowiedzialna za wytworzenie poszczególnych składników wchodzących w skład każdej z rodzin składników. Innymi słowy, nasza fabryka będzie musiała produkować ciasto, sosy, sery i tak dalej... Niebawem przekonasz się, w jaki sposób chcemy rozwiązać problem różnic regionalnych.

Rozpocznijmy od zdefiniowania interfejsu dla fabryki, która będzie wytwarzała wszystkie niezbędne składniki pizzy:

```
public interface FabrykaSkładnikówPizzy {

    public Ciasto utwórzCiasto();
    public Sos utwórzSos();
    public Ser utwórzSer();
    public Warzywa[] utwórzWarzywa();
    public Pepperoni utwórzPepperoni();
    public Małże utwórzMałże();

}
```

Dla każdego składnika definiujemy osobną metodę w naszym interfejsie.

Tutaj widzimy dużą liczbę nowych klas, po jednej na każdy składnik.

Jeżeli mielibyśmy jakieś wspólne „urządzenia" do zaimplementowania w każdej instancji fabryki, moglibyśmy zamiast interfejsu użyć klasy abstrakcyjnej...

Oto, co mamy teraz zamiar zrobić:

1. **Wybudujemy osobną fabrykę dla każdego regionu. Aby tego dokonać, musimy utworzyć odpowiednie klasy podrzędne, dziedziczące po klasie FabrykaSkładnikówPizzy, które będą implementowały odpowiednie metody tworzące poszczególne składniki.**
2. **Zaimplementujemy szereg klas opisujących poszczególne składniki wykorzystywane przez fabrykę, takich jak SerReggiano, CzerwonaPapryka, GrubeChrupkieCiasto. Klasy te mogą być używane w różnych regionach (o ile będą się do tego nadawały).**
3. **Wreszcie, będziemy musieli to wszystko razem opracować i połączyć naszą nową fabrykę składników z istniejącym do tej pory kodem Pizzerii.**

Budujemy włoską fabrykę składników pizzy

OK, poniżej znajdziesz przykładową implementację włoskiej fabryki składników pizzy. Ta fabryka specjalizuje się w produkcji sosu Marinara, sera Reggiano i pakowaniu świeżych małży...

Włoska fabryka składników pizzy implementuje interfejs wspólny dla wszystkich fabryk składników.

```
public class WłoskaFabrykaSkładnikówPizzy implements FabrykaSkładnikówPizzy {

    public Ciasto utwórzCiasto() {
        return new CienkieChrupkieCiasto();
    }

    public Sos utwórzSos() {
        return new SosMarinara();
    }

    public Ser utwórzSer() {
        return new SerReggiano();
    }

    public Warzywa[] utwórzWarzywa() {
        Warzywa warzywa[] = { new Czosnek(), new Cebula(), new Pieczarki(),
                              new CzerwonaPapryka() };
        return warzywa;
    }

    public Pepperoni utwórzPepperoni() {
        return new PlastryPepperoni();
    }

    public Małże utwórzMałże() {
        return new ŚwieżeMałże();
    }
}
```

Dla każdego składnika wchodzącego w skład rodziny składników tworzymy jego włoski odpowiednik.

W przypadku warzyw zwracamy całą tablicę Warzywa. W tym miejscu „na sztywno" zakodowaliśmy listę warzyw. Co prawda, moglibyśmy tutaj zastosować jakieś bardziej wyrafinowane rozwiązanie, ale i tak nie dodałoby to niczego nowego do procesu poznawania wzorca Fabryka, więc postanowiliśmy pójść po linii najmniejszego oporu i pozostawić takie proste rozwiązanie.

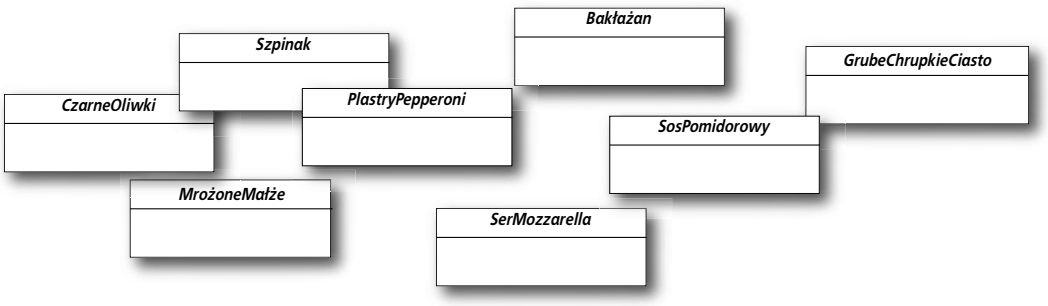
Najwyższej jakości plasterki pepperoni. Ten składnik jest używany zarówno do produkcji pizzy włoskiej, jak i amerykańskiej. Upewnij się, że robisz z niego użytek na następnej stronie, gdzie przyjdzie Ci samodzielnie zaimplementować amerykańską fabrykę składników pizzy.

Włoska pizza powinna być podawana ze świeżymi małżami — i takie też otrzymuje z fabryki. Jej amerykański odpowiednik zadowala się małżami mrożonymi.



Zaostrz ołówek

Spróbuj samodzielnie utworzyć kod dla klasy AmerykańskaFabryka SkładnikówPizzy. W implementacji możesz użyć odwołań do klas zamieszczonych na dole strony.



Aktualizacja kodu klas Pizza

Wszystkie nasze fabryki są już uruchomione i gotowe do produkcji najwyższej jakości składników; teraz musimy jeszcze tylko tak zmodyfikować nasze Pizze, aby mogły korzystać wyłącznie ze składników produkowanych w fabrykach. Rozpoczniemy od modyfikacji klasy abstrakcyjnej Pizza:

```
public abstract class Pizza {
    String nazwa;
    Ciasto ciasto;
    Sos sos;
    Warzywa warzywa[];
    Ser ser;
    Pepperoni pepperoni;
    Małże małże;

    abstract void przygotowanie();

    void pieczenie() {
        System.out.println("Pieczenie: 25 minut w temperaturze 180 stopni Celsjusza");
    }

    void krojenie() {
        System.out.println("Krojenie pizzy na 8 kawałków");
    }

    void pakowanie() {
        System.out.println("Pakowanie pizzy w oficjalne pudełko naszej sieci
        Pizzerii.");
    }

    void ustawNazwa(String nazwa) {
        this.nazwa = nazwa;
    }

    String pobierzNazwa() {
        return nazwa;
    }

    public String toString() {
    // kod wyświetlający informacje o pizzy
    }
}
```

Każda pizza posiada przypisany odpowiedni zestaw składników, które są wykorzystywane do jej przygotowania.

Teraz utworzyliśmy abstrakcyjną metodę przygotowanie(). Właśnie w tym miejscu będziemy gromadzili wszystkie składniki niezbędne do wykonania pizzy. Oczywiście, będą one pochodziły z odpowiedniej fabryki składników.

Reszta naszych metod pozostanie niezmieniona (z wyjątkiem metody przygotowanie()).

Aktualizacji kodu klas Pizza ciąg dalszy...

Mamy już gotową definicję klasy abstrakcyjnej Pizza, więc możemy przystąpić do tworzenia włoskiej i amerykańskiej odmiany Pizzy — od tej chwili będą one otrzymywały niezbędne składniki tylko i wyłącznie z fabryki. Czasy, kiedy poszczególni dzierżawcy mogli oszczędzać na produkcji, zaniżając jakość używanych składników, odchodzą zatem w niepamięć!

Kiedy utworzyliśmy kod metody fabrykującej, mieliśmy zdefiniowane dwie klasy: WłoskaSerowaPizza oraz AmerykańskaSerowaPizza. Jeżeli rzucisz okiem na definicje tych dwóch klas, z pewnością zauważysz, że jedyną różnicą pomiędzy nimi jest zestaw używanych składników (inny region, inne składniki). Sposób robienia podstawowej pizzy pozostaje jednak taki sam (ciasto + sos + ser). To samo dotyczy pozostałych gatunków pizzy: wegetariańskiej, owoce morze i tak dalej. Procedura przygotowania każdej z nich jest identyczna; różnią się tylko składnikami.

Wynika stąd jasno, że tak naprawdę nie musimy tworzyć dwóch osobnych klas dla każdego rodzaju pizzy; sprawą różnych składników regionalnych zajmie się dla nas fabryka.
Oto definicja naszej nowej pizzy serowej:

```
public class SerowaPizza extends Pizza {
    FabrykaSkładnikówPizzy fabrykaSkładników;

    public SerowaPizza(FabrykaSkładnikówPizzy fabrykaSkładników) {
        this.fabrykaSkładników = fabrykaSkładników;
    }

    void przygotowanie() {
        System.out.println("Przygotowywanie: " + name);
        ciasto = fabrykaSkładników.utwórzCiasto();
        sos = fabrykaSkładników.utwórzSos();
        ser = fabrykaSkładników.utwórzSer();
    }
}
```

Aby teraz utworzyć pizzę, niezbędna jest fabryka, która dostarczy nam odpowiednich składników. Każda klasa opisująca dany gatunek pizzy otrzymuje gotową fabrykę, przekazywaną do niej w konstruktorze klasy i przechowywaną w odpowiedniej zmiennej obiektowej.

W tym miejscu dokonuje się magia!

Metoda przygotowanie() realizuje kolejne etapy tworzenia pizzy serowej i za każdym razem, kiedy potrzebuje kolejnego składnika, prosi fabrykę o jego wykonanie i dostarczenie.



Bliskie spotkania z kodem aplikacji

Kod klasy Pizza wykorzystuje odpowiednią fabrykę do produkcji składników danej pizzy. To, jakie składniki są produkowane, zależy od tego, jakiej fabryki używamy do ich produkcji. Klasa Pizza zupełnie o to nie dba; ona wie tylko, jak należy robić pizzę. Dzięki usunięciu sprzężenia z lokalnymi odmianami składników klasa Pizza jest teraz bardzo niezależna i wskutek tego może być łatwo wykorzystywana w wielu innych sytuacjach, np. kiedy powstaną kolejne fabryki produkujące składniki dla, powiedzmy, hiszpańskiej i niemieckiej odmiany pizzy.

```
sos = fabrykaSkładników.utwórzSos();
```

Ustawiamy zmienną składową klasy Pizza tak, aby odwoływała się do konkretnego rodzaju sosu stosowanego do wskazanego gatunku pizzy.

To jest nasza fabryka składników. Klasa Pizza zupełnie nie dba o to, której fabryki używa, dopóty, dopóki jest to nadal fabryka składników.

Metoda utwórzSos() zwraca odmianę sosu używanego w danym regionie (w oddziale lokalnym). Przykładowo, jeżeli jest to pizzeria włoska, rezultatem działania tej metody będzie sos Marinara.

Zobaczmy jeszcze, jak wygląda kod klasy OwoceMorzaPizza:

```
public class OwoceMorzaPizza extends Pizza {
    FabrykaSkładnikówPizzy fabrykaSkładników;

    public OwoceMorzaPizza(FabrykaSkładnikówPizzy fabrykaSkładników) {
        this.fabrykaSkładników = fabrykaSkładników;
    }

    void przygotowanie() {
        System.out.println("Przygotowywanie: " + nazwa);
        ciasto = fabrykaSkładników.utwórzCiasto();
        sos = fabrykaSkładników.utwórzSos();
        ser = fabrykaSkładników.utwórzSer();
        małże = fabrykaSkładników.utwórzMałże();
    }
}
```

Klasa OwoceMorzaPizza również wykorzystuje fabrykę składników.

Aby zrobić pizzę Owoce Morza, metoda przygotowanie() gromadzi tutaj niezbędne składniki prosto ze swojej lokalnej fabryki.

Jeżeli to włoska fabryka składników, małże będą świeże; jeżeli amerykańska, małże będą mrożone.

Odwiedzamy lokalne oddziały naszej sieci pizzerii

Prawie gotowe; musimy jeszcze tylko wybrać się na szybką wycieczkę do poszczególnych oddziałów naszej sieci pizzerii, aby upewnić się, że wszyscy dzierżawcy używają właściwej Pizzy; dodatkowo musimy im również udostępnić odwołania do odpowiednich lokalnych fabryk składników:

```
public class WłoskaPizzeria extends Pizzeria {

    protected Pizza utwórzPizza(String item) {
        Pizza pizza = null;
        FabrykaSkładnikówPizzy fabrykaSkładników =
            new WłoskaFabrykaSkładnikówPizzy();

        if (item.equals("serowa")) {

            pizza = new SerowaPizza(fabrykaSkładników);
            pizza.ustawNazwa("Włoska Pizza Serowa");

        } else if (item.equals("wegetariańska")) {

            pizza = new WegetariańskaPizza(fabrykaSkładników);
            pizza.ustawNazwa("Włoska Pizza Wegetariańska");

        } else if (item.equals("owoce morza")) {

            pizza = new OwoceMorzaPizza(fabrykaSkładników);
            pizza.ustawNazwa("Włoska Pizza Owoce Morza");

        } else if (item.equals("pepperoni")) {
            pizza = new PepperoniPizza(fabrykaSkładników);
            pizza.ustawNazwa("Włoska Pizza Pepperoni");

        }
        return pizza;
    }
}
```

Pizzeria włoska współpracuje z włoską fabryką składników pizzy. Takie powiązanie będzie wykorzystywane do produkcji składników niezbędnych do wytwarzania wszystkich rodzajów włoskiej pizzy.

Teraz każda pizza otrzymuje odwołanie do fabryki, która powinna zostać użyta do produkcji jej składników.

Powróć na chwilę do informacji zamieszczonych na poprzedniej stronie i upewnij się, że wiesz, w jaki sposób dana pizza i dana fabryka składników współpracują ze sobą!

Dla każdego rodzaju Pizzy tworzymy nowy obiekt klasy Pizza i udostępniamy mu fabrykę, która dostarcza mu odpowiednie składniki.



WYTĘŻ UMYSŁ

Porównaj tę wersję metody utwórzPizza() z jej implementacją, z którą spotkałeś się przy okazji omawiania metody fabrykującej we wcześniejszej części niniejszego rozdziału.

Czego udało nam się dokonać?

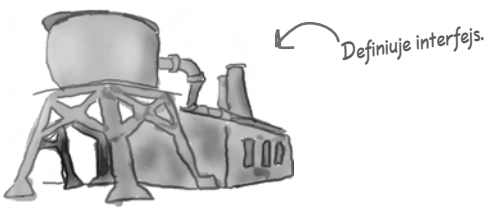
To była całkiem przyzwoita seria modyfikacji kodu; a zatem — czego tak naprawdę udało nam się dokonać?

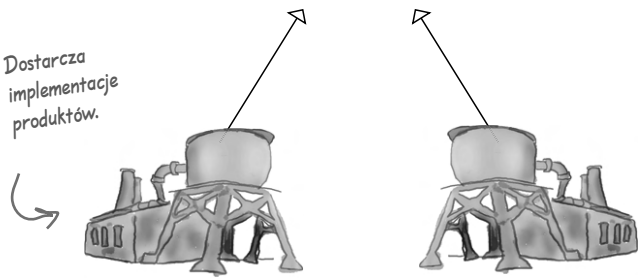
Zapewniliśmy skuteczne rozwiązanie, które umożliwia tworzenie rodziny składników dla poszczególnych gatunków pizzy; osiągnęliśmy to poprzez wprowadzenie nowego typu fabryki, zwanej fabryką abstrakcyjną (ang. Abstract Factory).

Fabryka abstrakcyjna daje nam interfejs umożliwiający tworzenie całych rodzin produktów. Poprzez napisanie kodu wykorzystującego interfejs możemy odseparować kod naszego programu (klienta) od fabryki tworzącej odpowiednie produkty. Takie rozwiązanie pozwala na implementację wielu różnych fabryk tworzących produkty przeznaczone dla różnych podmiotów — takich jak różne regiony, różne systemy operacyjne i różni inni odbiorcy.

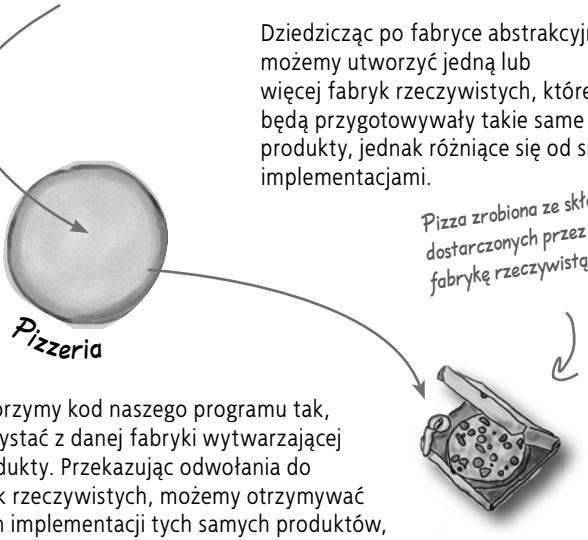
Ponieważ kod naszego programu jest oddzielony od fabryki produktów, jeżeli chcemy uzyskać różne produkty, możemy bez problemu podmieniać odpowiednie fabryki (np. aby otrzymać sos Marinara zamiast zwykłego sosu pomidorowego).

Fabryka abstrakcyjna zapewnia interfejs dla całej rodziny produktów. Czym jest rodzina produktów? W naszym przypadku rodzinę produktów stanowią wszystkie składniki niezbędne do utworzenia pizzy: ciasto, sos, ser, wędliny i warzywa.





Dziedzicząc po fabryce abstrakcyjnej, możemy utworzyć jedną lub więcej fabryk rzeczywistych, które będą przygotowywały takie same produkty, jednak różniące się od siebie implementacjami.



Następnie tworzymy kod naszego programu tak, aby mógł korzystać z danej fabryki wytwarzającej określone produkty. Przekazując odwołania do różnych fabryk rzeczywistych, możemy otrzymywać szereg różnych implementacji tych samych produktów, ale mimo to kod klienta pozostaje ciągle taki sam.

Jeszcze więcej pizzy dla Jacka i Eryka...


Za kulisami

Jacek i Eryk wciąż nie mają dosyć naszej wspaniałej pizzy! Jednak nadal pozostają w błogiej nieświadomości, że ich zamówienia są obecnie realizowane przy pomocy nowej fabryki składników. Więc teraz, kiedy składają zamówienie...



Pierwsza część procesu realizacji zamówienia w ogóle się nie zmieniła. Prześledźmy zatem jeszcze raz zamówienie Eryka:

1 Najpierw musimy utworzyć obiekt klasy WłoskaPizzeria:

```
Pizzeria włoskaPizzeria = new WłoskaPizzeria();
```

Tworzy nową instancję obiektu klasy WłoskaPizzeria.



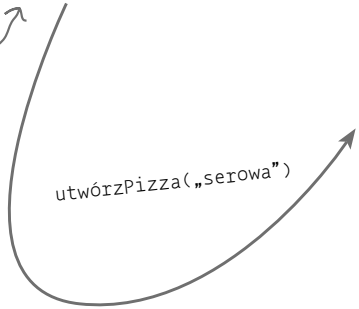
2 Mamy już odpowiedni oddział pizzerii, więc możemy złożyć zamówienie

```
włoskaPizzeria.zamówPizza("serowa");
```

Wywoływana metoda zamówPizza() jest metodą instancji obiektu włoskaPizzeria.

3 Następnie metoda zamówPizza() wywołuje metodę utwórzPizza():

```
Pizza pizza = utwórzPizza("serowa");
```



Od tego momentu przebieg procesu zmienia się, ponieważ korzystamy z fabryki składników.

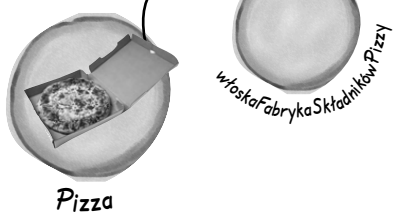


4 **Moment wywoływania metody utwórzPizza() jest właśnie tym momentem, w którym do procesu włącza się nasza fabryka składników:**

Egzemplarz obiektu reprezentującego wybraną fabrykę składników zostaje utworzony w klasie Pizzeria, a następnie przekazany do konstruktorów poszczególnych gatunków pizzy.

```
Pizza pizza = new SerowaPizza(włoskaFabrykaSkładnikówPizzy);
```

Tworzy obiekt klasy Pizza, który posiada przypisaną włoską fabrykę składników.



5 **Następnym etapem jest przygotowanie pizzy. Po wywołaniu metody przygotowanie() fabryka jest proszona o tworzenie i dostarczanie kolejnych składników:**

```
void przygotowanie() {
    ciasto = fabrykaSkładników.utwórzCiasto();
    sos = fabrykaSkładników.utwórzSos();
    ser = fabrykaSkładników.utwórzSer();
}
```

Cienkie ciasto

Sos Marinara

Ser Reggiano

przygotowanie()

Do wykonania pizzy dla Eryka używana jest fabryka włoska, stąd otrzymujemy składniki charakterystyczne dla tej odmiany pizzy.

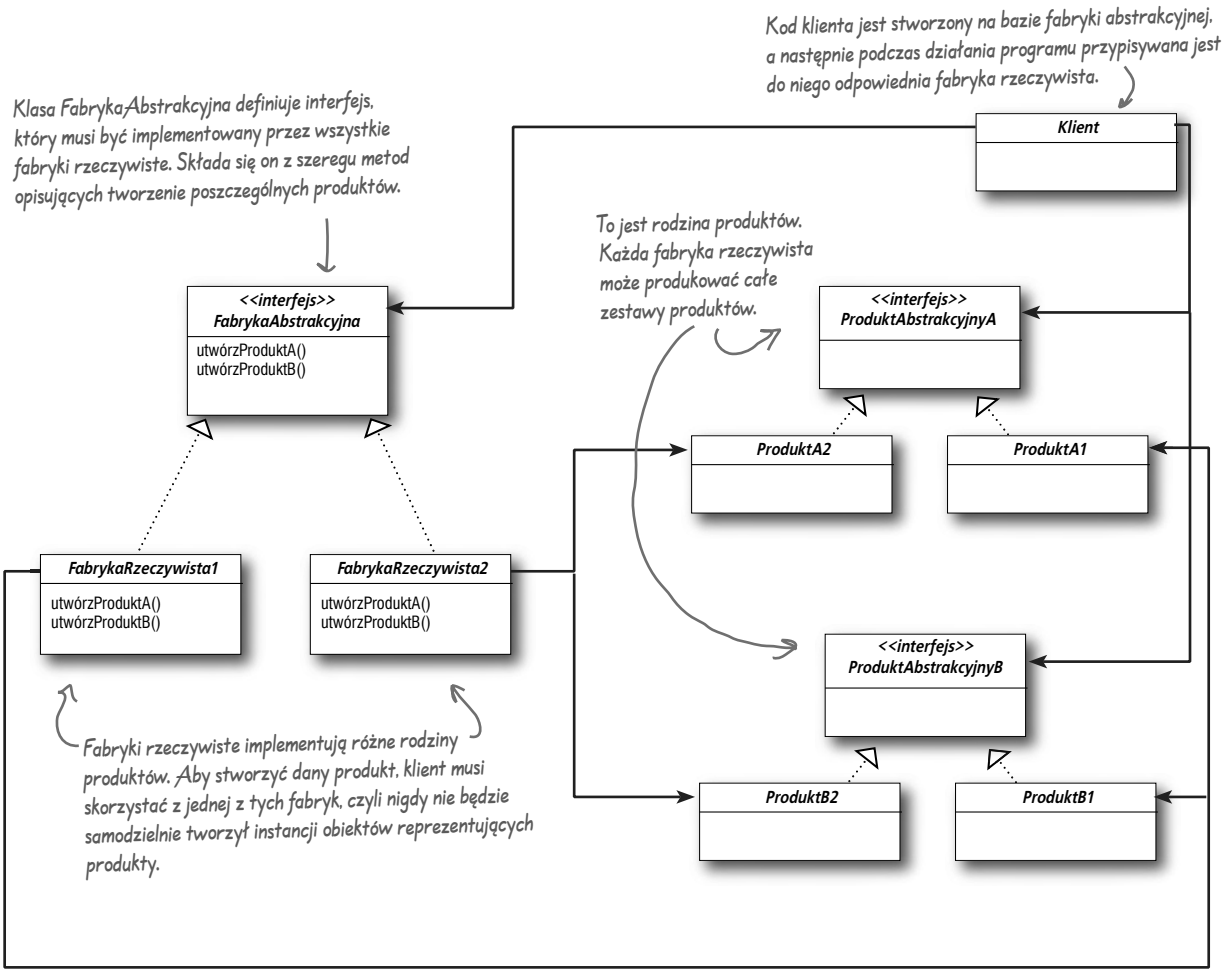
6 **Na zakończenie otrzymujemy gotową pizzę, a resztę zadań przejmuje metoda zamówPizza(), która zajmuje się pieczeniem, krojeniem i pakowaniem pizzy.**

Definicja wzorca Fabryka Abstrakcyjna

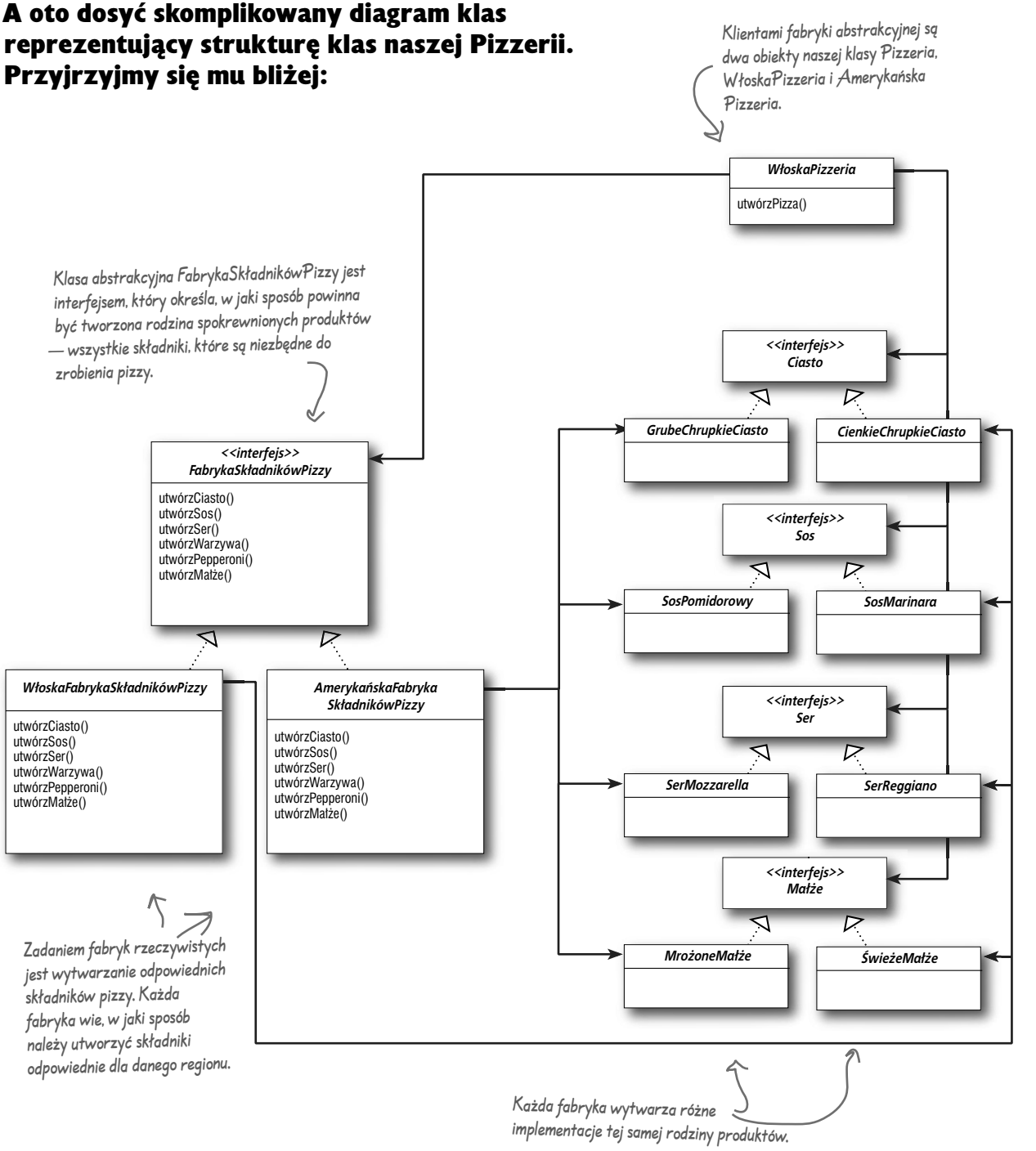
Nadszedł zatem czas, aby do naszej oswojonej grupy wzorców projektowych dołączyć kolejnego członka, który pozwala na tworzenie rodzin produktów. Zobaczmy zatem, jak wygląda oficjalna definicja tego wzorca:

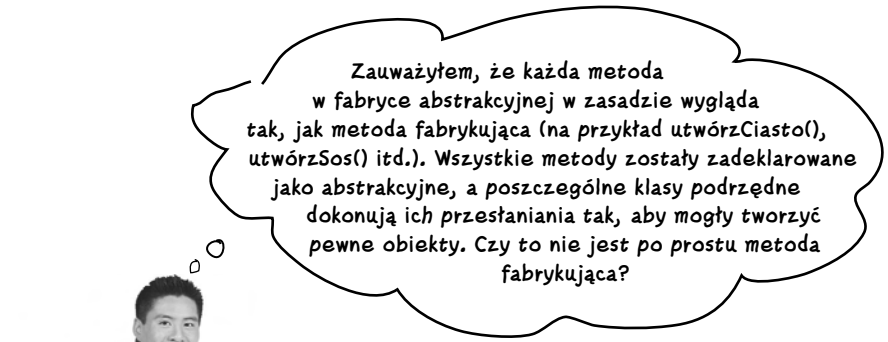
> **Wzorzec Fabryka Abstrakcyjna** dostarcza interfejs do tworzenia całych rodzin spokrewnionych lub zależnych od siebie obiektów bez konieczności określania ich klas rzeczywistych.

Widzieliśmy już, że fabryka abstrakcyjna pozwala klientowi na używanie interfejsu abstrakcyjnego do tworzenia zestawu spokrewnionych produktów bez znajomości szczegółów dotyczących tworzonych produktów. Dzięki takiemu rozwiązaniu kod klienta jest całkowicie pozbawiony sprzężenia (oddzielony) od jakichkolwiek szczegółów implementacji produktów rzeczywistych. Rzućmy zatem okiem na diagram klas i zobaczmy raz jeszcze, jak to wszystko funkcjonuje:



A oto dosyć skomplikowany diagram klas reprezentujący strukturę klas naszej Pizzerii. Przyjrzyjmy się mu bliżej:





Czy przypadkiem z wnętrza fabryki abstrakcyjnej nie wygląda dobrze nam znana metoda fabrykująca?

Znakomite pytanie! Tak, bardzo często zdarza się, że metody fabryki abstrakcyjnej zostają zaimplementowane w postaci metod fabrykujących. To jednak ma sens, prawda? Zadaniem fabryki abstrakcyjnej jest zdefiniowanie interfejsu pozwalającego na tworzenie określonego zestawu produktów. Poszczególne metody tego interfejsu są odpowiedzialne za tworzenie produktów rzeczywistych. Aby wesprzeć ten proces, tworzymy kolejne podklasy, które dziedziczą po fabryce abstrakcyjnej — stąd zastosowanie metod fabrykujących wydaje się niejako naturalnym sposobem implementacji metod odpowiedzialnych za tworzenie poszczególnych produktów w fabryce abstrakcyjnej.



Wzorce bez tajemnic

Wywiad tygodnia: Metoda Fabrykująca i Fabryka Abstrakcyjna, czyli traktat o pomocy wzajemnej

Rusz głową!: O rany! Wywiad z dwoma wzorcami naraz! To pierwszy taki przypadek w naszej karierze.

Metoda Fabrykująca: Taaak… jeśli mam być szczera, nie jestem do końca przekonana, czy chcę być łączona z fabryką abstrakcyjną — sam wiesz, jak to jest. Fakt, że obie jesteśmy wzorcami typu fabryka, nie znaczy wcale, że nie zasłużyłyśmy na osobne wywiady.

Rusz głową!: Oj, nie irytuj się tak… Zapraszając Was razem, chcieliśmy po prostu ułatwić naszym Czytelnikom zorientowanie się w sytuacji i jednoznacznie wyjaśnić, kto jest kim. Jesteście do siebie bardzo podobne i słyszeliśmy, że z tego powodu wielu ludziom często zdarza się Was pomylić.

Fabryka Abstrakcyjna: To prawda, zdarzały mi się sytuacje, kiedy byłam mylona z Metodą Fabrykującą, a z tego, co wiem, to i ona miała takie przeżycia. Obie wykonujemy kawał dobrej roboty, oddzielając kod klienta od szczegółów implementacji; ale każda z nas robi to w inny sposób. I naprawdę doskonale rozumiem, dlaczego ludzie mylą nas ze sobą…

Metoda Fabrykująca: No cóż, mimo wszystko takie podejście nadal mnie irytuje. Przecież ja wykorzystuję klasy, podczas gdy Ty wykorzystujesz obiekty; to przecież coś zupełnie innego!

Rusz głową!: Czy możesz to nieco szerzej objaśnić, Metodo Fabrykująca?

Metoda Fabrykująca: Jasne! Spójrz, zarówno Fabryka Abstrakcyjna, jak i ja możemy tworzyć obiekty — to w końcu nasze zadanie. Ale ja do tego celu wykorzystuję mechanizm dziedziczenia…

Fabryka Abstrakcyjna: … a ja dokonuję tego samego poprzez kompozycję obiektów.

Metoda Fabrykująca: Zgadza się. Czyli, krótko mówiąc, aby utworzyć obiekt przy użyciu Metody Fabrykującej, musisz rozszerzyć klasę bazową i przesłonić metodę fabrykującą.

Rusz głową!: A ta metoda fabrykująca czym się zajmuje?

Metoda Fabrykująca: Tworzeniem obiektów, oczywiście! To znaczy, miałam na myśli to, że tak naprawdę tworzeniem obiektów zajmują się odpowiednie klasy podrzędne. W ten sposób klient musi tylko wiedzieć, jakich typów abstrakcyjnych używa, a typami rzeczywistymi zajmują się odpowiednie podklasy. Innymi słowy, dzięki takiemu rozwiązaniu usuwam sprzężenie pomiędzy klientem a typami rzeczywistymi.

Fabryka Abstrakcyjna: I ja również, tyle że robię to w inny sposób.

Rusz głową!: Powiedz nam o tym więcej… wspominałaś coś o kompozycji obiektów?

Fabryka Abstrakcyjna: Udostępniam typ abstrakcyjny umożliwiający tworzenie rodziny produktów. Podklasy tego typu określają sposób przygotowania poszczególnych produktów. Aby skorzystać z fabryki, musisz utworzyć obiekt, który ją reprezentuje, a następnie przekazać go do odpowiedniego kodu, który wykorzystuje elementy typu abstrakcyjnego. Dzięki temu, podobnie jak w przypadku Metody Fabrykującej, utrzymuję moich klientów z dala od tworzenia produktów rzeczywistych, z których korzystają.

Rusz głową!: Oooo, rozumiem, czyli kolejną, dodatkową zaletą jest to, że grupujesz zestawy spokrewnionych ze sobą produktów.

Fabryka Abstrakcyjna: Zgadza się.

Rusz głową!: A co się dzieje, jeżeli zachodzi potrzeba rozbudowania zestawu wytwarzanych produktów — powiedzmy, że dodajemy do niego nowy produkt? Czy to przypadkiem nie wymaga zmiany Twojego interfejsu?

Fabryka Abstrakcyjna: To prawda; jeżeli dodajemy nowe produkty, mój interfejs musi zostać odpowiednio rozbudowany. Wiem, że ludzie tego nie lubią…

Metoda Fabrykująca: <parsknięcie śmiechem>

Fabryka Abstrakcyjna: No i czemu się ze mnie naśmiewasz, Metodo Fabrykująca?

Metoda Fabrykująca: Oj, daj spokój, przecież to nie są przelewki! Konieczność zmiany Twojego interfejsu oznacza konieczność dokonania takiej samej zmiany interfejsu w każdej klasie podrzędnej! Wygląda mi to na solidny kawał roboty do wykonania…

Fabryka Abstrakcyjna: No tak, ale ja przecież potrzebuję rozbudowanego interfejsu, ponieważ jestem wykorzystywana do tworzenia całych rodzin produktów. Ty produkujesz tylko jeden produkt, więc nie potrzebujesz dużych interfejsów — prawdę mówiąc, wystarczy Ci tylko jedna metoda.

Rusz głową!: Fabryko Abstrakcyjna, ale przecież słyszałem, że często wykorzystujesz metody fabrykujące do implementacji swoich fabryk rzeczywistych?

Fabryka Abstrakcyjna: No tak, muszę szczerze przyznać, że moje fabryki rzeczywiste często implementują metody fabrykujące do tworzenia swoich produktów. Ale w moim przypadku są one wykorzystywane tylko i wyłącznie do wytwarzania produktów…

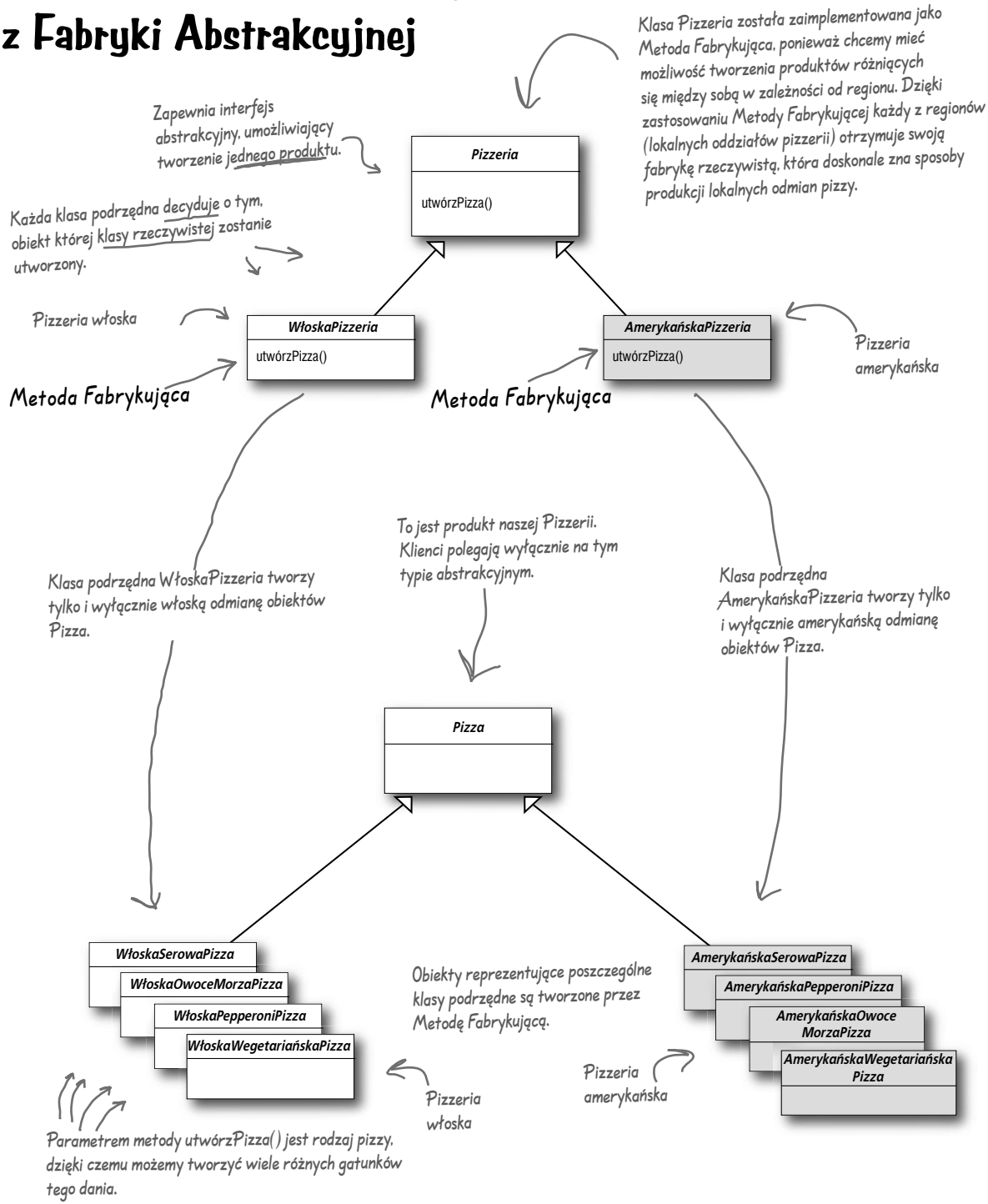
Metoda Fabrykująca: … podczas gdy ja zazwyczaj implementuję odpowiedni kod w konstruktorze klasy abstrakcyjnej, który wykorzystuje typy rzeczywiste tworzone przez klasy podrzędne.

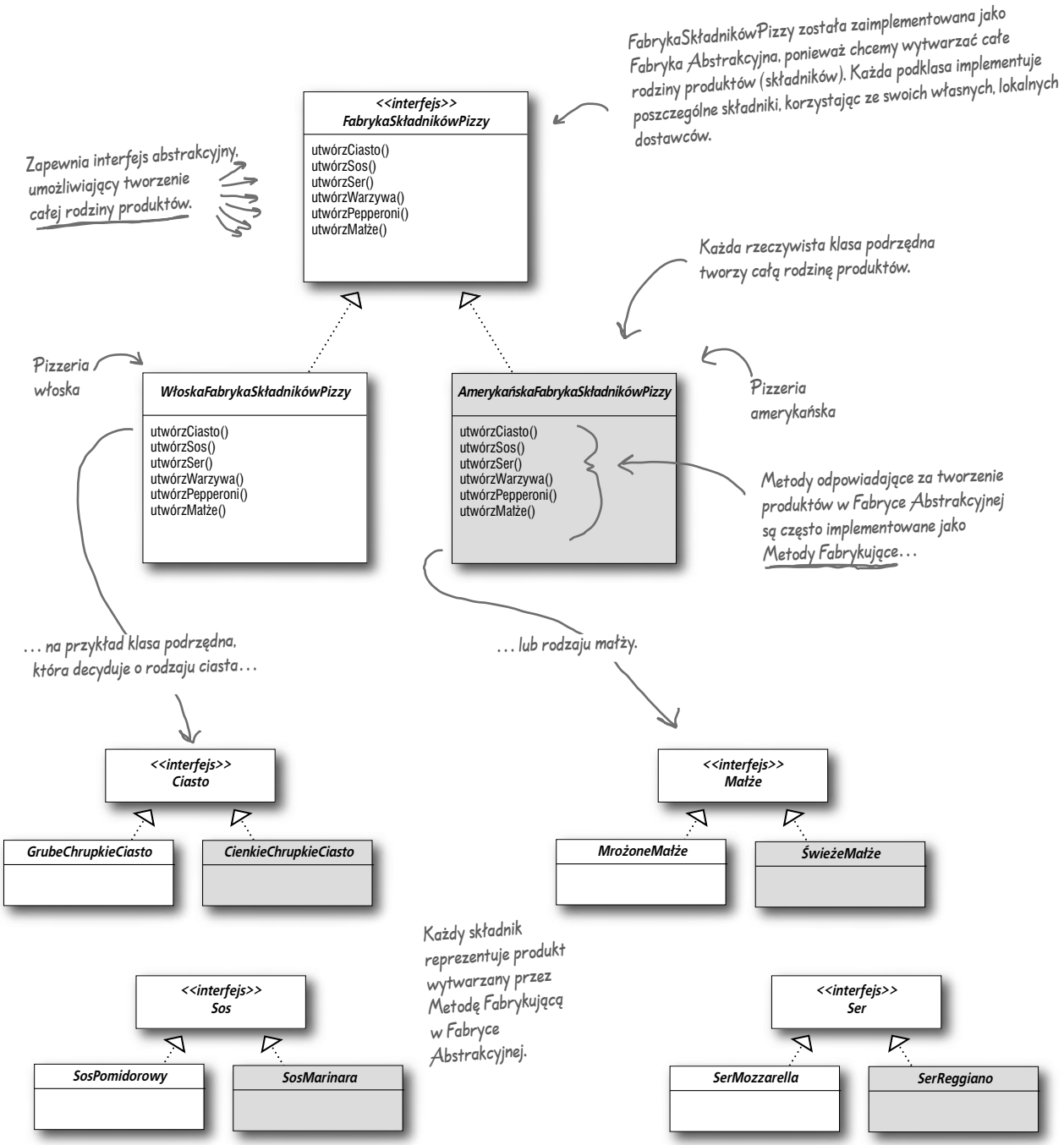
Rusz głową!: Wygląda na to, że obie jesteście dobre w tym, czym się zajmujecie. Jestem pewny, że ludzie lubią sytuacje, w których mają jakiś wybór; co więcej, jak by na to nie patrzeć, fabryki są tak bardzo użytecznym rozwiązaniem, że z pewnością będą chcieli z nich korzystać w wielu różnych sytuacjach. Obie powodujecie hermetyzację procesu tworzenia obiektów, umożliwiając dzięki temu zastosowanie luźnych powiązań kodu aplikacji, usunięcie sprzężeń i zmniejszenie zależności od konkretnych implementacji — i to jest naprawdę wspaniałe, niezależnie od tego, czy korzystamy z metody fabrykującej, czy też z fabryki abstrakcyjnej. Czy mogę prosić każdą z Was o małe podsumowanie na koniec naszego wywiadu?

Fabryka Abstrakcyjna: Dziękuję. Pamiętajcie o mnie, to ja, Fabryka Abstrakcyjna. Możecie mnie wykorzystywać zawsze wtedy, kiedy zaistnieje potrzeba wytwarzania rodziny produktów i chcecie mieć pewność, że poszczególni klienci wytwarzają spójne rodziny produktów.

Metoda Fabrykująca: Jestem Metodą Fabrykującą; wykorzystujcie mnie do oddzielania kodu klienta od klas rzeczywistych, których obiekty musicie tworzyć, lub gdy z góry nie wiecie, z jakich klas rzeczywistych przyjdzie Wam korzystać. Aby zastosować moje rozwiązania, wystarczy utworzyć klasy podrzędne i zaimplementować moją metodę fabrykującą!

Porównanie Metody Fabrykującej oraz Fabryki Abstrakcyjnej





Klasy podrzędne reprezentujące poszczególne produkty tworzą równoległe zestawy rodzin produktów. Tutaj mamy rodzinę produktów włoskich oraz rodzinę produktów amerykańskich.



Twoja skrzynka narzędziowa

Dzięki niniejszemu rozdziałowi dołożyliśmy dwa nowe narzędzia do Twojej skrzynki: Metodę Fabrykującą oraz Fabrykę Abstrakcyjną. Oba wzorce zapewniają hermetyzację procesu tworzenia obiektów i pozwalają na oddzielenie kodu klienta od implementacji typów rzeczywistych.



CELNE SPOSTRZEŻENIA

- Wszystkie fabryki powodują hermetyzację procesu tworzenia obiektów.
- Prosta Fabryka, pomimo iż nie jest pełnoprawnym wzorcem projektowym, jest prostym, ale skutecznym sposobem na oddzielenie klienta od implementacji klas rzeczywistych.
- Metoda Fabrykująca wykorzystuje mechanizm dziedziczenia: tworzenie obiektów jest delegowane do klas podrzędnych implementujących metodę fabrykującą.
- Fabryka Abstrakcyjna wykorzystuje kompozycję obiektów: tworzenie obiektów zostało zaimplementowane w metodach dostępnych poprzez interfejs fabryki.
- Wszystkie wzorce typu fabryka promują tworzenie luźnych powiązań poprzez redukcję zależności kodu aplikacji od implementacji klas rzeczywistych.
- Ideą Metody Fabrykującej jest zezwolenie superklasie na przekazywanie odpowiedzialności za tworzenie obiektów do klas podrzędnych.
- Ideą Fabryki Abstrakcyjnej jest tworzenie rodzin spokrewnionych obiektów bez konieczności polegania na ich klasach rzeczywistych.
- Reguła odwracania zależności (reguła DIP — ang. *Dependency Inversion Principle*) nakazuje unikanie uzależniania kodu od implementacji klas rzeczywistych i promuje programowanie pod kątem klas oraz interfejsów abstrakcyjnych.
- Fabryki są potężnymi technikami programistycznymi pozwalającymi na tworzenie kodu, który będzie uzależniony od abstrakcji, a nie od implementacji klas rzeczywistych.



Rozwiązania ćwiczeń



Zaostrz ołówek

Kod klasy WłoskaPizzeria jest już gotowy; pozostały do zrobienia jeszcze tylko dwie klasy i będziemy mogli uruchomić pierwsze oddziały naszej sieci pizzerii! Poniżej napisz odpowiedni kod dla klas AmerykańskaPizzeria oraz MeksykańskaPizzeria:

Obie definicje klas są niemal identyczne, jak definicja klasy WłoskaPizzeria… Różnica polega na tym, że tworzą inne rodzaje pizzy.

```
public class AmerykańskaPizzeria extends Pizzeria {
    protected Pizza utwórzPizza(String item) {
        if (item.equals("serowa")) {
            return new AmerykańskaSerowaPizza();
        } else if (item.equals("wegetariańska")) {
            pizza = new AmerykańskaWegetariańskaPizza();
        } else if (item.equals("owoce morza")) {
            pizza = new AmerykańskaOwoceMorzaPizza();
        } else if (item.equals("pepperoni")) {
            pizza = new AmerykańskaPepperoniPizza();
        } else return null;
    }
}
```

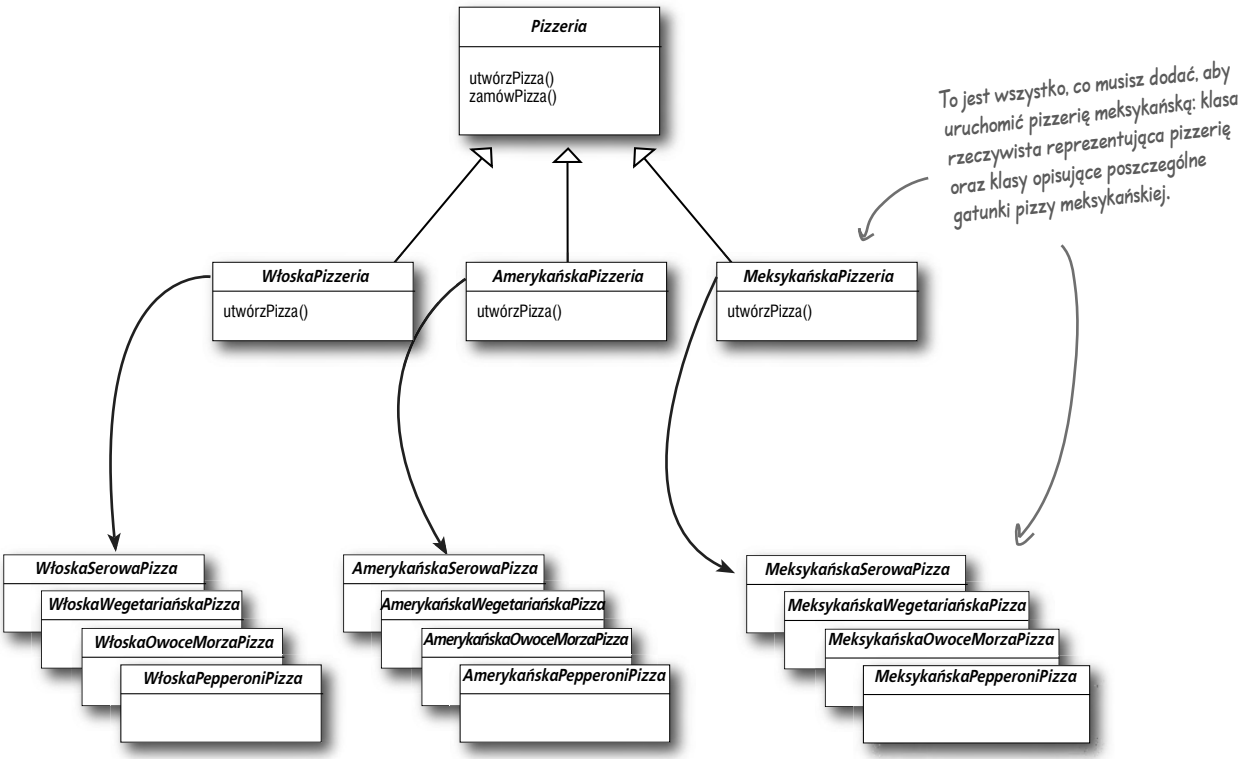
Musimy się upewnić, że pizzeria amerykańska będzie tworzyła pizzę w stylu amerykańskim…

```
public class MeksykańskaPizzeria extends Pizzeria {
    protected Pizza utwórzPizza(String item) {
        if (item.equals("serowa")) {
            return new MeksykańskaSerowaPizza();
        } else if (item.equals("wegetariańska")) {
            pizza = new MeksykańskaWegetariańskaPizza();
        } else if (item.equals("owoce morza")) {
            pizza = new MeksykańskaOwoceMorzaPizza();
        } else if (item.equals("pepperoni")) {
            pizza = new MeksykańskaPepperoniPizza();
        } else return null;
    }
}
```

… a pizzeria meksykańska będzie serwowała pizzę odmiany meksykańskiej.

Rozwiązanie łamigłówki projektowej

Będziemy potrzebować kolejnej odmiany dla szalonych koneserów pizzy meksykańskiej (oczywiście, „szalonych” w dobrym tego słowa znaczeniu). Narysuj zatem kolejną, równoległą hierarchię klas, która będzie niezbędna, aby rozszerzyć działalność Twojej sieci pizzerii na obszar Meksyku.



OK, a teraz wypisz poniżej pięć *najbardziej dziwacznych* dodatków do pizzy, jakie tylko jesteś sobie w stanie wyobrazić. Po zakończeniu będziesz naprawdę gotowy, aby uruchomić swój biznes w oddziale meksykańskim.

Oto nasze sugestie:

- Tłuczone ziemniaki zapiekane z czosnkiem
- Sos barbecue
- Młode, gotowane karczochy
- Drażetki M&M
- Orzeszki ziemne

Pizzeria mocno uzależniona



Zaostrz ołówek

Załóżmy, że nigdy w życiu nie słyszałeś o fabrykach obiektów i metodach fabrykujących. Poniżej zamieściliśmy wersję klasy Pizzeria, która nie korzysta z fabryki; postaraj się policzyć, od ilu rzeczywistych obiektów klasy Pizza ta klasa jest uzależniona. Jeżeli teraz dodamy do oferty pizzę meksykańską, to o ile zmieni się ilość klas?

```
public class PizzeriaMocnoUzależniona{

    public Pizza utwórzPizza(String style, String type) {
        Pizza pizza = null;
        if (style.equals("Włoska")) {
            if (type.equals("serowa")) {
                pizza = new WłoskaSerowaPizza();
            } else if (type.equals("wegetariańska")) {
                pizza = new WłoskaWegetariańskaPizza();
            } else if (type.equals("owoce morza")) {
                pizza = new WłoskaOwoceMorzaPizza();
            } else if (type.equals("pepperoni")) {
                pizza = new WłoskaPepperoniPizza();
            }
        } else if (style.equals("Amerykańska")) {
            if (type.equals("serowa")) {
                pizza = new AmerykańskaSerowaPizza();
            } else if (type.equals("wegetariańska")) {
                pizza = new AmerykańskaWegetariańskaPizza();
            } else if (type.equals("owoce morza")) {
                pizza = new AmerykańskaOwoceMorzaPizza();
            } else if (type.equals("pepperoni")) {
                pizza = new AmerykańskaPepperoniPizza();
            }
        } else {
            System.out.println(”Błąd: zamówiono niepoprawny rodzaj pizzy.”);
            return null;
        }
        pizza.przygotowanie();
        pizza.pieczenie();
        pizza.krojenie();
        pizza.pakowanie();
        return pizza;
    }
}
```

Obsługuje wszystkie gatunki pizzy przyrządzane na sposób włoski.

Obsługuje wszystkie gatunki pizzy przyrządzane na sposób amerykański.

odpowiedzi możesz wpisać tutaj:

8 ilość

12 ilość po wprowadzeniu pizzy meksykańskiej



Zaostrz ołówek

Spróbuj samodzielnie utworzyć kod dla klasy AmerykańskaFabrykaSkładnikówPizzy. W implementacji możesz użyć odwołań do klas zamieszczonych na dole strony.

```
public class AmerykańskaFabrykaSkładnikówPizzy
    implements FabrykaSkładnikówPizzy
{

    public Ciasto utwórzCiasto() {
        return new GrubeChrupkieCiasto();
    }

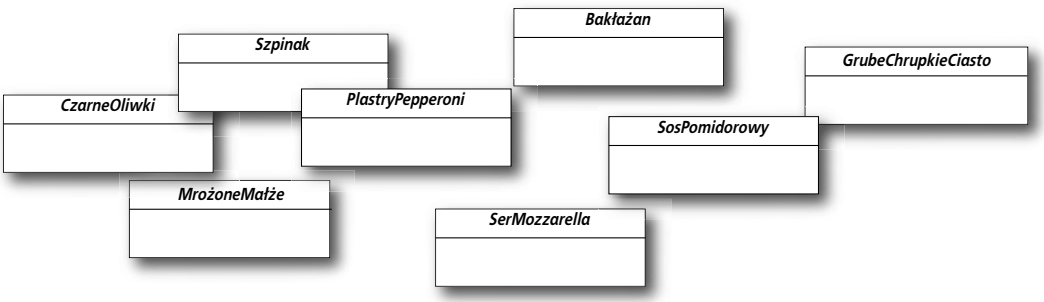
    public Sos utwórzSos() {
        return new SosPomidorowy();
    }

    public Ser utwórzSer() {
        return new SerMozzarella();
    }

    public Warzywa[] utwórzWarzywa() {
        Warzywa warzywa [] = { new CzarneOliwki(),
                               new Szpinak(),
                               new Bakłażan() };
        return warzywa;
    }

    public Pepperoni utwórzPepperoni() {
        return new PlastryPepperoni();
    }

    public Małże utwórzMałże() {
        return new MrożoneMałże();
    }
}
```



Obiekty jedyne w swoim rodzaju



Kolejnym przystankiem w naszej podróży jest wzorzec Singleton, czyli nasza przepustka do tworzenia jedynych w swoim rodzaju obiektów, posiadających tylko jedną instancję. Być może ucieszysz się na wieść o tym, że Singleton jest najprostszym z istniejących wzorców projektowych (przynajmniej pod względem kategorii stopnia złożoności jego diagramu klas); jak by na to nie patrzeć, jego diagram składa się tylko z jednej klasy! Ale nie wpadaj w euforię; niezależnie od prostoty diagramu klas tego wzorca na drodze prowadzącej do jego implementacji napotkamy całkiem sporo wybojów i dziur, więc lepiej zapnij mocno pasy!

Jeden i tylko jeden



Projektant: Jaki jest zakres zastosowań takiego wzorca?

Guru: W wielu różnych przypadkach będziemy potrzebowali tylko jednej instancji danego obiektu. Jako przykład można podać pule wątków, bufory podręczne, okna dialogowe, obiekty obsługujące właściwości i ustawienia rejestrów, obiekty wykorzystywane podczas procesu logowania czy też obiekty pełniące rolę sterowników dla takich urządzeń, jak drukarki i karty graficzne. Jeżeli zdarzyłaby się sytuacja, że dla któregoś z wymienionych typów obiektów stworzylibyśmy więcej niż jedną instancję obiektu, popadlibyśmy w poważne tarapaty, takie jak np. nieprawidłowe zachowanie programu, nadmierne zużycie zasobów systemowych czy niespójne wyniki działania.

Projektant: No dobrze, być może faktycznie są pewne klasy, których obiekty powinny być tworzone tylko i wyłącznie w jednym egzemplarzu, ale czy jest to wystarczający powód, aby poświęcać temu cały rozdział? Czy nie mógłbym tego dokonać zgodnie z przyjętą konwencją, czy też na przykład poprzez zmienne globalne? Sam wiesz, jak to jest — tak jak w języku Java, mógłbym tego dokonać przy użyciu zmiennej statycznej.

Guru: Z wielu względów można przyjąć, że wzorzec Singleton **jest konwencją**, która zapewnia, że zostanie stworzona tylko jedna instancja obiektu danej, ściśle określonej klasy. Jeżeli znasz jakieś lepsze rozwiązanie, z pewnością świat będzie chciał o nim usłyszeć; ale pamiętaj, Singleton, podobnie jak inne wzorce projektowe, jest rozwiązaniem, które już wielokrotnie sprawdziło się w zastosowaniach praktycznych jako znakomita metoda tworzenia tylko jednej instancji obiektów określonych klas. Ponadto wzorzec Singleton daje nam do dyspozycji jeden, globalny punkt dostępu do tej instancji; podobnie zresztą jak zmienna globalna, ale bez jej słabych stron.

Projektant: Jakich słabych stron?

Guru: Jakich? OK, dam Ci przykład: jeżeli przypisujesz obiekt do zmiennej globalnej, w momencie uruchomienia aplikacji* musisz utworzyć instancję takiego obiektu. Prawda? A co będzie w sytuacji, kiedy taki obiekt zużywa bardzo dużą ilość zasobów systemowych, a w Twojej aplikacji nigdy nie zajdzie zdarzenie, w którym będzie musiała z niego korzystać? Jak się niebawem przekonasz, dzięki wzorcowi Singleton możemy tworzyć takie obiekty tylko wtedy, kiedy są rzeczywiście potrzebne.

Projektant: To nadal nie wygląda mi na zbyt trudne i skomplikowane zagadnienie.

Guru: Jeżeli masz duże doświadczenie w operowaniu statycznymi zmiennymi i metodami klasy, jak również modyfikatorami dostępu, faktycznie, nie będzie to dla Ciebie zbyt trudne. Niemniej jednak, warto przyjrzeć się bliżej temu, jak działa wzorzec Singleton; jest to o tyle interesujące, że pomimo prosto brzmiącej nazwy uzyskanie poprawnie zaimplementowanego kodu wzorca Singleton wcale nie musi być takie proste. Spróbuj sam sobie odpowiedzieć na pytanie, w jaki sposób można uniemożliwić stworzenie więcej niż jednej instancji obiektu danej klasy? Nie jest to takie oczywiste, jak by się mogło wydawać, prawda?

* To zależy od implementacji. Niektóre maszyny wirtualne utworzą taki obiekt z opóźnieniem, w chwili, gdy będzie potrzebny.

Mały Singleton

Przeprowadzimy teraz małe ćwiczenie w nieco Sokratesowym stylu.

W jaki sposób utworzyłbyś nowy obiekt?

```
new MójObiekt();
```

A co zrobisz w sytuacji, kiedy inny obiekt chciałby stworzyć MójObiekt? Czy ten obiekt mógłby użyć operatora new i utworzyć kolejną instancję obiektu MójObiekt?

Oczywiście.

Czyli tak długo, jak długo mamy daną klasę, możemy zawsze utworzyć jeden lub kilka kolejnych obiektów tej klasy?

Tak. Ale oczywiście tylko wtedy, kiedy jest to klasa publiczna.

A jeżeli nie?

No cóż, jeżeli nie jest to klasa publiczna, obiekt tej klasy mogą stworzyć tylko i wyłącznie klasy znajdujące się w tym samym pakiecie (w tej samej bibliotece klas). Ale nadal mogą one utworzyć po kilka instancji tego samego obiektu.

Hmmm, interesujące.

Czy wiesz, że możesz postąpić w następujący sposób?

Nie, nigdy nie myślałem o takim rozwiązaniu, ale myślę, że to ma sens, ponieważ jest to konstrukcja całkowicie poprawna.

```
public MojaKlasa {

    private MojaKlasa() {}

}
```

Co zatem ona oznacza?

Przypuszczam, że dzięki takiemu rozwiązaniu nie będzie możliwe stworzenie obiektu tej klasy — konstruktor tej klasy jest prywatny.

No dobrze, a zatem, czy istnieje JAKIKOLWIEK obiekt, który mógłby skorzystać z tego prywatnego konstruktora?

Hmmm, myślę, że kod umieszczony wewnątrz klasy MojaKlasa mógłby wywołać prywatnego konstruktora tej klasy. Ale takie rozwiązanie nie ma chyba większego sensu.

Dlaczego nie?

Ponieważ, aby go wywołać, musiałbym już mieć instancję obiektu tej klasy, ale przecież nie mogę jej mieć, bo żadna inna klasa nie może go wcześniej utworzyć. To mi wygląda na stary problem jajka i kury: mogę użyć konstruktora tej klasy wyłącznie poprzez obiekt klasy MojaKlasa, ale z kolei nigdy nie będę w stanie utworzyć instancji takiego obiektu, ponieważ żaden inny obiekt nie może skorzystać z operatora „new MojaKlasa()".

OK. To była tylko taka pierwsza myśl.

A co znaczy coś takiego?

```
public MojaKlasa {

  public static MojaKlasa pobierzInstancję(){
  }
}
```

MojaKlasa jest klasą posiadającą metodę statyczną, którą możemy wywołać w następujący sposób:

```
MojaKlasa.pobierzInstancję();
```

Dlaczego użyłeś nazwy MojaKlasa zamiast nazwy jakiegoś obiektu?

Ponieważ metoda pobierzInstancję() jest metodą statyczną; czyli, innymi słowy, jest to metoda KLASOWA. W odwołaniu do metody statycznej musimy użyć nazwy klasy.

Bardzo interesujące. A co będzie, jeżeli złożymy te dwa elementy razem? Czy teraz mogę utworzyć instancję obiektu klasy MojaKlasa?

```
public MojaKlasa {
  private MojaKlasa() {}

  public static MojaKlasa pobierzInstancję(){
       return new MyClass();
  }
}
```

Uuuuuaaa, pewnie, że możesz!

A teraz, czy możesz pomyśleć o jeszcze innym sposobie stworzenia instancji takiego obiektu?

```
MojaKlasa.pobierzInstancję();
```

A czy teraz możesz dokończyć ten kod tak, aby można było stworzyć tylko JEDNĄ instancję obiektu MojaKlasa?

Tak, myślę, że tak….

(Dokończony kod znajdziesz na następnej stronie)

Analiza klasycznej implementacji wzorca Singleton



Uwaga!

Jeżeli po prostu prześlizgujesz się wzrokiem przez książkę, nie przepisuj na ślepo tego kodu; jak się niebawem przekonasz, posiada on parę pułapek, o których opowiemy w dalszej części rozdziału

Zmienimy nazwę klasy z MojaKlasa na Singleton.

Ta zmienna statyczna posłuży nam do przechowywania naszej jedynej instancji obiektu klasy Singleton.

```
public class Singleton {
    private static Singleton unikalnaInstancja;

    // tutaj umieść inne użyteczne zmienne obiektowe

    private Singleton() {}

    public static Singleton pobierzInstancję() {
        if (unikalnaInstancja == null) {
            unikalnaInstancja = new Singleton();
        }
        return unikalnaInstancja;
    }

    // tutaj umieść inne użyteczne metody
}
```

Konstruktor naszej klasy został zadeklarowany jako prywatny; tylko obiekt klasy Singleton może z niego skorzystać i utworzyć nową instancję obiektu tej klasy!

Metoda pobierzInstancję() udostępnia nam sposób stworzenia instancji obiektu klasy Singleton i zwrócenia go jako wyniku działania tej metody.

Oczywiście, Singleton jest zupełnie normalną klasą; może posiadać inne użyteczne zmienne obiektowe i metody.



Bliskie spotkania z kodem aplikacji

Zmienna unikalnaInstancja przechowuje naszą JEDYNĄ instancję obiektu; pamiętaj, że jest to zmienna statyczna.

Jeżeli zmienna unikalnaInstancja jest pusta, to znaczy, że jeszcze nie utworzyliśmy instancji tego obiektu...

```
if (unikalnaInstancja == null) {
            unikalnaInstancja = new Singleton();
        }
        return unikalnaInstancja;
```

... a jeżeli nie istnieje, tworzymy instancję obiektu klasy Singleton przy użyciu jej prywatnego konstruktora, a następnie przypisujemy ją do zmiennej unikalnaInstancja. Zwróć uwagę na fakt, że jeżeli nigdy nie będziemy potrzebowali instancji obiektu Singleton, instancja taka po prostu nigdy nie zostanie stworzona.

Jeżeli zmienna unikalnaInstancja nie jest pusta, to znaczy, że już wcześniej utworzyliśmy instancję obiektu Singleton i przeskakujemy od razu do polecenia `return`.

Zanim to polecenie zostanie wykonane po raz pierwszy, instancja obiektu Singleton zostanie już stworzona i będziemy mogli ją zwrócić jako wynik działania całej metody.



Wzorce bez tajemnic

Wywiad tygodnia:
Wyznania obiektu Singleton

Rusz głową!: Dzisiaj mamy niezwykłą przyjemność przeprowadzić dla Państwa wywiad z obiektem Singleton. Może rozpoczniemy od tego, że powiesz nam parę słów o sobie.

Singleton: No cóż, mogę powiedzieć, że jestem bardzo unikalny; zostałem stworzony tylko w jednym egzemplarzu!

Rusz głową!: Tylko w jednym?

Singleton: Tak, w jednym. Zostałem stworzony w oparciu o wzorzec Singleton, co zapewnia, że w każdym dowolnym momencie istnieje tylko jedna instancja mnie samego.

Rusz głową!: A czy nie jest to przypadkiem coś w stylu marnotrawstwa? Ktoś poświęcił przecież sporo czasu, aby przygotować pełnowymiarową definicję klasy, a w rezultacie otrzymujemy z tego tylko jeden obiekt?

Singleton: Nie tak do końca! Być JEDYNYM często oznacza być ważnym. Powiedzmy, że masz obiekt, który przechowuje wszystkie ustawienia rejestrów. Z pewnością nie chciałbyś, aby po całej aplikacji przewalały się niezliczone kopie tego obiektu i danych, którymi dysponuje — bez dwóch zdań doprowadziłoby to do strasznego chaosu. Dzięki zastosowaniu obiektów takich jak ja możesz być pewny, że każdy obiekt utworzony w Twojej aplikacji będzie korzystał z tych samych zasobów globalnych.

Rusz głową!: Kontynuuj, proszę…

Singleton: Och, jestem dobry niemal na wszystko. Wiesz, jak to jest, bycie jedynakiem czasami ma swoje zalety. Bardzo często jestem wykorzystywany do zarządzania pulami zasobów, takich jak na przykład pule wątków czy połączeń.

Rusz głową!: I ciągle zawsze sam? Pachnie mi to samotnością…

Singleton: Ponieważ zawsze do takich zadań jestem tylko ja, zwykle jestem bardzo zajęty, aczkolwiek miło by było, gdyby znało mnie nieco szersze grono programistów — wielu z nich pakuje się w poważne kłopoty, ponieważ wiele kopii ich obiektów wałęsa się po całej aplikacji i to w miejscach, o których oni nawet nie mają zazwyczaj pojęcia.

Rusz głową!: Jeżeli mogę zapytać: jaką masz pewność, że jesteś jedynym egzemplarzem samego siebie? Czy przypadkiem ktoś inny dysponujący operatorem new nie może wykonać po prostu polecenia „nowy Ty”?

Singleton: Nie ma szans! Jestem naprawdę unikalny.

Rusz głową!: A zatem, czy programiści składają jakąś przysięgę, że nie będą tworzyć większej ilości Twoich instancji?

Singleton: Oczywiście, że nie. Jeśli mam być szczery, to… hmmmm, to już nieco zbyt osobiste, ale niech tam… Ja nie mam swojego publicznego konstruktora.

Rusz głową!: NIE MASZ PUBLICZNEGO KONSTRUKTORA! Och, przepraszam, nie masz publicznego konstruktora?

Singleton: Zgadza się. Mój konstruktor został zadeklarowany jako prywatny.

Rusz głową!: Jak to zatem działa? W jaki sposób możesz KIEDYKOLWIEK zostać stworzony?

Singleton: Widzisz, aby otrzymać instancję mojego obiektu, nie możesz jej bezpośrednio stworzyć, musisz o nią poprosić. Po prostu, moja klasa macierzysta posiada statyczną metodę o nazwie pobierzInstancję(). Wywołaj ją, a ja od razu się pojawię, gotowy do działania. W praktyce może się nawet zdarzyć tak, że kiedy o mnie poprosisz, ja już będę od jakiegoś czasu pomagał innym obiektom.

Rusz głową!: No cóż, panie Singleton, wygląda na to, że kryją się za Panem całkiem poważne mechanizmy. Dziękujemy za uchylenie rąbka tajemnicy i mamy nadzieję, że niebawem znów będziemy mieli okazję porozmawiać!

Fabryka czekolady

Każdy doskonale zapewne wie, że wszystkie nowoczesne fabryki czekolady posiadają odpowiednie kotły sterowane przy użyciu komputerów. Zadaniem tych kotłów jest wymieszanie odpowiedniej ilości czekolady i mleka, podgrzanie ich do odpowiedniej temperatury i następnie przekazanie otrzymanego półfabrykatu do dalszego etapu produkcji, gdzie następuje tworzenie tak przez wszystkich ukochanych tabliczek.



Poniżej zamieściliśmy kod klasy sterującej przemysłowym kotłem do produkcji czekolady, wyprodukowanym przez firmę CzekoladoManiak sp. z o.o. Przeanalizuj ten kod; zapewne szybko zauważysz, że ich ludzie usiłowali działać bardzo ostrożnie, tak aby nie zdarzyły się różne przykre sytuacje, jak na przykład opróżnienie kotła z pięcioma tysiącami litrów niewymieszanej mikstury, przepełnienie kotła czy też podgrzewanie go w chwili, gdy jest pusty!

```
public class CzekoladowyKocioł {
    private boolean pusty;
    private boolean ugotowany;

    public CzekoladowyKocioł() {
        pusty = true;
        ugotowany = false;
    }

    public void napełniaj() {
        if (jestPusty()) {
            pusty = false;
            ugotowany = false;
            // napełniaj bojler mieszanką mleka i czekolady
        }
    }

    public void opróżniaj() {
        if (!jestPusty() && jestUgotowany()) {
            // opróżniaj bojler z ugotowanej mieszanki mleka i czekolady
            pusty = true;
        }
    }

    public void gotuj() {
        if (!jestPusty() && !jestUgotowany()) {
            // gotuj zawartość kotła
            ugotowany = true;
        }
    }

    public boolean jestPusty() {
        return pusty;
    }

    public boolean jestUgotowany() {
        return ugotowany;
    }
}
```

Ten kod jest uruchamiany wyłącznie w sytuacji, kiedy kocioł jest pusty.

Aby kocioł mógł być napełniany, wcześniej musi być pusty, a kiedy już zostanie napełniony, możemy odpowiednio ustawić flagi pusty i ugotowany.

Aby można było opróżnić kocioł, wcześniej musi być pełny (czyli nie pusty), a zawartość musi być ugotowana. Po opróżnieniu kotła ponownie ustawiamy flagę pusty na wartość prawda (true).

Aby ugotować mieszaninę, kocioł musi być pełny, a zawartość jeszcze nie ugotowana. Po zakończeniu procesu gotowania ustawiamy flagę ugotowany na wartość prawda (true).



WYTĘŻ UMYSŁ

Programiści z firmy CzekoladoManiak sp. z o.o. wykonali kawał dobrej roboty, starając się zabezpieczyć kod aplikacji przed wszelkimi niezbyt korzystnymi wydarzeniami, nie sądzisz? Mimo to prawdopodobnie zacząłeś już podejrzewać, że jeżeli dwa obiekty klasy CzekoladowyKocioł wymkną nam się spod kontroli, mogą się wydarzyć bardzo nieprzyjemne sytuacje.

Co złego może się wydarzyć, jeżeli w naszej aplikacji pojawi się więcej niż jeden obiekt CzekoladowyKocioł?



Zaostrz ołówek

Czy możesz pomóc programistom firmy CzekoladoManiak sp. z o.o. w poprawieniu definicji klasy CzekoladowyKocioł tak, aby była zgodna z regułami wzorca Singleton?

```
public class CzekoladowyKocioł {
    private boolean pusty;
    private boolean ugotowany;

    [                                                    ]

    [        ]CzekoladowyKocioł() {
        pusty = true;
        ugotowany = false;
    }

    [                                                    ]
    [                                                    ]
    [                                                    ]

    public void napełniaj() {
            if (jestPusty()) {
                pusty = false;
                ugotowany = false;
                // napełniaj bojler mieszanką mleka i czekolady
            }
        }
    // reszta kodu klasy CzekoladowyKocioł
}
```

Definicja wzorca Singleton

Poznałeś już klasyczną implementację wzorca Singleton, więc nadszedł czas, aby wygodnie usiąść w fotelu, rozpakować tabliczkę wspaniałej czekolady i na spokojnie przeanalizować zalety tego wzorca.

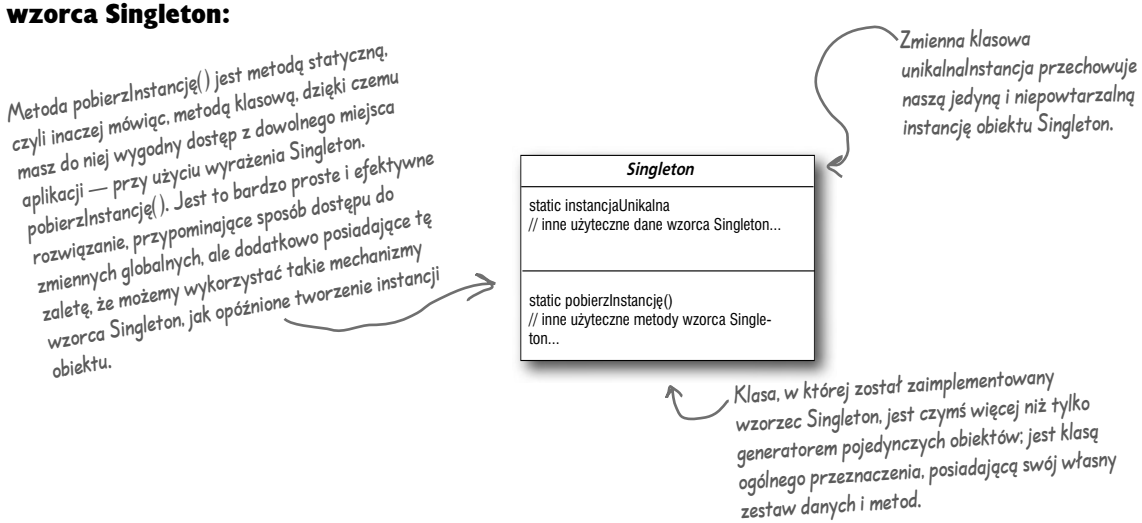
Rozpocznij zatem od przedstawienia zwartej definicji wzorca Singleton:

> **Wzorzec Singleton** zapewnia, że dana klasa będzie miała tylko i wyłącznie jedną instancję obiektu, i zapewnia globalny punkt dostępu do tej instancji.

Jak dotąd, żadnych niespodzianek. Ale spróbujmy zastanowić się nad tym nieco głębiej:

- Co tak naprawdę się tutaj dzieje? Bierzemy daną klasę i pozwalamy jej zarządzać pojedynczą instancją jej obiektu. Zapobiegamy również utworzeniu nowej instancji takiego obiektu przez inne klasy. Aby otrzymać jego instancję, musimy się posłużyć jego klasą macierzystą.
- Zapewniamy również jeden, globalny punkt dostępu do instancji tego obiektu: za każdym razem, kiedy go potrzebujesz, odpytujesz jego klasę macierzystą i ona zwraca Ci jego instancję. Jak już widziałeś, możemy zaimplementować wzorzec w taki sposób, że dany obiekt zostanie utworzony z opóźnieniem, co jest bardzo ważne zwłaszcza w przypadku obiektów, które zużywają dużą ilość zasobów systemowych.

No dobrze, a teraz zobaczmy, jak wygląda diagram klas wzorca Singleton:



Ups, mamy problem...

Wygląda na to, że klasa CzekoladowyKocioł zawiodła nas na całej linii... Pomimo, iż zdecydowanie ulepszyliśmy jej kod, implementując klasyczną wersję wzorca Singleton, metoda napełnij() nie wiadomo w jaki sposób była w stanie rozpocząć procedurę napełniania kotła, nie zważając na to, że był on już pełen gotującego się mleka i czekolady! A to oznacza pięć tysięcy rozlanych litrów mieszanki mleczno-czekoladowej! Co się zatem mogło tutaj stać?

Nie mamy pojęcia, co się stało! Nowy kod z zaimplementowanym wzorcem Singleton sprawował się znakomicie. Jedyne, co nam przychodzi na myśl, to fakt, że chcieliśmy trochę zoptymalizować działanie sterownika CzekoladowegoKotła i dodaliśmy nieco kodu wprowadzającego wielowątkowość w naszej aplikacji.





Czy dodanie wielowątkowości mogło być przyczyną tego całego zamieszania? Czy to nie jest właśnie taka sytuacja, w której, jeżeli zmienna unikalnaInstancja przechowuje jedyną instancję obiektu CzekoladowyKocioł, to wszystkie wywołania metody pobierzInstancję() powinny zwracać tę samą instancję obiektu? Mam rację?

Bądź wirtualną maszyną Java

Mamy dwa wątki, z których każdy wykonuje ten fragment kodu. Twoim zadaniem jest odegranie roli wirtualnej maszyny Java i sprawdzenie, czy zachodzi taka możliwość, aby dwa wątki aplikacji mogły otrzymać dwie różne instancje tego obiektu. Wskazówka: aby zorientować się, czy poszczególne elementy mogą się wzajemnie na siebie nakładać, musisz dokładnie przyjrzeć się sekwencji operacji wykonywanych przez metodę pobierzInstancję() i kolejnym wartościom zmiennej unikalnaInstancja.

Skorzystaj z zamieszczonych poniżej magnesików z kodem, które mogą Ci pomóc w znalezieniu odpowiedzi na pytanie, w jaki sposób kolejno przeplatają się wykonywane elementy kodu, tworzące w efekcie dwie instancje tego samego obiektu.



```
CzekoladowyKocioł kocioł =
    CzekoladowyKocioł.pobierzInstancję();
kocioł.napełnij();
kocioł.gotuj();
kocioł.opróżnij()
```

Zanim przejdziesz na kolejną stronę niniejszej książki, sprawdź i skonfrontuj swoje przemyślenia z rozwiązaniem zamieszczonym na stronie 215.

```
public static CzekoladowyKocioł
    pobierzInstancję() {
    if (unikalnaInstancja == null) {
        unikalnaInstancja =
            new CzekoladowyKocioł();
    }
    return unikalnaInstancja;
}
```

Wątek pierwszy	Wątek drugi	Wartość zmiennej unikalnaInstancja

Jak sobie radzić z wielowątkowością?

Wszystkie nasze nieszczęścia wynikające z zastosowania wielowątkowości mogą być ominięte w trywialnie prosty sposób: poprzez dodanie w deklaracji metody pobierzInstancję() modyfikatora synchronized (czyli inaczej mówiąc, przez zadeklarowanie metody pobierzInstancję() jako metody synchronizowanej):

```
public class Singleton {
    private static Singleton unikalnaInstancja;

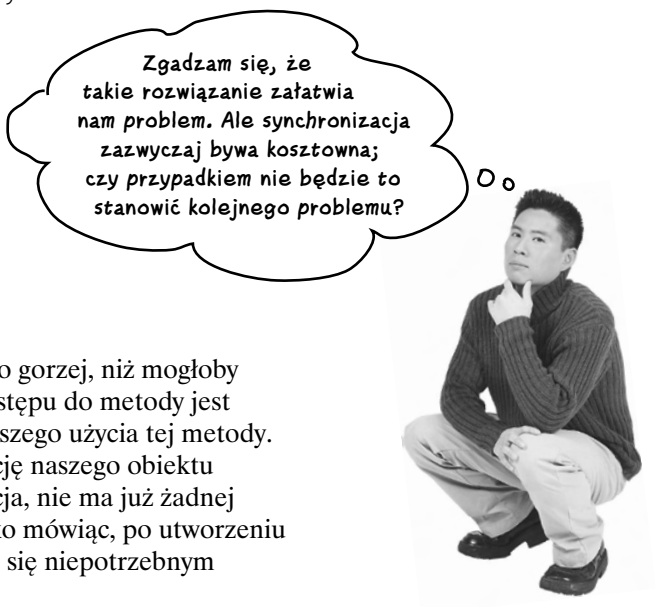
    // tutaj umieść inne użyteczne zmienne obiektowe

    private Singleton() {}

    public static synchronized Singleton pobierzInstancję()
    {
        if (unikalnaInstancja == null) {
            unikalnaInstancja = new Singleton();
        }
        return unikalnaInstancja;
    }

    // tutaj umieść inne użyteczne metody
```

Poprzez dodanie słowa kluczowego synchronized do deklaracji metody pobierzInstancję() zmuszamy poszczególne wątki do oczekiwania na swoją kolejkę w dostępie do tej metody. Dzięki takiemu rozwiązaniu zapobiegamy sytuacji, w której dwa wątki (lub więcej) uzyskują jednoczesny dostęp do tej metody.



Dobre pytanie, a co więcej, jest z tym nieco gorzej, niż mogłoby Ci się wydawać: synchronizacja czasów dostępu do metody jest niezbędna tylko i wyłącznie podczas pierwszego użycia tej metody. Innymi słowy, kiedy utworzymy już instancję naszego obiektu i zapiszemy ją w zmiennej unikalnaInstancja, nie ma już żadnej potrzeby stosowania synchronizacji. Krótko mówiąc, po utworzeniu obiektu utrzymywanie synchronizacji staje się niepotrzebnym obciążeniem dla całego systemu!

Czy możemy ulepszyć działanie wielowątkowości?

W przypadku zdecydowanej większości aplikacji Java musimy oczywiście upewnić się, że implementacja wzorca Singleton działa poprawnie w obecności wielu równolegle działających wątków. Ale przecież synchronizacja w czasie dostępu do metody pobierzInstancję() wygląda na ogromnie kosztowny sposób, więc co innego możemy zrobić z tym fantem?

No cóż, jak by na to nie patrzeć, mamy jeszcze w zanadrzu kilka sprytnych sposobów…

1. Jeżeli wydajność działania metody pobierzInstancję() nie jest krytyczna dla sprawnego funkcjonowania całej aplikacji, po prostu nie rób nic.

Dobrze słyszałeś; jeżeli wywoływanie synchronizowanej metody pobierzInstancję() nie powoduje znaczącego spowolnienia w działaniu Twojej aplikacji, po prostu zapomnij o całym problemie. Synchronizowanie działania metody pobierzInstancję() jest dosyć prostym i efektywnym procesem. Pamiętaj jednak, że w pewnych sytuacjach zastosowanie synchronizowania metody może nawet stukrotnie zmniejszyć wydajność działania całej aplikacji, więc jeżeli kod aplikacji bardzo często będzie korzystał z metody pobierzInstancję(), warto się jednak będzie zastanowić nad wprowadzeniem zmian.

2. Dokonaj modyfikacji kodu tak, aby żądany obiekt był tworzony z wyprzedzeniem, zamiast z opóźnieniem.

Jeżeli Twoja aplikacja zawsze będzie tworzyła instancję obiektu Singleton i korzystała z niej bądź też wczesne stworzenie takiego obiektu nie będzie stanowiło znaczącego dodatkowego obciążenia, być może będziesz chciał utworzyć taki obiekt z wyprzedzeniem, na przykład w taki sposób:

Wyjdź naprzeciw oczekiwaniom i utwórz odpowiednią instancję obiektu w statycznej metodzie inicjalizacyjnej. Taki sposób postępowania z całą pewnością będzie poprawnie funkcjonował w otoczeniu wielowątkowym!

```
public class Singleton {
    private static Singleton unikalnaInstancja = new Singleton();

    private Singleton() {}

    public static Singleton pobierzInstancję() {
        return unikalnaInstancja;
    }
}
```

Mamy już gotową instancję naszego obiektu, więc możemy ją zwracać.

Dzięki takiemu podejściu do całego zagadnienia tworzenie unikalnej instancji obiektu Singleton powierzamy wirtualnej maszynie Java, która dokonuje tego zaraz po załadowaniu naszej klasy. Sposób działania wirtualnej maszyny Java gwarantuje, że instancja obiektu zostanie utworzona na długo przed tym, zanim jakikolwiek wątek będzie usiłował uzyskać dostęp do statycznej zmiennej unikalnaInstancja.

3. Zastosuj metodę „podwójnego blokowania" do zredukowania niezbędnych synchronizacji wykonania metody pobierzInstancję().

Korzystając z metody podwójnego blokowania (ang. *double-checked locking*), najpierw sprawdzamy, czy instancja naszego obiektu już została utworzona; jeżeli nie, WTEDY uruchamiamy synchronizację metody. W ten sposób korzystamy z synchronizacji tylko raz, podczas tworzenia obiektu — czyli mamy dokładnie to, co chcieliśmy uzyskać.

Rzućmy okiem na kod naszej klasy:

```
public class Singleton {
    private volatile *static Singleton unikalnaInstancja;

    private Singleton() {}

    public static Singleton pobierzInstancję() {
        if (unikalnaInstancja == null) {
            synchronized (Singleton.class) {
                if (unikalnaInstancja == null) {
                    unikalnaInstancja = new Singleton();
                }
            }
        }
        return unikalnaInstancja;
    }
}
```

Sprawdzamy, czy została utworzona instancja obiektu — jeżeli nie, przechodzimy do bloku synchronizowanego.

Zwróć uwagę, że synchronizacja działa tylko podczas pierwszego przebiegu metody!

Po wejściu do bloku synchronizowanego ponownie sprawdzamy, czy zmienna ma nadal wartość null — jeżeli tak, tworzymy nową instancję obiektu.

**Zastosowanie słowa kluczowego volatile powoduje, że zmienna unikalnaInstancja jest prawidłowo traktowana w środowisku wielowątkowym, kiedy następuje przypisywanie do niej nowo utworzonej instancji naszego obiektu.*

Jeżeli wydajność działania metody pobierzInstancję jest dla Twojej aplikacji czynnikiem krytycznym, zastosowanie takiej metody implementacji wzorca Singleton może drastycznie zredukować ilość dodatkowego czasu niezbędnego do obsługi jej synchronizacji.



Uwaga!

Podwójne blokowanie nie działa na platformach Java w wersji 1.4 i starszych!

Niestety, zarówno na platformie Java w wersji 1.4, jak i na starszych, wiele maszyn wirtualnych Java posiada taką implementację słowa kluczowego volatile, która przy zastosowaniu metody podwójnego blokowania powoduje nieprawidłową synchronizację całej operacji. Jeżeli musisz korzystać z platformy innej niż Java 5, powinieneś rozważyć zastosowanie innej metody implementacji wzorca Singleton.

A w międzyczasie w fabryce czekolady...



W czasie, kiedy diagnozowaliśmy nasz problem z wielowątkowością, w fabryce czekolady naprawili uszkodzony po ostatniej awarii kocioł, który jest teraz w pełnej gotowości bojowej. Ale zanim go uruchomimy, musimy dokonać w kodzie aplikacji sterującej jego pracą odpowiednich poprawek, które usuną wykryte przez nas problemy z wielowątkowością. Mamy teraz w zanadrzu kilka gotowych rozwiązań, z których każde wymusza podjęcie takich czy innych kompromisów — zatem, na jakie rozwiązanie powinniśmy się zdecydować?



Zaostrz ołówek

Dla każdego rozwiązania opisz potencjalne możliwości jego zastosowania do usunięcia problemów z oprogramowaniem sterującym pracą CzekoladowegoKotła:

Synchronizacja metody pobierzInstancję():

_____

_____

Tworzenie z wyprzedzeniem odpowiedniej instancji obiektu:

_____

_____

Metoda podwójnego blokowania:

_____

_____

Gratulacje!

W tym momencie fabryka czekolady jest naszym kolejnym usatysfakcjonowanym klientem, a firma CzekoladoManiak sp. z o.o. jest bardzo zadowolona z naszej ekspertyzy i poprawek, jakie wprowadziliśmy do kodu ich oprogramowania sterującego pracą CzekoladowegoKotła. Niezależnie od tego, jakie rozwiązanie problemu wielowątkowości zostanie zaimplementowane, stabilność pracy kotła nie powinna być już dłużej zagrożona. Jeszcze raz gratulacje! W tym rozdziale poradziłeś sobie nie tylko z pięcioma tysiącami litrów gorącej czekolady, ale również dogłębnie poznałeś wszystkie potencjalne problemy, z jakimi możesz się spotkać podczas implementacji wzorca Singleton w swoich aplikacjach.

Nie istnieją głupie pytania

P: Jak na taki prosty wzorzec, składający się tylko z jednej klasy, Singleton boryka się z wieloma problemami...

O: No cóż, ostrzegaliśmy Cię o tym już na samym początku tego rozdziału. Ale nie pozwól, aby takie problemy Cię zniechęciły; pomimo, że *poprawna* implementacja wzorca Singleton może być małym wyzwaniem, po przeczytaniu tego rozdziału znasz już dobrze techniki i wskazówki umożliwiające zastosowanie go wszędzie tam, gdzie musisz całkowicie panować nad ilością tworzonych instancji poszczególnych obiektów.

P: Czy nie mogę po prostu stworzyć klasy, w której wszystkie metody i zmienne są zdefiniowane jako statyczne? Czy efekt nie byłby taki sam, jak w przypadku zastosowania wzorca Singleton?

O: Tak, jeżeli tylko ta klasa będzie całkowicie samodzielna i uniezależniona od złożonej inicjalizacji zmiennych. Z drugiej strony jednak, ze względu na sposób, w jaki w języku Java inicjalizowane są zmienne statyczne, cały proces może się stać niezwykle zagmatwany, zwłaszcza jeżeli w grę zostanie zaangażowanych wiele różnych klas. Często realizacja takiego scenariusza prowadzi do niezwykle uciążliwych, ale jednocześnie subtelnych i trudnych do zlokalizowania błędów, spowodowanych taką, a nie inną kolejnością inicjalizowania. Dopóki nie jesteś bezwzględnie zmuszony do zastosowania takiego sposobu implementacji „wzorca Singleton", zdecydowanie lepiej będzie, jeżeli pozostaniesz w klasycznym świecie obiektowym.

P: A co z ClassLoaderami? Słyszałem, że teoretycznie jest szansa, że użycie dwóch (lub więcej) Classloaderów może spowodować, że każdy z nich otrzyma swoją własną instancję obiektu Singleton.



Plotki, jakoby obiekty typu Singleton były pożerane przez procesy zwalniające nieużywane obszary pamięci, są mocno przesadzone.

Zanim powstała platforma Java w wersji 1.2, błąd w implementacji procesów zwalniających nieużywane obszary pamięci (ang. *garbage collector*s) powodował, że obiekty typu Singleton były przez takie procesy przedwcześnie usuwane, jeżeli nie istniało do nich żadne globalne odwołanie. Innymi słowy, mogłeś utworzyć obiekt typu Singleton i jeżeli jedyne odwołanie do tego obiektu znajdowało się wewnątrz jego klasy macierzystej, obiekt taki mógł zostać przedwcześnie zebrany i zniszczony przez proces zwalniający (lub, jak kto woli, proces usuwający śmieci). Zjawisko takie prowadziło do dosyć irytujących błędów, ponieważ po takim nieoczekiwanym zniszczeniu obiektu kolejne wywołanie metody pobierzInstancję() powodowało utworzenie całkowicie nowej instancji tego obiektu. W przypadku wielu aplikacji prowadziło to do dziwacznych zachowań aplikacji, takich jak resetowanie stanu do wartości domyślnych czy też resetowanie puli połączeń sieciowych.

Począwszy od wersji Java 1.2 błąd ten został poprawiony i globalne odwołanie do obiektu Singleton nie jest już dłużej wymagane. Jeżeli jednak z jakiegoś powodu jesteś zmuszony do używania JVM w wersji niższej niż 1.2, powinieneś pamiętać o takim zjawisku. Z drugiej strony, jeżeli używasz nowych wersji JVM, możesz spać spokojnie — żaden program porządkowania pamięci nie dobierze się przypadkowo do Twoich jedynych w swoim rodzaju obiektów.

O: Tak, to prawda, ponieważ każdy obiekt klasy ClassLoader definiuje swoją przestrzeń nazw. Jeżeli używasz dwóch lub więcej ClassLoaderów, możesz tę samą klasę załadować kilka razy (jeden raz na każdy ClassLoader). I teraz jeżeli okaże się, że taka wielokrotnie załadowana klasa to Singleton, ze względu na to, że mamy więcej niż jedną wersję tej klasy, możemy otrzymać więcej niż jedną instancję takiego obiektu. Stąd, jeżeli korzystasz z wielu ClassLoaderów oraz wielu klas typu Singleton, powinieneś być ostrożny. Jednym ze sposobów obejścia tego problemu jest własnoręczne określenie właściwego obiektu ClassLoader.

P: Wszyscy dookoła zawsze powtarzali mi, że każda klasa powinna realizować jedną i tylko jedną operację. Jeżeli dana klasa realizowała dwie operacje bądź więcej, uważana była za klasę zaprojektowaną niezgodnie z regułami programowania zorientowanego obiektowo. Czy przypadkiem wzorzec Singleton nie łamie tej reguły?

O: Powołujesz się tutaj na regułę „jedna klasa, jeden zakres odpowiedzialności". Tak, masz zupełną rację. Wzorzec Singleton jest nie tylko odpowiedzialny za zarządzanie jedną instancją danego obiektu (i zapewnienie jednego, globalnego punktu dostępu do tej instancji), ale jest również odpowiedzialny za to, co jest głównym zadaniem Twojej aplikacji, czymkolwiek by to zadanie nie było. Oczywiście, taki punkt widzenia może być różnie pojmowany i może stać się przedmiotem szerokiej dyskusji. Pomimo tego nietrudno jednak zauważyć, że w takiej klasie znajduje się narzędzie zarządzające instancją swojego własnego obiektu; takie podejście z pewnością powoduje uproszczenie całego zagadnienia. Co więcej, wielu programistów doskonale zna zasady posługiwania się wzorcem Singleton.

P: Chciałem utworzyć klasę, która dziedziczy po klasie Singleton, ale skończyło się to problemami. Czy można tworzyć podklasy dziedziczące po klasie Singleton?

O: Jednym z problemów z tworzeniem podklas dziedziczących po klasie Singleton jest fakt, że konstruktor tej klasy jest prywatny. Nie ma możliwości rozszerzenia klasy, której konstruktor jest prywatny. Stąd pierwszą operacją, jaką powinieneś wykonać, jest zmiana konstruktora klasy tak, aby był publiczny (public) lub chroniony (protected). Ale w takiej sytuacji klasa bazowa to nie jest już *prawdziwy* Singleton, ponieważ inne klasy mogą tworzyć instancje jej obiektów.

Jeżeli jednak zdecydujesz się na modyfikację konstruktora klasy, pojawia się kolejny problem. Implementacja klasy wzorca Singleton jest oparta na zmiennej statycznej, stąd jeżeli dokonasz dziedziczenia, wszystkie nowe podklasy będą korzystały z jednej i tej samej zmiennej statycznej — a Tobie chyba nie do końca o coś takiego chodziło.

Zanim zaczniesz myśleć, jak tego dokonać, zastanów się, co chcesz naprawdę osiągnąć, tworząc klasy podrzędne klasy Singleton. Podobnie jak większość wzorców, Singleton niekoniecznie musi być rozwiązaniem nadającym się do zamieszczenia w bibliotece klas.

Co więcej, zaimplementowanie takiego wzorca do dowolnej klasy jest zadaniem niemal trywialnym. Poza tym, jeżeli w Twojej aplikacji używasz dużej ilości klas typu Singleton, zazwyczaj coś jest nie tak z całym projektem aplikacji. Wzorzec Singleton jest pomyślany tak, aby korzystać z niego stosunkowo niezbyt często.

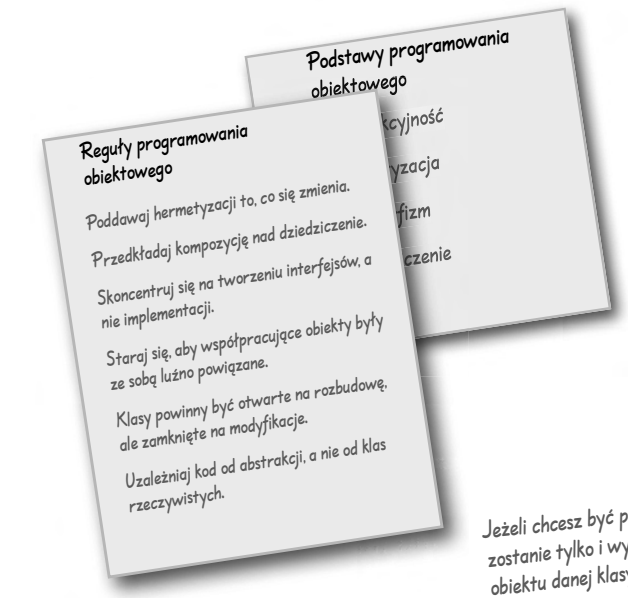
P: Mimo wszystko nadal kompletnie nie rozumiem, dlaczego zastosowanie zmiennych statycznych ma być gorszym rozwiązaniem niż zastosowanie wzorca Singleton.

O: W języku Java zmienne globalne są, ogólnie mówiąc, statycznymi odwołaniami do obiektów. Używanie zmiennych globalnych w taki sposób ma parę poważnych wad. Wspomnieliśmy już o jednej z nich: sprawa tworzenia obiektów z opóźnieniem kontra tworzenie obiektów z wyprzedzeniem. Ale musimy przede wszystkim pamiętać o przeznaczeniu tego wzorca: jego zadaniem jest zapewnienie, że utworzona zostanie tylko i wyłącznie jedna instancja danego obiektu, oraz zapewnienie globalnego punktu dostępu do tej instancji. Zmienna globalna może zapewnić realizację tego drugiego postulatu, ale nie spełnia pierwszego. Zmienne globalne dodatkowo mają tendencję do zachęcania programistów do zaśmiecania przestrzeni nazw dużą ilością globalnych odwołań do małych obiektów. Zastosowanie wzorca Singleton ogranicza takie dążenia, ale i tak mimo wszystko może on być nadużywany.

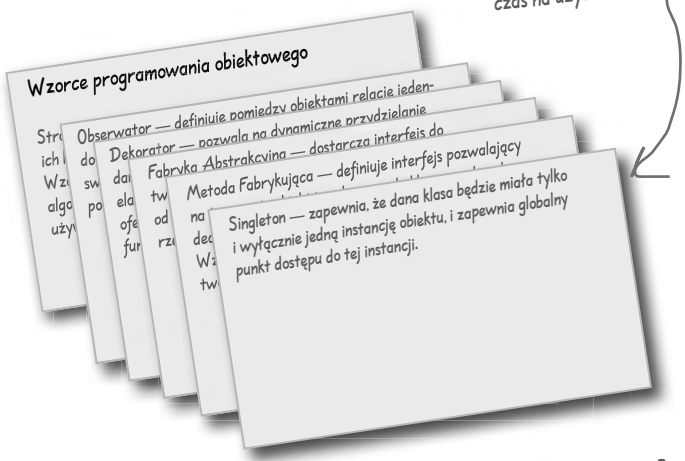


Twoja skrzynka narzędziowa

Właśnie dodałeś kolejny wzorzec do swojej skrzynki narzędziowej. Wzorzec Singleton daje Ci do dyspozycji kolejną metodę tworzenia obiektów — tym razem obiektów jedynych w swoim rodzaju.



Jeżeli chcesz być pewny, że utworzona zostanie tylko i wyłącznie jedna instancja obiektu danej klasy, to znaczy, że nadszedł czas na użycie wzorca Singleton.



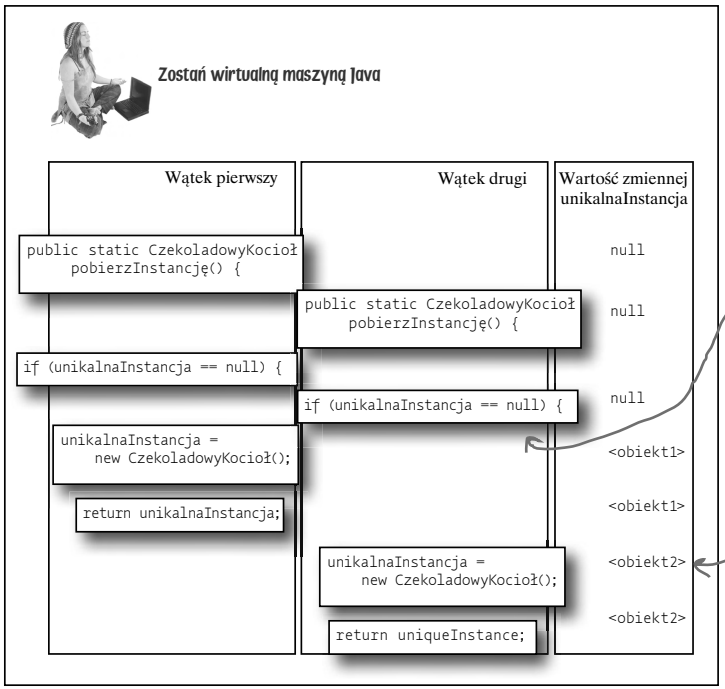
Jak się zdążyłeś przekonać, pomimo swojej pozornej prostoty implementacja wzorca Singleton posiada mnóstwo haczyków i pułapek. Nie zmienia to jednak w niczym faktu, że po przeczytaniu tego rozdziału jesteś gotowy do zastosowania tego wzorca na każdym polu bitwy.

CELNE SPOSTRZEŻENIA

- Wzorzec Singleton daje gwarancję, że będziesz miał tylko i wyłącznie jedną instancję obiektu danej klasy.
- Wzorzec Singleton zapewnia również globalny punkt dostępu do takiej instancji.
- Implementacja wzorca Singleton w języku Java wykorzystuje prywatny konstruktor klasy, jak również metody i zmienne statyczne.
- Wybór odpowiedniej implementacji wzorca Singleton w otoczeniu wielowątkowym powinien być poprzedzony szczegółową analizą wydajności aplikacji i ograniczeń zasobów (w zasadzie powinniśmy przyjąć, że wszystkie aplikacje są wielowątkowe).
- Uważaj na metodę podwójnego blokowania — jej implementacja w środowisku wielowątkowym nie będzie bezpieczna, jeżeli używasz platformy starszej niż Java 2 wersja 5.0.
- Uważaj na ClassLoadery; jeżeli używasz wielu ClassLoaderów, bezpieczna implementacja wzorca Singleton może zostać przełamana i w rezultacie otrzymasz wiele instancji „unikalnego" obiektu.
- Jeżeli używasz maszyny wirtualnej Java starszej niż wersja 1.2, powinieneś uważać na procesy zwalniające nieużywane obszary pamięci, które mogą przypadkowo usunąć Twoje obiekty.

Rozwiązania







Zaostrz ołówek

Czy możesz pomóc programistom firmy CzekoladoManiak sp. z o.o. w poprawieniu definicji klasy CzekoladowyKocioł tak, aby była zgodna z regułami wzorca Singleton?

```
public class CzekoladowyKocioł {
    private boolean pusty;
    private boolean ugotowany;

    private static CzekoladowyKocioł unikalnaInstancja;

    private CzekoladowyKocioł() {
        pusty = true;
        ugotowany = false;
    }

    public static CzekoladowyKocioł pobierzInstancję() {
        if (unikalnaInstancja == null) {
            unikalnaInstancja = new CzekoladowyKocioł ();
        }
        return unikalnaInstancja;
    }

    public void napełniaj() {
        if (jestPusty()) {
            pusty = false;
            ugotowany = false;
            // napełniaj bojler mieszanką mleka i czekolady
        }
    }
    // reszta kodu klasy CzekoladowyKocioł
}
```

Rozwiązania ćwiczeń



Zaostrz ołówek

Dla każdego rozwiązania opisz potencjalne możliwości jego zastosowania do usunięcia problemów z oprogramowaniem sterującym pracą CzekoladowegoKotła:

Synchronizacja metody pobierzInstancję():

Prosta technika, która daje gwarantowane rezultaty. Podczas sterowania kotłem z czekoladą raczej nie będziemy mieli żadnych problemów z wydajnością aplikacji, więc takie rozwiązanie wydaje się dobrym wyborem.

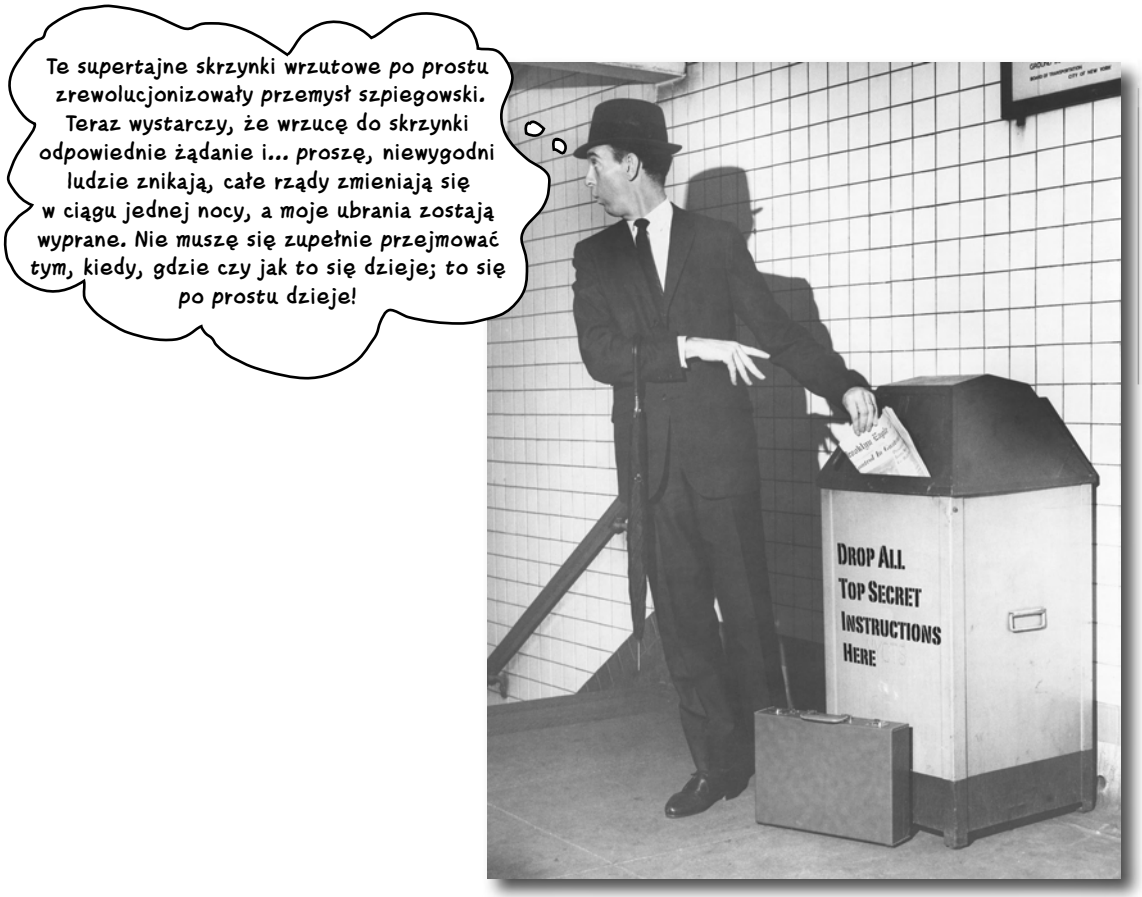
Tworzenie z wyprzedzeniem odpowiedniej instancji obiektu:

Zawsze będziemy chcieli stworzyć w naszej aplikacji obiekt reprezentujący kocioł z czekoladą, więc statyczna inicjalizacja instancji obiektu nie będzie tutaj stanowiła przeszkody. Takie rozwiązanie znakomicie się sprawdzi również jako metoda synchronizowana, chociaż może to już nie być takie oczywiste dla programistów przyzwyczajonych do klasycznej implementacji wzorca Singleton.

Metoda podwójnego blokowania:

Zakładając, że nie będziemy mieli problemów z wydajnością aplikacji, zastosowanie podwójnego blokowania jest chyba trochę przesadne. Co więcej, aby skorzystać z tego rozwiązania, musimy się upewnić, że używamy co najmniej platformy Java 2 w wersji 5.

Hermetyzacja wywołań



W niniejszym rozdziale przeniesiemy hermetyzację na zupełnie nowy poziom: mamy zamiar dokonać hermetyzacji wywołań metod. Zgadza się, dzięki hermetyzacji wywołań metod możemy wykrystalizować pewne fragmenty obliczeń tak, że obiekt wywołujący obliczenia nie musi się martwić, w jaki sposób je wykonać; po prostu wykorzystuje naszą metodę. Z takimi hermetyzowanymi wywołaniami metod możemy również dokonywać wielu zadziwiająco sprytnych operacji, takich jak na przykład zapisywanie ich do dzienników czy też ponowne wykorzystywanie w celu zaimplementowania mechanizmu Cofnij (ang. *Undo*) w naszej aplikacji.

Automatyka w domu i zagrodzie sp. z o.o.
ul. Przemysłowa 1221/2000
00-000 Miasto Przyszłości

Witam!

Niedawno otrzymałem od p. Janusza Monsuna, prezesa firmy Pogodynka sp. z o.o., prezentację i krótki opis ich najnowszej generacji, łatwej do dalszej rozbudowy stacji meteorologicznej. Muszę przyznać, że byłem pod tak wielkim wrażeniem wspaniałej architektury Waszego oprogramowania, że natychmiast postanowiłem poprosić Was o zaprojektowanie bibliotek API dla naszego nowego zdalnego systemu sterowania automatyką domową. Jako wynagrodzenie za Wasze usługi z przyjemnością zaoferujemy Wam pakiet opcji na zakup akcji naszej firmy Inteligentne mieszkanie sp. z o.o.

Do niniejszego pisma załączam przesyłkę, w której znajdziecie Państwo najnowszy prototyp naszego SuperPilota, umożliwiającego zdalne sterowanie automatyką domową. Urządzenie to dysponuje siedmioma programowanymi slotami (każdy slot może zostać przypisany do innego urządzenia gospodarstwa domowego) oraz siedmioma zestawami przycisków włącz-wyłącz, pozwalającymi na uruchamianie bądź zatrzymywanie urządzeń przypisanych do poszczególnych slotów. SuperPilot wyposażony jest również w przycisk Wycofaj, którego użycie powoduje anulowanie operacji wywołanej ostatnio naciśniętym przyciskiem.

Dodatkowo na dysku CD-R (również załączonym do niniejszego pisma) znajdziecie Państwo zestaw klas Java utworzonych przez zespoły programistów z różnych firm. Klasy te pozwalają na sterowanie różnymi urządzeniami gospodarstwa domowego, takimi jak system oświetlenia, wentylatory, jacuzzi, zestawy audio-wideo i inne.

Chcielibyśmy, aby Wasza firma zaprojektowała i stworzyła API pozwalające na programowanie naszego SuperPilota w taki sposób, aby każdy slot mógł sterować danym urządzeniem bądź nawet zestawem urządzeń. Proszę zwrócić szczególną uwagę na fakt, że niezmiernie ważną cechą naszego SuperPilota powinna być nie tylko możliwość obsługi urządzeń, których klasy sterujące załączyliśmy na dysku CD-R, ale również możliwość obsługi innych urządzeń, które pojawią się w przyszłości.

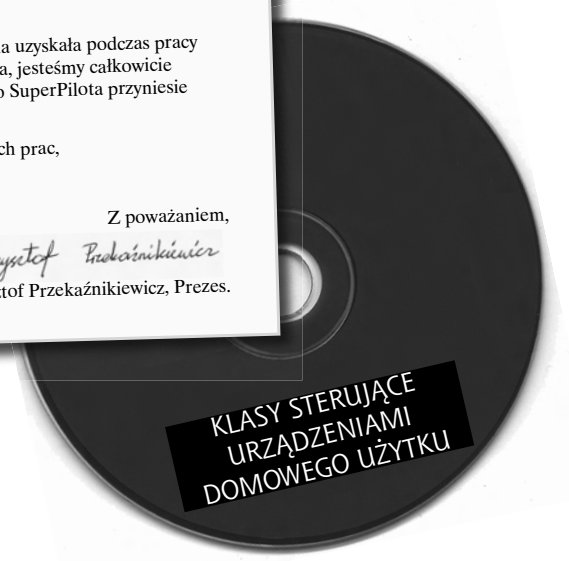
Mając na względzie znakomite rezultaty, jakie Wasza firma uzyskała podczas pracy nad oprogramowaniem stacji meteorologicznej Pogodynka, jesteśmy całkowicie pewni, że Wasza praca nad oprogramowaniem do naszego SuperPilota przyniesie równie wspaniałe efekty.

Z niecierpliwością oczekujemy na pierwsze wyniki Waszych prac,

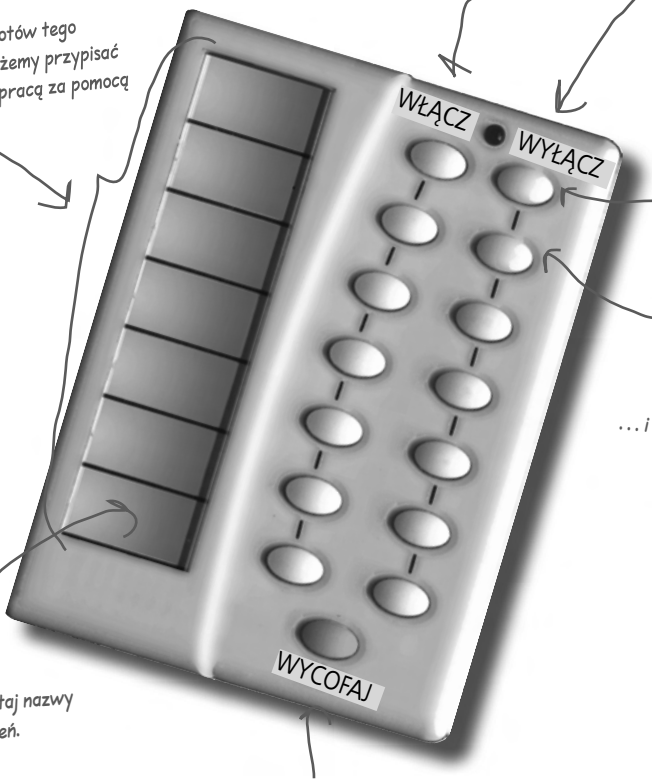
Z poważaniem,

Krzysztof Przekaźnikiewicz

Krzysztof Przekaźnikiewicz, Prezes.



Mamy nową zabawkę! Sprawdzamy, jak działa SuperPilot...



Do każdego z siedmiu slotów przypisany jest osobny zestaw przycisków „WŁĄCZ" oraz „WYŁĄCZ".

Musimy oprogramować siedem slotów tego urządzenia. Do każdego slotu możemy przypisać inne urządzenie i sterować jego pracą za pomocą odpowiednich przycisków.

Te dwa przyciski przeznaczone są do sterowania pracą urządzenia przypisanego do slotu numer jeden...

... a te dwa przyciski sterują pracą urządzenia przypisanego do slotu numer dwa...

... i tak dalej.

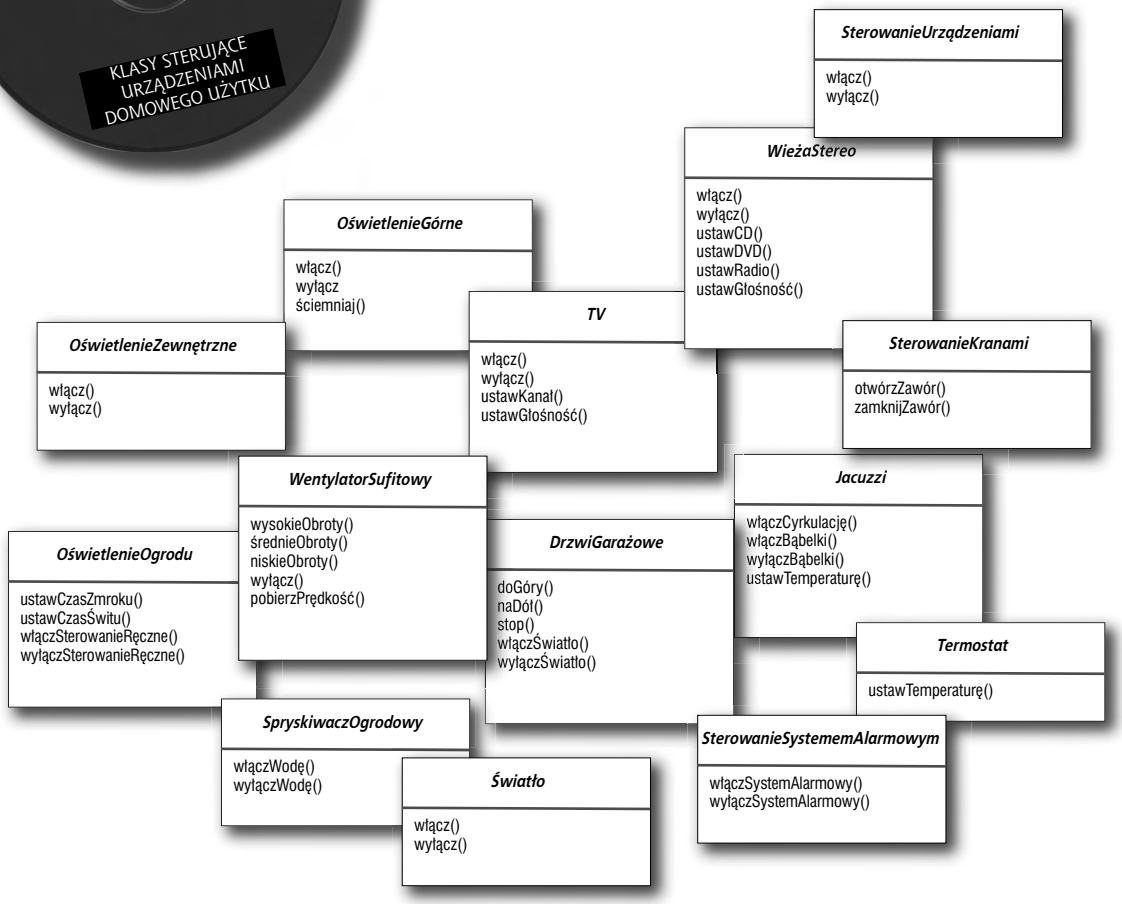
Weź ołówek i zapisz tutaj nazwy poszczególnych urządzeń.

Tutaj znajduje się globalny przycisk „WYCOFAJ", którego zadaniem jest anulowanie operacji wywołanej ostatnio naciśniętym przyciskiem.

Co zawiera otrzymany dysk CD-R?



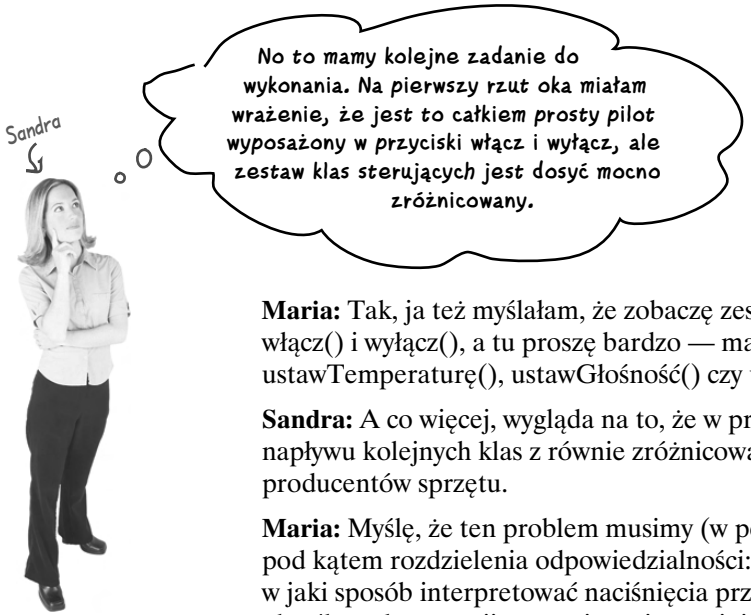
Rzućmy okiem na klasy sterujące urządzeniami domowego użytku, które otrzymaliśmy na dysku CD-R — dzięki temu powinniśmy sobie wyrobić jako taką opinię o interfejsach obiektów, które będą niezbędne do zdalnego sterowania poszczególnymi urządzeniami za pośrednictwem SuperPilota.



Wygląda na to, że mamy tutaj całkiem spory zestaw klas, a co gorsza, poszczególni producenci nie bardzo przykładali się do tego, aby utworzyć jakieś wspólne interfejsy. Co więcej, wszystko wskazuje na to, że w przyszłości możemy oczekiwać pojawienia się nowych urządzeń, a co za tym idzie, nowych klas sterujących. Krótko mówiąc, zaprojektowanie API dla tego systemu zdalnego sterowania może być bardzo interesującym przedsięwzięciem. A zatem, do dzieła!

Zasłyszane w sąsiednim boksie...

Wygląda na to, że Twoje koleżanki z zespołu już rozpoczęły dyskusję o tym, w jaki sposób najlepiej zaprojektować API do systemu zdalnego sterowania...



Maria: Tak, ja też myślałam, że zobaczę zestaw klas posiadających metody włącz() i wyłącz(), a tu proszę bardzo — mamy metody takie jak ściemniaj(), ustawTemperaturę(), ustawGłośność() czy ustawKanał().

Sandra: A co więcej, wygląda na to, że w przyszłości możemy się spodziewać napływu kolejnych klas z równie zróżnicowanymi metodami od różnych producentów sprzętu.

Maria: Myślę, że ten problem musimy (w pewnym sensie) przeanalizować pod kątem rozdzielenia odpowiedzialności: SuperPilot powinien wiedzieć, w jaki sposób interpretować naciśnięcia przycisków oraz jak żądać wykonania określonych operacji, natomiast nie powinien być obciążony informacjami na temat samych urządzeń, np. nie musi wiedzieć, w jaki sposób uruchamia się jacuzzi.

Sandra: Wygląda to całkiem sensownie. Ale z drugiej strony, jeżeli nasz SuperPilot będzie dosyć tępy i jego wiedza zamknie się w tym, w jaki sposób można żądać wykonania określonych operacji, to w jaki sposób oprogramujemy go tak, aby był w stanie na przykład zażądać zapalenia górnego oświetlenia w domu czy też otwarcia drzwi garażu?

Maria: Nie jestem pewna, ale wydaje mi się, że nie chcielibyśmy, aby SuperPilot musiał znać szczegóły dotyczące poszczególnych klas sterujących urządzeniami.

Sandra: Co masz na myśli?

Maria: Z pewnością nie chciałabym, aby oprogramowanie SuperPilota składało się z serii wyrażeń typu if, takich jak na przykład „if slot1 == Światło, then Światło.włącz(), else if slot1==Jacuzzi then Jacuzzi.włączBąbelki()". Przecież sama wiesz, że takie rozwiązanie nie jest poprawne.

Sandra: Zgadzam się z Tobą w zupełności. Za każdym razem, kiedy pojawiałaby się nowa klasa sterująca, musielibyśmy dokonywać ponownej modyfikacji naszego kodu, potencjalnie narażając go na wprowadzenie nowych błędów, nie mówiąc już o ilości dodatkowej pracy, którą musielibyśmy przy takiej okazji wykonać!



Hej, po prostu nie mogłem się powstrzymać, żeby Was nie podsłuchiwać. Już po przeczytaniu pierwszego rozdziału tej książki ostro zabrałem się za studiowanie wzorców projektowych i zgadnijcie, co odkryłem? Istnieje wzorzec, który idealnie nadaje się do zastosowania w opisywanej przez Was sytuacji — to wzorzec Polecenie.

Maria: Tak? W takim razie opowiedz o nim coś więcej.

Jacek: Wzorzec Polecenie *(ang. Command)* pozwala na usunięcie sprzężenia pomiędzy elementem wystawiającym żądanie a elementem, który finalnie wykonuje żądaną operację. W naszej sytuacji elementem wystawiającym żądanie będzie SuperPilot, a elementem wykonującym żądaną operację będzie instancja obiektu odpowiedniej klasy sterującej danym urządzeniem.

Sandra: Ale czy coś takiego jest naprawdę możliwe? W jaki sposób usunąć takie sprzężenie? Przecież, jak by na to nie patrzeć, gdy nacisnę odpowiedni przycisk, to właśnie pilot powinien spowodować, że np. włączy się oświetlenie.

Jacek: Możesz tego dokonać, tworząc tzw. obiekty — Polecenia. Obiekt-polecenie powoduje hermetyzację żądania wykonania danej operacji (na przykład włączenia oświetlenia) przez określony obiekt (na przykład obiekt reprezentujący oświetlenie w jadalni). Tak więc, jeżeli do każdego przycisku przypiszemy określony obiekt-polecenie, to kiedy dany przycisk zostanie wciśnięty, poprosimy, aby obiekt ten wykonał swoje zadanie. SuperPilot nie ma zielonego pojęcia, co to jest za zadanie — to właśnie obiekt-polecenie wie, w jaki sposób należy „potraktować" obiekt docelowy, aby została wykonana taka czy inna operacja. Czyli, jak same widzicie, w ten sposób nasz SuperPilot zostanie całkowicie pozbawiony sprzężenia z obiektem reprezentującym oświetlenie jadalni!

Sandra: To rzeczywiście brzmi bardzo obiecująco — wygląda na to, że poruszamy się w dobrym kierunku.

Maria: Niby tak, ale mimo wszystko jakoś nadal nie mogę zrozumieć zasad funkcjonowania tego wzorca…

Jacek: Jeżeli założymy, że nie istnieją sprzężenia pomiędzy poszczególnymi obiektami, to faktycznie zrozumienie zasad działania tego wzorca może sprawiać pewne problemy.

Maria: No dobrze, ale przynajmniej powiedz mi, czy chociaż ogólnie moje wyobrażenia na temat sposobu działania tego wzorca odpowiadają rzeczywistości: korzystając z niego, możemy utworzyć API pozwalające na załadowanie tych — jak Ty to nazywasz — obiektów-poleceń do odpowiednich slotów pilota odpowiadających poszczególnym przyciskom. Dzięki temu kod obsługujący samego SuperPilota pozostanie bardzo prosty. A obiekty-polecenia hermetyzują sposoby wykonywania poszczególnych operacji, współpracując w tym celu z obiektami docelowymi, których takie operacje dotyczą.

Jacek: Tak, można to tak określić. Myślę również, że zastosowanie tego wzorca może Wam zdecydowanie ułatwić implementację przycisku Wycofaj, aczkolwiek jeszcze nie zdążyłem doczytać wszystkiego na ten temat.

Maria: A jednak brzmi to bardzo zachęcająco, aczkolwiek wydaje mi się, że czeka mnie jeszcze całkiem sporo pracy, zanim uda mi się na dobre „oswoić" ten wzorzec.

Sandra: Mnie chyba też…

A w międzyczasie w naszym barze szybkiej obsługi...

czyli

krótkie wprowadzenie do wzorca Polecenie

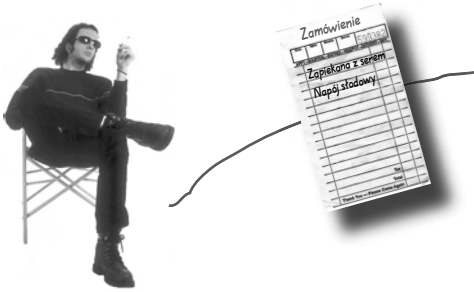


Jak już dowiedzieliśmy się od Jacka, zrozumienie zasad funkcjonowania wzorca Polecenie tylko i wyłącznie na podstawie ich opisu może sprawiać pewne problemy. Ale nie obawiaj się, mamy kilku przyjaciół, którzy nam chętnie pomogą w tej materii: pamiętasz historyjkę zasłyszaną w lokalnym barze szybkiej obsługi, który odwiedziliśmy w rozdziale 1.? Co prawda, minęło już sporo czasu od chwili, kiedy spotkaliśmy się z Alicją, Florentyną i obsługującym je kelnerem, ale właśnie teraz mamy dobry powód, aby tam na chwilę powrócić (i to nie tylko z powodu jedzenia i miłej atmosfery): zamówienie obiadu pomoże nam zrozumieć zasady funkcjonowania wzorca Polecenie.

Powróćmy zatem do naszego obiadu i postarajmy się przeanalizować interakcje, jakie zachodzą pomiędzy klientami, kelnerką, zamówieniami potraw oraz kucharzem, który je przygotowuje. Dzięki takiej analizie wzajemnych oddziaływań będziesz w stanie zrozumieć zadania poszczególnych obiektów wzorca Polecenie, jak również zorientować się, w jaki sposób działa usuwanie sprzężenia pomiędzy podmiotem zlecającym a wykonawcą zlecenia. A potem? Potem pozostanie już tylko z pasją zabrać się za rozgryzanie problemu API do naszego systemu zdalnego sterowania urządzeniami gospodarstwa domowego.

A zatem — idziemy do baru szybkiej obsługi!

Wszyscy wiemy, w jaki sposób zazwyczaj realizowane jest zamówienie w barze szybkiej obsługi:



❶ **Ty, czyli Klient, przekazujesz Kelnerce swoje Zamówienie.**



❷ **Kelnerka bierze Zamówienie, idzie do kuchni, kładzie je w okienku i krzyczy: „Kolejne zamówienie!”.**

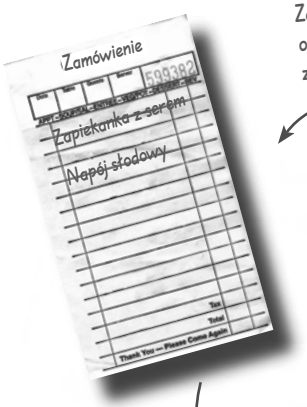




❸ **Kucharz przygotowuje określone danie na podstawie Zamówienia.**

Przyjrzyjmy się nieco dokładniej wzajemnym interakcjom...

... a zakładając, że nasz bar szybkiej obsługi znajduje się nie gdzie indziej, ale właśnie w Obiektowie, spróbujmy spojrzeć na cały proces, angażując już do niego pewne metody i obiekty!



Zamówienie składa się z blankietu zamówienia oraz z pozycji wybranych w menu, które zostały zapisane na blankiecie zamówienia.

utwórzZamówienie()

Poproszę zapiekankę z serem i napój słodowy.



Zacznij tutaj

Klient wie już, czego chce, i składa zamówienie.

pobierzZamówienie()



Kelnerka przyjmuje Zamówienie od klienta i kiedy już się z tym upora, wywołuje metodę realizujZamówienie(). Metoda ta rozpoczyna proces realizacji Zamówienia złożonego przez klienta.

realizujZamówienie()



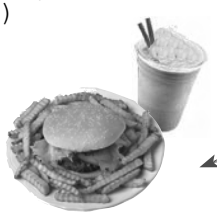
Na Zamówieniu znajdują się wszelkie instrukcje niezbędne do tego, aby przygotować właściwy posiłek. Zamówienie kieruje pracą Kucharza, posługując się metodami takimi, jak przygotujZapiekankę().

przygotujZapiekankę(), przygotujNapój()



Kucharz realizuje instrukcje podane w Zamówieniu i przygotowuje odpowiedni posiłek.

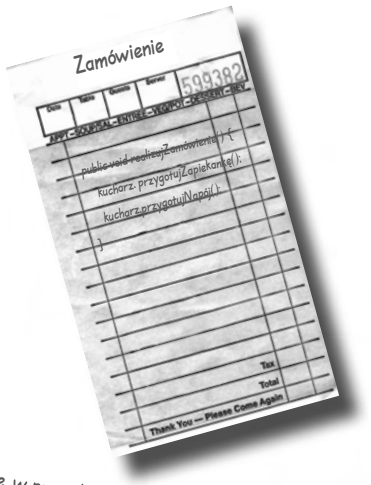
produkt końcowy



Zadania i zakresy odpowiedzialności

BlankietZamówienia hermetyzuje żądanie przygotowania posiłku

Pomyśl o BlankiecieZamówienia jako o obiekcie, który odgrywa rolę żądania przygotowania posiłku. Podobnie jak każdy inny obiekt, blankiet może być przekazywany pomiędzy różnymi podmiotami — od Kelnerki aż do okienka, w którym składane są blankiety zamówień lub po prostu do innej Kelnerki, która właśnie zaczyna swoją zmianę. Blankiet posiada również swój interfejs, który składa się tylko z jednej metody, realizujZamówienie(), hermetyzującej operacje niezbędne do przygotowania danego posiłku oraz posiada odwołania do obiektu, który taki posiłek musi przygotować (w naszym przypadku oczywiście chodzi o Kucharza). Zamówienie jest hermetyzowane również w tym sensie, że Kelnerka tak naprawdę nie musi wiedzieć, co zostało zamówione czy nawet, kto będzie przygotowywał taki posiłek; musi ona jedynie przekazać blankiet zamówienia do odpowiedniego okienka w kuchni i krzyknąć: „Kolejne zamówienie!”.



No dobrze, w prawdziwym barze kelnerka raczej świetnie wiedziałaby, co znajduje się na blankiecie zamówienia i kto je będzie realizował, ale przecież jesteśmy w Obiektowie… Przestań się czepiać i pracuj z nami dalej!

Zadaniem Kelnerki jest pobranie blankietu zamówienia, a następnie wywołanie metody realizujZamówienie()

Zadanie Kelnerki jest bardzo proste: musi odebrać zamówienie od klienta, w drodze powrotnej do okienka składania zamówień powinna pomagać innym klientom, a następnie, po dotarciu do okienka, ma obowiązek wywołać metodę realizujZamówienie, która spowoduje przygotowanie odpowiedniego posiłku. Jak już wcześniej tłumaczyliśmy, w Obiektowie panują takie zwyczaje, że Kelnerka nie musi przejmować się tym, co znajduje się w zamówieniu, ani tym bardziej, kto je będzie realizował; ona wie tylko, że zamówienie posiada metodę realizujZamówienie(), której wywołanie powoduje wykonanie zadania.

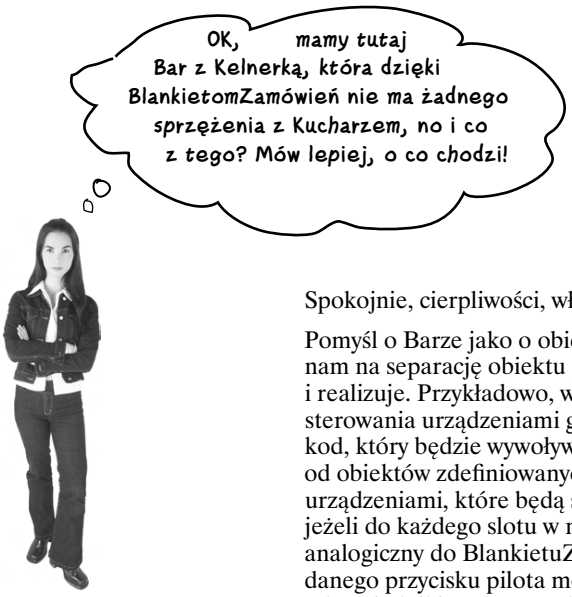
W miarę upływu czasu jej metoda pobierzZamówienie() staje się metodą parametryzowaną różnymi blankietami zamówień składanymi przez różnych klientów, ale to jej nie przeraża; Kelnerka wie, że wszystkie BlankietyZamówień posiadają zaimplementowaną metodę realizujZamówienie(), którą ona może wywołać zawsze wtedy, kiedy nadejdzie czas przygotowania danego posiłku.



Kucharz posiada wiedzę, która pozwala mu na przygotowywanie posiłków

To właśnie Kucharz jest obiektem, który naprawdę wie, jak przygotowywać posiłki. Po wywołaniu przez Kelnerkę metody realizujZamówienie() pałeczkę przejmuje Kucharz i rozpoczyna implementację wszystkich metod, które są niezbędne do przygotowania danego posiłku. Zwróć uwagę na brak jakiegokolwiek sprzężenia pomiędzy Kelnerką a Kucharzem; Kelnerka posiada BlankietyZamówień, które hermetyzują szczegółowe informacje o posiłku, i musi tylko dla każdego blankietu wywołać metodę realizujZamówienie(). Analogicznie nasz wesoły Kucharz otrzymuje swoje instrukcje na blankietach zamówień i nigdy nie musi bezpośrednio kontaktować się z Kelnerką.





Spokojnie, cierpliwości, właśnie dochodzimy do sedna sprawy…

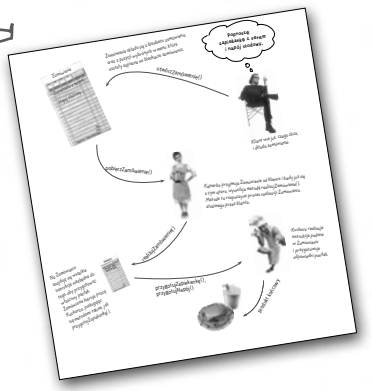
Pomyśl o Barze jako o obiektowym wzorcu projektowym, który pozwala nam na separację obiektu generującego żądania od obiektu, który je odbiera i realizuje. Przykładowo, w naszej bibliotece API dla systemu zdalnego sterowania urządzeniami gospodarstwa domowego musimy odseparować kod, który będzie wywoływany po naciśnięciu danego przycisku SuperPilota, od obiektów zdefiniowanych w klasach sterujących poszczególnymi urządzeniami, które będą się zajmowały realizacją takich żądań. A co będzie, jeżeli do każdego slotu w naszym SuperPilocie zostanie przypisany obiekt analogiczny do BlankietuZamówienia w naszym Barze? Wtedy po naciśnięciu danego przycisku pilota możemy wywołać metodę będącą swego rodzaju odpowiednikiem czy też ekwiwalentem barowej metody realizujZamówienie(), która na przykład włączy nam oświetlenie — a najlepsze jest to, że sam pilot nie będzie znał żadnych szczegółów odnośnie tego, w jaki sposób dana operacja została wykonana, co zostało wykonane czy też jakie obiekty były w to zaangażowane.

A teraz spróbujemy wrzucić drugi bieg i przenieść nasze dotychczasowe, barowe rozważania na kanwę poważnego wzorca Polecenie…



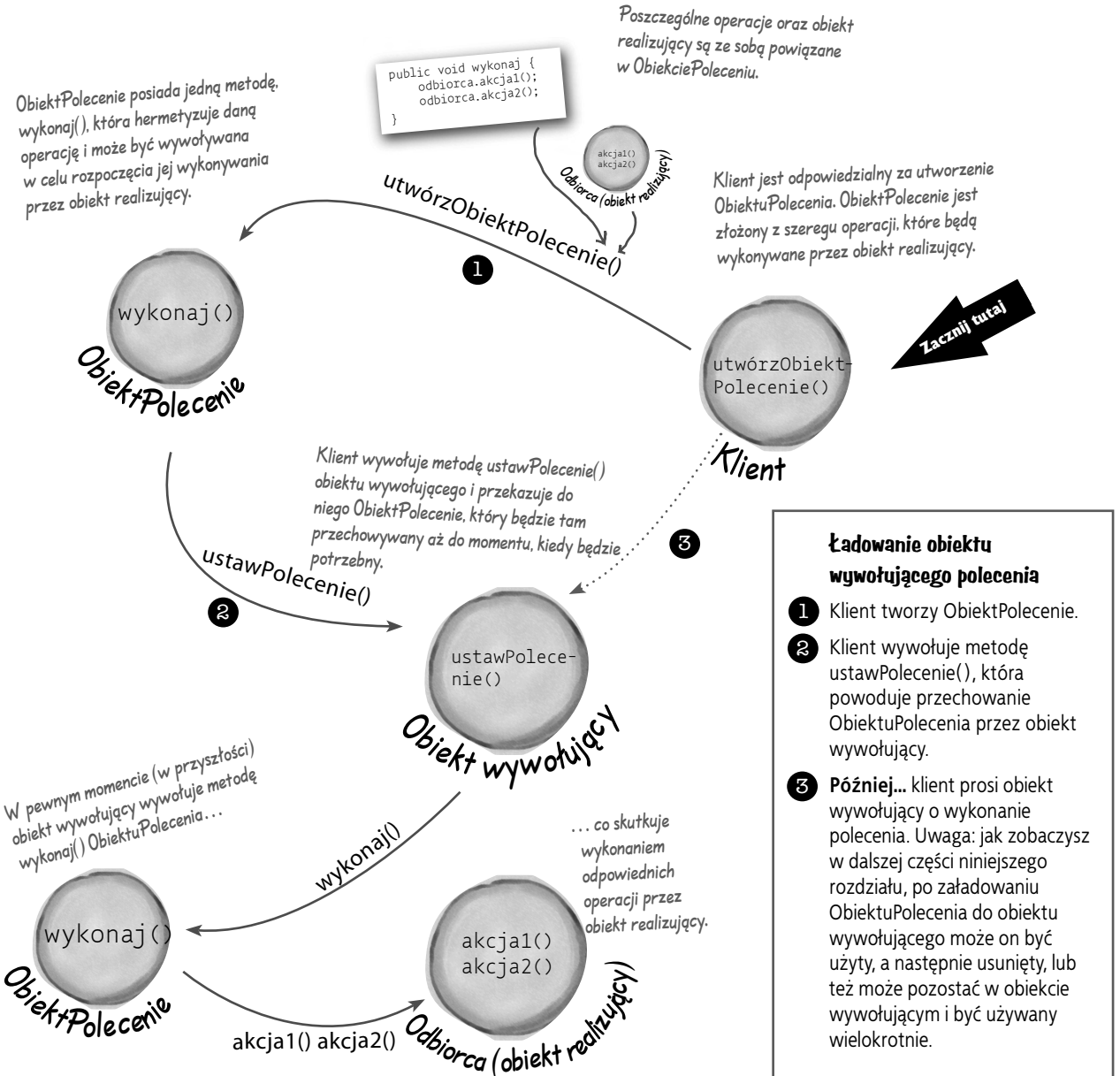
WYTĘŻ UMYSŁ

Zanim przejdziemy dalej, powinieneś poświęcić nieco czasu na ponowne, gruntowne przestudiowanie diagramu barowych zadań i zakresów odpowiedzialności, jaki zamieściliśmy dwie strony wcześniej. Kiedy będziesz już miał wszystkie tajniki barowych obiektów w małym palcu, będzie to znaczyło, że jesteś gotowy na spotkanie z prawdziwym wzorcem Polecenie!



Od Baru do wzorca Polecenie

No dobrze, spędziliśmy w Barze wystarczająco dużo czasu, aby doskonale poznać panujące tam zwyczaje, zależności i metody działania, zatem teraz ze spokojem możemy przystąpić do przeniesienia naszych barowych doświadczeń na nowy diagram odzwierciedlający wzorzec Polecenie. Przekonasz się, że wszyscy gracze pozostali Ci sami, zmieniły się tylko ich nazwy.



Ładowanie obiektu wywołującego polecenia

1. Klient tworzy ObiektPolecenie.
2. Klient wywołuje metodę ustawPolecenie(), która powoduje przechowanie ObiektuPolecenia przez obiekt wywołujący.
3. **Później...** klient prosi obiekt wywołujący o wykonanie polecenia. Uwaga: jak zobaczysz w dalszej części niniejszego rozdziału, po załadowaniu ObiektuPolecenia do obiektu wywołującego może on być użyty, a następnie usunięty, lub też może pozostać w obiekcie wywołującym i być używany wielokrotnie.



Połącz poszczególne obiekty i metody barowe z odpowiadającymi im elementami wzorca Polecenie.

Bar	Wzorzec Polecenie
Kelnerka	Polecenie
Kucharz	wykonaj()
realizujZamówienie()	Klient
Zamówienie	Obiekt wywołujący
Gość	Obiekt realizujący
pobierzZamówienie()	ustawPolecenie()

Nasze pierwsze POLECENIE



Czy nie nadszedł już czas na utworzenie naszego pierwszego POLECENIA? Ależ tak, więc spróbujmy napisać nieco kodu dla naszego systemu zdalnego sterowania urządzeniami gospodarstwa domowego. Ponieważ jeszcze nie zastanawialiśmy się, w jaki sposób zaprojektować naszą bibliotekę API, utworzenie kilku elementów znajdujących się na dole hierarchii może okazać się w tym cokolwiek pomocne…

Implementacja interfejsu Polecenie

Na początku to, co najważniejsze: wszystkie POLECENIA posiadają zaimplementowany ten sam interfejs, składający się tylko z jednej metody. W Barze nazywaliśmy ją realizacjaZamówienia(), ale zazwyczaj będziemy używali po prostu nazwy wykonaj().

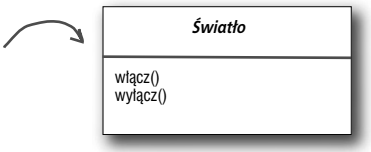
Oto kod interfejsu Polecenie:

```
public interface Polecenie {
    public void wykonaj();
}
```

Proste. Wszystko, czego nam trzeba, to jedna metoda nazywana wykonaj().

Implementacja POLECENIA odpowiedzialnego za włączanie oświetlenia

Załóżmy teraz, że chciałbyś zaimplementować polecenie, które będzie odpowiadało za włączanie oświetlenia. Jak wynika z dokumentacji dostarczonego zestawu klas sterujących, klasa Światło ma dwie metody: włącz() i wyłącz(). Poniżej przedstawiamy przykładowy sposób implementacji naszego POLECENIA:




```
public class PolecenieWłączŚwiatło implements
Polecenie {
    Światło światło;

    public PolecenieWłączŚwiatło(Światło światło)
    {
        this. światło = światło;
    }

    public void wykonaj() {
        światło.włącz();
    }
}
```

To jest polecenie, więc musimy zaimplementować interfejs Polecenie.

Do konstruktora klasy przekazywany jest rodzaj światła, jakim to polecenie ma sterować — powiedzmy, że chodzi o światło w jadalni — który jest następnie przechowywany w zmiennej obiektowej światło. Kiedy zostanie wywołana metoda wykonaj(),właśnie ten obiekt reprezentujący dane światło będzie obiektem realizującym nasze żądanie.

Metoda wykonaj() wywołuje metodę włącz() obiektu realizującego (czyli obiektu reprezentującego światło, którym sterujemy).

Mamy już gotowy kod klasy PolecenieWłączŚwiatło, więc zobaczmy, czy możemy go użyć…

Zastosowanie POLECENIA

Teraz na własne potrzeby dokonajmy pewnego uproszczenia: załóżmy, że nasz wspaniały pilot dysponuje tylko jednym przyciskiem i jednym slotem, do którego możemy przypisać wybrane urządzenie:

```
public class MiniPilot {
    Polecenie slot;

    public MiniPilot() {}

    public void ustawPolecenie(Polecenie polecenie) {
        slot = polecenie;
    }

    public void przyciskZostałNaciśnięty() {
        slot.wykonaj();
    }
}
```

Do przechowywania poleceń możemy wykorzystać tylko jeden slot, który będzie sterował tylko jednym urządzeniem.

Tutaj mamy metodę, przy użyciu której możemy ustawiać polecenie, z którego będzie korzystał slot pilota. Metoda ta może być wywoływana wielokrotnie, dzięki czemu możemy zmieniać zachowania przycisku naszego pilota.

Ta metoda jest wywoływana, kiedy zostanie naciśnięty przycisk naszego pilota. Wszystko, co musimy tutaj zrobić, to wywołać metodę wykonaj() polecenia, które jest aktualnie przypisane do slotu pilota.

Tworzymy prosty test naszego MiniPilota

Poniżej znajdziesz fragment kodu umożliwiający przeprowadzenie prostego testu naszego MiniPilota. Rzućmy na niego okiem i zastanówmy się, w jaki sposób poszczególne elementy składowe wpasowują się w diagram wzorca Polecenie:

W kategoriach wzorca Polecenie to właśnie jest nasz Klient.

MiniPilot jest naszym obiektem wywołującym; do niego właśnie zostanie przekazane POLECENIE, które będzie wykorzystane do wystawienia żądania.

```
public class MiniPilotTest {
    public static void main(String[] args) {
        MiniPilot pilot = new MiniPilot();
        Światło światło = new Światło();
        PolecenieWłączŚwiatło włączŚwiatło =
            new PolecenieWłączŚwiatło(światło);

        pilot.ustawPolecenie(włączŚwiatło);
        pilot.przyciskZostałNaciśnięty();

    }
}
```

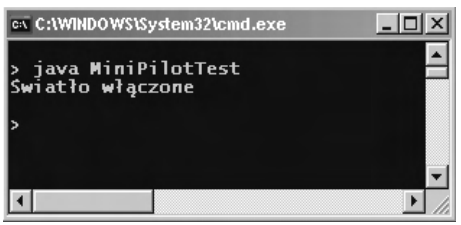
Teraz tworzymy obiekt Światło, który będzie obiektem realizującym nasze polecenie.

Tutaj tworzymy polecenie i wiążemy je z obiektem realizującym.

I wreszcie tutaj symulujemy naciśnięcie przycisku MiniPilota.

Tutaj następuje przekazanie polecenia do obiektu wywołującego.

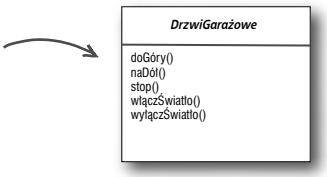
Oto wyniki działania kodu testującego.





Zaostrz ołówek

Nadszedł czas, abyś dokonał próby implementacji klasy PolecenieOtwórzDrzwiGarażowe. Najpierw musisz utworzyć kod klasy podanej poniżej, a do tego celu niezbędny będzie diagram klasy DrzwiGarażowe.




```
public class PolecenieOtwórzDrzwiGarażowe
        implements Polecenie {




}
```

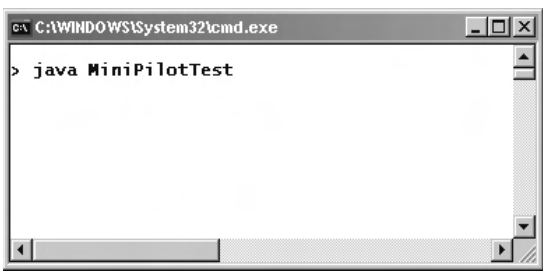
Tutaj wstaw swój kod.

A teraz, kiedy już masz gotową klasę, zastanów się, jakie będą wyniki działania poniższego kodu? (Wskazówka: metoda DrzwiGarażowe.doGóry() po zakończeniu procesu otwierania powoduje wyświetlenie komunikatu: „Drzwi garażowe są otwarte”).

```
public class MiniPilotTest {
    public static void main(String[] args) {
        MiniPilot pilot = new MiniPilot();
        Światło światło = new Światło();
        DrzwiGarażowe drzwiGarażowe = new DrzwiGarażowe();
        PolecenieWłączŚwiatło włączŚwiatło = new PolecenieWłączŚwiatło
(światło);
        PolecenieOtwórzDrzwiGarażowe otwórzDrzwiGarażowe =
            new PolecenieOtwórzDrzwiGarażowe(drzwiGarażowe);

        pilot.ustawPolecenie(włączŚwiatło);
        pilot.przyciskZostałNaciśnięty();
        pilot.ustawPolecenie(otwórzDrzwiGarażowe);
        pilot.przyciskZostałNaciśnięty();
    }
}
```

Tutaj umieść wyniki działania programu.



Definicja wzorca Polecenie

Spędziliśmy trochę czasu w miłym barze w Obiektowie i częściowo zaimplementowaliśmy bibliotekę API systemu zdalnego sterowania urządzeniami gospodarstwa domowego — dzięki temu zdążyłeś się już całkiem dobrze zorientować we wzajemnych zależnościach pomiędzy klasami oraz obiektami wzorca Polecenie. Teraz przedstawimy formalną definicję tego wzorca i objaśnimy niektóre jego detale.

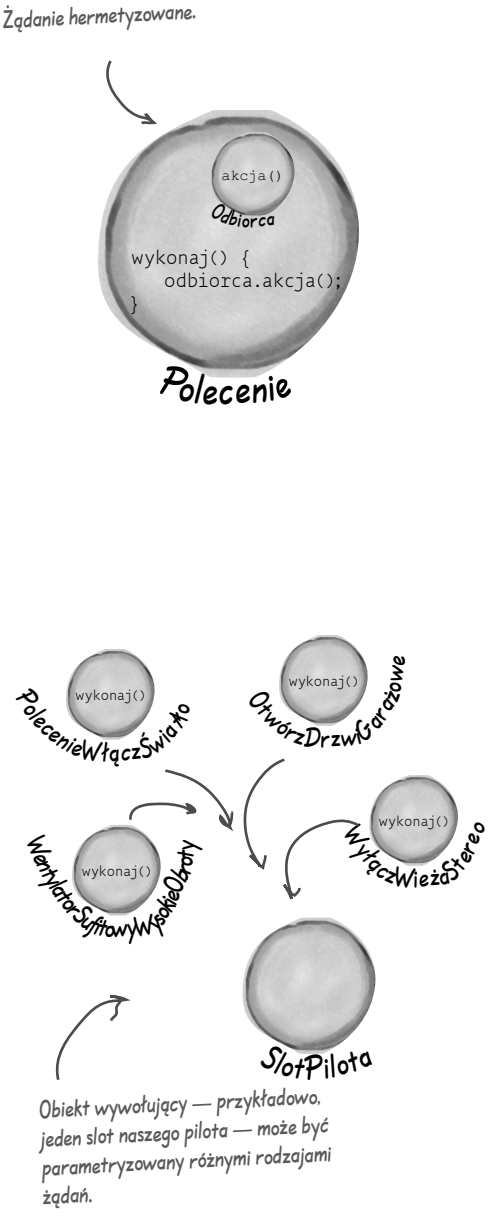
Rozpocznijmy zatem od oficjalnej definicji wzorca:

> **Wzorzec Polecenie** hermetyzuje żądania w postaci obiektów, co umożliwia parametryzowanie różnych obiektów zróżnicowanymi żądaniami (takimi jak np. żądania kolejkowania lub rejestracji) oraz obsługiwanie operacji, które można wycofać.

Spróbujmy przeanalizować tą definicję. Wiemy, że obiekt-polecenie *hermetyzuje żądanie* poprzez wzajemne powiązanie zestawu operacji do wykonania z określonym obiektem, który będzie je realizował. Aby osiągnąć taki rezultat, zestaw operacji i obiekt realizujący są pakowane razem do obiektu, który udostępnia tylko jedną metodę, wykonaj(), która po wywołaniu powoduje, że dany zestaw operacji jest wykonywany przez obiekt realizujący. Żaden inny obiekt znajdujący się na zewnątrz tak naprawdę nie wie, jakie operacje będą wykonywane ani kto jest obiektem realizującym takie operacje; obiekty zewnętrzne wiedzą tylko, że jeżeli wywołają metodę wykonaj(),ich żądanie zostanie zrealizowane.

Widzieliśmy również kilka przykładów *parametryzacji obiektów* różnymi poleceniami. Wracając do naszego baru, pamiętasz zapewne, że Kelnerka była „parametryzowana" rożnymi zamówieniami, które otrzymywała w ciągu dnia. W czasie zabawy z MiniPilotem najpierw do pierwszego (i, jak się wtedy okazało, jedynego) slotu ładowaliśmy polecenie „włączŚwiatło", a następnie zastępowaliśmy je poleceniem „otwórzDrzwiGarażowe". Podobnie jak Kelnerka, również i slot naszego MiniPilota zupełnie nie dbał o to, jaki obiekt-polecenie został do niego uprzednio załadowany — tak długo oczywiście, jak długo obiekt ten posiadał zaimplementowany interfejs Polecenie.

Elementem, z którym do tej pory jeszcze się nie spotkaliśmy, są *żądania kolejkowania i rejestracji, jak również obsługiwanie operacji, które można wycofać*. Ale nie masz się czym przejmować, gdyż są to stosunkowo proste rozszerzenia podstawowego wzorca Polecenie i wkrótce powrócimy do ich omawiania. Kiedy dobrze poznamy zasady funkcjonowania podstawowej wersji wzorca Polecenie, będziemy mogli również bez większych problemów skorzystać z rozszerzenia tego wzorca, znanego pod nazwą Meta Polecenie, który pozwala na tworzenie swego rodzaju makr pozwalających na wykonywanie szeregu poleceń w jednym wywołaniu.

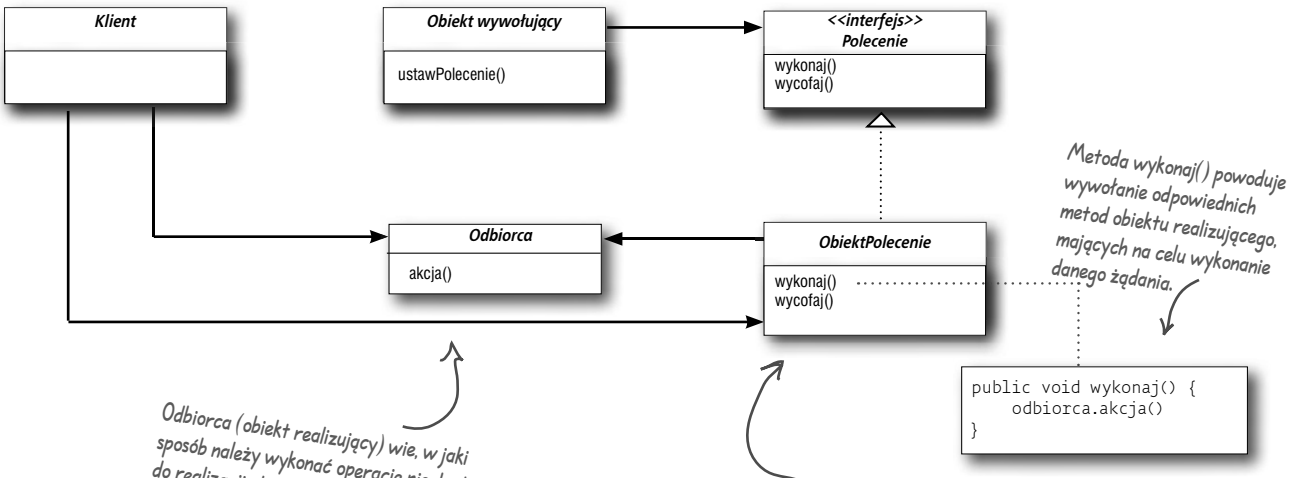


Definicja wzorca Polecenie
— diagram klas

Klient jest odpowiedzialny za stworzenie ObiektuPolecenia i za ustalenie jego Odbiorcy (obiektu realizującego).

Obiekt wywołujący przechowuje obiekt-polecenie i w określonym momencie prosi go o wykonanie żądania poprzez wywołanie jego metody wykonaj().

Polecenie jest interfejsem implementowanym przez wszystkie obiekty-polecenia. Jak już zdążyłeś się dowiedzieć, właściwe polecenie jest wywoływane poprzez jego metodę wykonaj(), która powoduje, że odbiorca (obiekt realizujący) rozpoczyna wykonywanie swojego zadania. Zwróć uwagę, że interfejs ten posiada również metodę wycofaj(), o której opowiemy w nieco dalszej części rozdziału.



Metoda wykonaj() powoduje wywołanie odpowiednich metod obiektu realizującego, mających na celu wykonanie danego żądania.

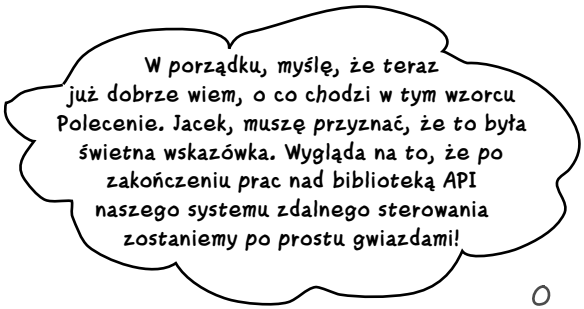
Odbiorca (obiekt realizujący) wie, w jaki sposób należy wykonać operacje niezbędne do realizacji określonego żądania. Rolę odbiorcy może spełniać dowolna klasa.

ObiektPolecenie tworzy powiązanie pomiędzy operacjami, które powinny zostać wykonane, a odbiorcą tych operacji, czyli obiektem, który będzie je realizował. Obiekt wywołujący wystawia żądanie poprzez wywołanie metody wykonaj() obiektu-polecenia, a ten powoduje wykonanie żądania poprzez wywołanie odpowiednich metod obiektu realizującego.



WYTĘŻ UMYSŁ

W jaki sposób wzorzec Polecenie powoduje usunięcie sprzężenia pomiędzy obiektem wywołującym żądanie a obiektem, który je realizuje?





Maria: Mam takie samo wrażenie.
Zatem, w którym miejscu rozpoczynamy?

Sandra: Podobnie, jak to uczyniłyśmy w przypadku MiniPilota, musimy zapewnić sposób przypisywania poleceń do poszczególnych slotów. W naszym przypadku tych slotów jest siedem. Każdy z nich wyposażony jest w osobny zestaw przycisków włącz-wyłącz. Sądzę zatem, że moglibyśmy przypisywać polecenia do pilota w następujący sposób:

```
polecenieWłącz[0] = polecenieWłącz
polecenieWyłącz[0] = polecenieWyłącz
polecenieWłącz[1] = polecenieWłącz
polecenieWyłącz[1] = polecenieWyłącz
polecenieWłącz[2] = polecenieWłącz
polecenieWyłącz[2] = polecenieWyłącz
polecenieWłącz[3] = polecenieWłącz
polecenieWyłącz[3] = polecenieWyłącz
polecenieWłącz[4] = polecenieWłącz
polecenieWyłącz[4] = polecenieWyłącz
polecenieWłącz[5] = polecenieWłącz
polecenieWyłącz[5] = polecenieWyłącz
polecenieWłącz[6] = polecenieWłącz
polecenieWyłącz[6] = polecenieWyłącz
```

Maria: To mogłoby działać, gdyby nie obiekty typu Światło. W jaki sposób pilot ma odróżnić oświetlenie jadalni od oświetlenia kuchni?

Sandra: Aaah, tylko to — otóż wcale nie musi! Pilot nie wie niczego poza tym, w jaki sposób ma wywołać metodę wykonaj() danego obiektu-polecenia po naciśnięciu określonego przycisku.

Maria: Taaak, teraz zaczynam łapać, o co tutaj chodzi! Ale jak w naszej implementacji zapewnimy, aby właściwe obiekty włączały bądź wyłączały właściwe urządzenia?

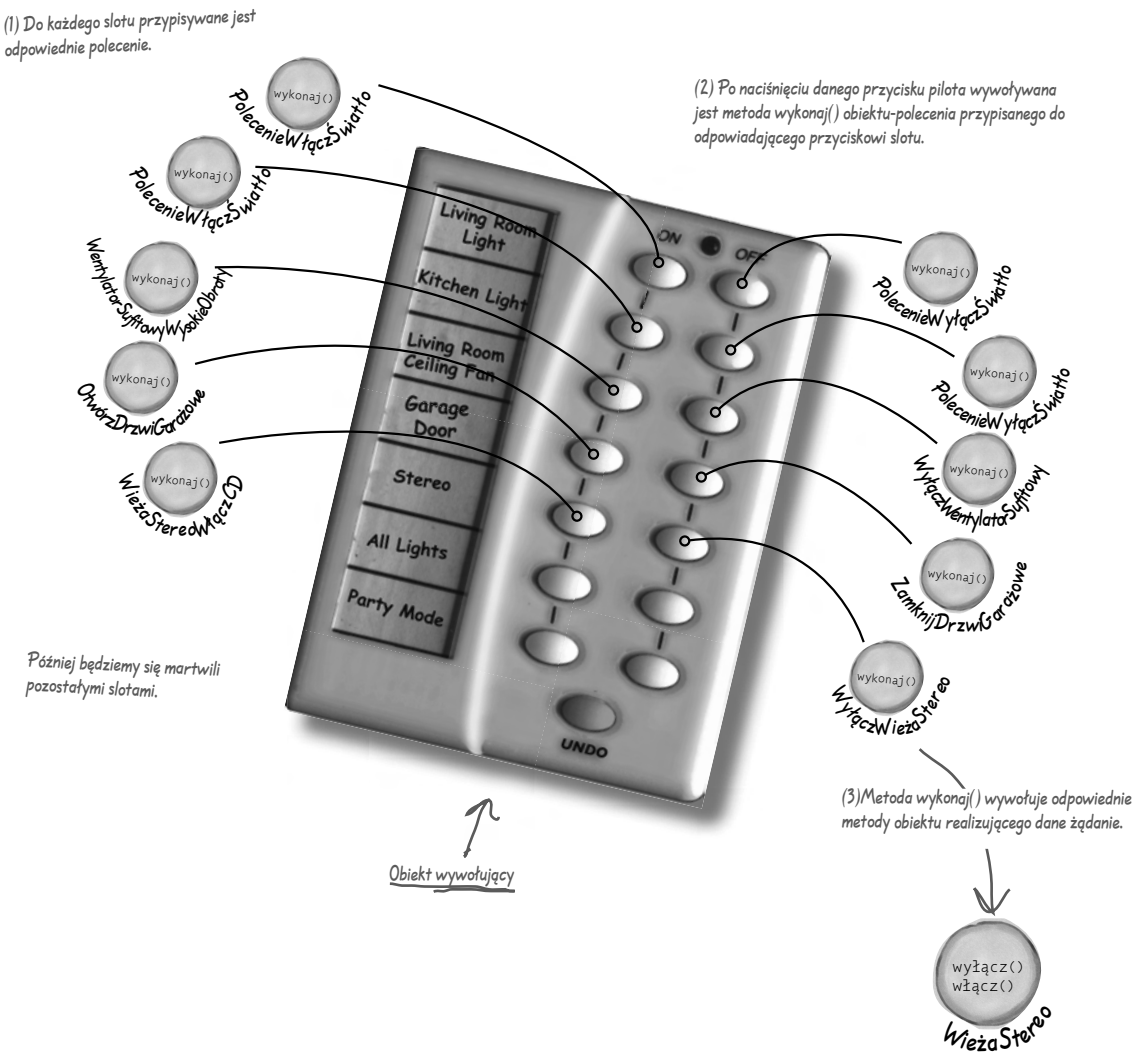
Sandra: Kiedy będziemy tworzyć odpowiednie polecenia, które następnie będą ładowane do naszego pilota, utworzymy jedno polecenie włączŚwiatło powiązane z oświetleniem jadalni i inne polecenie włączŚwiatło powiązane z oświetleniem kuchni. Pamiętaj, odbiorca żądania jest hermetyzowany razem z operacjami, jakie będą wykonane w danym obiekcie-poleceniu. Dzięki temu, kiedy wybrany przycisk pilota zostanie naciśnięty, nikt nie będzie się zastanawiał, które oświetlenie jest które. Po prostu po wywołaniu metody wykonaj() zostanie zrealizowana właściwa operacja.

Maria: Dobra, rozumiem, co jest grane. Zatem bierzmy się za bary z tą implementacją pilota i „w praniu” to wszystko nam się jeszcze z pewnością rozjaśni.

Sandra: Brzmi nieźle. Najpierw spróbujemy narysować schemat tego ustrojstwa…

Przypisywanie POLECEŃ do slotów

A więc mamy taki plan: musimy przypisać do każdego slotu SuperPilota odpowiednie POLECENIE. Dzięki temu SuperPilot zostanie naszym *obiektem wywołującym*. Po naciśnięciu wybranego przycisku pilota zostanie wywołana metoda wykonaj() obiektu przypisanego do slotu odpowiadającego temu przyciskowi, co w efekcie spowoduje wykonanie odpowiednich operacji przez obiekt realizujący (którym może być oświetlenie, wentylator sufitowy, wieża stereo i inne).



Implementujemy SuperPilota

```
public class SuperPilot {
    Polecenie[] poleceniaWłącz;
    Polecenie[] poleceniaWyłącz;

    public SuperPilot() {
        poleceniaWłącz = new Polecenie[7];
        poleceniaWyłącz = new Polecenie[7];

        Polecenie brakPolecenia = new BrakPolecenia();
        for (int i = 0; i < 7; i++) {
            poleceniaWłącz[i] = brakPolecenia;
            poleceniaWyłącz[i] = brakPolecenia;
        }
    }

    public void ustawPolecenie(int slot, Polecenie poleceniaWłącz, Polecenie poleceniaWy-
łącz) {
        poleceniaWłącz[slot] = poleceniaWłącz;
        poleceniaWyłącz[slot] = poleceniaWyłącz;
    }

    public void wciśniętoPrzyciskWłącz(int slot) {
        poleceniaWłącz[slot].wykonaj();
    }

    public void wciśniętoPrzyciskWyłącz(int slot) {
        poleceniaWyłącz[slot].wykonaj();
    }

    public String toString() {
        StringBuffer stringBuff = new StringBuffer();
        stringBuff.append("\n------ SuperPilot -------\n");
        for (int i = 0; i < poleceniaWłącz.length; i++) {
            stringBuff.append("[slot " + i + "] " + poleceniaWłącz[i].getClass().getName()
                + "    " + poleceniaWyłącz[i].getClass().getName() + "\n");
        }
        return stringBuff.toString();
    }
}
```

Od tej pory nasz SuperPilot będzie obsługiwał siedem poleceń typu włącz i wyłącz, które będą przechowywane w odpowiedniej tablicy.

Konstruktor klasy musi jedynie utworzyć obiekty reprezentujące poszczególne tablice i nadać im odpowiednie wartości początkowe.

Metoda ustawPolecenie() pobiera numer slotu oraz odpowiednie polecenia odpowiedzialne za włączanie i wyłączanie danego urządzenia. Następnie polecenia te zostają zapisane w odpowiednich tablicach celem późniejszego wykorzystania.

Kiedy zostaje naciśnięty dowolny przycisk Włącz lub Wyłącz, nasz pilot wywołuje odpowiednio metodę wciśniętoPrzyciskWłącz() lub wciśniętoPrzyciskWyłącz().

Tutaj przesłoniliśmy oryginalną metodę toString. Nasza wersja tej metody pozwala na wyświetlanie na ekranie numerów poszczególnych slotów razem z przypisanymi do nich odpowiednimi poleceniami. Będziemy z tego korzystali podczas testowania naszego nowego SuperPilota.

Implementacja POLECEŃ

Jak zapewne pamiętasz, nie tak dawno paraliśmy się już tworzeniem implementacji klasy PolecenieWłączŚwiatło dla naszego MiniPilota. Teraz możemy dołączyć ten sam kod tutaj i wszystko powinno działać, jak należy. Polecenia odpowiedzialne za wyłączanie poszczególnych urządzeń nie będą się zbytnio różniły; w praktyce definicja klasy PolecenieWyłączŚwiatło wygląda następująco:

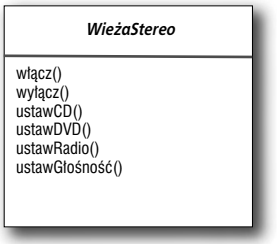
```
public class PolecenieWyłączŚwiatło implements Polecenie {
    Światło światło;

    public PolecenieWyłączŚwiatło(Światło światło) {
        this.światło = światło;
    }

    public void wykonaj() {
        światło.wyłącz();
    }
}
```

Klasa PolecenieWyłączŚwiatło funkcjonuje dokładnie w taki sam sposób, jak klasa PolecenieWłączŚwiatło, z tą tylko różnicą, że obiekt realizujący jest teraz powiązany z inną operacją: metodą wyłącz().

Spróbujmy teraz podjąć nieco większe wyzwanie: może zatem napisać odpowiednie polecenia włączające i wyłączające naszą ulubioną wieżę stereo? OK, wyłączanie wieży jest całkiem łatwe, wystarczy po prostu utworzyć klasę PolecenieWyłączWieżaStereo i powiązać obiekt realizujący (czyli obiekt klasy WieżaStereo) z metodą wyłącz(). Włączanie wieży może być nieco bardziej skomplikowane; powiedzmy, że chcemy utworzyć klasę PolecenieWieżaStereo WłączCD…




```
public class PolecenieWieżaStereoWłączCD implements Pole-
cenie{
    WieżaStereo wieżaStereo;

    public PolecenieWieżaStereoWłączCD (WieżaStereo wie-
żaStereo) {
        this.wieżaStereo = wieżaStereo;
    }

    public void wykonaj() {
        wieżaStereo.włącz();
        wieżaStereo.ustawCD();
        wieżaStereo.ustawGłośność(11);
    }
}
```

Podobnie, jak w przypadku klasy PolecenieWłączŚwiatło, przekazujemy instancję wieży stereo, którą będziemy sterować, i zapisujemy ją w lokalnej zmiennej obiektowej.

Aby zrealizować takie żądanie, musimy wywołać trzy metody klasy WieżaStereo: pierwsza z nich włącza całą wieżę, druga przełącza ją w tryb odtwarzania CD i wreszcie trzecia ustawia poziom siły głosu na wartość 11. Dlaczego właśnie 11? Bo to jest zawsze lepiej (i głośniej!) niż 10, prawda?

Całkiem nieźle. Rzuć teraz okiem na pozostałe klasy sterujące; od tej chwili z całą pewnością jesteś w stanie stworzyć odpowiednie klasy poleceń dla pozostałych urządzeń.

Sprawdzamy możliwości naszego SuperPilota

Nasza praca z oprogramowaniem SuperPilota jest już niemal całkiem na ukończeniu; musimy jeszcze tylko przeprowadzić parę testów i utworzyć odpowiedni zestaw dokumentacji, opisujący poszczególne elementy naszej biblioteki API. Zarząd firmy „Automatyka w domu i zagrodzie" z pewnością będzie pod wielkim wrażeniem, nie sądzisz? Udało nam się stworzyć projekt, dzięki któremu będą mogli produkować łatwe do rozbudowy i modyfikacji systemy zdalnego sterowania urządzeniami gospodarstwa domowego, a co więcej, nie będą mieli żadnych problemów z przekonaniem producentów innych urządzeń do tworzenia własnych klas sterujących w przyszłości, ponieważ są one wyjątkowo proste do utworzenia.

A teraz pora na trochę kodu, który przetestuje naszego SuperPilota!

```
public class ProgramowaniePilota {

    public static void main(String[] args) {
        SuperPilot pilot = new SuperPilot();

        Światło jadalniaŚwiatło = new Światło("Jadalnia");
        Światło kuchniaŚwiatło = new Światło("Kuchnia");
        WentylatorSufitowy wentylatorSufitowy=
                new WentylatorSufitowy("Jadalnia");
        DrzwiGarażowe drzwiGarażowe = new DrzwiGarażowe("");
        WieżaStereo wieżaStereo = new WieżaStereo("Jadalnia");

        PolecenieWłączŚwiatło jadalniaWłączŚwiatło =
                new PolecenieWłączŚwiatło(jadalniaŚwiatło);
        PolecenieWyłączŚwiatło jadalniaWyłączŚwiatło =
                new PolecenieWyłączŚwiatło(jadalniaŚwiatło);
        PolecenieWłączŚwiatło kuchniaWłączŚwiatło =
                new PolecenieWłączŚwiatło(kuchniaŚwiatło);
        PolecenieWyłączŚwiatło kuchniaWyłączŚwiatło=
                new PolecenieWyłączŚwiatło(kuchniaŚwiatło);

        PolecenieWłączWentylatorSufitowy włączWentylatorSufitowy=
               new PolecenieWłączWentylatorSufitowy(wentylatorSufitowy);
        PolecenieWyłączWentylatorSufitowy wyłączWentylatorSufitowy=
               new PolecenieWyłączWentylatorSufitowy(wentylatorSufitowy);

        PolecenieOtwórzDrzwiGarażowe otwórzDrzwiGarażowe =
                new PolecenieOtwórzDrzwiGarażowe(drzwiGarażowe);
        PolecenieZamknijDrzwiGarażowe zamknijDrzwiGarażowe =
                new PolecenieZamknijDrzwiGarażowe(drzwiGarażowe);

        PolecenieWieżaStereoWłączCD wieżaStereoWłączCD =
                new PolecenieWieżaStereoWłączCD(wieżaStereo);
        PolecenieWyłączWieżaStereo  wyłączWieżaStereo =
                new PolecenieWyłączWieżaStereo (wieżaStereo);
```

Tworzy wszystkie obiekty reprezentujące poszczególne urządzenia w odpowiednich miejscach.

Tworzy wszystkie obiekty-polecenia dla oświetlenia.

Tworzy obiekty-polecenia dla włączania i wyłączania wentylatora sufitowego.

Tworzy obiekty-polecenia otwierające i zamykające drzwi garażowe.

Tworzy obiekty-polecenia dla włączania i wyłączania wieży stereo.

```
        pilot.ustawPolecenie(0, jadalniaWłączŚwiatło, jadalniaWyłączŚwiatło);
        pilot.ustawPolecenie(1, kuchniaWłączŚwiatło, kuchniaWyłączŚwiatło);
        pilot.ustawPolecenie(2, włączWentylatorSufitowy, wyłączWentylatorSufitowy);
        pilot.ustawPolecenie(3, wieżaStereoWłączCD, WyłączWieżaStereo);

        System.out.println(pilot);

        pilot.wciśniętoPrzyciskWłącz(0);
        pilot.wciśniętoPrzyciskWyłącz(0);
        pilot.wciśniętoPrzyciskWłącz(1);
        pilot.wciśniętoPrzyciskWyłącz(1);
        pilot.wciśniętoPrzyciskWłącz(2);
        pilot.wciśniętoPrzyciskWyłącz(2);
        pilot.wciśniętoPrzyciskWłącz(3);
        pilot.wciśniętoPrzyciskWyłącz(3);
    }
}
```

Wszystkie obiekty-polecenia są już gotowe, więc możemy załadować je do odpowiednich slotów naszego pilota.

W tym miejscu zastosujemy naszą wersję metody toString() do wyświetlania na ekranie numerów poszczególnych slotów oraz poleceń, które zostały do nich przypisane.

Świetnie, zatem jesteśmy gotowi do zabawy! Przejrzymy teraz wszystkie sloty i będziemy naciskali kolejno ich przyciski Włącz i Wyłącz.

A teraz zobaczymy, jak wyglądają wyniki naszego testu...

```
C:\WINDOWS\System32\cmd.exe

> java ProgramowaniePilota

------ SuperPilot ------
[slot 0] headfirst.polecenie.pilot.PolecenieWłączŚwiatło            headfirst.polecenie.pilot.PolecenieWyłączŚwiatło
[slot 1] headfirst.polecenie.pilot.PolecenieWłączŚwiatło            headfirst.polecenie.pilot.PolecenieWyłączŚwiatło
[slot 2] headfirst.polecenie.pilot.PolecenieWłączWentylatorSufitowy headfirst.polecenie.pilot.PolecenieWyłączWentylatorSufitowy
[slot 3] headfirst.polecenie.pilot.PolecenieWieżaStereoWłączCD      headfirst.polecenie.pilot.PolecenieWyłączWieżaStereo
[slot 4] headfirst.polecenie.pilot.BrakPolecenia                    headfirst.polecenie.pilot.BrakPolecenia
[slot 5] headfirst.polecenie.pilot.BrakPolecenia                    headfirst.polecenie.pilot.BrakPolecenia
[slot 6] headfirst.polecenie.pilot.BrakPolecenia                    headfirst.polecenie.pilot.BrakPolecenia

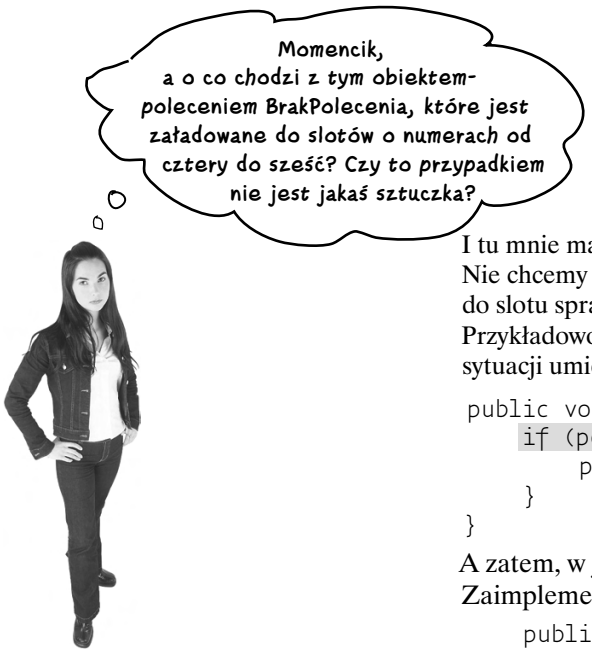
Jadalnia światło włączone
Jadalnia światło wyłączone
Kuchnia światło włączone
Kuchnia światło wyłączone
Jadalnia wentylator sufitowy włączony, wysokie obroty
Jadalnia wentylator sufitowy wyłączony, wysokie obroty
Jadalnia wieża stereo włączona
Jadalnia wieża stereo wybrano odtwarzacz CD
Jadalnia wieża stereo głośność ustawiona na 11
Jadalnia wieża stereo wyłączona

> _
```

Sloty Włącz

Sloty Wyłącz

Nasze obiekty-polecenia w akcji! Pamiętaj, wyświetlane informacje dotyczące poszczególnych urządzeń pochodzą bezpośrednio z klas sterujących. Przykładowo, kiedy odpowiedni obiekt spowoduje włączenie oświetlenia w jadalni, na ekranie pojawi się komunikat: „Jadalnia: światło włączone".



I tu mnie masz. Rzeczywiście, posłużyliśmy się tutaj pewnym wybiegiem. Nie chcemy w naszej aplikacji za każdym razem przed odwołaniem do slotu sprawdzać, czy jakieś polecenie zostało do niego załadowane. Przykładowo, w metodzie wciśniętoPrzyciskWłącz() musielibyśmy w takiej sytuacji umieścić np. następujący, dodatkowy fragment kodu:

```
public void wciśniętoPrzyciskWłącz(int slot) {
    if (poleceniaWłącz[slot] != null) {
        poleceniaWłącz[slot].wykonaj();
    }
}
```

A zatem, w jaki sposób rozwiązaliśmy nasz problem? Zaimplementowaliśmy polecenie, które nic nie robi!

```
public class BrakPolecenia implements Polecenie {
    public void wykonaj() {}
}
```

Następnie w konstruktorze klasy SuperPilot domyślnie ładujemy obiekt BrakPolecenia do każdego slotu i dzięki temu możemy być pewni, że zawsze mamy jakieś polecenia we wszystkich slotach.

```
Polecenie brakPolecenia = new BrakPolecenia();
for (int i = 0; i < 7; i++) {
    poleceniaWłącz[i] = brakPolecenia;
    poleceniaWyłącz[i] = brakPolecenia;
}
```

Dzięki takiemu rozwiązaniu w wynikach działania naszego programu mogłeś zobaczyć sloty, do których nie zostały przypisane żadne inne polecenia poza obiektem BrakPolecenia przypisanym przez konstruktor klasy.



Honorowy Członek Loży Wzorców Projektowych Rusz głową!

Obiekt BrakPolecenia jest znakomitym przykładem *obiektu pustego* (ang. *null object*). Obiekt pusty jest bardzo przydatny w sytuacji, kiedy nie jesteś w stanie zwrócić żadnego innego obiektu, a jednocześnie nie chcesz składać odpowiedzialności za obsługę wartości null na klienta. Przykładowo, w aplikacji obsługującej naszego pilota nie mieliśmy żadnego znaczącego obiektu, który mógłby zostać początkowo przypisany do poszczególnych slotów, więc utworzyliśmy obiekt BrakPolecenia, który odgrywa rolę namiastki obiektu właściwego i którego metoda wykonaj() nie wykonuje po wywołaniu żadnych operacji.

Jak się niebawem przekonasz, obiekty puste są często używane w połączeniu z wzorcami projektowymi, a czasami nawet uda Ci się zobaczyć pozycję Null Object na liście wzorców projektowych.

Nadszedł wreszcie czas, aby utworzyć trochę dokumentacji...

Biblioteka API dla systemu zdalnego sterowania urządzeniami gospodarstwa domowego, przygotowana dla firmy Automatyka w domu i zagrodzie sp. z o.o.

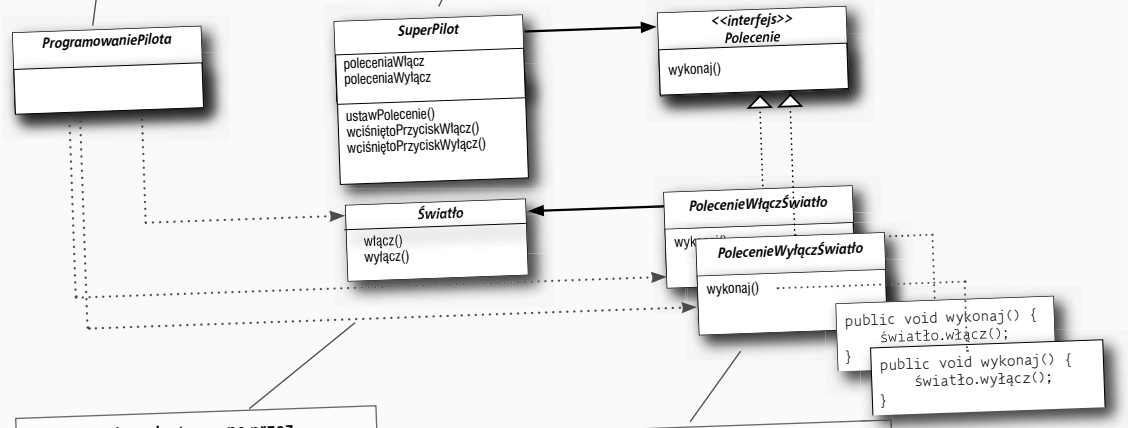
Mamy wielką przyjemność zaprezentować Państwu naszą bibliotekę API przeznaczoną dla Waszego systemu zdalnego sterowania urządzeniami gospodarstwa domowego. Głównym celem przyświecającym nam nad pracami projektowymi było stworzenie maksymalnie prostego i przejrzystego kodu aplikacji — aby nie wymagał on wprowadzania żadnych modyfikacji w miarę dodawania kolejnych klas sterujących nowymi urządzeniami gospodarstwa domowego, które z pewnością niebawem pojawią się na światowych rynkach. Aby tego dokonać, zastosowaliśmy w naszej aplikacji jeden z wzorców projektowych, który pozwala na logiczne usunięcie sprzężenia pomiędzy klasami SuperPilota a klasami sterującymi dostarczanymi przez producentów sprzętu. Wierzymy, że zastosowanie naszego rozwiązania spowoduje znaczącą redukcję kosztów produkcji SuperPilota, jak również przyczyni się do wręcz lawinowej redukcji Waszych przyszłych kosztów konserwacji, aktualizacji i serwisowania tych urządzeń.

Poniżej zamieszczamy diagram klas, który pozwoli Wam lepiej zrozumieć zasady działania naszego projektu:

Klasa ProgramowaniePilota tworzy szereg obiektów-poleceń, które są następnie ładowane do poszczególnych slotów SuperPilota. Każdy obiekt-polecenie hermetyzuje żądanie wykonania określonej operacji przez dane urządzenie gospodarstwa domowego.

Klasa SuperPilot zarządza zestawem obiektów-poleceń, po jednym na każdy przycisk pilota. Kiedy dany przycisk zostanie naciśnięty, wywoływana jest odpowiadająca mu metoda wciśniętoPrzyciskxxx(), która z kolei wywołuje metodę wykonaj() obiektu-polecenia związanego z danym przyciskiem. I to jest cały zakres wiedzy, jaką SuperPilot posiada o wywoływanych klasach; dzieje się tak dlatego, że obiekt-polecenie usuwa sprzężenie pomiędzy SuperPilotem a klasami sterującymi poszczególnymi urządzeniami gospodarstwa domowego.

Wszystkie obiekty-polecenia współpracujące z SuperPilotem posiadają zaimplementowany interfejs Polecenie, który składa się z jednej tylko metody: wykonaj(). Obiekty-polecenia dokonują hermetyzacji danego zestawu operacji, który będzie realizowany przez określoną klasę sterującą. Pilot inicjuje realizację takich operacji (takiego żądania) poprzez wywołanie metody wykonaj().



Klasy sterujące dostarczane przez producentów urządzeń są wykorzystywane do wykonywania określonych operacji bezpośrednio na sterowanych urządzeniach. W naszym przykładzie wykorzystujemy tylko klasę Światło.

Dzięki interfejsowi Polecenie każda operacja, która może zostać wykonana poprzez naciśnięcie odpowiedniego przycisku pilota, może zostać zaimplementowana w postaci prostego obiektu-polecenia. Obiekt taki przechowuje odwołanie do instancji obiektu odpowiedniej klasy sterującej i posiada zaimplementowaną metodę wykonaj(), której wywołanie powoduje wykonanie jednej lub kilku metod na obiekcie klasy sterującej. Na naszym diagramie zamieściliśmy tylko dwie klasy sterujące, które odpowiadają za włączanie i wyłączanie oświetlenia.



Uuuups! Faktycznie, prawie o tym zapomnieliśmy… na szczęście dzięki temu, że mamy już gotowy zestaw podstawowych klas wzorca Polecenie, dołożenie możliwości wycofywania ostatnio wykonanego polecenia jest bardzo proste. Przyjrzyjmy się zatem, jak można dołożyć takich mechanizm do naszego SuperPilota…

Co robimy?

Jak już powiedzieliśmy, naszym zadaniem będzie dołożenie do SuperPilota dodatkowego przycisku dającego możliwość wycofania ostatnio wykonanego polecenia. Taki mechanizm działa następująco: załóżmy, że oświetlenie jadalni jest wyłączone, a Ty naciskasz na pilocie przycisk Włącz. Oczywiście, zgodnie z oczekiwaniami, oświetlenie zostaje włączone. Teraz, kiedy naciśniesz przycisk Wycofaj, działanie ostatnio wykonanego polecenia zostanie odwrócone — czyli w naszym przypadku światło ponownie zgaśnie. Zanim pogrążymy się w bardziej wyrafinowanych przykładach, spróbujmy jednak najpierw uruchomić taki mechanizm dla oświetlenia:

1. Jeżeli poszczególne polecenia mają obsługiwać możliwość wycofywania ostatnio poczynionych operacji, muszą posiadać zaimplementowaną metodę wycofaj(), która będzie swego rodzaju lustrzanym odbiciem metody wykonaj(). Niezależnie od tego, jaką operację spowodowało wywołanie metody wykonaj(),wywołanie metody wycofaj() zawsze przyniesie skutek odwrotny. Zanim jednak będziemy mogli dodać metodę wycofaj() do naszych obiektów, musimy ją dołożyć do interfejsu Polecenie:

```
public interface Polecenie {
    public void wykonaj();
    public void wycofaj();
}
```

Oto nasza nowa metoda: wycofaj().

Ta część zadania była, jak widać, bardzo prosta.

A teraz zagłębimy się w kodzie klasy Światło i spróbujemy zaimplementować tam naszą nową metodę wycofaj().

❷ Rozpocznijmy od klasy PolecenieWłączŚwiatło: po wywołaniu metody wykonaj() tej klasy ostatnio wykonaną operacją będzie wywołanie metody włącz(). Wiemy zatem, że zadaniem metody wycofaj() będzie wykonanie operacji odwrotnej, czyli inaczej mówiąc, wywołanie metody wyłącz().

```
public class PolecenieWłączŚwiatło implements Polecenie {
    Światło światło;

    public PolecenieWłączŚwiatło(Światło światło) {
        this. światło = światło;
    }

    public void wykonaj() {
        światło.włącz();
    }
    public void wycofaj() {
        światło.wyłącz();
    }
}
```

Metoda wykonaj() powoduje włączenie oświetlenia, więc metoda wycofaj() po prostu je wyłącza.

Bułka z masłem! Teraz zabieramy się za klasę PolecenieWyłączŚwiatło. W tym przypadku metoda wycofaj() musi po prostu (jak się łatwo domyśleć) wywołać metodę włącz(), która spowoduje włączenie oświetlenia.

```
public class PolecenieWyłączŚwiatło implements Polecenie {
    Światło światło;

    public PolecenieWyłączŚwiatło(Światło światło) {
        this. światło = światło;
    }

    public void wykonaj() {
        światło.wyłącz();
    }
    public void wycofaj() {
        światło.włącz();
    }
}
```

A tutaj metoda wycofaj() powoduje ponowne włączenie oświetlenia.

Czy można sobie wyobrazić coś prostszego? No dobrze, jeszcze nie skończyliśmy; musimy włożyć nieco dodatkowej pracy w to, aby nasz SuperPilot był w stanie śledzić ostatnio wciśnięty przycisk oraz aby akceptował i odpowiednio obsługiwał naciśnięcia przycisku Wycofaj.

❸ Aby dołożyć obsługę przycisku Wycofaj, musimy dokonać w klasie SuperPilot tylko niewielu drobnych modyfikacji. Oto, jak mamy zamiar tego dokonać: dodamy nową zmienną obiektową, której zadaniem będzie śledzenie ostatnio wywoływanych poleceń; następnie za każdym razem, kiedy naciśnięty zostanie przycisk Wycofaj, będziemy odzyskiwali takie polecenie i wywoływali jego metodę wycofaj().

```
public class SuperPilotZWycofywaniem {
    Polecenie[] poleceniaWłącz;
    Polecenie[] poleceniaWyłącz;
    Polecenie polecenieWycofaj;

    public SuperPilotZWycofywaniem () {
        poleceniaWłącz = new Polecenie[7];
        poleceniaWyłącz = new Polecenie[7];

        Polecenie brakPolecenia = new BrakPolecenia();
        for (int i = 0; i < 7; i++) {
            poleceniaWłącz[i] = brakPolecenia;
            poleceniaWyłącz[i] = brakPolecenia;
        }
        polecenieWycofaj = brakPolecenia;
    }

    public void ustawPolecenie(int slot, Polecenie polecenieWłącz,
      Polecenie polecenieWyłącz) {
        poleceniaWłącz[slot] = polecenieWłącz;
        poleceniaWyłącz[slot] = polecenieWyłącz;
    }

    public void wciśniętoPrzyciskWłącz(int slot) {
        poleceniaWłącz[slot].wykonaj();
        polecenieWycofaj = poleceniaWłącz[slot];
    }

    public void wciśniętoPrzyciskWyłącz(int slot) {
        poleceniaWyłącz[slot].wykonaj();
        polecenieWycofaj = poleceniaWyłącz[slot];
    }

    public void wciśniętoPrzyciskWycofaj() {
        polecenieWycofaj.wycofaj();
    }

    public String toString() {
        // tutaj powinien się znajdować kod metody toString()...
    }
}
```

To jest zmienna, w której będziemy przechowywać informację o ostatnio wykonanym poleceniu, przeznaczoną dla przycisku Wycofaj.

Podobnie jak w przypadku pozostałych slotów, również i do Wycofaj początkowo przypisywany jest obiekt BrakPolecenia, dzięki czemu naciskanie przycisku Wycofaj przed wykonaniem jakiejkolwiek operacji nie będzie dawało żadnego efektu.

Po naciśnięciu przycisku najpierw wykonywane jest polecenie załadowane do slotu odpowiadającego danemu przyciskowi, a następnie w zmiennej obiektowej polecenieWycofaj zapisywane jest odwołanie do wykonanego polecenia. Taką operację wykonujemy zarówno dla poleceń związanych z przyciskami Włącz, jak i z przyciskami Wyłącz.

Kiedy zostanie naciśnięty przycisk Wycofaj, wywołujemy metodę wycofaj() polecenia zapisanego w zmiennej obiektowej polecenieWycofaj, co powoduje wykonanie operacji przeciwnej do operacji ostatnio wykonanej.

Sprawdzamy poprawność działania przycisku Wycofaj

Aby mieć możliwość przetestowania, czy przycisk Wycofaj działa zgodnie z przewidywaniami, musimy nieco zmodyfikować nasze środowisko testowe:

```
public class ProgramowaniePilota {

    public static void main(String[] args) {
        SuperPilotZWycofywaniem pilot = new SuperPilotZWycofywaniem();

        Światło jadalniaŚwiatło = new Światło("Jadalnia");

        PolecenieWłączŚwiatło jadalniaWłączŚwiatło =
                new PolecenieWłączŚwiatło(jadalniaŚwiatło);
        PolecenieWyłączŚwiatło jadalniaWyłączŚwiatło =
                new PolecenieWyłączŚwiatło(jadalniaŚwiatło);

        pilot.ustawPolecenie(0, jadalniaWłączŚwiatło, jadalniaWyłączŚwiatło);

        pilot.wciśniętoPrzyciskWłącz(0);
        pilot.wciśniętoPrzyciskWyłącz(0);
        System.out.println(pilot);
        pilot.wciśniętoPrzyciskWycofaj();
        pilot.wciśniętoPrzyciskWyłącz(0);
        pilot.wciśniętoPrzyciskWłącz(0);
        System.out.println(pilot);
        pilot.wciśniętoPrzyciskWycofaj();
    }
}
```

Tworzy obiekt reprezentujący oświetlenie oraz nowe obiekty-polecenia odpowiednio włączające i wyłączające światło i obsługujące mechanizm wycofywania.

Ładuje obiekty-polecenia sterujące oświetleniem do slotu numer 1 naszego pilota.

Włączamy światło, następnie je wyłączamy i wciskamy przycisk Wycofaj.

Teraz wyłączamy światło, następnie je włączamy i ponownie naciskamy przycisk Wycofaj.

A oto rezultaty naszego testu…

```
C:\WINDOWS\System32\cmd.exe

> java ProgramowaniePilota
Światło włączone
Światło wyłączone

------ SuperPilot ------
[slot 0] headfirst.polecenie.undo.PolecenieWłączŚwiatło    headfirst.polecenie.undo.PolecenieWyłączŚwiatło
[slot 1] headfirst.polecenie.undo.BrakPolecenia            headfirst.polecenie.undo.BrakPolecenia
[slot 2] headfirst.polecenie.undo.BrakPolecenia            headfirst.polecenie.undo.BrakPolecenia
[slot 3] headfirst.polecenie.undo.BrakPolecenia            headfirst.polecenie.undo.BrakPolecenia
[slot 4] headfirst.polecenie.undo.BrakPolecenia            headfirst.polecenie.undo.BrakPolecenia
[slot 5] headfirst.polecenie.undo.BrakPolecenia            headfirst.polecenie.undo.BrakPolecenia
[slot 6] headfirst.polecenie.undo.BrakPolecenia            headfirst.polecenie.undo.BrakPolecenia
[undo] headfirst.polecenie.undo.PolecenieWyłączŚwiatło

Światło włączone
Światło wyłączone
Światło włączone

------ SuperPilot ------
[slot 0] headfirst.polecenie.undo.PolecenieWłączŚwiatło    headfirst.polecenie.undo.PolecenieWyłączŚwiatło
[slot 1] headfirst.polecenie.undo.BrakPolecenia            headfirst.polecenie.undo.BrakPolecenia
[slot 2] headfirst.polecenie.undo.BrakPolecenia            headfirst.polecenie.undo.BrakPolecenia
[slot 3] headfirst.polecenie.undo.BrakPolecenia            headfirst.polecenie.undo.BrakPolecenia
[slot 4] headfirst.polecenie.undo.BrakPolecenia            headfirst.polecenie.undo.BrakPolecenia
[slot 5] headfirst.polecenie.undo.BrakPolecenia            headfirst.polecenie.undo.BrakPolecenia
[slot 6] headfirst.polecenie.undo.BrakPolecenia            headfirst.polecenie.undo.BrakPolecenia
[undo] headfirst.polecenie.undo.PolecenieWłączŚwiatło

Światło wyłączone
>
```

Najpierw włączamy oświetlenie jadalni, następnie je wyłączamy.

Tutaj mamy polecenia sterujące oświetleniem jadalni.

Naciśnięty został przycisk Wycofaj… metoda wycofaj() obiektu PolecenieWyłączŚwiatło powoduje ponowne włączenie oświetlenia jadalni.

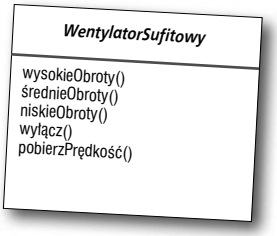
Teraz slot [wycofaj] przechowuje polecenie PolecenieWyłączŚwiatło, ponieważ ono było wykonane jako ostatnie.

Następnie wyłączamy oświetlenie i ponownie je włączamy.

Naciśnięty został przycisk Wycofaj i oświetlenie zostaje wyłączone.

Teraz slot [wycofaj] przechowuje polecenie PolecenieWłączŚwiatło, ponieważ ono było wykonane jako ostatnie.

Implementacja mechanizmu wycofywania przy użyciu stanów



Zapewne sam już doszedłeś do wniosku, że implementacja mechanizmu wycofywania poleceń na przykładzie oświetlenia, choć niewątpliwie bardzo kształcąca, była jednak zadaniem nieco zbyt prostym. Zazwyczaj do implementacji tego mechanizmu musimy wykorzystać przechowywanie stanu obiektów. Spróbujmy zatem zająć się nieco bardziej interesującym przypadkiem, jak np. klasa sterująca WentylatorSufitowy. Wentylator sufitowy oprócz metody, która go całkowicie wyłącza, dysponuje metodami pozwalającymi na jego pracę z trzema różnymi szybkościami.

Poniżej zamieszczamy kod źródłowy klasy WentylatorSufitowy:

```
public class WentylatorSufitowy {
    public static final int SZYBKO = 3;
    public static final int ŚREDNIO = 2;
    public static final int WOLNO = 1;
    public static final int WYŁĄCZ = 0;
    String lokalizacja;
    int prędkość;

    public WentylatorSufitowy(String lokalizacja) {
        this.lokalizacja = lokalizacja;
        prędkość = WYŁĄCZ;
    }

    public void wysokieObroty() {
        prędkość = SZYBKO;
        System.out.println(lokalizacja +
       " wentylator sufitowy włączony, wysokie obroty ");
        // kod przełączający wentylator na wysokie obroty
    }

    public void średnieObroty() {
        prędkość = ŚREDNIO;
        System.out.println(lokalizacja +
       " wentylator sufitowy włączony, średnie obroty ");
        // kod przełączający wentylator na średnie obroty
    }

    public void niskieObroty() {
        prędkość = WOLNO;
        System.out.println(lokalizacja +
        " wentylator sufitowy włączony, niskie obroty ");
        // kod przełączający wentylator na niskie obroty
    }

    public void wyłącz() {
        prędkość = WYŁĄCZ;
        System.out.println(lokalizacja +
         " wentylator sufitowy wyłączony ");
        // kod powodujący wyłączenie wentylatora
    }

    public int pobierzPrędkość() {
        return prędkość;
    }
}
```

Zwróć uwagę, że klasa WentylatorSufitowy posiada lokalny stan reprezentujący prędkość obrotów wentylatora.

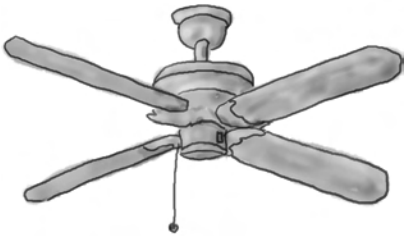
Te metody ustawiają szybkość obrotów wentylatora.

Wywołując metodę pobierzPrędkość(), możemy otrzymać informacje o aktualnej szybkości obrotów wentylatora.

Hmmm, aby we właściwy sposób zaimplementować mechanizm wycofywania, muszę wziąć pod uwagę poprzednio ustawioną szybkość obrotów wentylatora.



Dodajemy mechanizm wycofywania do poleceń sterujących wentylatorem



Spróbujemy teraz dołączyć mechanizm wycofywania do poleceń sterujących naszym wentylatorem sufitowym. Aby tego dokonać, musimy śledzić ostatnio ustawioną szybkość obrotów wentylatora — jeżeli zostanie wywołana metoda wycofaj(), musimy ustawić wentylator właśnie na taką szybkość. Oto kod klasy PolecenieWentylatorSufitowySzybko:

```
public class PolecenieWentylatorSufitowySzybko implements Polecenie{
    WentylatorSufitowy wentylatorSufitowy;
    int poprzedniaSzybkość;

    public PolecenieWentylatorSufitowySzybko (WentylatorSufitowy wenty-
latorSufitowy) {
        this.wentylatorSufitowy = wentylatorSufitowy;
    }

    public void wykonaj() {
        poprzedniaSzybkość = wentylatorSufitowy.pobierzSzybkość();
        wentylatorSufitowy.wysokieObroty();
    }

    public void wycofaj() {
        if (poprzedniaSzybkość == WentylatorSufitowy.SZYBKO) {
            wentylatorSufitowy.wysokieObroty();
        } else if (poprzedniaSzybkość == WentylatorSufitowy.ŚREDNIO) {
            wentylatorSufitowy.średnieObroty();
        } else if (poprzedniaSzybkość == WentylatorSufitowy.WOLNO) {
            wentylatorSufitowy.niskieObroty();
        } else if (poprzedniaSzybkość == WentylatorSufitowy.WYŁĄCZ) {
            wentylatorSufitowy.wyłącz();
        }
    }
}
```

Dodaliśmy lokalny stan obiektu, czyli zmienną, która będzie przechowywała informacje o poprzedniej szybkości obrotów wentylatora.

Zanim metoda wykonaj() zmieni szybkość obrotów wentylatora, musi najpierw zapamiętać jego poprzedni stan, na wypadek, gdyby konieczne okazało się wycofanie takiej operacji.

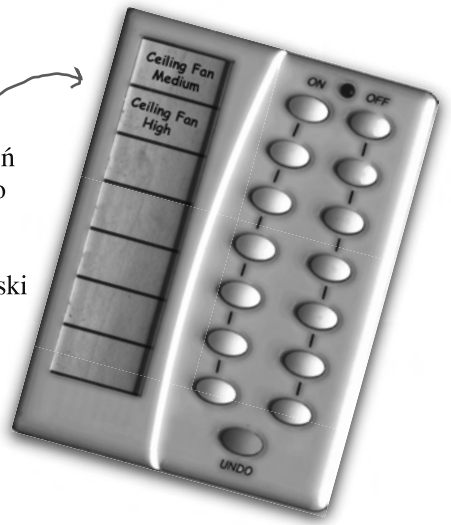
Wywołanie metody wycofaj() powoduje ustawienie szybkości obrotów wentylatora na poprzednią wartość.



WYTĘŻ UMYSŁ

Do napisania pozostały nam jeszcze trzy polecenia sterujące wentylatorem: ustawienie średnich obrotów, ustawienie niskich obrotów oraz całkowite wyłączenie wentylatora. Czy wiesz, jak należy zaimplementować odpowiednie polecenia?

Przygotowania do testowania wentylatora sufitowego



Nadszedł już czas na załadowanie do naszego SuperPilota poleceń sterujących wentylatorem sufitowym. Mamy zamiar załadować do slotu zerowego (przycisk Włącz) polecenie ustawienia średnich obrotów, a do slotu pierwszego (przycisk Włącz) polecenie ustawienia wysokich obrotów wentylatora. Dla obu slotów przyciski Wyłącz będą powodowały całkowite wyłączenie wentylatora.

Oto kod źródłowy naszego środowiska testowego:

```
public class ProgramowaniePilota {

    public static void main(String[] args) {
        SuperPilotZWycofywaniem pilot = new SuperPilotZWycofywaniem ();


        WentylatorSufitowy wentylatorSufitowy = new WentylatorSufitowy("Jadalnia:");

        PolecenieWentylatorSufitowyŚrednio wentylatorSufitowyŚrednio=
                new PolecenieWentylatorSufitowyŚrednio(wentylatorSufitowy);
        PolecenieWentylatorSufitowySzybko wentylatorSufitowySzybko =
                new PolecenieWentylatorSufitowySzybko(wentylatorSufitowy);
        PolecenieWyłączWentylatorSufitowy wyłączWentylatorSufitowy =
                new PolecenieWyłączWentylatorSufitowy (wentylatorSufitowy);

        pilot.ustawPolecenie(0, wentylatorSufitowyŚrednio, wyłączWentylatorSufitowy);
        pilot.ustawPolecenie(1, wentylatorSufitowySzybko, wyłączWentylatorSufitowy);

        pilot.wciśniętoPrzyciskWłącz(0);
        pilot.wciśniętoPrzyciskWyłącz(0);
        System.out.println(pilot);
        pilot.wciśniętoPrzyciskWycofaj();

        pilot.wciśniętoPrzyciskWłącz(1);
        System.out.println(pilot);
        pilot.wciśniętoPrzyciskWycofaj();
    }
}
```

Tutaj do slotu zerowego ładujemy polecenie ustawiające średnie obroty, a do slotu pierwszego wysokie obroty. Do obu slotów ładujemy również polecenie wyłączające wentylator.

Tutaj tworzymy instancje trzech obiektów reprezentujących następujące polecenia: szybkie obroty, średnie obroty i wyłączenie wentylatora.

Najpierw włączamy średnie obroty wentylatora.

Następnie całkowicie go wyłączamy.

Wycofaj! Ponownie powinny włączyć się średnie obroty wentylatora.

Teraz włączamy wysokie obroty wentylatora.

I jeszcze raz wycofujemy polecenie; ponownie włączają się średnie obroty wentylatora.

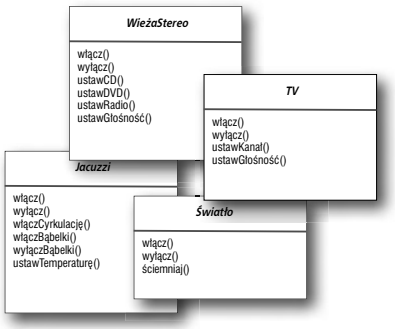
Testujemy wentylator sufitowy...

OK, zatem odpalamy naszego pilota, ładujemy do niego odpowiednie polecenia i zaczynamy zabawę z przyciskami!



Każdy pilot powinien posiadać tryb Impreza!

Jaki jest sens posiadania pilota, jeżeli nie będziesz mógł poprzez naciśnięcie jednego przycisku przyciemnić oświetlenia, włączyć wieży stereo i telewizora, przełączyć całego zestawu na DVD i przygotować wspaniałej kąpieli w jacuzzi?





Hmmm, nasz pilot będzie potrzebował osobnego przycisku do obsługi każdego z tych urządzeń. Nie sądzę, abyśmy mogli tego dokonać.

Sandra, wstrzymaj się z takim osądem przez chwilę. Mimo wszystko myślę, że możemy tego dokonać bez żadnej modyfikacji naszego pilota!



Pomysł Marii polega na tym, aby utworzyć nowy rodzaj Polecenia, które będzie w stanie wykonywać inne Polecenia… i to więcej niż jedno w sekwencji! Świetny pomysł, prawda?

```
public class MakroPolecenie implements Polecenie {
    Polecenie[] polecenia;

    public MakroPolecenie (Polecenie[] polecenia) {
        this.polecenia = polecenia;
    }

    public void wykonaj() {
        for (int i = 0; i < polecenia.length; i++) {
            polecenia[i].wykonaj();
        }
    }
}
```

Pobieramy tablicę Poleceń i przechowujemy ją w obiekcie MakroPolecenie.

Kiedy pilot wywoła nasze makropolecenie, ten szereg poleceń zostanie wykonany jedno po drugim.

Zastosowanie makropoleceń

Zobaczymy teraz krok po kroku, jak funkcjonują nasze makropolecenia:

1. Najpierw tworzymy zestaw poleceń, które chcemy umieścić w makropoleceniu:

```
Światło światło = new Światło("Salon:");
TV tv = new TV("Salon:");
WieżaStereo wieżastereo = new WieżaStereo("Salon:");
Jacuzzi jacuzzi = new Jacuzzi();

PolecenieWłączŚwiatło włączŚwiatło = new PolecenieWłączŚwiatło(światło);
PolecenieWłączWieżaStereo włączWieżaStereo =
          new PolecenieWłączWieżaStereo (wieżastereo);
PolecenieWłączTV włączTV = new PolecenieWłączTV(tv);
PolecenieWłączJacuzzi włączJacuzzi = new PolecenieWłączJacuzzi(jacuzzi);
```

Tworzymy obiekty reprezentujące poszczególne urządzenia: światło, TV, wieżę stereo oraz jacuzzi.

Teraz tworzymy wszystkie polecenia, które będą włączały poszczególne urządzenia.



Zaostrz ołówek

Będziemy również potrzebować wszystkich poleceń, które będą wyłączały poszczególne urządzenia; wpisz poniżej niezbędny fragment kodu:

2. Teraz tworzymy dwie tablice, jedną na polecenia typu Włącz i drugą na polecenia typu Wyłącz, a następnie wypełniamy je odpowiednimi sekwencjami poleceń:

```
Polecenie[] włączImprezę = { włączŚwiatło, włączWieżaStereo, włączTV,
włączJacuzzi};
Polecenie[] wyłączImprezę = { wyłączŚwiatło, wyłączWieżaStereo,
wyłączTV, wyłączJacuzzi};

MakroPolecenie makroWłączImprezę = new MakroPolecenie(włączImprezę);
MakroPolecenie makroWyłączImprezę = new MakroPolecenie(wyłączImprezę);
```

Tworzymy dwie tablice, jedną na polecenia typu Włącz i drugą na polecenia typu Wyłącz…

…i tworzymy dwa odpowiednie makropolecenia, które będą je przechowywać.

3. Następnie przypisujemy MakroPolecenia do wybranych przycisków, tak jak to robiliśmy do tej pory:

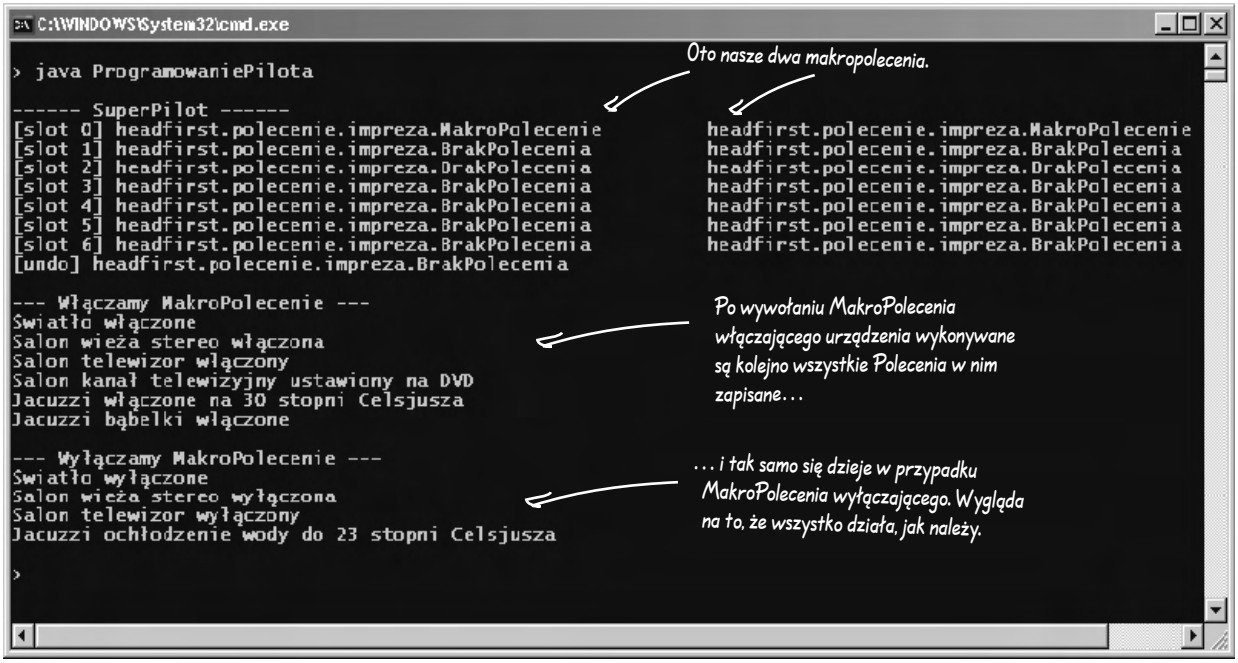
```
pilot.ustawPolecenie(0, makroWłączImprezę, makroWyłączImprezę);
```

Przypisujemy makropolecenia do przycisków tak, jak to robiliśmy do tej pory.

4 Wreszcie, musimy nacisnąć kilka przycisków i zobaczyć, czy wszystko działa tak, jak się tego spodziewaliśmy.

```
System.out.println(pilot);
System.out.println("--- Włączamy MakroPolecenie ---");
pilot.wciśniętoPrzyciskWłącz(0);
System.out.println("--- Wyłączamy MakroPolecenie ---");
pilot.wciśniętoPrzyciskWyłącz(0);
```

Oto wyniki działania naszego programu.





Jedynym elementem, jakiego nie posiadają nasze makropolecenia, jest mechanizm wycofywania poleceń. Kiedy po wykonaniu makropolecenia zostanie naciśnięty przycisk Wycofaj, wszystkie operacje wywołane przez to makropolecenie powinny powrócić do poprzedniego stanu. Poniżej zamieściliśmy kod klasy MakroPolecenie; Twoim zadaniem jest zaimplementowanie odpowiedniej metody wycofaj():

```
public class MakroPolecenie implements Polecenie {
    Polecenie[] polecenia;

    public MakroPolecenie (Polecenie[] polecenia) {
        this.polecenia = polecenia;
    }

    public void wykonaj() {
        for (int i = 0; i < polecenia.length; i++) {
            polecenia[i].wykonaj();
        }
    }
    public void wycofaj() {

        }
}
```

P: Czy zawsze będę potrzebował obiektu realizującego (odbiorcy)? Dlaczego cały kod niezbędny do realizacji metody wykonaj() nie mógłby zostać zaimplementowany wewnątrz obiektu-polecenia?

O: Ogólnie rzecz biorąc, powinniśmy się starać tworzyć proste obiekty-polecenia, których jedynym zadaniem jest wywołanie odpowiednich metod obiektu realizującego; z drugiej jednak strony, zapewne znajdziesz całe mnóstwo przykładów „inteligentnych" obiektów-poleceń, które implementują większość — jeżeli już nawet nie całość — logiki niezbędnej do realizacji danego żądania. Oczywiście, możesz postępować w taki sposób; pamiętaj jednak, że wtedy nie będziesz w stanie osiągnąć takiego stopnia oddzielenia obiektu wywołującego od obiektu realizującego, jak również nie będziesz miał możliwości parametryzacji obiektów-poleceń zróżnicowanymi żądaniami.

Nie istnieją głupie pytania

P: W jaki sposób mogę zaimplementować mechanizm zapamiętywania historii wycofywanych poleceń? Innymi słowy, chciałbym mieć możliwość wielokrotnego korzystania z przycisku Wycofaj.

O: Świetne pytanie! W zasadzie jest to całkiem proste; otóż zamiast przechowywać odwołanie tylko do ostatnio wykonanego Polecenia, powinieneś przechowywać na stosie odwołania do kolejno wykonywanych operacji. Dzięki temu za każdym razem, kiedy zostanie naciśnięty przycisk Wycofaj, obiekt wywołujący będzie zdejmował ze stosu pierwsze z góry Polecenie i wywoływał jego metodę wycofaj().

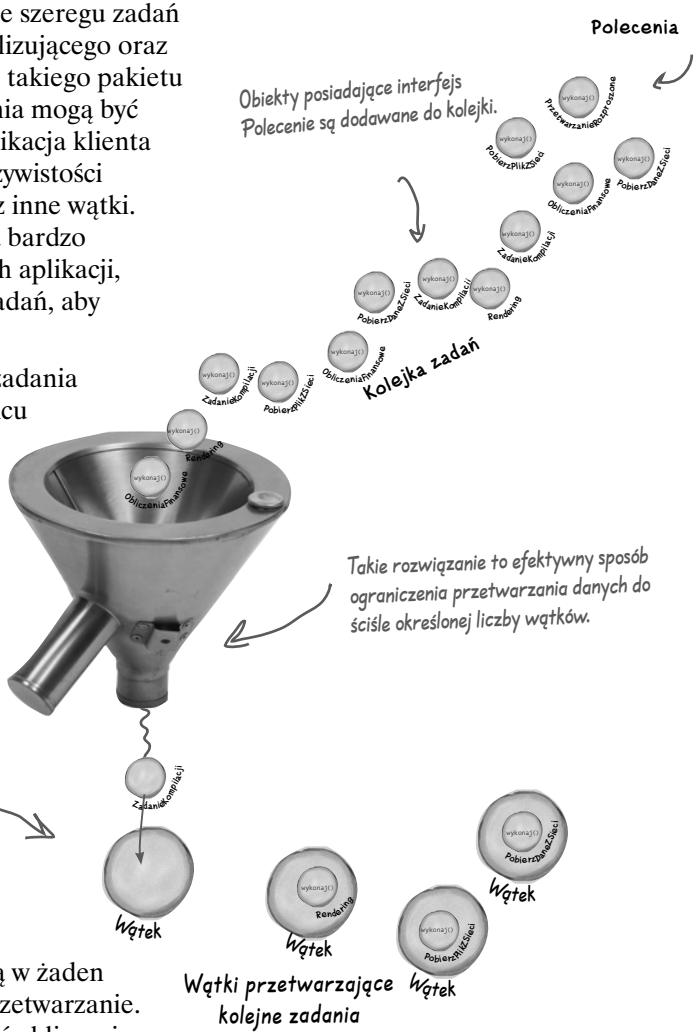
P: Czy przypadkiem nie można by zaimplementować „trybu Impreza" jako osobnego Polecenia poprzez utworzenie odpowiedniej klasy PoleceńeImpreza i umieszczenie wywołania innych Poleceń w jej metodzie wykonaj()?

O: A pewnie, że można, ale postępując w ten sposób musiałbyś „na sztywno" zakodować w klasie PoleceńeImpreza operacje niezbędne do rozruszania imprezy. Po co ładować się w tyle kłopotów? Korzystając z MakroPoleceń, możesz dynamicznie decydować, jakie polecenia będą się składały na uruchomienie kolejnej imprezy. Dzięki temu uzyskujesz o wiele bardziej elastyczne podejście do całego zagadnienia. Ogólnie mówiąc, zastosowanie MakroPoleceń jest o wiele bardziej eleganckim rozwiązaniem, a dodatkowo jego zastosowanie wymaga utworzenia mniejszej ilości nowego kodu.

Kolejne zastosowania wzorca Polecenie — kolejkowanie żądań

Obiekty-polecenia pozwalają nam na opakowanie szeregu zadań (z których każde składa się z danego obiektu realizującego oraz zestawu operacji do wykonania) i przekazywanie takiego pakietu jako normalnego obiektu. Co więcej, takie zadania mogą być wywoływane (realizowane) długo po tym, jak aplikacja klienta utworzyła odpowiedni obiekt-polecenie. W rzeczywistości takie zadania mogą być nawet wywoływane przez inne wątki. Jak łatwo się zorientować, taki scenariusz można bardzo szybko zaadaptować do wielu bardzo użytecznych aplikacji, takich jak terminarze, pule wątków czy kolejki zadań, aby wymienić tylko kilka z nich.

Wyobraź sobie kolejkę zadań: dodajesz kolejne zadania na koniec kolejki, podczas gdy na jej drugim końcu „siedzi" grupa realizujących je wątków. Poszczególne wątki działają według takiego skryptu: zdejmują kolejne zadanie z kolejki, wywołują jego metodę wykonaj(), oczekują na zakończenie działania tej metody, a następnie usuwają dane zadanie i pobierają nowe z kolejki.



Zwróć uwagę, że klasy obsługujące kolejkę nie są w żaden sposób sprzężone z obiektami, które realizują przetwarzanie. W danej chwili określony wątek może wykonywać obliczenia finansowe, a za chwilę pobierać dane z sieci. Z punktu widzenia obiektów obsługujących kolejkę to wszystko jedno; po prostu pobierają z niej kolejne Polecenie i wywołują jego metodę wykonaj(). Analogicznie jest w Twoim przypadku — tak długo, jak długo umieszczasz kolejne obiekty w kolejce implementującej wzorzec Polecenie, metody wykonaj() poszczególnych Poleceń będą wywoływane, kiedy kolejny wątek będzie dostępny.

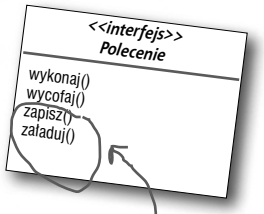


WYTĘŻ UMYSŁ

W jaki sposób serwer WWW może skorzystać z mechanizmu kolejek? Jakie inne aplikacje przychodzą Ci również do głowy?

Kolejne zastosowania wzorca Polecenie — żądania rejestracji

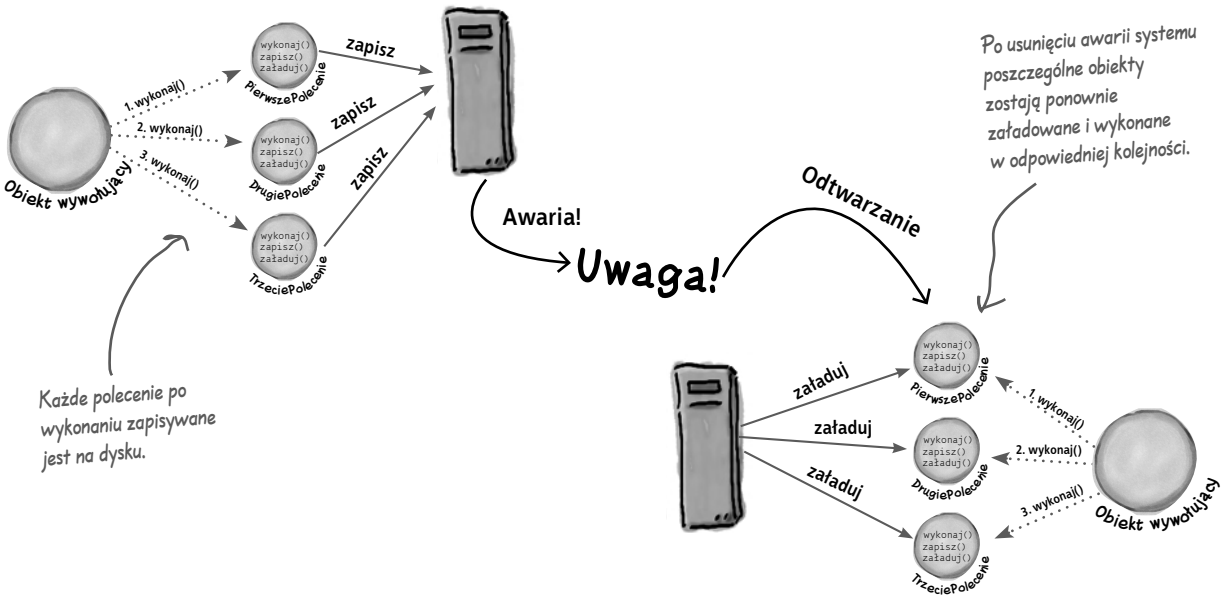
Specyfika niektórych aplikacji wymaga, aby wszystkie wykonywane operacje były rejestrowane tak, aby w razie awarii systemu po jej usunięciu można było odtworzyć stan aplikacji na podstawie rejestru wykonanych wcześniej operacji. Wzorzec Polecenie obsługuje tworzenie takich aplikacji poprzez implementację dwóch dodatkowych metod: zapisz() i załaduj(). W języku Java do implementacji tych metod moglibyśmy wykorzystać serializację obiektów, aczkolwiek spotkalibyśmy się tutaj z pewnymi uporczywymi ograniczeniami tego procesu.

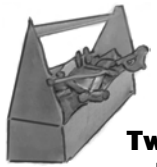


Dodajemy tutaj dwie metody do rejestracji zdarzeń.

Jak to zatem działa? W miarę, jak wykonujemy kolejne operacje, zapisujemy historię ich wykonania na dysku twardym komputera. Kiedy wydarzy się awaria, załadujemy z dysku niezbędne obiekty-polecenia i wywołamy w odpowiedniej kolejności ich metody wykonaj().

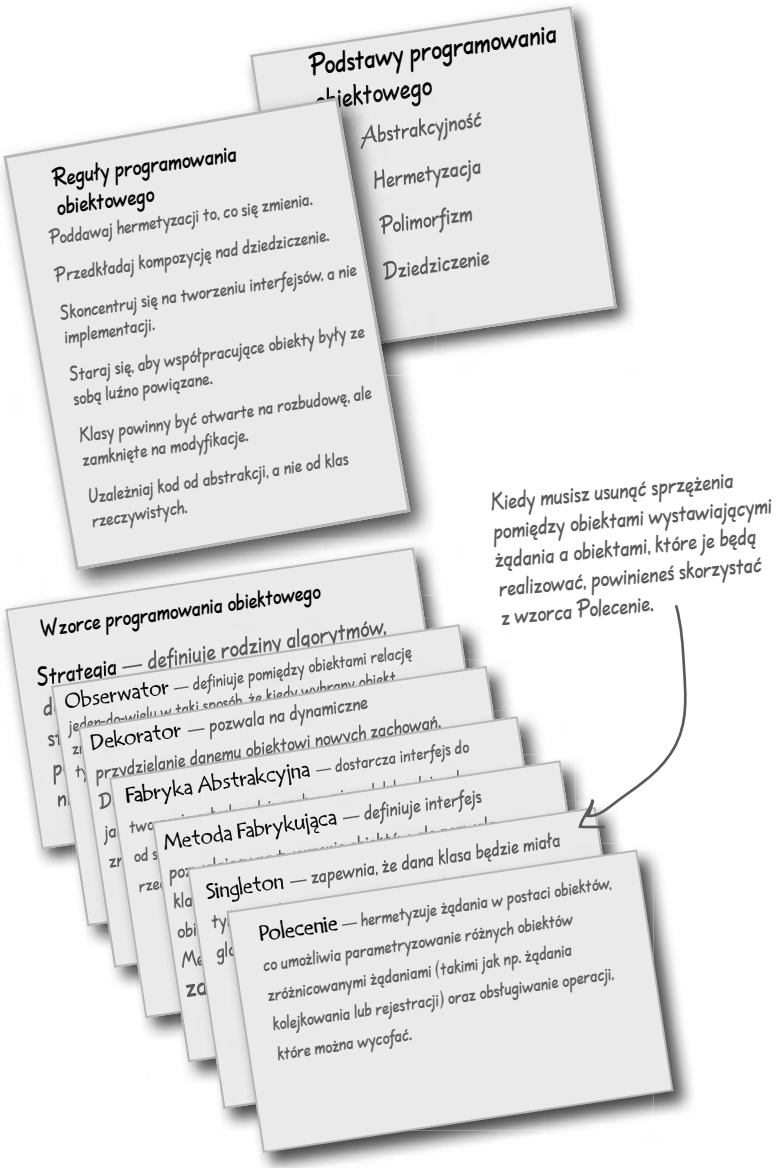
Oczywiście, w naszym przypadku, czyli do pilota systemu zdalnego sterowania urządzeniami gospodarstwa domowego zastosowanie takiego modelu rejestracji wykonywanych operacji nie ma najmniejszego sensu; z drugiej strony zaś, istnieje cały szereg aplikacji wykonujących różnego rodzaju operacje na ogromnych strukturach danych, które nie mogą być łatwo i szybko zapisywane po wykonaniu każdej, nawet najmniejszej zmiany. Dzięki zastosowaniu rejestracji zdarzeń możemy zapisywać listę wszystkich operacji, jakie zostały wykonane od ostatniego punktu kontrolnego. W przypadku zaistnienia awarii systemu (a właściwie tuż po jej usunięciu) możemy ponownie wykonać zapisany ciąg operacji od ostatniego punktu kontrolnego, przywracając tym samym stan aplikacji do momentu tuż przed awarią. Weźmy na przykład arkusz kalkulacyjny: zamiast zapisywana pełnej kopii arkusza po dokonaniu każdej zmiany jego zawartości z pewnością znacznie korzystniej będzie zapisywać tylko historię wykonywanych operacji. W bardziej zaawansowanych aplikacjach takie techniki mogą być rozciągnięte do wykonywania zestawu operacji transakcyjnych tak, że albo wszystkie operacje zostaną poprawnie wykonane i potwierdzone, albo nie zostanie zachowana żadna z nich.





Twoja skrzynka narzędziowa

Twoja skrzynka narzędziowa zaczyna się pomału robić ciężka! W tym rozdziale dołożyliśmy do niej wzorzec, który pozwala na hermetyzację metod w postaci obiektów-poleceń: dzięki temu możesz je przechowywać, przekazywać do innych obiektów i wywoływać, kiedy tylko będzie Ci to potrzebne.



CELNE SPOSTRZEŻENIA



- Wzorzec Polecenie usuwa sprzężenie pomiędzy obiektem wystawiającym żądanie a obiektem, który wie, jak należy je zrealizować.
- Obiekt-polecenie jest kluczowym elementem usuwającym takie sprzężenie i jednocześnie obiektem, który hermetyzuje odbiorcę (obiekt realizujący) z operacją lub szeregiem operacji, które mają być zrealizowane.
- Obiekt wywołujący inicjuje realizację żądania poprzez wywołanie metody wykonaj() obiektu-polecenia, co powoduje rozpoczęcie realizacji poszczególnych operacji przez odbiorcę żądania.
- Obiekty wywołujące mogą być parametryzowane Poleceniami, nawet dynamicznie, podczas działania programu.
- Polecenia mogą obsługiwać mechanizm wycofywania, który powoduje przywrócenie obiektu do stanu sprzed wywołania metody wykonaj().
- MakroPolecenia są prostym rozszerzeniem Poleceń, pozwalającym na wywołanie sekwencji kolejno dwóch lub więcej Poleceń. Podobnie jak Polecenia, również one w pełni mogą obsługiwać mechanizm wycofywania.
- W rozwiązaniach praktycznych często spotyka się „inteligentne" obiekty-polecenia, które zamiast delegować wykonanie żądania do obiektu realizującego, same w sobie implementują wszystkie niezbędne operacje.
- Obiekty-polecenia mogą być również wykorzystywane do implementacji mechanizmów rejestracji, kolejkowania i obsługi systemów transakcyjnych.

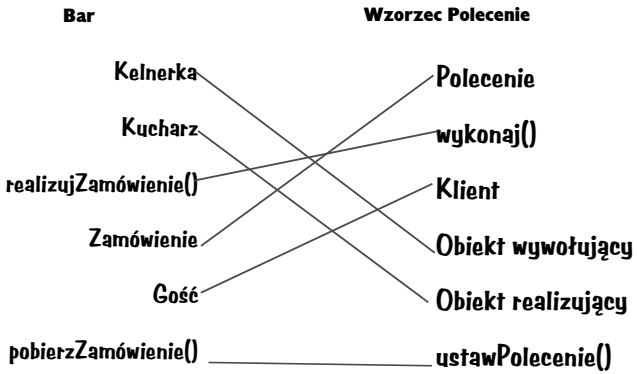


Rozwiązania ćwiczeń

Kto co robi

Kto co robi?

Połącz poszczególne obiekty i metody barowe z odpowiadającymi im elementami wzorca Polecenie.

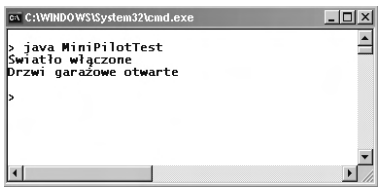




Zaostrz ołówek

```
public class PolecenieOtwórzDrzwiGarażowe implements Polecenie {
    DrzwiGarażowe drzwiGarażowe;
    public PolecenieOtwórzDrzwiGarażowe(DrzwiGarażowe drzwiGarażowe) {
        this.drzwiGarażowe = drzwiGarażowe;
    }

    public void wykonaj() {
        drzwiGarażowe.doGóry();
    }
}
```





Rozwiązania ćwiczeń



Twoim zadaniem jest zaimplementowanie odpowiedniej metody wycofaj():

```
public class MakroPolecenie implements Polecenie {
    Polecenie[] polecenia;

    public MakroPolecenie (Polecenie[] polecenia) {
        this.polecenia = polecenia;
    }

    public void wykonaj() {
        for (int i = 0; i < polecenia.length; i++) {
            polecenia[i].wykonaj();
        }
    }
    public void wycofaj() {
   for (int i = polecenia.length-1; i >= 0; i--) {
polecenia[i].wycofaj();
   }
    }
}
```



Zaostrz ołówek

Będziemy również potrzebować wszystkich poleceń, które będą wyłączały poszczególne urządzenia; wpisz poniżej niezbędny fragment kodu:

```
PolecenieWyłączŚwiatło wyłączŚwiatło = new PolecenieWyłączŚwia-
tło(światło);
PolecenieWyłączWieżaStereo WyłączWieżaStereo =
          new PolecenieWyłączWieżaStereo (wieżastereo);
PolecenieWyłączTV wyłączTV = new PolecenieWyłączTV(tv);
PolecenieWyłączJacuzzi wyłączJacuzzi = new PolecenieWyłączJacuz-
zi(jacuzzi);
```

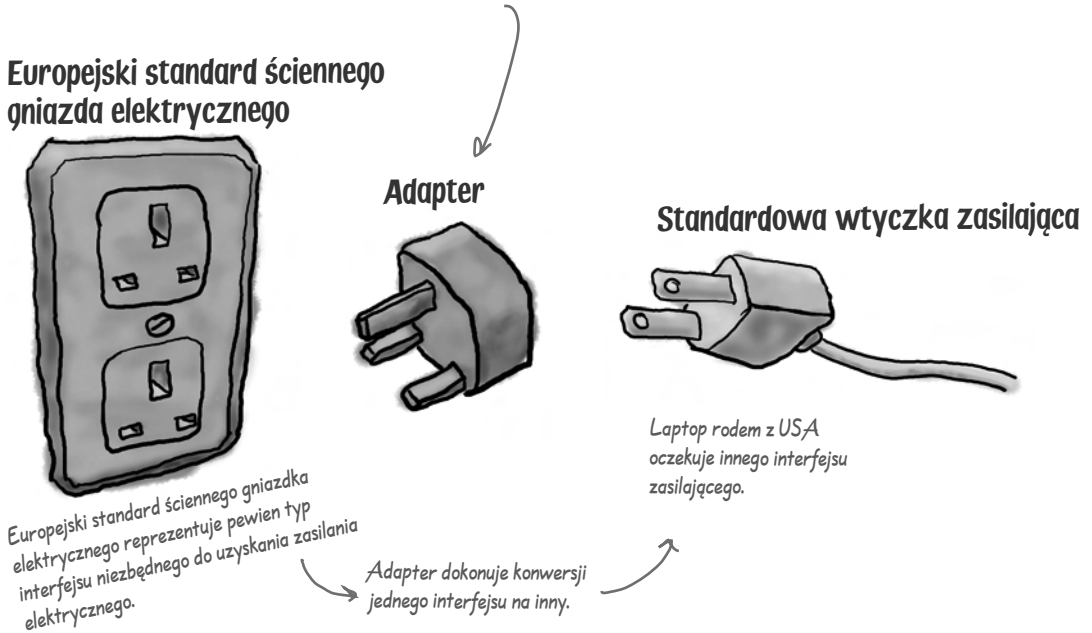
Zdolność do adaptacji



W niniejszym rozdziale mamy zamiar dokonać paru niesamowitych wyczynów z dziedziny rzeczy niemożliwych, takich jak na przykład włożenie kwadratowego kołka do okrągłego otworu. Brzmi nierealnie? Nie wtedy, kiedy mamy pod ręką odpowiednie wzorce projektowe. Pamiętasz wzorzec Dekorator? Podczas pracy z nim owijaliśmy obiekty innymi obiektami tak, aby nadać im nowe zachowania. Teraz mamy zamiar postępować tak samo, ale w nieco innym celu: chcemy sprawić, by ich interfejsy wyglądały jak coś, czym nie są. Dlaczego jednak mielibyśmy to robić? Na przykład po to, aby zaadaptować projekt oczekujący danego interfejsu do klasy, która implementuje zupełnie inny interfejs. To jeszcze nie wszystko; skoro już jesteśmy przy tym temacie, przyjrzymy się również innemu wzorcowi, który opakowuje obiekty w celu uproszczenia ich interfejsów.

Adaptery są wśród nas

Nie powinieneś mieć żadnych problemów ze zrozumieniem roli adapterów w programowaniu obiektowym, ponieważ realny świat dookoła nas także jest ich pełen. Oto przykład: czy kiedykolwiek zdarzyło Ci się podróżować z laptopem wyprodukowanym w USA do krajów w Europie? Jeżeli tak, zapewne potrzebny Ci był odpowiedni adapter do zasilacza...



Wiesz, co robi adapter zasilania: siedzi sobie pomiędzy wtyczką Twojego laptopa a gniazdkiem w ścianie, a jego zadaniem jest adaptacja europejskiego standardu gniazd zasilających tak, abyś bez problemu mógł podłączyć do nich swojego laptopa i otrzymać zasilanie. Możesz również spojrzeć na cały problem nieco inaczej: adapter zasilania zmienia interfejs gniazda zasilającego na taki, którego oczekuje Twój laptop.

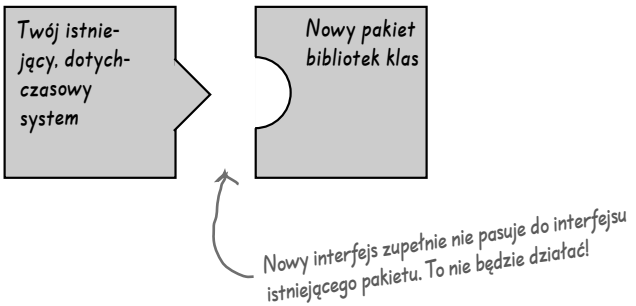
Jakie inne przykłady adapterów spotykanych na co dzień w Twoim otoczeniu przychodzą Ci jeszcze do głowy?

Niektóre adaptery zasilania są bardzo proste — ich zadaniem jest tylko i wyłącznie zmiana kształtu gniazda tak, aby pasowało do Twojej wtyczki, podczas gdy połączenia elektryczne pozostają bez zmian — ale oczywiście istnieją również inne, bardziej skomplikowane adaptery, posiadające całkiem sporo wbudowanej elektroniki mającej na celu zmniejszenie bądź zwiększenie napięcia zasilającego tak, aby dopasować je do indywidualnych wymagań danego urządzenia odbiorczego.

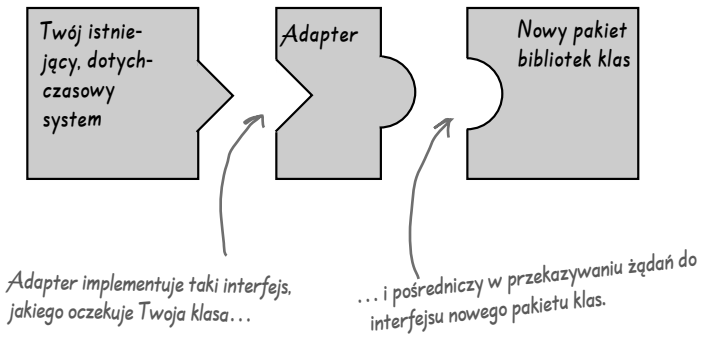
OK, tak to wygląda w świecie rzeczywistym, ale jak można zastosować podobne sztuczki do adapterów zorientowanych obiektowo? No cóż, krótko mówiąc, adaptery zorientowane obiektowo spełniają dokładnie taką samą rolę, jak ich współplemieńcy ze świata rzeczywistego: biorą interfejs danego obiektu i zmieniają go tak, aby dopasować go do wymagań danego klienta.

Adaptery zorientowane obiektowo

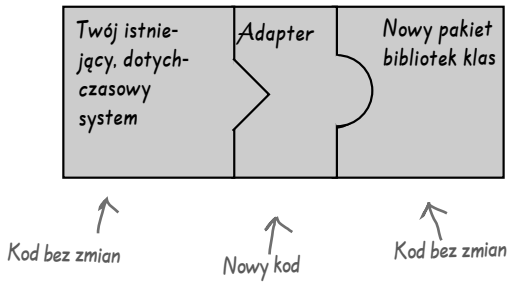
Załóżmy, że masz gotowy, istniejący pakiet oprogramowania, który musi współpracować z biblioteką klas nowego producenta, ale, niestety, programiści tej nowej firmy zaprojektowali zupełnie inny interfejs niż ich poprzednicy:



Na pewno nie chcesz rozwiązywać tego problemu poprzez modyfikację istniejącego kodu (a nie możesz zmieniać kodu pakietu klas dostarczanych przez tę nową firmę). Co zatem można począć w takiej trudnej sytuacji? No cóż, nie pozostaje Ci nic innego, jak tylko napisać klasę, której zadaniem będzie zaadaptować interfejs nowych klas do takiej postaci, jakiej oczekujesz.



Adapter spełnia rolę pośrednika, otrzymującego żądania od klienta i dokonującego ich konwersji na żądania, które mają sens dla klas po jego drugiej stronie.



Czy możesz wyobrazić sobie rozwiązanie, które nie będzie od CIEBIE wymagało napisania ŻADNEGO dodatkowego kodu, a pozwoli na integrację nowej biblioteki klas? Może spróbujemy zrobić tak, aby nowe klasy współpracowały z klasami naszego adaptera?

Jeżeli to coś chodzi jak kaczka i kwacze jak kaczka, ~~musi być~~ może być ~~kaczką~~ indykiem zapakowanym w adapter udający kaczkę...



Najwyższy czas zobaczyć jakiś adapter w akcji. Pamiętasz nasze kaczki z rozdziału 1.? Przyjrzyjmy się zatem nieco uproszczonej wersji „kaczych" interfejsów i klas:

```
public interface Kaczka {
    public void kwacz();
    public void lataj();
}
```

Od tej pory wszystkie nasze kaczki będą implementowały interfejs Kaczka, który będzie im pozwalał na kwakanie i latanie.

Oto klasa DzikaKaczka, czyli klasa podrzędna superklasy Kaczka.

```
public class DzikaKaczka implements Kaczka {
    public void kwacz() {
        System.out.println("Kwa! Kwa!");
    }

    public void lataj() {
        System.out.println("O rany! Latam!");
    }
}
```

Prosta implementacja: kaczka będzie wyświetlała na ekranie nazwę wykonywanej czynności.

A teraz nadszedł czas, aby poznać najnowszy nabytek w naszym stadku drobiu:

```
public interface Indyk {
    public void gulgocz();
    public void lataj();
}
```

Indyki nie kwaczą, ale gulgoczą.

Indyki mogą latać, aczkolwiek zazwyczaj latają tylko na bardzo krótkich dystansach.

```
public class DzikiIndyk implements Indyk {
    public void gulgocz() {
        System.out.println("Gulgulgul!");
    }

    public void lataj() {
        System.out.println("O rany! Latam!... ale tylko na krótkich dystansach!");
    }
}
```

Tutaj mamy przykład rzeczywistej implementacji Indyka; podobnie jak Kaczka, Indyk wyświetla na ekranie nazwy wykonywanych czynności.

Teraz powiedzmy, że zaczyna nam nieco brakować obiektów typu Kaczka i chcesz użyć zamiast nich obiektów typu Indyk (których masz w nadmiarze). Oczywiście, nie możemy użyć takich obiektów bezpośrednio, ponieważ posiadają zupełnie inny interfejs.

A zatem musimy napisać odpowiedni adapter:



Bliskie spotkania z kodem aplikacji

Najpierw musimy zaimplementować interfejs takiego typu, do jakiego się adaptujemy. To jest właśnie interfejs, jakiego oczekuje Twój klient.

```
public class IndykAdapter implements Kaczka {
    Indyk indyk;

    public IndykAdapter(Indyk indyk) {
        this.indyk = indyk;
    }

    public void kwacz() {
        indyk.gulgocz();
    }

    public void lataj() {
        for(int i=0; i < 5; i++) {
            indyk.lataj();
        }
    }
}
```

Następnie musimy otrzymać odwołanie do obiektu, który adaptujemy; tutaj dokonujemy tego w konstruktorze klasy.

musimy zaimplementować wszystkie metody interfejsu; jak widać, translacja metody kwacz() pomiędzy klasami jest bardzo prosta, wystarczy wywołać w zamian metodę gulgocz().

Pomimo, że oba interfejsy posiadają metodę lataj(), Indyki mogą latać tylko krótkimi zrywami — zatem w przeciwieństwie do Kaczek nie mogą wykonywać lotów na długich dystansach. Aby mniej więcej dopasować metodę lataj() Kaczki do możliwości Indyka, musimy wywołać metodę lataj() tego ostatniego co najmniej pięć razy pod rząd.

Testujemy adapter dla Indyka

Pozostało nam jeszcze napisanie niezbyt dużej ilości kodu, którego zadaniem będzie przygotowanie odpowiedniego środowiska testowego dla naszego nowego adaptera:

```
public class TestowanieKaczki {
    public static void main(String[] args) {
        DzikaKaczka kaczka = new DzikaKaczka();

        DzikiIndyk indyk = new DzikiIndyk();
        Kaczka indykAdapter = new IndykAdapter(indyk);

        System.out.println("---- Indyk powiada tak...");
        indyk.gulgocz();
        indyk.lataj();

        System.out.println("\n---- Kaczka powiada tak...");
        testujKaczkę(kaczka);

        System.out.println("\n---- A IndykAdapter powiada tak...");
        testujKaczkę(indykAdapter);
    }

    static void testujKaczkę (Kaczka kaczka) {
        kaczka.kwacz();
        kaczka.lataj();
    }
}
```

Utwórzmy Kaczkę...

oraz Indyka.

A następnie zapakujmy indyka w adapter IndykAdapter, który spowoduje, że będzie on wyglądał jak przyzwoita Kaczka.

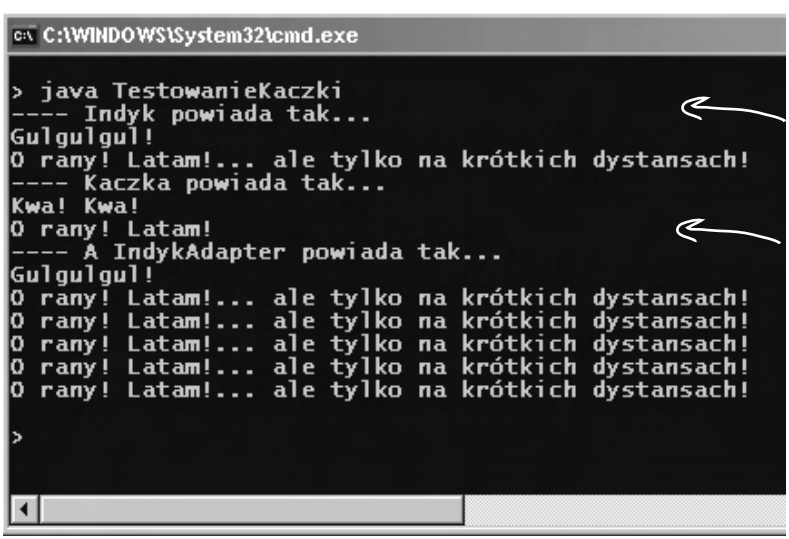
Teraz przetestujmy zachowanie Indyka: najpierw zmusimy go do gulgotania, a potem do latania.

A teraz weźmiemy się za Kaczkę, wywołując metodę testujKaczkę(), która oczekuje obiektu typu Kaczka.

A teraz pora na duży przekręt: metodzie testujKaczkę() spróbujemy teraz wcisnąć Indyka zamiast Kaczki...

Oto nasza metoda testująca Kaczki; pobiera obiekt typu Kaczka i kolejno wywołuje jego metody kwacz() i lataj().

Przebieg testowy



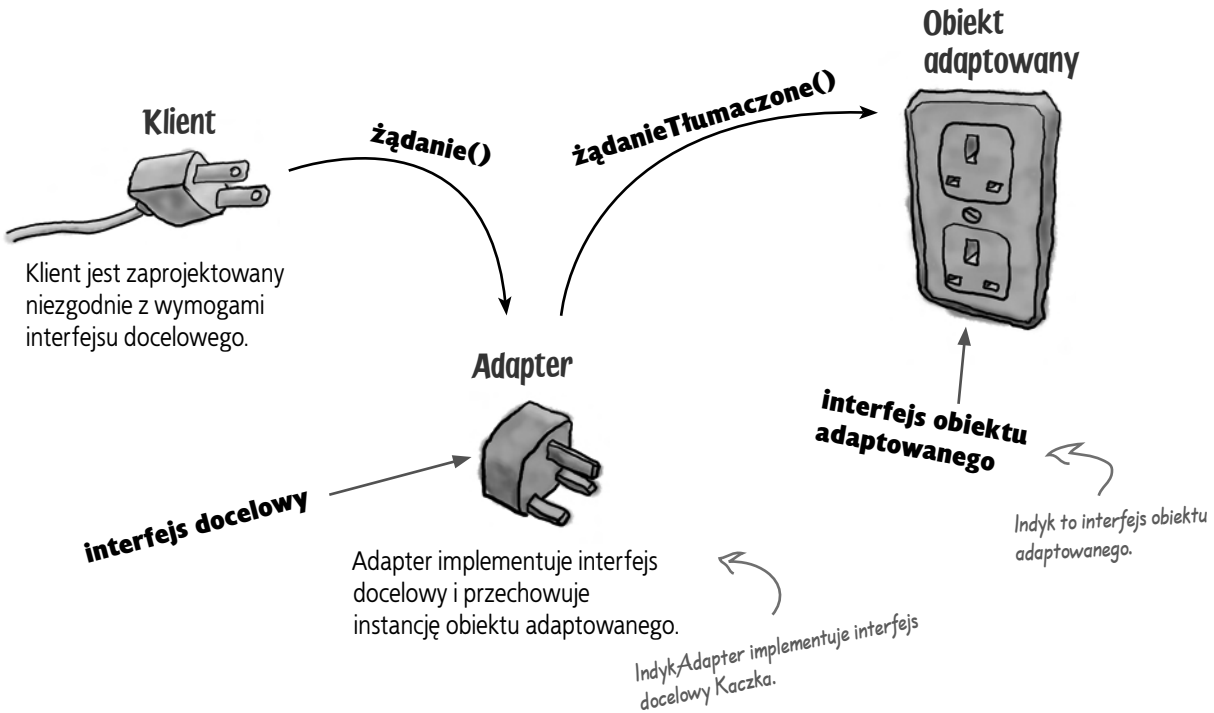
Indyk gulgocze i lata na krótkich dystansach.

Kaczka kwacze i lata tak dobrze, jak mógłbyś tego oczekiwać.

Natomiast adapter gulgocze jak trzeba, kiedy zostanie wywołana metoda kwacz(), i wykonuje po kilka krótkich lotów, kiedy wywołana zostanie metoda lataj(). Metoda testujKaczkę() nigdy nie zorientuje się, że ktoś tak naprawdę podsunął jej Indyka zamiast Kaczki!

Wzorzec Adapter bez tajemnic

Teraz, gdy mamy już pewne pojęcie, czym jest Adapter, cofnijmy się do początków niniejszego rozdziału i rozważmy ponownie problem gniazdka.



Oto, w jaki sposób Klient korzysta z Adaptera:

1. **Klient wystawia żądanie do adaptera poprzez wywołanie metody interfejsu docelowego.**

2. **Adapter dokonuje „przetłumaczenia" żądania na jedno lub więcej wywołań metod obiektu adaptowanego, wykorzystując do wywołań jego interfejs.**

3. **Klient otrzymuje rezultaty swojego żądania i nigdy nie będzie wiedział, że „po drodze" znajdował się adapter dokonujący odpowiednich konwersji.**

Zwróć uwagę, że nie istnieje żadne sprzężenie pomiędzy Klientem a obiektem adaptowanym — obie strony nic o sobie wzajemnie nie wiedzą.



Zaostrz ołówek

Załóżmy, że będzie nam również potrzebny Adapter, który dokonuje konwersji Kaczki na Indyka. Nazwijmy go KaczkaAdapter. Twoim zadaniem będzie napisanie poniżej kodu odpowiedniej klasy:

W jaki sposób rozwiązałeś problem z metodą lataj() (w końcu wszyscy doskonale zdajemy sobie sprawę z tego, że kaczka może latać dużo dalej niż indyk)? Nasze rozwiązanie zamieściliśmy na końcu rozdziału — sprawdź i porównaj ze swoim. Czy wpadłeś na lepszy pomysł?

P: Do jakiego stopnia Adapter powinien być „adaptujący”? Wygląda na to, że jeżeli będzie trzeba zaimplementować mocno rozbudowany interfejs docelowy, mogę mieć z tym paskudnie DUŻO roboty.

O: Pewnie, że możesz. Ilość pracy, jaką musisz włożyć w odpowiednie przygotowanie danego adaptera, jest wprost proporcjonalna do rozmiarów interfejsu docelowego, który musisz obsługiwać. Pomyśl jednak również o innych dostępnych rozwiązaniach tego problemu. Możesz na przykład odpowiednio zmodyfikować fragmenty kodu, które po stronie klienta dokonują wywołań metod interfejsu, aczkolwiek takie rozwiązanie zazwyczaj również wymaga dużej ilości analiz i modyfikacji istniejącego kodu. Innym rozwiązaniem może być na przykład utworzenie nowej klasy, która dokona hermetyzacji wszystkich niezbędnych zmian w jednej klasie.

Nie istnieją głupie pytania

P: Czy Adapter zawsze dokonuje konwersji tylko jednej klasy?

O: Zadaniem wzorca Adapter jest dokonywanie konwersji jednego interfejsu na inny. Pomimo, iż zdecydowana większość spotykanych przykładów zastosowania tego wzorca przedstawia właśnie taki rodzaj konwersji, wszyscy zdajemy sobie sprawę, że w świecie rzeczywistym nie zawsze jest to takie proste. Nigdy nie można wykluczyć sytuacji, w której dany Adapter będzie musiał obsługiwać dwa lub nawet więcej obiektów adaptowanych i dopiero takie rozwiązanie pozwoli na pełną implementację interfejsu docelowego.

Takie rozważania prowadzą nas prosto do wzorca Fasada; zresztą wielu programistów bardzo często myli ze sobą te dwa wzorce. Przypomnij nam, abyśmy wrócili do tego zagadnienia, kiedy będziemy omawiali wzorzec Facade w nieco dalszej części rozdziału.

P: A co mam zrobić, jeżeli w swoim systemie mam zarówno stare, jak i nowe moduły? Stare wymagają posługiwania się poprzednim, starym interfejsem, ale równocześnie pojawiły się już nowe, korzystające z zupełnie innego interfejsu. Myślę, że korzystanie w jednej części aplikacji z odpowiedniego Adaptera, a w innej z bezpośrednich odwołań do interfejsu może powodować spore problemy i nieporozumienia. Czy nie lepiej będzie po prostu przepisać od nowa starą część kodu i zupełnie darować sobie te gadki o Adapterach?

O: Niekoniecznie. Jednym z proponowanych w takiej sytuacji rozwiązań jest utworzenie Adaptera dwukierunkowego, który będzie obsługiwał oba rodzaje interfejsów. Aby utworzyć taki Adapter, wystarczy zaimplementować oba przedmiotowe interfejsy, dzięki czemu nasz Adapter będzie mógł odgrywać rolę zarówno starego, jak i nowego interfejsu.

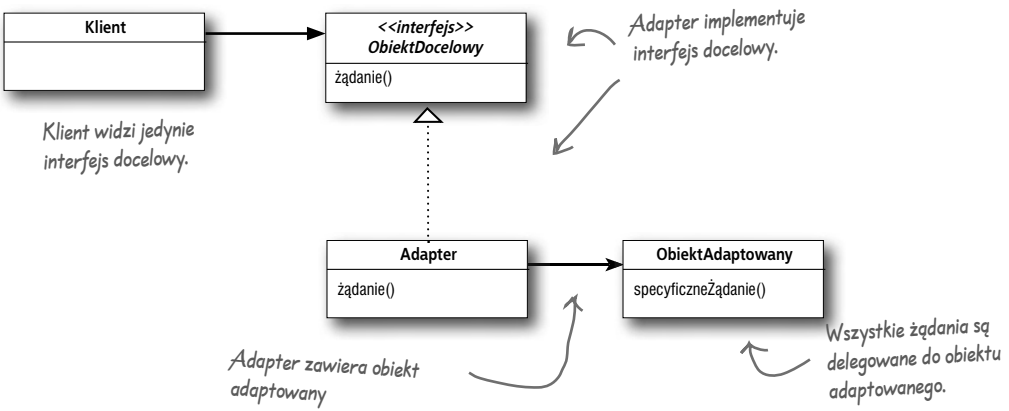
Definicja wzorca Adapter

Wystarczy już tych wszystkich kaczek, indyków i adapterów zasilania; powróćmy zatem do rzeczywistości i przyjrzyjmy się oficjalnej definicji wzorca Adapter:

> **Wzorzec Adapter** dokonuje konwersji interfejsu danej klasy do postaci innego interfejsu, zgodnego z oczekiwaniami klienta. Adapter pozwala na wzajemną współpracę klas, które ze względu na niekompatybilne interfejsy wcześniej nie mogły ze sobą współpracować.

Teraz już doskonale wiemy, że zastosowanie wzorca Adapter pozwala na współpracę klienta z niekompatybilnym interfejsem dzięki stworzeniu adaptera, który będzie dokonywał odpowiedniej konwersji. Rozwiązanie takie pozwala na odseparowanie klienta od implementowanego interfejsu, a jeżeli spodziewamy się, że interfejs będzie się zmieniał w miarę upływu czasu, taka zmiana będzie hermetyzowana wewnątrz adaptera. Dzięki temu kod klienta nie musi być modyfikowany za każdym razem, kiedy będzie współpracował z innym interfejsem.

Widzieliśmy już, co prawda, wzorzec Adapter w akcji, podczas działania programu, ale warto również rzucić okiem na jego diagram klas:



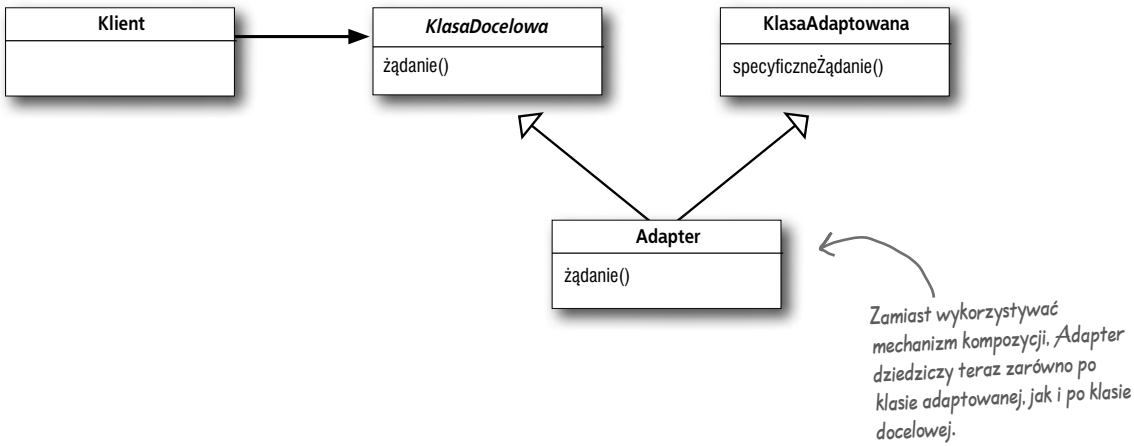
Wzorzec Adapter jest pełen dobrych reguł programowania obiektowego: sprawdź, w jaki sposób zastosowano tu metodę kompozycji do zapakowania obiektu adaptowanego łącznie z modyfikowanym interfejsem. Takie rozwiązanie posiada pewną dodatkową zaletę: adapter może być używany z dowolną podklasą obiektu adaptowanego.

Warto również sprawdzić, w jaki sposób wzorzec dokonuje powiązania klienta z interfejsem, a nie konkretną implementacją; moglibyśmy tutaj użyć kilku różnych adapterów, z których każdy dokonywałby konwersji innego zestawu klas docelowych; równie dobrze moglibyśmy tutaj dołożyć kilka nowych implementacji klas, oczywiście pod warunkiem, że byłyby one zgodne z interfejsem docelowym.

Adaptery obiektów i klas

Niezależnie od tego, że sam wzorzec mamy już zdefiniowany, nie opowiedzieliśmy Ci jeszcze tej historii do samego końca. Tak naprawdę istnieją *dwa* rodzaje adapterów: adaptery *obiektów* i adaptery *klas*. Do tej pory w niniejszym rozdziale mówiliśmy o adapterach obiektów i diagram klas zamieszczony na poprzedniej stronie jest diagramem klas adaptera obiektów.

A zatem, czym jest adapter klas i dlaczego do tej pory o nich nie mówiliśmy? Przyczyna jest bardzo prosta: otóż do ich implementacji niezbędny jest mechanizm wielokrotnego dziedziczenia, który nie jest obsługiwany w języku Java. Ale, oczywiście, nie oznacza to wcale, że nie możesz skorzystać z takiego rozwiązania, kiedy będziesz go potrzebował, tworząc nową aplikację w którymś ze swoich ulubionych języków programowania obsługujących taki mechanizm! Rzućmy zatem okiem na diagram klas wykorzystujący dziedziczenie wielokrotne.



Wygląda znajomo? Zgadza się — jedyna różnica polega na tym, adapter klasy dziedziczy zarówno po klasie adaptowanej, jak i po klasie docelowej, podczas gdy adapter obiektów do przekazywania żądań do obiektu adaptowanego wykorzystywał mechanizm kompozycji.



WYTĘŻ UMYSŁ

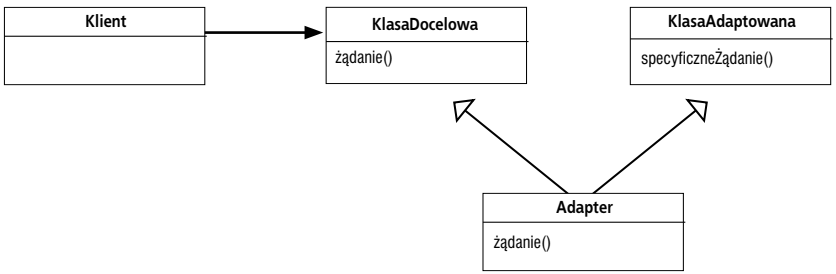
Adaptery obiektów i adaptery klas wykorzystują dwa różne sposoby adaptacji elementu adaptowanego (kompozycja kontra dziedziczenie). Jaki wpływ mają te różnice w implementacji na elastyczność samego adaptera?



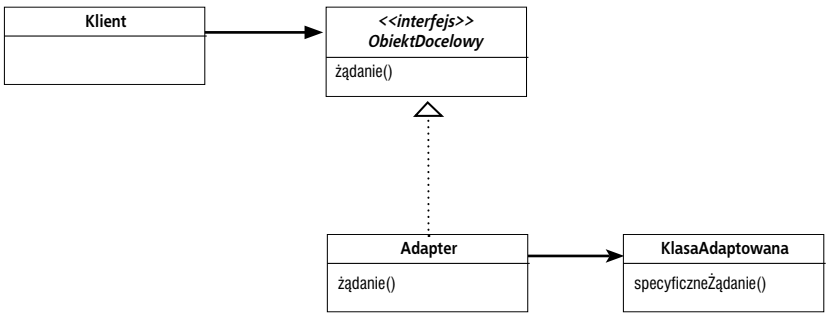
Magnesiki z Kaczkami

Na dole strony zamieściliśmy „magnesiki” z rysunkami przedstawiającymi Kaczki oraz Indyki. Twoim zadaniem jest ustawienie poszczególnych magnesików w wybranych miejscach diagramów, odpowiadających rolom odgrywanym przez poszczególne ptaszyska w naszym poprzednim przykładzie. Na diagramy powinieneś również ponanosić swoje notatki, opisujące sposób działania poszczególnych adapterów.

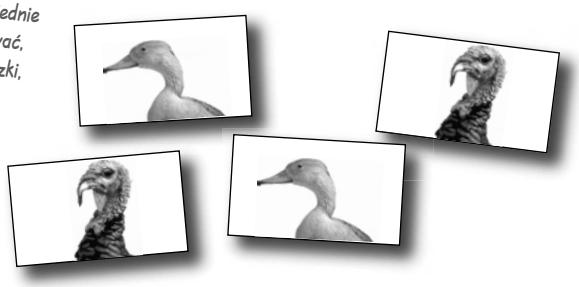
Adapter klas



Adapter obiektów



Przeciągnij wybrane magnesiki na odpowiednie pozycje diagramów klas tak, aby zobrazować, które części diagramów reprezentują Kaczki, a które Indyki.

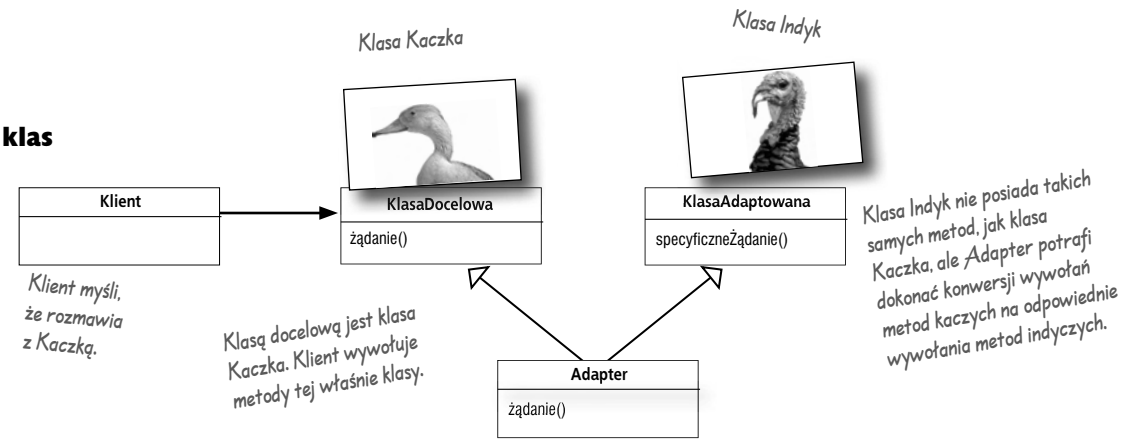




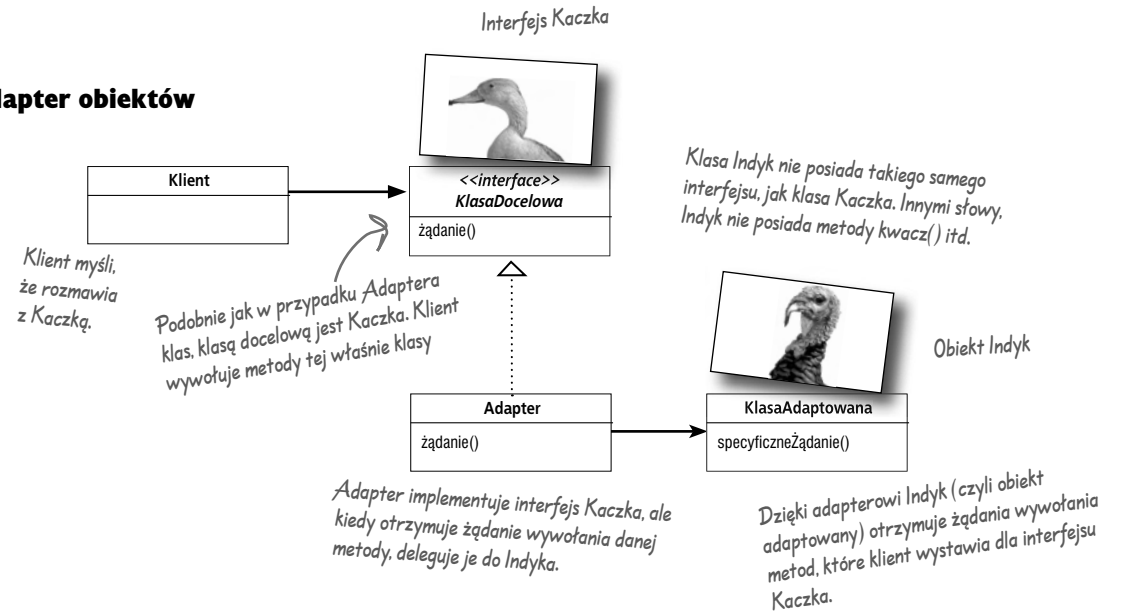
Magnesiki z Kaczkami
Rozwiązanie

Uwaga: adapter klas wymaga zastosowania mechanizmu wielokrotnego dziedziczenia, więc nie możesz z niego korzystać w języku Java...

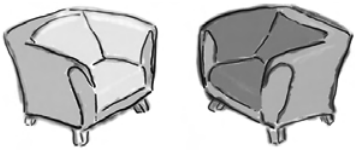
Adapter klas



Adapter obiektów



Pogawędki przy kominku



Temat dzisiejszej wieczornej pogawędki:

Adapter obiektów i Adapter klas — spotkanie twarzą w twarz.

Adapter obiektów

Ponieważ wykorzystuję mechanizm kompozycji, jestem bardzo przydatny. Mogę nie tylko dokonywać konwersji klasy adaptowanej, ale również każdej z jej klas podrzędnych.

W części świata, z której pochodzę, zdecydowanie preferujemy stosowanie kompozycji zamiast dziedziczenia; być może faktycznie oszczędzasz w ten sposób kilka wierszy kodu, ale z drugiej strony, wszystko, czego ja potrzebuję, to kilka wierszy kodu, który deleguje żądania do obiektu adaptowanego. Bardzo lubimy elastyczność we wszystkich projektach.

Zamartwiasz się tym jednym, małym obiektem? Być może faktycznie umiesz szybko przesłonić wybraną metodę swoich superklas, ale pamiętaj o tym, że każde nowe zachowanie, która dołożę do swojego kodu, będzie działało nie tylko z moją klasą adaptowaną, *ale również* z jej wszystkimi klasami podrzędnymi

Hej, odpuść sobie — w takiej sytuacji skorzystam z kompozycji z tą podklasą i znów wszystko będzie działać, jak należy.

Chcesz zobaczyć bałagan? Spójrz w lustro!

Adapter klas

To prawda, mam z tym pewne problemy, ponieważ jestem wiernie oddany tylko jednej, specyficznej klasie adaptowanej, ale dzięki temu mam też wielką zaletę: nie muszę dokładnie reimplementować całej struktury mojej klasy adaptowanej. Jeżeli jest taka potrzeba, mogę również przesłaniać jej zachowania, ponieważ ja po prostu dziedziczę po tej klasie.

Może to jest rzeczywiście elastyczne rozwiązanie, ale czy efektywne? Na pewno nie. Jeżeli korzystasz z adaptera klas, masz tylko mnie, a nie adapter i coś, co jest adaptowane.

Jaaasne, ale co Ty, biedaku, zrobisz jeżeli dana podklasa klasy adaptowanej zechce dodać sobie kilka nowych zachowań? Co wtedy?

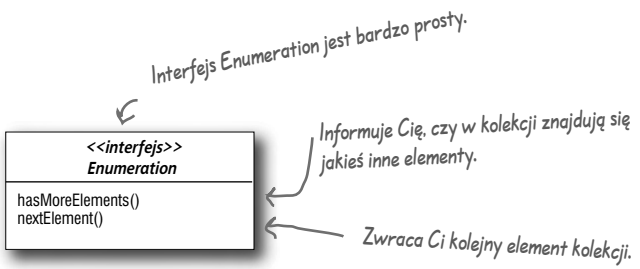
Mnie to wygląda na straszny bałagan…

Adaptery w świecie rzeczywistym

Spróbujmy teraz przyjrzeć się praktycznemu zastosowaniu prostego Adaptera (w końcu powiemy sobie o czymś bardziej poważnym niż ptaszyska różnej maści…).

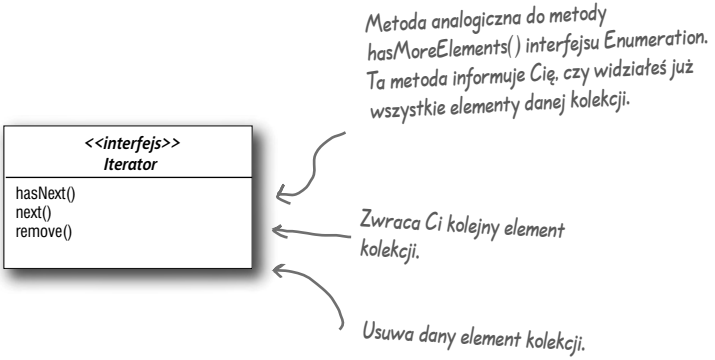
Enumeratory w wiekach średnich

Jeżeli już od pewnego czasu zajmujesz się językiem Java, zapewne pamiętasz, że wczesne Kolekcje (takie jak np. Vector, Stack, Hashtable i kilka innych) implementowały metodę elements(), która zwraca obiekt implementujący interfejs Enumeration. Interfejs Enumeration pozwala na przechodzenie przez kolejne elementy kolekcji bez konieczności posiadania jakiejkolwiek wiedzy na temat tego, jak są one zarządzane wewnątrz kolekcji.



Iteratory ery nowożytnej

Kiedy firma Sun wypuściła na rynek nowsze wersje klas Collections, zaczęły one korzystać z interfejsu Iterator, który — podobnie jak interfejs Enumeration — pozwala Ci na poruszanie się po kolejnych elementach kolekcji, ale jednocześnie pozwala na usuwanie wybranych elementów.

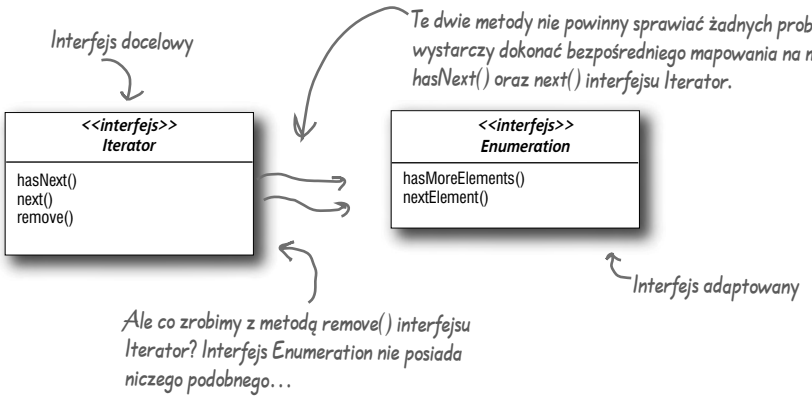


A dzisiaj...

Jeszcze bardzo często spotykamy się ze starszymi aplikacjami, których kod wykorzystuje stary interfejs Enumeration, a zapewne chcielibyśmy, aby jedynym interfejsem używanym w nowych modułach kodu był Iterator. Jak zatem umożliwić ich współdziałanie? Wygląda na to, że musimy utworzyć odpowiedni adapter.

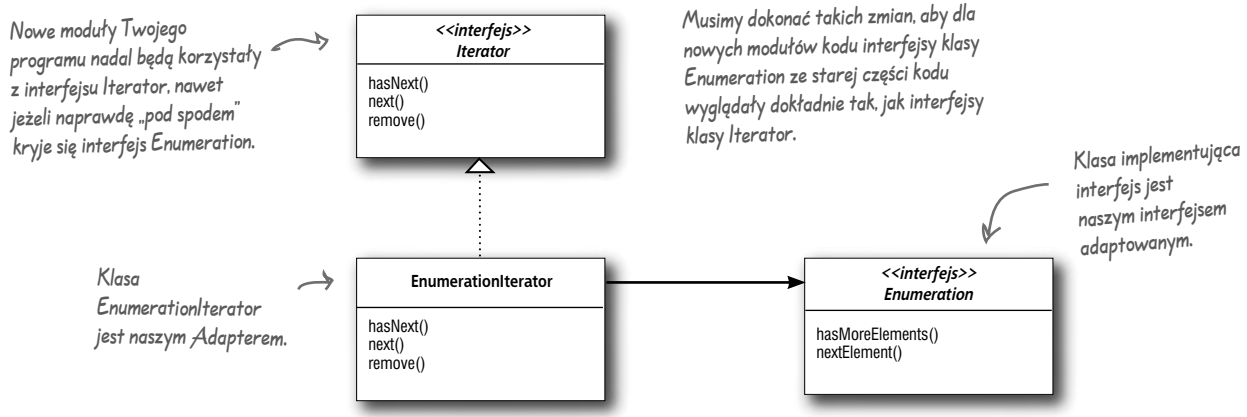
Adaptujemy interfejs Enumeration do wymagań interfejsu Iterator

Najpierw rzucimy okiem na oba interfejsy, aby przekonać się, w jaki sposób powinniśmy dokonać ponownego mapowania poszczególnych metod. Innymi słowy, musimy zorientować się, jaką metodę interfejsu adaptowanego musimy wywołać w momencie, kiedy klient wywoła daną metodę obiektu docelowego.



Projektujemy Adapter

Oto, jak powinny wyglądać nasze klasy: będziemy potrzebowali adaptera, który implementuje interfejs docelowy i został utworzony metodą kompozycji z interfejsu adaptowanego. Metody hasNext() oraz next() będą przenoszone bezpośrednio z interfejsu docelowego na interfejs adaptowany: po prostu będziemy je bezpośrednio delegować. Ale co zrobimy z metodą remove()? Zastanów się nad tym przez chwilę (nasze rozwiązanie omówimy na następnej stronie). A teraz pokażemy Ci diagram klas naszego adaptera:



Co robimy z metodą remove()

Doskonale wiemy, że interfejs Enumeration nie posiada metody remove(), jest de facto interfejsem typu „read only", czyli pozwalającym tylko na odczytywanie kolejnych elementów kolekcji. Nie mamy żadnej możliwości implementacji w pełni funkcjonalnej metody remove() dla tego interfejsu, więc najlepsze, co możemy zrobić, to wygenerować odpowiedni wyjątek. Na szczęście projektanci interfejsu Iterator przewidzieli taką potrzebę i zdefiniowali metodę remove() w ten sposób, że obsługuje wyjątek UnsupportedOperationException.

I to jest właśnie sytuacja, w której nasz adapter nie jest idealnym rozwiązaniem: klient będzie musiał zwracać uwagę na potencjalne wyjątki, ale o ile będzie wiedział, jak sobie z tym poradzić, a operacja ta będzie dobrze udokumentowana, będzie to dobre i bardzo rozsądne rozwiązanie tego problemu.

Tworzymy adapter EnumerationIterator

Oto prosty, ale bardzo efektywny kod adaptera przeznaczonego dla tych wszystkich klas, które jeszcze do tej pory implementują interfejsy Enumeration:

```
public class EnumerationIterator implements Iterator
{
    Enumeration enumeration;

    public EnumerationIterator(Enumeration enumeration) {
        this.enumeration = enumeration;
    }

    public boolean hasNext() {
        return enumeration.hasMoreElements();
    }

    public Object next() {
        return enumeration.nextElement();
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

Ponieważ adaptujemy interfejs Enumeration do interfejsu Iterator, nasz adapter implementuje interfejs Iterator... przecież musi z zewnątrz wyglądać dokładnie tak, jak ten interfejs.

Obiekt z interfejsem Enumeration, który adaptujemy. Używamy kompozycji, więc przechowujemy go w zmiennej obiektowej.

Metoda hasNext() interfejsu Iterator jest delegowana bezpośrednio do metody hasMoreElements() interfejsu Enumeration.....

... a metoda next() interfejsu Iterator jest delegowana bezpośrednio do metody nextElement() interfejsu Enumeration.

Niestety, nie możemy tutaj obsługiwać metody remove() interfejsu Iterator, więc musimy spasować (innymi słowy, po prostu poddajemy się!). Wszystko, na co nas tutaj stać, to wygenerowanie odpowiedniego wyjątku.



Ćwiczenia

Podczas gdy język Java nieodwołalnie ewoluuje w stronę powszechnego stosowania interfejsu Iterator, nadal istnieje (i jeszcze długo istnieć będzie) cała masa nieco przestarzałego oprogramowania klientów uzależnionego od funkcjonowania interfejsu Enumeration. Zatem Adapter, który będzie dokonywał konwersji interfejsu Iterator na interfejs Enumeration, również zdaje się być niezwykle użyteczny i ze wszech miar pożądany.

Twoim zadaniem będzie napisanie odpowiedniego Adaptera, który dokona adaptacji interfejsu Iterator na interfejs Enumeration. Działanie interfejsu możesz przetestować, dokonując adaptacji klasy ArrayList, która obsługuje interfejs Iterator, ale nie obsługuje interfejsu Enumeration (a przynajmniej jeszcze nie...).



WYTĘŻ UMYSŁ

Niektóre adaptery zasilania oferują nieco więcej niż tylko prostą zamianę interfejsu — mogą również dodawać nowe mechanizmy, takie jak ochrona przed przepięciami, diody sygnalizacyjne i różne inne mniej lub bardziej przydatne gadżety.

Jeżeli chciałbyś zaimplementować tego typu mechanizmy, jakiego wzorca projektowego chciałbyś użyć?



Temat dzisiejszej wieczornej pogawędki: **wzorce Dekorator i Adapter dyskutują o wzajemnych różnicach.**

Dekorator

Jestem bardzo ważny. Cała moja praca jest związana z *odpowiedzialnością* — sam wiesz, jak to jest — jeżeli Dekorator jest w coś zaangażowany, z pewnością jest to związane z nowym zakresem odpowiedzialności lub nowymi zachowaniami dodanymi do istniejącego kodu.

Może to i racja, ale nie myśl sobie, że my nie pracujemy równie ciężko. Kiedy trzeba odpowiednio udekorować duży interfejs, oooo, wtedy możemy potrzebować naprawdę dużej ilości kodu.

Fajne! Ale nie myśl, że zgarniamy całą chwałę; czasami jestem tylko jednym z dekoratorów, który jest opakowany Bóg wie iloma innymi. Kiedy wywołanie danej metody zostanie w końcu do Ciebie oddelegowane, nie masz zielonego pojęcia, ile innych dekoratorów dotychczas z nim się użerało, a co więcej, nie wiesz nawet, czy dostaniesz jakiekolwiek potwierdzenie Twoich wysiłków przy obsłudze takiego żądania.

Adapter

Pewnie, chcielibyście zgarnąć całą chwałę i splendor, podczas gdy my, adaptery, jesteśmy zawsze na pierwszej linii frontu, odwalając za Was całą czarną robotę: konwersję interfejsów. Nasza praca nie jest może olśniewająca i nie zniewala wszystkich dookoła, ale z pewnością nasi klienci ogromnie doceniają to, że powodujemy, że ich życie staje się prostsze.

A spróbuj kiedyś postawić się w sytuacji adaptera, który, aby spełnić wymagania interfejsu klienta, musi zebrać razem kilka klas. To jest naprawdę ciężka robota. Ale mamy między sobą takie nasze powiedzenie: „odseparowany klient to zadowolony klient”.

Dobra, dobra, a jak my dobrze wykonujemy swoją ciężką pracę, nasi klienci nawet nie mają pojęcia, że tam jesteśmy. To bywa naprawdę niewdzięczna robota.

Dekorator

No cóż, dekoratory robią w zasadzie to samo, tyle że w inny sposób — pozwalamy po prostu, aby nowe zachowania zostały dodane do danej klasy i — jak sam zauważyłeś — również bez konieczności jakiejkolwiek modyfikacji istniejącego kodu. Mimo wszystko nadal twierdzę, że adaptery to taka wymyślna odmiana dekoratora — mam na myśli to, że Wy, podobnie jak i my, w gruncie rzeczy opakowujecie obiekty.

O nie! Naszą misją jest rozszerzanie zachowań i zakresów odpowiedzialności opakowywanych obiektów, nie jesteśmy tylko *prostymi pośrednikami*.

Może jednak powinniśmy się zgodzić co do tego, że się zupełnie nie zgadzamy. Wygląda na to, że na papierze jesteśmy do siebie troszkę podobni, ale nasze *zadania* zdecydowanie pochodzą z dwóch różnych bajek.

Adapter

Naszą wielką zaletą jest to, że umożliwiamy naszym klientom korzystanie z wielu nowych bibliotek i pakietów bez konieczności modyfikacji *jakiegokolwiek* ich kodu. Oni polegają wyłącznie na nas i wierzą, że dokonamy dla nich poprawnej konwersji. Niby nic wielkiego, ale jesteśmy w tym naprawdę dobrzy.

Nie, nie, nie, nie, to nie tak. My *zawsze* dokonujemy konwersji interfejsów opakowywanych obiektów, a Wy nie robicie tego *nigdy*. Ja bym raczej powiedział, że dekorator działa podobnie do adaptera; tyle tylko, że nie potrafi zmieniać interfejsów!

Hej, kogo Ty nazywasz prostym pośrednikiem? Zejdź na ziemię, zajmij się konkretną robotą i wtedy zobaczymy, jak długo wytrzymasz, dokonując konwersji kilku interfejsów!

O tak! Tutaj muszę się z Tobą zgodzić!

A teraz dla odmiany...

W tym rozdziale będziemy omawiali jeszcze jeden wzorzec.

Widziałeś już, w jaki sposób wzorzec Adapter dokonuje konwersji interfejsu określonej klasy do postaci oczekiwanej przez danego klienta. Wiesz także, że w języku Java dokonujemy tego poprzez opakowywanie obiektu posiadającego niekompatybilny interfejs innym obiektem, który dysponuje poprawnym interfejsem.

Teraz mamy zamiar zająć się wzorcem, który również dokonuje pewnej modyfikacji interfejsów, ale z zupełnie innego powodu: aby doprowadzić do ich uproszczenia. Ktoś kiedyś bardzo zresztą trafnie nazwał ten wzorzec „Fasada”, ponieważ potrafi on ukrywać całe skomplikowane mechanizmy jednej czy też kilku klas za jedną, zgrabną, prostą i ładnie podświetloną fasadą.

Kto co robi

Dopasuj nazwy poszczególnych wzorców do zadań, które realizują.

Wzorzec	Przeznaczenie
Dekorator	Dokonuje konwersji jednego interfejsu na inny.
Adapter	Nie modyfikuje interfejsu, ale dodaje zachowania.
Fasada	Powoduje uproszczenie interfejsu.

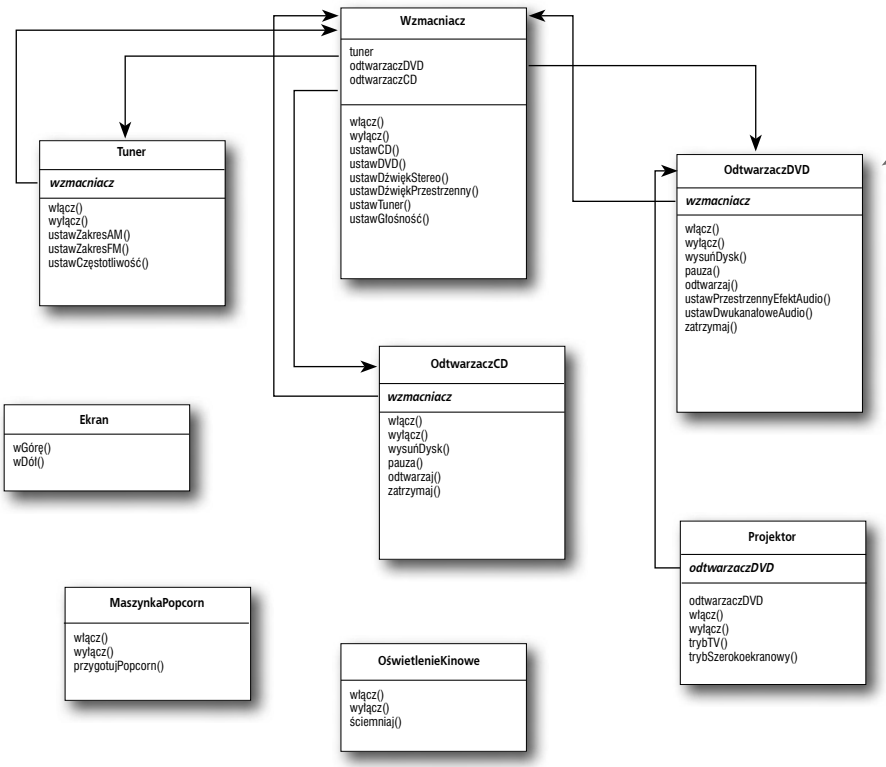
Nie ma to jak kino domowe



Zanim zagłębimy się w szczegółowe omawianie zagadnień wzorca Fasada, przyjrzyjmy się jednej z rozwijających się w niemal zastraszającym tempie obsesji narodowych: budowie własnego kina domowego.

Załóżmy, że przeprowadziłeś już niezbędną analizę rynku i skompletowałeś naprawdę zabójczy, kompletny system z odtwarzaczem DVD, dużym ekranem projektorem, automatycznie wysuwanym ekranem, systemem dźwięku przestrzennego i nawet specjalną maszynką do robienia prażonej kukurydzy.

Sprawdź teraz, czy wszystkie podzespoły zostały dobrze podłączone:



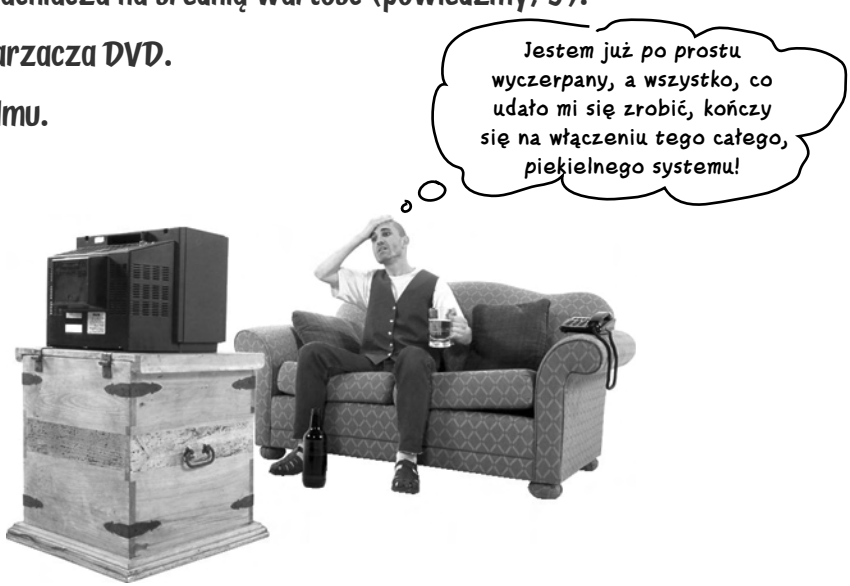
Mamy sporo klas, wiele interakcji i wielki zestaw interfejsów do nauczenia się i wykorzystania

Spędziłeś już całe tygodnie, ciągnąc okablowanie, montując projektor, sprawdzając setki połączeń oraz testując i kalibrując cały system. Nadszedł wreszcie wspaniały moment uruchomienia tego całego ustrojstwa, zapalenia hawańskiego cygara i rozkoszowania się wspaniałym obrazem filmowym…

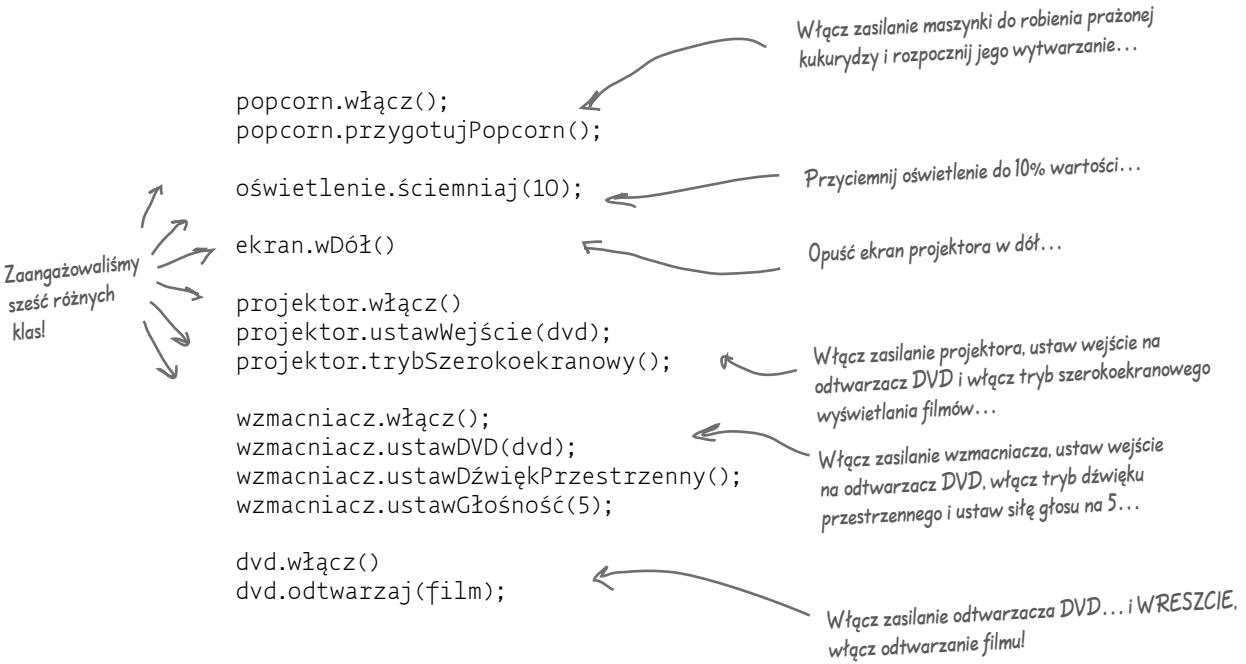
Oglądanie filmów (wersja dla prawdziwych twardzieli)

Wybierz ze swojego zbioru odpowiednią płytę DVD z filmem, zrelaksuj się i nastaw psychikę na niezapomniane chwile, które niesie ze sobą magia kina. Och, byłbym zapomniał — aby obejrzeć wybrany film, musisz jeszcze wcześniej wykonać kilka „drobnych" operacji:

1. Włącz zasilanie maszynki do robienia prażonej kukurydzy.
2. Wsyp odpowiednią ilość kukurydzy i uruchom wytwarzanie prażonej kukurydzy.
3. Przyciemnij światła.
4. Opuść ekran projektora w dół.
5. Włącz zasilanie projektora.
6. Ustaw wejście projektora na odtwarzacz DVD.
7. Ustaw projektor w tryb wyświetlania szerokoekranowego.
8. Włącz zasilanie wzmacniacza audio.
9. Ustaw wejście wzmacniacza audio na odtwarzacz DVD.
10. Ustaw wyjście wzmacniacza na tryb dźwięku przestrzennego.
11. Ustaw siłę głosu wzmacniacza na średnią wartość (powiedzmy, 5).
12. Włącz zasilanie odtwarzacza DVD.
13. Włącz odtwarzanie filmu.



Przeanalizujmy jeszcze raz te same operacje, ale w kontekście klas i wywołań metod niezbędnych do ich realizacji:



Ale jest jeszcze coś więcej...

- Kiedy napisy końcowe filmu będą już daleko za nami, w jaki sposób wyłączyć całe to ustrojstwo? Czy przypadkiem nie będziesz musiał wykonać ponownie całej tej sekwencji operacji, tyle, że w odwrotnej kolejności?
- Czy przypadkiem prozaiczne słuchanie CD czy też radia nie będzie wymagało przynajmniej częściowego przeprowadzenia takiej procedury?
- Jeżeli zdecydujesz się na rozbudowę systemu, prawdopodobnie będziesz musiał nauczyć się nowej, nieco zmodyfikowanej procedury jego obsługi.

W takim razie, co tu zrobić? Stopień złożoności procedury korzystania z tego systemu „kina domowego” robi się już zbyt oczywisty!

Zobaczmy zatem, w jaki sposób wzorzec Fasada może wybawić nas z kłopotów, tak abyśmy mogli czerpać prawdziwą przyjemność z oglądania filmów w naszym prywatnym, domowym, kinowym zaciszu…

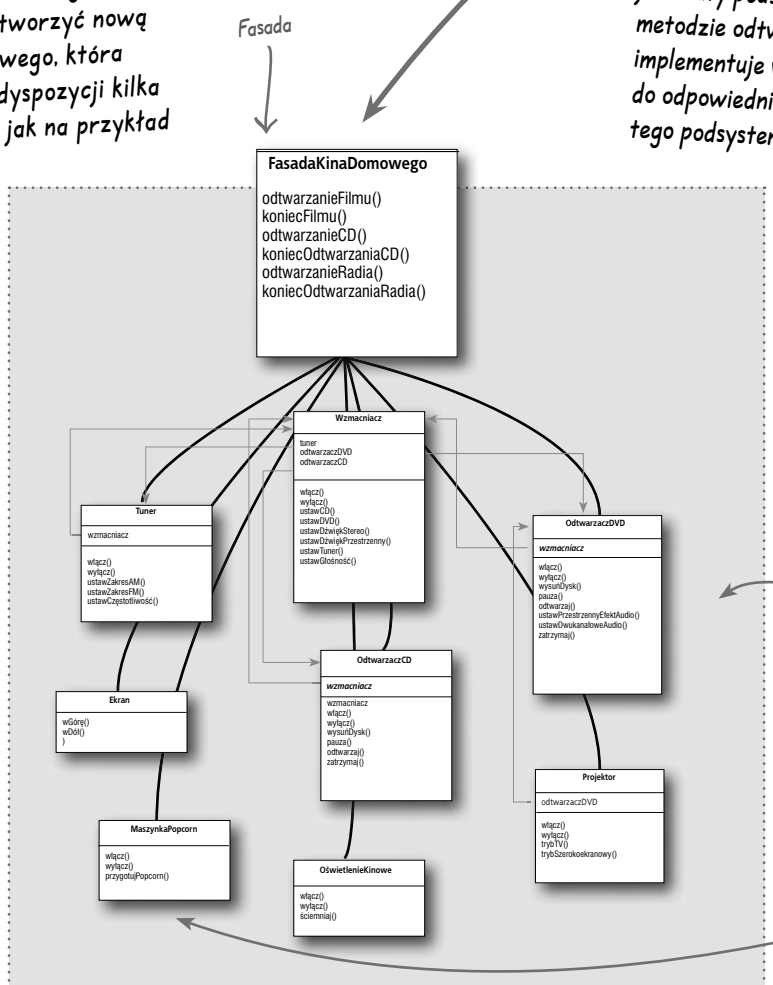
Światła, kamera, fasada!

Wzorzec Fasada to wszystko, czego Ci aktualnie potrzeba: korzystając z niego możesz wziąć niemal dowolnie skomplikowany podsystem i zdecydowanie uprościć procedurę jego obsługi, implementując klasę Fasada zapewniającą jeden, bardziej przejrzysty i spójny interfejs. Nie martw się — jeżeli potrzebujesz pełnej mocy bojowej takiego skomplikowanego podsystemu, zapewniam Cię, że przez cały czas pozostanie ona w pełni do Twojej dyspozycji; jeżeli jednak wszystko, czego teraz potrzebujesz, to prosty, intuicyjny interfejs do sterowania całością, Fasada z przyjemnością Ci go udostępni.

Przyjrzyjmy się zatem bliżej, w jaki sposób funkcjonuje wzorzec Facade:

❶ Nadszedł czas, aby utworzyć odpowiednią Fasadę dla naszego niewiarygodnego systemu kina domowego. Aby tego dokonać, musimy utworzyć nową klasę, FasadaKinaDomowego, która będzie nam dawała do dyspozycji kilka prostych metod, takich jak na przykład odtwarzanieFilmu().

❷ Klasa Fasada traktuje poszczególne składniki kina domowego jako cały podsystem i w swojej metodzie odtwarzanieFilmu() implementuje wywołania do odpowiednich elementów tego podsystemu.



Podsystem, którego obsługa będzie upraszczana przez wzorzec Facade.

odtwarzanieFilmu()

Klient ~~podsystemu~~ fasady



3. Kod Twojego klienta wywołuje teraz odpowiednie metody Fasady systemu kina domowego, a nie samego podsystemu. Teraz, jeżeli chcesz oglądać film, wystarczy, że wywołasz tylko jedną metodę, odtwarzanieFilmu(), a ona w odpowiedni sposób „dogada" się z oświetleniem, odtwarzaczem DVD, projektorem, wzmacniaczem, ekranem i maszynką do robienia prażonej kukurydzy.



Ten facet był kiedyś przewodniczącym naszego szkolnego kółka Szalonych Naukowców.

4. Zastosowanie Fasady w niczym nie ogranicza możliwości bezpośredniego korzystania z poszczególnych elementów składowych podsystemu. Jeżeli potrzebujesz zaawansowanych mechanizmów poszczególnych klas składowych podsystemu, możesz z nich korzystać, kiedy tylko zechcesz.

Nie istnieją głupie pytania

P: Jeżeli wzorzec Fasada hermetyzuje klasy podsystemu, to w jaki sposób klient, który potrzebuje mechanizmów niskiego poziomu, może mieć do nich dostęp?

O: Wzorzec Fasada tak naprawdę nie „hermetyzuje" klas podsystemu; zapewnia jedynie uproszczony interfejs do obsługi ich mechanizmów. Klasy podsystemu nadal są bezpośrednio dostępne dla każdego klienta, który musi korzystać z bardziej specyficznych interfejsów. Jest to bardzo miła właściwość wzorca Fasada: zapewnia uproszczony interfejs do całości, podczas gdy pełna funkcjonalność systemu jest nadal dostępna dla wszystkich, którzy mogą jej potrzebować.

P: Czy wzorzec Fasada dodaje jakieś swoje mechanizmy, czy też po prostu spełnia rolę sprytnego pośrednika odpowiednio przekazującego sekwencje odwołań do podsystemu?

O: Fasada pozwala na dodawanie dowolnej ilości swoich własnych „inteligentnych" rozwiązań do całego systemu. Przykładowo, Fasada naszego wspaniałego systemu kina domowego nie pozwala, co prawda, na implementację żadnych nowych zachowań podsystemu, ale jest wystarczająco „sprytna", aby wiedzieć, że zanim maszynka do robienia prażonej kukurydzy zacznie z siebie wyrzucać całe tony tej wspaniałej przekąski, warto jej najpierw włączyć zasilanie.

P: Czy każdy podsystem może mieć tylko jedną Fasadę?

O: Niekoniecznie. Definicja tego wzorca pozwala na utworzenie dla danego podsystemu dowolnej ilości różnych fasad.

P: Jakie są zalety stosowania wzorca Fasada, oczywiście poza tą, że udostępnia nam uproszczoną wersję interfejsu?

O: Wzorzec Fasada pozwala również na odseparowanie implementacji klienta od poszczególnych podsystemów. Załóżmy, że zdecydowałeś się na radykalny krok wymiany wszystkich podzespołów kina na nowe. Wspaniale, tyle że teraz wszystkie nowe podzespoły będą miały nowe, inne interfejsy. Jeżeli jednak podczas tworzenia aplikacji kod klienta napisałeś pod kątem Fasady, a nie konkretnych implementacji podsystemu, nie będziesz musiał tego kodu w żaden sposób modyfikować. Modyfikacji będzie podlegał tylko kod Fasady (a jest duża szansa, że tę część zadania weźmie na siebie dostawca nowych urządzeń systemu kina domowego!).

P: Czyli w zasadzie można powiedzieć, że różnica pomiędzy wzorcem Adapter a wzorcem Fasada polega na tym, że adapter może opakować tylko jedną klasę, a Fasada może reprezentować wiele klas?

O: Nie! Pamiętaj, wzorzec Adapter zamienia interfejsy jednej lub większej ilości klas w jeden interfejs zgodny z oczekiwaniami klienta. Chociaż większość przykładów zastosowania tego wzorca spotykanych w podręcznikach dotyczy adaptacji tylko jednej klasy, nic nie stoi na przeszkodzie, aby do stworzenia docelowego interfejsu oczekiwanego przez klienta użyć tylu klas adaptowanych, ile będzie niezbędne. Analogicznie, wzorzec Fasada może w razie potrzeby zapewniać uproszczony interfejs nawet tylko dla jednej klasy dysponującej bardzo rozbudowanym interfejsem.

Różnic pomiędzy tymi dwoma wzorcami nie możemy rozpatrywać w kategoriach ilości klas, które jeden bądź drugi wzorzec może obsługiwać, ale raczej w kategoriach przeznaczenia poszczególnych wzorców. Przeznaczeniem wzorca Adapter jest zmiana danego interfejsu tak, aby spełniał oczekiwania narzucone przez klienta, natomiast przeznaczeniem wzorca Fasada jest zapewnienie uproszczonego interfejsu do danego podsystemu.

Wzorzec Fasada nie tylko zapewnia uproszczony interfejs, ale również umożliwia odseparowanie kodu klienta od kodu poszczególnych składników podsystemu. Fasady i adaptery mogą obsługiwać tak samo wiele klas, ale przeznaczeniem fasady jest upraszczanie interfejsu, podczas gdy przeznaczeniem adaptera jest konwersja interfejsu do innej postaci.

Konstruujemy fasadę naszego systemu kina domowego

Przeanalizujemy teraz szczegółowo konstrukcję naszej klasy FasadaKinaDomowego: pierwszym etapem postępowania jest zastosowanie kompozycji tak, aby fasada miała dostęp do wszystkich elementów składowych podsystemu:

```
public class FasadaKinaDomowego {
    Wzmacniacz wzmacniacz;
    Tuner tuner;
    OdtwarzaczDVD dvd;
    OdtwarzaczCD cd;
    Projektor projektor;
    OświetlenieKinowe oświetlenie;
    Ekran ekran;
    MaszynkaPopcorn popcorn;

    public FasadaKinaDomowego(Wzmacniacz wzmacniacz,
            Tuner tuner,
            OdtwarzaczDVD dvd,
            OdtwarzaczCD cd,
            Projektor projektor,
            Ekran ekran,
            OświetlenieKinowe oświetlenie,
            MaszynkaPopcorn popcorn) {

        this.wzmacniacz = wzmacniacz;
        this.tuner = tuner;
        this.dvd = dvd;
        this.cd = cd;
        this.projektor = projektor;
        this.ekran = ekran;
        this.oświetlenie = oświetlenie;
        this.popcorn = popcorn;
    }
        // tutaj umieść pozostałe metody
}
```

Tutaj dokonujemy kompozycji: są tu wszystkie elementy składowe podsystemu, z których mamy zamiar korzystać.

Fasada otrzymuje odwołanie do każdego z elementów składowych podsystemu w swoim konstruktorze, a następnie zapisuje poszczególne odwołania w odpowiednich zmiennych obiektowych.

Już niebawem zaczniemy wypełniać również i tę część kodu fasady…

Implementujemy uproszczony interfejs

Teraz nadszedł czas, aby umieścić poszczególne elementy składowe podsystemu w ramach jednego, zunifikowanego interfejsu. Na początek zaimplementujemy metody odtwarzanieFilmu() oraz koniecFilmu():

```
public void odtwarzanieFilmu(String film) {
    System.out.println("---- Przygotuj się na seans filmowy...");
    popcorn.włącz();
    popcorn.przygotujPopcorn();
    oświetlenie.ściemniaj(10);
    ekran.wDół();
    projektor.włącz();
    projektor.trybSzerokoekranowy ();
    wzmacniacz.włącz();
    wzmacniacz.ustawDvd(dvd);
    wzmacniacz.ustawDźwiękPrzestrzenny ();
    wzmacniacz.ustawGłośność(5);
    dvd.włącz();
    dvd.odtwarzaj(film);
}

public void koniecFilmu() {
    System.out.println("\n---- Koniec seansu, wyłączam kino domowe...");
    popcorn.wyłącz();
    oświetlenie.włącz ();
    ekran.wGórę();
    projektor.wyłącz ();
    wzmacniacz.wyłącz ();
    dvd.zatrzymaj();
    dvd.wysuńDysk();
    dvd.wyłącz ();
}
```

Metoda odtwarzanieFilmu() wykorzystuje tę samą sekwencję poleceń, którą musieliśmy uprzednio wykonywać ręcznie, ale tym razem poszczególne operacje są ładnie zapakowane do tej metody, która wykonuje za nas całą robotę. Zwróć uwagę, że odpowiedzialność za wykonanie poszczególnych zadań za każdym razem delegujemy do odpowiedniego elementu składowego podsystemu kina domowego.

Metoda koniecFilmu() bierze na siebie wszystkie zadania związane z wyłączeniem całego ustrojstwa po zakończeniu seansu. I znów, odpowiedzialność za wykonanie poszczególnych zadań delegujemy do odpowiednich elementów składowych podsystemu kina domowego.



WYTĘŻ UMYSŁ

Pomyśl, jakie fasady spotkałeś do tej pory w bibliotekach API języka Java? W jakich miejscach chciałbyś zamieścić kilka nowych fasad?

Czas na seans (proste podejście)



Czas na SEANS FILMOWY!

```
public class FasadaKinaDomowegoTest {
    public static void main(String[] args) {
        // tutaj umieść składniki tworzące odpowiednie obiekty

        FasadaKinaDomowego kinoDomowe =
            new FasadaKinaDomowego (wzmacniacz, tuner, dvd, cd,
            projektor, ekran, oświetlenie, popcorn);
        kinoDomowe.odtwarzajFilm("Poszukiwacze zaginionej arki");
        kinoDomowe.koniecFilmu();
    }
}
```

Tutaj tworzymy poszczególne obiekty dla naszego środowiska testowego. W normalnych warunkach, kiedy klient korzysta z fasady, nie musi jej sam sobie tworzyć.

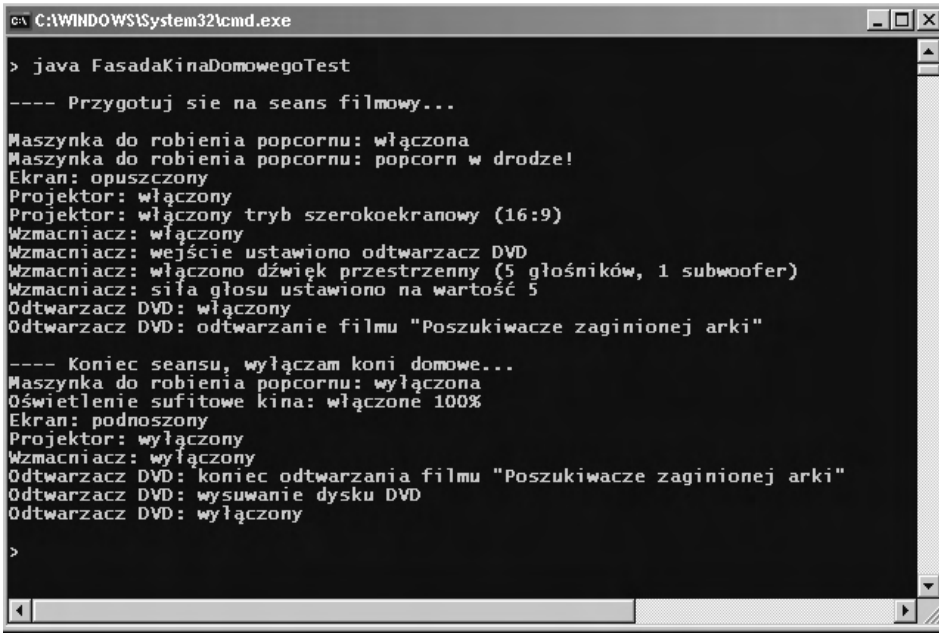
Najpierw musimy utworzyć instancję obiektu Fasady posiadającą wszystkie składniki danego podsystemu.

Wykorzystujemy uproszczony interfejs do uruchomienia odtwarzania filmu, a następnie, po zakończeniu seansu — do wyłączenia całego systemu.

Oto wyniki działania naszego programu.

Wywołanie metody odtwarzanieFilmu() Fasady kina domowego wykonuje za nas całą robotę…

… a tutaj, już po zakończeniu seansu, wywołanie metody koniecFilmu() powoduje wyłączenie całego systemu.



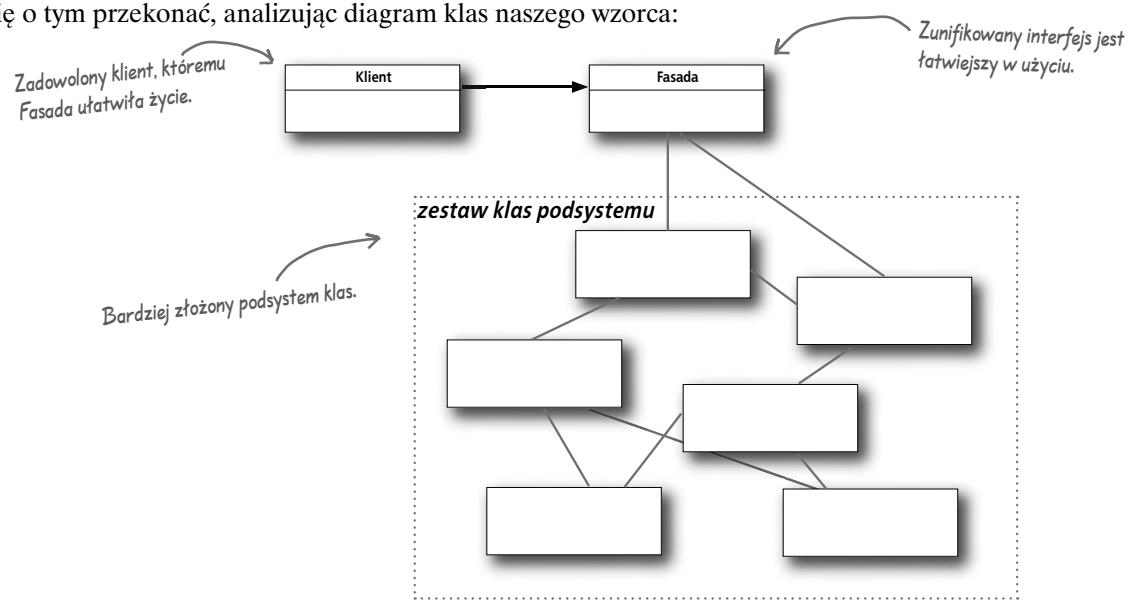
Definicja wzorca Fasada

Aby skorzystać z wzorca Fasada, musimy utworzyć nową klasę, której zadaniem jest uproszczenie i unifikacja całego zestawu złożonych klas wchodzących w skład danego podsystemu. W przeciwieństwie do wielu innych wzorców, wzorzec Fasada jest całkiem prosty; zrozumienie jego zasad funkcjonowania nie wymaga żadnej abstrakcyjnej i skomplikowanej gimnastyki umysłu. Jego prostota nie umniejsza jednak w niczym ogromnych możliwości: zastosowanie Fasady pozwala na uniknięcie ciasnego sprzężenia pomiędzy oprogramowaniem klientów a poszczególnymi składnikami podsystemu, a także, jak się już za moment przekonasz, pozwala na pozostawanie w zgodzie z naszą kolejną, bardzo ważną regułą programowania obiektowego.

Zanim jednak poznasz tę nową regułę, zobaczmy, jak wygląda oficjalna definicja wzorca Fasada:

> **Wzorzec Fasada** zapewnia jeden, zunifikowany interfejs dla całego zestawu interfejsów określonego podsystemu. Fasada tworzy nowy interfejs wysokiego poziomu, który powoduje, że korzystanie z całego podsystemu staje się zdecydowanie łatwiejsze.

W zasadzie nie pozostało już do omówienia wiele rzeczy, o których byś nie wiedział, ale warto jeszcze raz zwrócić uwagę na fakt, że najważniejszą cechą każdego wzorca projektowego jest jego przeznaczenie. Powyższa definicja jasno i wyraźnie daje nam do zrozumienia, że przeznaczeniem wzorca Fasada jest ułatwienie sposobu korzystania z danego podsystemu poprzez zastosowanie nowego, uproszczonego interfejsu. Możesz się o tym przekonać, analizując diagram klas naszego wzorca:



To wszystko; właśnie dołożyłeś nowy wzorzec projektowy do swojej kolekcji! A teraz nadszedł czas na zaprezentowanie nowej reguły projektowania obiektowego. Ale uważaj, może ona być sporym wyzwaniem!

Reguła ograniczania interakcji

Nasza tytułowa reguła zaleca ograniczanie interakcji pomiędzy obiektami do kilku najbliższych „przyjaciół". Reguła ta zazwyczaj jest definiowana następująco:



Reguła projektowania

Reguła ograniczania interakcji — rozmawiaj tylko z najbliższymi przyjaciółmi.

Ale co to oznacza w praktyce? Mniej więcej tyle, że podczas tworzenia projektu danego systemu powinieneś zwracać baczną uwagę na ilość klas, z którymi współpracuje każdy obiekt, jak również na sposób, w jaki z nimi współpracuje.

Zastosowanie tej reguły pozwala uniknąć tworzenia projektów, w których wiele różnych klas jest ze sobą ściśle powiązanych — tak, że zmiana w jednej z nich wymusza lawinę zmian w innych klasach. Kiedy tworzysz wiele zależności pomiędzy klasami, system staje się bardzo wrażliwy na modyfikacje, kosztowny w utrzymaniu i serwisowaniu, jak również (ze względu na swoją złożoność) trudny do analizy dla innych.



WYTĘŻ UMYSŁ

Ile klas jest powiązanych ze sobą w poniższym przykładzie?

```
public float pobierzTemperaturę() {
    return stacjaMeteo.pobierzTermometr().pobierzTemperaturę();
}
```

Jak zrazić do siebie przyjaciół, czyli interakcje między obiektami

No dobrze, ale jak się od tego powstrzymać? Nasza reguła daje kilka wskazówek: dla dowolnego obiektu powinniśmy wywoływać tylko takie metody, które należą do:

- samego obiektu
- obiektów przekazywanych jako argumenty metody
- dowolnego obiektu, który tworzy dana metoda
- dowolnego składnika danego obiektu

Zwróć uwagę, że te wytyczne nie pozwalają na wywoływanie metod obiektów, które zostały zwrócone jako wynik wywołania innych metod!

Jako „składniki" powinieneś rozumieć dowolne obiekty, do których odwołania są przechowywane w zmiennych obiektowych. Innymi słowy, chodzi o takie obiekty, do których obiekt podmiotowy posiada relację MA.

Brzmi trochę restrykcyjnie, prawda? W końcu co jest złego w wywoływaniu metody obiektu, który otrzymujemy jako rezultat wywołania innej metody? No cóż, jeżeli będziemy tak postępować, okaże się, że wystawiamy żądanie do zupełnie innego obiektu (a tym samym zwiększymy ilość obiektów, które „znamy" osobiście). W takich sytuacjach nasza wspaniała reguła wymusza, aby takie żądanie wystawił dla nas inny obiekt; dzięki temu nie musimy nic wiedzieć o składnikach takiego obiektu (i jednocześnie nasz krąg „przyjaciół" pozostaje cały czas niewielki). Przykładowo:

Niezgodnie z regułą

```
public float pobierzTemperaturę() {
    Termometr termometr = stacjaMeteo.pobierzTermometr();
    return termometr.pobierzTemperaturę();
}
```

Tutaj otrzymujemy ze stacji meteo obiekt reprezentujący termometr i sami wywołujemy jego metodę pobierzTemperaturę().

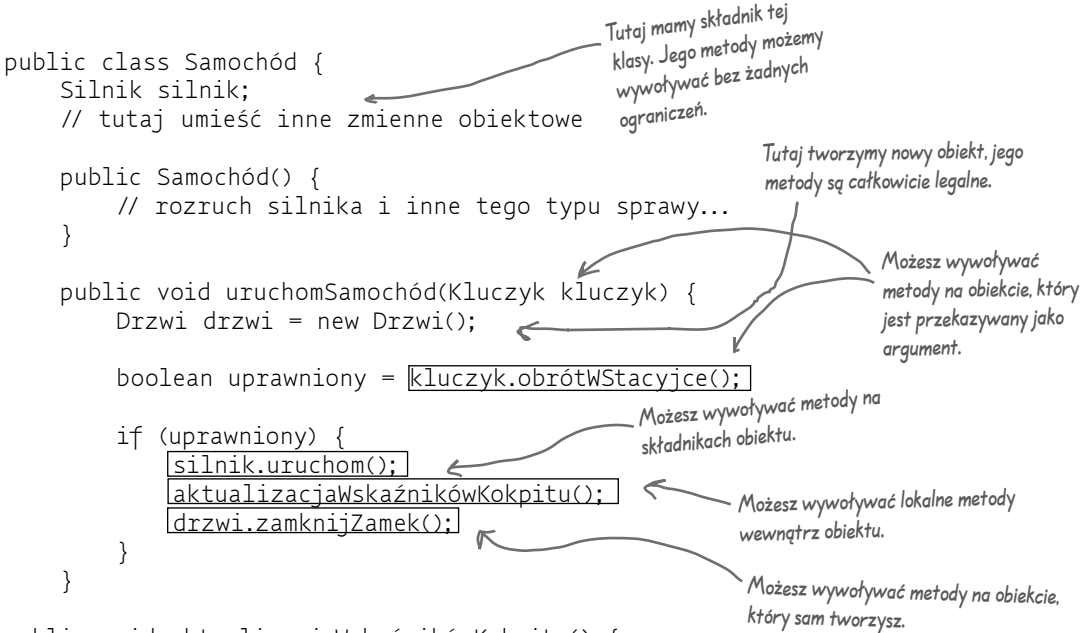
Zgodnie z regułą

```
public float pobierzTemperaturę() {
    return stacjaMeteo.pobierzTemperaturę();
}
```

Kiedy postępujemy zgodnie z regułą, dodajemy metodę do klasy stacjaMeteo, a ona dokonuje za nas tego wywołania. W ten sposób redukujemy ilość klas, od których jesteśmy uzależnieni.

Utrzymywanie wywołań metod w rozsądnych granicach...

Poniżej zamieściliśmy kod klasy Samochód, która demonstruje wszystkie dopuszczalne przez regułę ograniczania interakcji sposoby wywoływania metod:




```
public class Samochód {
    Silnik silnik;
    // tutaj umieść inne zmienne obiektowe

    public Samochód() {
        // rozruch silnika i inne tego typu sprawy...
    }

    public void uruchomSamochód(Kluczyk kluczyk) {
        Drzwi drzwi = new Drzwi();

        boolean uprawniony = kluczyk.obrótWStacyjce();

        if (uprawniony) {
            silnik.uruchom();
            aktualizacjaWskaźnikówKokpitu();
            drzwi.zamknijZamek();
        }
    }

public void aktualizacjaWskaźnikówKokpitu() {
    // aktualizacja informacji wyświetlanych na desce
rozdzielczej
    }
}
```

Nie istnieją głupie pytania

P: Istnieje jeszcze inna reguła, nazywana prawem Demeter; w jaki sposób te dwie reguły są ze sobą powiązane?

O: Obie reguły to tak naprawdę jedno i to samo; zapewne często będziesz spotykał zarówno jedną, jak i drugą nazwę.
My preferujemy nazwę „reguła ograniczania interakcji", a to z kilku powodów: (1) nazwa reguły jest bardziej intuicyjna i łatwiejsza do zapamiętania i (2) zastosowanie słowa „prawo" implikuje, że zawsze musimy postępować zgodnie z tą regułą.
W rzeczywistości żadna reguła nie stanowi prawa i wszystkie powinny być stosowane tam, gdzie ich użycie jest uzasadnione i przyniesie wymierne korzyści. W każdym projekcie będziemy musieli przystać na mniejsze bądź większe kompromisy (zastosowanie modeli abstrakcyjnych kontra szybkość działania aplikacji, przestrzeń kontra czas i tak dalej) i choć reguły wyznaczają pewien sposób postępowania, decyzja o ich zastosowaniu w praktyce zawsze powinna być podjęta po rozważeniu wszystkich argumentów za i przeciw.

P: Czy są jeszcze jakieś inne przeciwwskazania do korzystania z reguły ograniczania interakcji?

O: Tak; pomimo tego, że zastosowanie tej reguły znacząco redukuje występujące zależności pomiędzy obiektami oraz ułatwia serwisowanie i modyfikację aplikacji, z drugiej strony wymusza powstawanie dużej ilości dodatkowych klas „osłonowych", które obsługują wywołania metod innych obiektów. Może to prowadzić do zwiększenia stopnia złożoności kodu aplikacji i zwiększenia czasu niezbędnego do jej utworzenia, jak również może negatywnie wpływać na jej wydajność.



Zaostrz ołówek

Czy któraś z poniższych klas nie jest zgodna z regułą ograniczania interakcji? Dlaczego lub dlaczego nie?

```
public Dom {
    StacjaMeteo stacjaMeteo;

    // tutaj umieść inne metody i konstruktor klasy

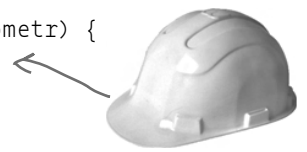
    public float pobierzTemperaturę() {
        return stacjaMeteo.pobierzTermometr().pobierzTemperaturę();
    }
}
```

```
public Dom {
    StacjaMeteo stacjaMeteo;

    // tutaj umieść inne metody i konstruktor klasy

    public float pobierzTemperaturę() {
        Termometr termometr = stacjaMeteo.pobierzTermometr();
        return pobierzTemperaturęZTermometra(termometr);
    }

    public float pobierzTemperaturęZTermometra(Termometr termometr) {
        return termometr.pobierzTemperaturę();
    }
}
```



ZAŁÓŻ KASK — STREFA NIEBEZPIECZNA — UWAGA NA SPADAJĄCE ZAŁOŻENIA



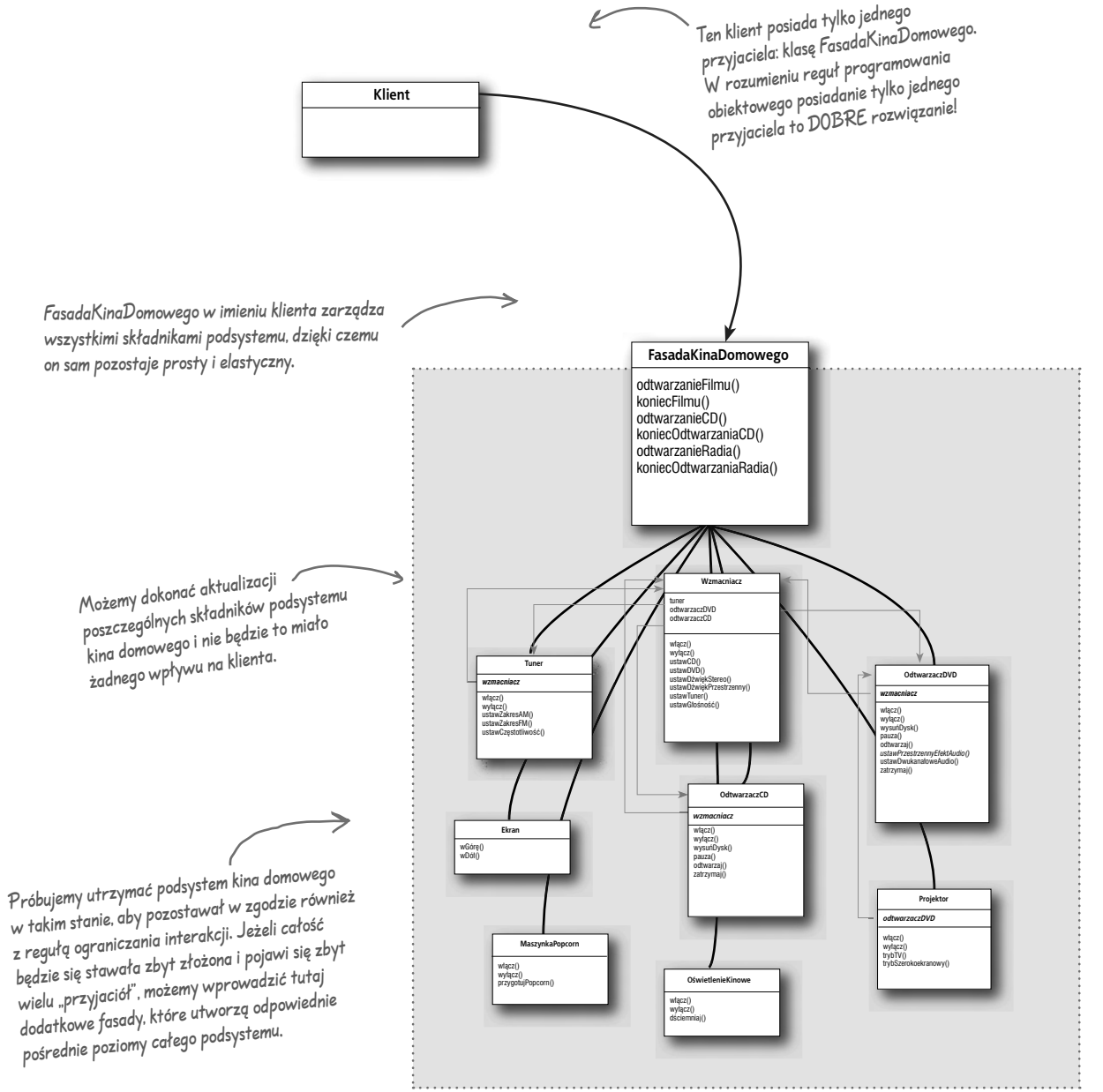
WYTĘŻ UMYSŁ

Zastanów się, czy znasz jakieś powszechne konstrukcje języka Java, które łamią postanowienia reguły ograniczania interakcji?

Czy powinieneś się tym przejmować?

A co powiesz na System.out.println()?

Wzorzec Fasada kontra reguła ograniczania interakcji





Twoja skrzynka narzędziowa

Twoja skrzynka narzędziowa zaczyna się pomału robić naprawdę ciężka! W tym rozdziale dołożyliśmy do niej parę wzorców, które pozwalają na modyfikację interfejsów oraz na redukcję zależności pomiędzy klientami a systemami, z których korzystają.



Dysponujemy teraz nową techniką pozwalającą na zarządzanie zależnościami na niskich poziomach naszych projektów (pamiętaj, rozmawiaj tylko z najbliższymi przyjaciółmi...)



... oraz mamy DWA nowe wzorce projektowe. Każdy z nich ma inny wpływ na interfejsy — adapter dokonuje konwersji interfejsów, a fasada je upraszcza i unifikuje.

CELNE SPOSTRZEŻENIA

- Kiedy chcesz skorzystać z danej klasy, ale posiada ona niewłaściwy interfejs, użyj adaptera.
- Kiedy chcesz uprościć i zunifikować rozbudowany interfejs lub złożony zestaw interfejsów, użyj fasady.
- Adapter dokonuje konwersji interfejsu do postaci zgodnej z oczekiwaniami klienta.
- Fasada odseparowuje klienta od złożonych podsystemów.
- Implementacja adaptera może wymagać niewielkiego nakładu pracy lub wręcz przeciwnie, ogromnego wysiłku. Wszystko zależy od rozmiarów i stopnia złożoności interfejsu docelowego.
- Implementacja fasady wymaga dokonania jej kompozycji z podsystemem i delegowania żądań do poszczególnych elementów podsystemu.
- Istnieją dwa rodzaje adapterów: adapter obiektów i adapter klas. Implementacja adaptera klas wymaga zastosowania mechanizmu wielokrotnego dziedziczenia.
- W danym podsystemie możesz zaimplementować więcej niż jedną fasadę.
- Adapter opakowuje obiekt w celu zmiany jego interfejsu, dekorator opakowuje obiekt, dodając nowe zachowania, a fasada „opakowuje" zestawy obiektów w celu uproszczenia ich obsługi.



Rozwiązania ćwiczeń

Zaostrz ołówek

Załóżmy, że będzie nam również potrzebny Adapter, który dokonuje konwersji Kaczki na Indyka. Nazwijmy go KaczkaAdapter. Twoim zadaniem będzie napisanie poniżej kodu odpowiedniej klasy:

```
public class KaczkaAdapter implements Indyk {
    Kaczka kaczka;

    public KaczkaAdapter(Kaczka kaczka) {
        this.kaczka = kaczka;
        rand = new Random();
    }

    public void gulgocz() {
        kaczka.kwacz();
    }

    public void lataj() {
        if (rand.nextInt(5) == 0) {
            kaczka.lataj();
        }
    }
}
```

Teraz adaptujemy Indyki do postaci Kaczek, więc musimy zaimplementować interfejs Indyk.

Zapamiętujemy w zmiennej obiektowej odwołanie do Kaczki, którą będziemy adaptować.

Tworzymy również obiekt reprezentujący pewną wartość losową; zobacz, jak go użyjemy w metodzie lataj().

Gulgotanie właśnie zaczęło brzmieć jak kwakanie.

Ponieważ kaczki z reguły potrafią latać dużo dalej niż indyki, podjęliśmy decyzję, że jeden lot naszej „indykokaczki" będzie przypadał średnio raz na pięć prób.

Zaostrz ołówek

Czy któraś z poniższych klas nie jest zgodna z regułą ograniczania interakcji? Dlaczego lub dlaczego nie?

```
public Dom {
    StacjaMeteo stacjaMeteo;

    // tutaj umieść inne metody i konstruktor klasy

    public float pobierzTemperaturę() {
        return stacjaMeteo.pobierzTermometr().pobierzTemperaturę();
    }
}
```

Łamie regułę ograniczania interakcji! Wywołujesz metodę na obiekcie, który otrzymaliśmy jako rezultat wywołania innej metody.

```
public Dom {
    StacjaMeteo stacjaMeteo;

    // tutaj umieść inne metody i konstruktor klasy

    public float pobierzTemperaturę() {
        Termometr termometr = stacjaMeteo.pobierzTermometr();
        return pobierzTemperaturęZTermometra(termometr);
    }

    public float pobierzTemperaturęZTermometra(Termometr termometr) {
        return termometr.pobierzTemperaturę();
    }
}
```

Nie łamie reguły ograniczania interakcji! Wygląda na to, że udało nam się obejść postanowienia tej reguły. Czy coś tak naprawdę się zmieniło od czasu, kiedy przenieśliśmy wywołanie do innej metody?



Rozwiązania ćwiczeń

Widziałeś już, w jaki sposób można zaimplementować adapter, który dopasuje interfejs Enumeration do interfejsu Iterator; Twoim zadaniem będzie napisanie odpowiedniego Adaptera, który dokona adaptacji interfejsu Iterator na interfejs Enumeration.

```
public class IteratorEnumeration implements Enumeration {
    Iterator iterator;

    public IteratorEnumeration(Iterator iterator) {
        this.iterator = iterator;
    }

    public boolean hasMoreElements() {
        return iterator.hasNext();
    }

    public Object nextElement() {
        return iterator.next();
    }
}
```

Kto co robi

Dopasuj nazwy poszczególnych wzorców do zadań, które realizują.

Wzorzec	Przeznaczenie
Dekorator	Dokonuje konwersji jednego interfejsu na inny.
Adapter	Nie modyfikuje interfejsu, ale dodaje zachowania.
Fasada	Powoduje uproszczenie interfejsu.

Hermetyzacja algorytmów

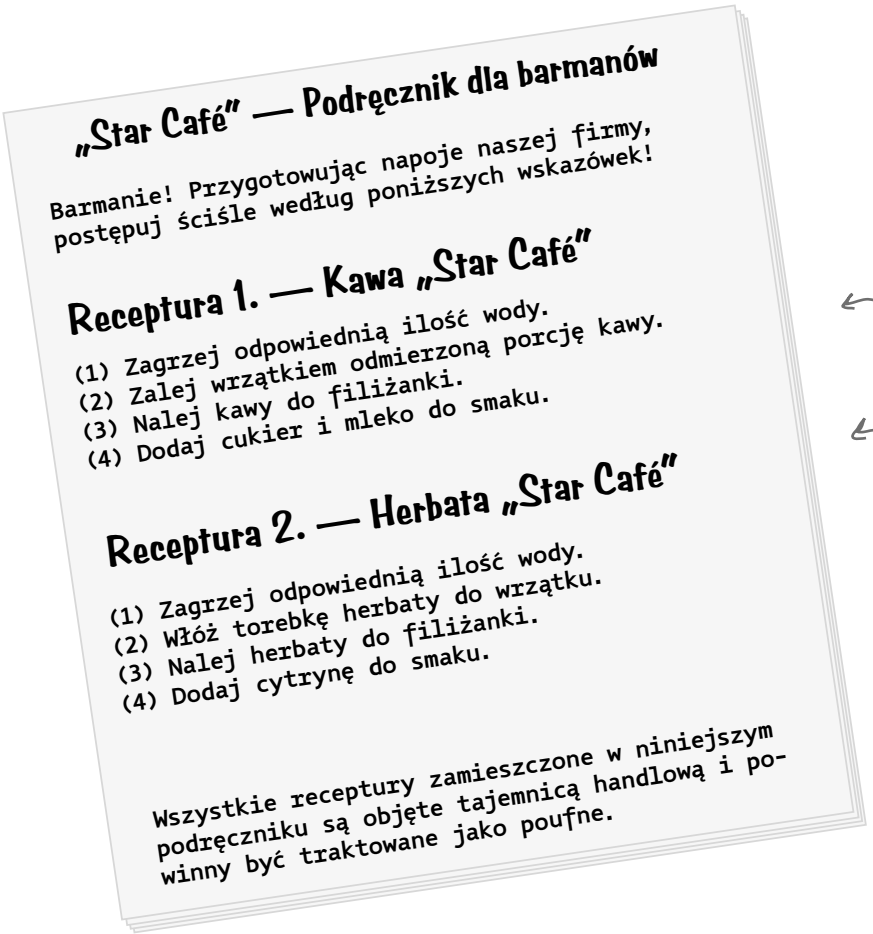


Jesteśmy jak w transie: hermetyzowaliśmy już proces tworzenia obiektów, wywołania metod, złożone interfejsy, kaczki, indyki, pizze… ciekawe, co będzie następne? Otóż teraz mamy zamiar zająć się hermetyzacją fragmentów algorytmów, tak aby klasy podrzędne mogły „podczepiać się" w różnych miejscach wykonywanych obliczeń. Co więcej, zajmiemy się również regułą projektowania, której korzenie wywodzą się w prostej linii z... Hollywood.

Czas na nieco więcej kofeiny...

Niektórzy ludzie nie wyobrażają sobie życia bez kawy, a jeszcze inni praktycznie nie mogą żyć bez herbaty. Jaki jest wspólny mianownik tych upodobań? Oczywiście, kofeina!

Ale jest coś jeszcze: zarówno herbatę, jak i kawę przyrządza się w dosyć podobny sposób. Zobaczmy zatem, jak to wygląda:



„Star Café" — Podręcznik dla barmanów

Barmanie! Przygotowując napoje naszej firmy, postępuj ściśle według poniższych wskazówek!

Receptura 1. — Kawa „Star Café"

(1) Zagrzej odpowiednią ilość wody.
(2) Zalej wrzątkiem odmierzoną porcję kawy.
(3) Nalej kawy do filiżanki.
(4) Dodaj cukier i mleko do smaku.

Receptura 2. — Herbata „Star Café"

(1) Zagrzej odpowiednią ilość wody.
(2) Włóż torebkę herbaty do wrzątku.
(3) Nalej herbaty do filiżanki.
(4) Dodaj cytrynę do smaku.

Wszystkie receptury zamieszczone w niniejszym podręczniku są objęte tajemnicą handlową i powinny być traktowane jako poufne.

Receptura parzenia kawy wygląda dosyć podobnie do tej opisującej przyrządzanie herbaty, prawda?

Tworzymy klasy reprezentujące kawę i herbatę (w języku Java)



Zabawimy się teraz w swego rodzaju „barmana-programistę" i napiszemy trochę kodu, który pomoże nam w przyrządzaniu kawy i herbaty.

Oto kod dla kawy:

Oto nasza klasa Kawa, która będzie nam pomagała parzyć ten doskonały napój.

```
public class Kawa {

    void recepturaParzenia() {
        gotowanieWody();
        zaparzanieKawy();
        nalewanieDoFiliżanki();
        dodawanieCukruOrazMleka();
    }

    public void gotowanieWody() {
        System.out.println("Gotowanie wody");
    }

    public void zaparzanieKawy() {
        System.out.println("Zaparzanie i przesączanie kawy przez filtr");
    }

    public void nalewanieDoFiliżanki() {
        System.out.println("Nalewanie kawy do filiżanki");
    }

    public void dodawanieCukruOrazMleka() {
        System.out.println("Dodawanie cukru i mleka do smaku");
    }
}
```

Tutaj mamy recepturę przyrządzania kawy dokładnie według wytycznych z podręcznika barmana.
Każdy z etapów parzenia kawy został zaimplementowany w postaci osobnej metody.

Każda z tych metod implementuje jeden etap działania algorytmu. Mamy tutaj osobne metody do gotowania wody, zaparzania kawy, nalewania jej do filiżanek oraz dodawania cukru i mleka.

a teraz pora na herbatę...



To tutaj wygląda bardzo podobnie do tego, co zaimplementowaliśmy w klasie Kawa; co prawda etap drugi i czwarty są inne, ale ogólnie chodzi mniej więcej o taką samą recepturę.

```
public class Herbata{

    void recepturaParzenia() {
        gotowanieWody();
        wkładanieTorebkiHerbatyDoWrzątku();
        nalewanieDoFiliżanki();
        dodawanieCytryny();
    }

    public void gotowanieWody() {
        System.out.println("Gotowanie wody");
    }

    public void wkładanieTorebkiHerbatyDoWrzątku() {
        System.out.println("Wkładanie torebki herbaty do wrzątku");
    }

    public void dodawanieCytryny() {
        System.out.println("Dodawanie cytryny");
    }

    public void nalewanieDoFiliżanki() {
        System.out.println("Nalewanie do filiżanki");
    }
}
```

Te dwie metody są charakterystyczne dla klasy Herbata.

Zwróć uwagę, że te dwie metody są dokładnie takie same, jak w klasie Kawa! Wynika stąd jasno, że w tych miejscach mamy po prostu duplikację kodu.



Kiedy spotykamy się ze zjawiskiem duplikacji kodu, jest to nieomylny znak, że musimy uporządkować i wyczyścić nasz projekt. Ponieważ przygotowywanie kawy i herbaty w tak małym stopniu różni się od siebie, wygląda na to, że moglibyśmy umieścić elementy wspólne tych klas jako metody abstrakcyjne w klasie bazowej.

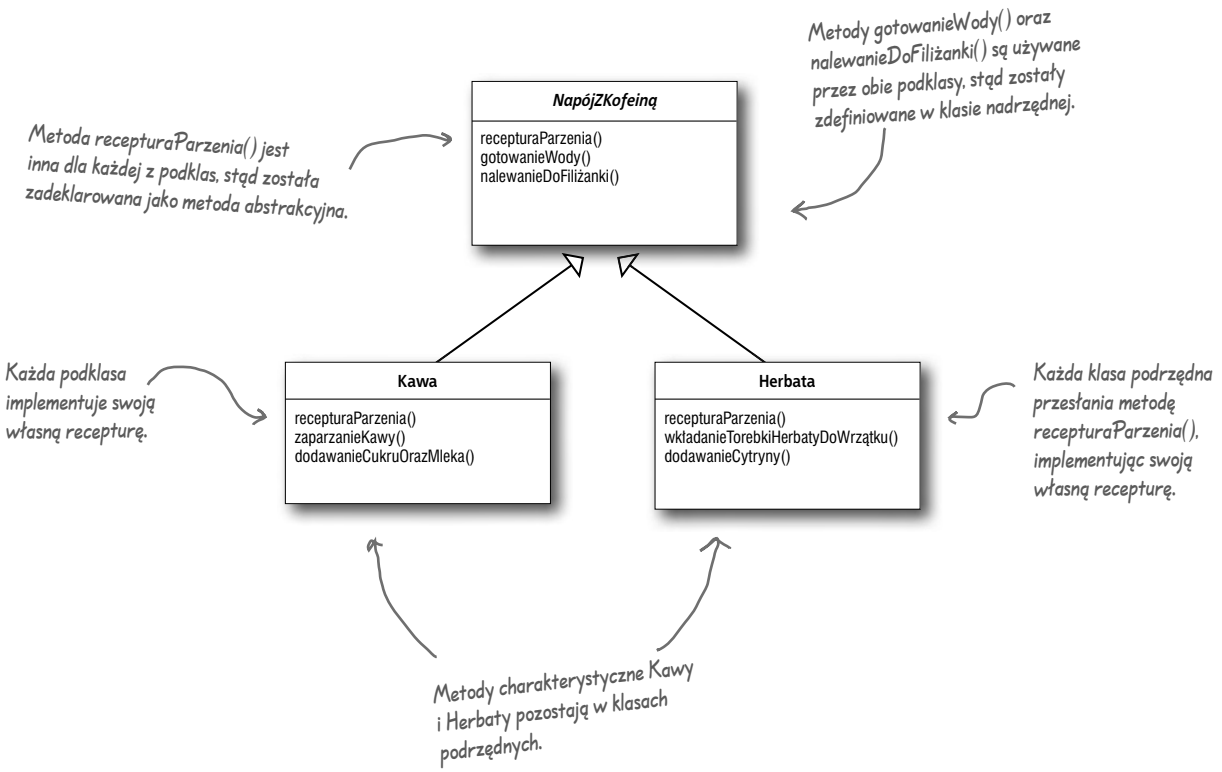


Łamigłówka projektowa

Jak już zauważyliśmy, w klasach Kawa i Herbata występuje całkiem spore zjawisko duplikacji kodu. Rzuć jeszcze raz krytycznym okiem na definicje tych klas i narysuj tutaj diagram klas pokazujący, w jaki sposób należałoby przeprojektować je tak, aby uniknąć niepożądanej redundancji kodu.

Kawa i herbata, czyli klasy abstrakcyjne

Wygląda na to, że dzięki klasom Kawa i Herbata mamy doskonałą okazję do przeprowadzenia prostego ćwiczenia w projektowaniu klas. Twoje pierwsze cięcia mogłyby wyglądać na przykład tak:





WYTĘŻ UMYSŁ

Czy dobrze wykonaliśmy nasze zadanie przeprojektowania klas? Hmmmm, przyjrzyj się jeszcze raz dokładnie. Czy przeoczyliśmy jakieś cechy wspólne? W jaki jeszcze sposób klasy Kawa i Herbata są do siebie podobne?

Ciągniemy nasz projekt o krok dalej...

Co jeszcze klasy Kawa i Herbata mają ze sobą wspólnego? Rozpocznijmy analizę od odpowiednich receptur.

Receptura 1. — Kawa „Star Café"

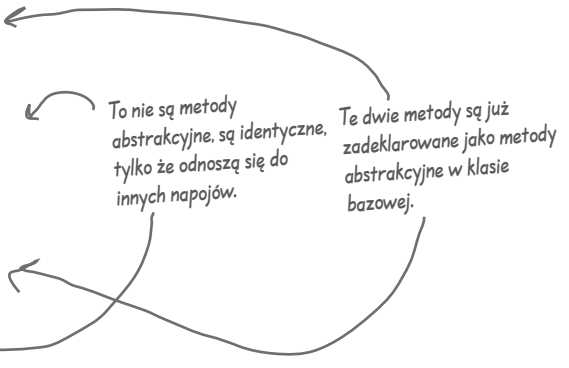
```
(1) Zagrzej odpowiednią ilość wody.
(2) Zalej wrzątkiem odmierzoną porcję kawy.
(3) Nalej kawy do filiżanki.
(4) Dodaj cukier i mleko do smaku.
```

Receptura 2. — Herbata „Star Café"

```
(1) Zagrzej odpowiednią ilość wody.
(2) Włóż torebkę herbaty do wrzątku.
(3) Nalej herbaty do filiżanki.
(4) Dodaj cytrynę do smaku.
```

Zwróć uwagę, że obydwie receptury wykorzystują ten sam algorytm:

1. Zagrzej odpowiednią ilość wody.
2. Użyj wrzątku do zaparzenia kawy lub herbaty.
3. Nalej uzyskany napój do filiżanki.
4. Domieszaj odpowiednie dodatki do napoju.



A teraz, czy możemy znaleźć jakiś sposób, aby metoda recepturaParzenia() również mogła być zadeklarowana jako metoda abstrakcyjna? Tak, zaraz przekonasz się, w jaki sposób jest to możliwe...

Wydobywanie metody recepturaParzenia()

Przejdziemy teraz krok po kroku proces wydobywania metody recepturaParzenia() z poszczególnych klas podrzędnych (to znaczy z klas Kawa i Herbata)...

❶ Pierwszy problem, z jakim się spotkamy, to fakt, że klasa Kawa wykorzystuje metody zaparzanieKawy() i dodawanieCukruOrazMleka(), podczas gdy klasa Herbata korzysta z metod wkładanieTorebkiHerbatyDoWrzątku() i dodawanieCytryny().

Kawa

```
void recepturaParzenia() {
    gotowanieWody();
    zaparzanieKawy();
    nalewanieDoFiliżanki();
    dodawanieCukruOrazMleka();
}
```

Herbata

```
void recepturaParzenia() {
    gotowanieWody();
    wkładanieTorebkiHerbatyDoWrzątku();
    nalewanieDoFiliżanki();
    dodawanieCytryny();
}
```

Pomyślmy w ten sposób: parzenie Kawy i wkładanie torebki herbaty do wrzątku nie różnią się od siebie aż tak bardzo; to w zasadzie podobne operacje. Utworzymy więc całkiem nową nazwę metody, powiedzmy, zaparzanie(), i będziemy z niej korzystać niezależnie od tego, czy zaparzamy kawę, czy herbatę.

Analogicznie rzecz ujmując, dodawanie cukru i mleka jest znów bardzo podobną operacją do dodawania cytryny: obie operacje po prostu dodają coś do napoju. I znów, utworzymy nową nazwę metody, powiedzmy, domieszanieDodatków(), która powinna nam załatwić problem. A zatem, nasza nowa metoda recepturaParzenia() będzie teraz wyglądała następująco:

```
void recepturaParzenia() {
    gotowanieWody();
    zaparzanie();
    nalewanieDoFiliżanki();
    domieszanieDodatków();
}
```

❷ Teraz mamy już, co prawda, nową metodę recepturaParzenia(), ale musimy jeszcze dopasować ją naszego kodu. Aby tego dokonać, musimy rozpocząć prace od utworzenia superklasy NapójZKofeiną:

Klasa NapójZKofeiną jest klasą abstrakcyjną, tak jak w diagramie klas.

```
public abstract class NapójZKofeiną {

    final void recepturaParzenia() {
        gotowanieWody();
        zaparzanie();
        nalewanieDoFiliżanki();
        domieszanieDodatków();
    }

    abstract void zaparzanie();

    abstract void domieszanieDodatków();

    void gotowanieWody() {
        System.out.println("Gotowanie wody");
    }

    void nalewanieDoFiliżanki() {
        System.out.println("Nalewanie do filiżanki");
    }
}
```

Teraz do przygotowywania zarówno Kawy, jak i Herbaty będzie używana jedna i ta sama metoda recepturaParzenia(). Jest ona zadeklarowana jako final, ponieważ nie chcemy, aby nasze klasy podrzędne miały możliwość przesłonięcia tej metody i zmiany receptury! Etap drugi oraz etap czwarty zostały uogólnione do metod odpowiednio: zaparzanie() oraz domieszanieDodatków().

Ponieważ zarówno klasa Kawa, jak i klasa Herbata obsługują te metody na różne sposoby, będą one musiały zostać zadeklarowane jako metody abstrakcyjne. Niech podklasy się tym zajmą!

Pamiętaj, te dwie klasy zostały już wcześniej przeniesione do superklasy NapójZKofeiną (sprawdź w diagramie klas!).

❸ Wreszcie na koniec musimy się zająć klasami Kawa i Herbata. Ponieważ teraz samą recepturą zajmuje się superklasa NapójZKofeiną, te dwie klasy muszą zadbać tylko o obsługę zaparzania i dodatków:

Tak jak w diagramie klas, klasy Kawa i Herbata dziedziczą po klasie NapójZKofeiną.

```
public class Herbata extends NapójZKofeiną {
    public void zaparzanie() {
        System.out.println("Wkładanie torebki herbaty do
wrzątku");
    }
    public void domieszanieDodatków() {
        System.out.println("Dodawanie cytryny");
    }
}

public class Kawa extends NapójZKofeiną  {
    public void zaparzanie() {
        System.out.println("Zaparzanie i przesączanie kawy
przez filtr ");
    }
    public void domieszanieDodatków() {
        System.out.println("Dodawanie cukru oraz mleka do
smaku");
    }
}
```

W klasie Herbata muszą zostać zdefiniowane dwie metody, które zostały zadeklarowane jako metody abstrakcyjne w superklasie: zaparzanie() oraz domieszanieDodatków().

To samo robimy w klasie Kawa, z tym, że klasa ta wykorzystuje kawę mieloną oraz cukier i mleko zamiast torebek herbaty i cytryny.



Zaostrz ołówek

Twoim zadaniem będzie narysowanie nowego diagramu klas odzwierciedlającego przeniesienie implementacji metody recepturaParzenia() do klasy nadrzędnej NapójZKofeiną:

Czego już dokonaliśmy?

Zauważyliśmy, że dwie receptury są bardzo do siebie podobne, chociaż niektóre ich etapy wymagają różnych implementacji. Dzięki temu spostrzeżeniu mogliśmy uogólnić recepturę i umieścić ją w klasie bazowej.

Herbata

1. Zagrzej odpowiednią ilość wody.
2. Włóż torebkę herbaty do wrzątku.
3. Nalej herbaty do filiżanki.
4. Dodaj cytrynę do smaku.

Kawa

1. Zagrzej odpowiednią ilość wody.
2. Zalej wrzątkiem odmierzoną porcję kawy.
3. Nalej kawy do filiżanki.
4. Dodaj cukier i mleko do smaku.

Napoje zawierające kofeinę

1. Zagrzej odpowiednią ilość wody.
2. Zaparz napój.
3. Nalej uzyskany napój do filiżanki.
4. Domieszaj do napoju odpowiednie dodatki.

uogólnienie

uogólnienie

niektóre etapy delegowane są do klasy podrzędnej

niektóre etapy delegowane są do klasy podrzędnej

Podklasa Herbata



2. Włóż torebkę herbaty do wrzątku.
4. Dodaj cytrynę.

Klasa NapójZKofeiną doskonale zna poszczególne etapy receptury, choć sama wykonuje tylko etap pierwszy i trzeci, zostawiając realizację etapów drugiego i czwartego klasom Kawa i Herbata.

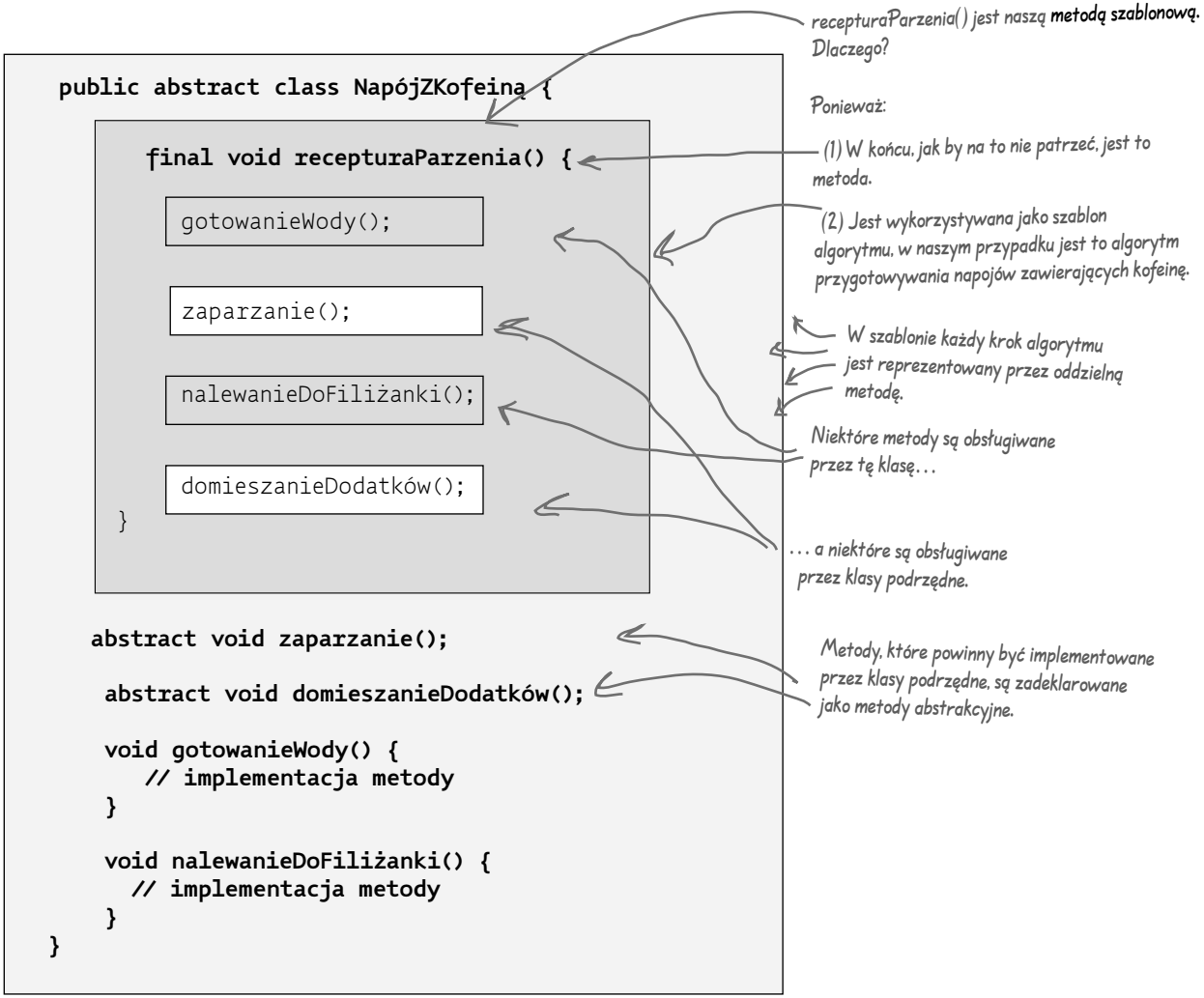
Podklasa Kawa



2. Zaparz odmierzoną porcję kawy.
4. Dodaj cukier i mleko.

Spotkanie z wzorcem Metoda Szablonowa

W zasadzie z czystym sumieniem możemy powiedzieć, że już zaimplementowaliśmy wzorzec Metoda Szablonowa. Co to takiego? Rzućmy okiem na strukturę klasy NapójZKofeiną; to właśnie ona zawiera ową tytułową „metodę szablonową":



Wzorzec Metoda Szablonowa definiuje kolejne kroki danego algorytmu i pozwala klasom podrzędnym na dostarczanie implementacji dla jednego lub więcej z nich.

Zróbmy sobie herbatę...



Przejdziemy teraz krok po kroku przez proces parzenia herbaty i prześledzimy, w jaki sposób działa nasza metoda szablonowa. Zaraz zobaczysz, że metoda szablonowa kontroluje działanie danego algorytmu; w określonych punktach pozwala klasom podrzędnym na dostarczenie implementacji danego kroku...

```
gotowanieWody();
zaparzanie();
nalewanieDoFiliżanki();
domieszanieDodatków();
```

Metoda recepturaParzenia() kontroluje działanie algorytmu i pozwala, aby niektóre jego kroki były implementowane przez klasy podrzędne; nikt też nie może jej zmienić.

1. No dobrze, najpierw będziemy potrzebowali nowego obiektu klasy Herbata…

```
Herbata mojaHerbata = new Herbata();
```

2. Następnie wywołujemy metodę szablonową:

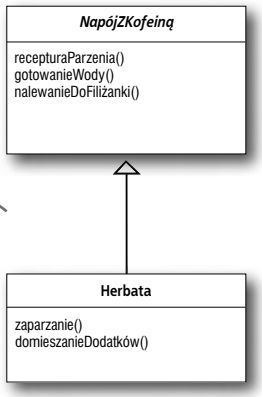
```
mojaHerbata.recepturaParzenia();
```

która postępuje zgodnie z algorytmem przygotowywania napojów zawierających kofeinę…

3. Najpierw gotujemy wodę:

```
gotowanieWody();
```

co jest realizowane przez klasę NapójZKofeiną.



4. Następnie musimy zaparzyć herbatę, ale jak to zrobić, wie już tylko klasa podrzędna:

```
zaparzanie();
```

5. Teraz nalewamy herbaty do filiżanki; sposób postępowania jest tutaj taki sam dla wszystkich napojów, zatem zajmuje się tym klasa NapójZKofeiną:

```
nalewanieDoFiliżanki();
```

6. Wreszcie na koniec wrzucamy dodatki, które są inne dla każdego napoju, zatem znów zadanie to spadnie na odpowiednią klasę podrzędną:

```
domieszanieDodatków();
```

Co nam daje zastosowanie metody szablonowej?



Standardowa implementacja Kawy i Herbaty, bez „dopalacza"

Imprezę prowadzą klasy Kawa i Herbata; one kontrolują działanie algorytmu.

W klasach Kawa i Herbata występuje niepotrzebna redundancja kodu.

Zmiany kodu implementującego algorytm wymagają często znacznych modyfikacji kodu klas podrzędnych.

Struktura klas jest tak zorganizowana, że dodanie nowego napoju z kofeiną wymaga dużego nakładu pracy.

Informacja o sposobie działania algorytmu i jego implementacji jest rozproszona pomiędzy wieloma klasami.



Nowy, czadowy NapójZKofeiną, zasilany metodą szablonową

Informacja o sposobie działania algorytmu i jego implementacji jest rozproszona pomiędzy wieloma klasami.

Klasa NapójZKofeiną maksymalizuje ilość kodu wspólnego dla wszystkich klas podrzędnych.

Algorytm jest zlokalizowany tylko w jednym miejscu, stąd potencjalne zmiany muszą być przeprowadzane tylko tutaj.

Zastosowanie metody szablonowej udostępnia pewien szkielet całego systemu, pozwalający na łatwe dodawanie nowych napojów z kofeiną. Wprowadzenie nowego napoju wymaga jedynie zaimplementowania kilku prostych metod.

Informacja i cała „wiedza" o sposobie działania algorytmu jest skoncentrowana w klasie NapójZKofeiną; niektóre kroki algorytmu są implementowane przez klasy podrzędne.

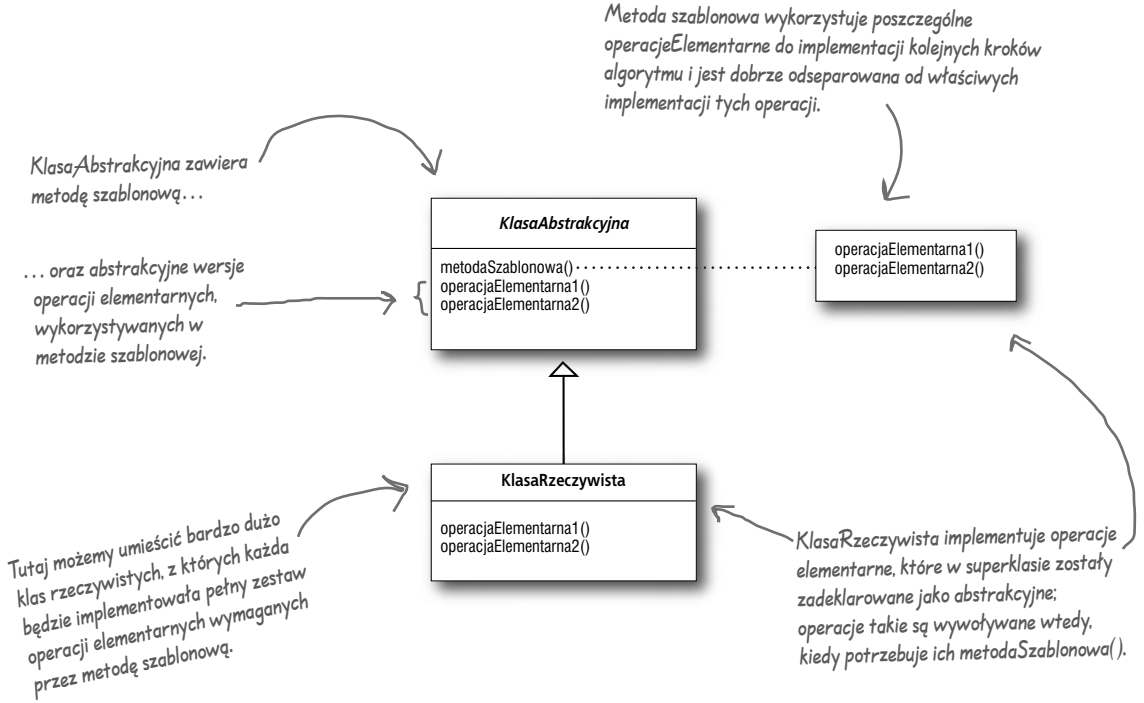
Definicja wzorca Metoda Szablonowa

Nasz przykładowy problem herbaciano-kawowy pozwolił Ci na dokładne zapoznanie się z praktycznymi zasadami funkcjonowania wzorca Metoda Szablonowa; teraz zaś nadszedł czas na jego oficjalną definicję oraz naświetlenie jeszcze kilku szczegółów:

> **Wzorzec Metoda Szablonowa** definiuje szkielet danego algorytmu w określonej metodzie, przekazując realizację niektórych jego kroków do klas podrzędnych. Wzorzec ten pozwala klasom podrzędnym na redefiniowanie pewnych kroków algorytmu, ale jednocześnie uniemożliwia zmianę jego struktury.

Rola wzorca Metoda Szablonowa sprowadza się w całości do utworzenia szablonu dla implementacji określonego algorytmu. Czym zatem jest szablon? Sam pewnie zauważyłeś, że jest to po prostu metoda; a mówiąc bardziej ściśle, jest to metoda, która definiuje dany algorytm w postaci zestawu kolejnych kroków. Co najmniej jeden z nich jest zadeklarowany jako metoda abstrakcyjna i dlatego musi być implementowany w klasach podrzędnych. Takie rozwiązanie zapewnia, że struktura algorytmu pozostanie nienaruszona, podczas gdy klasy podrzędne mogą przesłaniać pewne określone kroki implementacji.

Sprawdzimy teraz, jak wygląda diagram klas tego wzorca:





Bliskie spotkania z kodem aplikacji

Przyjrzyjmy się teraz z bliska, w jaki sposób zbudowana jest KlasaAbstrakcyjna, włączając w to metodę szablonową oraz operacje elementarne.

Tutaj mamy naszą klasę abstrakcyjną: jest zadeklarowana jako abstrakcyjna; dziedziczą po niej wszystkie klasy podrzędne implementujące operacje elementarne.

Tutaj mamy metodę szablonową. Została zadeklarowana jako metoda final, aby uniemożliwić klasom podrzędnym jej przesłonięcie i modyfikację sekwencji poszczególnych kroków algorytmu.

```
abstract class KlasaAbstrakcyjna {

    final void metodaSzablonowa() {
        operacjaElementarna1();
        operacjaElementarna2();
        rzeczywistaOperacjaElementarna();
    }

    abstract void operacjaElementarna1();

    abstract void operacjaElementarna2();

    void rzeczywistaOperacjaElementarna() {
        // kod implementujący
    }
}
```

Metoda szablonowa definiuje sekwencję kroków danego algorytmu, z których każdy jest reprezentowany przez osobną metodę.

W naszym przykładzie dwie operacje elementarne muszą być implementowane przez podklasy rzeczywiste.

W klasie abstrakcyjnej mamy również zdefiniowaną jedną operację rzeczywistą. Za chwilę powiemy nieco więcej o tego typu metodach...



Jeszcze bliższe spotkania z kodem aplikacji

A teraz przyjrzymy się jeszcze bliżej rodzajom metod, jakie mogą być stosowane w klasie abstrakcyjnej:

Zmieniliśmy nieco metodę szablonową, umieszczając w niej wywołanie nowej metody.

```
abstract class KlasaAbstrakcyjna {

    final void metodaSzablonowa() {
        operacjaElementarna1();
        operacjaElementarna2();
        rzeczywistaOperacjaElementarna();
        haczyk();
    }

    abstract void operacjaElementarna1();

    abstract void operacjaElementarna2();

    final void rzeczywistaOperacjaElementarna()
    {
        // kod implementujący
    }

    void haczyk() {}

}
```

Nadal posiadamy tutaj nasze metody elementarne; są zadeklarowane jako metody abstrakcyjne i implementowane w klasach podrzędnych.

Operacja rzeczywista jest definiowana w klasie abstrakcyjnej. Metoda ta została zadeklarowana jako final, dzięki czemu żadna podklasa nie będzie mogła jej przesłonić. Metoda ta może być wykorzystywana bezpośrednio przez metodę szablonową bądź przez klasy podrzędne.

Metoda rzeczywista, która nic nie robi!

Możemy mieć tutaj również metody rzeczywiste, które z definicji nic nie robią; nazywamy je „haczykami". Klasy podrzędne nie mają żadnych ograniczeń w ich przesłanianiu, ale nie muszą tego robić. Zastosowanie takich nieco dziwnych metod omówimy na następnej stronie.

Haczyk na wzorzec Metoda Szablonowa...

Haczyk jest metodą rzeczywistą zadeklarowaną w klasie abstrakcyjnej, ale posiadająca albo pustą, albo inną, domyślną implementację. Takie rozwiązanie daje klasom podrzędnym możliwość „podwieszenia się" do realizowanego algorytmu w różnych jego punktach, jeżeli sobie tego życzą; oczywiście, klasa podrzędna może po prostu zignorować nasz haczyk.

Haczyki mają kilka ciekawych zastosowań; przyjrzyjmy się teraz jednemu z nich. O kilku innych zastosowaniach usłyszysz niebawem:




```
public abstract class NapójZKofeinąZHaczykiem {

    void recepturaParzenia(){
        gotowanieWody();
        zaparzanie();
        nalewajDoFiliżanki();
        if (czyKlientChceDodatki()) {
            domieszanieDodatków();
        }
    }

    abstract void zaparzanie();

    abstract void domieszanieDodatków();

    void gotowanieWody() {
        System.out.println("Gotowanie wody");
    }

    void nalewajDoFiliżanki() {
        System.out.println("Nalewanie do filiżanki");
    }

    boolean czyKlientChceDodatki() {
        return true;
    }
}
```

Dodaliśmy tutaj niewielkie wyrażenie warunkowe, które opiera się na wynikach działania rzeczywistej metody czyKlientChceDodatki(). Metoda domieszajDodatki() zostanie wywołana tylko i wyłącznie wówczas, kiedy klient CHCE dodatki.

Tutaj zdefiniowaliśmy metodę z (prawie) pustą implementacją domyślną. Ta metoda zwraca tylko wartość true (prawda) i to wszystko; nie robi niczego więcej.

*To jest właśnie nasz **haczyk**, ponieważ klasa podrzędna może przesłonić tę metodę, ale wcale robić tego nie musi.*

Zastosowanie haczyka

Aby skorzystać z haczyka, musimy przesłonić go w wybranej klasie podrzędnej. W poniższym przykładzie nasz haczyk decyduje o tym, czy klasa NapójZKofeiną będzie wykonywała określony fragment algorytmu; to znaczy, mówiąc ściślej, czy do napoju zostaną dodane odpowiednie składniki.

A skąd będziemy wiedzieli, czy nasz klient chce, aby do napoju zostały wrzucone dodatki? Po prostu go o to zapytamy!

```
public class KawaZHaczykiem extends NapójZKofeinąZHaczykiem {

    public void zaparzanie() {
        System.out.println("Zaparzanie i przesączanie kawy przez filtr ");
    }

    public void domieszanieDodatków() {
        System.out.println("Dodawanie cukru oraz mleka do smaku ");
    }

    public boolean czyKlientChceDodatki() {

        String odpowiedź = pobierzOdpowiedzi();

        if (odpowiedź.toLowerCase().startsWith("t")) {
            return true;
        } else {
            return false;
        }
    }

    private String pobierzOdpowiedzi() {
        String odpowiedź = null;

        System.out.print("Czy życzy Pan sobie dodania cukru i mleka do kawy (t/n)? ");

        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        try {
            odpowiedź = in.readLine();
        } catch (IOException ioe) {
            System.err.println("Błąd wejścia-wyjścia podczas odczytywania odpowiedzi");
        }
        if (odpowiedź == null) {
            return "nie";
        }
        return odpowiedź;
    }
}
```

Właśnie tutaj dokonujemy przesłonięcia haczyka i dostarczamy naszą własną funkcjonalność tej metody.

Pobierz odpowiedź użytkownika na pytanie o dodatki i na jej podstawie zwróć wartość true lub false.

Ten fragment kodu pyta użytkownika, czy chce on, aby do kawy zostały dodane cukier i mleko, a następnie pobiera jego odpowiedź bezpośrednio z wiersza poleceń.

Testujemy naszą aplikację

Świetnie! Woda się już gotuje... Poniżej zamieściliśmy kod środowiska testowego, dzięki któremu przyrządzimy za chwilę wspaniałą gorącą kawę i herbatę.

```
public class NapojeJazdaPróbna {
    public static void main(String[] args) {

        Herbata herbata = new Herbata();
        Kawa kawa = new Kawa();

        System.out.println("\nPrzyrządzam wspaniałą, gorącą herbatę...");
        herbata.recepturaParzenia();

        System.out.println("\nA teraz pora na gorącą, aromatyczną kawę...");
        kawa.recepturaParzenia();
    }
}
```

Robimy herbatę

Teraz kawa

I w obu przypadkach wywołujemy metodę recepturaParzenia()!

A teraz pora na wyniki działania...

```
C:\WINDOWS\System32\cmd.exe
> java NapojeJazdaPróbna

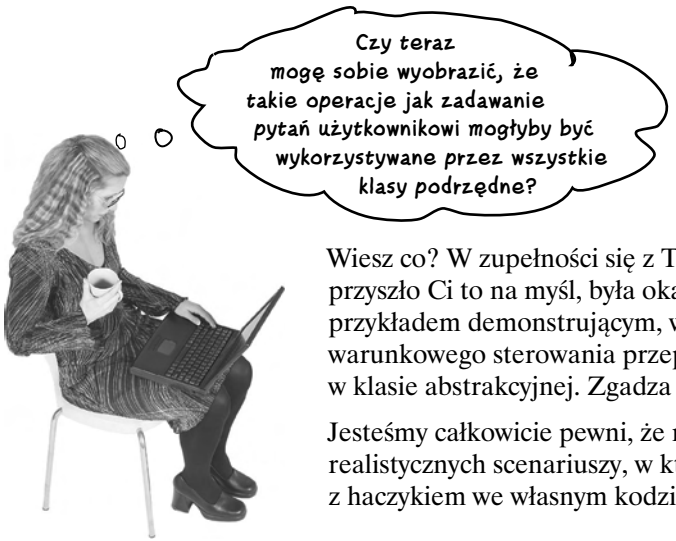
Przyrządzam wspaniałą, gorącą herbatę...
Gotowanie wody
Zaparzanie herbaty
Nalewanie do filiżanki
Czy życzy Pan sobie dodania cytryny do herbaty (t/n)? t
Dodawanie cytryny


A teraz pora na gorącą, aromatyczną kawę...
Gotowanie wody
Zaparzanie i przesączanie kawy przez filtr
Nalewanie do filiżanki
Czy życzy Pan sobie dodania cukru i mleka do kawy (t/n)? n

>
```

Pełna filiżanka gorącej herbaty i tak, oczywiście, że chcemy cytrynę!

Wspaniała filiżanka gorącej, aromatycznej kawy, więc nie będziemy jej psuć zbędnymi dodatkami.



Wiesz co? W zupełności się z Tobą zgadzamy. Ale musisz przyznać, że zanim przyszło Ci to na myśl, była okazja zapoznania się z naszym całkiem fajnym przykładem demonstrującym, w jaki sposób haczyk może być użyty do warunkowego sterowania przepływem sterowania w danym algorytmie w klasie abstrakcyjnej. Zgadza się?

Jesteśmy całkowicie pewni, że możesz znaleźć jeszcze dziesiątki innych, realistycznych scenariuszy, w których da się zastosować metodę szablonową z haczykiem we własnym kodzie.

P: Kiedy tworzę metodę szablonową, skąd mam wiedzieć, kiedy użyć metody abstrakcyjnej, a kiedy haczyka?

O: Z metod abstrakcyjnych powinieneś korzystać zawsze wtedy, kiedy klasa podrzędna MUSI dostarczyć implementacji danej metody bądź danego kroku realizowanego algorytmu. Z haczyków powinieneś korzystać w sytuacjach, gdy pewne części algorytmu są opcjonalne. Jeżeli użyjesz metody-haczyka, klasa podrzędna może przesłonić tę metodę, ale wcale robić tego nie musi.

P: Jakie jest tak naprawdę przeznaczenie haczyków?

O: Generalnie haczyki mają kilka podstawowych zastosowań. Jak już wspominaliśmy, haczyk może stanowić dla klasy podrzędnej sposób na implementację opcjonalnej części algorytmu, aczkolwiek jeżeli dla danej podklasy jest on zbędny, może ona spokojnie go pominąć. Innym zastosowaniem haczyków jest danie klasie podrzędnej szansy wcześniejszego reagowania na pewne kroki algorytmu w metodzie szablonowej, które za moment będą realizowane bądź których realizacja właśnie się zakończyła. Przykładowo, metoda-haczyk listaPrzestawiona() umożliwia klasie podrzędnej wykonanie pewnych określonych operacji (takich jak aktualizacja danych wyświetlanych na ekranie) po zmianie kolejności poszczególnych elementów na hipotetycznej wewnętrznej liście danych. Jak pokazano w jednym z przykładów, zastosowanie odpowiedniego haczyka daje klasie podrzędnej również możliwość podejmowania decyzji właściwych dla klasy nadrzędnej.

Nie istnieją głupie pytania

P: Czy klasa podrzędna musi implementować wszystkie metody abstrakcyjne, które zostały zadeklarowane w klasie abstrakcyjnej?

O: Tak, każda podrzędna klasa rzeczywista musi definiować cały zestaw metod abstrakcyjnych klasy nadrzędnej, zapewniając w ten sposób pełną implementację tych kroków algorytmu, które nie zostały zdefiniowane w metodzie szablonowej.

P: Wygląda na to, że powinienem starać się utrzymywać stosunkowo niewielką liczbę metod abstrakcyjnych, w przeciwnym wypadku będę miał bardzo dużo pracy z implementowaniem ich w klasach podrzędnych.

O: To jest coś, o czym warto pamiętać podczas tworzenia metod szablonowych. Często można osiągnąć całkiem dobre rezultaty, unikając nadmiernego rozdrabniania algorytmu na poszczególne elementy składowe. Ale, jak zwykle, jest to pewien kompromis: im mniejsze rozdrobnienie algorytmu, tym mniejsza jego elastyczność.

Pamiętaj również, że niektóre części algorytmu mogą być opcjonalne; w takiej sytuacji powinny zostać zaimplementowane raczej jako haczyki, a nie metody abstrakcyjne, zmniejszając tym samym obciążenie klas podrzędnych.

Reguła Hollywood

Mamy dla Ciebie kolejną regułę projektowania; powszechnie jest ona nazywana *regułą Hollywood:*



Reguła Hollywood

Nie dzwoń do nas, my zadzwonimy do Ciebie.

Słyszałeś to już podczas naszej poprzedniej rozmowy, ale powtórzę Ci to jeszcze raz: nie dzwoń do mnie, ja zadzwonię do Ciebie!



Łatwo zapamiętać, prawda? Ale co to ma wspólnego z programowaniem zorientowanym obiektowo?

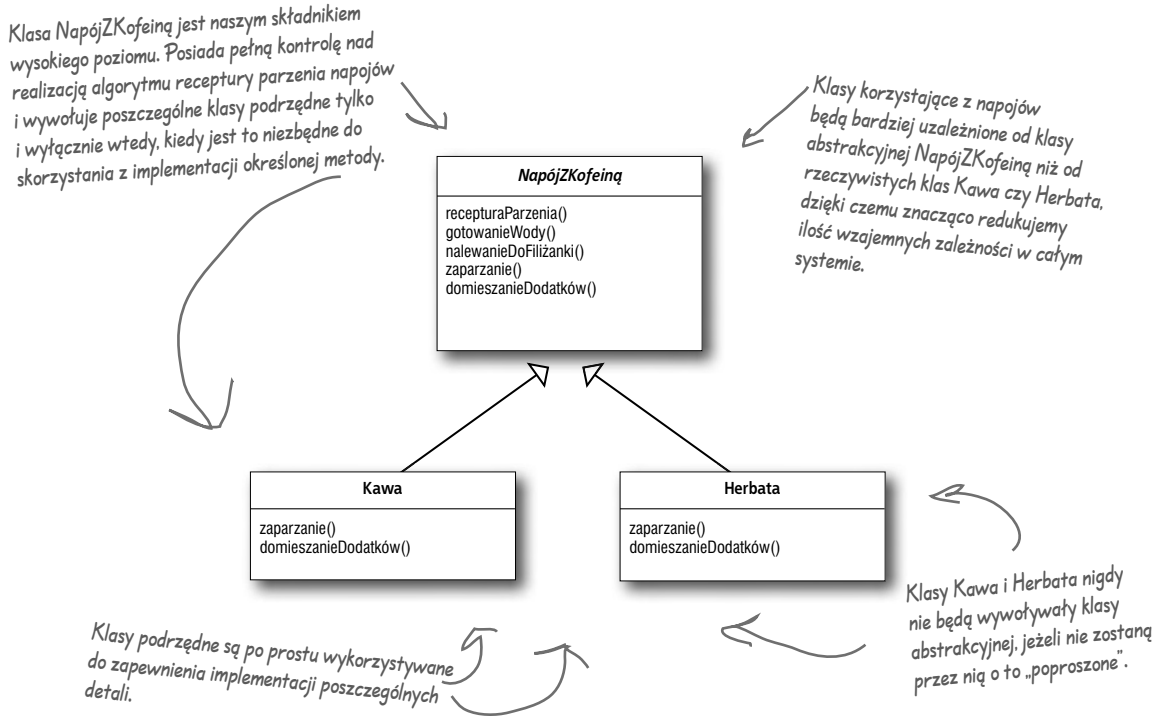
Reguła Hollywood pozwala nam uniknąć zjawiska niezbyt elegancko nazywanego „butwieniem drzewa zależności". Występuje ono wtedy, kiedy w systemie pojawiają się składniki wysokiego poziomu, uzależnione od określonych, innych składników wysokiego poziomu, które z kolei są uzależnione od pewnych składników bocznych poziomów, zależnych z kolei od jeszcze innych składników na niskim poziomie i tak dalej. Kiedy rozpoczyna się proces butwienia, nikt już nie jest w stanie łatwo zorientować się, jak właściwie dana aplikacja została zaprojektowana.

Wdrażając w życie regułę Hollywood, pozwalamy składnikom niskiego poziomu egzystować gdzieś w systemie, ale to składniki wysokiego poziomu decydują o tym, kiedy i w jaki sposób z nich korzystają. Innymi słowy, składniki wysokiego poziomu traktują składniki niskiego poziomu tak, jak to często ma miejsce w Hollywood: „Nie dzwoń do nas, to my zadzwonimy do Ciebie".



Reguła Hollywood a wzorzec Metoda Szablonowa

Powiązania pomiędzy regułą Hollywood a wzorcem Metoda Szablonowa są prawdopodobnie dla wszystkich dosyć oczywiste: kiedy tworzymy projekt zgodnie z założeniami wzorca Metoda Szablonowa, tym samym mówimy klasom podrzędnym: „Nie dzwoń do nas, to my zadzwonimy do Ciebie”. Jak to się dzieje? Rzućmy jeszcze raz okiem na diagram klas NapójZKofeiną, Herbata i Kawa:





WYTĘŻ UMYSŁ

Jakie inne wzorce również korzystają z reguły Hollywood?

Odp.: Wzorce Metoda Fabrykująca i Obserwator; czy jeszcze jakieś inne?

Nie istnieją głupie pytania

P: Jak reguła Hollywood ma się do reguły odwracania zależności, o której mówiliśmy kilka rozdziałów temu?

O: Reguła odwracania zależności zaleca, aby w miarę możliwości unikać korzystania z klas rzeczywistych i w ich miejsce używać abstrakcji. Reguła Hollywood jest techniką, która pozwala na budowę struktury systemu w taki sposób, że składniki niskiego poziomu mogą brać bezpośredni udział w przetwarzaniu danych, ale bez tworzenia zależności pomiędzy sobą i bez zależności od składników wysokiego poziomu. Jak widać, obie reguły prowadzą do usuwania sprzężeń, aczkolwiek nie da się ukryć, że to reguła odwracania zależności sugeruje o wiele bardziej zdecydowany sposób unikania zależności w projektowanym systemie.

Reguła Hollywood daje nam do ręki narzędzie umożliwiające tworzenie projektów, w których struktury niskiego poziomu mogą ze sobą współpracować, ale jednocześnie zapobiegają temu, aby inne klasy zbyt mocno się od nich uzależniły.

P: Czy składniki niskiego poziomu nie mogą w żaden sposób wywoływać metod składników wysokiego poziomu?

O: Nie tak do końca. W rzeczywistości bardzo często zdarza się, że składnik niskiego poziomu wywołuje metodę zdefiniowaną w jego klasie nadrzędnej, co wynika wprost z mechanizmu dziedziczenia, aczkolwiek naszym głównym celem jest unikanie tworzenia zależności wzajemnych pomiędzy składnikami niskiego i wysokiego poziomu.

Kto co robi?

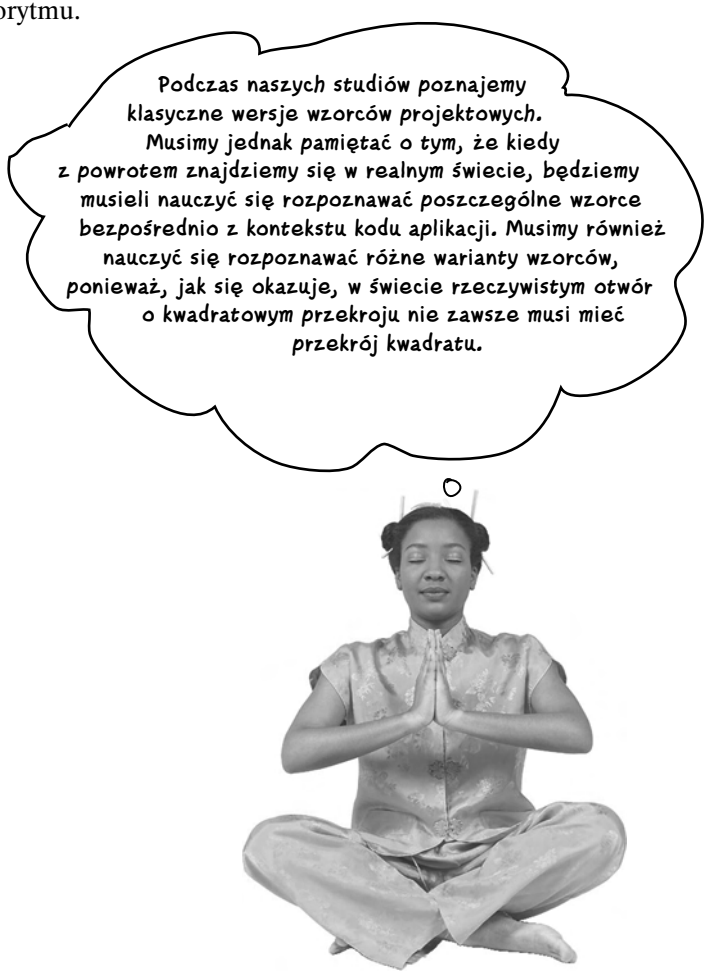
Dopasuj nazwy poszczególnych wzorców do odpowiednich opisów.

Wzorzec	Opis
Metoda Szablonowa	Hermetyzuje zachowania wymienne i wykorzystuje mechanizm delegacji do wyboru, które zachowanie zostanie użyte.
Strategia	Klasy podrzędne decydują o tym, w jaki sposób zostaną zaimplementowane poszczególne kroki algorytmu.
Metoda Fabrykująca	Klasy podrzędne decydują o tym, jakiej klasy rzeczywistej obiekt zostanie utworzony.

Wzorzec Metoda Szablonowa w głębokiej kniei...

Wzorzec Metoda Szablonowa jest bardzo popularnym i szeroko stosowanym wzorcem projektowym, stąd w głębokim gąszczu kodów źródłowych aplikacji z pewnością napotkasz wiele z nich. Czasami musisz jednak dysponować niemal sokolim wzrokiem, ponieważ istnieje wiele implementacji metod szablonowych, które wyglądają zupełnie inaczej niż klasyczne rozwiązania tego wzorca spotykane w podręcznikach.

Ogromna popularność tego wzorca wynika stąd, że jest to po prostu znakomite narzędzie projektowe służące do tworzenia takich struktur aplikacji, w których sama struktura kontroluje co prawda sposób realizacji określonego algorytmu, ale pozostawia Ci (czyli osobie jej używającej) pełne pole manewru w definiowaniu szczegółów implementacji poszczególnych kroków tego algorytmu.



Sortowanie przy użyciu wzorca Metoda Szablonowa



Jaka jest jedna z najczęściej wykonywanych na tablicach operacji?

Oczywiście, ich sortowanie!

Ponieważ projektanci klasy Arrays zdawali sobie z tego sprawę, wyposażyli ją w bardzo wygodną metodę szablonową, służącą właśnie do celów sortowania zawartości tablicy. Rzućmy okiem, w jaki sposób ta metoda funkcjonuje:

W celu zwiększenia przejrzystości kodu dokonaliśmy jego pewnego odchudzenia. Jeżeli koniecznie chcesz zobaczyć cały kod, pobierz pliki źródłowe bezpośrednio z witryny internetowej firmy Sun…

Mamy tutaj dwie metody, które dzięki wzajemnej współpracy zapewniają proces sortowania.

Pierwsza metoda, sort(), jest w zasadzie metodą pomocniczą, która tworzy kopię tablicy i przekazuje ją do miejsca przeznaczenia jako docelową tablicę metody mergeSort(). Metoda sort() przekazuje również rozmiar tablicy oraz informację, że sortowanie powinno się rozpocząć, począwszy od pierwszego elementu tablicy.

```
public static void sort(Object[]a){
    Object aux[] = (Object[])a.clone();
    mergeSort(aux, a, 0, a.length, 0);
}
```

Metoda mergeSort() posiada zaimplementowany algorytm sortowania, a powodzenie funkcjonowania samego algorytmu zależy od implementacji metody compareTo().

*Pomyśl o tym jako o **metodzie szablonowej**.*

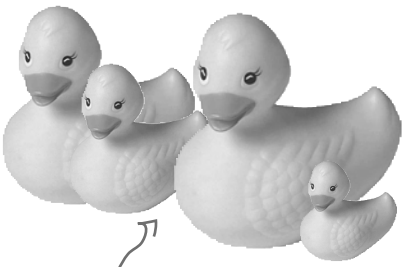
```
private static void mergeSort(Object src[], Object dest[],
                int low, int high, int off)
{
    for (int i=low; i<high; i++) {
        for (int j=i; j>low &&
            ((Comparable)dest[j-1]).compareTo((Comparable)dest[j])>0; j--)
        {
            swap(dest, j, j-1);
        }
    }
    return;
}
```

To jest metoda rzeczywista, która została już zdefiniowana w klasie Arrays.

Aby „wypełnić" szablon wzorca Template Method, musimy zaimplementować metodę compareTo().

A teraz musimy posortować trochę kaczek...

Załóżmy, że jesteś szczęśliwym posiadaczem tablicy pełnej ślicznych, żółtych kaczuszek, które musisz posortować. Jak masz zamiar tego dokonać? No cóż, metoda szablonowa sort klasy Arrays daje nam algorytm, ale mimo to i tak musisz jej powiedzieć, w jaki sposób porównywać kaczki. Można tego dokonać, odpowiednio implementując metodę compareTo()… Czy to brzmi rozsądnie?



Mamy tutaj tablicę pełną kaczek, które musimy posortować.



Niestety, chyba nie. Czy przypadkiem nie mieliśmy tutaj tworzyć jakichś klas podrzędnych? Myślałam, że właśnie o to chodzi w tym wzorcu Metoda Szablonowa. Nie mamy tutaj żadnego dziedziczenia, więc nie wiem, jak możemy korzystać z metody sort().

Dobre pytanie. Spieszymy wyjaśnić: projektanci metody sort() chcieli, aby można było z niej korzystać w odniesieniu do wszystkich tablic. Musieli więc zadeklarować ją jako metodę statyczną, co powoduje, że można jej używać z dowolnego miejsca. Ale takie rozwiązanie jest jak najbardziej w porządku, metoda funkcjonuje niemal tak samo, jakby była zdefiniowana w superklasie. Mamy tutaj jeszcze jeden szczegół: ponieważ metoda sort() nie jest tak naprawdę zdefiniowana w naszej superklasie, musi ona wiedzieć, że zaimplementowałeś już metodę compareTo(); w przeciwnym wypadku nie masz kluczowego fragmentu kodu niezbędnego do funkcjonowania algorytmu sortowania.

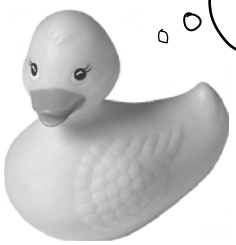
Aby się z tym uporać, projektanci wykorzystali interfejs Comparable. A zatem wszystko, co musisz zrobić, polega na zaimplementowaniu tego interfejsu, który posiada tylko jedną metodę (uwaga, niespodzianka!): compareTo().

Czym jest metoda compareTo()?

Metoda compareTo() porównuje dwa obiekty i jako wynik zwraca informację, czy jeden z nich jest mniejszy, równy lub większy od drugiego. Metoda sort() wykorzystuje ją w charakterze podstawowego mechanizmu porównywania obiektów w sortowanej tablicy.

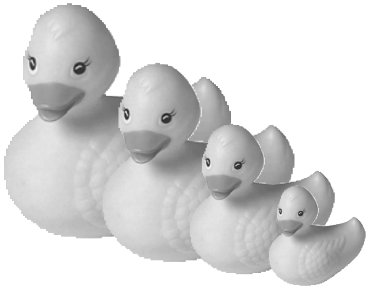


Czy jestem od Ciebie większa?



Nie wiem, o tym właśnie ma nas powiadamiać metoda compareTo().

Porównywanie kaczek z innymi kaczkami



No dobrze, a zatem już wiesz, że jeżeli chcesz sortować kaczki, musisz zaimplementować metodę compareTo(); dzięki temu dasz klasie Arrays narzędzie niezbędne do poprawnego funkcjonowania algorytmu sortowania.

Poniżej zamieściliśmy implementację klasy Kaczka:

Pamiętaj, ponieważ nie korzystamy z mechanizmu dziedziczenia, musimy zaimplementować interfejs Comparable.

```
public class Kaczka implements Comparable {
    String nazwa;
    int waga;

    public Kaczka(String nazwa, int waga) {
        this.nazwa = nazwa;
        this.waga = waga;
    }

    public String toString() {
        return nazwa + " waży " + waga;
    }

    public int compareTo(Object object) {

        Kaczka innaKaczka = (Kaczka)object;

        if (this.waga < innaKaczka.waga) {
            return -1;
        } else if (this.waga == innaKaczka.waga) {
            return 0;
        } else { // this.waga > innaKaczka.waga
            return 1;
        }
    }
}
```

Nasze kaczki posiadają nazwy i wagę.

Staramy się utrzymywać prostotę projektu; nasze kaczki będą jedynie drukować na ekranie swoje nazwy i wagi.

Tutaj mamy to, czego potrzebuje metoda sort().

Metoda compareTo() pobiera kolejną kaczkę i porównuje ją z bieżącą kaczką.

W tym miejscu określamy, w jaki sposób mają być porównywane nasze Kaczki. Jeżeli dana kaczka waży mniej niż inna, zwracana jest wartość -1, jeżeli ważą tyle samo, zwracamy wartość 0, a jeżeli dana kaczka waży więcej niż inna, zwracamy wartość 1.

No to posortujmy sobie trochę Kaczek

Oto kod środowiska testowego, które pozwoli nam na testowanie sortowania Kaczek...

```
public class TestSortowaniaKaczek {

    public static void main(String[] args) {
        Kaczka[] kaczki = {
                          new Kaczka("Kaczor Daffy", 8),
                          new Kaczka("Kaczor Dewey", 2),
                          new Kaczka("Kaczor Howard", 7),
                          new Kaczka("Kaczor Louie", 2),
                          new Kaczka("Kaczor Donald", 10),
                          new Kaczka("Kaczor Huey", 2)
                          };

        System.out.println("Przed sortowaniem:");
        wyświetlaj(kaczki);

        Arrays.sort(kaczki);

        System.out.println("\nPo zakończeniu sortowania:");
        wyświetlaj(kaczki);
    }

    public static void wyświetlaj(Kaczka[] kaczki) {
        for (int i = 0; i < kaczki.length; i++) {
            System.out.println(kaczki[i]);
        }
    }
}
```

Potrzebna nam jest tablica pełna Kaczek, ale Kaczory też wyglądają dobrze.

Zwróć uwagę, że wywołujemy statyczną metodę sort() klasy Arrays i przekazujemy jej stado naszych Kaczek (a w zasadzie Kaczorów...).

Wydrukujemy informacje o nazwach poszczególnych ptaszysk i ich wagach.

Nadszedł czas na sortowanie!

I znów wydrukujemy informacje o nazwach poszczególnych ptaszysk i ich wagach.

Niech się stanie sortowanie!

```
C:\WINDOWS\System32\cmd.exe

> java TestSortowaniaKaczek

Przed sortowaniem:
Kaczor Duffy waży 8
Kaczor Dewey waży 2
Kaczor Howard waży 7
Kaczor Louie waży 2
Kaczor Donald waży 10
Kaczor Huey waży 2

Po zakończeniu sortowania:
Kaczor Dewey waży 2
Kaczor Louie waży 2
Kaczor Huey waży 2
Kaczor Howard waży 7
Kaczor Duffy waży 8
Kaczor Donald waży 10

>
```

Kaczory przed sortowaniem

Kaczory po sortowaniu

Robimy maszynę do sortowania kaczek



Prześledzimy teraz sposób działania metody szablonowej sort() klasy Arrays. Zobaczymy, w jaki sposób metoda ta kontroluje działanie algorytmu oraz jak w określonych punktach algorytmu możemy poprosić nasze Kaczki o pomoc w implementacji algorytmu...

```
for (int i=low; i<high; i++){
    ... compareTo() ...
    ... swap() ...
}
```

1. Najpierw musimy utworzyć tablicę Kaczek:

```
Kaczka[] kaczki = { new Kaczka("Kaczor Daffy", 8), ... };
```

2. Następnie wywołujemy metodę szablonową sort() klasy Arrays i przekazujemy do niej nasze poczciwe ptaszyska:

```
Arrays.sort(kaczki);
```

Metoda sort() kontroluje sposób funkcjonowania algorytmu i żadna klasa nie może tego zmienić. Implementacja metody compareTo() została tutaj osiągnięta poprzez zastosowanie interfejsu Comparable.

Metoda sort() (wraz ze swoją metodą pomocniczą, mergeSort()) kontrolują przebieg operacji sortowania.

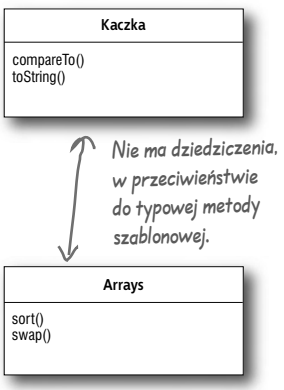
3. Aby posortować tablicę, musimy porównywać ze sobą każde kolejne dwa elementy dopóty, dopóki tablica nie zostanie posortowana.

Kiedy przychodzi do samego procesu porównywania kaczek, metoda sort() całkowicie zdaje się na klasę Kaczka i jej metodę compareTo(). Metoda compareTo() jest wywoływana na obiekcie reprezentującym pierwszą kaczkę i jako parametr wywołania otrzymuje kolejną kaczkę, z którą ma być porównywana:

```
kaczki[0].compareTo(kaczki[1]);
```

Pierwsza Kaczka

Druga Kaczka do porównania



4. Jeżeli Kaczki nie są jeszcze ułożone w prawidłowej kolejności, są zamieniane miejscami; dokonuje tego metoda swap() klasy Arrays:

```
swap()
```

5. Metoda sort() kontynuuje proces porównywania i zamiany miejscami kaczek dopóty, dopóki cała zawartość tablicy nie zostanie posortowana poprawnie!

Nie istnieją głupie pytania

P: Czy to jest naprawdę wzorzec Metoda Szablonowa, czy po prostu usiłujecie coś naciągnąć?

O: Wzorzec wywołuje odpowiednie metody realizujące poszczególne kroki algorytmu i pozwala na implementację tych kroków w klasach podrzędnych — jak widać, metoda sort() klasy Arrays ewidentnie tego nie robi! Ale jak wiemy, wzorce w świecie rzeczywistym niekoniecznie muszą wyglądać tak, jak ich podręcznikowe odpowiedniki, gdyż zazwyczaj muszą być odpowiednio modyfikowane, aby spełnić wymagania i ograniczenia danego projektu.

Projektanci metody sort() klasy Arrays musieli się borykać z kilkoma poważnymi ograniczeniami. Ogólnie rzecz biorąc, w języku Java nie ma możliwości dziedziczenia tablic, a oni chcieli, aby metoda sort() mogła być używana w odniesieniu do wszystkich tablic (a każda tablica to inna klasa). Aby tego dokonać, utworzyli statyczną metodę sort() i pozostawili w rękach programisty tę część algorytmu, która odpowiada za bezpośrednie porównywanie sortowanych elementów.

Z tego powodu, pomimo iż takie rozwiązanie niezbyt przypomina swój podręcznikowy pierwowzór, jego implementacja jest nadal utrzymana w duchu wzorca Metoda Szablonowa. Dodatkowo, dzięki temu, że aby użyć tego algorytmu, nie trzeba korzystać z mechanizmu dziedziczenia i tworzyć żadnych klas podrzędnych, udało im się stworzyć bardzo elastyczną, a przez to jeszcze bardziej użyteczną metodę sortowania.

P: Taka implementacja algorytmu sortowania wygląda bardziej jak wzorzec Strategia niż Metoda Szablonowa. Dlaczego mimo to zaliczamy ją do kategorii Metoda Szablonowa?

O: Prawdopodobnie myślisz tak dlatego, że wzorzec Strategia korzysta z kompozycji obiektów. Do pewnego stopnia masz tutaj rację — do sortowania naszej tablicy *używamy* obiektu klasy Arrays i z tego powodu całość nieco przypomina wzorzec Strategia. Ale pamiętaj, we wzorcu Strategia klasa, której używamy do kompozycji, implementuje *cały* algorytm, natomiast algorytm, który implementuje klasa Arrays, jest niekompletny; do jego realizacji niezbędna jest dodatkowa klasa, która implementuje brakującą metodę compareTo() — i właśnie z tego powodu to rozwiązanie jest bliższe wzorcowi Metoda Szablonowa.

P: Czy w bibliotece Java API są jeszcze inne przykłady zastosowania metod szablonowych?

O: Tak, znajdziesz je jeszcze w kilku miejscach. Przykładowo, w pakiecie java.io istnieje metoda read() klasy InputStream, która musi być implementowana w klasach podrzędnych i jest wykorzystywana przez metodę szablonową read(byte b[], int off, int len).



WYTĘŻ UMYSŁ

Pamiętasz, że jedna z reguł programowania obiektowego zaleca przedkładanie kompozycji nad dziedziczenie, prawda? Jak widać, projektanci metody sort() zdecydowali się nie korzystać z dziedziczenia i zamiast tego zaimplementowali ją jako metodę statyczną komponowaną w czasie działania programu z interfejsem Comparable. Czy takie rozwiązanie jest lepsze? A może jest jednak gorsze? Jak Ty rozwiązałbyś ten problem? Czy specyfika tablic w języku Java powoduje, że trzeba tutaj używać różnych sztuczek?



WYTĘŻ UMYSŁ

Pomyśl o innym wzorcu projektowym, który jest szczególnym przypadkiem metody szablonowej. W jego implementacji operacje elementarne są wykorzystywane do tworzenia i zwracania obiektów. Jaki to wzorzec?

Zabawy z ramkami



Kolejną przygodą, z którą zmierzymy się podczas naszego „wzorcowego" safari, będzie… spotkanie ze swingującymi ramkami JFrame!

Jeżeli do tej pory nie miałeś okazji spotkać się z klasą JFrame, powiemy tylko pokrótce, że jest to najbardziej podstawowy kontener pakietu Swing, który dziedziczy metodę paint(). Domyślnie metoda ta nie realizuje żadnej operacji, ponieważ jest to nasz dobrze już znany ***haczyk***! Przesłaniając metodę paint(), możemy „włączyć się" do algorytmu wyświetlania realizowanego przez klasę JFrame i wykorzystać ją do rysowania swoich własnych elementów graficznych. Poniżej zamieściliśmy aż przesadnie prosty przykład zastosowania klasy JFrame do przesłonięcia haczyka, czyli metody paint(), i wykorzystania jej do własnych potrzeb:

Dziedziczymy po klasie JFrame, która posiada metodę update() kontrolującą działanie algorytmu obsługującego odświeżanie ekranu graficznego. Możemy się „podczepić" do tego algorytmu, przesłaniając nasz haczyk, czyli metodę paint().

```
public class MojaRamka extends JFrame {

    public MojaRamka(String title) {
        super(title);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        this.setSize(300,300);
        this.setVisible(true);
    }

    public void paint(Graphics graphics) {
        super.paint(graphics);
        String msg = "Teraz moje tu będą rządy!!";
        graphics.drawString(msg, 100, 100);
    }

    public static void main(String[] args) {
        MojaRamka MojaRamka = new MojaRamka("Wzorce projektowe. Rusz głową!");
    }
}
```

Nie zaglądaj za kurtynę! Mamy tutaj nieco operacji niezbędnych do inicjalizacji tego całego ustrojstwa…

Zaimplementowany w klasie JFrame algorytm odświeżania ekranu graficznego wywołuje metodę paint(). Jak już wiesz, domyślnie ta metoda nie robi nic… to tylko haczyk. Przesłaniamy zatem metodę paint() i dzięki temu nakazujemy klasie JFrame wyświetlać na ekranie graficznym naszą wiadomość.



Oto wiadomość, która zostaje wyświetlona w naszej ramce, ponieważ udało nam się „podczepić" do metody paint().

Aplety Java



Powoli zbliżamy się do ostatniego przystanku na trasie naszego safari: do apletów Java.

Prawdopodobnie doskonale wiesz o tym, że aplet Java to mały programik, który działa wewnątrz strony WWW. Każdy aplet musi dziedziczyć po klasie Applet i, jak się można było tego spodziewać, również ta klasa posiada kilka haczyków. Rzućmy okiem na kilka z nich:

```
public class MójApplet extends Applet {
    String komunikat;

    public void init() {
        komunikat = "Witajcie Ziemianie! Jestem tutaj!";
        repaint();
    }

    public void start() {
        komunikat = "Teraz się uruchamiam...";
        repaint();
    }

    public void stop() {
        komunikat = "Oooops, zaraz zostanę zatrzymany...";
        repaint();
    }

    public void destroy() {
        komunikat = "Żegnajcie, okrutni Ziemianie!";
        repaint();
    }

    public void paint(Graphics g) {
        g.drawString(komunikat, 5, 15);
    }
}
```

Pierwszy haczyk, metoda init(), pozwala apletowi wykonać, co tylko mu się żywnie podoba, podczas pierwszego uruchomienia.

Metoda repaint() jest metodą rzeczywistą zdefiniowaną w klasie Applet, informującą składniki wysokiego poziomu, że aplet musi zostać przerysowany.

Kolejny haczyk, metoda start(), pozwala apletowi na wykonanie danych operacji na chwilę przed tym, zanim zostanie wyświetlony na stronie WWW.

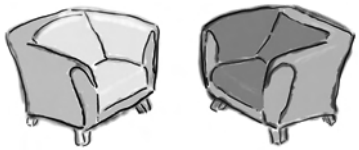
Jeżeli użytkownik przechodzi na kolejną stronę WWW, można użyć haczyka stop(), który pozwala apletowi na wykonanie operacji niezbędnych do zatrzymania jego pracy.

Wreszcie haczyk destroy() może być wykorzystywany w sytuacji, kiedy aplet musi zostać zniszczony, na przykład w sytuacji, kiedy użytkownik zamyka dane okno przeglądarki. Możemy tutaj wyświetlić jakiś komunikat, ale czy to ma sens?

Patrzcie, patrzcie! Kogo my tutaj mamy — nasz stary przyjaciel, metoda paint()! Jak widać, aplety również korzystają z tego haczyka.

W zastosowaniach praktycznych aplety bardzo często korzystają z haczyków do obsługi swoich własnych zachowań. Ponieważ jednak wymienione metody zostały zaimplementowane jako haczyki, aplety mogą, ale nie muszą z nich korzystać.

Pogawędki przy kominku



Temat dzisiejszej wieczornej pogawędki:

wzorce Metoda Szablonowa oraz Strategia porównują swoje metody

Metoda Szablonowa

Hej, Strategia, co Ty robisz w moim rozdziale? Myślałem, że utknę tu z kimś tak nudnym jak wzorzec Fabryka!

No dobrze, tylko żartowałem! Ale poważnie, co Ty tu robisz? Nie mieliśmy już z Tobą kontaktu od jakichś ośmiu rozdziałów!

To może przypomnisz coś o sobie Czytelnikom, bo w końcu tak długo Cię nie było….

Ha, Twój opis brzmi bardzo podobnie do tego, co ja robię! Ale moje przeznaczenie jest nieco inne niż Twoje: moim zadaniem jest zdefiniowanie szkieletu algorytmu, ale pozwalam jednak klasom podrzędnym wykonać nieco pracy. W ten sposób mogę mieć różne implementacje poszczególnych kroków algorytmu, ale nadal mimo to zachowuję pełną kontrolę nad jego działaniem. A wygląda na to, że Ty już całkiem podarowałeś sobie kontrolę nad swoimi algorytmami.

Strategia



To naprawdę ja, ale uważaj, co mówisz — w końcu Ty i wzorzec Fabryka jesteście spokrewnieni, nieprawdaż?

Słyszałem, że zbliżasz się już do końca swojego rozdziału i po prostu pomyślałem sobie, że wpadnę zobaczyć, jak sobie radzisz. Mamy ze sobą wiele wspólnego, więc być może będę mógł w czymś pomóc…

No, nie wiem, bo od czasu zakończenia pierwszego rozdziału ludzie często zatrzymują mnie na ulicy i mówią: „Przepraszam, czy Ty przypadkiem nie jesteś tym wzorcem…?”. Wnioskuję stąd, że jednak wiedzą, kim jestem. Ale z szacunku do Ciebie powiem jednak, że jestem wzorcem, który definiuje rodziny algorytmów i powoduje, że są wzajemnie wymienne. Ponieważ każdy algorytm jest odpowiednio hermetyzowany, każdy klient może łatwo korzystać z różnych algorytmów.

No, ja bym tego w ***ten*** sposób nie ujął… ale niech tam, przynajmniej do uzyskania implementacji algorytmów nie muszę się bawić w dziedziczenie. Moim klientom oferuję pełną gamę dobrze hermetyzowanych algorytmów dostępnych poprzez kompozycję.

Metoda Szablonowa

A tak, pamiętam. Ale musisz przyznać, że ja mam większą kontrolę nad moimi algorytmami, a co więcej, nie powoduję niepotrzebnej duplikacji kodu. W praktyce każda część mojego algorytmu jest taka sama — za wyjątkiem, powiedzmy, jednego wiersza; dzięki temu moje klasy są zdecydowanie bardziej wydajne niż Twoje. Cały kod duplikowany jest umieszczony w klasie nadrzędnej, więc wszystkie klasy podrzędne mogą z niego wspólnie korzystać.

No cóż, *bardzo* się cieszę Twoim szczęściem, ale nie zapominaj, że to ja jestem najczęściej wykorzystywanym wzorcem. Dlaczego? Ponieważ zapewniam najbardziej fundamentalną metodę wielokrotnego wykorzystywania kodu, który pozwala klasom podrzędnym na implementację specyficznych zachowań. Jestem pewny, że sam przyznasz, że jest to idealne rozwiązanie do tworzenia struktury aplikacji.

A to dlaczego? Przecież moja klasa bazowa jest klasą abstrakcyjną.

Oj, Strategia, już powiedziałem, naprawdę *bardzo* się cieszę Twoim szczęściem. Dzięki, że wpadłaś, ale teraz mam jeszcze kilka spraw do skończenia w tym rozdziale.

Spoko! Ale nie dzwoń więcej, to my zadzwonimy do Ciebie…

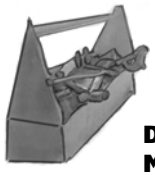
Strategia

Być może jesteś nieco bardziej wydajny (ale tylko nieco) i wymagasz nieco mniejszej liczby obiektów. W porównaniu do mojego modelu delegacji zachowań możesz być również nieco mniej skomplikowany, ale za to ja jestem z pewnością bardziej elastyczny, ponieważ korzystam z mechanizmu kompozycji obiektów. Dzięki mnie klient może zmieniać swoje algorytmy w czasie działania programu, po prostu wybierając inną instancję mojego obiektu. No, daj spokój, w końcu nie bez powodu to ja zostałem wybrany do pierwszego rozdziału!

No, pewnie tak… ale co z dziedziczeniem? W końcu Ty posiadasz o wiele więcej zależności niż ja.

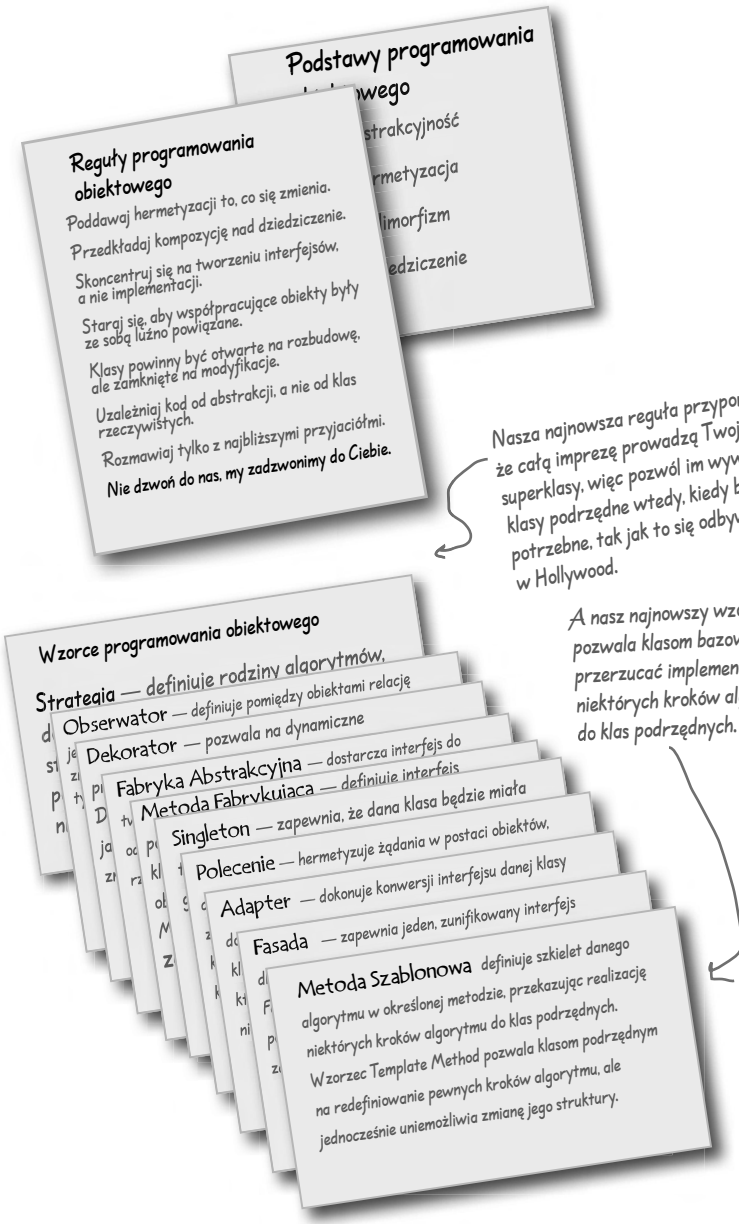
No tak, ale musisz przecież polegać na metodach implementowanych w klasie bazowej, które są częścią Twojego algorytmu. Ja nie jestem uzależniony od nikogo; mogę samodzielnie zrealizować cały algorytm!

Dobrze, już dobrze, nie bądź taki drażliwy. Przecież pozwolę Ci popracować, ale jeżeli kiedykolwiek będziesz potrzebował moich specjalnych umiejętności, zawsze z przyjemnością Ci pomogę.



Twoja skrzynka narzędziowa

Do Twojej skrzynki narzędziowej dodaliśmy wzorzec Metoda Szablonowa. Korzystając z niego, możesz wielokrotnie wykorzystywać te same fragmenty kodu (jak zawodowiec) i nadal utrzymywać pełną kontrolę nad działaniem algorytmów.



CELNE SPOSTRZEŻENIA

- „Metoda szablonowa" definiuje poszczególne kroki algorytmu, przerzucając implementację wybranych kroków do klas podrzędnych.
- Wzorzec Metoda Szablonowa udostępnia bardzo ważną technikę umożliwiającą wielokrotne wykorzystywanie tego samego kodu.
- Klasy abstrakcyjne wzorca Metoda Szablonowa mogą definiować metody rzeczywiste, metody abstrakcyjne oraz metody-haczyki.
- Metody abstrakcyjne są implementowane w klasach podrzędnych.
- Haczyki to metody, które nie robią nic bądź które mają pewne określone zachowania domyślne w klasach abstrakcyjnych, ale mogą być jednak przesłaniane przez klasy podrzędne.
- Aby klasy podrzędne nie mogły modyfikować sposobu działania algorytmu, metoda szablonowa powinna być zadeklarowana jako metoda typu final.
- Reguła Hollywood sugeruje, aby proces podejmowania decyzji był umieszczony w modułach wysokiego poziomu, które mogą samodzielnie decydować, jak i kiedy wywoływać moduły niskiego poziomu.
- W kodzie rzeczywistych aplikacji znajdziesz wiele przykładów zastosowania wzorca Metoda Szablonowa, aczkolwiek nie możesz oczekiwać, że wszystkie rozwiązania tego wzorca (dotyczy to zresztą i pozostałych wzorców) będą wyglądały tak, jak ich książkowe pierwowzory.
- Zarówno wzorzec Strategia, jak i Metoda Szablonowa dokonują hermetyzacji algorytmów, odpowiednio wykorzystując kompozycję i dziedziczenie.
- Wzorzec Metoda Fabrykująca jest szczególnym przypadkiem wzorca Metoda Szablonowa.

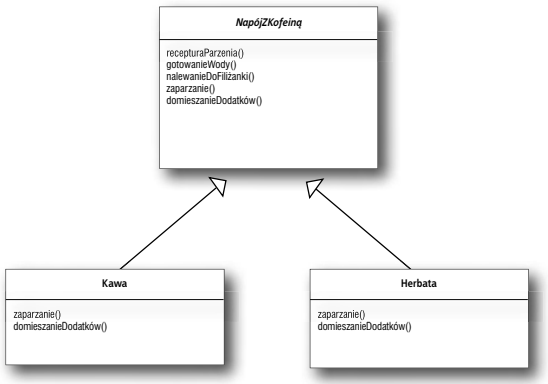


Rozwiązania ćwiczeń



Zaostrz ołówek

Twoim zadaniem będzie narysowanie nowego diagramu klas odzwierciedlającego przeniesienie implementacji metody recepturaParzenia() do klasy nadrzędnej NapójZKofeiną:



Kto co robi

Dopasuj nazwy poszczególnych wzorców do odpowiednich opisów.

Wzorzec	Opis
Metoda Szablonowa	Hermetyzuje zachowania wymienne i wykorzystuje mechanizm delegacji do wyboru, które zachowanie zostanie użyte.
Strategia	Klasy podrzędne decydują o tym, w jaki sposób zostaną zaimplementowane poszczególne kroki algorytmu.
Metoda Fabrykująca	Klasy podrzędne decydują o tym, jakiej klasy rzeczywistej obiekt zostanie utworzony.

Zarządzanie kolekcjami



Jest wiele sposobów grupowania obiektów w kolekcje. Można utworzyć obiekty Array, Stack, List, Hashtable. Każdy z nich ma swoje zalety i wady. Jednak w pewnym momencie klient rozpocznie iteracyjne przetwarzanie elementów kolekcji. Czy wtedy ujawnisz mu swoją implementację? Mam nadzieję, że nie. To nie byłoby profesjonalne. Nie musisz się jednak obawiać, Twoja kariera zawodowa nie jest zagrożona. W tym rozdziale przedstawimy metodę, która umożliwia klientom przetwarzanie iteracyjne bez wiedzy o tym, jak obiekty są przechowywane. Przedstawimy też technikę tworzenia *superkolekcji* obiektów, które pozwalają na obsługę bardzo rozbudowanych struktur danych. Będziemy także pisać o odpowiedzialności obiektów.

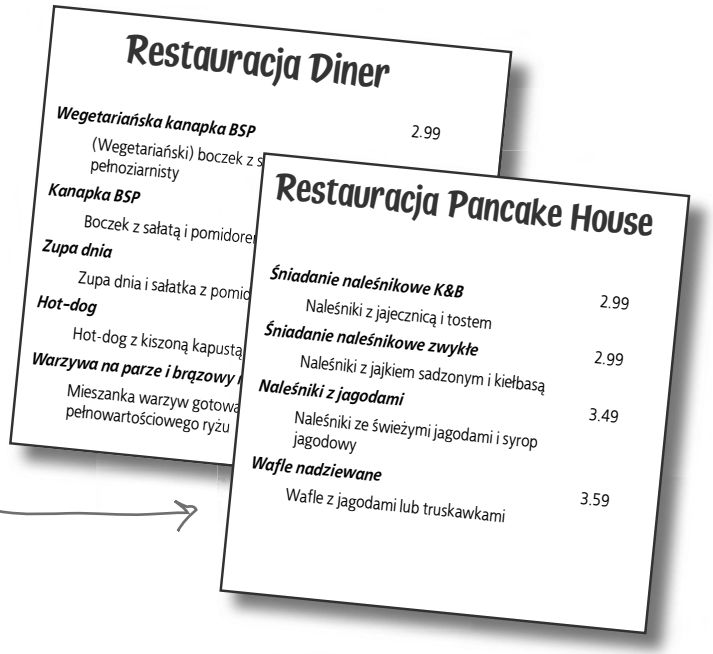
Z ostatniej chwili — Fuzja restauracji Diner i Pancake House

To dopiero wiadomość! Smakowite naleśniki z Pancake House i wyśmienite lunche z Dinera — teraz dostaniemy je w Obiektowie w jednym miejscu. Ale jest jeden mały problem…



Przegląd pozycji menu

Łukasz i Miłosz zgadzają się przynajmniej w kwestii implementacji obiektów PozycjaMenu. Przyjrzyjmy się pozycjom menu i ich implementacji.



Menu Dinera ma wiele pozycji odpowiednich, aby zjeść je w porze lunchu, a menu Pancake House zawiera pozycje śniadaniowe. Każda z nich posiada własną nazwę, opis i cenę.

```
public class PozycjaMenu {
    String nazwa;
    String opis;
    boolean wegetariańska;
    double cena;

    public PozycjaMenu(String nazwa,
                       String opis,
                       boolean wegetariańska,
                       double cena)
    {
        this.nazwa = nazwa;
        this.opis = opis;
        this.wegetariańska = wegetariańska;
        this.cena = cena;
    }

    public String pobierzNazwa() {
        return nazwa;
    }

    public String pobierzOpis() {
        return opis;
    }

    public double pobierzCena() {
        return cena;
    }

    public boolean jestWegetariańska() {
        return wegetariańska;
    }
}
```

Obiekt PozycjaMenu zawiera nazwę, opis, znacznik sygnalizujący, czy pozycja menu jest wegetariańska, oraz cenę. Wszystkie te wartości pobiera konstruktor.

Metody typu getter umożliwiają dostęp do pól pozycji menu.

Implementacje menu Łukasza i Miłosza

Przyjrzyjmy się teraz przedmiotowi konfliktu Łukasza i Miłosza. Obaj zainwestowali znaczną ilość czasu w przygotowanie kodu odpowiedzialnego za przechowywanie pozycji menu. Jest też inny kod, zależny od implementacji menu.



Oto implementacja menu Pancake House przygotowana przez Łukasza.

```
public class PancakeHouseMenu {
    ArrayList pozycjeMenu;

    public PancakeHouseMenu() {
        pozycjeMenu = new ArrayList();

        dodajElement("Śniadanie naleśnikowe K&B",
            "Naleśniki z jajecznicą i tostem",
            true,
            2.99);

        dodajElement("Śniadanie naleśnikowe zwykłe",
            "Naleśniki z jajkiem sadzonym i kiełbasą",
            false,
            2.99);

        dodajElement("Naleśniki z jagodami",
            "Naleśniki ze świeżymi jagodami",
            true,
            3.49);

        dodajElement("Wafle nadziewane",
            "Wafle z jagodami lub truskawkami",
            true,
            3.59);
    }

    public void dodajElement(String nazwa, String opis,
                         boolean wegetariańska, double cena)
    {
        PozycjaMenu pozycjaMenu = new PozycjaMenu(nazwa, opis, wegetariańska, cena);
        pozycjeMenu.add(pozycjaMenu);
    }

    public ArrayList pobierzPozycjeMenu() {
        return pozycjeMenu;
    }

    public Iterator utwórzIterator() {
        return new PancakeHouseMenuIterator(pozycjeMenu);
    }

  // inne metody menu
}
```

Łukasz przechowuje pozycje menu w obiekcie ArrayList.

Wszystkie pozycje menu są dodawane do kolekcji ArrayList tutaj, w konstruktorze.

Każda pozycja menu ma nazwę, opis, znacznik (informujący o tym, czy jest wegetariańska) i cenę.

Aby dodać pozycję menu, Łukasz tworzy nowy obiekt PozycjaMenu, przekazując w jednym wywołaniu wszystkie parametry, po czym dodaje ten obiekt do kolekcji.

Metoda pobierzPozycjeMenu() zwraca listę pozycji menu.

Łukasz ma jeszcze inny kod menu, który wymaga implementacji opartej na obiekcie ArrayList. I nie chce przepisywać tego kodu!



Aaa! ArrayList... Ja użyłem PRAWDZIWEJ tablicy (Array), żeby mieć kontrolę nad rozmiarem menu i pobierać pozycje bez wskazywania typu.

Oto implementacja menu Miłosza.

Miłosz ma inne podejście. Używa tablicy Array, która zapewnia mu kontrolę nad rozmiarem menu i pozwala pobierać elementy bez wskazywania typu obiektu.

Podobnie jak Łukasz, Miłosz tworzy pozycje menu w wywołaniu konstruktora, korzystając z metody dodajElement().

```
public class DinerMenu {
    static final int MAKS_LICZBA_ELEMENTÓW = 6;
    int liczbaElementów = 0;
    PozycjaMenu[] pozycjeMenu;

    public DinerMenu() {
        pozycjeMenu = new PozycjaMenu[MAKS_LICZBA_ELEMENTÓW];

        dodajElement("Wegetariańska kanapka BSP",
            "(Wegetariański) Boczek z sałatą i pomidorem, chleb pszenny pełnoziarnisty",
true, 2.99);
        dodajElement("Kanapka BSP",
            "Boczek z sałatą i pomidorem, chleb pszenny pełnoziarnisty", false, 2.99);
        dodajElement("Zupa dnia",
            "Zupa dnia i sałatka z pomidora", false, 3.29);
        dodajElement("Hot-dog",
            "Hot-dog z kiszoną kapustą, rzodkiewką, cebulą i dodatkiem sera",
            false, 3.05);

        // dalsze pozycje menu

    }

    public void dodajElement(String nazwa, String opis,
                             boolean wegetariańska, double cena)
    {
        PozycjaMenu pozycjaMenu = new PozycjaMenu(nazwa, opis, wegetariańska, cena);
        if (liczbaElementów >= MAKS_LICZBA_ELEMENTÓW) {
            System.err.println("Niestety, menu jest pełne! Nie można dodać nowej pozycji");
        } else {
            pozycjeMenu[liczbaElementów] = pozycjaMenu;
            liczbaElementów = liczbaElementów + 1;
        }
    }

    public PozycjaMenu[] pobierzPozycjeMenu() {
        return pozycjeMenu;
    }

    // inne metody
}
```

dodajElement() pobiera wszystkie parametry niezbędne do utworzenia obiektu PozycjaMenu i inicjalizuje go. Sprawdza też, czy nie przekroczono limitu rozmiaru menu.

Miłosz nie dopuszcza, aby liczba pozycji menu przekroczyła pewną granicę (zapewne dlatego, że nie chce pamiętać zbyt wielu przepisów).

Metoda pobierzPozycjeMenu() zwraca tablicę pozycji menu.

Podobnie jak Łukasz, Miłosz ma pewną ilość kodu, który wymaga, aby menu było tablicą. Jest zbyt zajęty przyrządzaniem potraw, aby przepisywać ten kod.

Jaki problem stwarzają dwie reprezentacje menu?

Aby przekonać się, w jaki sposób dwie różne reprezentacje menu utrudniają pracę programisty, spróbujemy stworzyć implementację klienta, który z nich korzysta. Wyobraź sobie, że zostajesz zatrudniony przez nową firmę, powstałą w wyniku połączenia restauracji Diner i Pancake House. Masz zaprojektować kelnerkę Javy (jesteśmy w Obiektowie…). Specyfikacja kelnerki Javy wymaga umiejętności drukowania menu dostosowanych do potrzeb klienta oraz podawania informacji o tym, czy dana pozycja jest wegetariańska, czy nie, bez pytania kucharza.

Zapiszmy specyfikację i spróbujmy przeprowadzić jej implementację…



Specyfikacja kelnerki Javy

Kelnerka Javy: nazwa kodowa "Alicja"

```
drukujMenu()
  - drukuje wszystkie pozycje menu

drukujMenuŚniadań()
  - drukuje wyłącznie pozycje śniadaniowe

drukujMenuLunchu()
  - drukuje menu pory lunchu

drukujMenuWegetariańskie()
  - drukuje menu wegetariańskie

jestWegetariańska(nazwa)
  - pobiera nazwę pozycji menu i zwraca true,
    gdy pozycja jest wegetariańska; w innych
    przypadkach zwraca false
```

Rozważmy teraz implementowanie metody drukujMenu():

1. Aby wydrukować wszystkie pozycje obu menu, niezbędne są wywołania metod pobierzPozycjeMenu() obiektów PancakeHouseMenu i DinerMenu. Każda z nich zwraca inny typ:

```
PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
ArrayList pozycjeŚniadaniowe = pancakeHouseMenu.pobierzPozycje-
Menu();

DinerMenu dinerMenu = new DinerMenu();
PozycjaMenu[] pozycjePoryLunchu = dinerMenu.pobierzPozycjeMenu();
```

Wywołania wyglądają podobnie, ale każda z metod zwraca inny typ.

Implementacja musi być dostosowana do tego, że jedno menu to ArrayList, a drugie — tablica.

2. Aby wydrukować menu naleśników, niezbędna jest pętla dla kolejnych elementów kolekcji ArrayList pozycjeŚniadaniowe; aby wydrukować menu lunchu, niezbędna jest pętla wywołująca elementy kolekcji Array.

Przetwarzanie obiektów kolekcji w obu implementacjach wymaga zastosowania dwóch pętli...

```
for (int i = 0; i < pozycjeŚniadaniowe.size(); i++) {
    PozycjaMenu pozycjaMenu = (PozycjaMenu)pozycjeŚniadaniowe.get(i);
    System.out.print(pozycjaMenu.pobierzNazwa() + " ");
    System.out.println(pozycjaMenu.pobierzCena() + " ");
    System.out.println(pozycjaMenu.pobierzOpis());
}

for (int i = 0; i < pozycjePoryLunchu.length; i++) {
    PozycjaMenu pozycjaMenu = pozycjePoryLunchu[i];
    System.out.print(pozycjaMenu.pobierzNazwa() + " ");
    System.out.println(pozycjaMenu.pobierzCena() + " ");
    System.out.println(pozycjaMenu.pobierzOpis());
}
```

... pierwsza dla kolekcji ArrayList...

i druga dla tablicy.

3. Podobnie będzie przebiegać implementowanie każdej innej metody kelnerki. Niezbędne będzie wywoływanie dwóch menu i tworzenie dwóch pętli. Dołączenie kolejnej restauracji, z inną implementacją, wymusi konieczność korzystania z *trzech* pętli.



Zaostrz ołówek

Które stwierdzenia są prawdziwe dla naszej implementacji metody drukujMenu()?

- ❑ A. Piszemy kod pod kątem konkretnych implementacji obiektów PancakeHouseMenu i DinerMenu, a nie interfejsu.
- ❑ B. Klasa kelnerki nie implementuje Java Waitress API, a więc nie jest klasą standardową.
- ❑ C. Gdybyśmy zdecydowali się zastąpić klasę DinerMenu innym typem menu, w którym lista pozycji ma postać kolekcji Hashtable, wymagałoby to zmiany dużej ilości kodu klasy kelnerki.
- ❑ D. Klasa kelnerki musi znać reprezentację wewnętrznej kolekcji każdego menu; burzy to zasadę hermetyzacji.
- ❑ E. Musimy powtarzać fragmenty kodu: metoda drukujMenu() wymaga dwóch niezależnych pętli, aby przetworzyć zawartość dwóch różnych menu. Dodanie trzeciego menu prowadzi do konieczności użycia kolejnej takiej pętli.
- ❑ F. Implementacja jest oparta na MXML (Menu XML), więc jej współpraca z innymi elementami aplikacji jest ograniczona.

Co teraz?

Miłosz i Łukasz stawiają nas przed trudnym problemem. Nie chcą zmieniać własnych implementacji, ponieważ wymagałoby to jednoczesnej zmiany znacznych ilości kodu w klasach menu. Jeżeli żaden nie ustąpi, implementacja kodu kelnerki będzie od początku trudna do modyfikowania i testowania.

Eleganckim rozwiązaniem byłoby znalezienie sposobu, który umożliwi Miłoszowi i Łukaszowi zaimplementowanie tego samego interfejsu obu menu (są już tego bliscy, problemem jest jednak typ zwracany przez metodę pobierzPozycjeMenu()). Pozwoliłoby to ograniczyć stosowanie odwołań konkretnych w kodzie kelnerki i być może uwolniłoby ich od konieczności używania dwóch pętli.

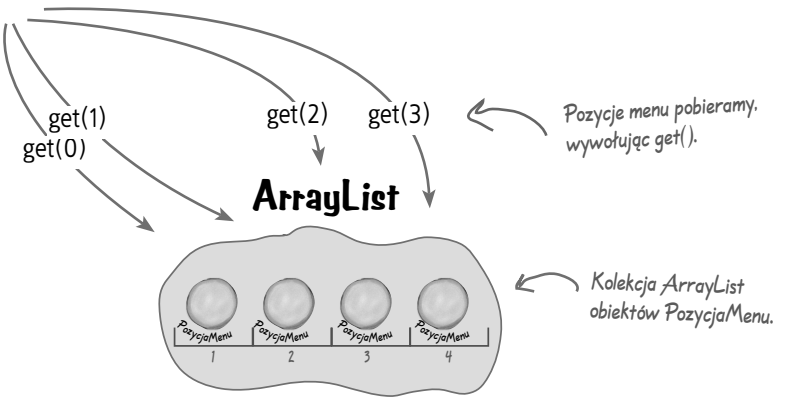
Brzmi nieźle? A więc, jak to zrobimy?

Czy można hermetyzować iteracje?

Jeżeli mielibyśmy wybrać jedną ważną rzecz do zapamiętania z całej treści tej książki, byłaby to zasada hermetyzowania elementów, które mogą się zmieniać. Zmianom w tym przypadku ulega iteracja, różna dla każdej z dwóch kolekcji obiektów menu. Jak ją hermetyzować? Spróbujmy wprowadzić w życie pewien pomysł…

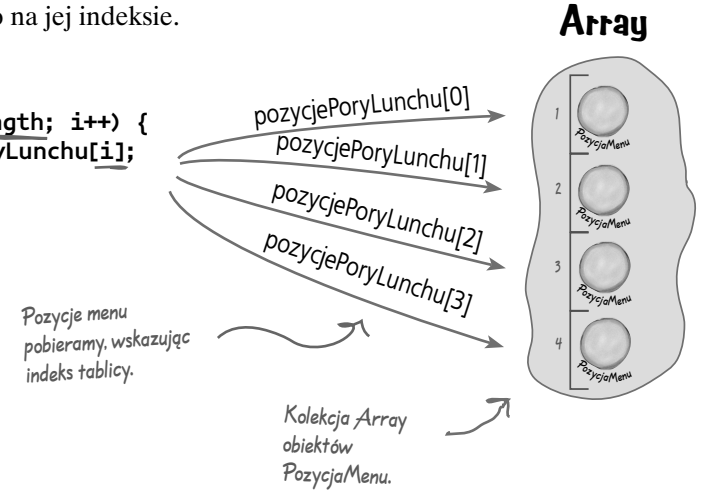
❶ Aby iterować pozycje śniadaniowe, używamy metod size() i get() obiektu ArrayList:

```
for (int = 0; i < pozycjeŚniadaniowe.size(); i++) {
    PozycjaMenu pozycjaMenu = (PozycjaMenu)pozycjeŚniadaniowe.get(i);
}
```



❷ Aby iterować pozycje przeznaczone do jedzenia podczas lunchu, korzystamy z pola długości tablicy i wywołania opartego na jej indeksie.

```
for (int i = 0; i < pozycjePoryLunchu.length; i++) {
    PozycjaMenu pozycjaMenu = pozycjePoryLunchu[i];
}
```



❸ Utwórzmy więc obiekt Iterator, który hermetyzuje sposób iterowania kolekcji obiektów. Najpierw dla kolekcji ArrayList...

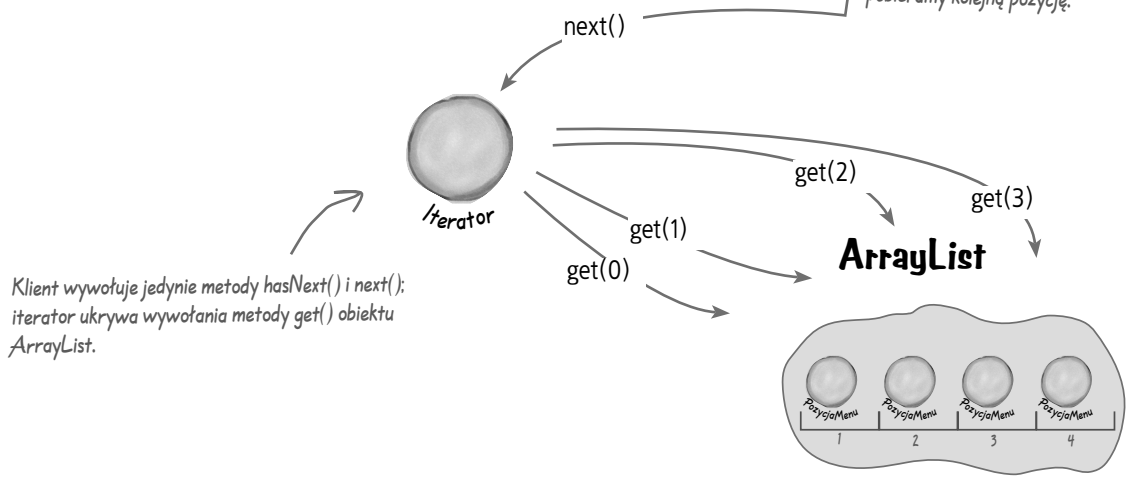
```
Iterator iterator = menuŚniadaniowe.utwórzIterator();

while (iterator.hasNext()) {
    PozycjaMenu pozycjaMenu = (PozycjaMenu)iterator.next();
}
```

Wywołujemy metodę obiektu menuŚniadaniowe, aby uzyskać jego iterator.

I dopóki pozostały jeszcze elementy...

pobieramy kolejną pozycję.



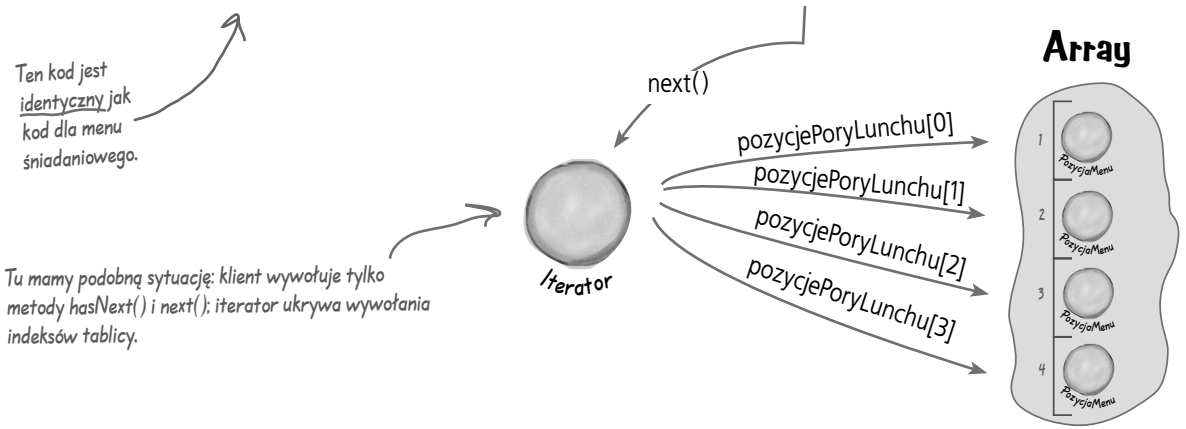
Klient wywołuje jedynie metody hasNext() i next(); iterator ukrywa wywołania metody get() obiektu ArrayList.

❹ Podobnie będzie wyglądać obsługa tablicy Array:

```
Iterator iterator = menuPoryLunchu.utwórzIterator();

while (iterator.hasNext()) {
    PozycjaMenu pozycjaMenu = (PozycjaMenu)iterator.next();
}
```

Ten kod jest identyczny jak kod dla menu śniadaniowego.

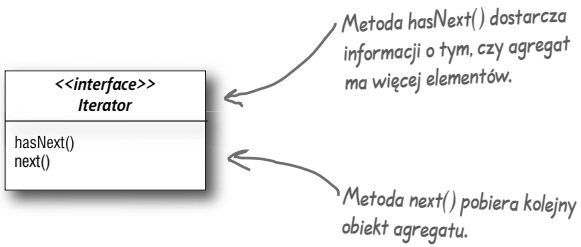


Tu mamy podobną sytuację: klient wywołuje tylko metody hasNext() i next(); iterator ukrywa wywołania indeksów tablicy.

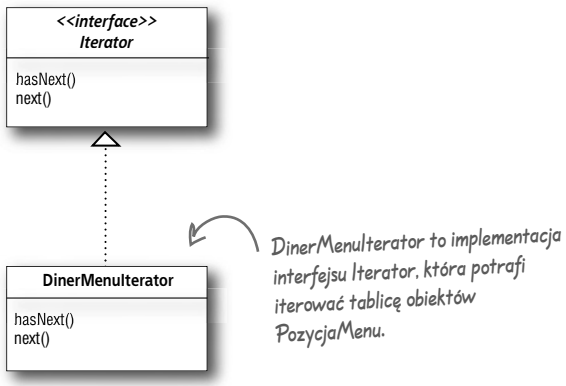
Wzorzec Iterator

Jak widać, hermetyzacja iteracji może być całkiem dobrym rozwiązaniem. Zapewne domyśliłeś się, że na tym właśnie polega wzorzec projektowy Iterator.

Pierwszą informacją o wzorcu Iterator powinno być to, że implementująca klasa prezentuje interfejs o nazwie Iterator. Przykładem tego rodzaju interfejsu może być:



Dostosowując klasy do takiego interfejsu, możemy implementować iteratory dla dowolnych kolekcji obiektów: tablic, list, tablic asocjacyjnych i innych. Przykładem może być implementacja wzorca Iterator dla tablic wykorzystywanych w kolekcji DinerMenu. Diagram klas prezentuje się następująco:



Gdy używamy określenia KOLEKCJA (ang. *collection*), mówimy o grupie obiektów. Obiekty te mogą być przechowywane w różnych strukturach danych, takich jak listy, tablice czy tablice asocjacyjne. Każda z nich jest kolekcją. Inna nazwa to AGREGAT (ang. *aggregate*).



Iterator DinerMenu

Wprowadzanie iteratora obiektów DinerMenu rozpoczniemy od zdefiniowania interfejsu:

```
public interface Iterator {
    boolean hasNext();
    Object next();
}
```

Oto nasze dwie metody.

Metoda hasNext() zwraca wartość logiczną, sygnalizującą, czy w pętli iteracji znajduje się więcej pozycji…

… a metoda next() zwraca kolejny element.

Kolejnym krokiem jest implementacja rzeczywistej klasy iteratora, która współpracuje z menu restauracji Diner:

```
public class DinerMenuIterator implements Iterator {
    PozycjaMenu[] elementy;
    int pozycja = 0;

    public DinerMenuIterator(PozycjaMenu[] elementy) {
        this.elementy = elementy;
    }

    public Object next() {
        PozycjaMenu pozycjaMenu = elementy[pozycja];
        pozycja = pozycja + 1;
        return pozycjaMenu;
    }

    public boolean hasNext() {
        if (pozycja >= elementy.length || elementy[pozycja] == null) {
            return false;
        } else {
            return true;
        }
    }
}
```

Implementujemy interfejs Iterator.

Pole pozycja przechowuje indeks iterowanego elementu.

Konstruktor pobiera tablicę pozycji menu, które będą iterowane.

Metoda next() zwraca kolejny element tablicy i zwiększa wartość zmiennej pozycja.

Metoda hasNext() sprawdza, czy trwa przetwarzanie elementów tablicy, i zwraca true, gdy istnieją dalsze elementy.

Ponieważ szef kuchni restauracji Diner był przewidujący i określił największy dopuszczalny rozmiar tablicy, musimy sprawdzać nie tylko, czy osiągnęliśmy jej koniec, ale również, czy kolejny element nie ma wartości null, co oznacza brak dalszych elementów.

Wiązanie iteratora z obiektem menu

W porządku, mamy już iterator. Musimy jeszcze zmienić nieco klasę DinerMenu, dodając do niej metodę, która tworzy obiekt DinerMenuIterator i zwraca go klientowi:

```
public class DinerMenu {
    static final int MAKS_LICZBA_ELEMENTÓW = 6;
    int liczbaElementów = 0;
    PozycjaMenu[] pozycjeMenu;

  // konstruktor

  // metoda dodajElement

  public PozycjaMenu[] pobierzPozycjeMenu() {
      return pozycjeMenu;
  }

    public Iterator utwórzIterator() {
        return new DinerMenuIterator(pozycjeMenu);
    }

  // dalsze metody menu
}
```

Metoda pobierzPozycjeMenu() nie będzie już potrzebna. Jest wręcz niepożądana, bo ujawnia implementację!

Nowa metoda utwórzIterator(). W oparciu o tablicę pozycjeMenu tworzy obiekt DinerMenuIterator i zwraca go klientowi.

Zwracamy interfejs Iterator. Klient nie musi znać szczegółów obsługi pozycji menu w klasie DinerMenu ani implementacji klasy DinerMenuIterator. Będzie korzystał jedynie z iteratorów umożliwiających wywoływanie kolejnych pozycji menu.



Ćwiczenia

Samodzielnie napisz kod klasy PancakeHouseIterator i wprowadź zmiany niezbędne do powiązania jej z obiektem PancakeHouseMenu.

Poprawiamy kod kelnerki

Zintegrujemy teraz nowy kod iteratorów z kodem klasy kelnerki. Jednym z efektów powinno być usunięcie powtórzeń kodu. Operacja jest stosunkowo prosta: najpierw tworzymy metodę drukujMenu(), która pobiera iterator, a następnie korzystamy z metody utwórzIterator() każdego z menu, aby pobrać odpowiedni iterator i przekazać go nowej metodzie.



Nowa, lepsza, z iteratorem.

```
public class Kelnerka {
    PancakeHouseMenu pancakeHouseMenu;
    DinerMenu dinerMenu;

    public Kelnerka(PancakeHouseMenu pancakeHouseMenu, DinerMenu dinerMenu) {
        this.pancakeHouseMenu = pancakeHouseMenu;
        this.dinerMenu = dinerMenu;
    }

    public void drukujMenu() {
        Iterator pancakeIterator = pancakeHouseMenu.utwórzIterator();
        Iterator dinerIterator = dinerMenu.utwórzIterator();

        System.out.println("MENU\n----\nŚNIADANIA");
        drukujMenu(pancakeIterator);
        System.out.println("\nLUNCH");
        drukujMenu(dinerIterator);
    }

    private void drukujMenu(Iterator iterator) {
        while (iterator.hasNext()) {
            PozycjaMenu pozycjaMenu = (PozycjaMenu)iterator.next();
            System.out.print(pozycjaMenu.pobierzNazwa() + ", ");
            System.out.print(pozycjaMenu.pobierzCena() + " -- ");
            System.out.println(pozycjaMenu.pobierzOpis());
        }
    }

  // pozostałe metody menu
}
```

Konstruktor kelnerki pobiera oba menu.

Metoda drukujMenu() tworzy teraz dwa iteratory, po jednym dla każdego menu.

Po czym wywołuje metodę drukujMenu() w wersji z jednym parametrem, którym jest iterator.

Sprawdzamy, czy są dalsze elementy.

Pobieramy kolejny element.

Przeciążona metoda drukujMenu() wykorzystuje iterator do wywoływania kolejnych pozycji menu i drukowania informacji o nich.

Zwróćmy uwagę, że teraz mamy jedną pętlę.

Pobrana pozycja menu dysponuje metodami do pobierania nazwy, ceny i opisu.

Testowanie kodu

Czas poddać nowy kod testom. Napiszmy krótką procedurę pomocniczą i zobaczmy kelnerkę przy pracy…

```
public class MenuTest {
    public static void main(String args[]) {
        PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
        DinerMenu dinerMenu = new DinerMenu();

        Kelnerka kelnerka = new Kelnerka(pancakeHouseMenu, dinerMenu);

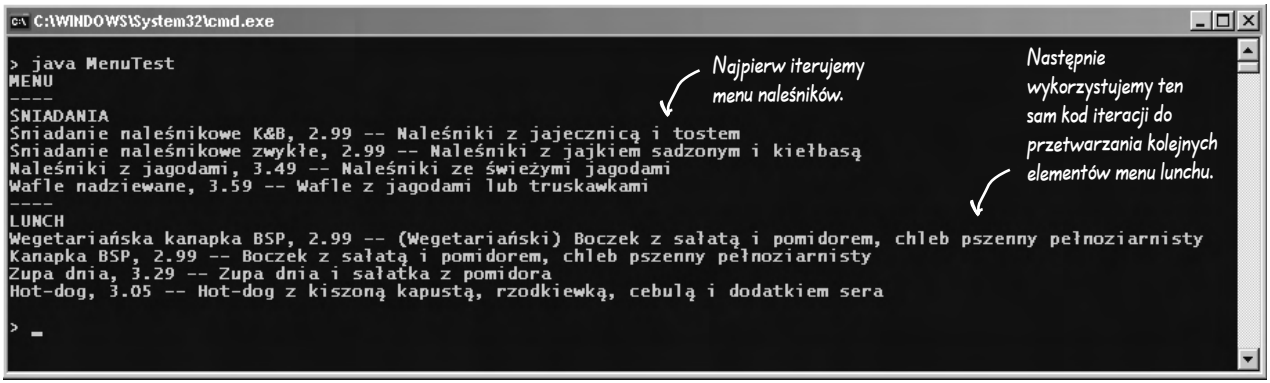
        kelnerka.drukujMenu();
    }
}
```

Najpierw tworzymy nowe menu.

Następnie tworzymy egzemplarz klasy Kelnerka, przekazując doń nowe menu.

Kolejnym krokiem jest wydrukowanie go.

Oto przebieg testu...



Krótkie podsumowanie — co już zrobiliśmy?

Przede wszystkim spełniliśmy oczekiwania obu szefów kuchni. Problem różnic został rozwiązany, a każdy mógł zachować swoją implementację. Daliśmy im klasy PancakeHouseIterator i DinerMenuIterator, a im pozostało jedynie wprowadzenie metody utwórzIterator().

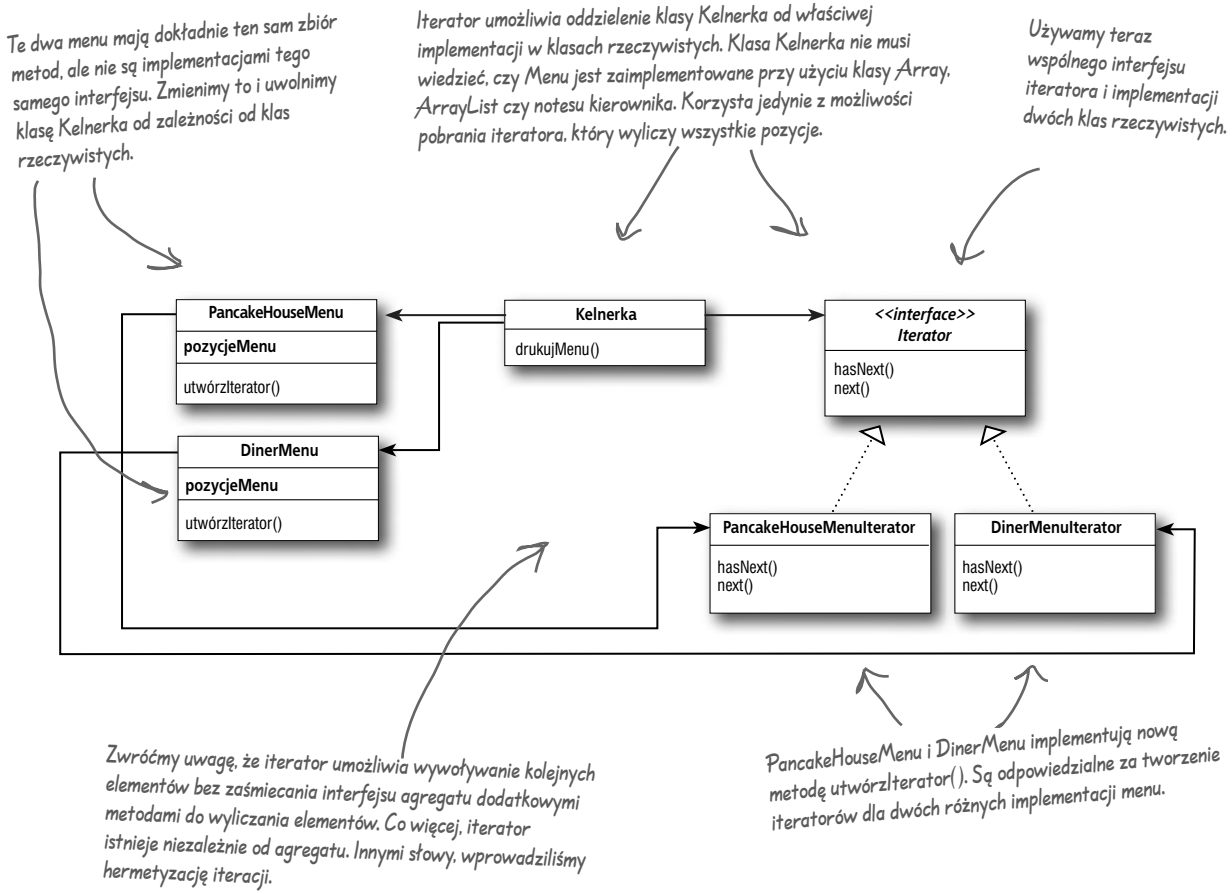
Doprowadziliśmy też do uproszczenia i zapewnienia rozszerzalności kodu klasy Kelnerka. Przyjrzyjmy się bliżej charakterowi wprowadzonych zmian i wynikających z nich korzyści:



Nieprzejrzysta implementacja kelnerki	Nowa, sprawna kelnerka z iteratorem
Menu nie są dobrze hermetyzowane. Widać, że Diner korzysta z klasy ArrayList, a PancakeHouse z klasy Array.	Implementacje menu są w pełni hermetyzowane. Kelnerka nie dysponuje żadną wiedzą o tym, w jaki sposób obiekty Menu przechowują swoje kolekcje pozycji menu.
Iterowanie pozycji menu wymaga dwóch pętli.	Używamy pętli, która obsługuje kolekcje elementów polimorficznie, o ile tylko udostępniają one iterator.
Kelnerka jest powiązana z klasami rzeczywistymi (PozycjaMenu[] i ArrayList).	Kelnerka korzysta z interfejsu (Iterator).
Kelnerka jest powiązana z dwiema różnymi klasami rzeczywistymi menu, pomimo że ich interfejsy są niemal identyczne.	Interfejsy menu są dokładnie takie same, choć — och! — wciąż nie mamy interfejsu wspólnego, co oznacza, że kelnerka pozostaje powiązana z dwiema klasami rzeczywistymi. Lepiej to naprawić...

Co już mamy...

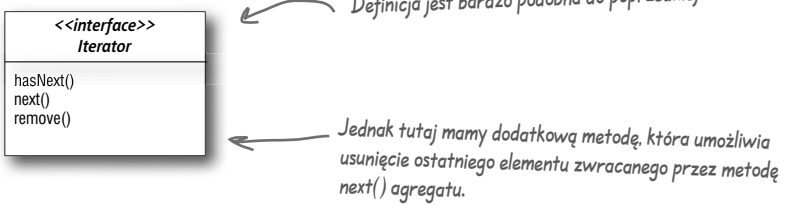
Zanim wprowadzimy dalsze usprawnienia, pozwólmy sobie na nieco szersze spojrzenie na kod naszego projektu.



Dalsze usprawnienia...

W porządku, wiemy, że interfejsy obiektów PancakeHouseMenu i DinerMenu są dokładnie takie same, ale nie zdefiniowaliśmy jeszcze dla nich interfejsu wspólnego. Gdy to zrobimy, klasa Kelnerka będzie jeszcze prostsza.

Możesz się zastanawiać, dlaczego nie używamy interfejsu Iterator języka Java — ma to na celu przede wszystkim przedstawienie sposobu budowania iteratora od podstaw. Po takim wstępie przejdziemy do korzystania ze standardowego interfejsu języka. nam to istotne korzyści. Jakie? Przekonamy się wkrótce.



Zmiana jest bardzo prosta: musimy tylko zmienić interfejs rozszerzany przez obie klasy, PancakeHouseMenuIterator i DinerMenuIterator, prawda? Prawie… w rzeczywistości jest to nawet prostsze. Pakiet java.util zawiera już interfejs Iterator, a klasa ArrayList ma metodę iterator(), która zwraca iterator. Innymi słowy, implementowanie własnego iteratora obiektu ArrayList nie było potrzebne. Nasza implementacja wciąż musi obejmować iterator obiektu DinerMenu, ponieważ ten jest oparty na tablicy, a ona nie ma metody iterator() (ani żadnej innej, która zwraca odpowiedni obiekt).

Nie istnieją głupie pytania

P: Co zrobić, gdy nie chcę wprowadzać możliwości usuwania elementów kolekcji?

O: Metoda remove() ma charakter opcjonalny. Jej funkcja nie musi zostać zaimplementowana. Wymóg odpowiedniej deklaracji narzuca jednak interfejs Iterator. Jeżeli rezygnujemy z funkcji usuwania elementów obsługiwanych przez iterator, powinniśmy wprowadzić wyjątek czasu wykonania java.lang.UnsupportedOperationException. Dokumentacja interfejsu Iterator określa, że metoda remove() może generować taki wyjątek. Każdy poprawnie napisany klient powinien sprawdzać, czy on nie wystąpił, zawsze, gdy korzysta z tej metody.

P: Jak działa metoda remove() w programie wielowątkowym, gdzie różne iteratory mogą mieć dostęp do tych samych kolekcji?

O: Nie zdefiniowano wymaganego działania metody remove() dla przypadków, gdy kolekcja ulega zmianom w trakcie iterowania. Zmusza to do szczególnej ostrożności, gdy projektujemy kod wielowątkowy, z wieloma jednoczesnymi operacjami dostępu do kolekcji.

Uproszczenia po wprowadzeniu interfejsu java.util.Iterator

Rozpocznijmy od klasy PancakeHouseMenu. Wprowadzenie w niej odwołania do interfejsu java.util.Iterator będzie proste. Wystarczy usunąć klasę PancakeHouseMenuIterator, wprowadzić import klasy java.util.Iterator na początku definicji PancakeHouseMenu i zmienić jeden wiersz w tejże klasie menu:

```
    public Iterator utwórzIterator() {
        return pozycjeMenu.iterator();
    }
```

W miejsce tworzenia własnego iteratora korzystamy z metody iterator() klasy ArrayList wykorzystywanej do obsługi kolekcji pozycji menu.

Na tym kończy się nasza praca z klasą PancakeHouseMenu.

Teraz definiujemy opartą na tym samym interfejsie java.util.Iterator klasę DinerMenuIterator.

Rozpoczynamy od zaimportowania implementowanego interfejsu java.util.Iterator.

```
import java.util.Iterator;

public class DinerMenuIterator implements Iterator {
    PozycjaMenu[] lista;
    int pozycja = 0;

    public DinerMenuIterator(PozycjaMenu[] lista) {
        this.lista = lista;
    }

    public Object next() {
        // implementacja
    }

    public boolean hasNext() {
        // implementacja
    }

    public void remove() {
        if (pozycja <= 0) {
            throw new IllegalStateException
                ("Nie możesz usunąć elementu przed pierwszym wywołaniem metody next()");
        }
        if (lista[pozycja-1] != null) {
            for (int i = pozycja-1; i < (lista.length-1); i++) {
                lista[i] = lista[i+1];
            }
            lista[lista.length-1] = null;
        }
    }
}
```

Implementacja pozostaje niezmieniona,...

... ale niezbędne jest zaimplementowanie metody remove(). Ponieważ szef kuchni korzysta z tablicy o stałym rozmiarze, usuwanie będzie polegać na przesunięciu części elementów o jedną pozycję.

Prawie skończyliśmy...

Musimy tylko wprowadzić wspólny interfejs dla obu menu i nieco zmodyfikować klasę Kelnerka. Interfejs menu jest stosunkowo prosty. Można go uzupełnić o dodatkowe metody, takie jak dodajElement(). Tymczasem pozostawmy obu restauracjom kontrolę nad ich własnymi menu, pozostawiając tę metodę poza interfejsem publicznym:

Prosty interfejs umożliwiający pobranie iteratora pozycji menu.

```
public interface Menu {
    public Iterator utwórzIterator();
}
```

Teraz musimy dopisać implements Menu w definicjach klas PancakeHouseMenu i DinerMenu. Następnie poprawiamy klasę kelnerki:

Teraz kelnerka również będzie używać interfejsu java.util.Iterator.

Konkretne klasy menu zastępujemy interfejsem Menu.

```
import java.util.Iterator;

public class Kelnerka {
    Menu pancakeHouseMenu;
    Menu dinerMenu;

    public Kelnerka(Menu pancakeHouseMenu, Menu dinerMenu) {
        this.pancakeHouseMenu = pancakeHouseMenu;
        this.dinerMenu = dinerMenu;
    }

    public void drukujMenu() {
        Iterator pancakeIterator = pancakeHouseMenu.utwórzIterator();
        Iterator dinerIterator = dinerMenu.utwórzIterator();

        System.out.println("MENU\n----\nŚNIADANIA");
        drukujMenu(pancakeIterator);
        System.out.println("\nLUNCH");
        drukujMenu(dinerIterator);
    }

    private void drukujMenu(Iterator iterator) {
        while (iterator.hasNext()) {
            PozycjaMenu pozycjaMenu = (PozycjaMenu)iterator.next();
            System.out.print(pozycjaMenu.pobierzNazwa() + ", ");
            System.out.print(pozycjaMenu.pobierzCena() + " -- ");
            System.out.println(pozycjaMenu.pobierzOpis());
        }
    }

    // pozostałe metody
}
```

Tutaj nic się nie zmienia.

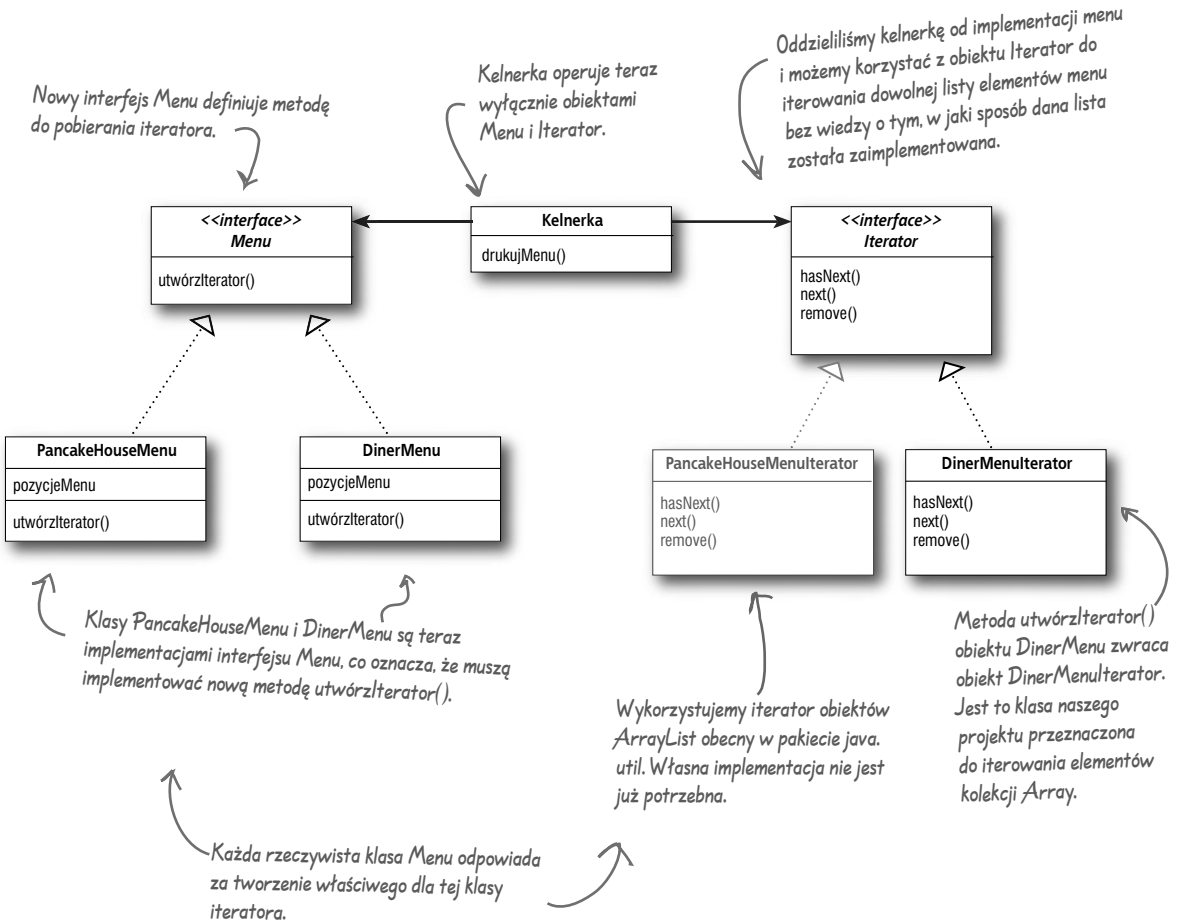
Jaki jest efekt końcowy?

Klasy PancakeHouseMenu i DinerMenu są teraz implementacjami interfejsu Menu. Kelnerka może się odwoływać do obiektów menu przy użyciu interfejsu, a nie klasy rzeczywistej. Redukujemy w ten sposób zależność między klasą Kelnerka a klasami rzeczywistymi przez „programowanie pod kątem interfejsu, a nie implementacji".

To rozwiązuje problem zależności kelnerki od klas rzeczywistych menu.

Nowy interfejs Menu ma jedną metodę, utwórzIterator(), implementowaną w klasach PancakeHouseMenu i DinerMenu. Każda klasa menu przyjmuje odpowiedzialność za utworzenie rzeczywistego obiektu Iterator, odpowiedniego do obsługiwanej implementacji menu.

To rozwiązuje problem zależności kelnerki od implementacji pozycjeMenu.



Wzorzec Iterator — definicja

Przedstawiliśmy już implementację wzorca Iterator przy użyciu własnego iteratora. Zwróciliśmy również uwagę na dostępność iteratorów dla pewnych klas kolekcji, takich jak ArrayList. Spójrzmy teraz na pełną definicję wzorca:

> **Wzorzec Iterator** zapewnia metodę dostępu sekwencyjnego do elementów obiektu zagregowanego bez ujawniania jego reprezentacji wewnętrznej.

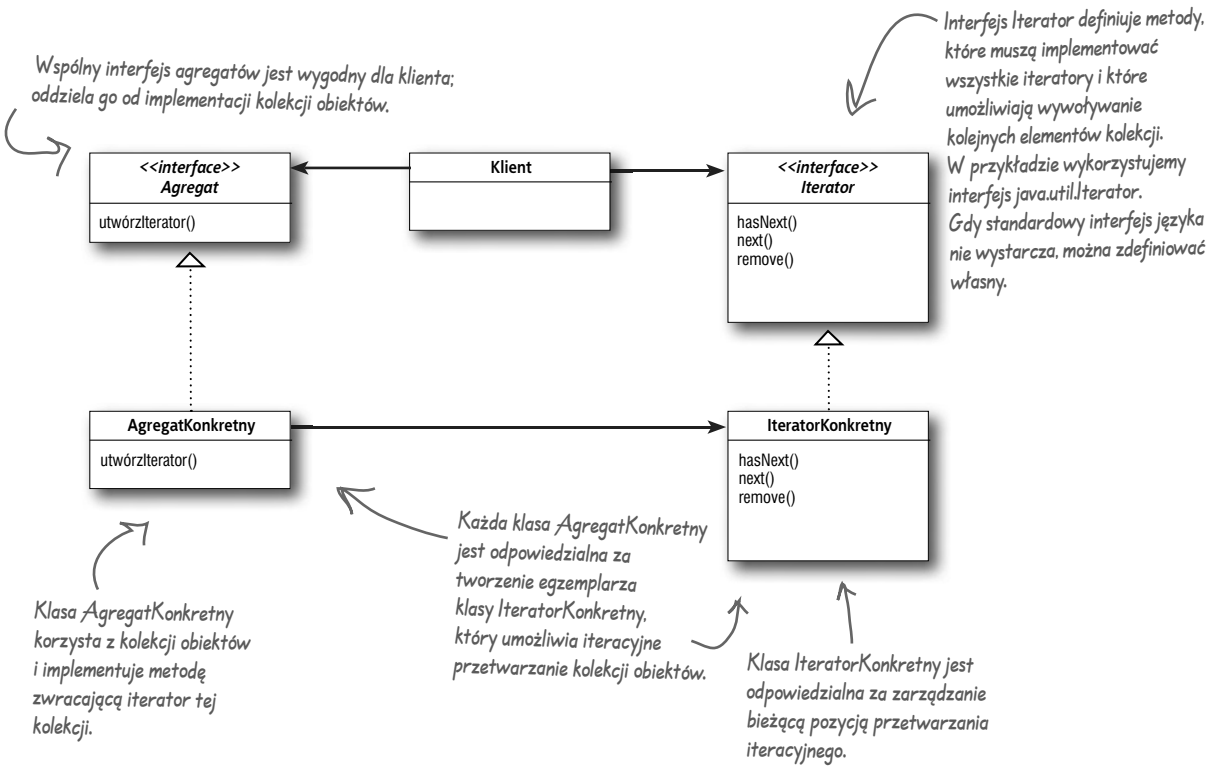
Ma to duże znaczenie: wzorzec daje klientowi sposób wyliczania elementów agregatu bez uzależniania go od zastosowanej implementacji. Zilustrowaliśmy to na przykładzie dwóch implementacji menu. Równie ważny jest wpływ, jaki wywiera wprowadzenie iteratorów w projekcie: gdy istnieje jednolity sposób wywoływania elementów wszystkich obiektów zagregowanych, można tworzyć kod polimorficzny pracujący z *dowolnym* agregatem — podobnie jak w przypadku metody drukujMenu(), dla której nie ma znaczenia, czy elementy menu są przechowywane w kolekcji Array, czy ArrayList (czy jakiejkolwiek innej, która potrafi utworzyć iterator), o ile tylko przekażemy jej odpowiedni obiekt Iterator.

Innym istotnym czynnikiem jest przejęcie przez iterator odpowiedzialności za wybieranie kolejnych elementów. Implementacja zostaje przeprowadzona w obiekcie pomocniczym, a nie w samym agregacie. Pozwala to zachować przejrzystość interfejsu agregatu i jego implementacji, jak również wyłączyć zeń operacje związane z iterowaniem, aby sama kolekcja wykonywała operacje związane z zarządzaniem jej obiektami.

Uporządkujmy wzorzec iteratora w ogólny diagram klas…

Wzorzec Iterator umożliwia przechodzenie między elementami agregatu bez ujawniania ich implementacji.

Dodatkowo przenosi implementację przechodzenia między elementami z agregatu do iteratora, co upraszcza interfejs agregatu i implementację oraz prowadzi do właściwego przypisania odpowiedzialności.





WYTĘŻ UMYSŁ

Diagram klas wzorca Iterator wygląda bardzo podobnie do diagramu jednego z wzorców przedstawianych wcześniej. Zastanów się, do którego? Wskazówka: Podklasa decyduje o tym, jaki obiekt zostanie utworzony.

Nie istnieją głupie pytania

P: Widziałem, że inne książki pokazują w diagramie klas wzorca Iterator metody first(), next(), isDone() i currentItem(). Dlaczego tutaj jest inaczej?

O: Jest to tradycyjny zestaw nazw metod. Schemat ten powoli ewoluuje i obecnie nawet w klasie java.util.Iterator znajdziemy metody next(), hasNext() i remove().

Porównajmy te interfejsy. Dwie kolejno wywoływane metody next() i currentItem() zostały połączone w java.util w jedną. isDone() jedynie zmieniła nazwę na hasNext(). Nie ma natomiast metody odpowiadającej first(). Jest to związane z tym, że w języku Java standardem jest pobieranie nowego iteratora dla każdej pętli. Można jednak stwierdzić, że interfejsy te nie różnią się znacząco. Zestaw metod iteratora może być o wiele bardziej rozbudowany. Metoda remove() jest najprostszym przykładem rozszerzenia wzorca.

P: Słyszałem o „wewnętrznych" i „zewnętrznych" iteratorach. Czym one się różnią? Które z nich zostały zaimplementowane w przedstawionym przykładzie?

O: W przykładzie przedstawiliśmy iterator zewnętrzny, co oznacza, że klient steruje operacją iterowania, wywołując metodę next(). Praca iteratora wewnętrznego nie zapewnia takiej kontroli. W przypadku, kiedy iterator sam wywołuje elementy, musi również posiadać wiedzę o operacjach, które mają zostać na tych elementach wykonane. Mamy wówczas do czynienia z przekazywaniem operacji procedurze iteratora. Iteratory wewnętrzne mają ograniczone możliwości, bo klient nie ma bezpośredniej kontroli nad iteracjami. Z drugiej strony, ograniczenie czynności klienta do wskazania obiektu i operacji może być uznane za bardziej pożądane.

P: Czy mogę zaimplementować iterator, który pozwala przeglądać elementy kolekcji w obu kierunkach?

O: Oczywiście. Taki iterator ma zazwyczaj dwie dodatkowe metody, jedną do pobierania elementu poprzedniego i drugą do określania, czy osiągnięty został początek kolekcji. Java Collection Framework zawiera ten rodzaj iteratora, nosi on nazwę ListIterator i jest wyposażony między innymi w metodę previous(). Jego obsługę zapewnia każda kolekcja implementująca interfejs List.

P: Jak określana jest kolejność iterowania kolekcji takich, jak Hashtable, a więc kolekcji z definicji nieuporządkowanych?

O: Iteratory nie wymuszają porządkowania elementów. Kolekcja może być nieuporządkowana, może nawet zawierać duplikaty. Uporządkowanie to właściwość kolekcji i jej implementacji. Nie należy oczekiwać żadnej określonej kolejności przetwarzania elementów, o ile nie mówi o niej dokumentacja kolekcji.

P: W tekście znalazłem wzmiankę o możliwości tworzenia „kodu polimorficznego"; co to dokładnie oznacza?

O: Gdy piszemy metody pobierające iterator jako parametr, stosujemy iterowanie polimorficzne. Oznacza to, że tworzymy kod, który może iterować dowolną kolekcję, o ile tylko można pobrać jej iterator. Implementacja kolekcji nie ma znaczenia dla kodu, który wykonuje na niej operacje.

P: Używam języka Java. Czy podczas pisania własnych implementacji iteratorów zawsze będę korzystał z interfejsu java.util.Iterator, aby mogły z nich korzystać również inne klasy odwołujące się standardowych iteratorów?

O: Najczęściej tak. Jednolity interfejs iteratora ułatwia korzystanie z różnych agregatów, własnych i standardowych, jak ArrayList czy Vector. Nie powinniśmy jednak zapominać, że dodając nowe funkcje do interfejsów iteratorów własnych agregatów, zawsze możemy utworzyć rozszerzenie interfejsu java.util.Iterator

P: Spotkałem się wcześniej z interfejsem `Enumeration`. Czy nie jest to implementacja wzorca Iterator?

O: Pisaliśmy o tym przy okazji wzorca Adapter. java.util.Enumeration to starsza implementacja interfejsu java.util.Iterator. Ma dwie metody, hasMoreElements() — odpowiednik hasNext() i nextElement() — odpowiednik next(). Podstawową przewagą interfejsu Iterator jest to, że współpracuje z nim więcej klas języka. W rozdziale poświęconym wzorcowi Adapter piszemy o konwersji pomiędzy tymi dwoma interfejsami.

Jeden zakres odpowiedzialności

Co stanie się, gdy zaimplementujemy w naszych agregatach ich wewnętrzne kolekcje, związane z nimi operacje i metody iteracji? Oczywiście, zwiększy to liczbę metod agregatu, ale co z tego wynika? Co w tym złego?

Aby uznać korzyści z wprowadzenia iteratora, musimy przede wszystkim zdawać sobie sprawę z tego, że gdy pozwalamy klasie wykraczać poza jej podstawowy zakres odpowiedzialności (zarządzanie pewnym agregatem), wprowadzając usługi dodatkowe (jak iterowanie), będzie ona miała więcej powodów, aby ulegać zmianom: po pierwsze, zajdą zmiany dotyczące samej kolekcji, po drugie — zmiany w sposobie iterowania. Ponownie nasz stary przyjaciel ZMIANA pojawi się w centrum zasady projektowej:

Każdy zakres odpowiedzialności klasy to obszar potencjalnych zmian. Większy zakres odpowiedzialności poszerza pole do zmian. Każda klasa powinna mieć jeden obszar odpowiedzialności.



Reguła projektowania

Klasa powinna mieć tylko jeden powód do zmian.

Zmian w klasach unikamy, oczywiście, jak zarazy — modyfikowanie kodu daje tysiąc okazji do wprowadzenia nowych zakłóceń. Wprowadzenie dwóch czynników sprzyjających zmianom zwiększa prawdopodobieństwo przyszłych zmian implementacji klasy, które mogą dotyczyć albo jednego, albo drugiego z nich.

Rozwiązanie? Przedstawiona zasada nakazuje przypisywanie każdego logicznego zakresu odpowiedzialności jednej i tylko jednej klasie.

Faktycznie, jest to dosyć proste, choć może nie do końca: rozdzielanie odpowiedzialności jest jedną z najtrudniejszych czynności wykonywanych przez projektanta aplikacji. Umysł ludzki automatycznie wyróżnia i grupuje zbiory zachowań, nawet wtedy, gdy obejmują one w rzeczywistości dwa lub więcej zakresów odpowiedzialności. Niezbędna jest więc bardzo uważna analiza projektów i wrażliwość na każdy sygnał, który może wskazywać, że rozwój aplikacji prowadzi do różnych typów zmian danej klasie.



Kohezja to termin używany do określania, jak dokładnie klasa lub moduł ogranicza swoje działanie do ściśle określonego celu lub zakresu odpowiedzialności.

Mówimy o *wysokiej kohezji* (ang. *high cohesion*), gdy moduł lub klasa realizuje zbiór powiązanych funkcji, a o *niskiej kohezji* (ang. *low cohesion*), gdy funkcje są powiązane stosunkowo luźno.

Kohezja to pojęcie ogólniejsze niż *zasada jednego zakresu odpowiedzialności*. Oba pojęcia są ściśle powiązane. Klasy, których konstrukcja odpowiada wzmiankowanej zasadzie, mają zasadniczo wyższy stopień kohezji i są bardziej przejrzyste niż klasy, których zakres odpowiedzialności jest szeroki.



WYTĘŻ UMYSŁ

Przyjrzyj się poniższym klasom i wskaż te, które mają więcej niż jeden zakres odpowiedzialności.

Gra
zaloguj()
zarejestruj()
przesuwaj()
strzelaj()
odpoczywaj()

Osoba
setImięNazwisko()
setAdres()
setNumerTelefonu()
zapisz()
ładuj()

Telefon
wybierz()
zakończ()
mów()
wyślijDane()
zeruj()

MaszynaDoWyrobuKulek
pobierzLiczba()
pobierzStan()
pobierzLokalizacja()

KartyDoGry
jestNastępna()
następna()
usuń()
dodajKartę()
usuńKartę()
sortuj()

Koszyk
dodaj()
usuń()
płać()
zapiszNaPóźniej()

Iterator
hasNext()
next()
remove()



UWAGA NA SPADAJĄCE ZAŁOŻENIA



WYTĘŻ UMYSŁ

Określ, czy przedstawione klasy mają wysoką, czy niską kohezję.

Gra
zaloguj()
zarejestruj()
przesuń()
strzelaj()
odpoczywaj()
pobierzWyniki()
pobierzImię()

SesjaGry
zaloguj()
zarejestruj()

CzynnościGracza
przesuwaj()
strzelaj()
odpoczywaj()

Gracz
pobierzWyniki()
pobierzImię()



Dobrze, że zajęliśmy się wzorcem Iterator, bo właśnie okazało się, że w Obiektowie nastąpiła nowa fuzja. Teraz łączymy się z restauracją U Jacka, integrując ich menu obiadowe.

A my myśleliśmy, że sytuacja była skomplikowana! Co teraz zrobimy?



Daj spokój, myśl pozytywnie. Na pewno znajdzie się sposób na zintegrowanie ich z wzorcem Iterator.



Poznajemy menu restauracji U Jacka

Poniżej przedstawiamy implementację menu restauracji U Jacka. Włączenie go do przygotowanego wcześniej schematu nie wygląda na trudne… Sprawdźmy.

Klasa UJackaMenu nie implementuje naszego interfejsu Menu, ale to łatwo naprawić.

Kawiarnia przechowuje pozycje menu w tablicy Hashtable. Czy ta kolekcja udostępnia iterator? Wkrótce się przekonamy.

Podobnie jak w innych menu, poszczególne pozycje są inicjalizowane w konstruktorze.

```
public class UJackaMenu {
    Hashtable pozycjeMenu = new Hashtable();

    public UJackaMenu() {
        dodajElement("Kanapka wegetariańska z frytkami",
            "Kanapka wegetariańska z sałatą i pomidorem, frytki",
            true, 3.99);
        dodajElement("Zupa dnia",
            "Filiżanka zupy dnia, sałatka",
            false, 3.69);
        dodajElement("Burito",
            "Duże burito z łaciatą fasolą, sosem salsa i guacamole (awokado)",
            true, 4.29);
    }

    public void dodajElement(String nazwa, String opis,
                             boolean wegetariańska, double cena)
    {
        PozycjaMenu pozycjaMenu = new PozycjaMenu(nazwa, opis, wegetariańska, cena);
        pozycjeMenu.put(pozycjaMenu.pobierzNazwa(), pozycjaMenu);
    }

    public Hashtable pobierzPozycjeMenu() {
        return pozycjeMenu;
    }
```

Tutaj tworzymy nową pozycję menu i dodajemy ją do tablicy asocjacyjnej.

kluczem jest nazwa pozycji

wartością jest obiekt PozycjaMenu

Tej metody nie będziemy potrzebować.



Zaostrz ołówek

Zanim przejdziesz do czytania kolejnej strony, wypisz trzy rzeczy, które musimy zrobić, aby włączyć powyższy kod do naszego schematu:

1. _____

2. _____

3. _____

Modyfikowanie kodu restauracji U Jacka

Włączenie menu kawiarni nie będzie już trudne. Dlaczego? Ponieważ Hashtable jest jedną z kolekcji języka Java, które zapewniają obsługę interfejsu Iterator. Jednak korzystanie z niego będzie wyglądać nieco inaczej…

Menu restauracji U Jacka implementuje interfejs Menu, dzięki czemu kelnerka może korzystać z niego podobnie, jak z wcześniejszych dwóch.

```
public class UJackaMenu implements Menu {
    Hashtable pozycjeMenu = new Hashtable();

    public UJackaMenu() {
        // kod konstruktora
    }

    public void dodajElement(String nazwa, String opis,
                         boolean wegetariańska, double cena)
    {
        PozycjaMenu pozycjaMenu = new PozycjaMenu(nazwa, opis, wegetariańska, cena);
        pozycjeMenu.put(pozycjaMenu.pobierzNazwa(), pozycjaMenu);
    }

    public Hashtable pobierzPozycjeMenu() {
        return pozycjeMenu;
    }

    public Iterator utwórzIterator() {
        return pozycjeMenu.values().iterator();
    }
}
```

Hashtable to jedna z najbardziej typowych struktur do przechowywania zbiorów wartości. Alternatywą jest nowsza klasa HashMap.

Podobnie jak wcześniej, możemy zrezygnować z metody pobierzPozycjeMenu(), aby nie ujawniać kelnerce implementacji struktury pozycjeMenu.

Tutaj implementujemy metodę wymaganą przez interfejs Menu. Zwróć uwagę, że nie pobieramy iteratora tablicy asocjacyjnej, a jedynie jej wartości.



Bliskie spotkania z kodem aplikacji

Hashtable to struktura nieco bardziej złożona niż ArrayList, ponieważ można w niej określać klucze i przypisane im wartości. Wciąż można jednak pobrać iterator samych wartości (którymi są pozycje menu).

```
public Iterator utwórzIterator() {
  return pozycjeMenu.values().iterator();
}
```

Najpierw pobieramy wartości tablicy asocjacyjnej — kolekcję przechowywanych w niej obiektów.

Pobrana kolekcja dysponuje metodą iterator(), która zwraca obiekt typu java.util.Iterator.

Dołączanie menu restauracji U Jacka do kodu kelnerki

Dotąd wszystko było proste. Jak jednak zmienimy kod klasy Kelnerka, aby zapewnić obsługę nowego menu? Teraz, gdy jej praca opiera się na iteratorach, nie będzie to duża zmiana.

Menu kawiarni jest przekazywane kelnerce w konstruktorze, podobnie jak pozostałe menu. Do jego przechowywania użyjemy zmiennej egzemplarza.

```
public class Kelnerka {
    Menu pancakeHouseMenu;
    Menu dinerMenu;
    Menu uJackaMenu;

    public Kelnerka(Menu pancakeHouseMenu, Menu dinerMenu, Menu uJackaMenu) {
        this.pancakeHouseMenu = pancakeHouseMenu;
        this.dinerMenu = dinerMenu;
        this.uJackaMenu = uJackaMenu;
    }

    public void drukujMenu() {
        Iterator pancakeIterator = pancakeHouseMenu.utwórzIterator();
        Iterator dinerIterator = dinerMenu.utwórzIterator();
        Iterator uJackaIterator = uJackaMenu.utwórzIterator();

        System.out.println("MENU\n----\nŚNIADANIA");
        drukujMenu(pancakeIterator);
        System.out.println("\nLUNCH");
        drukujMenu(dinerIterator);
        System.out.println("\nOBIADY");
        drukujMenu(uJackaIterator);
    }

    private void drukujMenu(Iterator iterator) {
        while (iterator.hasNext()) {
            PozycjaMenu pozycjaMenu = (PozycjaMenu)iterator.next();
            System.out.print(pozycjaMenu.pobierzNazwa() + ", ");
            System.out.print(pozycjaMenu.pobierzCena() + " -- ");
            System.out.println(pozycjaMenu.pobierzOpis());
        }
    }
}
```

Menu kawiarni posłuży do stworzenia zestawu obiadowego. Jedyne, czego potrzebujemy, aby wydrukować jego elementy, to utworzenie iteratora i przekazanie go procedurze drukujMenu(). To wszystko!

Tutaj nic się nie zmienia.

Śniadanie, lunch i obiad

Aby sprawdzić, czy całość kodu pracuje poprawnie, zaktualizujemy procedurę testową.

```
public class MenuTest {
    public static void main(String args[]) {
        PancakeHouseMenu pancakeHouseMenu = new PancakeHouseMenu();
        DinerMenu dinerMenu = new DinerMenu();
        UJackaMenu uJackaMenu = new UJackaMenu();

        Kelnerka kelnerka = new Kelnerka(pancakeHouseMenu, dinerMenu, uJackaMenu);

        kelnerka.drukujMenu();
    }
}
```

Tworzymy menu kawiarni...

... i egzemplarz klasy Kelnerka, któremu przekazujemy nowe menu.

Potem drukujemy wszystkie trzy menu.

Oto przebieg testu; teraz nowe menu obiadowe!

```
C:\WINDOWS\System32\cmd.exe

> java MenuTest
MENU
----
ŚNIADANIA
Śniadanie naleśnikowe K&B, 2.99 -- Naleśniki z jajecznicą i tostem
Śniadanie naleśnikowe zwykłe, 2.99 -- Naleśniki z jajkiem sadzonym i kiełbasą
Naleśniki z jagodami, 3.49 -- Naleśniki ze świeżymi jagodami
Wafle nadziewane, 3.59 -- Wafle z jagodami lub truskawkami
----
LUNCH
Wegetariańska kanapka BSP, 2.99 -- (Wegetariański) Boczek z sałatą i pomidorem, chleb pszenny pełnoziarnisty
Kanapka BSP, 2.99 -- Boczek z sałatą i pomidorem, chleb pszenny pełnoziarnisty
Zupa dnia, 3.29 -- Zupa dnia i sałatka z pomidora
Hot-dog, 3.05 -- Hot-dog z kiszoną kapustą, rzodkiewką, cebulą i dodatkiem sera
----
OBIADY
Kanapka wegetariańska z frytkami, 3.99 -- Kanapka wegetariańska z sałatą i pomidorem, frytki
Zupa dnia, 3.69 -- Filiżanka zupy dnia, sałatka
Burito, 4.29 -- Duże burito z łaciatą fasolą, sosem salsa i guacamole (awokado)

>
```

Najpierw iterujemy menu naleśników.

W drugiej kolejności — menu lunchu.

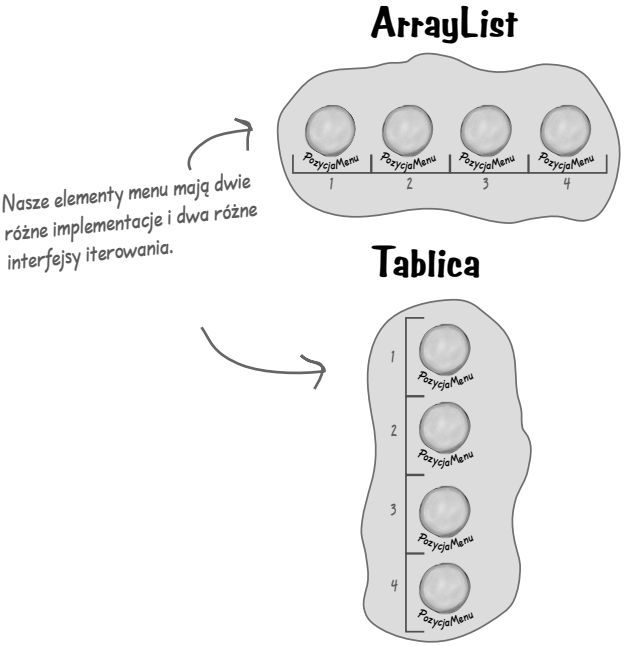
Na końcu, przy użyciu tego samego kodu iteracji, drukowane jest nowe menu obiadowe.

Co zrobiliśmy?

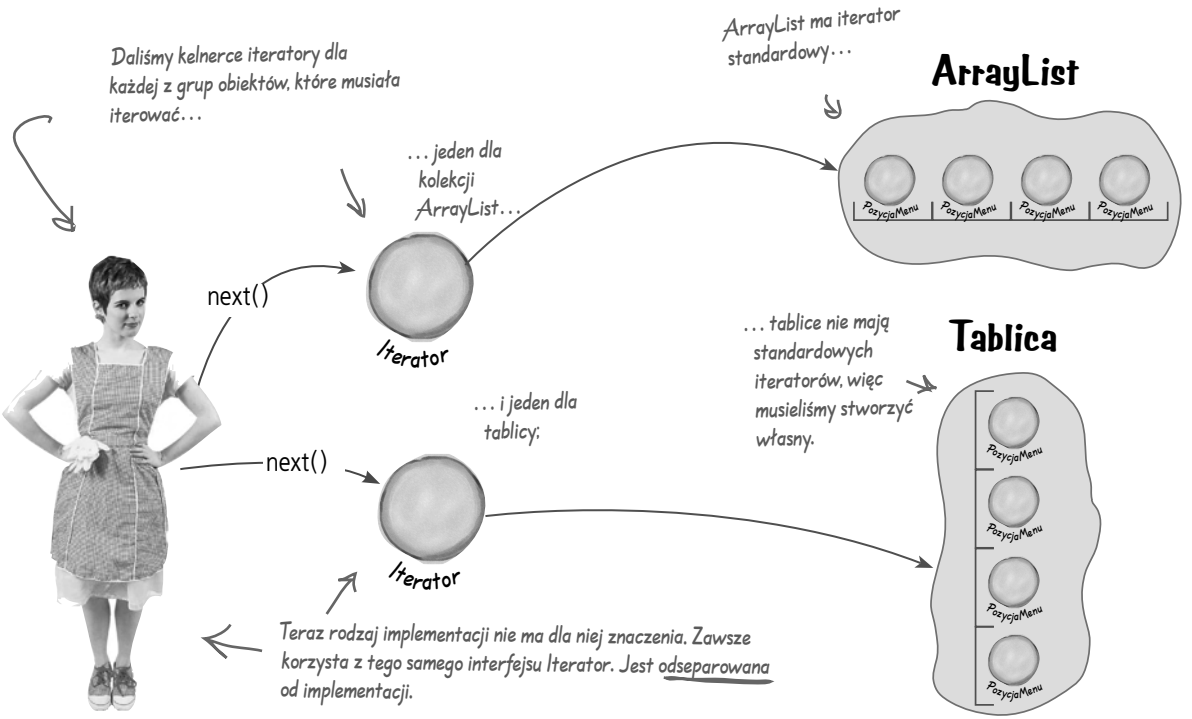


Chcieliśmy zapewnić kelnerce prosty sposób iterowania pozycji menu...

... unikając zarazem ujawniania jej tego, jak pozycje menu zostały zaimplementowane.



Odseparowaliśmy kelnerkę...



... i ułatwiliśmy jej rozbudowywanie



Dając kelnerce interfejs Iterator, odseparowaliśmy ją od implementacji pozycji menu, co znacznie ułatwia dodawanie nowych list menu, gdy zajdzie taka potrzeba.

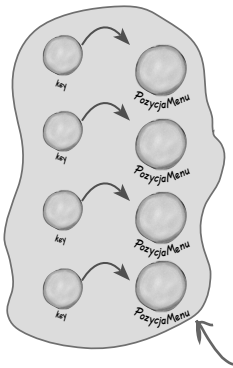
next()

Iterator

Jest to dla niej lepsze, ponieważ teraz może korzystać z tego samego kodu przy przetwarzaniu iteracyjnym dowolnej grupy obiektów. Jest też lepsze dla nas, ponieważ szczegóły implementacji pozostają ukryte.

wo dodaliśmy kolejną implementację pozycji menu. Dzięki standardowemu interfejsowi Iterator kelnerka wie, jak z nich korzystać.

Hashtable



Stworzenie iteratora wartości tablicy Hashtable było proste. Wystarczyło wywołanie values().iterator().

Ale to nie wszystko!

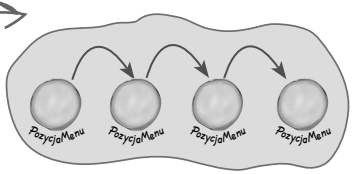
Java oferuje programiście rozbudowany zestaw klas kolekcji, które umożliwiają przechowywanie i pobieranie grup obiektów. Przykładami mogą być Vector i LinkedList.

Zazwyczaj kolekcje różnią się interfejsami.

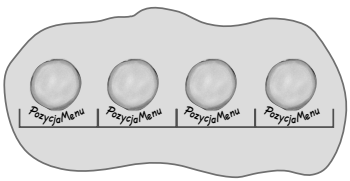
Prawie wszystkie potrafią zwrócić obiekt Iterator.

Jeżeli kolekcja nie dysponuje iteratorem, można napisać własną klasę.

LinkedList



Vector



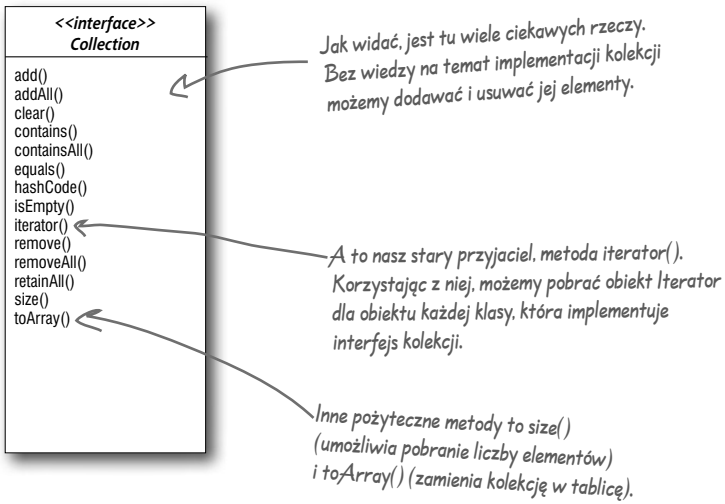
... i inne!

Iteratory i kolekcje



Użyliśmy w tym rozdziale kilku klas należących do Java Collections Framework (schemat kolekcji języka Java). Tak dumnie nazwany został zbiór klas i interfejsów, który obejmuje ArrayList (prawdopodobnie najpopularniejszą spośród nich), jak również kilka innych, w tym Vector, LinkedList, Stack i PriorityQueue. Każda z tych klas implementuje interfejs java.util.Collection zawierający szereg metod przydatnych przy zarządzaniu grupą obiektów.

Przyjrzyjmy się interfejsowi Collection jako całości:





Uwaga!

Hashtable to jedna z niewielu klas, która zapewnia obsługę iteratora *pośrednio*. Jak się przekonaliśmy, implementując klasę UJackaMenu, można pobrać iterator tablicy asocjacyjnej. Jednak wcześniej trzeba pobrać kolekcję o nazwie values. Gdy dłużej się nad tym zastanowimy, łatwo zrozumieć, dlaczego tak jest: tablica asocjacyjna przechowuje dwa zbiory obiektów — klucze i wartości. Aby przetwarzać iteracyjnie wartości, pobieramy tę kolekcję, a następnie jej iterator.

Każdy obiekt implementujący interfejs Collection może utworzyć odpowiedni obiekt Iterator. Wywołanie metody iterator() obiektu ArrayList zapewnia, że otrzymamy iterator dostosowany do takich obiektów. Nigdy jednak nie musimy zajmować się klasą rzeczywistą. Jedyne, z czego korzystamy, to interfejs iterowania.



Iteratory i kolekcje w języku Java 5



Język Java 5 ma nową odmianę instrukcji `for`, nazwaną **for-in**, która umożliwia iterowanie kolekcji lub tablicy bez jawnego tworzenia iteratora.

Oto, jak wygląda składnia pętli **for-in**:

Iterujemy każdy obiekt kolekcji.

W każdym przebiegu pętli zmienna obiekt reprezentuje kolejny element kolekcji.

```
for (Object obiekt: kolekcja) {
    ...
}
```

Oto przykład użycia pętli **for-in** do iterowania kolekcji ArrayList:

Wypełniamy kolekcję pozycjami menu.

```
ArrayList elementy = new ArrayList();
elementy.add(new PozycjaMenu("Naleśniki", "smakowite naleśniki", true, 1.59));
elementy.add(new PozycjaMenu("Wafle", "smakowite wafle", true, 1.99));
elementy.add(new PozycjaMenu("Tost", "smakowity tost", true, 0.59));

for (PozycjaMenu element: elementy) {
    System.out.println("Pozycja śniadaniowa: " + element);
}
```

Iterujemy całą listę, drukując poszczególne elementy.



Uwaga!

Aby zapewnić bezpieczeństwo typologiczne w pętli **for-in**, niezbędne jest korzystanie z mechanizmu *generics* języka Java 5. Zanim zaczniemy korzystać z obu nowych elementów języka, warto o przeczytać o nich nieco więcej.

Magnesiki z kodem



Szefowie kuchni zdecydowali, że wprowadzą możliwość zmieniania pozycji menu lunchu. Innymi słowy, będą oferować pewne pozycje w poniedziałki, środy, piątki i niedziele, a inne we wtorki, czwartki i soboty. Ktoś już napisał kod nowego, „zmieniającego się" iteratora klasy DinerMenu, który pozwala modyfikować listę pozycji, ale dla żartu pomieszał ten kod i przykleił na lodówce na zapleczu. Czy możesz go poskładać? Niektóre z nawiasów klamrowych upadły na podłogę i były zbyt małe, żeby je znaleźć, możesz więc swobodnie dodawać nowe.

```
PozycjaMenu pozycjaMenu = elementy[pozycja];
pozycja = pozycja + 2;
return pozycjaMenu;
```

```
import java.util.Iterator;
import java.util.Calendar;
```

```
public Object next() {
```

```
{
```

```
public WymienneDinerMenuIterator(PozycjaMenu[] elementy)
```

```
this.elementy = elementy;
Calendar teraz = Calendar.getInstance();
pozycja = teraz.DAY_OF_WEEK % 2;
```

```
implements Iterator
```

```
public void remove() {
```

```
PozycjaMenu[] elementy;
int pozycja;
```

```
}
```

```
public class WymienneDinerMenuIterator
```

```
public boolean hasNext() {
```

```
throw new UnsupportedOperationException(
    "WymienneDinerMenuIterator nie zapewnia obsługi operacji usuwania elementów");
```

```
if (pozycja >= elementy.length || elementy[pozycja] == null) {
    return false;
} else {
    return true;
}
```

```
}
```



Czy kelnerka jest już gotowa?

Kelnerka zmieniła się już znacząco, trzeba jednak przyznać, że trzy wywołania metody drukujMenu() nie wyglądają ładnie.

Bądźmy realistami, za każdym razem, gdy będziemy dodawać nowe menu, konieczne będzie otwieranie implementacji kelnerki i dopisywanie kolejnych wierszy kodu. Czy nie należałoby tego określić jako naruszenia zasady otwarcia na rozbudowę, a zamknięcia na modyfikacje?

Trzy wywołania utwórzIterator().

```
public void drukujMenu() {
    Iterator pancakeIterator = pancakeHouseMenu.utwórzIterator();
    Iterator dinerIterator = dinerMenu.utwórzIterator();
    Iterator uJackaIterator = uJackaMenu.utwórzIterator();

    System.out.println("MENU\n----\nŚNIADANIA");
    drukujMenu(pancakeIterator);

    System.out.println("\nLUNCH");
    drukujMenu(dinerIterator);

    System.out.println("\nOBIADY");
    drukujMenu(uJackaIterator);
}
```

Trzy wywołania drukujMenu().

Każda operacja dodania lub usunięcia menu zmusza do otwarcia klasy i wprowadzania zmian.

To nie wina kelnerki. Separowanie implementacji menu i wyodrębnienie iteracji przez wprowadzenie iteratora zostały wykonane właściwie. Poszczególne menu wciąż pozostają niezależnymi obiektami. Warto zadbać o uporządkowanie zarządzania nimi.



WYTĘŻ UMYSŁ

Kelnerka wciąż musi wywoływać metodę drukujMenu() trzykrotnie, dla każdego z kolejnych menu. Czy możemy znaleźć sposób połączenia ich w taki sposób, aby używać tylko jednego wywołania? Albo może kelnerka mogłaby korzystać z jednego, wspólnego iteratora?



Wygląda na to, że szef kuchni ma dobry pomysł.
Spróbujmy:

```
public class Kelnerka {
    ArrayList menu;

    public Kelnerka(ArrayList menu) {
        this.menu = menu;
    }

    public void drukujMenu() {
        Iterator menuIterator = menu.iterator();
        while(menuIterator.hasNext()) {
            Menu menu = (Menu)menuIterator.next();
            drukujMenu(menu.utwórzIterator());
        }
    }

    void drukujMenu(Iterator iterator) {
        while (iterator.hasNext()) {
            PozycjaMenu pozycjaMenu = (PozycjaMenu)iterator.next();
            System.out.print (pozycjaMenu.pobierzNazwa() + ", ");
            System.out.print(pozycjaMenu.pobierzCena() + " -- ");
            System.out.println(pozycjaMenu.pobierzOpis());
        }
    }
}
```

Pobieramy kolekcję ArrayList zawierającą wszystkie menu.

I iterujemy menu, przekazując iterator każdego z nich do przeciążonej metody drukujMenu().

Tu nic się nie zmienia.

Wygląda to już dość dobrze, choć straciliśmy nazwy poszczególnych menu. Dodanie ich nie powinno sprawić trudności.

I gdy już miało być tak dobrze...

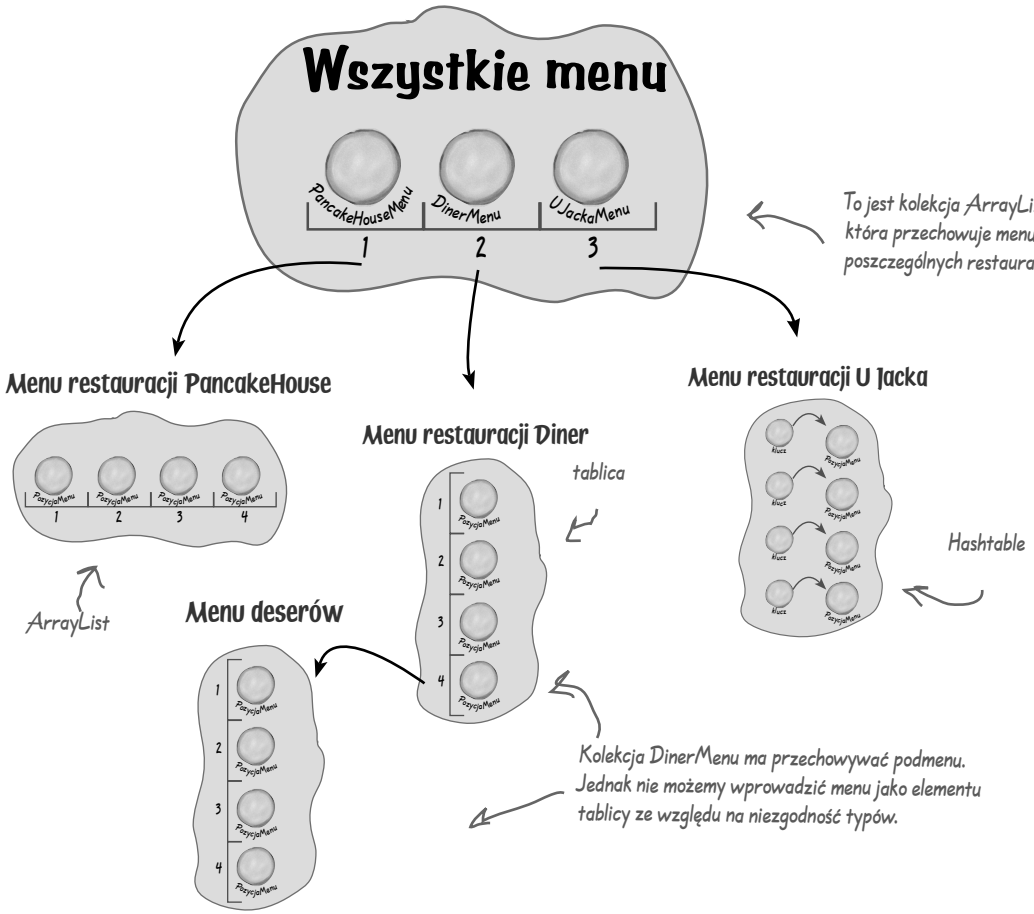
Firma wprowadza menu deserów.

I co teraz? Musimy zapewnić obsługę nie tylko wielu list menu, ale i podmenu wewnątrz innego menu.

Najprościej byłoby wprowadzić menu deserów jako element kolekcji DinerMenu, jednak nie pozwala na to bieżąca implementacja.



O co nam chodzi (w przybliżeniu):



Tak nie można zrobić!

Nie możemy wprowadzić menu deserów do tablicy obiektów PozycjaMenu.

Czas na zmianę!

Czego potrzebujemy?

Nadszedł czas, aby podjąć decyzję o zmianie wcześniejszej implementacji na taką, która zapewni wystarczający poziom ogólności, aby pracować ze wszystkimi menu, a także ich podmenu. Nie obejdzie się przy tym bez poważnych zmian.

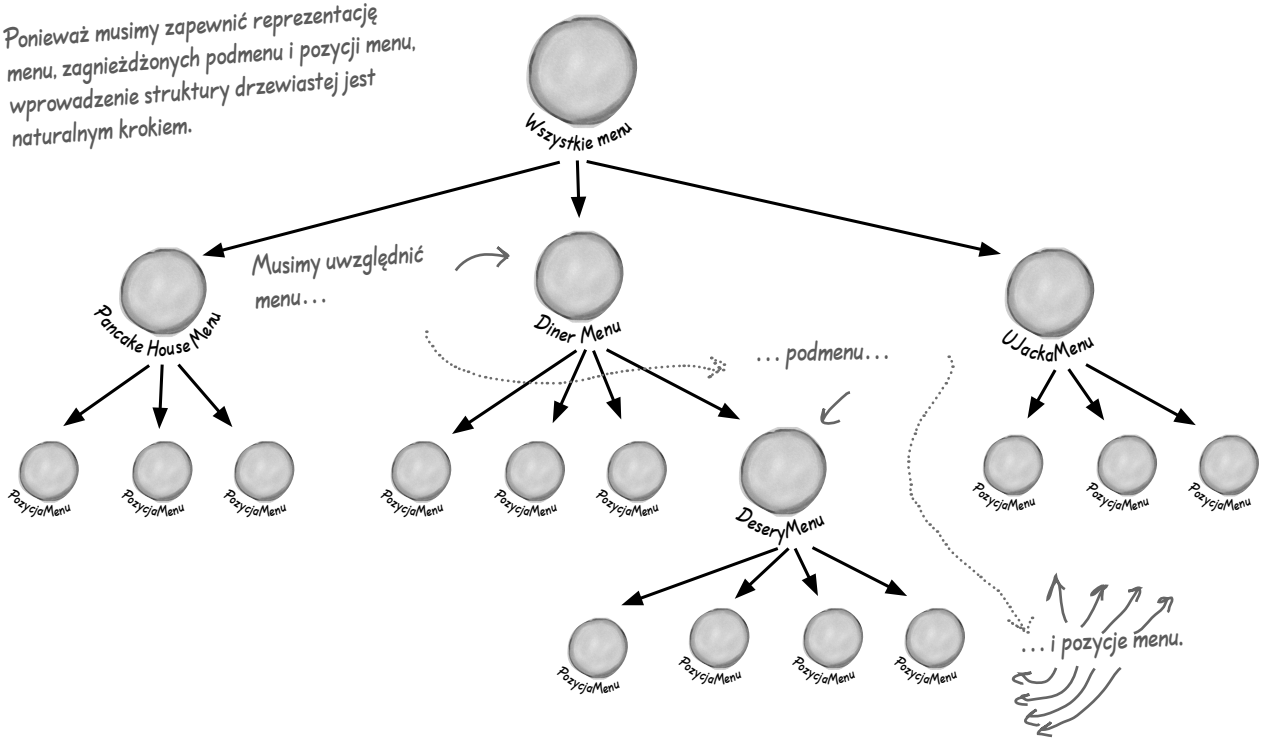
Dochodząc do tego etapu, osiągnęliśmy taki poziom złożoności, że bez zmiany wykorzystywanej struktury nie będziemy w stanie dostosować naszego projektu do dalszej rozbudowy firmy i wielopoziomowej struktury menu.

Czego więc wymagamy od nowej struktury?

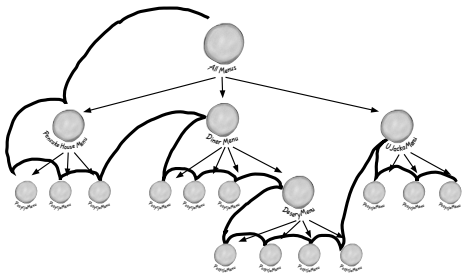
- ❑ Potrzebujemy pewnego rodzaju struktury drzewiastej, która pozwoli na tworzenie menu, podmenu i pozycji menu.
- ❑ Musimy zapewnić, że dostępny będzie sposób iterowania pozycji menu co najmniej tak wygodny jak stosowany dotąd, oparty na iteratorach.
- ❑ Może pojawić się potrzeba iterowania pozycji menu w bardziej elastyczny sposób. Przykładem niech będzie iterowanie wyłącznie menu deserowego restauracji Diner albo iterowanie całego menu restauracji, łącznie z podmenu.



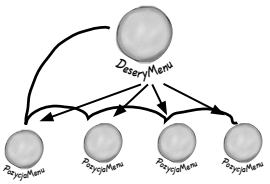
Ponieważ musimy zapewnić reprezentację menu, zagnieżdżonych podmenu i pozycji menu, wprowadzenie struktury drzewiastej jest naturalnym krokiem.



Wciąż wymagana jest możliwość iterowania wszystkich elementów drzewa.



Wymagamy również bardziej wyszukanego iterowania, na przykład pojedynczego menu.





WYTĘŻ UMYSŁ

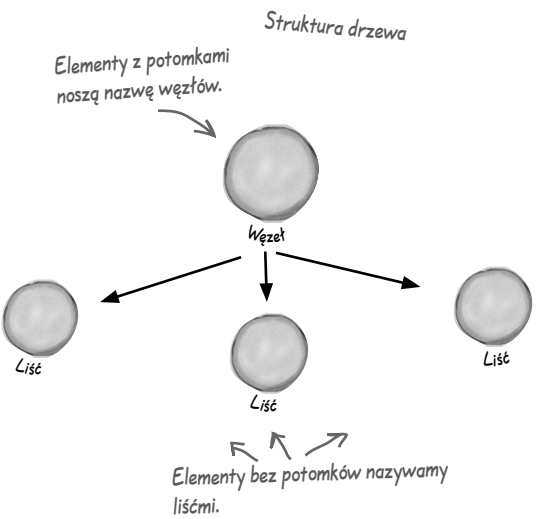
Jak rozwiązałbyś problem nowych wymagań? Pomyśl o tym, zanim przejdziesz do kolejnej strony.

Wzorzec Kompozyt

Zgadza się, podczas rozwiązywania tego problemu przedstawimy Ci nowy wzorzec projektowy. Nie rezygnujemy z wzorca Iterator — będzie on częścią nowego rozwiązania — ale w tym przypadku problem zarządzania wieloma menu ma nowy wymiar, którego same iteratory nie mogą rozwiązać. Cofniemy się więc o krok i wprowadzimy wzorzec Kompozyt.

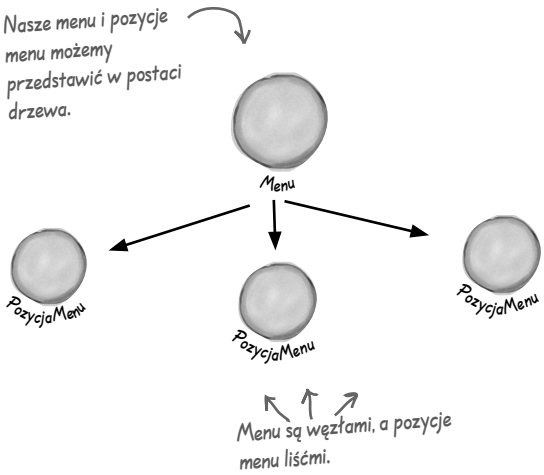
Aby nie rozwodzić się nad nim zbytnio, przedstawimy od razu jego oficjalną definicję:

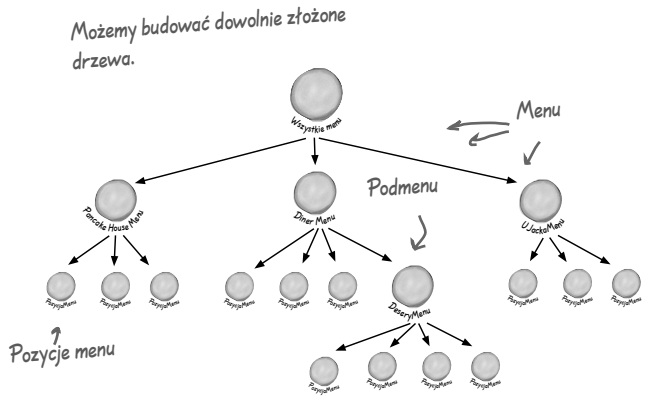
> **Wzorzec Kompozyt (ang. composite, obiekt złożony)** pozwala łączyć obiekty w struktury drzewiaste, które reprezentują hierarchie część-całość. Jego zastosowanie pozwala klientom jednolicie obsługiwać zarówno pojedyncze obiekty, jak i ich kompozycje.

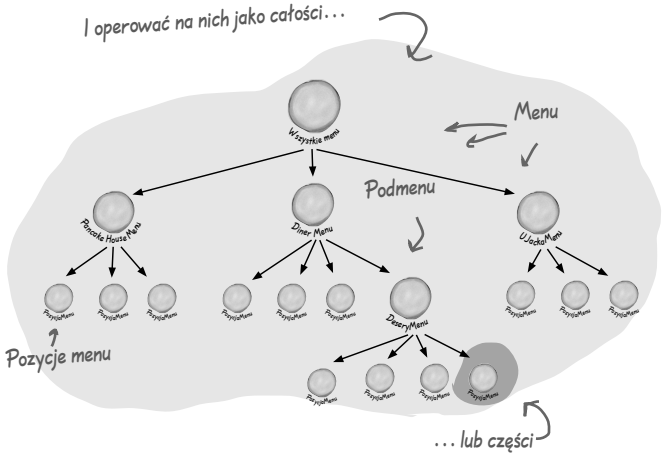


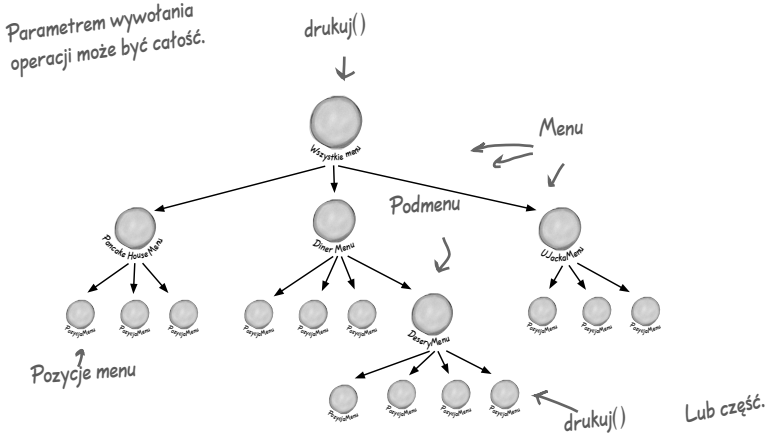
Rozważmy ten wzorzec na przykładzie naszych menu: uzyskujemy możliwość utworzenia struktury drzewiastej, która może obsługiwać na tych samych zasadach grupę zagnieżdżonych menu i ich pozycji. Umieszczając menu i ich pozycje w tej samej strukturze, tworzymy hierarchię część-całość — drzewo obiektów, które jest zbudowane z części (menu i pozycji menu), ale może być traktowane jako całość, czyli jak jedno duże menu.

Kiedy już mamy takie duże, złożone menu, możemy wykorzystać ten wzorzec, aby „jednolicie obsługiwać zarówno pojedyncze obiekty, jak i ich kompozycje". Co to znaczy? To znaczy, że jeżeli mamy drzewiastą strukturę menu, podmenu i (być może) podpodmenu wraz z pozycjami menu, to dowolne menu jest „kompozycją", ponieważ może zawierać zarówno inne menu, jak i pozycje menu. Pojedyncze obiekty są po prostu pozycjami menu — nie przechowują innych obiektów. Zaraz się przekonamy, że wprowadzenie rozwiązania opartego na wzorcu Kompozyt umożliwi nam napisanie prostego kodu, który wykona tę samą operację (na przykład drukowanie) na wszystkich elementach struktury menu.





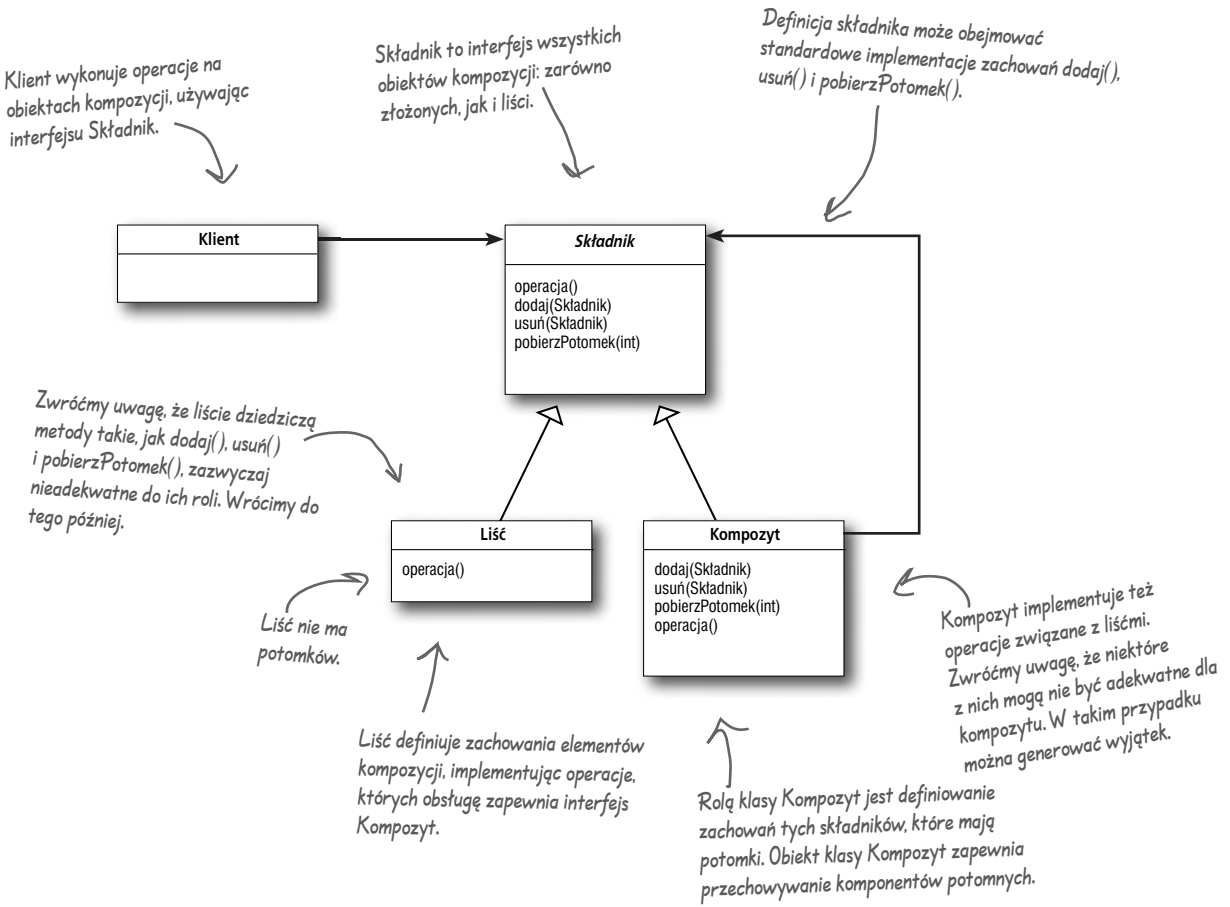




Wzorzec Kompozyt umożliwia budowanie struktur obiektów będących drzewami, których węzłami są zarówno kompozycje obiektów, jak i pojedyncze elementy.

Używając takiej struktury kompozytowej, możemy wykonywać operacje zarówno na całości, jak i na pojedynczych obiektach. Innymi słowy, w większości przypadków możemy zignorować różnice między kompozycjami obiektów a indywidualnymi obiektami.

Diagram klas **wzorca Kompozyt**



Nie istnieją głupie pytania

P: Składnik, kompozyt, drzewa? Pomieszało mi się...

O: Kompozyt (ang. *composite*) zawiera składniki (ang. *component*). Składniki mają dwie odmiany: kompozyty i liście (ang. *leaf*). Oczywiście, mamy tu rekurencję. Kompozyt przechowuje zbiór potomków, którymi mogą być inne kompozyty lub liście.

Gdy porządkujemy dane w taki sposób, powstaje struktura drzewiasta (właściwie jest to drzewo ustawione korzeniem do góry). Korzeniem jest kompozyt, który rozgałęzia się poprzez kolejne kompozyty zakończone liśćmi.

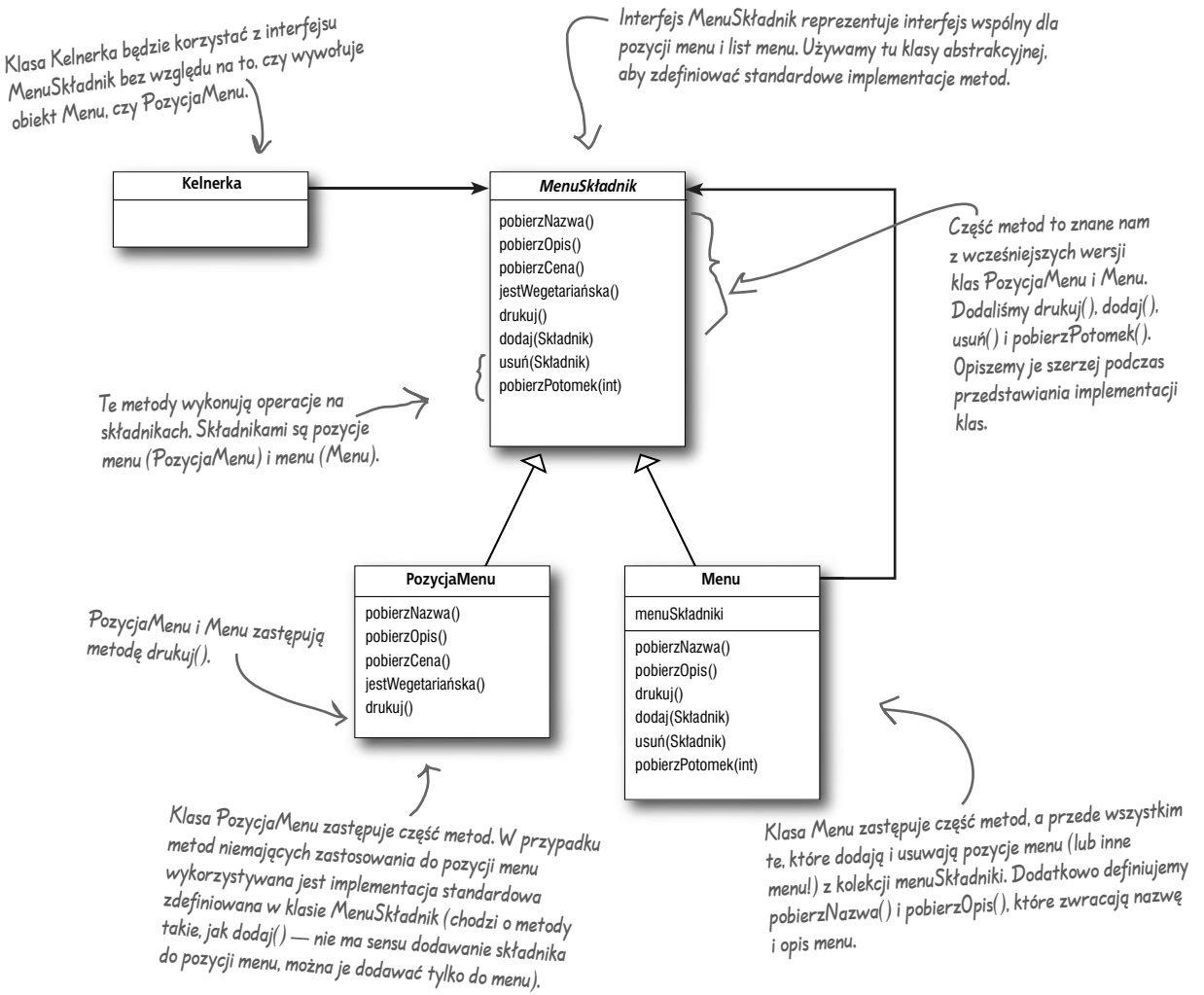
P: A co z iteratorami?

O: Pamiętajmy, że rozpoczynamy tworzenie rozwiązania opartego na innym podejściu. Menu zostaną zaimplementowane ponownie z wykorzystaniem wzorca Kompozyt. Nie należy więc szukać żadnych magicznych przekształceń iteratora w kompozyt. Oba wzorce będą wykorzystane jednocześnie. Jak wkrótce się przekonamy, w implementacji opartej na wzorcu Kompozyt iteratory można wykorzystać na kilka sposobów.

Projektujemy menu oparte na wzorcu Kompozyt

Jak więc zastosować wzorzec Kompozyt w naszym konkretnym przykładzie? Rozpoczniemy od zdefiniowania interfejsu składnika. Będzie to wspólny interfejs menu i pozycji menu, umożliwiający traktowanie obu w taki sam sposób. Innymi słowy, będziemy mogli wywoływać *tę samą* metodę menu i pozycji menu.

W przypadku pewnych metod wywoływanie ich dla pozycji menu lub samego menu może nie mieć *sensu*. Zajmiemy się tym za chwilę. Tymczasem przyjrzyjmy się diagramowi, który ilustruje, jak wzorzec Kompozyt może być zastosowany w naszych menu:



Implementacja klasy MenuSkładnik

Pracę z kodem rozpoczniemy od klasy abstrakcyjnej MenuSkładnik. Pamiętaj, że rolą klasy MenuSkładnik jest określenie interfejsu węzłów-liści i węzłów kompozytowych. Być może zadajesz sobie teraz pytanie" „Czy klasa MenuSkładnik nie odgrywa tutaj dwóch różnych ról?". Jest to pytanie uzasadnione i wrócimy do niego później. W tym momencie stworzymy jedynie domyślne implementacje metod, zapewniające, że gdy klasa PozycjaMenu (liść) lub Menu (kompozyt) nie implementuje jednej z nich (jak pobierzPotomek() w przypadku liścia), istnieje pewne zachowanie standardowe.

Wszystkie składniki muszą implementować interfejs MenuSkładnik. Ponieważ jednak rola liści i węzłów jest inna, nie możemy w każdym przypadku definiować domyślnej implementacji każdej metody. W pewnych przypadkach najlepsze, co możemy zrobić, to generowanie wyjątku.

Ponieważ część z tych metod ma sens tylko w przypadku właściwych pozycji menu, a inne — tylko w przypadku podmenu, standardowa implementacja to wyjątek UnsupportedOperationException. Dzięki temu, jeżeli klasa PozycjaMenu lub Menu nie zapewnia wykonania pewnej operacji, nie musi ona być definiowana. Klasa korzysta wówczas z dziedziczenia implementacji domyślnej.

Klasa MenuSkładnik zapewnia domyślną implementację każdej z metod.

```
public abstract class MenuSkładnik {

    public void dodaj(MenuSkładnik menuSkładnik) {
        throw new UnsupportedOperationException();
    }
    public void usuń(MenuSkładnik menuSkładnik) {
        throw new UnsupportedOperationException();
    }
    public MenuSkładnik pobierzPotomek(int i) {
        throw new UnsupportedOperationException();
    }

    public String pobierzNazwa() {
        throw new UnsupportedOperationException();
    }
    public String pobierzOpis() {
        throw new UnsupportedOperationException();
    }
    public double pobierzCena() {
        throw new UnsupportedOperationException();
    }
    public boolean jestWegetariańska() {
        throw new UnsupportedOperationException();
    }

    public void drukuj() {
        throw new UnsupportedOperationException();
    }
}
```

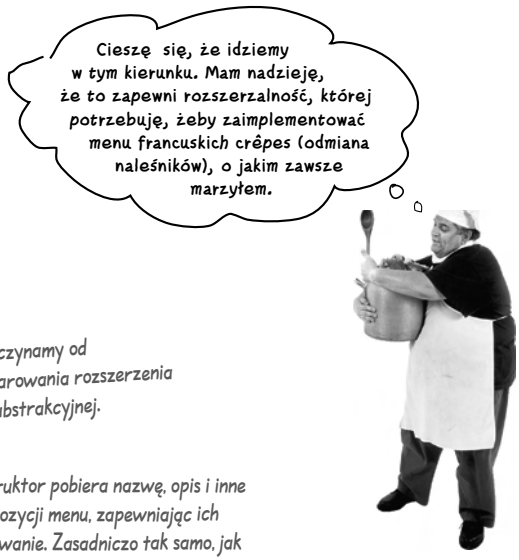
Zestawiliśmy razem metody „kompozytowe", czyli te, które dodają, usuwają i pobierają pozycje menu.

To jest grupa metod „operacyjnych". Są one wykorzystywane przez pozycje menu. Okazuje się też, że część z nich można wykorzystać w klasie Menu, o czym przekonamy się nieco dalej, gdy zajmiemy się jej implementacją.

drukuj() to metoda „operacyjna", którą implementują obie klasy, Menu i PozycjaMenu. Nie przeszkadza to jednak wprowadzeniu prostej implementacji standardowej.

Implementacja klasy PozycjaMenu

Kolejną klasą będzie PozycjaMenu. Pamiętaj, że jest to klasa typu liść, implementująca zachowania elementów kompozytu.




```
public class PozycjaMenu extends MenuSkładnik {
    String nazwa;
    String opis;
    boolean wegetariańska;
    double cena;

    public PozycjaMenu(String nazwa,
                       String opis,
                       boolean wegetariańska,
                       double cena)
    {
        this.nazwa = nazwa;
        this.opis = opis;
        this.wegetariańska = wegetariańska;
        this.cena = cena;
    }

    public String pobierzNazwa() {
        return nazwa;
    }

    public String pobierzOpis() {
        return opis;
    }

    public double pobierzCena() {
        return cena;
    }

    public boolean jestWegetariańska() {
        return wegetariańska;
    }

    public void drukuj() {
        System.out.print("  " + pobierzNazwa());
        if (jestWegetariańska()) {
            System.out.print("(w)");
        }
        System.out.println(", " + pobierzCena());
        System.out.println("     -- " + pobierzOpis());
    }
}
```

Rozpoczynamy od zadeklarowania rozszerzenia klasy abstrakcyjnej.

Konstruktor pobiera nazwę, opis i inne dane pozycji menu, zapewniając ich składowanie. Zasadniczo tak samo, jak w naszej wcześniejszej implementacji.

Gettery — takie same jak wcześniej.

Ta część różni się od wcześniejszej implementacji. Zastępujemy metodę drukuj() klasy MenuSkładnik. W przypadku pozycji menu metoda ta wyświetla komplet danych pozycji menu: nazwę, opis, cenę i informację o tym, czy potrawa jest wegetariańska.

Implementacja klasy Menu

Następnym krokiem będzie implementacja klasy kompozytowej, którą ponownie nazwiemy Menu. Pamiętajmy, że klasa kompozytowa może przechowywać zarówno obiekty PozycjaMenu, jak i Menu. Pewnych metod klasy MenuSkładnik nie będziemy tu implementować: pobierzCena() i jestWegetariańska(). Nie są one adekwatne dla obiektów typu menu.

Klasa Menu, podobnie jak PozycjaMenu, dziedziczy po klasie MenuSkładnik.

Menu może mieć dowolną liczbę potomków typu MenuSkładnik. Będziemy je przechowywać w kolekcji ArrayList.

```
public class Menu extends MenuSkładnik {
    ArrayList menuSkładniki = new ArrayList();
    String nazwa;
    String opis;

    public Menu(String nazwa, String opis) {
        this.nazwa = nazwa;
        this.opis = opis;
    }

    public void dodaj(MenuSkładnik menuSkładnik) {
        menuSkładniki.add(menuSkładnik);
    }

    public void usuń(MenuSkładnik menuSkładnik) {
        menuSkładniki.remove(menuSkładnik);
    }

    public MenuSkładnik pobierzPotomek(int i) {
        return (MenuSkładnik)menuSkładniki.get(i);
    }

    public String pobierzNazwa() {
        return nazwa;
    }

    public String pobierzOpis() {
        return opis;
    }

    public void drukuj() {
        System.out.print("\n" + pobierzNazwa());
        System.out.println(", " + pobierzOpis());
        System.out.println("---------------------");
    }
}
```

Konstruktor różni się od wcześniejszego: każde menu będzie miało nazwę i opis. Wcześniej różniły się one klasą.

Tutaj dodajemy pozycje menu lub inne menu do kolekcji. Ponieważ zarówno obiekty PozycjaMenu, jak i Menu są obiektami MenuSkładnik, do obsługi obu typów wystarczy jedna metoda.

Można też usuwać i pobierać elementy kolekcji.

Metody typu getter do pobierania nazwy i opisu.

Zwróćmy uwagę, że nie zastępujemy metod pobierzCena() i jestWegetariańska(), ponieważ nie mają one odniesienia do menu (choć, oczywiście, jestWegetariańska() mogłaby kiedyś znaleźć zastosowanie). Próba wywołania jednej z nich doprowadzi do wyjątku UnsupportedOperationException.

Wyświetlanie menu polega na wyświetleniu jego nazwy i opisu.



Chwileczkę, nie rozumiem implementacji metody drukuj(). Myślałem, że miała na kompozycie wykonywać te same operacje, co na liściu. Przy takiej implementacji uzyskam jedynie nazwę menu i jego opis. Metoda nie wyświetli danych KOMPOZYTU.

Doskonałe spostrzeżenie. Ponieważ menu jest kompozytem i zawiera zarówno pozycje menu, jak i inne menu, jego metoda drukuj() powinna wyświetlać całą zawartość struktury. W przeciwnym wypadku iterowanie całego kompozytu i wyświetlanie każdego elementu musiałby przeprowadzać klient. Wprowadzanie struktury kompozytowej nie miałoby wówczas żadnego sensu.

Zaraz się przekonamy, że właściwe zaimplementowanie metody drukuj() jest bardzo proste, ponieważ możemy wykorzystać to, że każdy komponent potrafi wyświetlić swoje własne dane. Całość jest idealnie rekurencyjna i w ogóle cudowna. Zobaczmy:

Poprawianie metody drukuj()

```
public class Menu extends MenuSkładnik {
    ArrayList menuSkładniki = new ArrayList();
    String nazwa;
    String opis;

    // kod konstruktora

    // kod innych metod

    public void drukuj() {
        System.out.print("\n" + pobierzNazwa());
        System.out.println(", " + pobierzOpis());
        System.out.println("---------------------");

        Iterator iterator = menuSkładniki.iterator();
        while (iterator.hasNext()) {
            MenuSkładnik menuSkładnik =
                (MenuSkładnik)iterator.next();
            menuSkładnik.drukuj();
        }
    }
}
```

Wszystko, co musimy zmienić, to metoda drukuj(), która powinna wyświetlać nie tylko informacje o menu, ale również jego składniki: inne menu i pozycje menu.

Oto jest! Wykorzystaliśmy iterator. Zapewnia iterowanie wszystkich składników menu… czyli pozycji menu lub podmenu. Ponieważ zarówno menu, jak i pozycje menu dysponują metodą drukuj(), wystarczy ją tylko wywołać.

UWAGA: jeżeli w trakcie iterowania napotkamy inny obiekt typu Menu, jego metoda drukuj() rozpocznie kolejny cykl iterowania.

Przygotowanie do testów...

Już za chwilę zajmiemy się kodem sprawdzającym naszą implementację. Musimy jeszcze uaktualnić kod klasy Kelnerka. Jak by nie patrzeć, jest to główny klient naszych nowych klas:

```
public class Kelnerka {
    MenuSkładnik wszystkieMenu;

    public Kelnerka(MenuSkładnik wszystkieMenu)
    {
        this.wszystkieMenu = wszystkieMenu;
    }

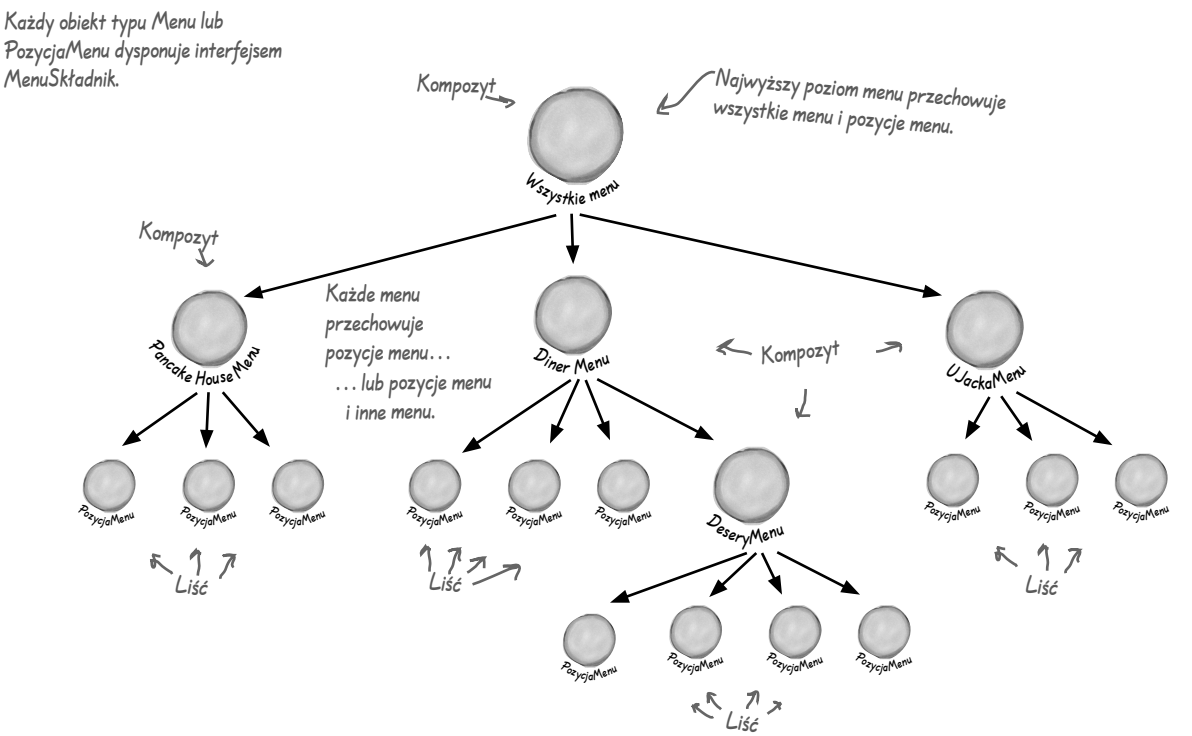
    public void drukujMenu() {
        wszystkieMenu.drukuj();
    }
}
```

Tak właśnie! Kod kelnerki jest naprawdę taki prosty. Teraz przekazujemy jej tylko główny składnik kompozycji, zawierający wszystkie pozostałe menu. Nazwiemy go wszystkieMenu.

Wszystko, co musi zrobić, to wydrukować całą strukturę — wszystkie menu i pozycje menu. Zapewni to metoda drukuj() głównego składnika kompozycji.

Naprawdę uszczęśliwimy kelnerkę.

I jeszcze jedna rzecz. Spójrzmy, jak będzie wyglądała kompozytowa struktura menu w trakcie pracy programu:



A teraz testujemy...

Teraz potrzebujemy właściwej procedury testowej. W przeciwieństwie do poprzedniej wersji wprowadzimy tu wszystkie czynności niezbędne do utworzenia menu. Możemy, oczywiście, poprosić szefów kuchni, aby dostarczyli nam swoje menu, ale lepiej najpierw przeprowadzić testy. Oto kod:

```
public class MenuTest {
    public static void main(String args[]) {
        MenuSkładnik pancakeHouseMenu =
            new Menu("MENU PANCAKE HOUSE", "Śniadania");
        MenuSkładnik dinerMenu =
            new Menu("MENU RESTAURACJI DINER", "Lunch");
        MenuSkładnik uJackaMenu =
            new Menu("MENU U JACKA", "Obiady");
        MenuSkładnik deseryMenu =
            new Menu("MENU DESERÓW", "Desery, oczywiście!");

        MenuSkładnik wszystkieMenu = new Menu("WSZYSTKIE MENU", "Połączone menu");

        wszystkieMenu.dodaj(pancakeHouseMenu);
        wszystkieMenu.dodaj(dinerMenu);
        wszystkieMenu.dodaj(uJackaMenu);

        // kod dodawania elementów menu

        dinerMenu.dodaj(new PozycjaMenu(
            "Spaghetti",
            "Makaron spaghetti z sosem marinara i kromka chleba na zakwasie",
            true,
            3.89));

        dinerMenu.dodaj(deseryMenu);

        deseryMenu.dodaj(new PozycjaMenu(
            "Szarlotka",
            "Szarlotka z lodami waniliowymi",
            true,
            1.59));

        // dalsze pozycje menu

        Kelnerka kelnerka = new Kelnerka(wszystkieMenu);

        kelnerka.drukujMenu();
    }
}
```

Najpierw utworzymy wszystkie obiekty menu.

Potrzebujemy też korzenia naszego drzewa, który nazwiemy wszystkieMenu.

Używamy metody dodaj() kompozytu, aby dodać poszczególne menu do listy zbiorczej o nazwie wszystkieMenu.

Teraz dodajemy wszystkie pozycje menu. Tutaj widzimy tylko przykład. W kompletnym kodzie źródłowym muszą być zapisane wszystkie elementy.Teraz dodajemy wszystkie pozycje menu. Tutaj widzimy tylko przykład. W kompletnym kodzie źródłowym muszą być zapisane wszystkie elementy.

Do jednego menu dodajemy inne menu. Jedyne, co ma znaczenie dla klasy dinerMenu, to żeby każdy dodawany element dysponował interfejsem klasy MenuSkładnik.

W menu deserowym będzie szarlotka...

Po zbudowaniu całej hierarchii menu przekazujemy ją kelnerce. Wyświetlenie nowej oferty będzie już banalne.

I po testach...

UWAGA: wydruk powstał przy użyciu pełnego kodu źródłowego.

```
F:\WINDOWS\system32\cmd.exe
F:\1pawel\helion\Head First Design Patterns\hfdp\classes>java headfirst.composit
e.menu.MenuTest
WSZYSTKIE MENU, Połączone menu
---------------------

MENU PANCAKE HOUSE, Śniadania
---------------------
  Śniadanie naleśnikowe K&B's(w), 2.99
     -- Naleśniki z jajecznicą i tostem
  Śniadanie naleśnikowe zwykłe, 2.99
     -- Naleśniki z jajkiem sadzonym i kiełbasą
  Naleśniki z jagodami(w), 3.49
     -- Naleśniki ze świeżymi jagodami
  Wafle nadziewane(w), 3.59
     -- Wafle z jagodami lub truskawkami

MENU RESTAURACJI DINER, Lunch
---------------------
  Wegetariańska kanapka BSP(w), 2.99
     -- (Wegetariański) Boczek z sałatą i pomidorem, chleb pszenny pełnoziarnist
y
  Kanapka BSP, 2.99
     -- Boczek z sałatą i pomidorem, chleb pszenny pełnoziarnisty
  Zupa dnia, 3.29
     -- Filiżanka zupy dnia, sałatka
  Hot-dog, 3.05
     -- Hot-dog z kapustą kiszoną, rzodkiewką, cebulą i dodatkiem sera
  Warzywa na parze i brązowy ryż(w), 3.99
     -- Mieszanka warzyw gotowanych na parze i porcja pełnowartościowego ryżu
  Spaghetti(w), 3.89
     -- Makaron spaghetti z sosem marinara i kromka chleba na zakwasie

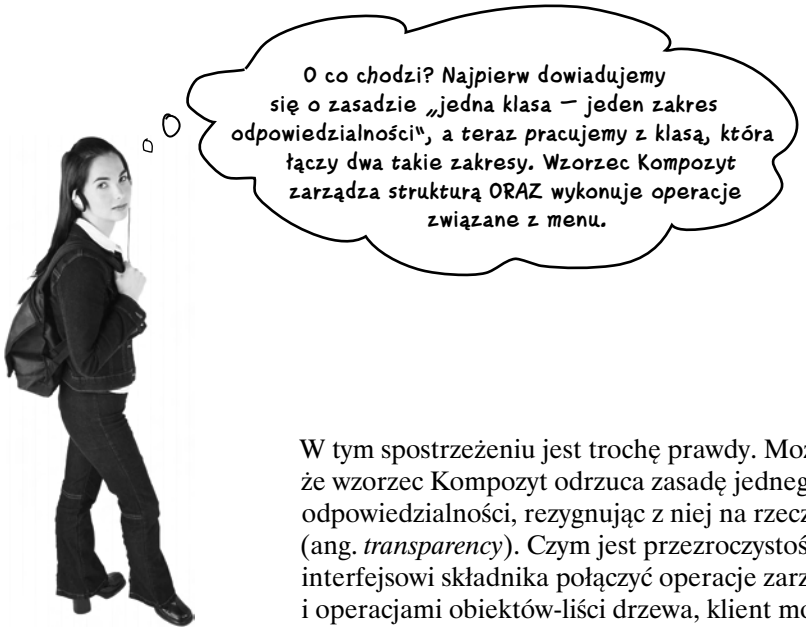
MENU DESERÓW, Desery, oczywiście!
---------------------
  Szarlotka(w), 1.59
     -- Szarlotka z lodami waniliowymi
  Sernik(w), 1.99
     -- Sernik nowojorski w polewie czekoladowej
  Sorbet(w), 1.89
     -- Poziomki z limonką

MENU CAFE, Obiady
---------------------
  Kanapka wegetariańska z frytkami(w), 3.99
     -- Kanapka wegetariańska z sałatą i pomidorem, frytki
  Zupa dnia, 3.69
     -- Filiżanka zupy dnia, sałatka
  Burito(w), 4.29
     -- Duże burito z łaciatą fasolą, sosem salsa i guacamole (awokado)

F:\1pawel\helion\Head First Design Patterns\hfdp\classes>
```

Wyświetlamy wszystkie menu... Wystarczyło wywołać jedną metodę drukuj().

Nowe menu deserów jest drukowane razem z elementami menu restauracji Diner.



W tym spostrzeżeniu jest trochę prawdy. Można stwierdzić, że wzorzec Kompozyt odrzuca zasadę jednego zakresu odpowiedzialności, rezygnując z niej na rzecz *przezroczystości* (ang. *transparency*). Czym jest przezroczystość? Otóż, pozwalając interfejsowi składnika połączyć operacje zarządzania potomkami i operacjami obiektów-liści drzewa, klient może jednolicie traktować zarówno kompozyty, jak i elementy samodzielne. To, czy element jest kompozytem, czy liściem, pozostaje dla niego „przezroczyste".

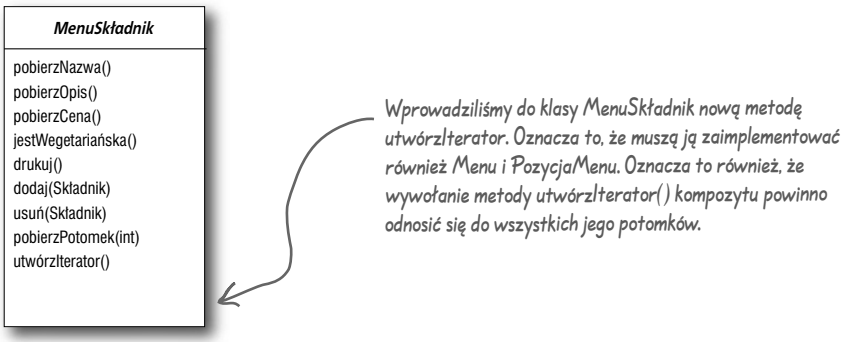
Umieszczając oba typy operacji w klasie składnika, tracimy nieco *bezpieczeństwa*, ponieważ klient może próbować wykonać operacje niewłaściwe lub nieadekwatne dla danego elementu (na przykład dodawanie menu do pozycji menu). Jest to decyzja konstrukcyjna. Mogliśmy skierować projekt w innym kierunku i oddzielić zakresy odpowiedzialności przy użyciu interfejsów. Zapewniłoby to bezpieczeństwo rozwiązania w tym znaczeniu, że każde niepoprawne wywołanie elementu zostałoby przechwycone w czasie kompilacji lub w czasie wykonywania. Utracilibyśmy jednak przezroczystość i nasz kod musiałby korzystać z instrukcji warunkowych oraz operatora `instanceof`.

Powracając do pytania: jest to typowy przypadek, w którym wybieramy „coś za coś". Kierujemy się zawsze zasadami projektowania, ale musimy jednocześnie zwracać uwagę na to, jaki mają one wpływ na budowane rozwiązanie. Niekiedy trzeba wprowadzić elementy, które wydają się łamać te zasady. W wielu przypadkach jest to jednak kwestia punktu widzenia. Przykładowo, obecność operacji zarządzania potomkami w obiektach-liściach może wydawać się błędem (chodzi o operacje takie, jak dodaj(), usuń() i pobierzPotomek()). Lecz jeśli spojrzymy na to z innej strony, liść jest po prostu węzłem bez potomków.

Powracamy do iteratora

Kilka stron wcześniej obiecaliśmy, że przedstawimy współpracę wzorców Iterator i Kompozyt. Widzieliśmy już iterator w wewnętrznej implementacji metody drukuj(). Możemy również umożliwić kelnerce iterowanie całego kompozytu. Może to być przydatne do przeglądania wszystkich dostępnych potraw i wybierania tych, które należą do grupy wegetariańskich.

Aby zaimplementować iterator kompozytu, dodamy do każdego składnika metodę utwórzIterator(). Rozpoczniemy od klasy abstrakcyjnej MenuSkładnik:



Teraz będziemy implementować tę metodę w klasach Menu i PozycjaMenu:

```
public class Menu extends MenuSkładnik {

    // pozostały kod bez zmian

    public Iterator utwórzIterator() {
        return new IteratorKompozytu(menuSkładniki.iterator());
    }

}


public class PozycjaMenu extends MenuSkładnik {

    // pozostały kod bez zmian

    public Iterator utwórzIterator() {
        return new IteratorPusty();
    }

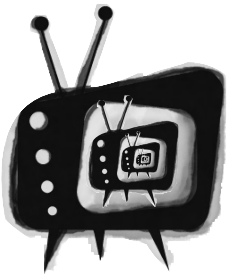
}
```

Wykorzystujemy tutaj nową klasę iteratora o nazwie IteratorKompozytu. Jej funkcją jest iterowanie dowolnego kompozytu. Przekazujemy jej iterator kompozytu bieżącego.

A teraz pozycja menu...

A cóż to? Co to jest IteratorPusty? Wyjaśnimy to dwie strony dalej.

Iterator kompozytu



UWAGA: STREFA REKURENCJI

Klasa IteratorKompozytu będzie POWAŻNYM iteratorem. Jej zadaniem jest iterowanie elementów PozycjaMenu kolekcji MenuSkładnik z uwzględnieniem iterowania wszystkich menu potomnych (i ich menu potomnych itd.).

Poniżej przedstawiamy odpowiedni kod. Nie jest on rozbudowany, ale jego analiza może zająć nieco czasu. Możemy sobie wówczas powtarzać: „rekurencja jest moim przyjacielem, rekurencja jest moim przyjacielem...".

Podobnie jak we wszystkich iteratorach, implementujemy interfejs java.util.Iterator.

```
import java.util.*;

public class IteratorKompozytu implements Iterator {
    Stack stos = new Stack();

    public IteratorKompozytu(Iterator iterator) {
        stos.push(iterator);
    }

    public Object next() {
        if (hasNext()) {
            Iterator iterator = (Iterator) stos.peek();
            MenuSkładnik składnik = (MenuSkładnik) iterator.next();
            if (składnik instanceof Menu) {
                stos.push(składnik.createIterator());
            }
            return składnik;
        } else {
            return null;
        }
    }

    public boolean hasNext() {
        if (stos.empty()) {
            return false;
        } else {
            Iterator iterator = (Iterator) stos.peek();
            if (!iterator.hasNext()) {
                stos.pop();
                return hasNext();
            } else {
                return true;
            }
        }
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

Przekazujemy iterator najwyższego poziomu iterowanego kompozytu. Zostaje on umieszczony w strukturze typu stos.

Jeśli klient żąda kolejnego elementu, rozpoczynamy od sprawdzenia, czy taki istnieje, korzystając z wywołania hasNext()...

Jeśli kolejny element istnieje, pobieramy bieżący iterator ze stosu i kolejny element z iteratora.

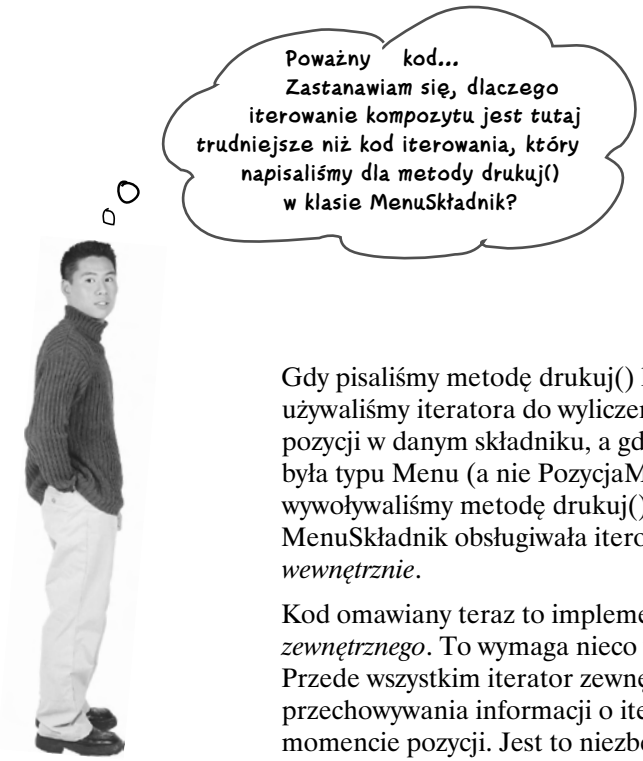
Jeżeli ten element to menu, mamy do czynienia z kolejnym kompozytem, który również musi być iterowany. Umieszczamy go więc na stosie. W każdym przypadku zwracanym obiektem będzie MenuSkładnik.

Aby sprawdzić, czy istnieje kolejny element, weryfikujemy stan stosu. Jeżeli jest on pusty, nie ma takiego elementu.

Jeżeli stos nie jest pusty, pobieramy z niego iterator i sprawdzamy, czy ten ma kolejny element. Jeżeli nie, usuwamy go ze stosu i ponownie wywołujemy hasNext().

W pozostałych przypadkach kolejny element istnieje, więc zwracaną wartością jest true.

Nie implementujemy operacji usuwania elementów.



Gdy pisaliśmy metodę drukuj() klasy MenuSkładnik, używaliśmy iteratora do wyliczenia wszystkich pozycji w danym składniku, a gdy pewna pozycja była typu Menu (a nie PozycjaMenu), rekurencyjnie wywoływaliśmy metodę drukuj(). Innymi słowy, klasa MenuSkładnik obsługiwała iterowanie samodzielnie, *wewnętrznie*.

Kod omawiany teraz to implementacja iteratora *zewnętrznego*. To wymaga nieco więcej pracy. Przede wszystkim iterator zewnętrzny wymaga przechowywania informacji o iterowanej w danym momencie pozycji. Jest to niezbędne, aby klient mógł korzystać z metod hasNext() i next() podczas kontynuowania procesu iterowania. W przypadku iteratora kompozytu niezbędne jest przechowywanie informacji o pozycji w strukturze rekurencyjnej. Stąd właśnie wykorzystanie stosu do przechowywania dodatkowych informacji, gdy przechodzimy pomiędzy różnymi poziomami hierarchii.



WYTĘŻ UMYSŁ

Narysuj diagram obiektów Menu i PozycjaMenu. Następnie prześledź pracę klasy IteratorKompozytu, gdy wywoływane są metody hasNext() i next(). Powinno to odpowiadać wykonywaniu następującego kodu:

```
public void testIteratorKompozytu(MenuSkładnik składnik) {
    IteratorKompozytu iterator = new IteratorKompozytu(składnik.iterator);

    while(iterator.hasNext()) {
        MenuSkładnik składnik = iterator.next();
    }
}
```

IteratorPusty

Czas wyjaśnić, jaką rolę pełni w naszym kodzie klasa IteratorPusty. Jej znaczenie można wyjaśnić w prosty sposób: obiekt PozycjaMenu nie ma elementów do iterowania, prawda? Jak więc zaimplementujemy metodę utwórzIterator() takiego składnika? Mamy dwie możliwości:

UWAGA: kolejny przykład „wzorca" Null Object.

Opcja pierwsza:

Zwrócić null

Wywołanie utwórzIterator() może zwracać null. Wówczas jednak w kodzie klienta trzeba wprowadzać instrukcje warunkowe, sprawdzające, czy wywołanie zwróciło null.

Opcja druga:

Zwrócić iterator, który przy każdym wywołaniu hasNext() zwraca false.

To wydaje się lepszym pomysłem. Zwracamy iterator, więc klient nie musi sprawdzać, czy wywołanie zwróciło null. Iterator musi być tak skonstruowany, aby nie prowadził do wykonania żadnych operacji.

Druga opcja jest zdecydowanie korzystniejsza. Poniżej przedstawiamy implementację klasy IteratorPusty.

To najbardziej leniwy iterator, jaki można spotkać. Każde wywołanie natychmiast zwraca kontrolę.

```
import java.util.Iterator;

public class IteratorPusty implements Iterator {

    public Object next() {
        return false;
    }

    public boolean hasNext() {
        return false;
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

Wywołanie next() prowadzi do zwrócenia null.

Co najważniejsze, wywołanie hasNext() prowadzi do zwrócenia wartości false.

Oczywiście, nie można usuwać elementów z iteratora pustego.

Menu dla wegetarian

Mamy już sposób, żeby iterować wszystkie pozycje wszystkich menu. Skorzystajmy z niego i stwórzmy dla kelnerki metodę, która umożliwi zwrócenie wszystkich pozycji wegetariańskich.

```
public class Kelnerka {
    MenuSkładnik wszystkieMenu;

    public Kelnerka(MenuSkładnik wszystkieMenu) {
        this.wszystkieMenu = wszystkieMenu;
    }

    public void drukujMenu() {
        wszystkieMenu.drukuj();
    }

    public void drukujMenuWegetariańskie() {
        Iterator iterator = wszystkieMenu.utwórzIterator();

        System.out.println("\nMENU WEGETARIAŃSKIE\n----");
        while (iterator.hasNext()) {
            MenuSkładnik menuSkładnik =
                    (MenuSkładnik)iterator.next();
            try {
                if (menuSkładnik.jestWegetariańska()) {
                    menuSkładnik.drukuj();
                }
            } catch (UnsupportedOperationException e) {}
        }
    }
}
```

Metoda drukujMenuWegetariańskie() pobiera iterator kompozytu wszystkieMenu. Jest to obiekt utworzonej wcześniej klasy IteratorKompozytu.

Iterujemy całość kompozytu.

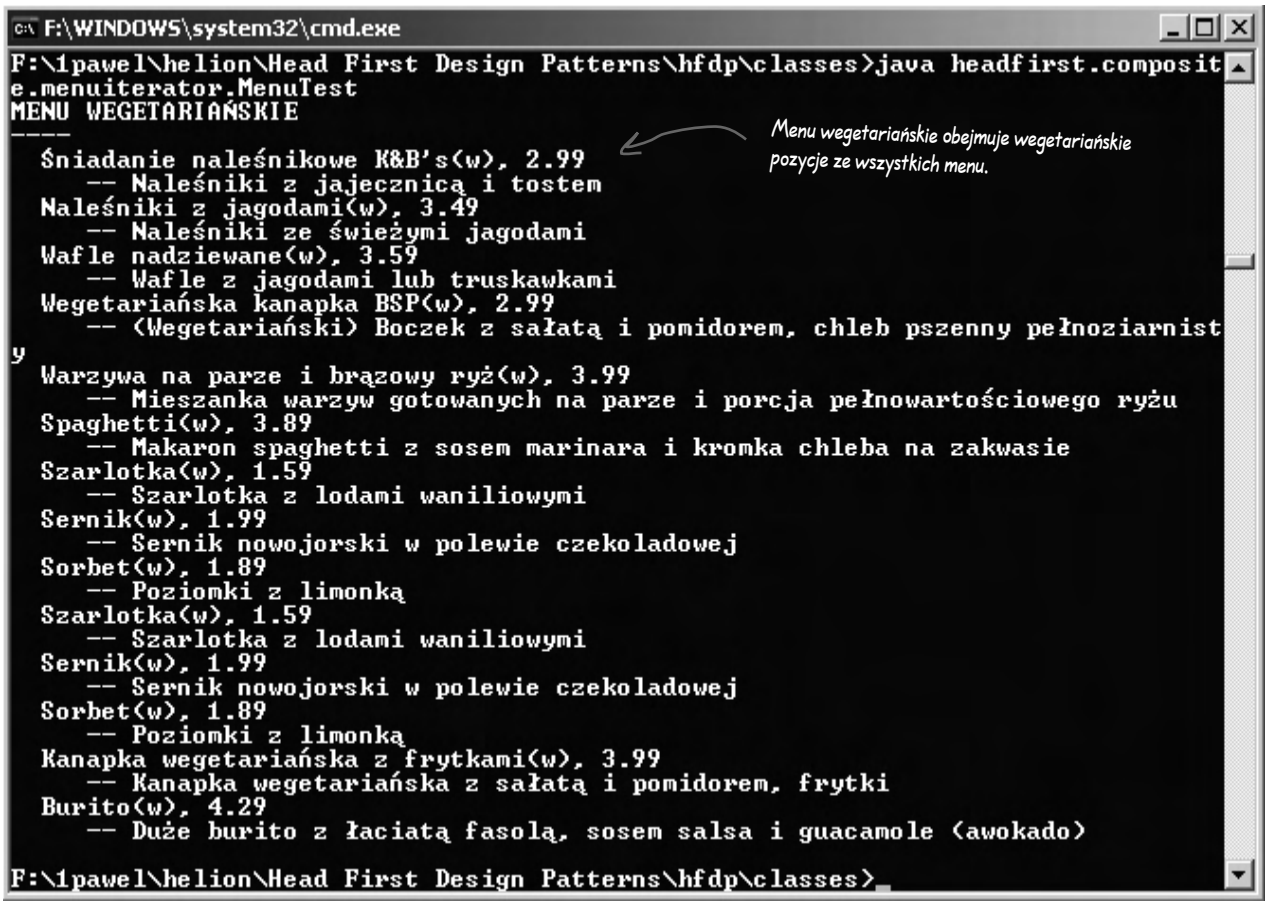
Wywołujemy metodę jestWegetariańska() poszczególnych elementów. Jeżeli zwraca ona wartość true, wywołujemy metodę drukuj() tego elementu.

Metoda drukuj() jest wywoływana wyłącznie dla pozycji menu, nigdy dla kompozytów. Czy rozumiesz, dlaczego?

Metoda jestWegetariańska() klasy Menu zawsze generuje wyjątek. Musi on zostać przechwycony, aby iterowanie było kontynuowane.

Wzorce Iterator i Kompozyt razem...

Uff! Doprowadzenie kodu do obecnej postaci zajęło trochę czasu. Nowa, uogólniona struktura menu powinna nadążać za rozwojem restauracji Diner przynajmniej przez pewien czas. Możemy się zrelaksować i zamówić coś wegetariańskiego:





Zauważyłem, że w metodzie drukujMenuWegetariańskie() instrukcja try-catch została wykorzystana do obsługi logiki menu, które nie zapewniają możliwości korzystania z metody jestWegetariańska(). Zawsze słyszałem, że nie jest to dobry styl programowania.

Przyjrzyjmy się tej części kodu, która budzi wątpliwości:

```
try {
    if (menuSkładnik.jestWegetariańska()) {
        menuSkładnik.drukuj();
    }
} catch (UnsupportedOperationException e) {}
```

Wywołujemy metodę jestWegetariańska() dla każdego obiektu MenuSkładnik. Menu generują wyjątek, bo nie zapewniają takiej operacji.

Jeżeli składnik menu nie zapewnia obsługi operacji, pomijamy go.

Ogólnie trudno nie zgodzić się z taką uwagą. Instrukcja **try-catch** ma zapewnić obsługę błędów, a nie realizować logikę programu. Jakie mamy inne możliwości? Moglibyśmy sprawdzać typ składnika menu w trakcie wykonywania programu, korzystając z operatora `instanceof`. Wówczas przed wywołaniem jestWegetariańska() upewnilibyśmy się, że dana pozycja to element typu PozycjaMenu. Postępując w ten sposób, tracimy jednak *przezroczystość* — pozycje menu i listy menu nie są traktowane jednolicie.

Możemy też zmienić metodę jestWegetariańska() menu tak, aby zwracała wartość false. Jest to rozwiązanie proste i pozwalające zachować przezroczystość.

W naszym rozwiązaniu wybieramy przejrzystość: chcemy przekazywać informację o tym, że Menu nie zapewnia obsługi operacji (co istotnie różni się od stwierdzenia, że wartością wywołania jestWegetariańska() jest false). Co więcej, takie podejście pozostawia otwarte pole programiście, który w przyszłości może zaimplementować metodę jestWegetariańska() dla menu. Nic nie będzie wówczas stało na przeszkodzie, aby zapewnić jej współpracę z pozostałym kodem.

Decyzja została podjęta i będziemy się jej trzymać.



Wzorce bez tajemnic

Wywiad tygodnia:
wzorzec Kompozyt; rozmawiamy o zagadnieniach implementacji

Rusz głową!: Spotykamy się dzisiaj z wzorcem Kompozyt. Może na początek opowiesz nam coś o sobie?

Wzorzec Kompozyt: Oczywiście… Jestem wzorcem przeznaczonym do stosowania w sytuacjach, gdy operujemy kolekcjami obiektów połączonych relacjami część-całość i gdy wymagane jest, aby wszystkie te obiekty były traktowane jednakowo.

Rusz głową!: OK, spróbujmy przeanalizować to stwierdzenie… co rozumiesz przez relację część-całość?

Wzorzec Kompozyt: Wyobraźmy sobie graficzny interfejs użytkownika. Znajdziemy w nim zazwyczaj element najwyższego poziomu, taki jak Frame czy Panel, który zawiera inne składniki, takie jak menu, pola tekstowe, paski przewijania i przyciski. A więc interfejs użytkownika składa się z wielu części, ale gdy je wyświetlamy, traktujemy go jako całość. Nakazujemy wyświetlenie elementu najwyższego poziomu i oczekujemy, że ten wyświetli wszystkie swoje części. Wywołujemy składniki, które zawierają inne składniki, obiekty złożone i składniki, które nie mają podskładników, czyli obiekty-liście.

Rusz głową!: Czy to jest to, co rozumiesz przez jednakowe traktowanie obiektów? To, że dysponujemy wspólnymi metodami, które można wywoływać zarówno dla kompozytów, jak i liści?

Wzorzec Kompozyt: Właśnie. Mogę nakazać wyświetlenie się obiektowi kompozytowemu lub obiektowi-liściowi i każdy z nich zrobi to, co powinien. Obiekt złożony wyświetli się, nakazując wykonanie takiej operacji wszystkim swoim składnikom.

Rusz głową!: To implikuje, że każdy obiekt ma ten sam interfejs. A co z przypadkami, kiedy obiekty kompozytu mają różne funkcje?

Wzorzec Kompozyt: Aby kompozyt był dla klienta przezroczysty, wszystkie składające się nań obiekty muszą implementować ten sam interfejs. W przeciwnym wypadku klient musiałby znać interfejsy każdego z wykorzystywanych obiektów i próba wprowadzenia wzorca straciłaby sens. Oczywiście, konsekwencją takiego rozwiązania jest to, że pewne metody nie mają odniesienia do części obiektów.

Rusz głową!: Co wtedy?

Wzorzec Kompozyt: Jest kilka możliwości. Czasem można taką sytuację zignorować, można też zwrócić wartość null lub false — cokolwiek będzie wygodniejsze w budowanej aplikacji. W innych przypadkach trzeba wygenerować wyjątek. Wówczas oczywiście klient musi być gotów wykonać nieco dodatkowej pracy i zadbać o sprawdzenie, czy metoda zadziałała poprawnie.

Rusz głową!: Jednak jeżeli klient nie zna typu obiektu, którym operuje, skąd ma znać dostępne wywołania?

Wzorzec Kompozyt: Odrobina kreatywności umożliwi takie ustrukturalizowanie metod, aby implementacja domyślna wykonywała pewną adekwatną operację. Przykładem może być wywołanie metody pobierzPotomek(). W przypadku kompozytu jej wywołanie jest zdecydowanie adekwatne. Można też stwierdzić, że nie jest ono pozbawione sensu w przypadku liścia, o ile tylko potraktujemy węzeł-liść jako obiekt bez potomków.

Rusz głową!: Sprytne… Słyszałem jednak, że pewne klienty są tak wyczulone na to zagadnienie, że wymagają osobnych interfejsów dla każdego z obiektów, co ma uniemożliwić im generowanie bezsensownych wywołań metod. Czy to jest jeszcze wzorzec Composite?

Wzorzec Kompozyt: Tak. To znacznie bezpieczniejsza wersja wzorca Composite, ale wymaga ona, aby klient sprawdzał typ każdego obiektu przed wywołaniem metody. Ma to zapewnić właściwą konwersję typu.

Rusz głową!: Powiedz nam coś jeszcze o strukturze obiektów złożonych i obiektów-liści.

Wzorzec Kompozyt: Zazwyczaj jest to struktura drzewa, pewien rodzaj hierarchii. Korzeniem jest obiekt kompozytowy najwyższego poziomu. Jego potomki są kolejnymi obiektami złożonymi lub liśćmi.

Rusz głową!: Czy potomki odwołują się do rodziców?

Wzorzec Kompozyt: Tak, składnik może mieć wskaźnik do rodzica. Ułatwia to przechodzenie między obiektami struktury. Staje się to bardzo istotne, gdy mamy odwołanie do potomka i pojawia się potrzeba jego usunięcia. Gdy procedura usuwania jest metodą rodzica, odpowiednie odwołanie w danych potomka jest bardzo pomocne.

Rusz głową!: Implementowanie wzorca wymaga więc rozważenia kilku istotnych zagadnień. Czy jest jeszcze coś, o czym powinniśmy pamiętać?

Wzorzec Kompozyt: Faktycznie, jest jeszcze coś… Kolejność potomków. Co dzieje się w przypadku, gdy obiekt złożony musi przechowywać potomki w określonej kolejności? Ich dodawanie i usuwanie wymaga wówczas bardziej wyszukanych procedur. Nie jest też bez znaczenia przy przeglądaniu struktury.

Rusz głową!: O tym dotąd nie pomyślałem, a to faktycznie ważne.

Wzorzec Kompozyt: A zastanawiałeś się nad buforowaniem?

Rusz głową!: Buforowaniem?

Wzorzec Kompozyt: Tak, buforowaniem. W pewnych przypadkach, gdy struktura jest złożona lub jej przeglądanie jest kosztowne, warto wprowadzić buforowanie węzłów. Przykładowo, jeżeli regularnie przeglądasz obiekt główny i jego potomki w celu obliczenia pewnych wartości, wartości tego mogą zostać zapisane, aby uniknąć wielokrotnego przeglądania drzewa.

Rusz głową!: Okazuje się, że wzorzec Composite nie jest taki banalny, jak mogłoby się z początku wydawać. Zanim zakończymy tę rozmowę, jedno, ostanie pytanie: co uważasz za swoją największą zaletę, za swoją mocną stronę?

Wzorzec Kompozyt: Będzie to przede wszystkim uproszczenie procedur klienckich. Mój klient nie musi interesować się tym, czy pracuje z obiektem złożonym, czy prostym. Nie trzeba więc pisać wszędzie instrukcji sprawdzających, z jakim obiektem mamy do czynienia, aby wywoływać właściwe metody. W wielu przypadkach wystarczy jedno wywołanie metody, aby wykonać niezbędną operację na całej strukturze.

Rusz głową!: Faktycznie, brzmi to bardzo dobrze. Niewątpliwie jesteś bardzo pożytecznym wzorcem do przechowywania obiektów i zarządzania nimi. Ale nasz czas się kończy. Dziękuję i do zobaczenia w kolejnym wywiadzie z cyklu „Wzorce bez tajemnic”.



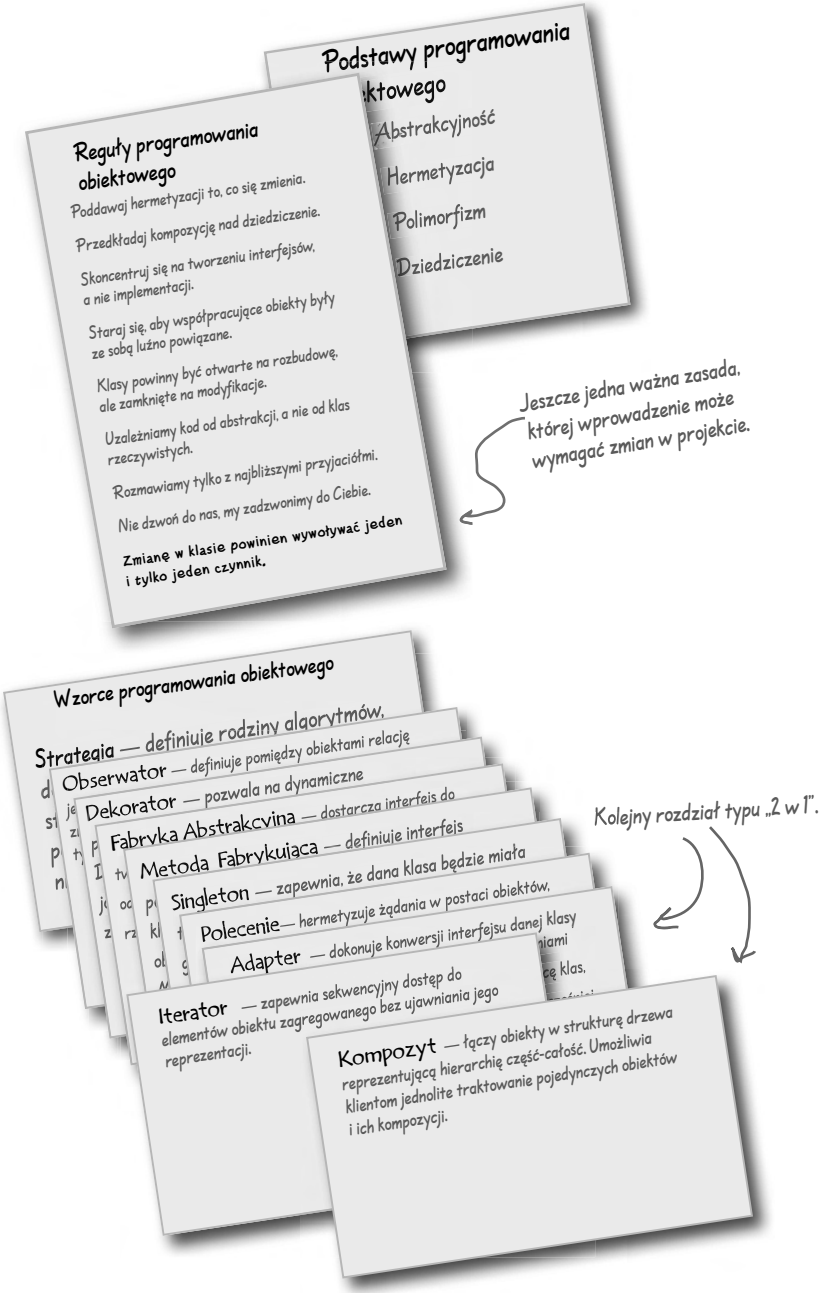
Dopasuj wzorce do opisów:

Wzorzec	Opis
Stan	Klienty dysponują ujednoliconym dostępem do kolekcji obiektów i pojedynczych obiektów.
Adapter	Zapewnia możliwość przeglądania kolekcji obiektów bez wiedzy o jej implementacji.
Iterator	Upraszcza interfejs grupy klas.
Fasada	Zmienia interfejs jednej lub większej liczby klas.
Kompozyt	Umożliwia powiadamianie grupy obiektów o zmianie stanu.
Obserwator	Umożliwia obiektowi zmianę zachowania w momencie zmiany stanu.



Twoja skrzynka narzędziowa

Dwa nowe wzorce — dwa wygodne sposoby operowania kolekcjami obiektów.



CELNE SPOSTRZEŻENIA

- Iterator umożliwia dostęp do elementów agregatu bez ujawniania jego struktury wewnętrznej.
- Zadaniem iteratora jest wyliczanie elementów agregatu i hermetyzowanie tych operacji w osobnym obiekcie.
- Gdy korzystamy z iteratora, zwalniamy agregat z odpowiedzialności za obsługę operacji przeglądania danych.
- Iterator zapewnia wspólny interfejs przeglądania elementów agregatu, umożliwiając korzystanie z polimorfizmu przy pisaniu kodu, który korzysta z tych elementów.
- Powinniśmy dążyć do przypisywania każdej klasie pojedynczego zakresu odpowiedzialności.
- Wzorzec Kompozyt to struktura do przechowywania obiektów i ich kompozycji.
- Wzorzec Kompozyt umożliwia klientom jednolite traktowanie kompozytów i pojedynczych obiektów.
- Składnik to obiekt w strukturze kompozytu. Składnikami są inne kompozyty lub węzły-liście.
- Przy implementowaniu wzorca Kompozyt musimy rozważyć wiele czynników. Najważniejszym jest wybór pomiędzy przezroczystością a bezpieczeństwem.



Rozwiązania ćwiczeń



Zaostrz ołówek

Które stwierdzenia są prawdziwe dla naszej implementacji metody drukujMenu()?

- ☑ A. Kodujemy pod kątem konkretnych implementacji obiektów PancakeHouseMenu i DinerMenu, a nie interfejsu.
- ☐ B. Klasa kelnerki nie implementuje Java Waitress API, a więc nie jest klasą standardową.
- ☑ C. Gdybyśmy zdecydowali się zastąpić klasę DinerMenu innym typem menu, w którym lista pozycji ma postać kolekcji Hashtable, wymagałoby to zmiany dużej ilości kodu klasy kelnerki.
- ☑ D. Klasa kelnerki musi znać reprezentację wewnętrznej kolekcji każdego menu; burzy to zasadę hermetyzacji.
- ☑ E. Musimy powtarzać fragmenty kodu: metoda drukujMenu() wymaga dwóch niezależnych pętli, aby przetworzyć zawartość dwóch różnych menu. Dodanie trzeciego menu prowadzi do konieczności użycia kolejnej takiej pętli.
- ☐ F. Implementacja jest oparta na MXML (Menu XML), więc jej współpraca z innymi elementami aplikacji jest ograniczona.



Zaostrz ołówek

Zanim przejdziesz do czytania kolejnej strony, wypisz trzy rzeczy, które musimy zrobić, aby włączyć powyższy kod do naszego schematu:

1. *Zaimplementować interfejs Menu*
2. *Usunąć metodę pobierzPozycjeMenu()*
3. *Dodać metodę utwórzIterator() i zapewnić zwracanie iteratora, który przegląda pozycje kolekcji Hashtable.*



Magnesiki z kodem. Rozwiązanie

Oto uporządkowany kod nowego, „zmieniającego się" iteratora klasy DinerMenu.

```
import java.util.Iterator;
import java.util.Calendar;

public class WymienneDinerMenuIterator implements Iterator {

    PozycjaMenu[] elementy;
    int pozycja;

    public WymienneDinerMenuIterator(PozycjaMenu[] elementy) {
        this.elementy = elementy;
        Calendar teraz = Calendar.getInstance();
        pozycja = teraz.DAY_OF_WEEK % 2;
    }

    public boolean hasNext() {
        if (pozycja >= elementy.length || elementy[pozycja] == null)
        {
            return false;
        } else {
            return true;
        }
    }

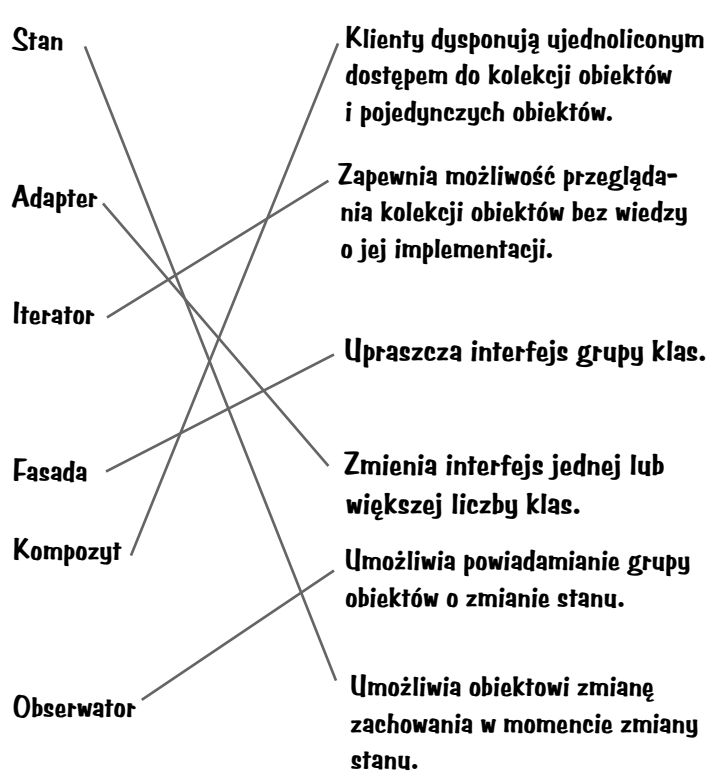
    public Object next() {
        PozycjaMenu pozycjaMenu = elementy[pozycja];
        pozycja = pozycja + 2;
        return pozycjaMenu;
    }

    public void remove() {
        throw new UnsupportedOperationException(
            "WymienneDinerMenuIterator nie zapewnia obsługi operacji usuwania elementów");
    }
}
```

Ta implementacja interfejsu Iterator nie zapewnia obsługi operacji remove().

Kto co robi

Dopasuj wzorce do opisów:



Stan obiektu



Mało znany fakt: wzorce Strategia i Stan to bliźniaki, rozdzielone zaraz po narodzinach. Jak już wiemy, wzorzec Strategia umożliwił przeprowadzenie wielu niezwykle udanych przedsięwzięć opartych na zamiennie stosowanych algorytmach. Wzorzec Stan ma inną rolę. Jest nią wspomaganie obiektów w kontrolowaniu ich własnych zachowań poprzez wewnętrzną zmianę stanu. Łatwo usłyszeć, jak mówi swoim podopiecznym: „Powtarzaj za mną: jestem wystarczająco zdolny, jestem wystarczająco dobry, dam radę to zrobić...".

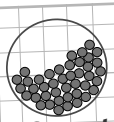
Guma do żucia i Java



Tostery z Javą to lata dziewięćdziesiąte. Dzisiaj Java opanowuje *prawdziwe* urządzenia, między innymi automaty z gumą do żucia. Okazuje się, że sprzedawanie gumy może być zaawansowaną technologią. Producenci automatów sprzedających odkryli, że umieszczenie procesora w ich urządzeniu pozwala zwiększyć sprzedaż, zdalnie kontrolować stan automatu do sprzedawania gumy i dokładniej mierzyć satysfakcję klientów.

Przynajmniej tak mówią. Być może po prostu znudzili się dziewiętnastowieczną technologią i poszukiwali urozmaicenia w pracy.

Ponieważ owi producenci lepiej znają się na automatach sprzedających niż na pisaniu programów, zwracają się do Ciebie o pomoc:

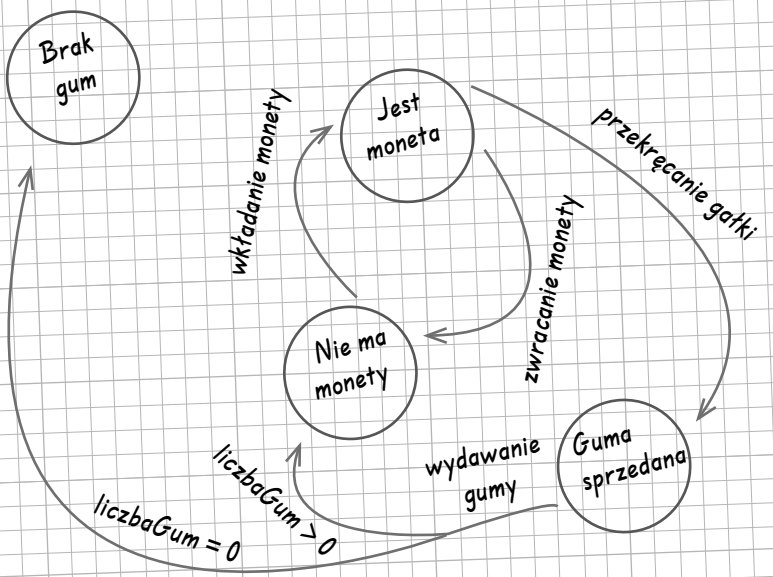


Automaty Sprzedające SA

gdzie automatyczny sprzedawca nigdy nie jest w połowie pusty...

Poniżej przedstawiamy naszą koncepcję pracy kontrolera automatu sprzedającego gumę do żucia. Mamy nadzieję, że zaimplementowanie tego schematu w Javie nie sprawi Wam trudności. W przyszłości będziemy zapewne dodawać nowe zachowania, więc prosimy o projekt możliwie elastyczny i łatwy do modyfikowania.

— Inżynierowie firmy Automaty Sprzedające



Krótka narada



Anna: Ten rysunek wygląda jak diagram stanów.

Jacek: No właśnie, każde z tych kółek to stan…

Anna: … a strzałki to przejścia między stanami.

Franek: Moment, już minęło trochę czasu, odkąd uczyłem się o diagramach stanu. Możecie przypomnieć mi, o co chodzi?

Anna: Jasne. Spójrz na kółka na rysunku. To są stany. NIE_MA_MONETY to zapewne stan początkowy automatu, kiedy jest tylko włączony i czeka na klienta. Poszczególne stany to różne konfiguracje urządzenia. Każda z nich ma własny zespół zachowań. Część z nich powoduje przejścia do innych stanów.

Jacek: Widać to na rysunku. Aby przejść do innego stanu, trzeba włożyć monetę, zwrócić monetę albo przekręcić gałkę. Widzisz strzałkę od NIE_MA_MONETY do JEST_MONETA?

Franek: Tak…

Jacek: Ta strzałka oznacza, że gdy automat jest w stanie NIE_MA_MONETY, włożenie monety spowoduje przejście do stanu JEST_MONETA. Wówczas mówimy o zmianie stanu automatu.

Franek: Rozumiem. Gdy automat jest w stanie JEST_MONETA, przekręcanie gałki zmieni jego stan na GUMA_SPRZEDANA. Można też zażądać zwrotu monety i powrócić do stanu NIE_MA_MONETY.

Anna: Właśnie!

Franek: Więc nie jest tak źle. Mamy tu cztery stany i chyba cztery operacje: „wkładanie monety”, „zwracanie monety”, „przekręcanie gałki” i „wydawanie gumy”. Ale… po wydaniu gumy, kiedy automat jest w stanie GUMA_SPRZEDANA, sprawdzamy liczbę pozostałych gum, po czym przechodzimy do stanu NIE_MA_MONETY lub BRAK_GUM. Jest więc w sumie pięć zmian z jednego stanu do drugiego.

Anna: Operacja sprawdzania liczby gum implikuje śledzenie tej wartości. Każda wydana guma może być ostatnią. Gdy zostanie wydana, automat przechodzi w stan BRAK_GUM.

Jacek: Nie można też zapominać o tym, że klient może próbować czynności, które nie mają sensu, jak zwracanie monety, gdy automat jest w stanie NIE_MA_MONETY lub włożenie dwóch monet.

Franek: Och, nie pomyślałem o tym. Trzeba to wziąć pod uwagę.

Jacek: Każda operacja musi rozpoczynać się od sprawdzenia stanu automatu. Dopiero wówczas możemy wykonać właściwą czynność. Damy radę! Rozpocznijmy już zapisywanie diagramu stanów w kodzie…

Maszyny stanowe 101

Jak przejdziemy od diagramu stanów do kodu aplikacji?
Oto krótki kurs implementowania maszyn stanowych:

1. Rozpoczynamy od wyliczenia stanów:states:



To są stany — w sumie cztery.

2. Deklarujemy zmienną konkretnego egzemplarza maszyny do zapisywania stanu i definiujemy cztery wartości:

BRAK_GUM oznacza pusty automat.

```
final static int BRAK_GUM = 0;
final static int NIE_MA_MONETY = 1;
final static int JEST_MONETA = 2;
final static int GUMA_SPRZEDANA = 3;

int stan = BRAK_GUM;
```

Poszczególne stany reprezentują różne liczby całkowite...

... a to jest zmienna egzemplarza, która przechowuje bieżący stan automatu. Ustawiamy jej wartość na BRAK_GUM, ponieważ fabrycznie nowa maszyna jest do czasu pierwszego napełnienia pusta.

3. Teraz wyliczamy operacje:

wkładanie monety

przekręcanie gałki

zwracanie monety

Te operacje to interfejs automatu sprzedającego, czyli to, co można z nim zrobić.

wydawanie gumy

Wydawanie to właściwie wewnętrzna operacja, inicjowana samoczynnie przez automat.

Zgodnie z diagramem wywołanie każdej z tych operacji powoduje zmianę stanu.

4 Tworzymy klasę, która będzie pracować na zasadzie automatu stanowego. Dla każdej operacji opracowujemy metodę, która korzysta z instrukcji wykonania warunkowego, aby określić odpowiednie dla bieżącego stanu zachowanie. Przykładem może być metoda odpowiadająca operacji wkładania monety:

```
public void włóżMonetę() {

    if (stan == JEST_MONETA) {

       System.out.println("Nie możesz włożyć więcej niż jednej monety");

    } else if (stan == BRAK_GUM) {

       System.out.println("Nie możesz włożyć monety, gdy automat jest pusty");

    } else if (stan == GUMA_SPRZEDANA) {

       System.out.println("Proszę czekać na gumę");

    } else if (stan == NIE_MA_MONETY) {

       stan = JEST_MONETA;
       System.out.println("Moneta przyjęta");
    }
}
```

Każdemu ze stanów odpowiada instrukcja wykonania warunkowego...

... zapewniająca, że zachowanie obiektu będzie odpowiadało jego stanowi...

... a w pewnych przypadkach, wyróżnionych w diagramie, prowadząca do zmiany stanu.

Opisujemy tu dość typową technikę modelowania stanu obiektu przez zmienną egzemplarza, w której zapisywana jest wartość stanu. Wykonywanie czynności odpowiednich dla bieżącego stanu zapewnia ciąg instrukcji warunkowych w każdej z udostępnianych metod.



Czas na implementację automatu do sprzedawania gumy!

Piszemy kod

Przedstawimy teraz implementację maszyny do sprzedawania gumy do żucia. Wiemy już, że informację o bieżącym stanie urządzenia będziemy przechowywać w zmiennej egzemplarza. Musimy teraz zapewnić obsługę wszystkich potencjalnych akcji, zachowań i zmian stanu. Będziemy implementować akcje wkładania monety, zwracania monety, przekręcania gałki i wydawania gumy. Musimy również wziąć pod uwagę sytuację opróżnienia maszyny.

```
public class AutomatSprzedający {

    final static int BRAK_GUM = 0;
    final static int NIE_MA_MONETY = 1;
    final static int JEST_MONETA = 2;
    final static int GUMA_SPRZEDANA = 3;

    int stan = BRAK_GUM;
    int liczba = 0;

    public AutomatSprzedający(int liczba) {
        this.liczba = liczba;
        if (liczba > 0) {
            stan = NIE_MA_MONETY;
        }
    }

    public void włóżMonetę() {
        if (stan == JEST_MONETA) {
           System.out.println("Nie możesz włożyć więcej niż jednej monety");
        } else if (stan == NIE_MA_MONETY) {
           stan = JEST_MONETA;
           System.out.println("Moneta przyjęta");
        } else if (stan == BRAK_GUM) {
           System.out.println("Nie możesz włożyć monety, gdy automat jest pusty");
        } else if (stan == GUMA_SPRZEDANA) {
           System.out.println("Proszę czekać na gumę");
        }
    }
```

To są cztery stany, takie same jak w diagramie od zleceniodawcy, czyli firmy Automaty Sprzedające SA.

To jest zmienna egzemplarza, która posłuży do śledzenia bieżącego stanu obiektu. Inicjowana wartością BRAK_GUM.

Druga zmienna egzemplarza przechowuje informację o liczbie gum w automacie.

Konstruktor pobiera informację o początkowej liczbie gum. Jeżeli jest ich więcej niż zero, automat wchodzi w stan NIE_MA_MONETY, co oznacza, że oczekuje na klienta. Gdy gum nie ma, automaty pozostaje w stanie BRAK_GUM.

Teraz rozpoczynamy implementowanie akcji jako metod…

Przy wkładaniu monety, jeżeli…

inna moneta została już włożona, informujemy o tym klienta;

w przeciwnym wypadku przyjmujemy monetę i przechodzimy do stanu JEST_MONETA.

Jeżeli klient właśnie kupił gumę, przed włożeniem kolejnej monety musi zaczekać na zakończenie transakcji.

a jeśli automat jest pusty, odmawiamy przyjęcia monety.

```
    public void zwróćMonetę() {
        if (stan == JEST_MONETA) {
            System.out.println("Moneta zwrócona");
            stan = NIE_MA_MONETY;
        } else if (stan == NIE_MA_MONETY) {
            System.out.println("Nie włożyłeś monety");
        } else if (stan == GUMA_SPRZEDANA) {
            System.out.println("Niestety, nie można zwrócić monety po przekręceniu gałki");
        } else if (stan == BRAK_GUM) {
            System.out.println("Nie włożyłeś monety");
        }
    }

    public void przekręćGałkę() {
        if (stan == GUMA_SPRZEDANA) {
            System.out.println("Nie dostaniesz gumy tylko dlatego, że przekręciłeś drugi raz!");
        } else if (stan == NIE_MA_MONETY) {
            System.out.println("Zanim przekręcisz gałkę, wrzuć monetę");
        } else if (stan == BRAK_GUM) {
            System.out.println("Przekręciłeś gałkę, ale automat jest pusty");
        } else if (stan == JEST_MONETA) {
            System.out.println("Obróciłeś gałkę...");
            stan = GUMA_SPRZEDANA;
            wydaj();
        }
    }

    public void wydaj() {
        if (stan == GUMA_SPRZEDANA) {
            System.out.println("Wypada guma");
            liczba = liczba - 1;
            if (liczba == 0) {
                System.out.println("Ups, koniec gum!");
                stan = BRAK_GUM;
            } else {
                stan = NIE_MA_MONETY;
            }
        } else if (stan == NIE_MA_MONETY) {
            System.out.println("Najpierw zapłać");
        } else if (stan == BRAK_GUM) {
            System.out.println("Nie wydano gumy");
        } else if (stan == JEST_MONETA) {
            System.out.println("Nie wydano gumy");
        }
    }

     // inne metody, jak toString() i napełnij()
}
```

Teraz, przy próbie odzyskania monety przez klienta...

Jeżeli w automacie jest moneta, zwracamy ją i powracamy do stanu NIE_MA_MONETY.

Jeżeli nie ma monety, nie można jej zwrócić.

Nie można zwrócić monety, gdy automat jest pusty, bo przecież w tym stanie nie mógł jej przyjąć!

Jeżeli klient właśnie przekręcił gałkę, nie można oddać monety. Klient już czeka na gumę!

Klient próbuje przekręcić gałkę...

Ktoś próbuje oszukiwać.

Najpierw moneta.

Nie wydamy żadnych gum. Automat jest pusty.

Sukces! Klient otrzyma gumę. Zmieniamy stan na GUMA_SPRZEDANA i wywołujemy metodę wydawania gumy: wydaj().

Metoda wywoływana w celu wydania gumy.

Jesteśmy w stanie GUMA_SPRZEDANA; dajemy gumę!

Tutaj zajmujemy się warunkiem „brak gum". Po wydaniu ostatniej gumy przestawiamy automat w stan BRAK_GUM. W innych przypadkach przechodzimy do stanu oczekiwania na monetę.

Żadna z tych sytuacji nie powinna mieć miejsca, ale gdyby tak się stało, wynikiem operacji musi być błąd, a nie wydanie gumy.

Test wewnętrzny

Wygląda na to, że — korzystając ze starannie dobranej metodyki — stworzyliśmy całkiem porządny projekt, prawda? Zanim przekażemy program firmie, a ta umieści go w swoich automatach, przeprowadźmy wewnętrzny test. Oto mechanizm testujący:

```
public class AutomatSprzedającyTest {

    public static void main(String[] args) {
        AutomatSprzedający automatSprzedający = new AutomatSprzedający(5);

        System.out.println(automatSprzedający);

        automatSprzedający.włóżMonetę();
        automatSprzedający.przekręćGałkę();

        System.out.println(automatSprzedający);

        automatSprzedający.włóżMonetę();
        automatSprzedający.zwróćMonetę();
        automatSprzedający.przekręćGałkę();

        System.out.println(automatSprzedający);

        automatSprzedający.włóżMonetę();
        automatSprzedający.przekręćGałkę();
        automatSprzedający.włóżMonetę();
        automatSprzedający.przekręćGałkę();
        automatSprzedający.zwróćMonetę();

        System.out.println(automatSprzedający);

        automatSprzedający.włóżMonetę();
        automatSprzedający.włóżMonetę();
        automatSprzedający.przekręćGałkę();
        automatSprzedający.włóżMonetę();
        automatSprzedający.przekręćGałkę();
        automatSprzedający.włóżMonetę();
        automatSprzedający.przekręćGałkę();

        System.out.println(automatSprzedający);
    }
}
```

Ładujemy automat pięcioma gumami.

Wypisujemy informację o stanie automatu.

Wrzucamy monetę...

Przekręcamy dźwignię: powinniśmy otrzymać gumę.

Jeszcze raz wypisujemy informację o stanie automatu.

Wrzucamy monetę...

Żądamy zwrotu.

Przekręcamy gałkę: nie powinniśmy dostać gumy.

Ponownie wypisujemy informację o stanie automatu.

Wrzucamy monetę...

Przekręcamy gałkę: powinniśmy otrzymać gumę.

Wrzucamy monetę...

Przekręcamy gałkę: powinniśmy otrzymać gumę.

Żądamy zwrotu monety, której nie włożyliśmy.

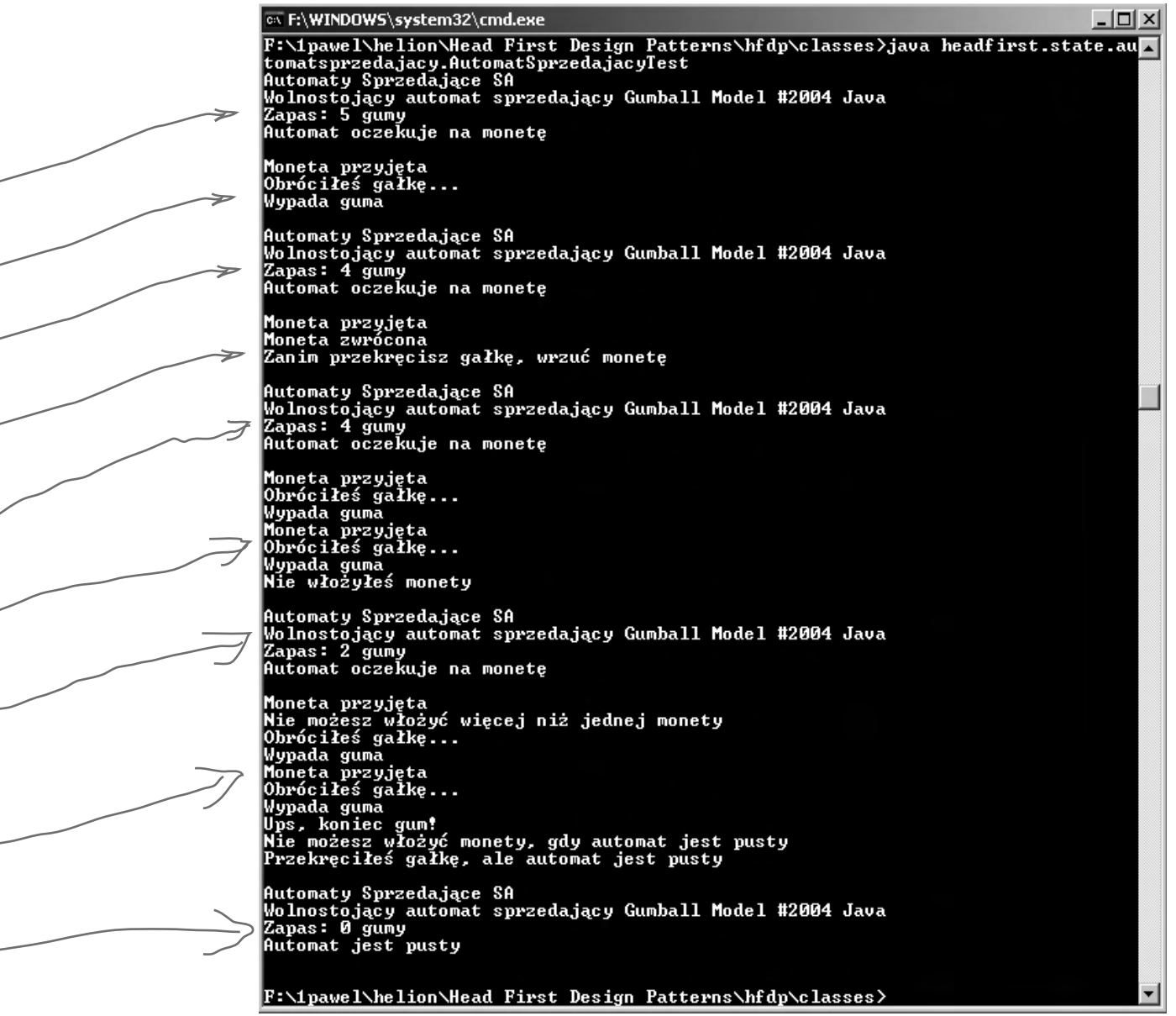
Ponownie wypisujemy informację o stanie automatu.

Wrzucamy DWIE monety...

Przekręcamy gałkę: powinniśmy otrzymać gumę.

Jeszcze trochę testów wytrzymałościowych... ☺

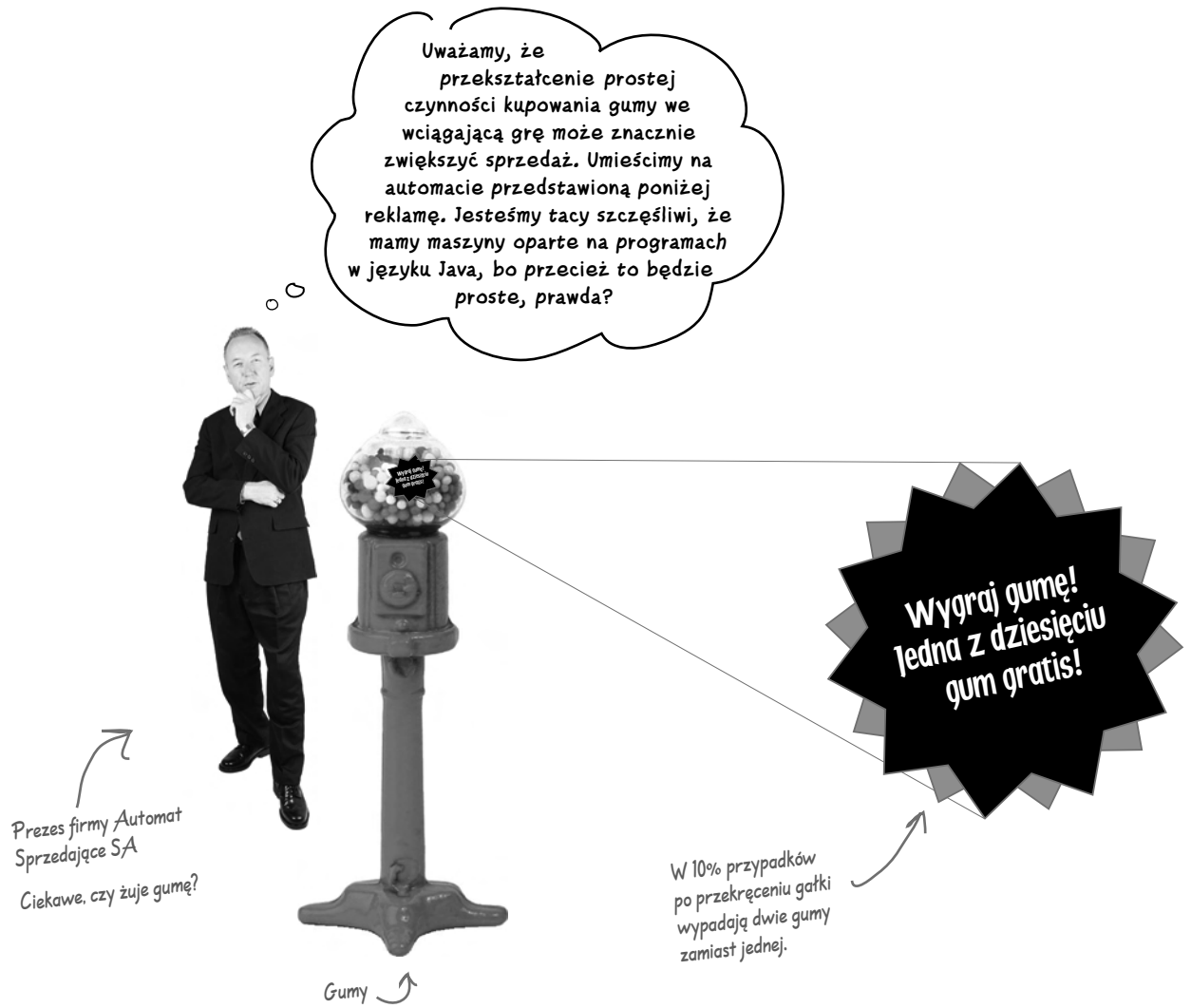
Jeszcze raz wypisujemy informację o stanie.


F:\WINDOWS\system32\cmd.exe
F:\1pawel\helion\Head First Design Patterns\hfdp\classes>java headfirst.state.automatsprzedajacy.AutomatSprzedajacyTest
Automaty Sprzedające SA
Wolnostojący automat sprzedający Gumball Model #2004 Java
Zapas: 5 gumy
Automat oczekuje na monetę
Moneta przyjęta
Obróciłeś gałkę...
Wypada guma
Automaty Sprzedające SA
Wolnostojący automat sprzedający Gumball Model #2004 Java
Zapas: 4 gumy
Automat oczekuje na monetę
Moneta przyjęta
Moneta zwrócona
Zanim przekręcisz gałkę, wrzuć monetę
Automaty Sprzedające SA
Wolnostojący automat sprzedający Gumball Model #2004 Java
Zapas: 4 gumy
Automat oczekuje na monetę
Moneta przyjęta
Obróciłeś gałkę...
Wypada guma
Moneta przyjęta
Obróciłeś gałkę...
Wypada guma
Nie włożyłeś monety
Automaty Sprzedające SA
Wolnostojący automat sprzedający Gumball Model #2004 Java
Zapas: 2 gumy
Automat oczekuje na monetę
Moneta przyjęta
Nie możesz włożyć więcej niż jednej monety
Obróciłeś gałkę...
Wypada guma
Moneta przyjęta
Obróciłeś gałkę...
Wypada guma
Ups, koniec gum!
Nie możesz włożyć monety, gdy automat jest pusty
Przekręciłeś gałkę, ale automat jest pusty
Automaty Sprzedające SA
Wolnostojący automat sprzedający Gumball Model #2004 Java
Zapas: 0 gumy
Automat jest pusty
F:\1pawel\helion\Head First Design Patterns\hfdp\classes>

Wiedziałeś, że to jest blisko... zmiana!

Firma Automaty Sprzedające SA załadowała kod do swoich najnowszych automatów i zajęli się nim specjaliści od jakości. Jak dotąd, z ich punktu widzenia wszystko wygląda świetnie.

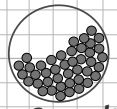
Właściwie, wszystko poszło tak dobrze, że zleceniodawca postanowił pójść o krok dalej...





Łamigłówka projektowa

Narysuj diagram stanów automatu sprzedającego gumy do żucia, który zapewnia obsługę promocji. W 10% przypadków przejście w stan GUMA_SPRZEDANA prowadzi do wydania dwóch gum zamiast jednej. Zanim zaczniesz czytać dalej, porównaj własne rozwiązanie z tym przedstawionym na końcu rozdziału, aby upewnić się, że zgadzamy się w naszym podejściu...



Automaty Sprzedające SA

gdzie automatyczny sprzedawca nigdy nie jest w połowie pusty...

Wykorzystaj kartkę w kratkę firmy Automaty Sprzedające SA do narysowania diagramu stanów

Kłopotliwy STAN rzeczy...

Sam fakt napisania kodu automatu sprzedającego w oparciu o dobrze przemyślaną metodę nie jest równoznaczny z tym, że będzie to kod, który łatwo rozszerzać. W rzeczywistości, gdy ponownie przyjrzymy się naszej klasie i rozważymy, które elementy wymagają zmian…

```
final static int BRAK_GUM = 0;
final static int NIE_MA_MONETY = 1;
final static int JEST_MONETA = 2;
final static int GUMA_SPRZEDANA = 3;


public void włóżMonetę() {
    // kod procedury wrzucania monety
}

public void zwróćMonetę() {
    // kod procedury zwracania monety
}

public void przekręćGałkę() {
    // kod procedury obracania gałki
}

public void wydaj() {
    // kod procedury wydawania gumy
}
```

Przede wszystkim musisz dodać nowy stan WYGRANA. To jeszcze nie jest trudne…

… ale potem w każdej metodzie trzeba dodać instrukcję warunkową uwzględniającą nowy stan; to dużo pracy.

metoda przekręćGałkę() będzie szczególnie kłopotliwa, ponieważ wymaga dodania kodu, który sprawdzi, czy klient wygrał gumę, i wybierze między stanami WYGRANA i GUMA_SPRZEDANA.



Zaostrz ołówek

Które stwierdzenia są prawdziwe dla bieżącego stanu naszej implementacji?

- ❑ A. Ten kod na pewno nie spełnia zasady otwarcia na rozbudowę i zamknięcia na modyfikacje.
- ❑ B. Programista języka FORTRAN byłby z takiego programu dumny.
- ❑ C. Ten projekt nie jest nawet obiektowy.
- ❑ D. Zmiany stanu nie są jawne; giną pośród ogromnej liczby instrukcji warunkowych.
- ❑ E. Nie wprowadziliśmy żadnej hermetyzacji elementów zróżnicowanych.
- ❑ F. Dalsza rozbudowa kodu spowoduje zapewne wiele drobnych błędów.



No tak, to nie jest dobre. Myślę, że pierwsza wersja była w porządku, ale nie sprawdzi się, jeśli zleceniodawca będzie żądał ciągle nowych funkcji. Nie możemy sobie pozwolić na pisanie programu tak podatnego na błędy.

Jacek: Masz rację! Musimy przekształcić ten kod tak, żeby było łatwo go zmieniać i rozwijać.

Anna: Zdecydowanie powinniśmy wprowadzić hermetyzację implementacji zachowań związanych z poszczególnymi stanami, żeby zmiana jednego stanu nie prowadziła do modyfikowania całego programu.

Jacek: Słusznie. Innymi słowy, stosujemy zasadę hermetyzacji tego, co odmienne.

Anna: Właśnie.

Jacek: Jeżeli umieścimy zachowania właściwe dla danego stanu w osobnej klasie, każda z takich klas będzie implementować zbiór akcji jednego stanu.

Anna: Zgadza się. A klasa główna automatu może posłużyć się delegacją do obiektu reprezentującego bieżący stan.

Jacek: A to dobre: kompozycja jest lepsza niż… Kolejna zasada w działaniu.

Anna: Dzięki. Nie jestem w stu procentach pewna, że to zadziała, ale myślę, że jesteśmy na dobrej drodze.

Jacek: Zastanawiam się, czy to ułatwi wprowadzanie nowych stanów?

Anna: Tak mi się wydaje… Wciąż musimy wprowadzać zmiany w kodzie, ale ich zakres będzie bardziej ograniczony, bo dodawanie stanu będzie oznaczało po prostu zaimplementowanie nowej klasy. Oczywiście, niezbędne będą też odpowiednie procedury zmiany stanu.

Jacek: Podoba mi się. Bierzmy się do pracy!

Nowy projekt

Wygląda na to, że mamy nowy plan: zamiast rozwijać wcześniejszy kod, będziemy budować go od nowa tak, aby zapewnić hermetyzację obiektów stanu we własnych klasach i korzystać z delegowania do obiektu stanu bieżącego, gdy wywoływana jest akcja automatu.

Postępujemy tutaj zgodnie z zasadami projektowymi, więc wynikiem naszej pracy powinien być projekt, którego dalsze rozwijanie i poprawianie będzie wygodniejsze. Zrobimy tak:

1. **Najpierw zdefiniujemy interfejs Stan, który będzie zawierał metody odpowiadające każdej operacji automatu sprzedającego.**

2. **Następnie zaimplementujemy klasy reprezentujące poszczególne stany automatu. Każda z nich będzie odpowiedzialna za działanie automatu w pewnym stanie.**

3. **Na końcu usuniemy wszystkie instrukcje wykonania warunkowego i wprowadzimy delegację do klas reprezentujących stany.**

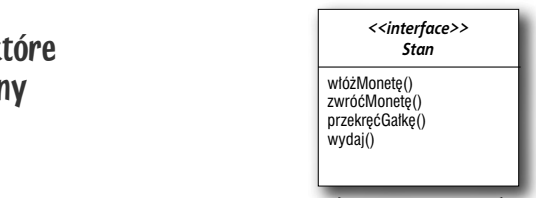
Nie tylko stosujemy zasady projektowe, ale również, jak łatwo zauważyć, wprowadzamy wzorzec Stan. Jego pełniejsze omówienie odłożymy do czasu zakończenia bieżących zmian w kodzie…



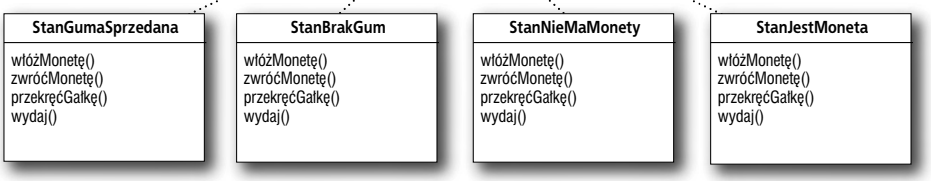
Definiowanie interfejsów i klas reprezentacji stanu

Najpierw utworzymy interfejs Stan, implementowany przez wszystkie klasy stanów:

To jest interfejs wszystkich stanów. Operacje automatu sprzedającego są mapowane bezpośrednio do metod (tych samych, których używano wcześniej).



Następnie utworzymy klasy, które hermetyzują poszczególne stany i implementują interfejs Stan.



Aby określić, jakich stanów potrzebujemy, sięgamy do wcześniejszego kodu…

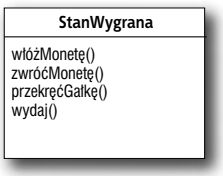
```
public class AutomatSprzedający {

    final static int BRAK_GUM = 0;
    final static int NIE_MA_MONETY = 1;
    final static int JEST_MONETA = 2;
    final static int GUMA_SPRZEDANA = 3;

    int stan = BRAK_GUM;
    int liczba = 0;
```

… i mapujemy stany do klas.

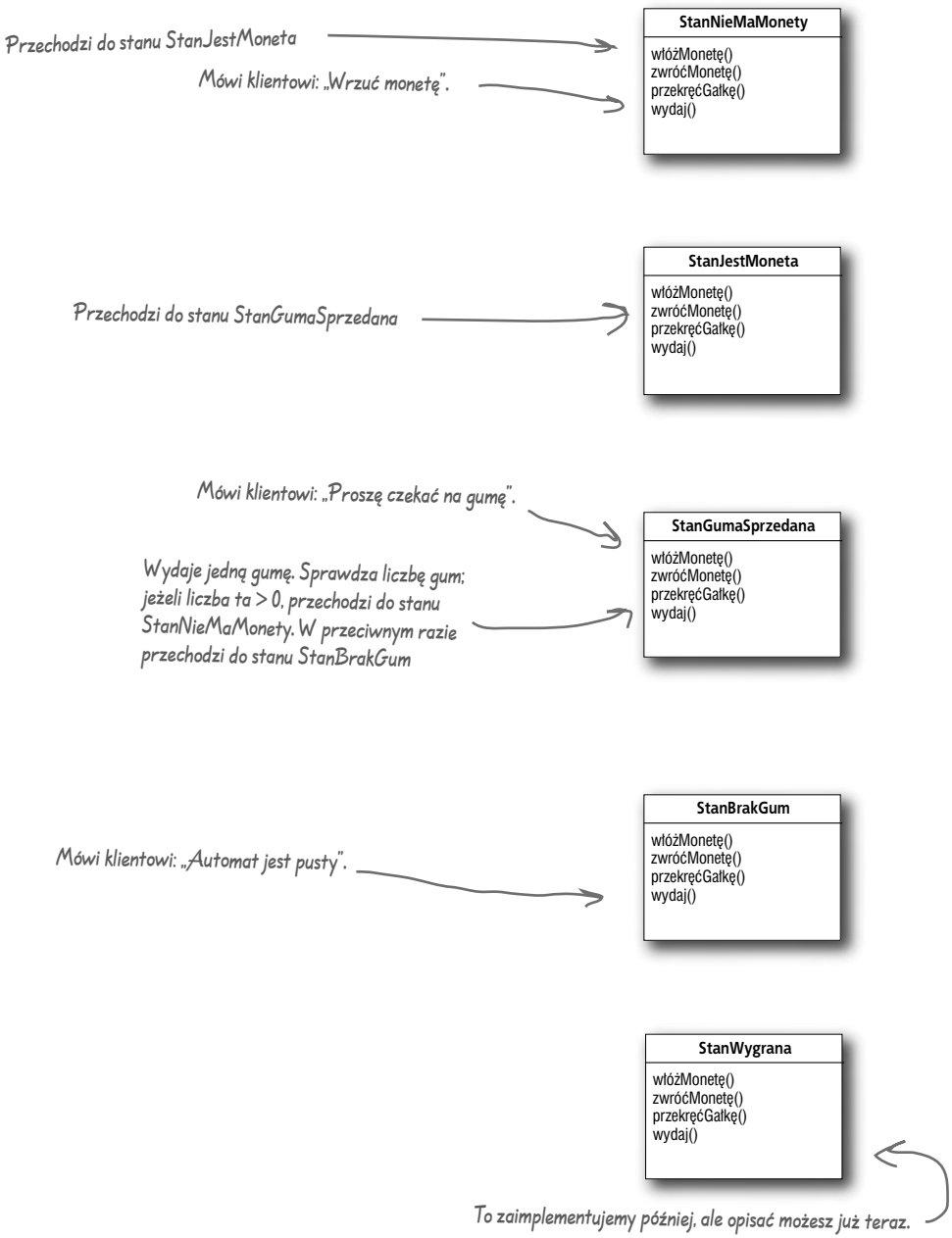
Nie zapominajmy, że potrzebujemy nowego stanu StanWygrana, o tym samym interfejsie. Powrócimy do niego po zakończeniu pracy z nową implementacją pierwotnej wersji oprogramowania automatu.





Zaostrz ołówek

Aby zaimplementować wymagane stany, musimy określić zachowania klas w chwili, gdy wywoływane są poszczególne akcje. Uzupełnij poniższy diagram opisami zachowań przypisanych każdej akcji w poszczególnych klasach. Kilka przykładowych opisów już wprowadziliśmy.



Implementowanie klas Stan

Czas na implementację klas reprezentujących stany. Znamy już wymagane zachowania, pozostaje tylko zapisać je w kodzie. Będziemy się starali wykorzystać fragmenty wcześniejszego kodu, ale tym razem będzie on zawarty w różnych klasach.

Rozpocznijmy od stanu StanNieMaMonety:

Implementujemy interfejs Stan.

Konstruktor pobiera odwołanie do klasy głównej automatu sprzedającego. Zapisujemy je w zmiennej egzemplarza.

```
public class StanNieMaMonety implements Stan {
    AutomatSprzedający automatSprzedający;

    public StanNieMaMonety(AutomatSprzedający automatSprzedający) {
        this.automatSprzedający = automatSprzedający;
    }

    public void włóżMonetę() {
        System.out.println("Moneta przyjęta");
        automatSprzedający.setStan(automatSprzedający.pobierzStanJestMoneta());
    }

    public void zwróćMonetę() {
        System.out.println("Nie włożyłeś monety");
    }

    public void przekręćGałkę() {
        System.out.println("Zanim przekręcisz gałkę, wrzuć monetę");
     }

    public void wydaj() {
        System.out.println("Najpierw zapłać");
    }
```

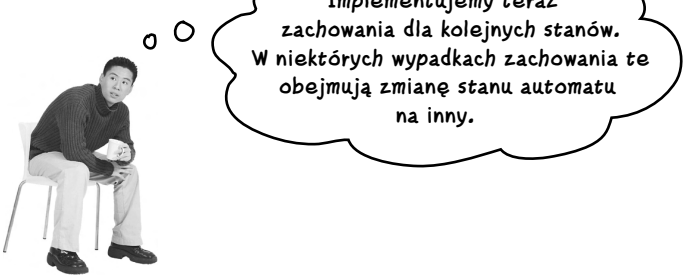
Po włożeniu monety wyświetlamy komunikat potwierdzający jej przyjęcie i zmieniamy stan automatu na StanJestMoneta.

Tym fragmentem zajmiemy się za chwilę...

Nie zwracamy nieotrzymanych pieniędzy!

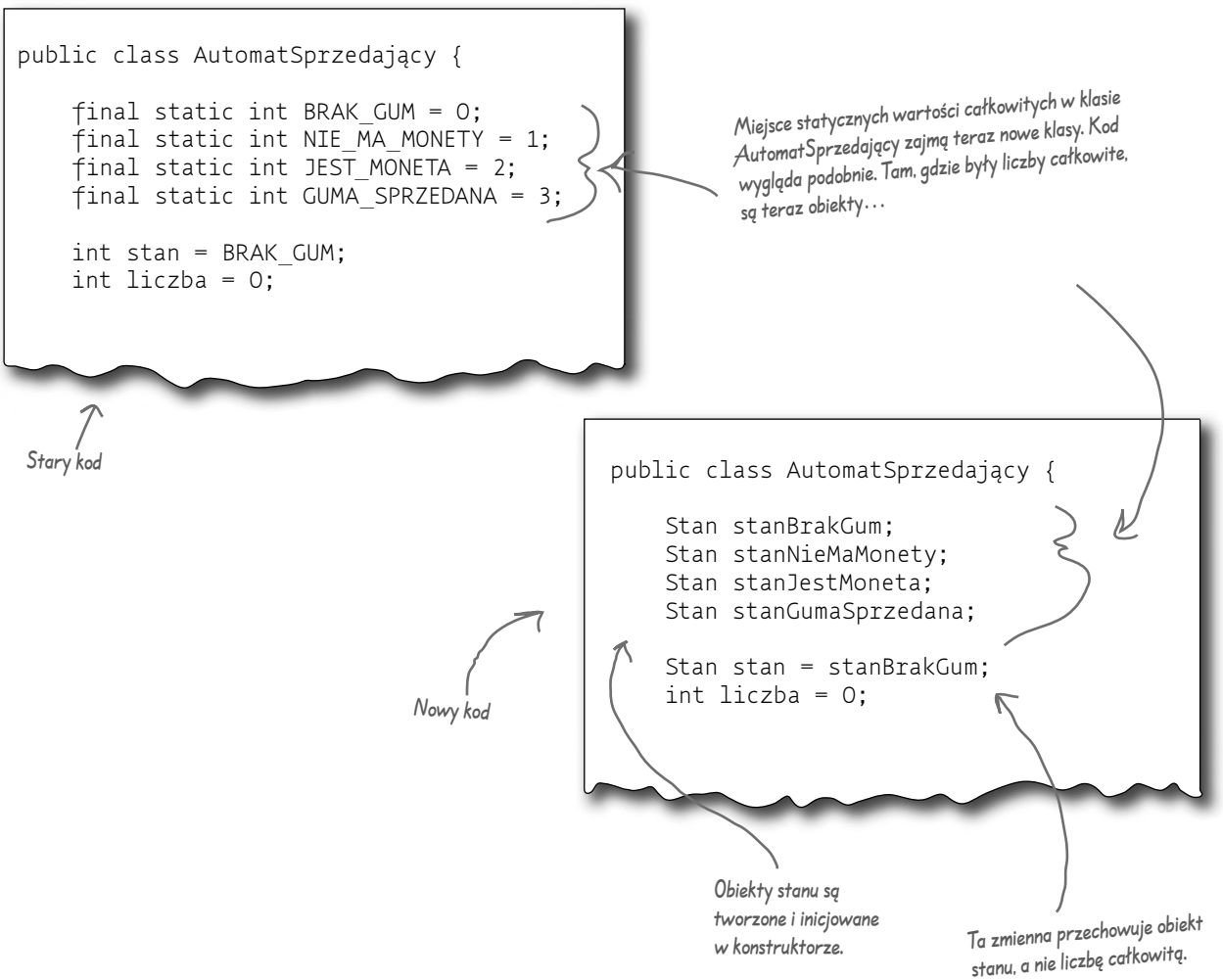
Nie dajemy gumy za darmo.

W tym stanie na pewno nie wydamy gumy.



Nowa wersja automatu sprzedającego

Zanim zakończymy implementowanie klas reprezentujących stan automatu, musimy zmodyfikować klasę AutomatSprzedający — umożliwi to sprawdzenie, czy różne elementy kodu do siebie pasują. Rozpocznijmy od zmiennych egzemplarza wykorzystywanych do zapisywania informacji o stanie. Teraz znaczniki w postaci liczb całkowitych zastąpią obiekty stanu.



Spójrzmy na nową klasę AutomatSprzedający...

```
public class AutomatSprzedający {

    Stan stanBrakGum;
    Stan stanNieMaMonety;
    Stan stanJestMoneta;
    Stan stanGumaSprzedana;

    Stan stan = stanBrakGum;
    int liczba = 0;

    public AutomatSprzedający(int liczbaGum) {
        stanBrakGum = new StanBrakGum(this);
        stanNieMaMonety = new StanNieMaMonety(this);
        stanJestMoneta = new StanJestMoneta(this);
        stanGumaSprzedana = new StanGumaSprzedana(this);

        this.liczba = liczbaGum;
        if (liczbaGum > 0) {
            stan = stanNieMaMonety;
        }
    }

    public void włóżMonetę() {
        stan.włóżMonetę();
    }

    public void zwróćMonetę() {
        stan.zwróćMonetę();
    }

    public void przekręćGałkę() {
        stan.przekręćGałkę();
        stan.wydaj();
    }

    void ustawStan(Stan stan) {
        this.stan = stan;
    }

    void zwolnijGumę() {
        System.out.println("Wypada guma...");
        if (liczba != 0) {
            liczba = liczba - 1;
        }
    }

    // Dalsze metody, w tym gettery stanów...
}
```

Ponownie deklarujemy stany...

... i zmienną egzemplarza z informacją o stanie bieżącym.

Zmienna egzemplarza count przechowuje informację o liczbie gum — początkowo automat jest pusty.

Konstruktor pobiera początkową liczbę gum i zapisuje ją w zmiennej egzemplarza.

Tworzy również egzemplarze klas stanu.

Jeżeli jest więcej niż jedna guma, ustawiamy automat w stan StanNieMaMonety.

Teraz akcje. Te będzie teraz BARDZO ŁATWO zapisać. Ograniczamy się do delegacji do obiektu stanu.

Uwaga: Zwróćmy uwagę, że nie potrzebujemy w tej klasie metody wydaj(). Jest to operacja inicjowana wewnętrznie i użytkownik nie ma możliwości wywołania jej bezpośrednio. Wywołujemy jednak taką metodę obiektu stanowego w trakcie operacji przekręcania gałki.

Ta metoda pozwala innym obiektom (takim jak nasze obiekty stanu) na zmianę stanu automatu.

Automat zapewnia obsługę metody pomocniczej zwolnijGumę(), która zwalnia gumę i zmniejsza wartość zmiennej egzemplarza count o 1.

Tu zawarte są metody takie jak pobierzStanNieMaMonety() (służąca do pobierania obiektów stanu) i pobierzLiczba() (służąca do pobierania wartości reprezentującej liczbę gum).

Implementowanie kolejnych stanów

Mamy już teraz ogólną orientację dotyczącą tego, jak będzie wyglądać nowy program automatu sprzedającego. Zaimplementujemy teraz dwie kolejne klasy stanu…

```
public class StanJestMoneta implements Stan {
    AutomatSprzedający automatSprzedający;

    public StanJestMoneta(AutomatSprzedający automatSprzedający) {
        this.automatSprzedający = automatSprzedający;
    }

    public void włóżMonetę() {
        System.out.println("Nie możesz włożyć więcej niż jednej monety");
    }

    public void zwróćMonetę() {
        System.out.println("Moneta zwrócona");
        automatSprzedający.ustawStan(automatSprzedający.pobierzStanNieMaMonety());
    }

    public void przekręćGałkę() {
        System.out.println("Obróciłeś gałkę...");
        automatSprzedający.ustawStan(automatSprzedający.pobierzStanGumaSprzedana());
    }

    public void wydaj() {
        System.out.println("Nie wydano gumy");
    }
}
```

Gdy tworzymy egzemplarz klasy stanu, przekazujemy mu odwołanie do obiektu AutomatSprzedający. Jest ono wykorzystywane przy zmianach stanu.

Akcja niedopuszczalna w tym stanie.

Zwracamy klientowi monetę i powracamy do stanu StanNieMaMonety.

Obrócenie gałki powoduje przejście automatu do stanu StanGumaSprzedana. Realizuje to wywołanie metody setStan(), która wymaga przekazania obiektu reprezentującego nowy stan. Obiekt taki pobieramy metodą getStanGumaSprzedana(), która jest prostym getterem klasy AutomatSprzedający (każdy stan ma osobną metodę typu getter).

Kolejna akcja niedopuszczalna w tym stanie automatu.

I klasa reprezentująca stan StanGumaSprzedana…

```
public class StanGumaSprzedana implements Stan {

    AutomatSprzedający automatSprzedający;

    public StanGumaSprzedana(AutomatSprzedający automatSprzedający) {
        this.automatSprzedający = automatSprzedający;
    }

    public void włóżMonetę() {
        System.out.println("Proszę czekać na gumę");
    }

    public void zwróćMonetę() {
        System.out.println("Niestety, nie można zwrócić monety po przekręceniu gałki");
    }

    public void przekręćGałkę() {
        System.out.println("Nie dostaniesz gumy tylko dlatego, że przekręciłeś drugi raz!");
    }


    public void wydaj() {
        automatSprzedający.zwolnijGumę();
        if (automatSprzedający.pobierzLiczba() > 0) {
            automatSprzedający.ustawStan(automatSprzedający.pobierzStanNieMaMonety());
        } else {
            System.out.println("Ups, koniec gum!");
            automatSprzedający.ustawStan(automatSprzedający.pobierzStanBrakGum());
        }
    }
}
```

Akcje niedopuszczalne w tym stanie.

Jesteśmy w stanie StanGumaSprzedana, co oznacza, że klient zapłacił. Pierwszą czynnością będzie więc zwolnienie jednej gumy.

Następnie pytamy automatu, ile gum zostało, i przechodzimy do stanu StanNieMaMonety lub StanBrakGum.

Tutaj zaczyna się właściwa praca…



WYTĘŻ UMYSŁ

Przyjrzyj się implementacji klasy AutomatSprzedający. Jeżeli gałka została przekręcona, ale właściwa operacja nie została wykonana (na przykład wtedy, gdy klient nie wrzucił monety), wciąż wywołujemy metodę wydaj(). Jak można tego uniknąć?



Zaostrz ołówek

Pozostaje jedna niezaimplementowana klasa: StanBrakGum. Może sam ją napiszesz? Wymaga to uważnego rozważenia czynności automatu w każdej sytuacji. Przed dalszą lekturą sprawdź na końcu rozdziału, jak wygląda nasza implementacja...

```
public class StanBrakGum implements [                ] {
    AutomatSprzedający automatSprzedający;

    public StanBrakGum(AutomatSprzedający automatSprzedający) {



    }

    public void włóżMonetę() {



    }

    public void zwróćMonetę() {



    }

    public void przekręćGałkę() {



    }

    public void wydaj() {



    }

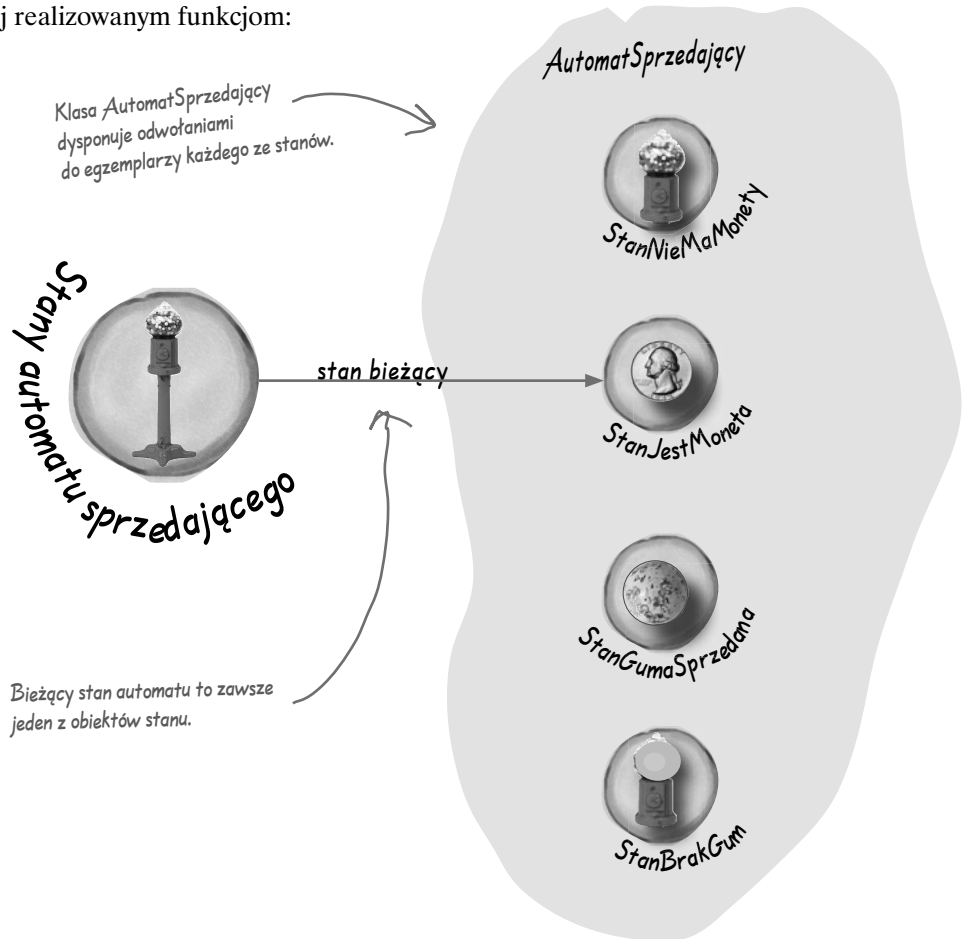
}
```

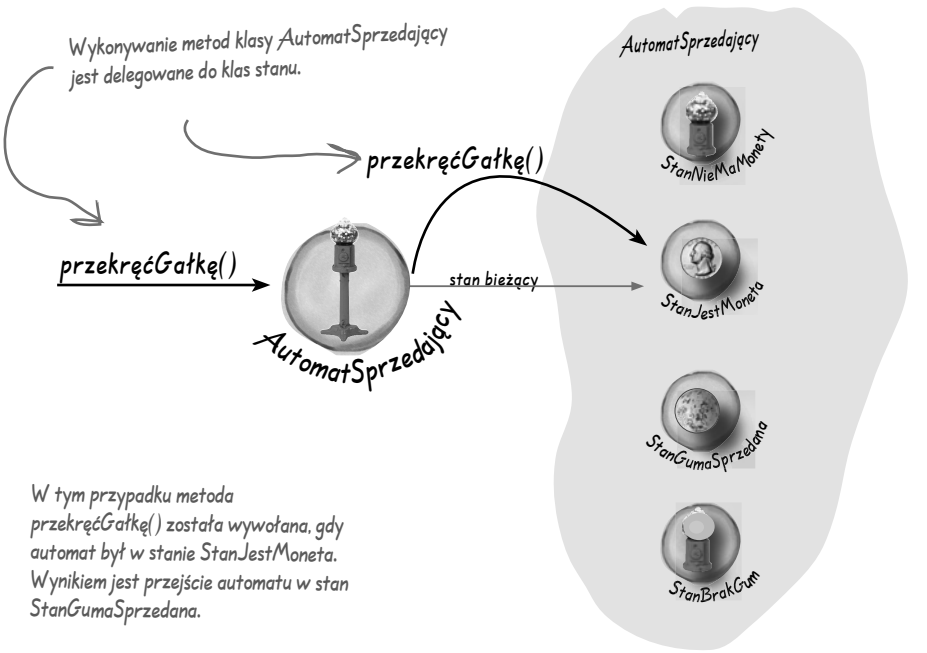
Krótkie podsumowanie — co już zrobiliśmy?

Przede wszystkim, mamy implementację automatu sprzedającego, która jest *strukturalnie* odmienna od poprzedniej, podczas gdy *funkcjonalnie pozostaje nie zmieniona*. Zmiana strukturalna doprowadziła do:

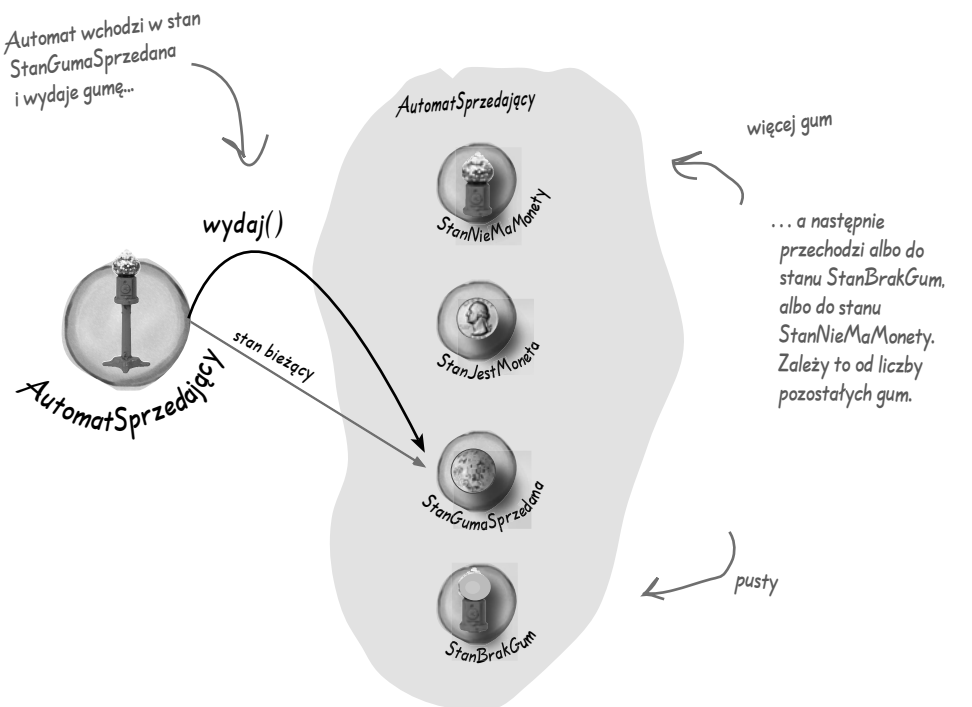
- Zlokalizowania zachowań każdego stanu w osobnych klasach.
- Usunięcia wszystkich kłopotliwych instrukcji `if`, które bardzo utrudniają rozbudowę i poprawianie programu.
- Zamknięcia każdego stanu dla modyfikacji. Automat wciąż można rozbudowywać poprzez dodawanie nowych klas stanu (i za chwilę to zrobimy).
- Utworzenia kodu i struktury klas, które są znacznie lepiej dopasowane do diagramu zleceniodawcy. Aplikacja jest bardzo przejrzysta.

Przyjrzyjmy się teraz bliżej realizowanym funkcjom:





PRZEJŚCIE DO STANU
StanGumaSprzedana





Zaostrz ołówek

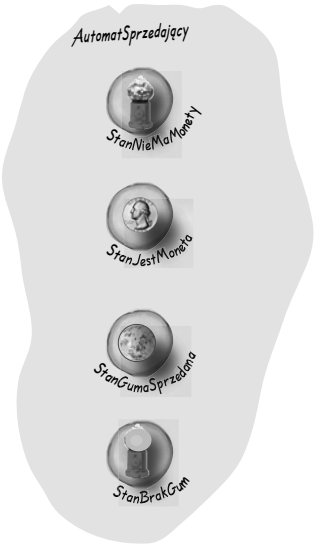
Prześledź działanie automatu sprzedającego, począwszy od stanu StanNieMaMonety. Uzupełnij diagram wywoływanymi akcjami i danymi wyjściowymi. W tym ćwiczeniu możesz założyć, że w automacie zawsze jest wystarczająca liczba gum.



Za kulisami
samodzielna wycieczka

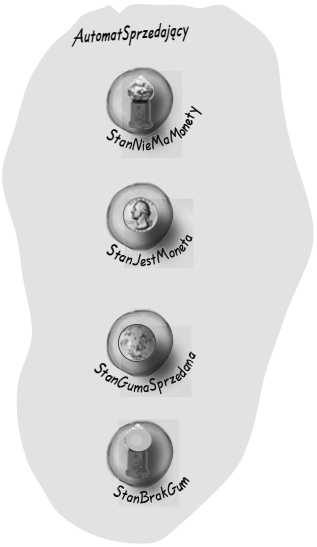
1





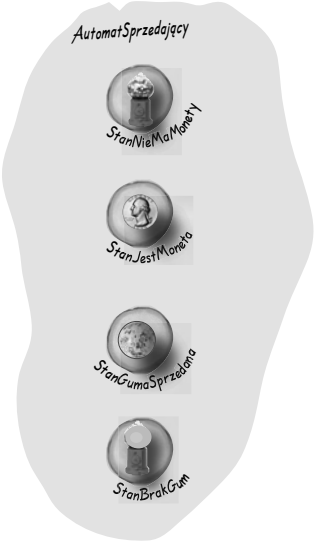
2





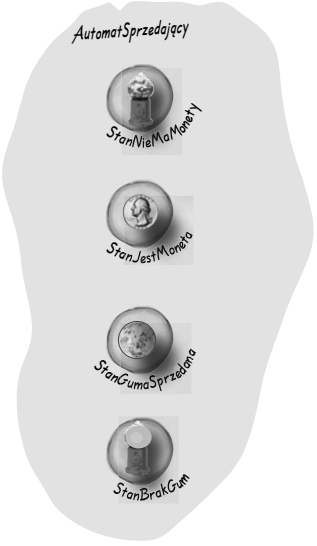
3





4





Definicja wzorca Stan

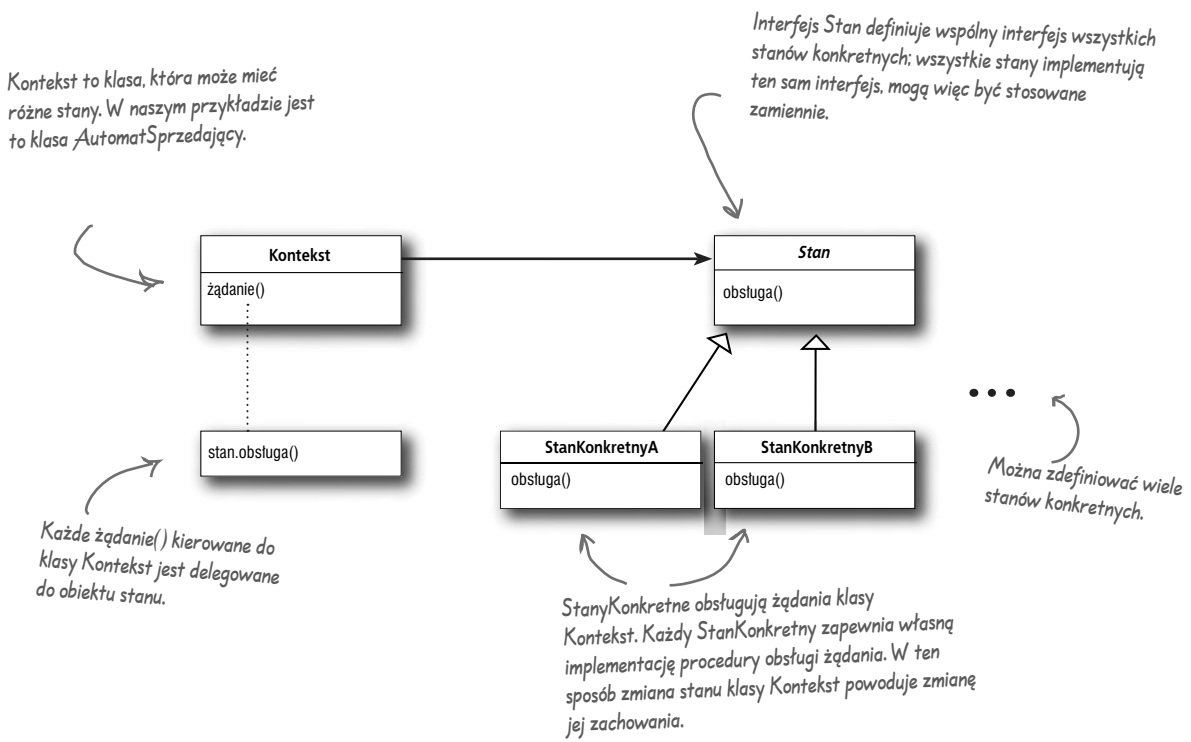
Zgadza się, właśnie zaimplementowaliśmy wzorzec Stan. Spójrzmy teraz na jego opis:

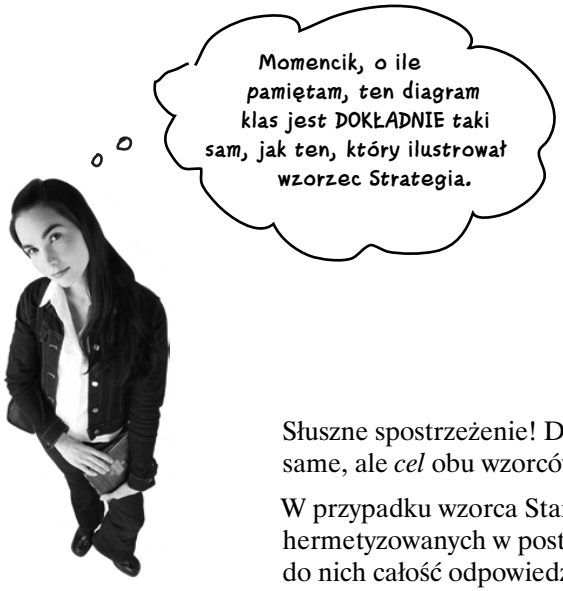
> ***Wzorzec Stan*** *umożliwia obiektowi zmianę zachowania wraz ze zmianą jego wewnętrznego stanu. Po takiej zmianie funkcjonuje on jak inna klasa.*

Pierwsza część opisu wydaje się teraz oczywista, prawda? Ponieważ wzorzec hermetyzuje stan obiektu w postaci osobnych klas i deleguje odpowiedzialność do obiektu reprezentującego stan bieżący, wynikająca ze zmiany stanu modyfikacja zachowania jest zrozumiała. Automat sprzedający był dobrym przykładem: gdy znajduje się on w stanie NieMaMonety i klient wrzuci monetę, automat przyjmuje ją, czyli zachowuje się zupełnie inaczej niż w sytuacji, gdy moneta zostanie wrzucona, a on jest w stanie JestMoneta (wówczas odmawia przyjęcia monety).

Co z drugą częścią definicji? Co to znaczy, że obiekt „funkcjonuje jak inna klasa”? Spójrzmy z perspektywy klienta: jeżeli wykorzystywany obiekt całkowicie zmienia swoje zachowania, może się wydawać, że jest to już obiekt zupełnie innej klasy. W rzeczywistości korzystamy z kompozycji, aby zapewniać złudzenie zmiany klasy poprzez proste odwoływanie się do różnych obiektów stanu.

Przyjrzyjmy się teraz diagramowi klas wzorca State:





Słuszne spostrzeżenie! Diagramy klas pozostają zasadniczo takie same, ale *cel* obu wzorców jest inny.

W przypadku wzorca Stan mamy zespół zachowań hermetyzowanych w postaci obiektów stanu. Kontekst deleguje do nich całość odpowiedzialności. Gdy aplikacja pracuje, bieżący stan zmienia się i różne obiekty reprezentują wewnętrzny stan kontekstu. W efekcie zmienia się zachowanie obiektu Kontekst. Klient zazwyczaj wie niewiele lub nie wie nic o obiektach stanu.

W przypadku wzorca Strategia klient zazwyczaj wskazuje wykorzystywany przez kontekst obiekt strategii. Wzorzec jest wystarczająco elastyczny, aby umożliwiać zmianę obiektu strategii w trakcie wykonywania programu, ale najczęściej istnieje jeden taki obiekt, dopasowany do danego obiektu kontekstu. Przykładowo, w rozdziale 1. część kaczek została skonfigurowana do latania przy użyciu typowego zachowania latania, a inne — przy użyciu zachowania latania, które w rzeczywistości trzymało je na ziemi.

Ogólnie wzorzec Strategia można traktować jako elastyczną alternatywę dla tworzenia podklas. Jeżeli podczas definiowania zachowania klasy korzystamy z dziedziczenia, jego zmiana będzie znacznie utrudniona. Gdy korzystamy z wzorca Strategia, możemy zmieniać zachowanie, wprowadzając nowe obiekty.

Wzorzec Stan jest alternatywą dla szeregu instrukcji warunkowych. Po wprowadzeniu hermetyzacji zachowań w różnych obiektach stanu zmianę zachowania można uzyskać przez prostą zmianę obiektu stanu.

Nie istnieją głupie pytania

P: W aplikacji automatu sprzedającego klasy stanu decydują o jego zmianie. Czy zawsze tak jest, że klasy StanKonkretny same decydują o zmianie?

O: Nie, nie zawsze. Alternatywą jest pozostawienie decyzji o zmianach stanu klasie Kontekst.

Ogólną zasadą jest to, że gdy zmiany stanu są trwale określone, można je zapisać w klasie Kontekst. Gdy są bardziej dynamiczne, umieszczamy je w klasach stanów (przykładowo, w automacie sprzedającym — decyzja o przejściu do stanu NieMaMonety lub BrakGum zależała od liczby gum umieszczonych w automacie w czasie wykonywania programu).

Wadą implementowania zmian stanu w klasach stanu są zależności powstające między tymi klasami. W naszej implementacji automatu sprzedającego dążyliśmy do ich ograniczenia poprzez użycie metod typu getter, udostępnianych przez klasę Kontekst.

Zwróćmy uwagę, że podejmując taką decyzję, decydujemy zarazem o tym, które klasy zostaną zamknięte dla zmian — Kontekst czy klasy stanu.

P: Czy klienty kiedykolwiek wchodzą w bezpośrednie interakcje ze stanami?

O: Nie. Klasy stanu są wykorzystywane przez obiekt Kontekst do reprezentacji jego wewnętrznego stanu i zachowań. Wszystkie wywołania pochodzą wyłącznie z obiektu Kontekst. Klienty nie zmieniają też stanu obiektu Kontekst bezpośrednio. Obiekt ten sam jest odpowiedzialny za swój stan i zazwyczaj nie chcemy, aby klient miał możliwość ustawiania stanu bez wiedzy obiektu.

P: Jeżeli w aplikacji występuje wiele egzemplarzy klasy Kontekst, czy mogą one wspólnie wykorzystywać pewne obiekty stanu?

O: Zdecydowanie tak. Jest to dość typowe. Jedynym warunkiem jest to, żeby obiekty stanu nie przechowywały własnych informacji o stanie. Jeżeli obiekt ma własne informacje o stanie, konieczne jest stosowanie osobnych egzemplarzy. Aby zapewnić możliwość współużytkowania stanów, będziemy zazwyczaj przypisywać je do statycznych zmiennych egzemplarza. Jeżeli obiekty stanu muszą korzystać z metod lub zmiennych egzemplarza klasy Kontekst, każdej metodzie obsługa() musi zostać przekazane odwołanie do właściwego egzemplarza.

P: Wygląda na to, że użycie wzorca Stan zawsze prowadzi do zwiększenia liczby klas w projekcie. Spójrzmy, ile klas przybyło w naszym przykładzie!

O: Słusznie. Podczas stosowania hermetyzacji zachowań stanowych w osobnych klasach zawsze uzyskamy większą liczbę klas. Jest to cena za elastyczność. O ile budowany kod nie jest implementacją „jednorazową", której więcej nie będziemy rozbudowywać, wprowadzenie dodatkowych klas będzie praktycznie zawsze korzystne. Zwróćmy uwagę, że tym, co jest często najważniejsze, jest liczba klas ujawnianych klientom. Pozostałe zawsze można ukryć, w ten czy inny sposób (na przykład ograniczając ich widoczność do pakietu).

Rozważmy też alternatywę: jeżeli mamy aplikację, w której występuje dużo stanów i przejść między nimi, a nie zdecydujemy się na użycie osobnych obiektów, wynikiem naszej pracy będą ogromne, rozbudowane i nieprzejrzyste instrukcje warunkowe. Kod będzie trudno rozbudowywać i poprawiać. Użycie obiektów zapewnia, że implementacja jest jasna, a zapoznanie się z nią i wprowadzanie zmian to czynności niewymagające dużej ilości czasu.

P: Diagram klas wzorca Stan przedstawia Stan jako klasę abstrakcyjną. Ale przecież w implementacji automatu sprzedającego użyliśmy interfejsu...

O: Tak. Użyliśmy interfejsu, ponieważ nie było żadnych wspólnych funkcji, które można zaimplementować w klasie abstrakcyjnej. Przy rozwiązywaniu innych problemów zawsze warto rozważyć wprowadzenie klasy. Umożliwia to, między innymi, zapisywanie w niej wspólnych zachowań w trakcie dalszej pracy nad aplikacją. W efekcie unikamy modyfikowania wielu klas stanu jednocześnie.

Została jeszcze promocja 1 z 10

Jeszcze nie skończyliśmy. Mamy do zaimplementowania grę promocyjną. Teraz, gdy zastosowaliśmy wzorzec Stan, operacja powinna pójść gładko. Przede wszystkim musimy wprowadzić do klasy automatu nowy stan:

```
public class AutomatSprzedający {

    Stan stanBrakGum;
    Stan stanNieMaMonety;
    Stan stanJestMoneta;
    Stan stanGumaSprzedana;
    Stan stanWygrana;

    Stan stan = stanBrakGum;
    int liczba = 0;

    // metody
}
```

Tutaj dodajemy tylko deklarację nowego stanu. Zmienną zainicjujemy w konstruktorze.

Nie zapomnijmy, że musimy dodać metodę typu getter dla nowego stanu.

Teraz zaimplementujemy klasę StanWygrana, w dużej mierze podobną do klasy StanGumaSprzedana:

```
public class StanWygrana implements Stan {

    // zmienne egzemplarza i konstruktor

    // komunikat błędu przy wrzucaniu monety

    // komunikat błędu przy próbie zwrócenia monety

    // komunikat błędu przy przekręcaniu gałki

    public void wydaj() {
        System.out.println("WYGRAŁEŚ! Dostajesz drugą gumę");
        automatSprzedający.zwolnijGumę();
        if (automatSprzedający.pobierzLiczba() == 0) {
            automatSprzedający.ustawStan(automatSprzedający.pobierzStanBrakGum());
        } else {
            automatSprzedający.zwolnijGumę();
            if (automatSprzedający.pobierzLiczba() > 0) {
                automatSprzedający.ustawStan(automatSprzedający.pobierzStanNieMaMonety());
            } else {
                System.out.println("Ups, koniec gum!");
                automatSprzedający.ustawStan(automatSprzedający.pobierzStanBrakGum());
            }
        }
    }
}
```

Podobnie jak w klasie StanGumaSprzedana.

Zwalniamy dwie gumy, po czym przechodzimy do stanu StanNieMaMonety lub StanBrakGum.

Jeśli tylko mamy drugą gumę, zwalniamy ją z automatu.

Kończymy implementowanie promocji

Musimy wprowadzić jeszcze jedną zmianę: zaimplementować grę losową i przejście do stanu StanWygrana. Odpowiedni kod znajdzie się w klasie StanJestMoneta, bo właśnie w tym stanie klient przekręca gałkę, która zwalnia jedną lub dwie gumy:

```
public class StanJestMoneta implements Stan {
    Random losowaWygrana = new Random(System.currentTimeMillis());
    AutomatSprzedający automatSprzedający;

    public StanJestMoneta(AutomatSprzedający automatSprzedający) {
        this.automatSprzedający = automatSprzedający;
    }

    public void włóżMonetę() {
        System.out.println("Nie możesz włożyć więcej niż jednej monety");
    }

    public void zwróćMonetę() {
        System.out.println("Moneta zwrócona");
        automatSprzedający.setStan(automatSprzedający.pobierzStanNieMaMonety());
    }

    public void przekręćGałkę() {
        System.out.println("Obróciłeś gałkę...");
        int wygrana = losowaWygrana.nextInt(10);
        if (wygrana == 0) {
            automatSprzedający.ustawStan(automatSprzedający.pobierzStanWygrana());
        } else {
            automatSprzedający.ustawStan(automatSprzedający.pobierzStanGumaSprzedana());
        }
    }

    public void wydaj() {
        System.out.println("Nie wydano gumy");
    }
}
```

Najpierw inicjujemy generator liczb losowych, który posłuży do wybierania wygranych (z 10-procentowym prawdopodobieństwem)...

... następnie określamy, czy klient wygrał.

W przypadku, gdy klient wygrał, a w automacie jest wystarczająca liczba gum, aby wydać dwie, przechodzimy do stanu StanWygrana. W pozostałych przypadkach przechodzimy do stanu GumaSprzedana (zgodnie z wcześniejszą procedurą).

To było naprawdę proste! Dodaliśmy nowy stan automatu i zaimplementowaliśmy go. Potem pozostało już tylko zapisanie kodu gry i przejścia do nowego stanu. Wygląda na to, że nowe podejście zaczyna przynosić pierwsze korzyści…

Wersja demo dla prezesa

Prezes naszego zleceniodawcy, firmy Automaty Sprzedające SA, przyszedł, aby zobaczyć demonstrację pracy nowego kodu. Miejmy nadzieję, że wszystko jest w porządku! Nasze demo będzie krótkie i przyjemne (prezesi, jak wiadomo, nie są zbyt cierpliwi). Powinno być jednak wystarczająco długie, aby chociaż raz wygrać.

Kod testu nie zmienił się. Został tylko nieco skrócony.

Ponownie rozpoczniemy od 5 gum w automacie.

```
public class AutomatSprzedającyTest {

    public static void main(String[] args) {
        AutomatSprzedający automatSprzedający = new AutomatSprzedający(5);

        System.out.println(automatSprzedający);

        automatSprzedający.włóżMonetę();
        automatSprzedający.przekręćGałkę();

        System.out.println(automatSprzedający);

        automatSprzedający.włóżMonetę();
        automatSprzedający.przekręćGałkę();
        automatSprzedający.włóżMonetę();
        automatSprzedający.przekręćGałkę();

        System.out.println(automatSprzedający);
    }
}
```

Chcemy uzyskać wygraną, więc będziemy po prostu wrzucać monety i przekręcać gałkę. Za każdym razem wyświetlamy stan urządzenia…



Cały zespół czeka w pokoju obok, ciekawy, czy nowy, oparty na wzorcu Stan projekt będzie działał!!




```
F:\WINDOWS\system32\cmd.exe
F:\1pawel\helion\Head First Design Patterns\hfdp\classes>java headfirst.state.au
tomatsprzedajacywygrana.AutomatSprzedajacyTest
Automaty Sprzedające SA
Wolnostojący automat sprzedający Gumball Model #2004 Java
Zapas: 5 gumy
Automat oczekuje na monetę

Moneta przyjęta
Obróciłeś gałkę...
WYGRAŁEŚ! Dostajesz drugą gumę
Wypada guma...
Wypada guma...

Automaty Sprzedające SA
Wolnostojący automat sprzedający Gumball Model #2004 Java
Zapas: 3 gumy
Automat oczekuje na monetę

Moneta przyjęta
Obróciłeś gałkę...
WYGRAŁEŚ! Dostajesz drugą gumę
Wypada guma...
Wypada guma...
Moneta przyjęta
Obróciłeś gałkę...
Wypada guma...
Ups, koniec gum!

Automaty Sprzedające SA
Wolnostojący automat sprzedający Gumball Model #2004 Java
Zapas: 0 gumy
Automat pusty


F:\1pawel\helion\Head First Design Patterns\hfdp\classes>_
```

Nie istnieją głupie pytania

P: Po co nam klasa StanWygrana? Czy klasa StanGumaSprzedana nie może zwalniać dwóch gum?

O: Dobre pytanie. Klasy StanGumaSprzedana i StanWygrana są niemal identyczne, z wyjątkiem tego, że druga z nich wydaje dwie gumy. Kod wydawania dwóch gum niewątpliwie można umieścić w klasie StanGumaSprzedana. Jednak wadą takiego rozwiązania będzie to, że jedna klasa stanu reprezentuje wówczas DWA stany automatu: stan, w którym klient wygrał darmową gumę i ma dostać dwie, i stan, w którym zwyczajnie kupił gumę i czeka na jej otrzymanie. Jest to wybór między przejrzystością kodu a jego powielaniem. Innym aspekt takiego rozwiązania wiąże się z zasadą przedstawioną w poprzednim rozdziale: jedna klasa, jeden zakres odpowiedzialności. Przypisanie obsługi stanu, w którym wydajemy dwie gumy, klasie StanGumaSprzedana, prowadzi do objęcia tą klasą DWÓCH zakresów odpowiedzialności. Co stanie się, gdy promocja zostanie zakończona? Albo zmienią się jej zasady? Mamy tu więc do czynienia z istotną dla projektu, wcale nieoczywistą, decyzją.



Weryfikacja projektu...

Pomysłami prezesa na razie nie będziemy się zajmować. Zastanowimy się natomiast nad pewnymi aspektami tego projektu, które powinny zwrócić naszą uwagę w trakcie pracy nad ostateczną wersją programu:

- Klasy StanGumaSprzedana i StanWygrana mają wiele powtarzających się wierszy kodu. Być może warto pokusić się tu o pewną optymalizację. Jak można uniknąć takich powtórzeń? Możemy zastąpić interfejs Stan klasą abstrakcyjną i zapisać w niej pewne standardowe zachowania metod. Ostatecznie komunikaty o błędach, takie jak „Nie możesz włożyć więcej niż jednej monety", nie będą przecież wyświetlane klientowi. Wszystkie zachowania związane z obsługą błędów mogą więc być uogólnione i dziedziczone po klasie abstrakcyjnej Stan.
- Metoda wydaj() jest wywoływana za każdym razem, nawet gdy gałka zostanie obrócona jeszcze przed włożeniem monety. Automat działa, co prawda, poprawnie i nie wydaje gum, kiedy nie jest w odpowiednim stanie, ale można to nieco uprościć, wprowadzając parametr wyjściowy metody przekręćGałkę() w postaci wartości binarnej lub używając wyjątków. Co będzie najlepsze?
- Cała inteligencja zmian stanów została zapisana w klasach Stan. Jakie mogą z tego wyniknąć problemy? Czy nie byłoby lepiej przenieść tej logiki do klasy AutomatSprzedający? Jakie są wady i zalety każdego z tych rozwiązań?
- Czy będziemy tworzyć wiele egzemplarzy obiektów typu AutomatSprzedający? Jeżeli tak, korzystniejsze może być przeniesienie egzemplarzy stanu do statycznych zmiennych egzemplarza i współużytkowanie ich. Jakich zmian by to wymagało?

Kurczę. Jasiu, jestem automatem do sprzedawania gumy, a nie komputerem.



Dzisiaj: ***wzorce Strategia i Stan — rozdzieleni bracia***

Strategia

Cześć, braciszku. Słyszałeś, że byłem w rozdziale pierwszym?

Byłem właśnie niedaleko, pomagałem gościowi od wzorca Metoda Szablonowa — potrzebowali mnie do skończenia tamtego rozdziału. No więc, cóż zamierza robić mój szlachetny brat?

No nie… cokolwiek od Ciebie słyszę, brzmi to tak, jakbyś po prostu kopiował ode mnie i tylko opisywał to innymi słowami. Pomyśl: ja pozwalam obiektom korzystać z różnych zachowań lub algorytmów poprzez użycie kompozycji i delegacji. Małpujesz mnie.

Tak? Jak to? Nie bardzo wiem, co masz na myśli.

Tak, to było *ładne*… mam nadzieję, że rozumiesz, o ile większe możliwości dawało takie podejście niż próba podążania Twoją ścieżką?

Chyba będziesz musiał to wyjaśnić.

Stan

Tak, trudno było nie zauważyć.

To samo, co zwykle — pomagać obiektom w przejawianiu zachowań zróżnicowanych odpowiednio do ich stanów.

Przyznaję, że to, co robimy, na pewno jest jakoś zbliżone, jednak moje zamiary są zupełnie inne niż Twoje. No i sposób, w jaki uczę moich klientów korzystania z kompozycji i delegacji, jest zupełnie inny.

Gdybyś spędził trochę czasu, myśląc o czymś innym niż *Ty sam*, może byś w końcu załapał. Ale dobrze, zastanów się nad tym, jak pracujesz: tworzysz egzemplarz klasy i dajesz jej obiekt strategii, który implementuje pewne zachowania. W taki sposób przydzielałeś zachowanie kwakania w rozdziale pierwszym, prawda? Prawdziwe kaczki dostawały prawdziwe kwakanie, a gumowe — jakieś piski.

Tak, oczywiście. Ale teraz pomyśl o tym, w jaki sposób ja działam. Moje podejście jest zupełnie inne.

Strategia

Zaraz, ja też mogę zmieniać zachowania w czasie wykonywania programu; przecież o to chodzi w kompozycji!

No cóż, musze przyznać, że faktycznie nie zachęcam moich obiektów do ścisłego definiowania zbioru dopuszczalnych zmian stanów. Właściwie to wolę kontrolować to, której strategii używają.

No jasne, trzymaj się swoich marzeń, bracie. Zachowujesz się, jakbyś był równie poważnym wzorcem co ja, a przecież ja byłem w rozdziale pierwszym, a Ciebie wepchnęli do dziesiątego. Chodzi mi o to, komu się chce tyle czytać?

Cały mój brat, wieczny marzyciel.

Stan

Gdy ja tworzę obiekt Kontekst, mogę określić jego stan początkowy. Jednak potem on zmienia stan.

Oczywiście, że możesz, ale moja praca opiera się na stanach dyskretnych; moje obiekty Kontekst zmieniają stan odpowiednio do pewnych ściśle zdefiniowanych zasad. Inaczej mówiąc, zmiany zachowania są wewnętrznym elementem mojego schematu — o to chodzi!

Posłuchaj, już uzgodniliśmy, że nasza struktura jest podobna, ale cele dość odmienne. Musimy się zgodzić, że każdy z nas ma swoje zastosowania.

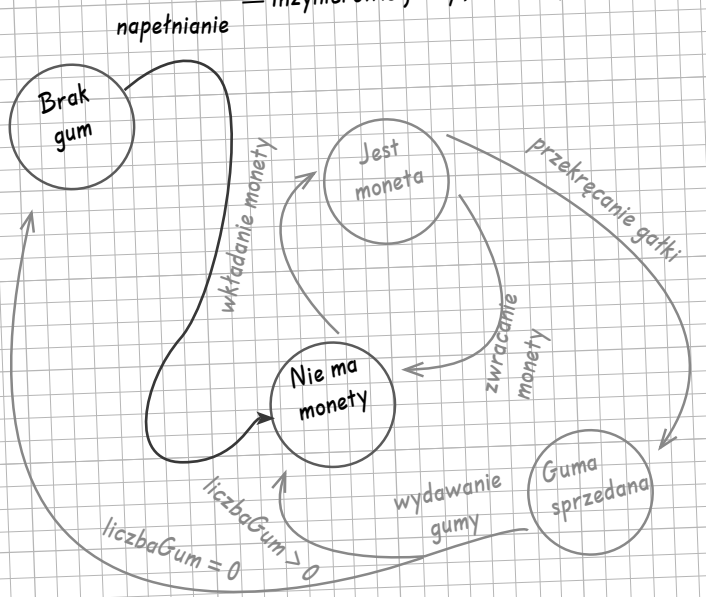
Żartujesz? To jest książka z serii *Rusz głową!*. Jej czytelnicy są niesamowici. Jasne, że dojdą do dziesiątego rozdziału!

Niemal zapomnieliśmy!

Automaty Sprzedające SA
gdzie automatyczny sprzedawca nigdy nie jest w połowie pusty...

Istnieje jeszcze jedna zmiana stanu, o której zapomnieliśmy w pierwotnej specyfikacji... Potrzebujemy sposobu, aby móc załadować urządzenie nowymi gumami, gdy obecne się skończą! Przedstawiamy więc nowy diagram — czy możecie to zaimplementować? Z całą resztą poszło Wam tak dobrze, że nie wątpimy, iż taka modyfikacja okaże się bardzo prosta!

— Inżynierowie firmy Automaty Sprzedające





Zaostrz ołówek

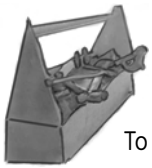
Musimy napisać metodę napełnij(). Pobiera ona jeden parametr — liczbę ładowanych gum. Powinna aktualizować przechowywaną w automacie informację o liczbie gum i odpowiednio zmieniać stan urządzenia.





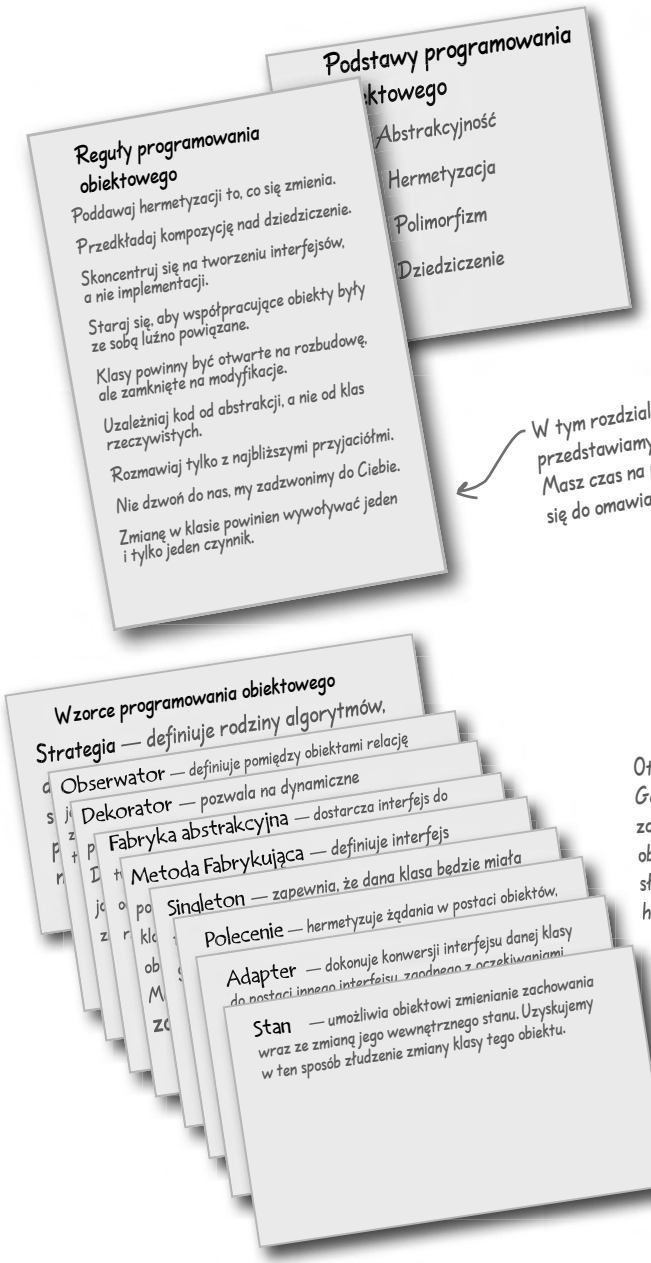
Dopasuj wzorce do opisów:

Wzorzec	Opis
Stan	Hermetyzuje wymienne zachowania i wykorzystuje mechanizm delegacji do wyboru, które zachowanie zostanie użyte.
Strategia	Klasy podrzędne decydują o tym, w jaki sposób zostaną zaimplementowane poszczególne kroki algorytmu.
Metoda Szablonowa	Hermetyzuje zachowania stanowe i deleguje zachowanie do stanu bieżącego.



Twoja skrzynka narzędziowa

To koniec kolejnego rozdziału; znasz już wystarczająco dużo wzorców, aby zabłysnąć na niejednej rozmowie kwalifikacyjnej!

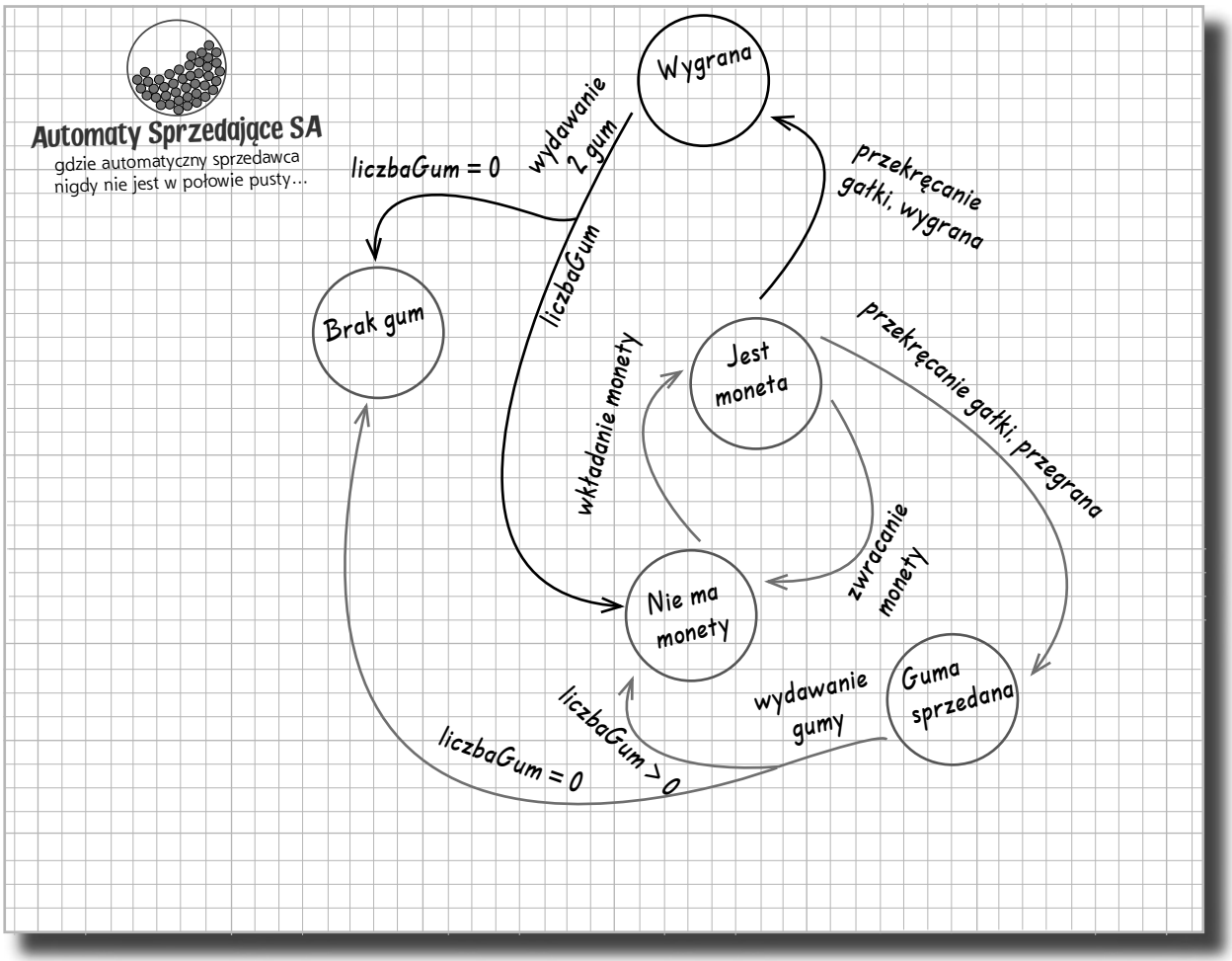


CELNE SPOSTRZEŻENIA

- Wzorzec Stan umożliwia obiektowi przejawianie wielu różnych zachowań, odpowiednio do jego wewnętrznego stanu.
- W przeciwieństwie do proceduralnej maszyny stanowej wzorzec Stan prowadzi do reprezentacji stanu jako samodzielnej klasy.
- Obiekt Kontekst realizuje swoje zachowania poprzez delegację do bieżącego obiektu stanu.
- Poprzez hermetyzację stanów w klasach lokalizujemy przyszłe zmiany w kodzie.
- Wzorce Stan i Strategia mają ten sam diagram klas, różni się cel ich stosowania.
- Gdy stosowany jest wzorzec Strategia, obiekty Kontekst są konfigurowane przy użyciu zachowania lub algorytmu.
- Gdy stosowany jest wzorzec Stan, obiekt Kontekst zmienia swoje zachowanie odpowiednio do zmian swojego stanu.
- Zmiany stanu mogą być inicjowane przez klasy stanu lub klasę Kontekst.
- Użycie wzorca Stan prowadzi do zwiększenia liczby klas projektu.
- Klasy stanu mogą być współużytkowane przez różne egzemplarze klasy Kontekst.



Rozwiązania ćwiczeń





Rozwiązania ćwiczeń



Zaostrz ołówek

Które stwierdzenia są prawdziwe dla bieżącego stanu naszej implementacji?

- [x] A. Ten kod na pewno nie spełnia zasady otwarcia na rozbudowę i zamknięcia na modyfikacje.
- [x] B. Programista języka FORTRAN byłby z takiego programu dumny.
- [x] C. Ten projekt nie jest nawet obiektowy
- [x] D. Zmiany stanu nie są jawne; giną pośród ogromnej liczby instrukcji warunkowych.
- [x] E. Nie wprowadziliśmy żadnej hermetyzacji elementów zróżnicowanych.
- [x] F. Dalsza rozbudowa kodu spowoduje zapewne wiele drobnych błędów.



Zaostrz ołówek

Pozostaje jedna niezaimplementowana klasa: StanBrakGum. Może sam ją napiszesz? Wymaga to uważnego rozważenia czynności automatu w każdej sytuacji. Przed dalszą lekturą sprawdź, jak wygląda nasza implementacja...

Pusty automat nie wykona żadnych operacji do momentu ponownego napełnienia.

```
public class StanBrakGum implements Stan {
    AutomatSprzedający automatSprzedający;

    public StanBrakGum(AutomatSprzedający automatSprzedający) {
        this.automatSprzedający = automatSprzedający;
    }

    public void włóżMonetę() {
        System.out.println("Nie można wrzucić monety, automat jest pusty");
    }

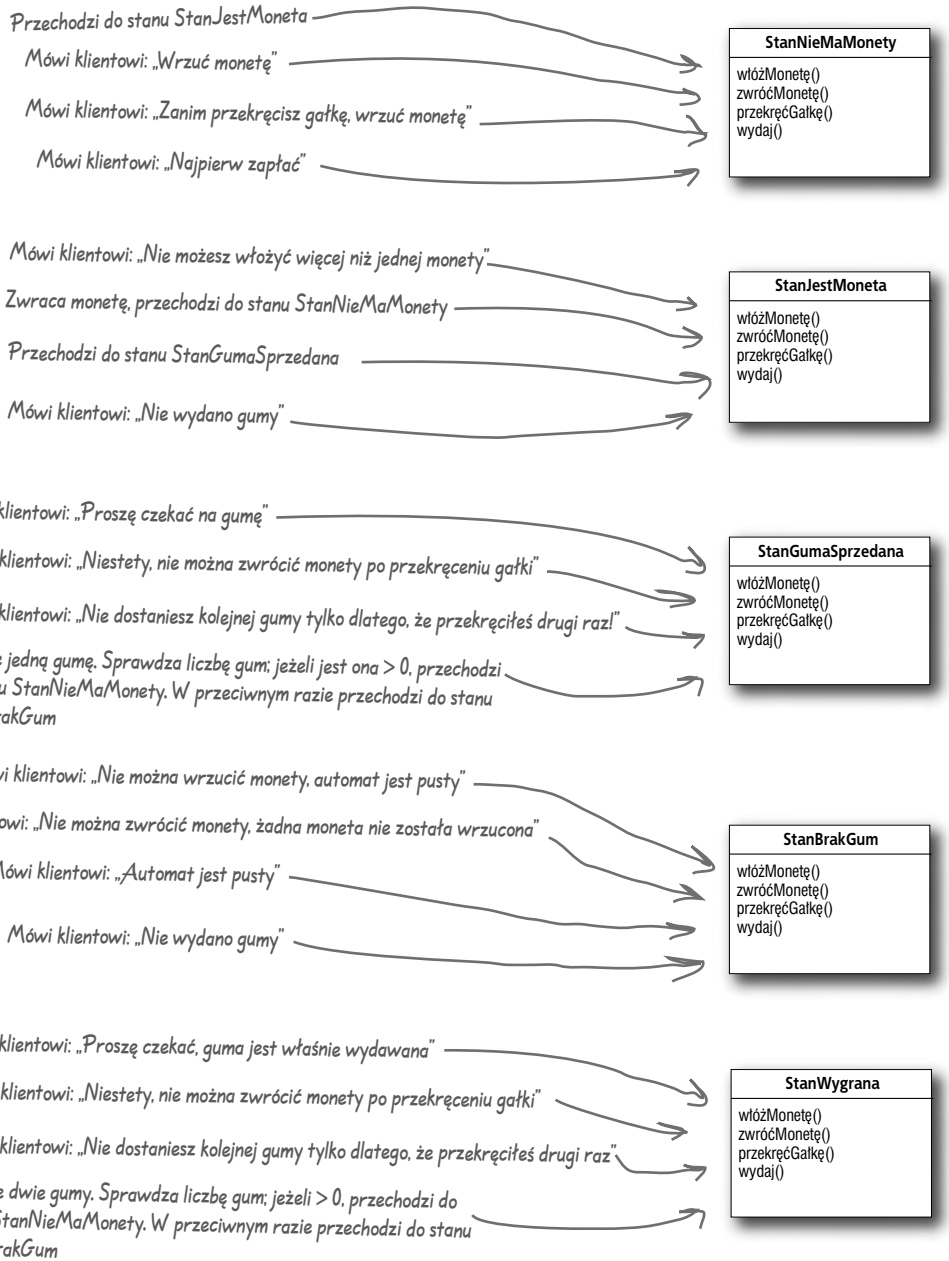
    public void zwróćMonetę() {
        System.out.println("Nie można zwrócić monety, żadna moneta nie została wrzucona");
    }

    public void przekręćGałkę() {
        System.out.println("Obróciłeś gałkę, ale automat jest pusty");
    }

    public void wydaj() {
        System.out.println("Nie wydano gumy");
    }
}
```

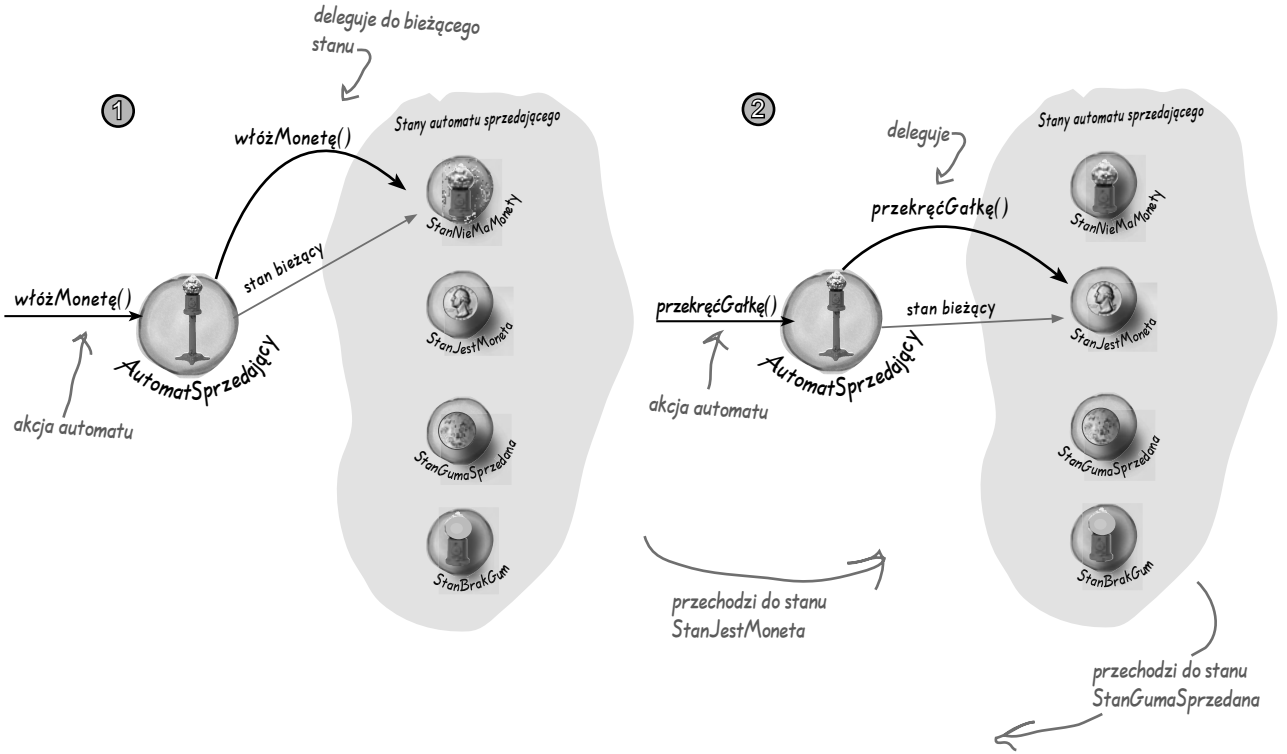
Zaostrz ołówek

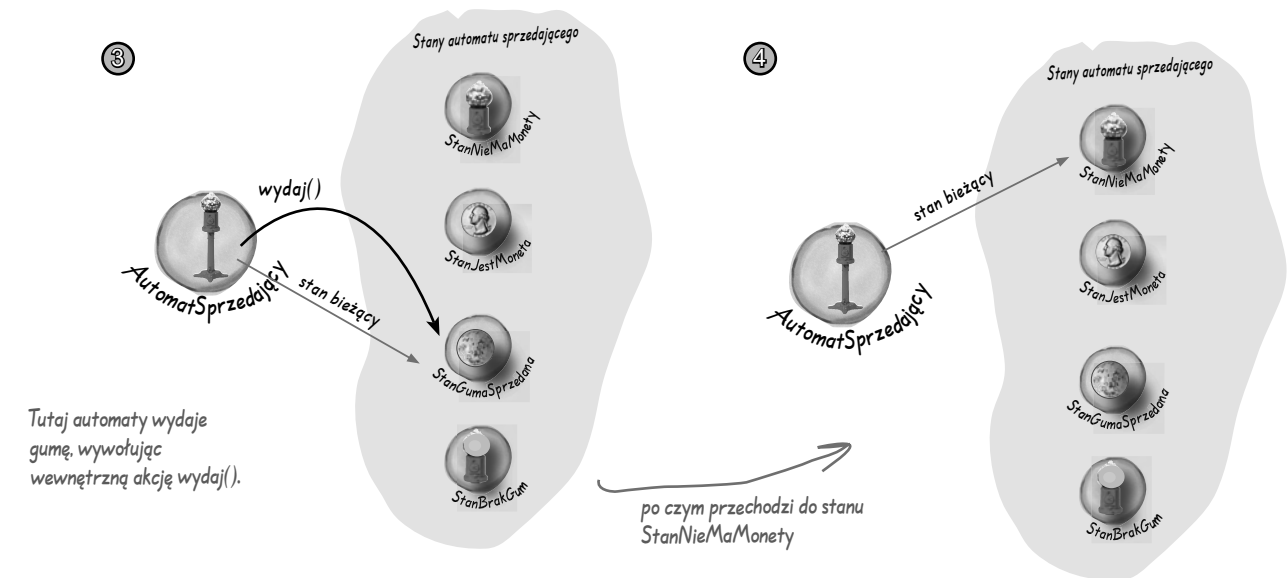
Aby zaimplementować wymagane stany, musimy określić zachowania klas w chwili, gdy wywoływane są poszczególne akcje. Uzupełnij poniższy diagram opisami zachowań przypisanych każdej akcji w poszczególnych klasach. Kilka przykładowych opisów już wprowadziliśmy.



Za kulisami: *samodzielna wycieczka*

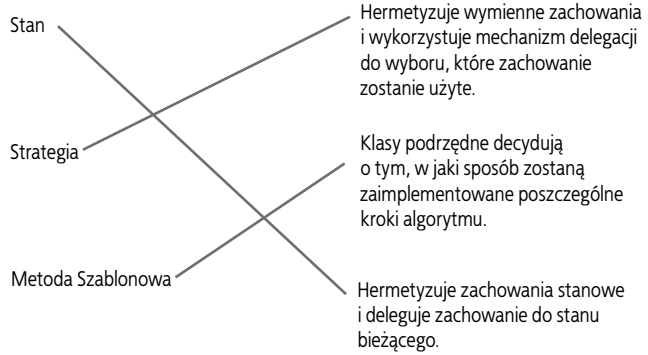






Kto co robi

Dopasuj wzorce do opisów:



Zaostrz ołówek

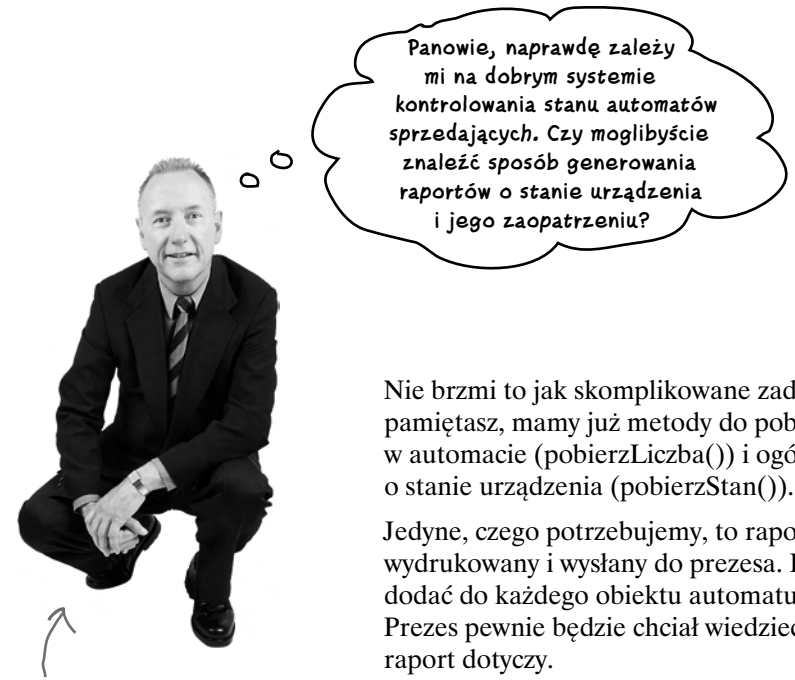
Musimy napisać metodę napełnij(). Pobiera ona jeden parametr — liczbę ładowanych gum. Powinna aktualizować przechowywaną w automacie informację o liczbie gum i odpowiednio zmieniać stan urządzenia.

```
void napełnij(int liczba) {
    this.liczba = liczba;
    stan = stanNieMaMonety;
}
```

Kontrola dostępu do obiektu



Próbowałeś kiedyś stosować metodę „dobrego i złego"? Ty jesteś tym dobrym, który zrobi wszystko, o co się go poprosi, który jest zawsze miły i uprzejmy. Nie chcesz jednak, żeby *każdy* mógł prosić o Twoje usługi. To jest miejsce dla „złego", który będzie *kontrolował dostęp* do Ciebie. Takie jest właśnie zadanie pośredników (ang. *proxy*) w modelu obiektowym — kontrolowanie dostępu i zarządzanie nim. Jak się przekonamy, istnieje *bardzo wiele* schematów takiego pośrednictwa. Obiekty Proxy mogą przekazywać wywoływanie metody obiektowi w innym węźle internetu; bywa też, że zastępują wyjątkowo leniwe obiekty.



Nie brzmi to jak skomplikowane zadanie. Jak zapewne pamiętasz, mamy już metody do pobierania liczby gum w automacie (pobierzLiczba()) i ogólnej informacji o stanie urządzenia (pobierzStan()).

Jedyne, czego potrzebujemy, to raport, który może zostać wydrukowany i wysłany do prezesa. Hmm, może warto dodać do każdego obiektu automatu pole lokalizacji. Prezes pewnie będzie chciał wiedzieć, którego urządzenia raport dotyczy.

Przejdźmy od razu do kodowania odpowiedniej procedury. Zaskoczymy prezesa szybkimi wynikami.

Kod monitora

Rozpocznijmy od wprowadzenia pola lokalizacji automatu:

```
public class AutomatSprzedający {
    // inne zmienne egzemplarza
     String lokalizacja;

    public AutomatSprzedający(String lokalizacja, int liczba) {
        // kod konstruktora
        this.lokalizacja = lokalizacja;
    }

    public String pobierzLokalizacja() {
        return lokalizacja;
    }

    // inne metody
}
```

Lokalizacja to po prostu String.

Lokalizacja jest przekazywana do konstruktora i przechowywana w zmiennej egzemplarza.

Dodajmy jeszcze metodę pobierającą ciąg lokalizacji.

Stwórzmy teraz nową klasę, AutomatSprzedającyMonitor, która pobiera informacje o lokalizacji automatu, jego stanie i liczbie pozostałych gum, po czym wyświetla ładny raport:

```
public class AutomatSprzedającyMonitor {
    AutomatSprzedający automat;

    public AutomatSprzedającyMonitor(AutomatSprzedający automat) {
        this.automat = automat;
    }

    public void raport() {
        System.out.println("Automat sprzedający: " + automat.pobierzLokalizacja());
        System.out.println("Towar: " + automat.pobierzLiczba() + " szt.");
        System.out.println("Bieżący stan urządzenia: " + automat.pobierzStan());
    }
}
```

Konstruktor monitora pobiera odwołanie do automatu i zapisuje je w zmiennej egzemplarza automat.

Raport obejmuje informację o lokalizacji automatu, jego napełnieniu i stanie.

Testowanie monitora

Implementacja była błyskawiczna. Prezes będzie zaskoczony i na pewno doceni nasze umiejętności.

Pozostaje utworzyć egzemplarz monitora i przekazać mu automat do kontrolowania:

```
public class AutomatSprzedającyTest {
  public static void main(String[] args) {
    int liczba = 0;

    if (args.length < 2) {
        System.out.println("AutomatSprzedający <nazwa> <towar>");
        System.exit(1);
    }

    liczba = Integer.parseInt(args[1]);
    AutomatSprzedający automatSprzedający = new AutomatSprzedający(args[0], liczba);

    AutomatSprzedającyMonitor monitor = new AutomatSprzedającyMonitor(automatSprzedający);

    // pozostały kod testu

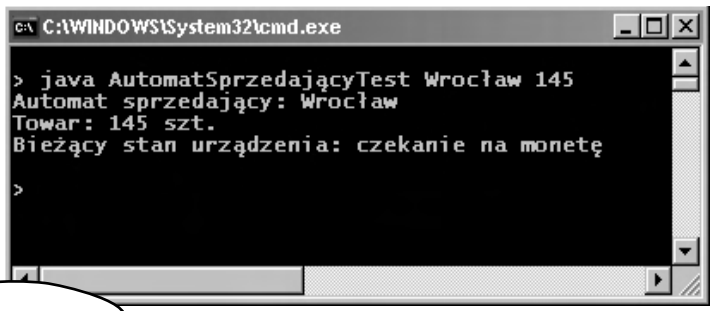
    monitor.raport();
  }
}
```

W wierszu poleceń przekazujemy lokalizację i początkową liczbę gum.

Nie zapomnijmy przekazać konstruktorowi lokalizacji i liczby...

... i utworzyć egzemplarza monitora, przekazując mu odwołanie do kontrolowanego automatu.

Gdy potrzebujemy raportu o stanie urządzenia, korzystamy z metody raport().



A o to wynik działania procedury testowej!



Wydruk monitora wygląda świetnie, ale widzę, że nie wyrażałem się do końca jasno. Potrzebuję możliwości ZDALNEJ kontroli stanu automatów! Niezbędne sieci są już gotowe. No, panowie, podobno jesteście generacją internetu!



Jacek: Zdalny co?

Franek: Zdalny Pośrednik. Kod monitora już mamy, prawda? Dajemy obiektowi AutomatSprzedającyMonitor odwołanie do automatu i on drukuje raport. Problemem jest to, że monitor pracuje w tej samej maszynie wirtualnej co automat, a prezes chce siedzieć sobie przy biurku i *zdalnie* kontrolować automaty! Może więc dałoby się pozostawić klasę monitora bez zmian, ale przekazywać jej obiekt pośredniczący — „pośrednik" obiektu *zdalnego*.

Jacek: No, nie wiem, czy rozumiem.

Janusz: Ja chyba też nie.

Franek: Zacznijmy od początku… *Pośrednik* (ang. *proxy*) zastępuje *właściwy obiekt*. W naszym przypadku pośrednik zachowuje się tak samo, jak obiekt AutomatSprzedający, jednak w rzeczywistości komunikuje się przez sieć z właściwym automatem i właściwym obiektem AutomatSprzedający.

Janusz: Więc pozostawimy nowy kod bez zmian, a monitorowi przekażemy odwołanie do obiektu, który reprezentuje automat sprzedający, ale nim nie jest…

Jacek: A ten obiekt będzie udawał, że jest automatem sprzedającym, podczas gdy w rzeczywistości cały czas będzie trwała komunikacja przez sieć z właściwym obiektem.

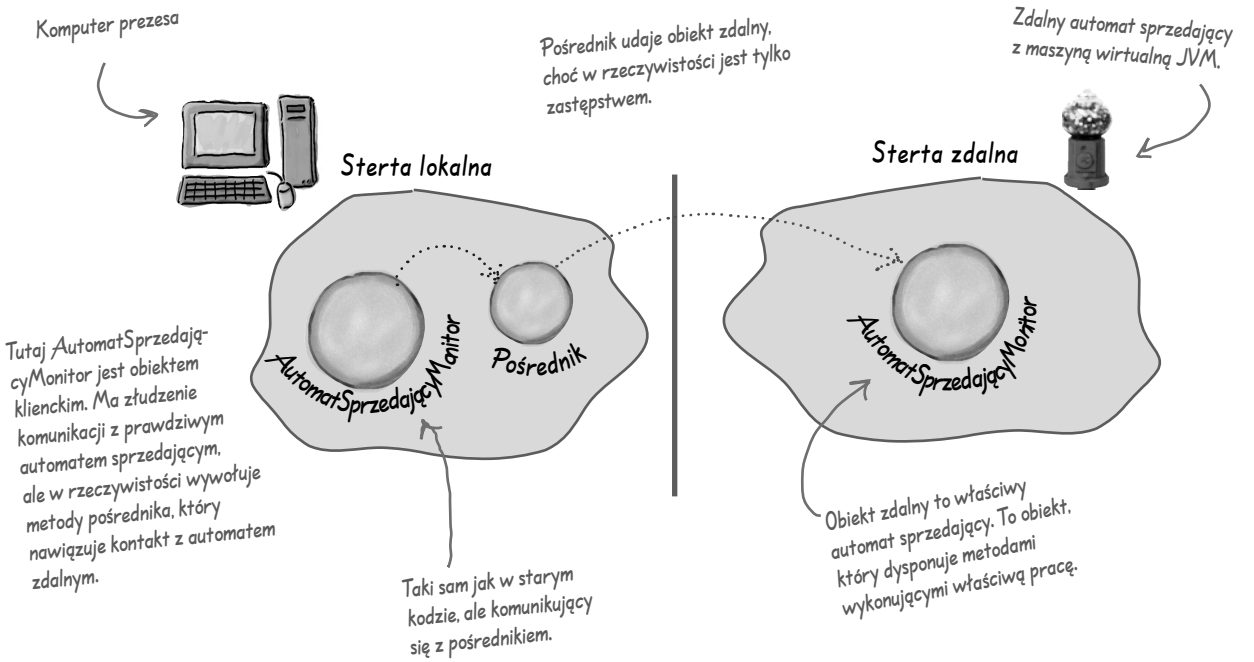
Franek: To w zasadzie wszystko na ten temat.

Jacek: Ta kombinacja wygląda na coś z gatunku „łatwo powiedzieć, trudniej zrobić".

Franek: Być może, ale myślę, że nie będzie tak źle. Musimy zapewnić automatowi sprzedającemu możliwość pracy w charakterze usługi i przyjmowania żądań z sieci. Musimy też umożliwić przekazywanie monitorowi obiektów pośredniczących. Standardowe mechanizmy języka Java bardzo nam pomogą. Omówmy jeszcze nieco szerzej zagadnienie zdalnych pośredników…

Rola „zdalnego pośrednika"

Zdalny pośrednik to *lokalny reprezentant zdalnego obiektu*. Co to jest „zdalny obiekt"? To obiekt na stercie innej wirtualnej maszyny języka Java (lub ogólniej, obiekt pracujący w innej przestrzeni adresowej). Dlaczego „lokalny reprezentant"? To obiekt, który można wywoływać lokalnie i który przekazuje wszystkie wywołania do obiektu zdalnego.



Obiekt kliencki działa tak, jakby korzystał z wywołań zdalnych. W rzeczywistości wywołuje metody obiektu-pośrednika (proxy), który pracuje w lokalnej maszynie JVM i zapewnia niezbędne procedury komunikacji sieciowej.





WYTĘŻ UMYSŁ

Zanim przejdziemy w naszych rozważaniach dalej, zastanów się nad tym, jak Ty zaprojektowałbyś system, który umożliwi korzystanie ze zdalnych wywołań metod. W jaki sposób zapewniłbyś programiście wygodę i możliwość pisania jak najmniejszej ilości kodu? Co zrobić, aby zdalne wywołania były jak najmniej kłopotliwe?



WYTĘŻ UMYSŁ

Czy korzystanie z wywołań zdalnych powinno być całkowicie przezroczyste? Czy to na pewno jest właściwe podejście? Co może być problemem?

Wprowadzanie zdalnego pośrednika do kodu monitora

Na papierze wygląda to ładnie, ale jak napisać klasę pośrednika, która potrafi wywołać metodę obiektu w zdalnej maszynie wirtualnej?

Hmm... Nie można pobrać odwołania do czegoś, co jest przechowywane na innej stercie, prawda? Innymi słowy, nie można napisać:

```
Kaczka d = <obiekt innej sterty>
```

Gdziekolwiek zmienna `d` miałaby się odwoływać, musi to znajdować się w tej samej przestrzeni sterty co kod wykonujący instrukcję. Co więc zrobić? Tu właśnie wkracza Java Remote Method Invocation (RMI, pol. *zdalne wywołania metod*)… RMI umożliwia odnalezienie obiektów zdalnej maszyny wirtualnej i korzystanie z ich metod.

Może pamiętasz omówienie RMI zamieszczone w książce „Java. Rusz głową! Wydanie II”? Jeżeli nie, zapoznaj się z poniższym omówieniem, które wprowadzi Cię w zagadnienia zdalnych wywołań. Dopiero potem zajmiemy się tym, do czego przede wszystkim zmierzamy, czyli wprowadzeniem obsługi klasy pośrednika w kodzie kontrolera automatu sprzedającego.

Oto, co zrobimy:

1. **Najpierw przedstawimy krótkie wprowadzenie do RMI. Nawet jeżeli znasz już ten mechanizm, warto przypomnieć sobie, z czym będziemy mieli teraz do czynienia.**

2. **Następnie przekształcimy klasę AutomatSprzedający w zdalną usługę udostępniającą zespół metod, które można wywoływać zdalnie.**

3. **Na końcu utworzymy klasę pośrednika, która wykorzystuje RMI, aby nawiązać komunikację z automatem sprzedającym. Ostatnim krokiem będzie takie poprawienie systemu kontrolowania, aby prezes w komfortowych warunkach mógł nadzorować dowolną liczbę automatów.**



RMI — wycieczka z przewodnikiem

Jeżeli po raz pierwszy spotykasz się z RMI, wybierz się na wycieczkę, która zajmie kilka kolejnych stron. Nawet jeżeli wywołania RMI nie są dla Ciebie nowością, powtórka będzie teraz na miejscu.



RMI — wycieczka z przewodnikiem

Zdalne wywołania metod 101

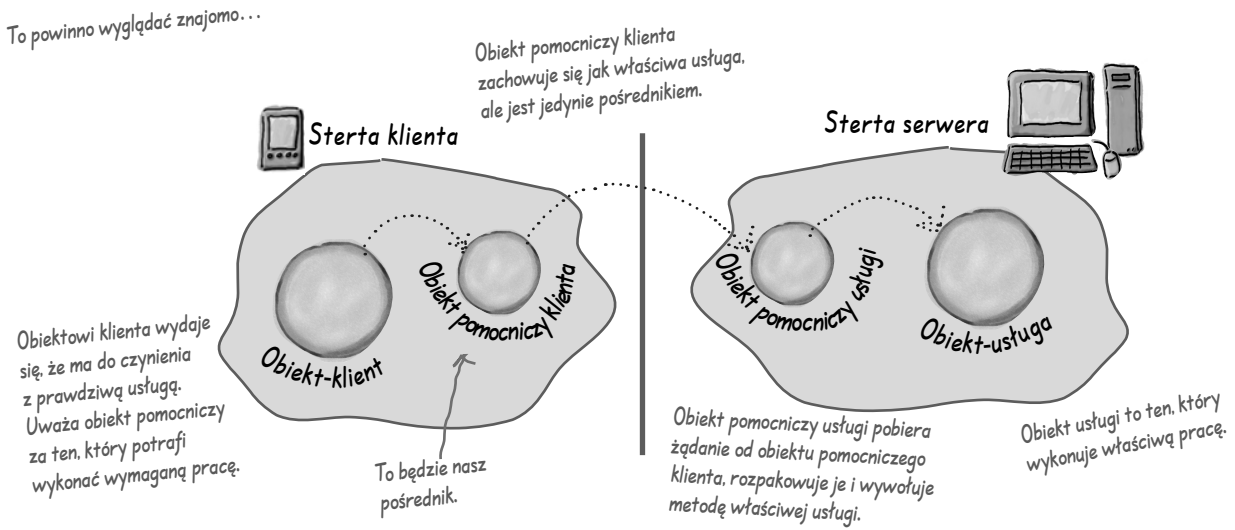
Rozważmy problem zaprojektowania systemu umożliwiającego wywoływanie lokalnego obiektu, który przekazuje żądania obiektowi zdalnemu. Jak może wyglądać taki mechanizm? Niezbędnych będzie kilka obiektów pomocniczych odpowiedzialnych za funkcje komunikacyjne. Dzięki nim klient będzie mógł pracować tak samo jak wtedy, gdy korzysta wyłącznie z wywołań lokalnych (bo faktycznie będzie z nich korzystał). Klient wywołuje wówczas metody obiektu pomocniczego tak, jakby to on był właściwą usługą. Funkcją obiektu pomocniczego jest przekazywanie żądań.

Innymi słowy, obiekt kliencki ma złudzenie wywoływania metody usługi pracującej zdalnie. Jest to możliwe dzięki temu, że obiekt pomocniczy „podszywa się" pod obiekt usługi. Klient wywołuje metodę obiektu pomocniczego, a uzyskiwane zachowanie odpowiada wywołaniu obiektu usługi.

Obiekt pomocniczy klienta nie jest w rzeczywistości usługą zdalną. Mimo, że jego zachowania są takie same (udostępnia te same metody, co usługa), obiekt pomocniczy nie zawiera żadnej logiki metody, której oczekuje klient. Jej miejsce zajmują procedury nawiązywania połączenia z serwerem, przekazywania informacji o wywołaniu (nazwa metody, jej argumenty itp.) i oczekiwania na dane wyjściowe.

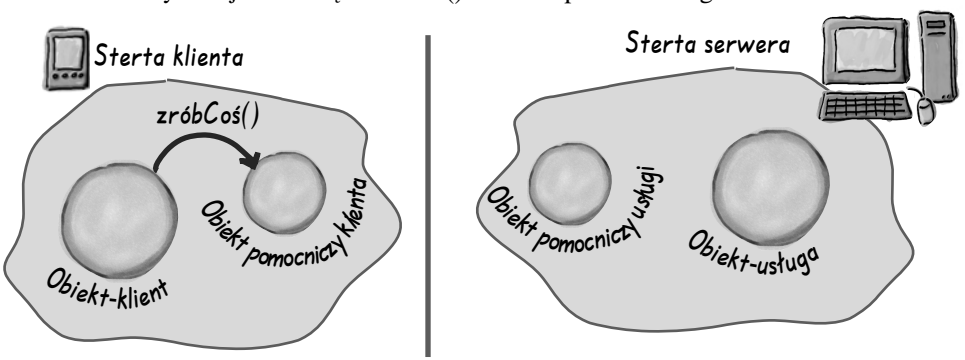
Po stronie serwera obiekt pomocniczy serwera odbiera żądanie obiektu pomocniczego klienta (korzystając z mechanizmu gniazd), rozpakowuje dane wywołania i wywołuje właściwą metodę obiektu usługi. Ostatecznie wywołanie obiektu usługi jest z punktu widzenia tej usługi wywołaniem lokalnym. Wywołującym jest obiekt pomocniczy serwera, a nie zdalny klient.

Obiekt pomocniczy usługi pobiera wartość zwracaną przez usługę, pakuje ją i przekazuje (korzystając ze strumienia wyjściowego gniazda) do obiektu pomocniczego klienta. Ten z kolei rozpakowuje odebrane dane i przekazuje wartość wyjściową klientowi.

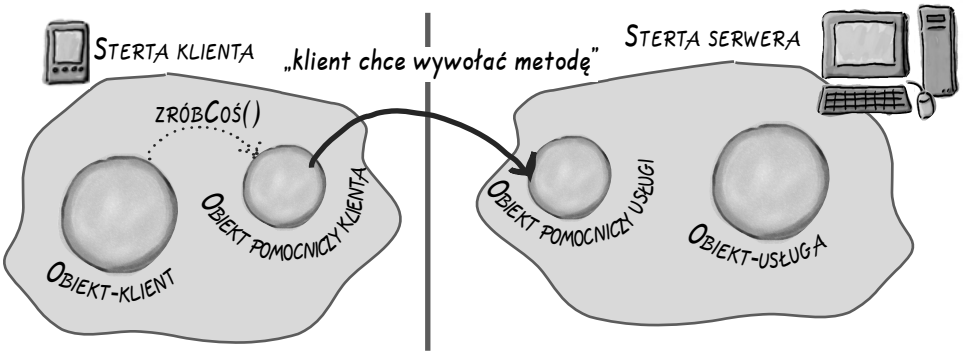


Jak przebiega wywołanie metody

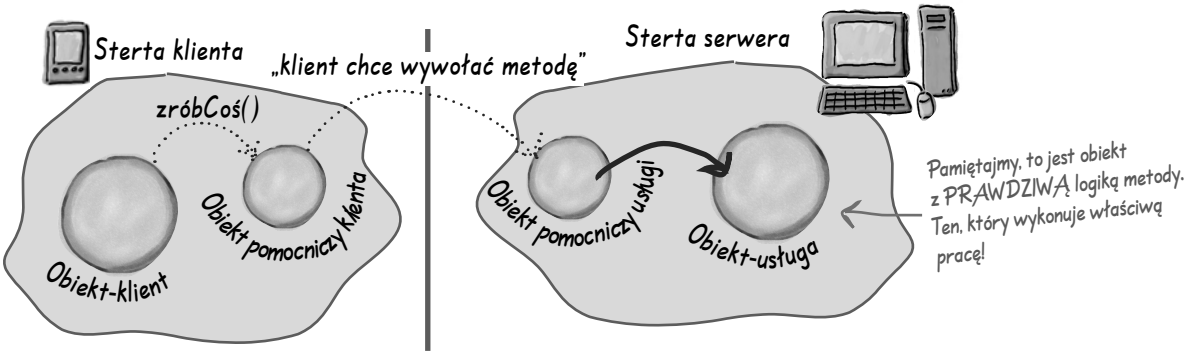
1. Obiekt-klient wywołuje metodę zrobCoś() obiektu pomocniczego klienta.



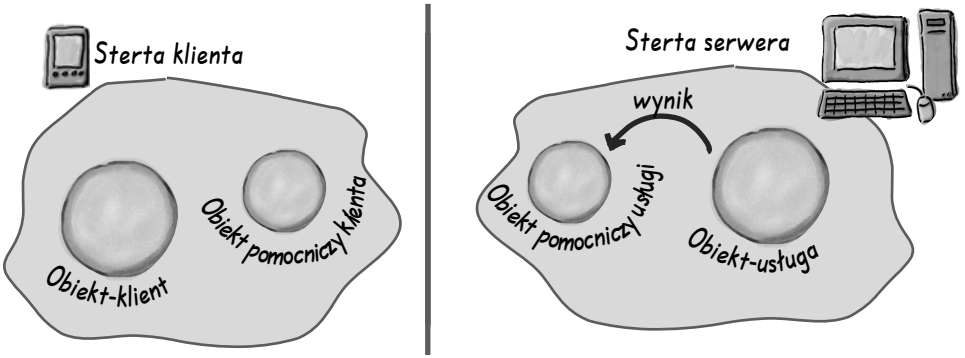
2. Obiekt pomocniczy klienta pakuje informacje o wywołaniu (argumenty, nazwę metody itp.), po czym przekazuje je poprzez sieć do obiektu pomocniczego usługi.



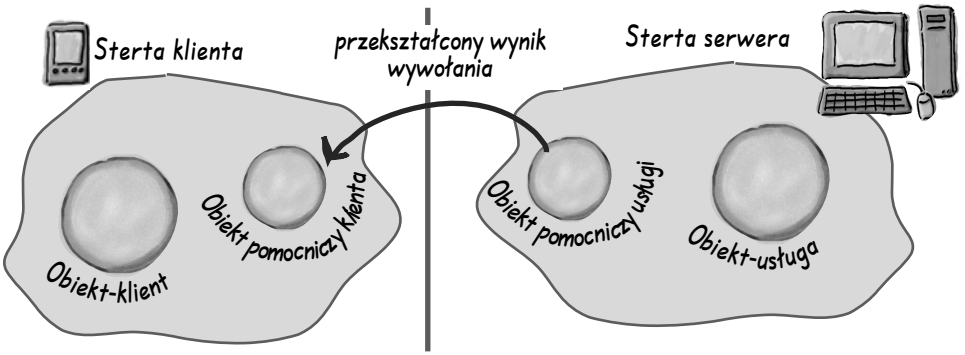
3. Obiekt pomocniczy usługi rozpakowuje informacje przekazane przez obiekt pomocniczy klienta, określa, którą metodę (i którego obiektu) wywołać, po czym wywołuje właściwą metodę właściwego obiektu usługi.



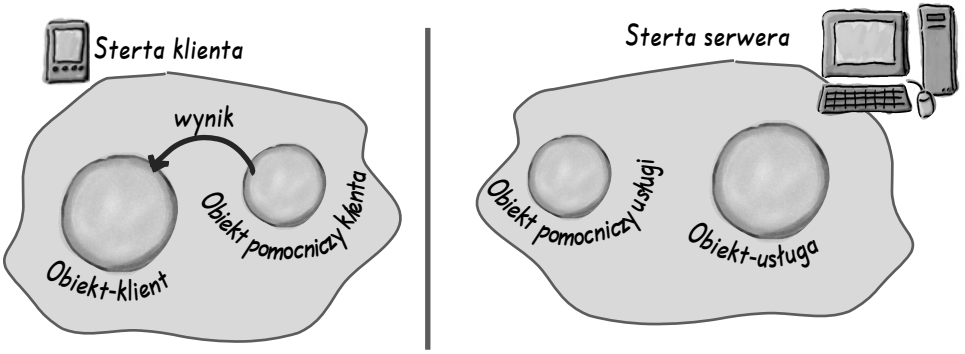
4 Wywołana metoda obiektu-usługi zwraca wynik obiektowi pomocniczemu usługi.



5 Obiekt pomocniczy usługi pakuje dane zwrócone przez wywołanie i przekazuje je przez sieć obiektowi pomocniczemu klienta.



6 Obiekt pomocniczy klienta rozpakowuje wynik i przekazuje go obiektowi-klientowi. Dla obiektu klienckiego cała operacja była przezroczysta.



Java RMI

Wiemy już mniej więcej, na czym polegają zdalne wywołania metod. Czas zapoznać się z mechanizmem Java RMI.

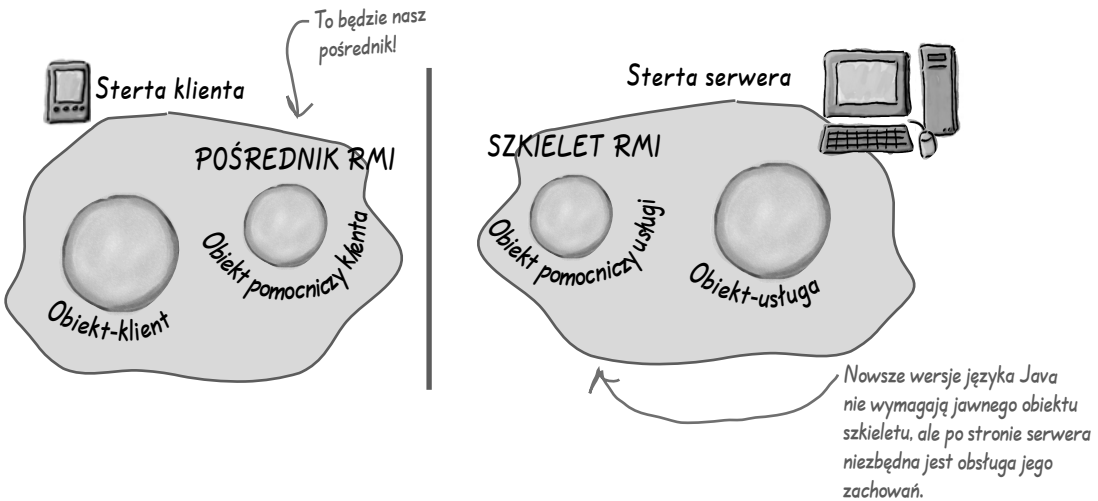
Java RMI zapewnia stworzenie obiektów pomocniczych klienta i usługi. W efekcie uzyskujemy obiekt pomocniczy klienta, który oferuje te same metody, co usługa zdalna. Zaletą RMI jest to, że programista może uniknąć samodzielnego pisania kodu współpracy sieciowej i procedur wejścia-wyjścia. Korzystając z gotowego obiektu pomocniczego, można wywoływać metody zdalne (metody właściwej usługi) tak samo, jak metody obiektów pracujących w lokalnej maszynie wirtualnej.

Java RMI zapewnia też całą infrastrukturę czasu wykonania, niezbędną do działania takiego mechanizmu, włączając w to usługę wyszukiwania, którą klient może wykorzystać do odnalezienia i uzyskania dostępu do zdalnych obiektów.

Istnieje jednak pewna różnica między wywołaniami RMI a wywołaniami lokalnymi (normalnymi). Mimo, że dla klienta wywołanie metody jest wywołaniem lokalnym, obiekt pomocniczy przesyła je przez sieć. Występuje więc komunikacja sieciowa i procedury wejścia-wyjścia. A co wiemy o komunikacji sieciowej i metodach wejścia-wyjścia?

Są ryzykowne! Mogą nie zostać wykonane! Jako takie, w wielu sytuacjach mogą zgłaszać wyjątki. Klient musi się z tym liczyć. Kilka stron dalej opiszemy szerzej, co to dokładnie oznacza.

Terminologia RMI: w mechanizmie RMI obiekt pomocniczy klienta to „pośrednik" (ang. *stub*, dosłownie „*pień*"), a obiekt pomocniczy usługi to „*szkielet*" (ang. *skeleton*).



Prześledźmy teraz czynności niezbędne do przekształcenia obiektu w usługę zdolną do przyjmowania wywołań zdalnych, jak również te, których wymaga klient, aby generować takie wywołania.

Zapnij pasy — przed nami długa i wyboista droga.



RMI — wycieczka z przewodnikiem

Budowanie usługi zdalnej

Poniżej przedstawiamy **krótki przegląd** pięciu kroków prowadzących do utworzenia usługi zdalnej. Są to czynności, które umożliwią zwykłemu obiektowi odbieranie wywołań przesyłanych przez zdalnego klienta. To właśnie zrobimy z naszą klasą AutomatSprzedający. Każdy z przedstawionych poniżej kroków omówimy na dalszych stronach bardziej szczegółowo.

Krok pierwszy:

Tworzenie ***interfejsu zdalnego***

Interfejs wywołań zdalnych definiuje metody, które klient może zdalnie wywoływać. Dla klienta będzie to klasa usługi. Zarówno właściwa usługa, jak i pośrednik muszą implementować ten typ!



MojaUsługa.java

Ten interfejs definiuje metody zdalne, które będą wywoływać klienty.

Krok drugi:

Budowanie ***implementacji zdalnej***

To jest klasa, która wykonuje właściwą pracę — czyli implementacja metod zdalnych zdefiniowanych jako interfejs zdalny. To właśnie te metody chciałby wywołać klient. Przykładem jest klasa AutomatSprzedający.



MojaUsługaImpl.java

Właściwa usługa, czyli klasa z metodami, które wykonują właściwą pracę. Implementacja interfejsu zdalnego.

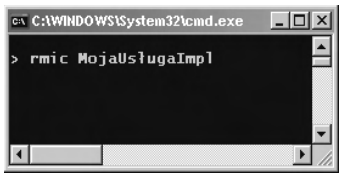
Na podstawie klasy implementującej właściwą usługę…

RMI tworzy dwie nowe klasy obiektów pomocniczych.

Krok trzeci:

Generowanie ***pośredników i szkieletów*** *przy użyciu rmic*

To są obiekty pomocnicze klienta i serwera. Nie musisz pisać tych klas samodzielnie ani nawet zaglądać do kodu, który je generuje. Niezbędne czynności wykonuje automatycznie narzędzie rmic dołączane do JDK.





MojaUsługaImpl_Stub.class

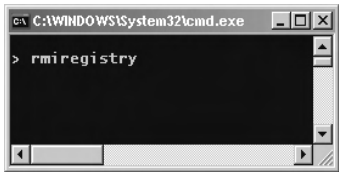


MojaUsługaImpl_Skel.class

Krok czwarty:

Uruchomienie ***rejestru RMI*** *(rmiregistry)*

Usługa *rmiregistry* pełni rolę katalogu. Tu klient szuka swojego pośrednika (obiektu pomocniczego, pośrednika klienta).

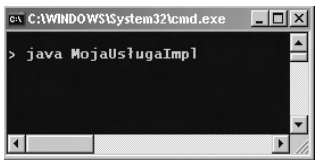


Uruchom w osobnym oknie terminala.

Krok piąty:

Uruchomienie ***usługi zdalnej***

Obiekt usługi musi rozpocząć pracę. Klasa implementacji tworzy egzemplarz usługi i rejestruje go w rejestrze RMI. Po zarejestrowania usługa jest dostępna dla klientów.



Krok pierwszy — tworzenie interfejsu zdalnego

① Rozszerzamy interfejs java.rmi.Remote.

Remote to interfejs określany jako „znakujący", czyli pozbawiony metod. Dla RMI ma on specjalne znaczenie, więc nie może zostać pominięty. Zwróćmy uwagę, że mówimy tu o *rozszerzeniu*, a nie o implementowaniu.

```
public interface Zdalny extends Remote {
```

To jest informacja o tym, że interfejs będzie wykorzystywany do obsługi wywołań zdalnych.

② Deklarujemy, że każda metoda interfejsu będzie generować ***wyjątek RemoteException***.

Klient używa interfejsu zdalnego jako typu usługi. Innymi słowy, wywołuje on metody obiektu, który implementuje interfejs zdalny. Jest to oczywiście na obiekt pomocniczy klienta. Ponieważ wykonuje on operacje sieciowe i wejścia-wyjścia, różne zdarzenia mogą zakłócić jego pracę. Klient musi być na nie przygotowany, zapewniając obsługę lub deklarację wyjątków wywołań zdalnych, czyli RemoteException. Przypomnijmy, że wprowadzenie deklaracji wyjątku w interfejsie wymusza wprowadzenie odpowiedniej procedury lub deklaracji w kodzie klienta.

```
import java.rmi.*;
```

Interfejs zdalny jest w java.rmi

```
public interface Zdalny extends Remote {
    public String powiedzHej() throws RemoteException;
}
```

Każde zdalne wywołanie metody to wywołanie obarczone ryzykiem. Deklarowanie wyjątku RemoteException w sygnaturze każdej metody zmusza klienta do uznania faktu, że wykonanie żądanej operacji może nie zakończyć się powodzeniem.

③ Dbamy o to, aby wszystkie argumenty i wartości zwracane przez metody były typami prostymi lub obiektami implementującymi interfejs Serializable

Argumenty i dane wyjściowe metody zdalnej muszą być wartościami prostymi lub obiektami, które można serializować. Gdy się chwilę nad tym zastanowisz, ten wymóg stanie się dość oczywisty. Każdy argument metody zdalnej musi zostać umieszczony w pakiecie danych i przekazany przez sieć. To wymaga mechanizmu serializacji. To samo można powiedzieć o wartościach wyjściowych. Gdy korzystamy z wartości prostych, ciągów znakowych i większości typów standardowego API (w tym tablic i kolekcji), nie ma problemu. Jeżeli korzystamy z własnych typów, musimy zadbać o to, aby implementowały one interfejs Serializable.

O interfejsie Serializable pisaliśmy szeroko w „Head First Java".

```
public String powiedzHej() throws RemoteException;
```

Ta wartość wyjściowa będzie przekazywana przez sieć do klienta, więc musi istnieć możliwość jej serializowania. Wymagane jest jej przekształcenie do postaci odpowiedniej do przesłania.



Krok drugi — budowanie implementacji interfejsu zdalnego

① Implementujemy nasze rozszerzenie interfejsu Remote.

Klasa usługi musi być implementacją zdalnego interfejsu — tego, w którym zdefiniowaliśmy wywoływane przez klienta metody.

```
public class ZdalnyImpl extends UnicastRemoteObject implements Zdalny {
    public String powiedzHej() {
        return "Serwer mówi 'Hej'";
    }
    // dalszy kod klasy
}
```

Kompilator sprawdzi, czy zaimplementowałeś wszystkie metody wskazanego interfejsu. W tym przypadku istnieje tylko jedna.

② Rozszerzamy klasę UnicastRemoteObject.

Aby pracować w charakterze usługi zdalnej, obiekt wymaga pewnych specyficznych funkcji. Najprościej jest skorzystać z możliwości rozszerzenia klasy UnicastRemoteObject dostępnej w pakiecie java.rmi.server. Wówczas możemy pozostawić całą pracę superklasie.

```
public class ZdalnyImpl extends UnicastRemoteObject implements Zdalny {
```

③ Piszemy bezargumentowy konstruktor, uwzględniając wyjątek RemoteException.

Nowa superklasa, UnicastRemoteObject, wprowadza drobny problem — jej konstruktor generuje wyjątek RemoteException. Jedynym sposobem poradzenia sobie z nim jest umieszczenie deklaracji konstruktora w implementacji klasy aplikacji. Zapewni to miejsce, aby zadeklarować generowany wyjątek. Nigdy nie wolno zapominać, że przy tworzeniu egzemplarza klasy zawsze wywoływany jest konstruktor jej superklasy. Jeżeli ten generuje wyjątek, nie ma innego wyjścia, niż uwzględnić ten wyjątek we własnym konstruktorze.

```
public ZdalnyImpl() throws Remote Exception { }
```

Konstruktor może pozostać pusty. Jest tutaj tylko po to, aby zadeklarować, że konstruktor superklasy generuje wyjątek.

④ Rejestrujemy usługę w rejestrze RMI.

Gdy usługa zdalna jest gotowa, musi zostać udostępniona klientom. W tym celu po utworzeniu egzemplarza usługi rejestrujemy ją w rejestrze RMI (który musiał zostać uruchomiony wcześniej, aby wykonanie przebiegło bez generowania wyjątków). W chwili zarejestrowania obiektu implementacji system RMI w rzeczywistości umieszcza w rejestrze odpowiedni *obiekt pośrednika klienta*, bo właśnie jego wymaga klient. Za rejestrację odpowiada statyczna metoda rebind() klasy java.rmi.Naming.

```
try {
    Zdalny usługa = new ZdalnyImpl();
    Naming.rebind("ZdalneHej", usługa);
} catch(Exception ex) {...}
```

Nadajemy usłudze nazwę (może z niej skorzystać klient, aby wyszukać usługę) i rejestrujemy ją. Polecenie wiązania powoduje w rzeczywistości zapisanie w rejestrze pośrednik usługi.

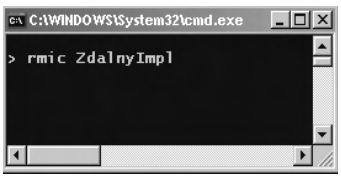
Krok trzeci — generujemy pośredników i szkielety

① Uruchamiamy rmic, wskazując klasę implementującą interfejs Remote (klasę, a nie interfejs rozszerzający).

Narzędzie rmic dołączane do pakietu JDK tworzy na podstawie implementacji usługi dwie nowe klasy: pośrednika i szkieletu. Ich nazwy zawierają nazwę implementacji i końcówkę, odpowiednio, _Stub lub _Skel. Podczas uruchamiania narzędzia można użyć też innych opcji, umożliwiających między innymi pominięcie generowania szkieletu, przeglądanie kodu źródłowego generowanych klas, a także oparcie ich pracy na protokole IIOP. Tutaj jednak opisujemy najbardziej typowy sposób postępowania. Klasy zostaną umieszczone w katalogu bieżącym (czyli tym, do którego dotarliśmy dzięki poleceniom `cd`). Nie wolno zapominać, że rmic musi znaleźć klasę implementacji. To skłania do uruchamiania narzędzia w katalogu zawierającym implementację usługi. (Pakietów nie używamy tu rozmyślnie, aby nie wprowadzać niepotrzebnych komplikacji. W praktyce zawsze niezbędne jest uwzględnienie struktury katalogów pakietów i stosowanie nazw kwalifikowanych).

Zwróćmy uwagę, że na końcu nie dodajemy „.class". Podajemy tu jedynie nazwę klasy.

RMIC generuje dwie nowe klasy obiektów pomocniczych.



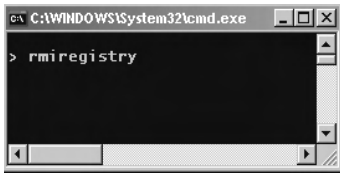

ZdalnyImpl_Stub.class


ZdalnyImpl_Skel.class

Krok czwarty — uruchamiamy rmiregistry

① Wywołujemy terminal i uruchamiamy usługę rmiregistry.

Rejestr musi zostać uruchomiony z katalogu, który zapewni mu dojście do udostępnianych klas. Najprościej będzie wywołać usługę z katalogu *classes*.



Krok piąty — uruchamiamy usługę

① Wywołujemy kolejny terminal i uruchamiamy właściwą usługę.

Może to zrobić metoda main() klasy implementacji usługi lub metoda osobnej klasy uruchamiającej. W tym prostym przykładzie umieściliśmy odpowiedni kod w klasie implementacji. Metoda main tworzy egzemplarz obiektu i zapisuje informację o nim w rejestrze RMI.



Kończymy kod strony serwera



Interfejs usługi zdalnej

Definicje wyjątku RemoteException i interfejsu Remote są zapisane w pakiecie java.rmi.

Interfejs usługi zdalnej musi być rozszerzeniem interfejsu java.rmi.Remote

```
import java.rmi.*;

public interface Zdalny extends Remote {

        public String powiedzHej() throws RemoteException;
}
```

Wszystkie metody zdalne muszą uwzględniać wyjątek RemoteException.

Implementacja usługi zdalnej

Klasę UnicastRemoteObject znajdziemy w pakiecie java.rmi.server.

Rozszerzenie klasy UnicastRemoteObject to najprostszy sposób utworzenia obiektu zdalnego.

```
import java.rmi.*;
import java.rmi.server.*;

public class ZdalnyImpl extends UnicastRemoteObject implements Zdalny {

    public String powiedzHej() {
        return "Serwer mówi 'Hej'";
    }


    public ZdalnyImpl() throws RemoteException { }

    public static void main (String[] args) {

        try {
            Zdalny usługa = new ZdalnyImpl();
            Naming.rebind("ZdalneHej", usługa);
        } catch(Exception ex) {
            ex.printStackTrace()'
        }
    }
}
```

Oczywiście, musisz zaimplementować wszystkie metody interfejsu. Zwróć uwagę, że tu nie pojawia się wyjątek RemoteException.

Nowa klasa MUSI być implementacją Twojego interfejsu zdalnego!

Konstruktor klasy UnicastRemoteObject zawiera deklarację wyjątku, więc MUSISZ napisać konstruktor podklasy, ponieważ ten będzie operował kodem zagrożonym wyjątkami (czyli superkonstruktorem).

Utwórz obiekt zdalny i odpowiedni wpis w rejestrze RMI, korzystając z metody statycznej Naming.rebind(). Podawana tu nazwa jest tą, z której będzie korzystał klient przy przeszukiwaniu rejestru.

Skąd klient bierze swój obiekt pomocniczy?

Klient musi dysponować pośrednikiem. To właśnie jego metody będzie wywoływał. Tu jest miejsce dla rejestru RMI. Klient przeprowadza „wyszukiwanie” — podaje nazwę i oczekuje odpowiedniego obiektu pomocniczego.

Spójrzmy na kod niezbędny do wyszukania i pobrania obiektu pomocniczego.

Tak to działa.



Bliskie spotkania z kodem aplikacji

Klient zawsze używa interfejsu zdalnego. Nigdy nie musi wiedzieć, jaka jest nazwa klasy usługi zdalnej.

lookup() to statyczna metoda klasy Naming.

Tu musi być nazwa, pod którą usługa została zarejestrowana.

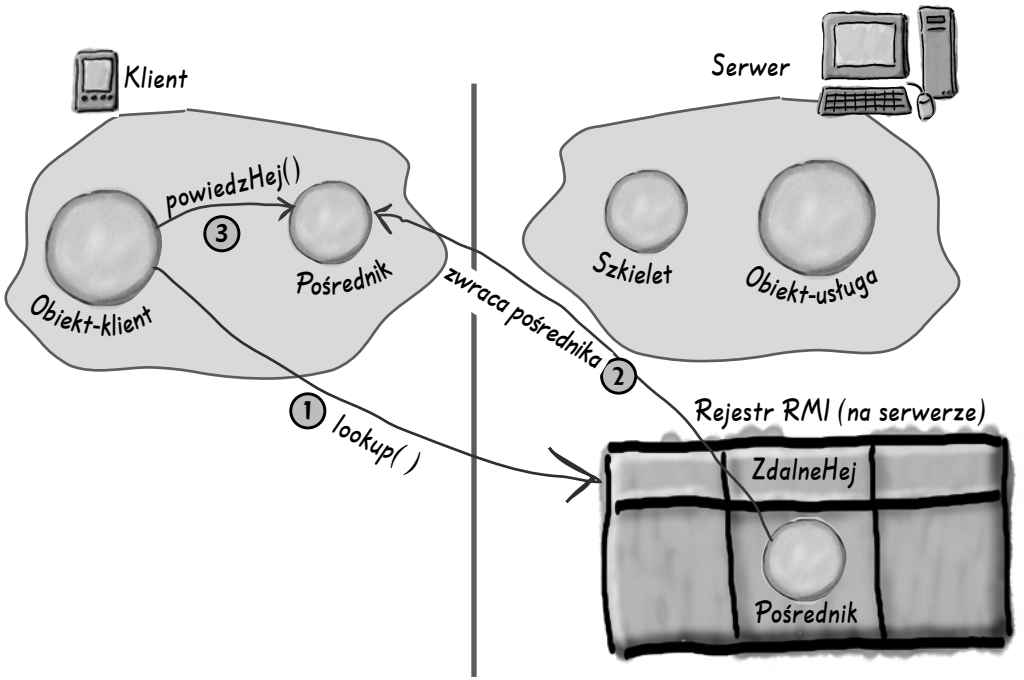
```
Zdalny usługa =
    (Zdalny) Naming.lookup("rmi://127.0.0.1/ZdalneHej");
```

Musisz zadeklarować typ interfejsu, ponieważ metoda wyszukująca zwraca klasę Object.

Nazwa stacji lub adres IP, gdzie usługa została uruchomiona.



RMI — wycieczka z przewodnikiem



Jak to działa...

1. Klient przeprowadza wyszukiwanie w rejestrze RMI.

```
Naming.lookup("rmi://127.0.0.1/ZdalneHej");
```

2. Rejestr RMI zwraca obiekt pomocniczy (pośrednika).
Jest to wartość zwracana przez metodę wyszukującą. RMI deserializuje pośrednika automatycznie. Aby to nastąpiło, klasa obiektu pomocniczego (wygenerowana za pomocą narzędzia rmic) musi być dostępna po stronie klienta.

3. Klient wywołuje metody pośrednika w taki sposób, jakby korzystał bezpośrednio z właściwej usługi.

Kompletny kod klienta

```
import java.rmi.*;

public class ZdalnyKlient {
    public static void main(String[] args) {
        new ZdalnyKlient().go();
    }

    public void go() {

        try {
            Zdalny usługa = (Zdalny) Naming.lookup("rmi://127.0.0.1/ZdalneHej");

            String s = usługa.zdalneHej();

            System.out.println(s);
        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

Klasa Naming (którą wykorzystamy do wyszukiwania w rejestrze RMI) znajduje się w pakiecie java.rmi.

Rejestr zwraca typ Object. Musimy określić typ.

Podajemy adres IP lub nazwę stacji.

i nazwę określoną przy rejestrowaniu usługi.

To wygląda jak zwyczajne wywołanie metody! (Choć pojawia się wyjątek — RemoteException).



Tylko dla wytrwałych

Skąd klient bierze plik klasy obiektu pomocniczego (pośrednika)?

To ciekawe pytanie. Klient musi dysponować klasą pośredniczącą (wygenerowaną wcześniej narzędziem rmic) w chwili wyszukiwania. Inaczej nie uda się etap deserializacji po stronie klienta i całość nie będzie działać. Klient potrzebuje też klas każdego serializowanego obiektu, który zwracają wywołania metod obiektu zdalnego. Tylko w najprostszych systemach można sobie pozwolić na przenoszenie tych klas ręcznie.

Istnieje znacznie wygodniejszy sposób, ale jego omówienie wykracza poza zakres tej książki. Jeżeli naprawdę Cię to interesuje, mechanizm taki nosi nazwę „dynamicznego pobierania klas" (ang. *dynamic class downloading*). Polega on na tym, że serializowane obiekty (takie jak obiekt pośrednika) są „znakowane" adresem URL, który wskazuje systemowi RMI klienta, gdzie znaleźć odpowiedni plik klasy. Wówczas, w momencie deserializowania obiektu, jeżeli RMI nie może znaleźć klasy w zasobach lokalnych, wykorzystuje ten adres URL do utworzenia wywołania HTTP Get i pobrania jej pliku. Pliki klas mogą być udostępniane przez prosty serwer WWW. Dodatkowo niezbędna jest zmiana pewnych parametrów zabezpieczeń klienta. Należałoby omówić jeszcze kilka aspektów tego mechanizmu, ale nie o tym ma być ta książka.

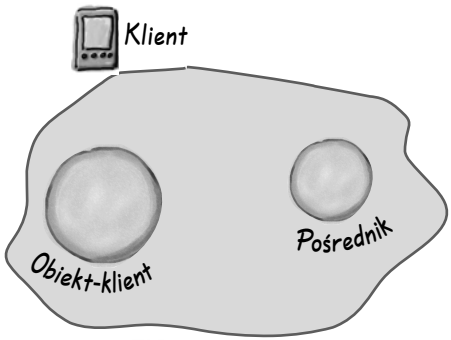
Jeśli chodzi o obiekt pośrednika, istnieje jeszcze inny sposób pozyskania klasy. Jest dostępny tylko w języku Java 5. Wrócimy do niego jeszcze na końcu rozdziału.

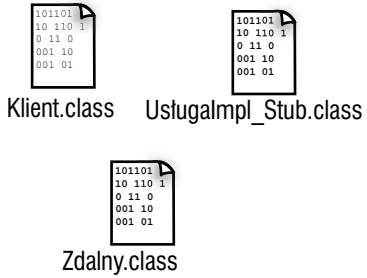


Uwaga!

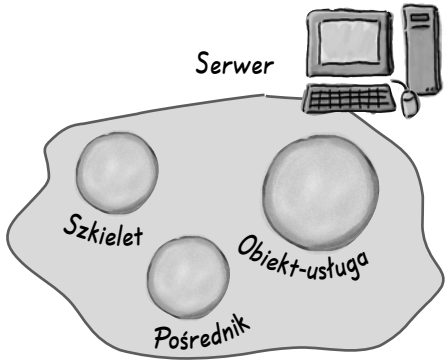
Trzy typowe błędy przy pracy z RMI to:

1) Zapominanie o uruchomieniu rejestru RMI przed uruchomieniem usługi (gdy wywołujemy metodę Naming.rebind(), aby zarejestrować usługę, proces rmiregistry musi być już uruchomiony).
2) Zapominanie o zapewnieniu serializacji argumentów i typów zwracanych przez metody (nie wykryje tego kompilator).
3) Zapominanie o skopiowaniu klasy pośrednika do klienta.





Nie zapominajmy, że klient wykorzystuje do wywoływania metod obiektu pomocniczego nasz interfejs zdalny. Maszyna wirtualna klienta musi dysponować klasą rzeczywistą mimo tego, że w kodzie klienta nie ma żadnych odwołań do niej. Klient zawsze odwołuje się do interfejsu zdalnego, tak jakby miał do czynienia bezpośrednio ze zdalnym obiektem usługi.



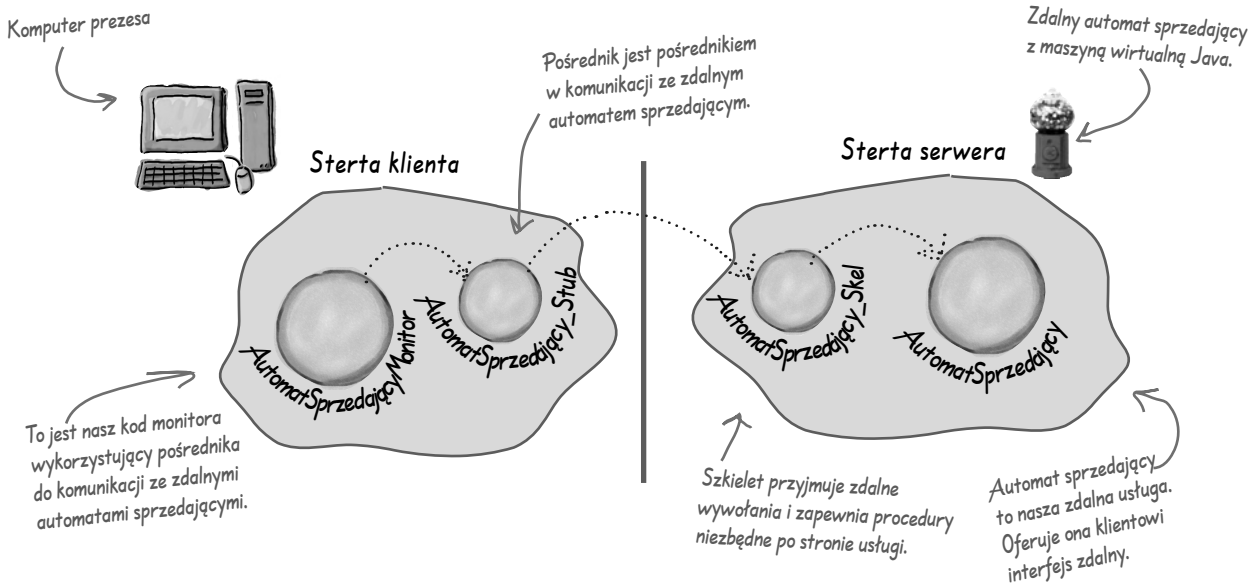


Serwer wymaga obu klas, _Stub i _Skel, oraz definicji interfejsu. Klasa pośrednika jest niezbędna, ponieważ to ona zajmuje miejsce właściwej usługi w momencie rejestracji w mechanizmie RMI.

Zdalny pośrednik automatu sprzedającego



Zakończymy teraz omówienie podstaw RMI i przejdziemy do implementowania zdalnego pośrednika automatu sprzedającego. Przyjrzyjmy się poniższemu rysunkowi, który ilustruje zastosowanie RMI w naszym przykładzie:



Przygotowanie klasy AutomatSprzedający do pracy w charakterze usługi zdalnej

Pierwszym krokiem do tego, aby nasz kod mógł korzystać ze zdalnego pośrednika, jest umożliwienie automatowi sprzedającemu obsługiwania zdalnych żądań klientów. Inaczej mówiąc, nasza klasa musi pracować jako usługa. Aby do tego doprowadzić, musimy:

1) Utworzyć zdalny interfejs obiektu AutomatSprzedający. Będzie on określał zbiór metod, które mogą być wywoływane zdalnie.
2) Upewnić się, że wszystkie typy wykorzystywane w interfejsie można serializować.
3) Zaimplementować interfejs.

Rozpoczniemy od interfejsu zdalnego:

Nie zapomnij zaimportować java.rmi.

To jest interfejs zdalny.

```
import java.rmi.*;

public interface AutomatSprzedającyZdalny extends Remote {
    public int pobierzLiczba() throws RemoteException;
    public String pobierzLokalizacja() throws RemoteException;
    public Stan pobierzStan() throws RemoteException;
}
```

Wszystkie używane typy muszą być typami prymitywnymi (prostymi) lub wartościami, które mogą być serializowane…

To są metody, których obsługę zapewnimy. Każda generuje wyjątek RemoteException.

Jeden z typów nie może być serializowany: klasa Stan. Zmieńmy to…

Klasa Serializable znajduje się w pakiecie java.io.

```
import java.io.*;

public interface Stan extends Serializable {
    public void włóżMonetę();
    public void zwróćMonetę();
    public void przekręćGałkę();
    public void wydaj();
}
```

Po prostu rozszerzamy interfejs Serializable, który nie ma żadnych metod. Teraz wszystkie obiekty stanu mogą być przesyłane przez sieć.

To nie koniec kwestii serializacji. Pozostaje jeszcze jeden problem z klasą Stan. Jak może pamiętasz, każdy obiekt stanu przechowuje odwołanie do automatu sprzedającego, aby wywoływać jego metody i zmieniać stan. Nie chcemy jednak, żeby cały obiekt AutomatSprzedający został zserializowany i przesłany przez sieć razem z obiektem Stan. Można tego uniknąć w prosty sposób:

```
public class StanNieMaMonety implements Stan {
    transient AutomatSprzedający automatSprzedający;

    // metody klasy
}
```

W każdej implementacji interfejsu Stan musimy uzupełnić deklarację zmiennej egzemplarza automatSprzedający słowem kluczowym transient. To zapewni, że maszyna wirtualna nie będzie serializować pola.

Klasa AutomatSprzedający jest już zaimplementowana, ale musimy sprawić, żeby mogła pracować w charakterze usługi i przyjmować żądania z sieci. Oznacza to przede wszystkim, że musi ona być implementacją interfejsu AutomatSprzedającyZdalny.

Jak przedstawiliśmy to wcześniej, w omówieniu RMI, jest to całkiem proste. Klasę uzupełniamy kilkoma standardowymi elementami…

Rozpoczynamy do importu pakietów rmi.

Klasa AutomatSprzedający będzie podklasą UnicastRemoteObject. To zapewni jej możliwość pracy w charakterze usługi zdalnej.

Klasa AutomatSprzedający implementuje interfejs zdalny…

```
import java.rmi.*;
import java.rmi.server.*;

public class AutomatSprzedający
        extends UnicastRemoteObject implements AutomatSprzedającyZdalny
{
    // zmienne egzemplarza

    public AutomatSprzedający(String lokalizacja, int liczbaGum) throws RemoteException {
        // kod konstruktora
    }

    public int pobierzLiczba() {
        return liczba;
    }

    public Stan pobierzStan() {
        return stan;
    }

    public String pobierzLokalizacja() {
        return lokalizacja;
    }

    // inne metody
}
```

…a konstruktor deklaruje wyjątek RemoteException (wymusza to superklasa).

To jest to! Tutaj nic się nie zmienia!

Wprowadzanie wpisu w rejestrze RMI

Usługa automatu sprzedającego jest już gotowa. Pozostaje już tylko jej uruchomienie, po którym zacznie się przyjmowanie nadchodzących żądań. Rozpoczynamy od rejestracji w rejestrze RMI.

Odpowiedni kod wprowadzimy w klasie testowej:

```
public class AutomatSprzedającyTest {

    public static void main(String[] args) {
        AutomatSprzedającyZdalny automatSprzedający = null;
        int liczba;

        if (args.length < 2) {
            System.out.println("AutomatSprzedający <nazwa> <towar>");
             System.exit(1);
        }

        try {
            liczba = Integer.parseInt(args[1]);

            automatSprzedający =
                new AutomatSprzedający(args[0], liczba);
            Naming.rebind("//" + args[0] + "/automatsprzedajacy", automatSprzedający);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Najpierw dodajemy blok try-catch tam, gdzie tworzymy egzemplarz automatu (bo konstruktor może zgłaszać wyjątki).

Dodajemy też wywołanie metody Naming.rebind(), która publikuje procedurę pomocniczą (pośrednika) automatu pod nazwą automatsprzedajacy.

Teraz przeprowadzimy nasz test:

To uruchamiamy najpierw.

To polecenie uruchomi rejestr RMI.



To polecenie uruchamiamy jako drugie.

Tak uruchamiamy AutomatSprzedający, rejestrując go jednocześnie w rejestrze RMI.

A teraz klient...

Pamiętasz jeszcze o klasie AutomatSprzedającyMonitor? Zamierzaliśmy wykorzystać ją do pracy w nowej, sieciowej konfiguracji bez wprowadzania zmian. Kilku drobnych modyfikacji nie możemy jednak uniknąć.

Musimy zaimportować pakiet RMI, bo będziemy przechwytywać wyjątek RemoteException...

```
import java.rmi.*;

public class AutomatSprzedającyMonitor {
    AutomatSprzedającyZdalny automat;

    public AutomatSprzedającyMonitor(AutomatSprzedającyZdalny automat) {
        this.automat = automat;
    }

    public void report() {
        try {
            System.out.println("Automat sprzedający: " + automat.pobierzLokalizacja());
            System.out.println("Towar: " + automat.pobierzLiczba() + " szt.");
            System.out.println("Bieżący stan urządzenia: " + automat.pobierzStan());
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
}
```

Teraz posługujemy się interfejsem, a nie klasą rzeczywistą.

Musimy zapewnić przechwytywanie wyjątku RemoteException, który może zostać wygenerowany w trakcie wywoływania metod korzystających z sieci.



... i test klienta

Mamy już wszystkie elementy układanki. Pozostaje powiązanie ich w całość, w kodzie, który pozwoli prezesowi kontrolować kilka przykładowych automatów:

To jest test monitora. Uruchomi go prezes!

```
import java.rmi.*;

public class AutomatSprzedającyMonitorTest {

  public static void main(String[] args) {
    String[] lokalizacja = {"rmi://gliwice.automatysprzedajace.com/automatsprzedajacy",
                            "rmi://wroclaw.automatysprzedajace.com/automatsprzedajacy",
                            "rmi://warszawa.automatysprzedajace.com/automatsprzedajacy"};

    AutomatSprzedającyMonitor[] monitor = new AutomatSprzedającyMonitor[lokalizacja.length];

    for (int i=0;i < lokalizacja.length; i++) {
         try {
             AutomatSprzedającyZdalny automat =
                  (AutomatSprzedającyZdalny) Naming.lookup(lokalizacja[i]);
             monitor[i] = new AutomatSprzedającyMonitor(automat);
             System.out.println(monitor[i]);
         } catch (Exception e) {
            e.printStackTrace();
         }
    }

      for(int i=0; i < monitor.length; i++) {
          monitor[i].raport();
      }
  }
}
```

To lista sprawdzanych lokalizacji.

Budujemy tablicę lokalizacji. W każdej jest jeden automat.

Budujemy też tablicę monitorów.

Teraz pobieramy pośredników poszczególnych automatów.

Ostatecznie iterujemy tablicę automatów, wyświetlając raport.



Bliskie spotkania z kodem aplikacji

To wywołanie zwraca pośrednika zdalnego automatu sprzedającego (lub generuje wyjątek, jeżeli nie można go znaleźć).

Pamiętajmy, że Naming.lookup() to statyczna metoda pakietu RMI, która pobiera lokalizację i nazwę usługi, aby wyszukać tę ostatnią w rejestrze RMI tej lokalizacji.

```
try {
        AutomatSprzedającyZdalny automat =
             (AutomatSprzedającyZdalny) Naming.lookup(lokalizacja[i]);

        monitor[i] = new AutomatSprzedającyMonitor(automat);
     System.out.println(monitor[i]);
} catch (Exception e) {
     e.printStackTrace();
}
```

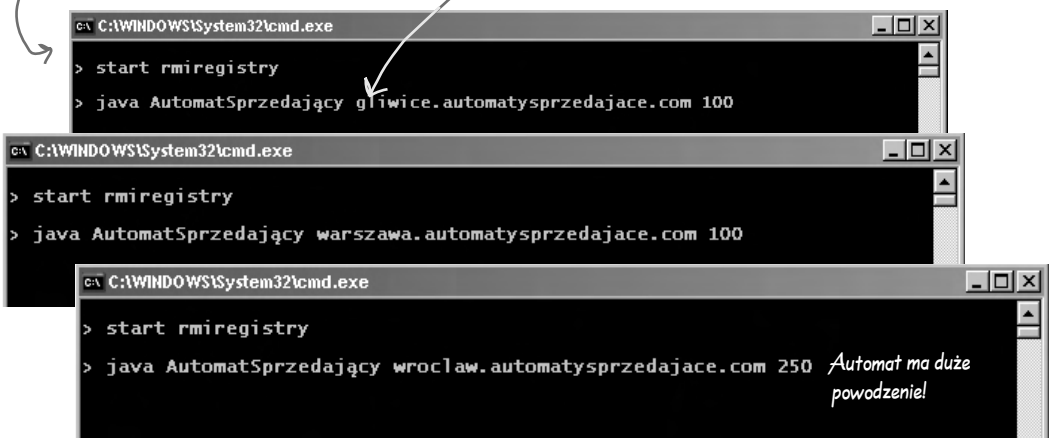
Dysponując pośrednikiem automatu zdalnego, tworzymy nowy monitor i wskazujemy mu ten „obiekt o interfejsie automatu zdalnego".

Kolejna demonstracja przed prezesem

Czas podsumować naszą pracę i przedstawić demo. Rozpocznijmy od uruchomienia kilku automatów z nowym kodem:

Na każdym komputerze uruchamiamy proces rejestru RMI, w tle lub w osobnym oknie terminala...

... następnie uruchamiamy automat sprzedający, określając przy tym nazwę lokalizacji i początkową liczbę gum.



A teraz przekazujemy monitor w ręce prezesa. Mamy nadzieję, że tym razem mu się spodoba

```
C:\WINDOWS\System32\cmd.exe

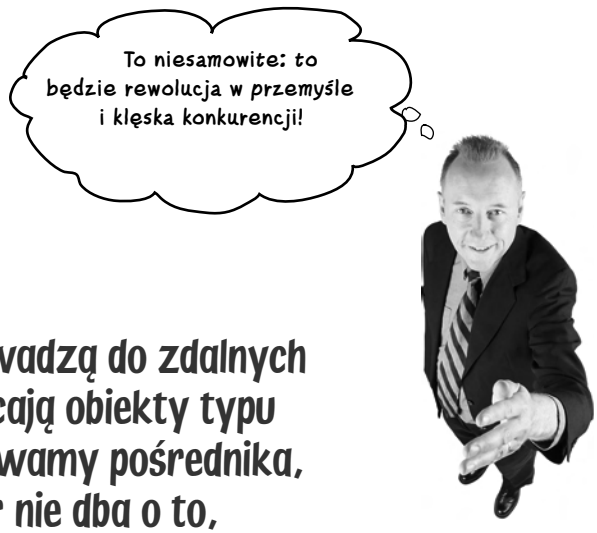
> java AutomatSprzedającyMonitorTest
Automat sprzedający: gliwice.automatysprzedajace.com
Towar: 99 szt.
Bieżący stan urządzenia: czekanie na monetę

Automat sprzedający: warszawa.automatysprzedajace.com
Towar: 44 szt.
Bieżący stan urządzenia: czekanie na przekręcenie gałki

Automat sprzedający: wroclaw.automatysprzedajace.com
Towar: 187 szt.
Bieżący stan urządzenia: czekanie na monetę

>
```

Monitor iteruje zbiór zdalnych automatów, wywołując metody pobierzLokalizacja(), pobierzLiczba() i pobierzStan().



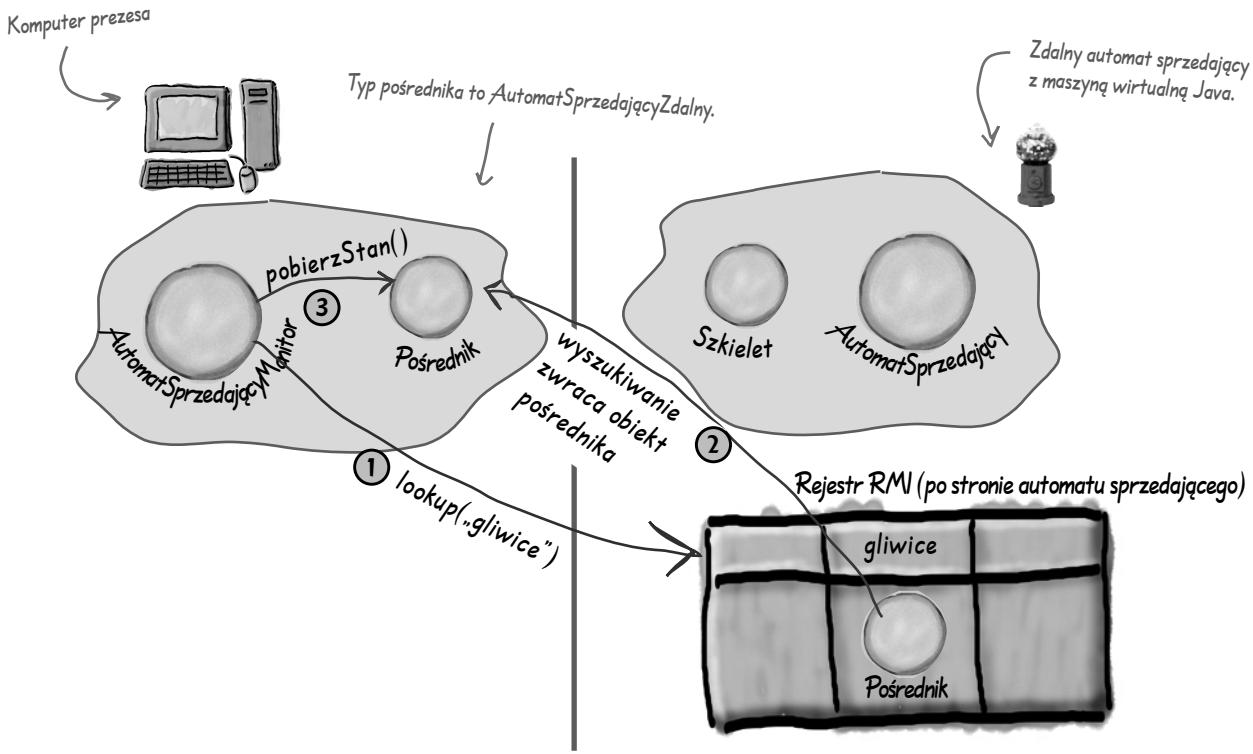
Wywołania metod pośrednika prowadzą do zdalnych wywołań sieciowych, które zwracają obiekty typu String, int lub Stan. Ponieważ używamy pośrednika, klasa AutomatSprzedającyMonitor nie dba o to, że wykorzystywany obiekt pracuje w lokalizacji zdalnej (choć musi zapewnić obsługę wyjątków związanych z komunikacją sieciową).



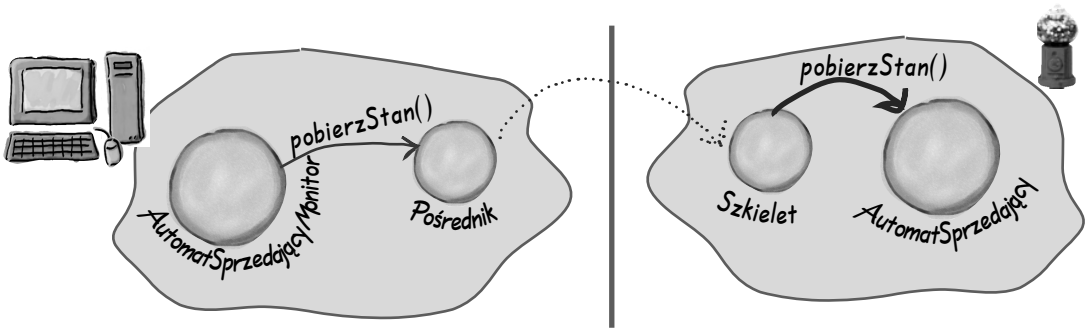


Za kulisami

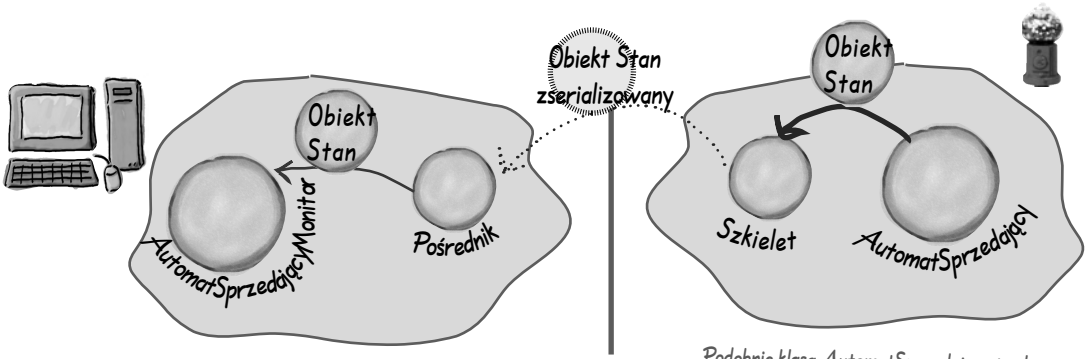
1. Prezes uruchamia monitor, który rozpoczyna pracę od pobierania pośredników zdalnych automatów sprzedających. Następnie wywołuje metodę pobierzStan() każdego z nich (jak również metody pobierzLiczba() i pobierzLokalizacja()).



❷ Wywołanie metody pobierzStan() pośrednika jest przekazywane do usługi zdalnej. Szkielet odbiera żądanie i przekazuje je automatowi sprzedającemu.



❸ Automat sprzedający zwraca obiekt stanu szkieletowi, który serializuje go i przesyła przez sieć pośrednikowi. Pośrednik przeprowadza deserializację i przekazuje obiekt monitorowi.



Monitor nie zmienił się wcale, poza tym, że jest przygotowany do obsługi wyjątków komunikacji sieciowej. Druga zmiana to zastąpienie odwołań do implementacji odwołaniami do interfejsu AutomatSprzedającyZdalny.

Podobnie klasa AutomatSprzedający implementuje nowy interfejs i zawiera deklarację wyjątku RemoteException w konstruktorze. Poza tym jej kod pozostał bez zmian.

Mamy też krótkie fragmenty kodu, które zapewniają rejestrowanie i zajmują się lokalizowaniem obiektów pomocniczych (pniaków) przy użyciu rejestru RMI. Każda aplikacja wykorzystująca internet wymaga pewnej usługi lokalizatora.

Definicja wzorca Proxy

Ten rozdział był już wystarczająco długi. Jak się okazuje, omówienie mechanizmu zdalnego pośrednika zajmuje nieco miejsca. Pomimo tego sama definicja i diagram klas wzorca nie są wyszukane. Zwróć uwagę, że nasz zdalny pośrednik to jedynie przykład implementacji ogólnego schematu pośrednika (ang. proxy). Można wyróżnić kilka typowych odmian wzorca Proxy. Zajmiemy się nimi kilka stron dalej. Teraz przyjrzyjmy się dokładniej jego podstawowej wersji.

Oto definicja wzorca Proxy:

> **Wzorzec Proxy** prowadzi do utworzenia obiektu pośredniczącego, który kontroluje dostęp do innego obiektu.

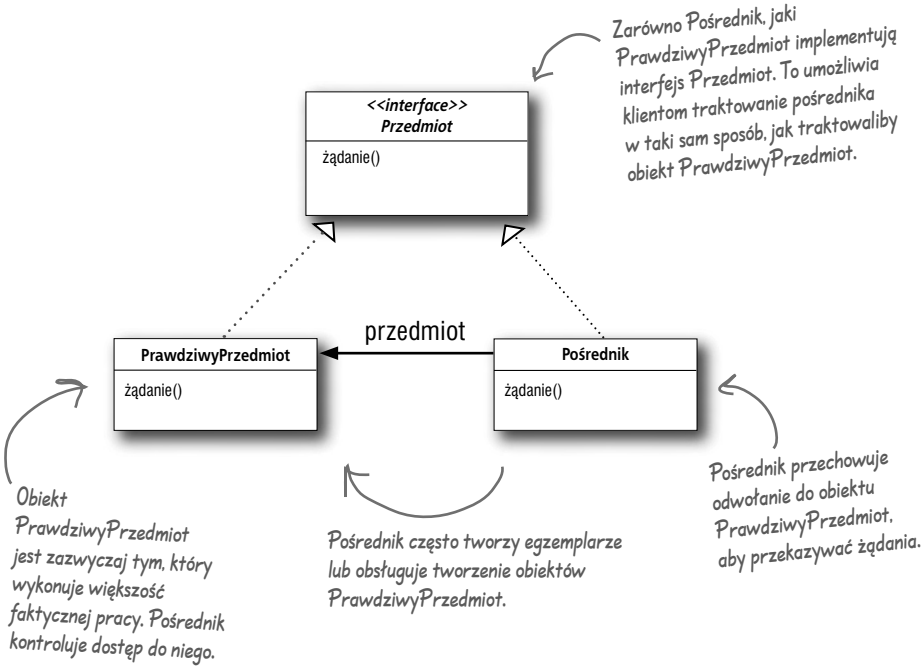
Wzorca Proxy używamy do stworzenia obiektu zastępczego, który kontroluje dostęp do innego obiektu, kiedy mamy do czynienia z takim obiektem zdalnym, którego stworzenie wiąże się z dużym kosztem lub który wymaga zabezpieczeń.

Mieliśmy już do czynienia z zastosowaniem pośrednika w charakterze obiektu zastępczego. Opisaliśmy też korzystanie z niego jako z „reprezentanta" innego obiektu.

A co z kontrolowaniem dostępu? Ta część definicji brzmi nieco obco. Bez obaw. Nawet w przypadku automatu sprzedającego nasz pośrednik kontroluje dostęp — sam klient, czyli monitor automatu, nie potrafi nawiązać komunikacji z obiektem zdalnym. Więc i tu w pewnym sensie zdalny pośrednik kontrolował dostęp, rozwiązując problem komunikacji sieciowej. Jak mówiliśmy, istnieje wiele odmian wzorca Proxy. Główne różnice między nimi to właśnie sposób „kontrolowania dostępu". Wrócimy jeszcze do tego tematu. Teraz zapoznamy się jedynie z kilkoma sposobami kontrolowania dostępu przez pośrednik:

- Zdalny pośrednik kontroluje dostęp do zdalnego obiektu.
- Wirtualny pośrednik kontroluje dostęp do zasobu, którego utworzenie jest kosztowne.
- Pośrednik chroniący kontroluje dostęp do zasobu w oparciu o prawa dostępu.

Po tym krótkim omówieniu ogólnego wzorca przejdźmy do diagramu klas…



Przyjrzyjmy się poszczególnym elementom diagramu…

Pierwszym jest interfejs Przedmiot, który definiuje metody obiektów PrawdziwyPrzedmiot i Pośrednik. Dzięki temu, że implementują one ten sam interfejs, obiekt Pośrednik może zastąpić PrawdziwyPrzedmiot w dowolnym miejscu.

PrawdziwyPrzedmiot to obiekt, który wykonuje właściwą pracę. To obiekt, który pośrednik reprezentuje i do którego kontroluje dostęp.

Obiekt Pośrednik przechowuje odwołanie do obiektu PrawdziwyPrzedmiot. W pewnych przypadkach klasa Pośrednik może odpowiadać za tworzenie i niszczenie obiektów PrawdziwyPrzedmiot. Klienty wchodzą w interakcje z obiektami PrawdziwyPrzedmiot tylko za pośrednictwem obiektów Pośrednik. Ponieważ zarówno klasa Pośrednik, jak i klasa PrawdziwyPrzedmiot są implementacjami interfejsu Przedmiot, pośrednik może zostać wykorzystany wszędzie tam, gdzie reprezentowany przezeń przedmiot. Pośrednik kontroluje też dostęp. Taka ochrona obiektu PrawdziwyPrzedmiot może być potrzebna, gdy obiekt PrawdziwyPrzedmiot pracuje na komputerze zdalnym, gdy tworzenie tego obiektu jest z pewnych przyczyn kosztowne lub gdy wymagane jest stosowane uprawnień dostępu.

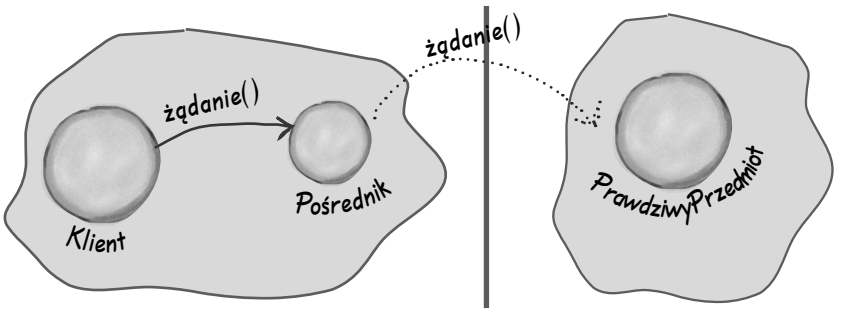
Gdy już zapoznaliśmy się z ogólną postacią wzorca, przejdźmy do innych typów pośrednika niż pośrednik zdalny…

Pośrednik wirtualny

Do tej pory zapoznaliśmy się z definicją wzorca Proxy i jednym z typowych przykładów jego zastosowania: *pośrednikiem zdalnym* (ang. *Remote Proxy*). Teraz przedstawimy inny rodzaj pośrednika, a mianowicie *pośrednik wirtualnego* (ang. *Virtual Proxy*). Jak się przekonamy, wzorzec Proxy ma wiele odmian. Wszystkie odpowiadają podobnemu schematowi, ale stosujemy je w różnych przypadkach. Spójrzmy na proste zestawienie schematów Virtual Proxy i Remote Proxy.

Remote Proxy — pośrednik zdalny

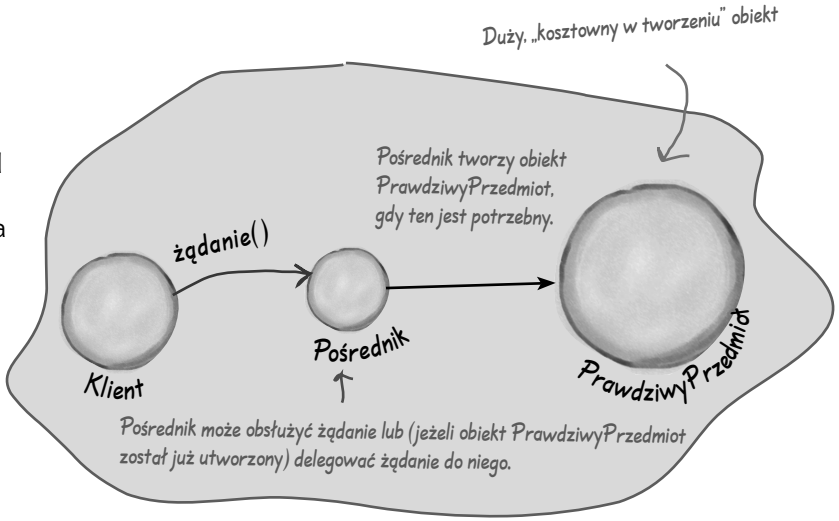
W schemacie Remote Proxy pośrednik jest lokalnym przedstawicielem obiektu pracującego w innej maszynie wirtualnej. Wywołanie metody pośrednika prowadzi do wywołania metody obiektu zdalnego. Wynik jest zwracany poprzez sieć i pośrednik do klienta.



Ten diagram znamy już dość dobrze…

Virtual Proxy — pośrednik wirtualny

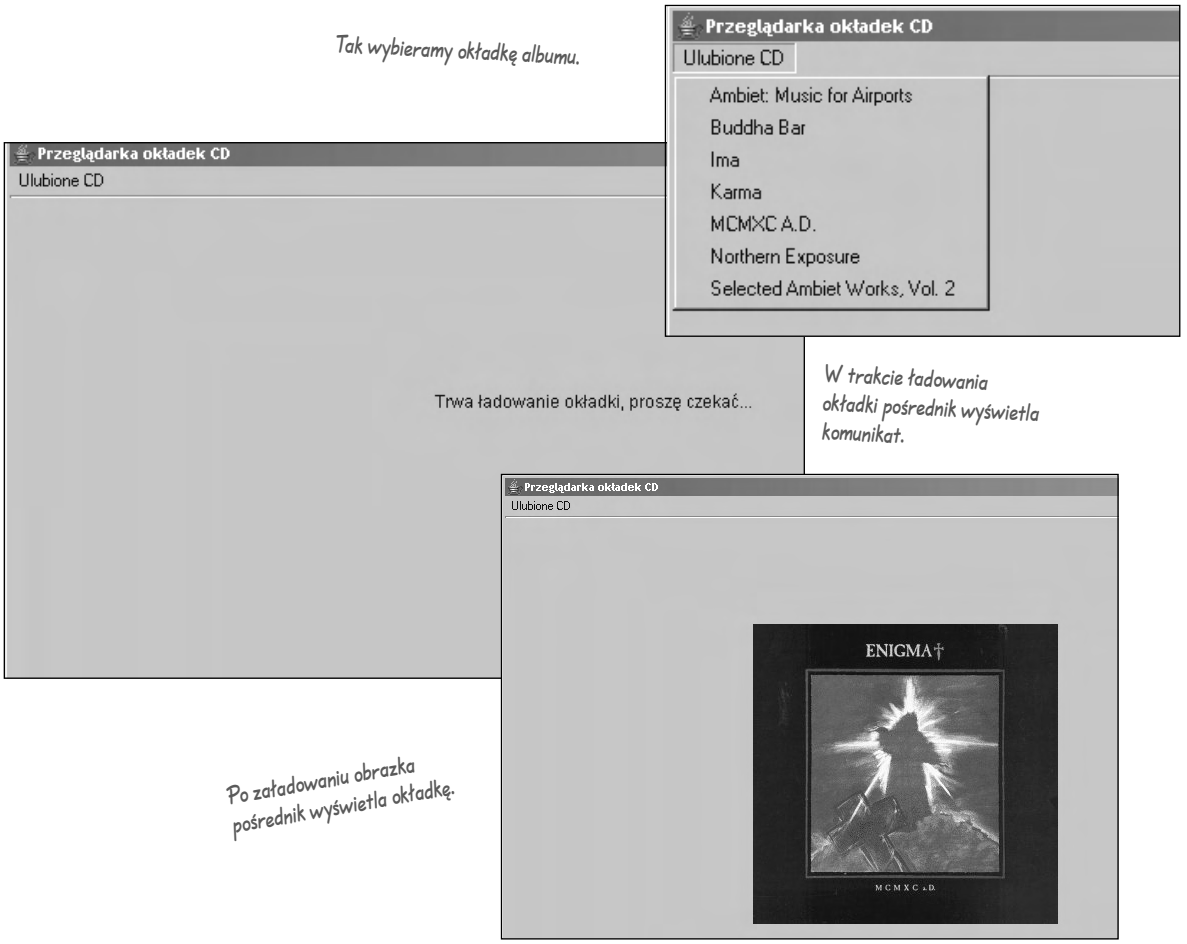
Pośrednik wirtualny to przedstawiciel obiektu, którego utworzenie jest w pewien sposób kosztowne. Pozwala to opóźnić utworzenie tego obiektu do momentu, gdy staje się on faktycznie niezbędny. Pośrednik zastępuje właściwy obiekt przed utworzeniem egzemplarza i w trakcie jego tworzenia. Później deleguje żądania bezpośrednio do obiektu PrawdziwyPrzedmiot.



Wyświetlanie okładek CD

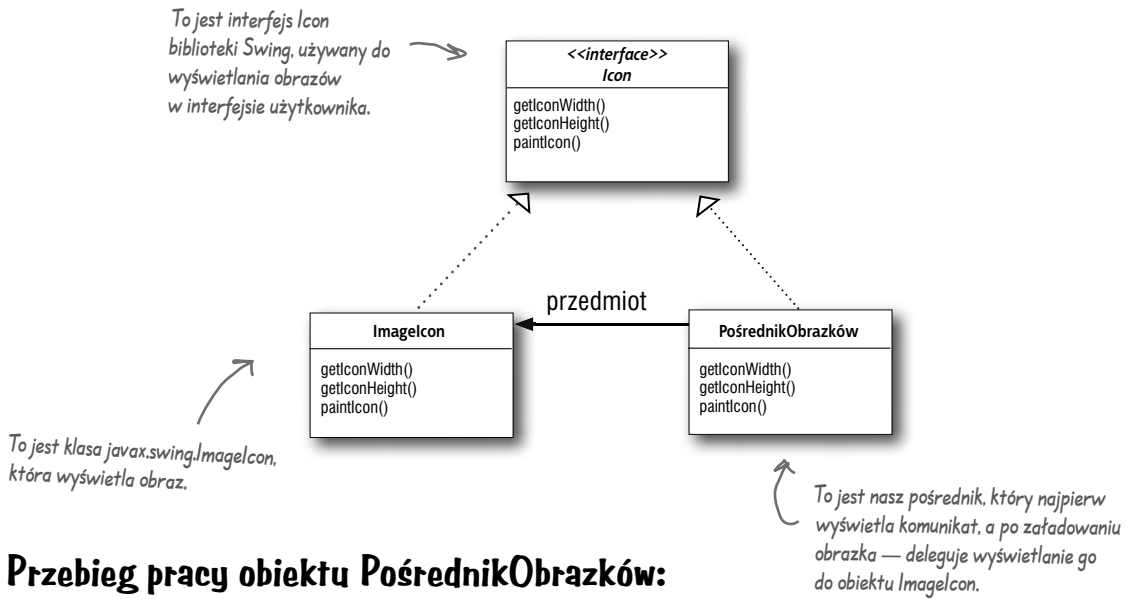
Wyobraźmy sobie, że piszemy aplikację, która wyświetla okładki dysków CD. Albumy będziemy wybierać z menu, a okładki pobierać z serwisu online, takiego jak Amazon.com. Jeżeli korzystamy z biblioteki Swing, możemy utworzyć obiekt typu Icon i nakazać mu ładowanie obrazu z sieci. Jedynym problemem jest to, że — w zależności od obciążenia sieci i jakości połączenia z internetem — pobieranie okładek może zająć pewną ilość czasu. Aplikacja powinna wyświetlać wówczas informację o tym, że obiekt jest w trakcie ładowania. Nie chcemy też blokować pracy całej aplikacji w czasie, gdy ta pobiera obrazek. Po jego załadowaniu informacja powinna zniknąć, aby jej miejsce mogła zająć wybrana okładka.

Wirtualny pośrednik pozwala na bardzo prostą implementację takiego mechanizmu. Będzie on reprezentował obiekt Icon, zarządzał realizowanym w tle ładowaniem i, zanim obrazek zostanie w pełni pobrany, będzie wyświetlał tekst: „Trwa ładowanie okładki, proszę czekać…”. Po załadowaniu obrazka pośrednik deleguje wyświetlanie do obiektu Icon.



Projektowanie wirtualnego pośrednika do wyświetlania okładek

Zanim rozpoczniemy pisanie kodu do naszego przykładu, spójrzmy na diagram klas. Jak widać poniżej, nie różni się on od diagramu klas wzorca Remote Proxy. Jednak tutaj rolą pośrednika jest ukrycie obiektu, którego utworzenie jest kosztowne (ponieważ pobieramy dane obiektu Icon z sieci). Ukrywany obiekt nie jest jednak takim, który został utworzony w innej maszynie wirtualnej.



Przebieg pracy obiektu PośrednikObrazków:

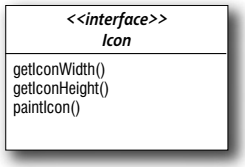
❶ **Obiekt PośrednikObrazków tworzy obiekt ImageIcon i rozpoczyna ładowanie danych na podstawie adresu URL.**

❷ **W trakcie, gdy dane z sieci są pobierane, obiekt PośrednikObrazków wyświetla komunikat: „Trwa ładowanie okładki, proszę czekać...".**

❸ **Po załadowaniu obrazka obiekt PośrednikObrazków deleguje wszystkie wywołania metod do obiektu ImageIcon. Będą to przede wszystkim wywołania paintIcon(), getWidth() i getHeight().**

❹ **Jeżeli użytkownik wydaje polecenie wyświetlenia nowego obrazu, tworzymy nowy obiekt pośrednika i cały proces rozpoczyna się od nowa.**

Kod klasy PośrednikObrazków



Klasa PośrednikObrazków jest implementacją interfejsu Icon.

Klasa ImageIcon to PRAWDZIWY obrazek, który zostanie wyświetlony (po załadowaniu).

Adres URL obrazu przekazujemy w konstruktorze. To jest właśnie to, co wyświetlimy, gdy tylko zostanie pobrane z sieci!

Do czasu załadowania obrazu zwracamy standardowe wartości szerokości i wysokości. Gdy jest on już załadowany, korzystamy z delegacji do obiektu ImageIcon.

Ta część kodu jest najbardziej interesująca. Wyświetla ona obrazek na ekranie (korzystając z delegacji do obiektu ImageIcon). Jeżeli obiekt ImageIcon jeszcze nie istnieje, tworzy go. Ten fragment będziemy jeszcze analizować na następnej stronie.

```
class PośrednikObrazków implements Icon {
    ImageIcon imageIcon;
    URL obrazekURL;
    Thread pobieranieThread;
    boolean pobieranie = false;

    public PośrednikObrazków(URL url) { obrazekURL = url; }

    public int getIconWidth() {
        if (imageIcon != null) {
            return imageIcon.getIconWidth();
        } else {
            return 800;
        }
    }

    public int getIconHeight() {
        if (imageIcon != null) {
            return imageIcon.getIconHeight();
        } else {
            return 600;
        }
    }

    public void paintIcon(final Component c, Graphics  g, int x,  int y) {
        if (imageIcon != null) {
            imageIcon.paintIcon(c, g, x, y);
        } else {
            g.drawString("Trwa ładowanie okładki, proszę czekać...", x+300, y+190);
            if (!pobieranie) {
                pobieranie = true;

                pobieranieThread = new Thread(new Runnable() {
                    public void run() {
                        try {
                            imageIcon = new ImageIcon(obrazekURL, "Okładka CD");
                            c.repaint();
                        } catch (Exception e) {
                            e.printStackTrace();
                        }
                    }
                });
                pobieranieThread.start();
            }
        }
    }
}
```



Bliskie spotkania z kodem aplikacji

Ta metoda jest wywoływana, aby wyświetlić obrazek na ekranie.

```
public void paintIcon(final Component c, Graphics  g, int x,  int y) {
    if (imageIcon != null) {
        imageIcon.paintIcon(c, g, x, y);
    } else {
        g.drawString("Trwa ładowanie okładki, proszę czekać...", x+300, y+190);
        if (!pobieranie) {
            pobieranie = true;

            pobieranieThread = new Thread(new Runnable() {
                public void run() {
                    try {
                        imageIcon = new ImageIcon(obrazekURL, "Okładka CD");
                        c.repaint();
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
            });
            pobieranieThread.start();
        }
    }
}
```

Jeżeli mamy już obrazek, nakazujemy mu, aby wyświetlił się na ekranie.

W przeciwnym wypadku wyświetlamy komunikat: „proszę czekać".

Tutaj ładujemy PRAWDZIWY obrazek. Zwróćmy uwagę, że ładowanie obrazka przy użyciu klasy ImageIcon jest synchroniczne — konstruktor nie zakończy pracy do czasu jego załadowania. Nie można połączyć tego z wyświetlaniem komunikatu, więc budujemy mechanizm asynchroniczny. Spójrzmy na „Bardzo bliskie spotkania z kodem" na następnej stronie…



Jeszcze bliższe spotkania z kodem aplikacji

Jeżeli pobieranie obrazu nie zostało jeszcze rozpoczęte...

... rozpoczynamy proces (jeżeli zastanawiasz się nad bezpieczeństwem pracy wielowątkowej, zwróć uwagę, że tylko jeden wątek wywołuje metodę wyświetlającą).

Nie chcemy wstrzymywać pracy całego interfejsu użytkownika, więc do pobrania obrazka wykorzystujemy osobny wątek.

```
if (!pobieranie) {
    pobieranie = true;

    pobieranieThread = new Thread(new Runnable() {
        public void run() {
            try {
                imageIcon = new ImageIcon(obrazekURL, "Okładka CD");
                c.repaint();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    });
    pobieranieThread.start();
}
```

W nowym wątku tworzymy obiekt ImageIcon. Jego konstruktor nie zakończy pracy do czasu pełnego załadowania danych.

Gdy już mamy obrazek, nakazujemy bibliotece Swing powtórzenie jego wyświetlenia.

Przy kolejnym wyświetlaniu, już po utworzeniu egzemplarza obiektu ImageIcon, metoda paintIcon spowoduje wyłącznie wyświetlenie obrazka i pominie komunikat ładowania.

Łamigłówka projektowa

Klasa PośrednikObrazków ma dwa stany zaimplementowane przy użyciu instrukcji warunkowych. Czy masz pomysł na inny wzorzec, który mógłby uporządkować ten kod? Jak można zmodyfikować klasę?

```
class PośrednikObrazków implements Icon {
    // zmienne egzemplarza i konstruktor

    public int getIconWidth() {
        if (imageIcon != null) {
            return imageIcon.getIconWidth();
        } else {
            return 800;
        }
    }

    public int getIconHeight() {
        if (imageIcon != null) {
            return imageIcon.getIconHeight();
        } else {
            return 600;
        }
    }

    public void paintIcon(final Component c, Graphics  g, int x,  int y) {
        if (imageIcon != null) {
            imageIcon.paintIcon(c, g, x, y);
        } else {
            g.drawString("Trwa ładowanie okładki, proszę czekać...", x+300, y+190);
            // dalszy kod
        }
    }
}
```

Dwa stany

Dwa stany

Dwa stany

Testowanie przeglądarki okładek



Kod gotowy do użycia

Czas przeprowadzić testy nowego pośrednika. „Za kulisami" zbudowaliśmy kod klasy PośrednikObrazkówTest, która tworzy okno i ramkę, instaluje menu i inicjalizuje pośrednik. Całego kodu nie będziemy omawiać tu szczegółowo. Jest on dostępny, podobnie jak pozostałe kody z książki. Przedstawiamy go też w całości na końcu rozdziału.

Oto najważniejsze elementy:

```
public class PośrednikObrazkówTest {
    ImageComponent imageComponent;
    public static void main (String[] args) throws Exception {
        PośrednikObrazkówTest test = new PośrednikObrazkówTest();
    }

    public PośrednikObrazkówTest() throws Exception{

        // przygotowanie ramki i menu

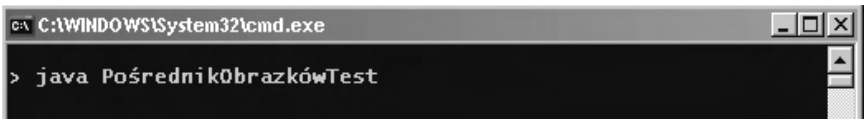
        Icon icon = new PośrednikObrazków(startURL);
        imageComponent = new ImageComponent(icon);
        frame.getContentPane().add(imageComponent);
    }
}
```

Tutaj tworzymy obiekt PośrednikObrazków i wskazujemy początkowy adres URL. Wybieranie nowych pozycji menu albumów CD prowadzi do utworzenia nowych obiektów PośrednikObrazków.

Następnie tworzymy osłonę pośrednika, którą możemy dodać do ramki. Klasa ImageComponent zadba przede wszystkim o wymiary obrazu.

Na końcu dodajemy pośrednik do ramki, aby mógł zostać wyświetlony.

I uruchamiamy test:



Uruchomienie powyższej klasy powinno doprowadzić do utworzenia takiego okna.



Warto spróbować...

1. **Użyć menu do ładowania różnych okładek CD; zwróćmy uwagę na wyświetlanie komunikatu o ładowaniu do czasu pobrania obrazka.**

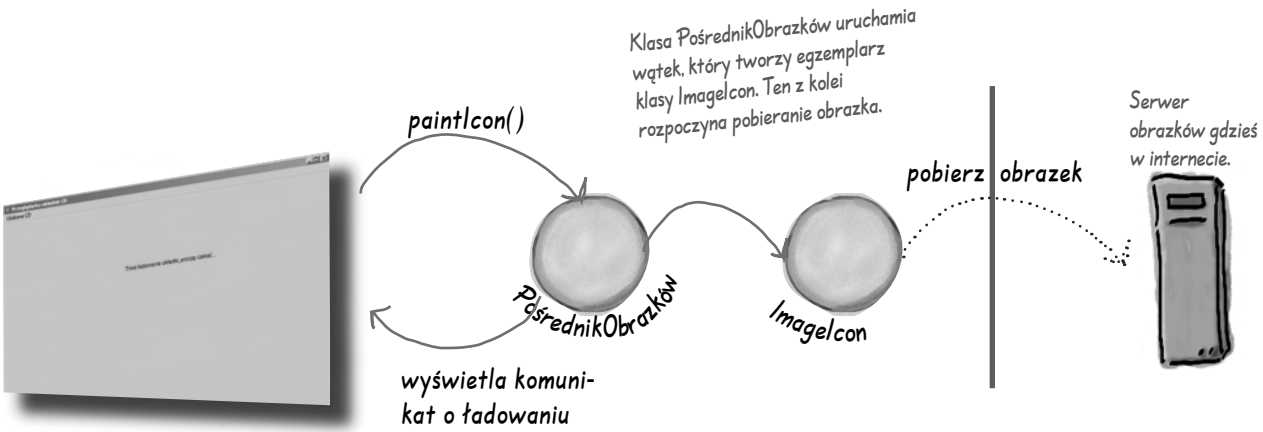
2. **Zmienić rozmiar okna w trakcie wyświetlania komunikatu o ładowaniu. Zwróćmy uwagę, że pośrednik obsługuje ładowanie bez blokowania okna.**

3. **Dodać do klasy testowej własne okładki.**

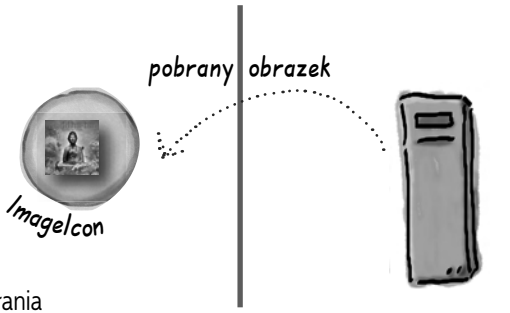
Co zrobiliśmy?



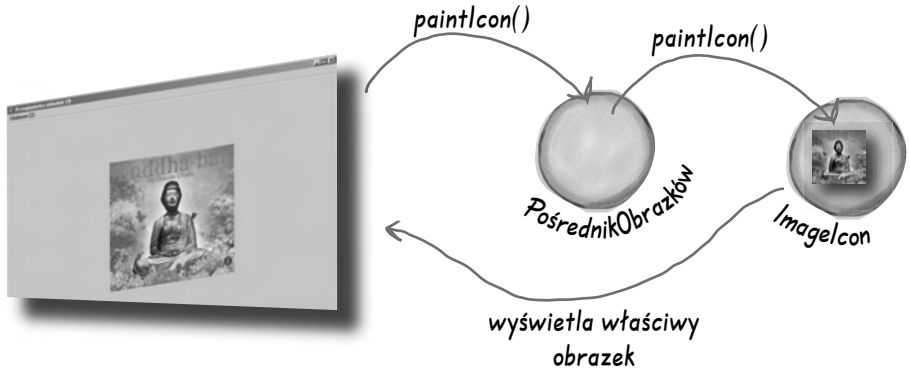
1. Utworzyliśmy klasę PośrednikObrazków obsługującą wyświetlanie pobranych obrazów. Wywołanie metody paintIcon() powoduje utworzenie nowego wątku, który pobiera obrazek i tworzy obiekt ImageIcon.



2. W pewnym momencie obrazek zostaje zwrócony, a obiekt ImageIcon kończy inicjalizację.



3. Po zakończeniu tworzenia egzemplarza obiektu ImageIcon kolejne wywołania metody paintIcon() są delegowane przez pośrednik do klasy ImageIcon.



Nie istnieją głupie pytania

P: Schematy Remote Proxy i Virtual Proxy wydają się dosyć odmienne; czy to naprawdę TEN SAM wzorzec?

O: W praktyce spotkać można bardzo wiele odmian wzorca Proxy. Ich cechą wspólną jest przechwytywanie generowanych przez klienta wywołań metod przedmiotu. Wprowadzenie takiego pośrednictwa pozwala zrealizować wiele różnych funkcji, między innymi zarządzanie żądaniami kierowanymi do przedmiotu zdalnego, zastępowanie obiektu, którego tworzenie jest kosztowne lub, jak się jeszcze przekonamy, zapewnianie pewnego poziomu ochrony i decydowanie o uprawnieniach klientów do wywoływania różnych metod. Przykłady przedstawione dotąd to dopiero początek. Ogólny wzorzec Proxy można wykorzystywać na różnorakie sposoby. Część potencjalnych możliwości opiszemy na końcu rozdziału.

P: Klasa PośrednikObrazków przypomina mi wzorzec Dekorator. Zasadniczo wygląda to tak, że osłaniamy jeden obiekt innym, po czym delegujemy wywołania do klasy ImageIcon. Na czym polega różnica?

O: Wzorce Proxy i Dekorator wyglądają czasem bardzo podobnie, ale ich cele są różne: Dekorator wzbogaca klasę o nowe zachowania, podczas gdy Proxy jedynie zarządza dostępem do niej. Można spytać: „Czy komunikat ładowania nie jest nowym zachowaniem?". W pewien sposób jest. Ważniejsze jest jednak to, że obiekt PośrednikObrazków kontroluje dostęp do obiektu ImageIcon. Na czym ta kontrola polega? Można to traktować jako przykład separowania klienta od klasy ImageIcon. Utrzymanie powiązania między nimi prowadziłoby do sytuacji, w której klient przed wyświetleniem całego interfejsu użytkownika musi czekać na pobranie odpowiedniego obrazka. Pośrednik kontroluje jednak dostęp do obiektu ImageIcon, zapewniając inną reprezentację ekranową do czasu zakończenia inicjalizacji. Gdy ta zostanie zakończona, pośrednik ogranicza swoją rolę do delegowania.

P: Jak mogę przygotować klienta do korzystania z klasy Pośrednik zamiast z klasy PrawdziwyPrzedmiot?

Odp.: Dobre pytanie. Jedną z typowych technik jest utworzenie fabryk, która tworzy egzemplarz i zwraca wykorzystywany przedmiot. Ponieważ odpowiednie czynności są realizowane przez metodę fabryki, przed zwróceniem przedmiotu można utworzyć jego osłonę w postaci pośrednika. Klient nie musi wiedzieć o tym, że korzysta z pośrednika, a nie z właściwego obiektu.

P: Zauważyłem, że w przykładzie z klasą PośrednikObrazków do wyświetlania obrazka zawsze wykorzystywany jest nowy obiekt ImageIcon. Dzieje się tak nawet wtedy, gdy obrazek został wcześniej pobrany. Czy można zaimplementować mechanizm podobny do tego, który buforuje wcześniejsze operacje pobierania danych z sieci?

O: Mowa tu o wyspecjalizowanej odmianie pośrednika wirtualnego, określanej nazwą *pośrednik buforujący* (ang. *Caching Proxy*). Pośrednik buforujący utrzymuje bufor utworzonych wcześniej obiektów i zapewnia wykorzystanie ich do obsługi żądań.

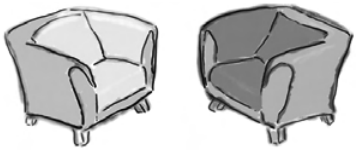
Różnymi odmianami wzorca Proxy zajmiemy się szerzej na końcu rozdziału.

P: Zwróciliśmy uwagę na podobieństwo wzorców Dekorator i Proxy. A co z wzorcem Adapter? Ten też wydaje się podobny.

O: Zarówno Pośrednik, jak i Adapter znajdują się „z przodu" innych obiektów i przekazują do nich żądania. Pamiętajmy jednak, że wzorzec Adapter służy do zmieniania interfejsu adaptowanych obiektów, podczas gdy Proxy polega na implementowaniu tego samego.

Jest jeszcze jedno podobieństwo, związane z *pośrednikiem chroniącym* (ang. *Protection Proxy*). Ten rodzaj pośrednika może zezwalać na dostęp do poszczególnych metod obiektu w oparciu o rolę klienta lub odmawiać tego dostępu. W takim przypadku pośrednik udostępnia klientowi niepełny interfejs. Takie rozwiązanie jest podobne do pewnych odmian adapterów. Schematem Protection Proxy zajmiemy się jeszcze szerzej w dalszej części rozdziału.

Pogawędki przy kominku



Dzisiaj: ***wzorce Proxy i Dekorator walczą o swoje***

Proxy

Witaj, kolego. Podejrzewam, że jesteś tutaj dlatego, że wielu ludzi nas ze sobą myli?

Ja kopiuję *Twoje* pomysły? Daj spokój. Ja kontroluję dostęp do obiektów. Ty je tylko upiększasz. Moje zadanie jest znacznie ważniejsze. Na tyle ważniejsze, że Twoje zarzuty wcale nie są śmieszne.

OK, może to nie jest do końca kwestia dekoracji… ale wciąż nie rozumiem, dlaczego uważasz, że kopiuję Twoje pomysły. Moim zadaniem nie jest wzbogacanie obiektów, ale reprezentowanie ich.

Wydaje mi się, że nie rozumiesz. Ja jestem pełnomocnikiem Przedmiotów. Nie jest tak, że po prostu uzupełniam je o dodatkowe zachowania. Klienty wykorzystują mnie jako surogat prawdziwego przedmiotu. Mogę chronić taki przedmiot przed niepożądanymi operacjami dostępu. Mogę zabezpieczać interfejs użytkownika przez wstrzymywaniem pracy w trakcie, gdy trwa ładowanie dużego obiektu. Mogę ukryć fakt, że Przedmiot pracuje na innym komputerze. Rzekłbym, że to zupełnie inne funkcje niż Twoje!

Dekorator

Hmm, myślę, że ludzie mylą nas z tego powodu, że przedstawiasz się im jako zupełnie inny wzorzec, podczas gdy w rzeczywistości jesteś po prostu wzorcem Dekorator w przebraniu. Wydaje mi się, że nie powinieneś tak kopiować moich pomysłów.

„Tylko” upiększam? Myślisz, że jestem tylko od nieistotnych dekoracji? Wytłumaczę Ci, kolego: ja dodaję *zachowania*. To jest najważniejsze w obiektach — to, co *robią*!

Możesz nazywać to „reprezentowaniem”, ale to wygląda jak kaczka i chodzi jak kaczka… tyle powiem. Spójrz choćby na schemat Virtual Proxy. To po prostu jeszcze jedna metoda dodawania zachowania — zachowania, które umożliwia wykonywanie pewnych czynności, gdy pewien duży, kosztowny obiekt jest w trakcie ładowania. Z kolei Remote Proxy to metoda komunikacji ze zdalnymi obiektami, której celem jest zwolnienie klientów od zajmowania się pewnymi operacjami. To wszystko kwestia zachowań.

Nazwij to, jak chcesz. Ja implementuję ten sam interfejs, co osłaniane obiekty. Ty też.

Proxy

Zwróć uwagę na to sformułowanie. Ty osłaniasz obiekt. Choć czasami mówi się, że pośrednik osłania Przedmiot, nie jest to określenie ścisłe.

Pomyśl o zdalnym pośredniku… jaki obiekt osłaniam? Obiekt, który reprezentuję i do którego kontroluję dostęp, jest na innym komputerze! Zobaczymy, jak Ty to zrobisz!

Jasne, wirtualny pośrednik… wróć do przykładu z przeglądarką okładek CD. Gdy klient po raz pierwszy używa mnie w charakterze pośrednika, przedmiot wywołań nawet nie istnieje! Co w tym przypadku osłaniam?

Nigdy bym się po Tobie nie spodziewał tylu problemów. Oczywiście, że czasami tworzę obiekty! Jak myślisz, skąd wirtualny pośrednik bierze swój przedmiot? Właśnie wskazałeś ogromną różnicę między nami: obaj wiemy, że Decorator zajmuje się ozdabianiem okien. Nigdy nie tworzy nowych egzemplarzy!

Hej, po tej rozmowie jestem już przekonany, że jesteś tylko uproszczonym pośrednikiem!

Bardzo rzadko zdarza się, żeby pośrednik osłaniał swój przedmiot więcej niż jeden raz. W gruncie rzeczy, jeżeli osłaniasz coś dziesięć razy, powinieneś raczej zastanowić się nad swoim projektem.

Dekorator

Ach, tak? A dlaczego nie?

W porządku, ale przecież wiemy, że pośrednik zdalny to nie jest typowy przykład. Masz jakiś inny? Wątpię.

Aha, a teraz jeszcze powiesz, że Twoim właściwym zadaniem jest tworzenie obiektów.

Tak? Utwórz egzemplarz `this`!

Uproszczonym pośrednikiem? Chciałbym zobaczyć, jak rekurencyjnie osłaniasz obiekt dziesięcioma obiektami Decorator i dajesz radę!

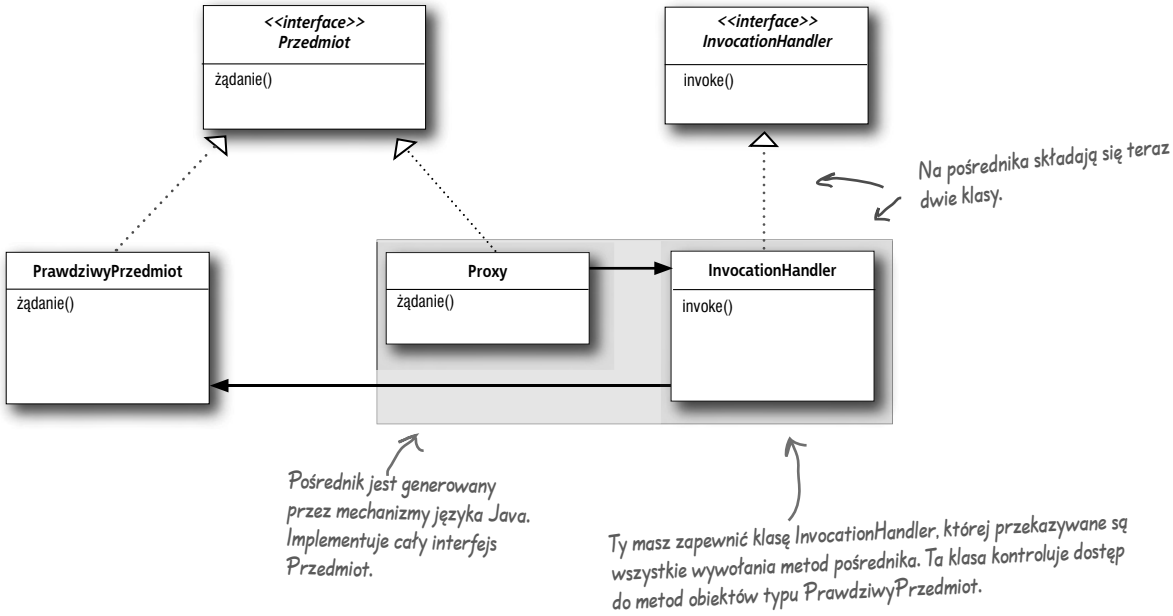
Spójrz na siebie, który tylko zastępujesz obiekty wykonujące rzeczywistą pracę. Wiesz, przykro mi z Tobą rozmawiać.

Wykorzystanie mechanizmów Java API do stworzenia pośrednika chroniącego



Język Java został wyposażony w standardową obsługę mechanizmu pośrednika. Niezbędne klasy znajdziemy w pakiecie java.lang.reflect. Korzystając z niego, możemy tworzyć klasy pośredniczące *w locie*. Będą one implementować jeden lub więcej interfejsów i przekazywać wywołania metod do wskazanej klasy. Ponieważ właściwa klasa pośrednika jest tworzona w czasie wykonywania programu, rozwiązanie to określa się terminem *pośrednik dynamiczny* (ang. *dynamic proxy*).

Mechanizm pośrednika dynamicznego wykorzystamy do utworzenia kolejnej implementacji wzorca Proxy, tym razem w wersji Protection Proxy (pośrednik chroniący). Rozpocznijmy od analizy diagramu klas. Podobnie jak niemal zawsze w świecie rzeczywistym, i tutaj spotkamy się z drobnymi odstępstwami od klasycznej definicji wzorca:



Ponieważ język Java tworzy klasę Proxy *za Ciebie*, potrzebny jest sposób, aby określić, co klasa pośrednika ma robić. Odpowiedniego kodu nie można umieścić w klasie Proxy, tak jak robiliśmy to wcześniej, ponieważ nie implementujemy takiej klasy bezpośrednio. A jeżeli nie można w klasie pośrednika, to gdzie? W implementacji interfejsu InvocationHandler. Zadaniem klasy InvocationHandler jest reagowanie na wszelkie wywołania metod kierowane do pośrednika. Obiekt InvocationHandler można więc traktować jako ten, do którego pośrednik przekazuje wykonywanie właściwej pracy.

Przyjrzyjmy się teraz czynnościom niezbędnym do zaimplementowania takiego schematu…

Swatanie w Obiektowie



Każde miasto potrzebuje serwisu towarzyskiego, prawda? Podjąłeś wyzwanie i zaimplementowałeś serwis randkowy w Obiektowie. Próbowałeś też być nowoczesny i wprowadziłeś funkcję „Ładny-Brzydki" — umożliwiającą uczestnikom wzajemne ocenianie się. Masz nadzieję, że to utrzyma klientów w napięciu i zachęci do przeglądania ofert. Oczywiście, jest też przy tym więcej zabawy.

Osią usługi jest obiekt OsobaKomponent umożliwiający wprowadzanie i pobieranie informacji o osobach

To jest interfejs. Implementacją zajmiemy się za chwilę...

Metody do pobierania informacji o imieniu, płci, zainteresowaniach i ocenie urody (1–10).

```
public interface OsobaKomponent {

    String pobierzImię();
    String pobierzPłeć();
    String pobierzZainteresowania();
    int pobierzRankingUrody();

    void ustawImię(String nazwa);
    void ustawPłeć(String płeć);
    void ustawZainteresowania(String zainteresowania);
    void ustawRankingUrody(int ranking);

}
```

Dzięki tym metodom możemy zapisywać informacje o osobach.

Ustaw RankingUrody pobiera wartość całkowitą i dodaje ją do średniej dla danej osoby.

Zobaczmy teraz implementację…

Implementacja interfejsu OsobaKomponent

Klasa OsobaBeanImpl jest implementacją interfejsu OsobaKomponent.

```
public class OsobaKomponentImpl implements OsobaKomponent {
    String nazwa;
    String płeć;
    String zainteresowania;
    int ranking;
    int rankingLiczba = 0;

    public String pobierzImię() {
        return nazwa;
    }

    public String pobierzPłeć() {
        return płeć;
    }

    public String pobierzZainteresowania() {
        return zainteresowania;
    }

    public int pobierzRankingUrody() {
        if (rankingLiczba == 0) return 0;
        return (ranking/rankingLiczba);
    }


    public void ustawImię(String nazwa) {
        this.nazwa = nazwa;
    }

    public void ustawPłeć(String płeć) {
        this.płeć = płeć;
    }

    public void ustawZainteresowania(String zainteresowania) {
        this.zainteresowania = zainteresowania;
    }

    public void ustawRankingUrody(int ranking) {
        this.ranking += ranking;
        rankingLiczba++;
    }
}
```

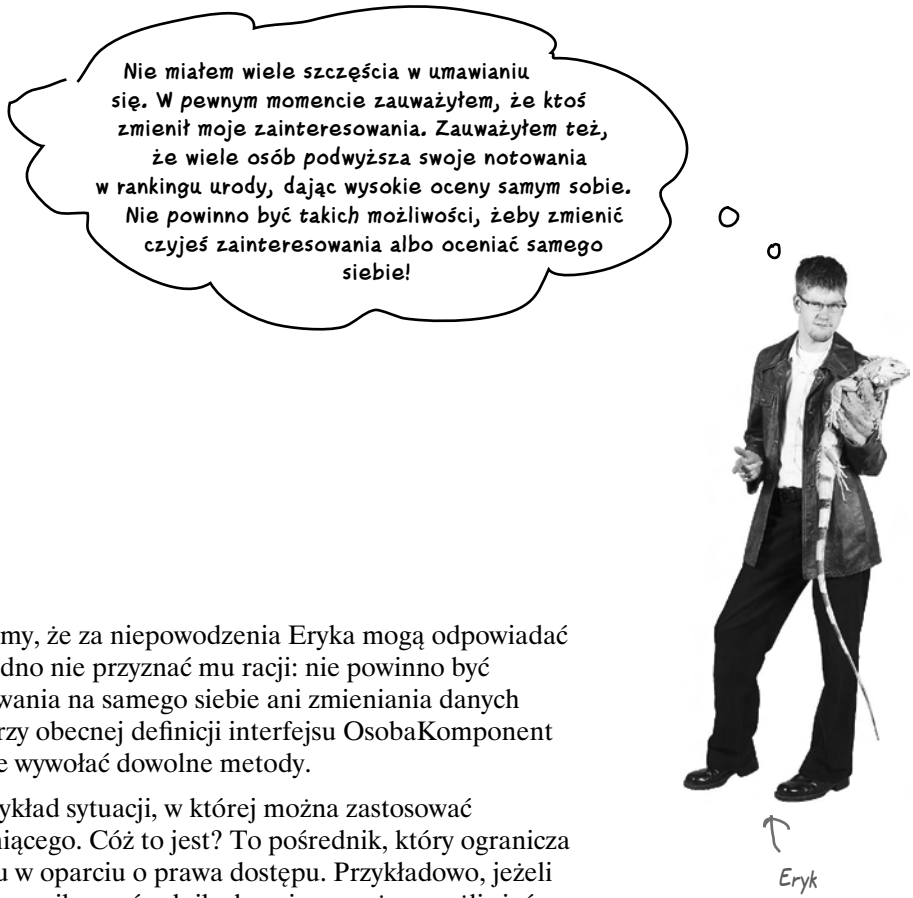
Zmienne egzemplarza.

Metody typu getter, zwracające zmienne egzemplarza...

... wyjątkiem jest pobierzRankingUrody(), która oblicza średnią ocen, dzieląc ich sumę przez liczbę.

A to metody typu setter, które ustawiają wartości zmiennych egzemplarza.

Metoda ustawRankingUrody() zwiększa wartość rankingLiczba o 1 i dodaje nową wartość do sumy, która jest podstawą średniej.



Choć podejrzewamy, że za niepowodzenia Eryka mogą odpowiadać inne czynniki, trudno nie przyznać mu racji: nie powinno być możliwości głosowania na samego siebie ani zmieniania danych innego klienta. Przy obecnej definicji interfejsu OsobaKomponent każdy klient może wywołać dowolne metody.

To doskonały przykład sytuacji, w której można zastosować pośrednika chroniącego. Cóż to jest? To pośrednik, który ogranicza dostęp do obiektu w oparciu o prawa dostępu. Przykładowo, jeżeli mamy obiekt pracownika, pośrednik chroniący może umożliwiać temu pracownikowi wywoływanie pewnego wąskiego zbioru metod. Jego kierownik będzie miał większe możliwości (będzie mógł dodatkowo korzystać z metod takich, jak ustawPensja()), a pracownik działu kadr — dostęp do każdej metody obiektu.

W naszym biurze pośrednictwa chodzi o to, aby klient mógł określać własne dane i aby nie mogli tego robić inni. Dokładnie odwrotnie powinno być z ocenami: inni mogą je wystawiać, a sam zainteresowany — nie. Obiekt OsobaKomponent ma też kilka metod typu getter, ale ponieważ żadna z nich nie ujawnia informacji poufnych, powinny być one powszechnie dostępne.



Teatrzyk — ochrona przedmiotów

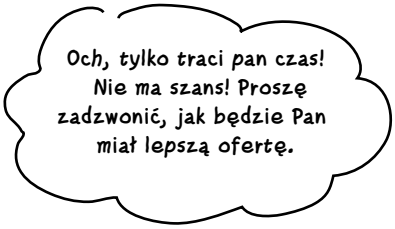
Boom na internet odszedł już chyba w przeszłość. W tamtych czasach znalezienie ciekawszej i lepiej płatnej pracy wymagało często tylko krótkiej wizyty w kawiarence internetowej. Wtedy rozwinęła się nawet osobna profesja agentów poszukujących programistów…

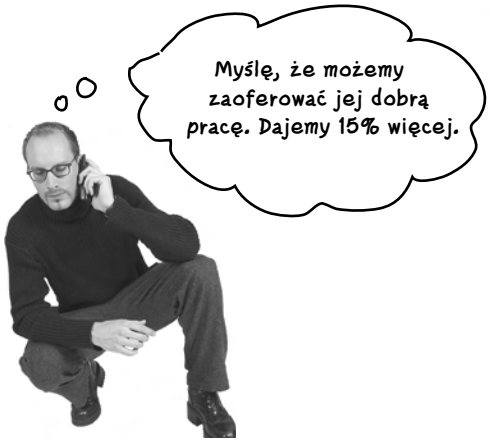






Podobnie jak pośrednik chroniący, agent chroni dostęp do swojego klienta, dopuszczając tylko wybrane formy kontaktu…





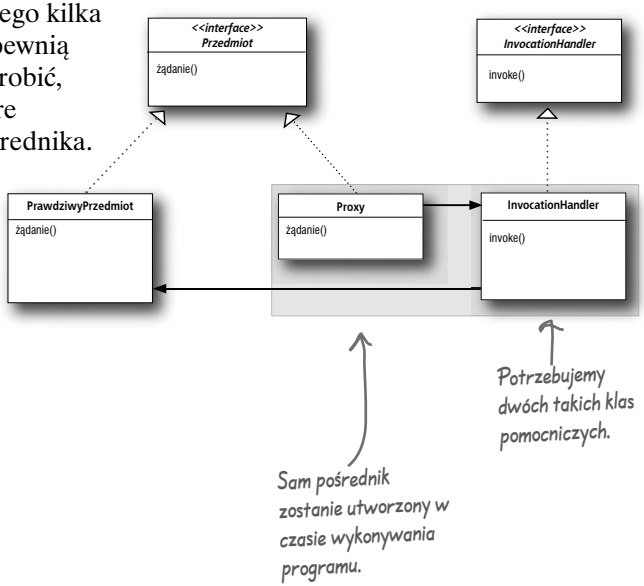


Z lotu ptaka — budowanie dynamicznego pośrednika klasy OsobaKomponent

Mamy do rozwiązania dwa problemy: klienci nie powinni mieć możliwości zmieniania wartości własnego rankingu; nie powinni też zmieniać danych innych klientów. Aby to zapewnić, utworzymy dwa pośredniki: pierwszy będzie mediatorem dostępu do własnego obiektu OsobaKomponent, a drugi — do obiektów OsobaKomponent innych klientów. W ten sposób pośredniki mogą kontrolować, jakie żądania będą realizowane w każdej sytuacji.

Aby utworzyć obiekty pośredników, skorzystamy z mechanizmu dynamicznego obiekty pośredników Java API, opisanego kilka stron wcześniej. Standardowe mechanizmy języka zapewnią utworzenie obu pośredników. Wszystko, co musimy zrobić, to zapewnić procedury obsługi decydujące o tym, które operacje zostają wykonane po wywołaniu metody pośrednika.

Ten diagram widzieliśmy już kilka stron wcześniej…



Krok pierwszy:

Tworzymy dwa obiekty IncovationHandler

Klasa InvocationHandler implementuje zachowania pośrednika. Jak się przekonamy, Java zadba o utworzenie właściwej klasy i obiektu pośrednika. My zapewniamy tylko klasę pomocniczą, która realizuje odpowiednie zachowania.

Krok drugi:

Piszemy kod zapewniający utworzenie dynamicznych pośredników.

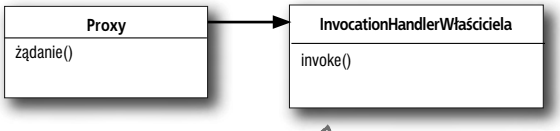
Aby klasa pośrednika została wygenerowana i aby można było utworzyć jej egzemplarz, niezbędnych będzie kilka dodatkowych wierszy kodu. Za chwilę zobaczymy, jak to będzie wyglądać.

Krok trzeci:

Osłaniamy obiekt OsobaKomponent odpowiednim pośrednikiem.

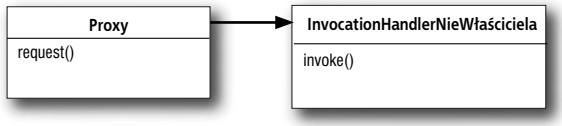
Gdy pojawia się potrzeba użycia obiektu OsobaKomponent, mamy do czynienia z jednym z dwóch przypadków. W pierwszym jest to obiekt klienta („właściciela”). W drugim jest to obiekt innego użytkownika serwisu.

W każdym z przypadków tworzymy odpowiedni pośrednik klasy OsobaKomponent.



Gdy klient próbuje uzyskać dostęp do własnego obiektu OsobaKomponent.

Gdy klient próbuje uzyskać dostęp do cudzego obiektu OsobaKomponent.

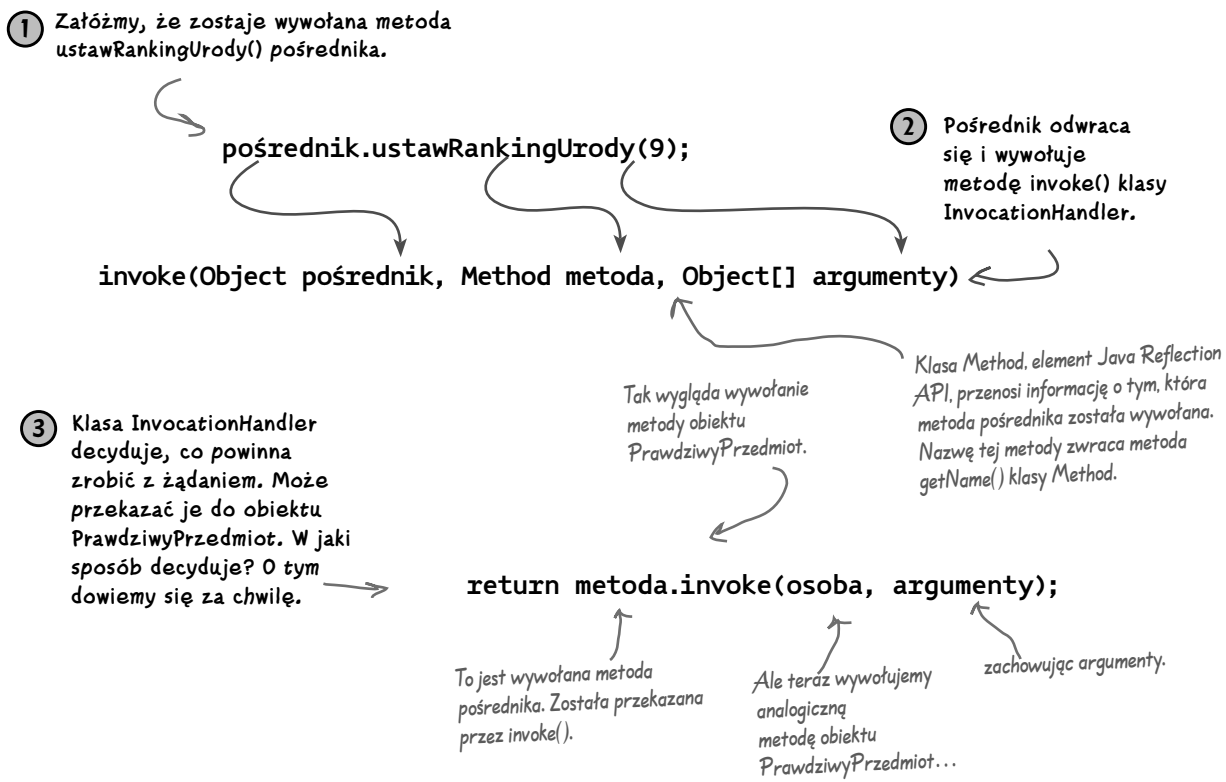


Krok pierwszy — implementowanie klas InvocationHandler

Musimy napisać dwie klasy obsługujące wywołania, jedną dla właścicieli i drugą dla pozostałych obiektów OsobaKomponent. Jaka będzie treść tych klas? Ich działanie powinno być następujące: gdy klient wywołuje metodę pośrednika, pośrednik przekazuje je do klasy InvocationHandler, ale *nie* wywołuje przy tym analogicznej metody. Co więc wywołuje? Spójrzmy jeszcze raz na interfejs:



Jest tylko jedna metoda, invoke(). Bez względu na to, która metoda pośrednika zostanie wywołana, skutkiem będzie wywołanie jedynej metody klasy InvocationHandler. Zobaczmy, jak to będzie wyglądać:



Implementowanie klas InvocationHandler — ciąg dalszy

Po wywołaniu przez pośrednika metody invoke(), skąd będziemy wiedzieć, co robić dalej? W typowym przypadku sprawdzamy, która metoda pośrednika została wywołana, a potem podejmujemy decyzję w oparciu o jej nazwę i, ewentualnie, o argumenty. Zaimplementujmy klasę InvocationHandlerWłaściciela:

Interfejs InvocationHandler jest elementem pakietu java.lang.reflect. Musimy zapewnić jego import.

Budujemy klasę, która będzie implementacją interfejsu InvocationHandler.

```
import java.lang.reflect.*;

public class InvocationHandlerWłaściciela implements InvocationHandler {
    OsobaKomponent osoba;

    public InvocationHandlerWłaściciela(OsobaKomponent osoba) {
        this.osoba = osoba;
    }

    public Object invoke(Object pośrednik, Method metoda, Object[] argumenty)
            throws IllegalAccessException {

        try {
            if (metoda.getName().startsWith("pobierz")) {
                return metoda.invoke(osoba, argumenty);
            } else if (metoda.getName().equals("ustawRankingUrody")) {
                throw new IllegalAccessException();
            } else if (metoda.getName().startsWith("ustaw")) {
                return metoda.invoke(osoba, argumenty);
            }
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

W konstruktorze przekazywany jest obiekt PrawdziwyPrzedmiot. Zachowujemy odwołanie do niego.

Tutaj mamy metodę invoke używaną przy każdym wywołaniu metody pośrednika.

Jeżeli ta metoda jest metodą typu getter, przekazujemy wywołanie do obiektu PrawdziwyPrzedmiot.

W przypadku, gdy jest to metoda ustawRankingUrody(), przerywamy przetwarzanie, generując wyjątek IllegalAccessException.

Tak będzie, gdy obiekt PrawdziwyPrzedmiot wygeneruje wyjątek.

Ponieważ mamy uprawnienia właściciela, każda metoda typu setter jest dozwolona. Przekazujemy wywołanie do obiektu PrawdziwyPrzedmiot.

W przypadku innych wywołań nie podejmujemy żadnych działań, ale po prostu zwracamy null.



Klasa InvocationHandlerNieWłaściciela działa podobnie jak klasa InvocationHandlerWłaściciela, ale *zezwala* na wywoływanie metody ustawRankingUrody() i *nie zezwala* na wywoływanie innych metod typu setter. Spróbuj napisać ją samodzielnie:

Krok drugi — tworzenie klasy Proxy i obiektu pośrednika

Pozostaje kwestia dynamicznego budowania klasy pośrednika i utworzenie jej egzemplarza. Rozpocznijmy od metody, która pobiera obiekt OsobaKomponent i umie utworzyć pośrednik właściciela. Utworzymy więc pewien rodzaj pośrednika, który przekazuje wywołania metod do klasy InvocationHandlerWłaściciela. Oto kod:

Ta metoda pobiera obiekt OsobaKomponent (PrawdziwyPrzedmiot) i zwraca jego obiekt pośrednika. Ponieważ pośrednik ma ten sam interfejs, zwracamy obiekt tego samego typu co argument.

Ta część kodu tworzy obiekt. Nie wygląda zbyt przejrzyście, więc zajmiemy się nią bardziej szczegółowo.

Aby utworzyć pośrednika, korzystamy ze statycznej metody newProxyInstance() klasy Proxy…

```
OsobaKomponent pobierzPośrednikWłaściciela(OsobaKomponent osoba) {

        return (OsobaKomponent) Proxy.newProxyInstance(
                osoba.getClass().getClassLoader(),
                osoba.getClass().getInterfaces(),
                new InvocationHandlerWłaściciela(osoba));
    }
```

Przekazujemy jej obiekt ClassLoader wykorzystywanego przedmiotu,…

…zbiór interfejsów, które pośrednik musi implementować…

Konstruktorowi klasy InvocationHandler przekazujemy obiekt PrawdziwyPrzedmiot. W kodzie klasy przedstawionym dwie strony wcześniej widać, że ten parametr wywołania umożliwia procedurom obsługi wywołań dostęp do przedmiotu.

… i obiekt implementujący interfejs InvocationHandler, w tym przypadku InvocationHandlerWłaściciela.



Zaostrz ołówek

Choć kod wygląda na zawiły, utworzenie dynamicznego pośrednika nie wymaga dużego wysiłku. Spróbuj napisać kod metody pobierzPośrednikNieWłaściciela(), która zwraca obiekt Proxy związany z obiektem InvocationHandlerNieWłaściciela:

Spróbuj pójść o krok dalej: czy potrafisz napisać pojedynczą metodę pobierzPośrednik(), która pobiera obiekty InvocationHandler i OsobaKomponent, aby zwrócić obiekt pośrednika, który korzysta z obiektu InvocationHandler?

Testowanie serwisu towarzyskiego

Przeprowadźmy test naszego serwisu i przekonajmy się, jak pośredniki kontrolują dostęp do metod typu setter.

```
public class BiuroMatrymonialneTest {
    // zmienne egzemplarza

    public static void main(String[] args) {
        BiuroMatrymonialneTest test = new BiuroMatrymonialneTest();
        test.testuj();
    }

    public BiuroMatrymonialneTest() {
        inicjalizujBD();
    }

    public void testuj() {
        OsobaKomponent jasiu = pobierzZBazyDanych("Jaś Fasolka");
        OsobaKomponent pośrednikWłaściciela = pobierzPośrednikWłaściciela(jasiu);
        System.out.println("Imię to " + pośrednikWłaściciela.pobierzImię());
        pośrednikWłaściciela.ustawZainteresowania("kręgle, Go");
        System.out.println("Pośrednik właściciela ustawił wartość zainteresowania");
        try {
            pośrednikWłaściciela.ustawRankingUrody(10);
        } catch (Exception e) {
            System.out.println("Nie można zmieniać oceny przy użyciu pośrednika właściciela");
        }
        System.out.println("Ocena to " + pośrednikWłaściciela.pobierzRankingUrody());

        OsobaKomponent pośrednikNieWłaściciela = pobierzPosrednikNieWłaściciela(jasiu);
        System.out.println("Imię to " + pośrednikNieWłaściciela.pobierzImię());
        try {
            pośrednikNieWłaściciela.ustawZainteresowania("kręgle, Go");
        } catch (Exception e) {
            System.out.println("Nie można zmienić zainteresowań przy użyciu pośrednika
obcego");
        }
        pośrednikNieWłaściciela.ustawRankingUrody(3);
        System.out.println("Pośrednik obcy zmienił wartość oceny");
        System.out.println("Ocena to " + pośrednikNieWłaściciela.pobierzRankingUrody());
    }

    // inne metody, takie jak pobierzPośrednikWłaściciela i pobierzPośrednikNieWłaści-
ciela
}
```

Procedura główna tworzy egzemplarz klasy testującej i wywołuje metodę testuj(), aby rozpocząć właściwą pracę.

Konstruktor inicjalizuje bazę danych osób, które zapisały się do serwisu.

Pobieramy osobę z bazy danych.

... i tworzymy pośrednik właściciela.

Wywołujemy metodę typu getter

następnie metodę typu setter

po czym próbujemy zmienić ocenę urody.

To nie powinno się udać!

Teraz tworzymy pośrednika obsługującego innych klientów niż właściciel

i wywołujemy metodę typu getter

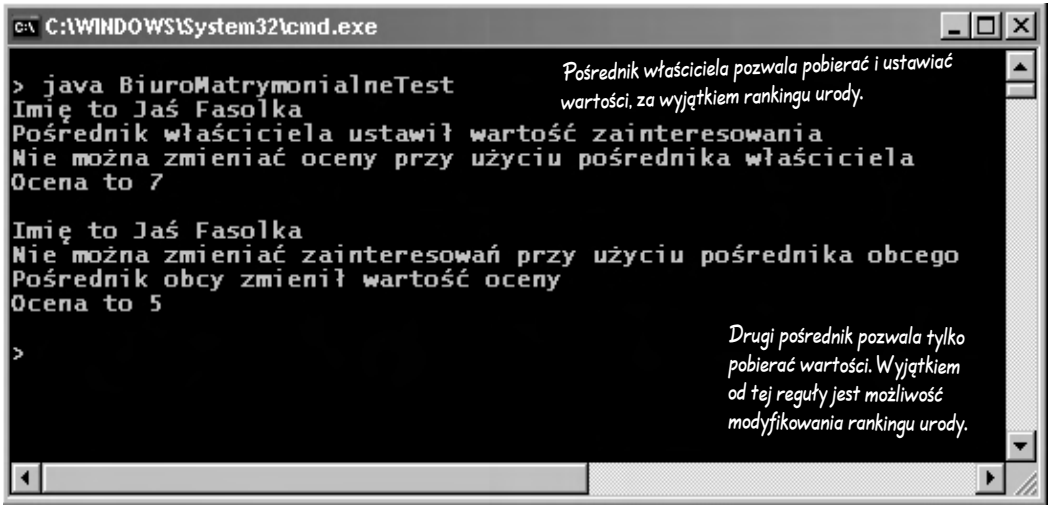
aby następnie wywołać metodę typu setter.

To nie powinno się udać!

Na końcu próbujemy zmienić ocenę.

To powinno się udać!

Uruchamiamy test...



Nie istnieją głupie pytania

P: Co jest takie „dynamiczne" w dynamicznych pośrednikach? Czy to, że tworzę egzemplarz i wiążę go z obiektem InvocationHandler w czasie wykonywania programu?

O: Nie, pośrednik jest dynamiczny, ponieważ jego klasa powstaje w czasie wykonywania programu. Dopóki aplikacja nie zostanie uruchomiona, klasa pośrednika w ogóle nie istnieje. Jest tworzona „na żądanie", w oparciu o zbiór przekazywanych interfejsów.

P: Moja klasa InvocationHandler wygląda jak bardzo dziwny pośrednik nieimplementujący żadnej z metod osłanianej klasy.

O: Klasa InvocationHandler nie jest pośrednikiem — to klasa, której pośrednik przekazuje obsługę wywołań metod. Sam pośrednik jest budowany dynamicznie w czasie wykonywania programu przy użyciu statycznej metody Proxy.newProxyInstance().

P: Czy istnieje sposób, aby określić, czy pewna klasa jest klasą Proxy?

O: Tak. Klasa Proxy ma statyczną metodę o nazwie isProxyClass(). Wywołanie tej metody zwraca wartość true, gdy klasa jest dynamicznym pośrednikiem. Poza tą jedną metodą klasa Proxy będzie funkcjonować tak samo, jak każda inna klasa implementująca pewien zbiór interfejsów.

P: Czy są jakieś ograniczenia, którym podlegają interfejsy przekazywane metodzie newProxyInstance()?

O: Tak, jest kilka takich ograniczeń. Po pierwsze, warto podkreślić, że metoda newProxyInstance() pobiera tablicę interfejsów, a nie klas. Najistotniejszym ograniczeniem jest to, że wszystkie interfejsy niepubliczne muszą należeć do tego samego pakietu. Nie można też wskazać dwóch interfejsów, w których pewne metody mają te same sygnatury. Dodatkowo istnieje kilka ograniczeń o mniejszym znaczeniu. Jeżeli zamierzamy korzystać z dynamicznych pośredników, powinniśmy przejrzeć dokumentację Javadoc, zwracając uwagę także na to, co jest w niej napisane drobniejszym drukiem.

P: Dlaczego używamy tu szkieletów? Wydawało mi się, że uwolniliśmy się od nich jeszcze w Java 1.2.

O: Masz rację; nie musimy generować szkieletów. Od wersji Java 1.2 biblioteka RMI może przekazywać wywołania klienta bezpośrednio do usługi zdalnej, opierając się na mechanizmie refleksji. Pojęcie szkieletu przedstawiliśmy celowo, ponieważ pomaga ono zrozumieć koncepcję mechanizmu. Ostatecznie, musi istnieć pewien ukryty element, który umożliwia komunikację między pniakiem klienta a usługą.

P: Słyszałem, że w Javie 5 nie są już nawet potrzebne obiekty pośredników (obiekty pomocnicze klienta). Czy to prawda?

O: Oczywiście. W Javie 5 mechanizmy RMI i Dynamic Proxy zostały połączone — obiekty pośredników są generowane dynamicznie przy użyciu mechanizmu pośrednika dynamicznego. Obiekt pośrednika obiektu zdalnego to egzemplarz klasy java.lang.reflect.Proxy (korzystający z obiektu InvocationHandler). Klasa ta jest generowana automatycznie i zapewnia wszelkie zachowania niezbędne do przekazania lokalnych wywołań klienta do obiektu zdalnego. Używanie narzędzia rmic w ogóle nie jest potrzebne. Wszelkie aspekty komunikacji klienta z obiektem zdalnym zostały ukryte.



Dopasuj wzorce do opisów:

Wzorzec	Opis
Dekorator	Osłania inny obiekt i zapewnia mu inny interfejs.
Fasada	Osłania inny obiekt i zapewnia mu dodatkowe zachowania.
Proxy	Osłania inny obiekt, aby kontrolować dostęp do niego
Adapter	Osłania grupę obiektów, upraszczając ich interfejs.

ZOO pośredników



Witamy w ogrodzie zoologicznym Obiektowa!

Poznaliśmy mechanizmy pośrednika zdalnego, wirtualnego i chroniącego. Jednak mutacji wzorca Proxy jest znacznie więcej. Tutaj, w kąciku ZOO poświęconym pośrednikom znajdziemy ciekawy zbiór pochwyconych na wolności okazów.

Nasza praca nie została zakończona. Nie ulega wątpliwości, że w przyszłości spotkasz znacznie więcej odmian omawianego tu wzorca. Przyjrzyjmy się kolekcji Obiektowa:



Środowisko naturalne: spotykany najczęściej w systemach zapór sieciowych przedsiębiorstw.

Firewall Proxy
— pośrednik-zapora sieciowa kontroluje dostęp do zbioru zasobów sieciowych, chroniąc przedmiot przed „szkodliwymi" klientami.

Pomóż określić środowisko naturalne.



Smart Reference Proxy
— inteligentny pośrednik odwołań wykonuje dodatkowe czynności przy każdym odwołaniu do przedmiotu; na przykład takie, jak zliczanie odwołań.



Caching Proxy
— pośrednik buforujący zapewnia tymczasowe składowanie wyników kosztownych operacji. Może też umożliwiać współużytkowanie wyników wielu klientom. Ogranicza ilość wykonywanych obliczeń i opóźnienia wynikające z pracy sieciowej.

Środowisko naturalne: często spotykany na serwerach proxy sieci WWW oraz w systemach zarządzania zawartością i publikowania.



Spotykany w systemach JavaSpace, gdzie zarządza synchronizacją dostępu do zbioru obiektów w środowisku rozproszonym.

Synchronization Proxy
— pośrednik synchronizujący zapewnia bezpieczny dostęp do przedmiotu w środowisku wielowątkowym.

Pomóż określić środowisko naturalne.

_____

_____

_____

_____

_____

Complexity Hiding Proxy
— pośrednik ukrywający złożoność



ukrywa złożoność i kontroluje dostęp do rozbudowanego zbioru klas. Z oczywistych powodów nazywany też Facade Proxy. Różni się od wzorca Fasady tym, że — jako pośrednik — kontroluje dostęp, podczas gdy zadaniem obiektu Fasady jest tylko zapewnienie alternatywnego interfejsu.



Copy-On-Write Proxy
— pośrednik kopiujący zarządza kopiowaniem obiektu, opóźniając tę operację do momentu, gdy obiekt jest faktycznie niezbędny klientowi. Jest to odmiana pośrednika wirtualnego.

Środowisko naturalne: widziany w pobliżu klasy CopyOnWriteArrayList języka Java 5.

Notatki: napisz, jakie inne obiekty typu Proxy spotkałeś w praktyce:

_____

_____

_____

_____

_____



Twoja skrzynka narzędziowa

Skrzynka narzędziowa jest niemal pełna; jesteś gotowy zmierzyć się z niemal każdym problemem projektowym, jaki stanie na Twojej drodze.



CELNE SPOSTRZEŻENIA

- Wzorzec Proxy zapewnia istnienie pośrednika, który reprezentuje inny obiekt, aby kontrolować dostęp do niego. Zarządzanie dostępem może być realizowane za pomocą wielu metod.
- Zdalny pośrednik (Remote Proxy) zarządza interakcjami między klientem a obiektem zdalnym.
- Wirtualny pośrednik (Virtual Proxy) zarządza dostępem do obiektu, którego utworzenie jest kosztowne.
- Pośrednik chroniący (Protection Proxy) zarządza dostępem do metod obiektu, ograniczając uprawnienia niektórych wywołujących.
- Istnieje wiele odmian wzorca Proxy, w tym pośrednik buforujący (Caching Proxy), pośrednik synchronizujący (Synchronization Proxy), pośrednik-zapora sieciowa (Firewall Proxy), pośrednik kopiujący (Copy-On-Write Proxy) itd.
- Wzorzec Proxy jest strukturalnie podobny do wzorca Dekorator, ale różni się ich przeznaczenie.
- Wzorzec Dekorator wzbogaca zachowania obiektu, podczas gdy wzorzec Proxy zarządza dostępem do niego.
- Standardowa obsługa mechanizmu pośrednika w języku Java umożliwia zbudowanie dynamicznej klasy Proxy w czasie wykonywania programu. Wywołania są przekazywane do klasy z interfejsem InvocationHandler.
- Podobnie jak każda inna osłona, pośrednik zwiększa liczbę klas i obiektów w projekcie.



Rozwiązania ćwiczeń



Klasa InvocationHandlerNieWłaściciela działa podobnie jak klasa InvocationHandlerWłaściciela, ale *zezwala* na wywoływanie metody ustawRankingUrody() i *nie zezwala* na wywoływanie innych metod typu setter. Spróbuj napisać ją samodzielnie:

```
import java.lang.reflect.*;

public class InvocationHandlerNieWłaściciela implements InvocationHandler {
    OsobaKomponent osoba;

    public InvocationHandlerNieWłaściciela(OsobaKomponent osoba) {
        this.osoba = osoba;
    }

    public Object invoke(Object pośrednik, Method metoda, Object[] argumenty)
            throws IllegalAccessException {

        try {
            if (metoda.getName().startsWith("pobierz")) {
                return metoda.invoke(osoba, argumenty);
            } else if (metoda.getName().equals("ustawRankingUrody")) {
                return metoda.invoke(osoba, argumenty);
            } else if (metoda.getName().startsWith("ustaw")) {
                throw new IllegalAccessException();
            }
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        }
        return null;
    }
}
```



Rozwiązania ćwiczeń

Rozwiązanie łamigłówki projektowej

Klasa PośrednikObrazków ma dwa stany, zaimplementowane przy użyciu instrukcji warunkowych. Czy masz pomysł na inny wzorzec, który mógłby uporządkować ten kod? Jak można zmodyfikować klasę?

Używamy wzorca Stan: implementujemy dwa stany, ObrazekZaładowany i ObrazekNiezaładowany. Kod z instrukcji if umieszczamy w odpowiednich klasach stanu. Rozpoczynamy pracę w stanie ObrazekNiezaładowany, po czym przechodzimy do stanu ObrazekZaładowany, gdy zakończy się pobieranie grafiki z sieci.



Zaostrz ołówek

Choć kod wygląda na zawiły, utworzenie dynamicznego pośrednika nie wymaga dużego wysiłku. Spróbuj napisać kod metody pobierzPośrednikNie Właściciela(), która zwraca obiekt InvocationHandlerNieWłaściciela:

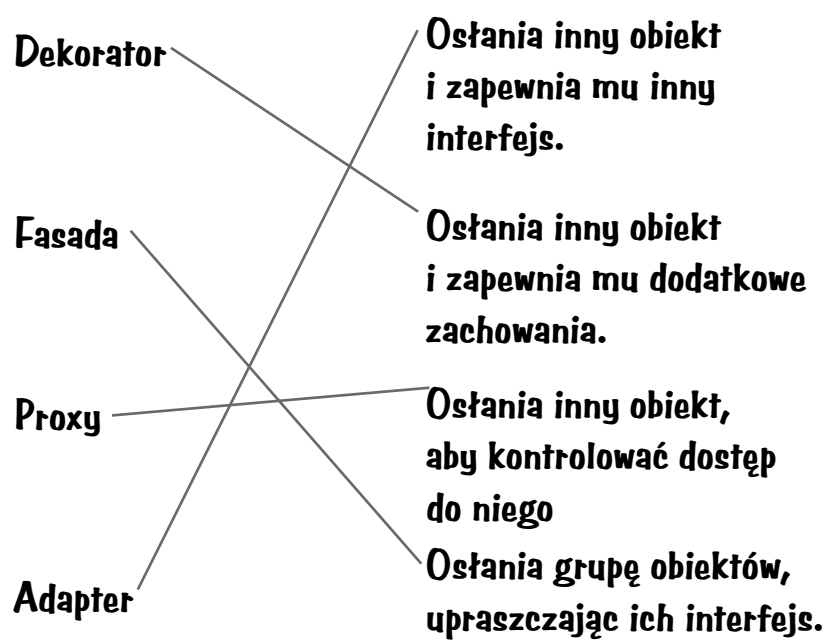
```
OsobaKomponent pobierzPośrednikNieWłaściciela(OsobaKomponent osoba) {

    return (OsobaKomponent) Proxy.newProxyInstance(
            osoba.getClass().getClassLoader(),
            osoba.getClass().getInterfaces(),
            new InvocationHandlerNieWłaściciela(osoba));
}
```

Rozwiązania ćwiczeń

Kto co robi?

Dopasuj wzorce do opisów:





Kod gotowy do użycia

Kod przeglądarki okładek CD

```
package headfirst.proxy.virtualproxy;

import java.net.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;

public class PośrednikObrazkówTest {
    ImageComponent imageComponent;
    JFrame frame = new JFrame("Przeglądarka okładek CD");
    JMenuBar menuBar;
    JMenu menu;
    Hashtable dyski = new Hashtable();

    public static void main (String[] args) throws Exception {
        PośrednikObrazkówTest test = new PośrednikObrazkówTest();
    }

    public PośrednikObrazkówTest() throws Exception{
        dyski.put("Ambient: Music for Airports","http://images.amazon.com/images/P/
B00000S2K.01.LZZZZZZZ.jpg");
        dyski.put("Buddha Bar","http://images.amazon.com/images/P/B0000I9XBYK.01.LZZZZZZZ.jpg");
        dyski.put("Ima","http://images.amazon.com/images/P/B00000I5IRM.01.LZZZZZZZ.jpg");
        dyski.put("Karma","http://images.amazon.com/images/P/B00000I5DCB.01.LZZZZZZZ.gif");
        dyski.put("MCMXC A.D.","http://images.amazon.com/images/P/B00000I2URV.01.LZZZZZZZ.jpg");
        dyski.put("Northern Exposure","http://images.amazon.com/images/P/B00000I3SFN.01.
LZZZZZZZ.jpg");
        dyski.put("Selected Ambient Works, Vol. 2","http://images.amazon.com/images/P/
B00000I2MNZ.01.LZZZZZZZ.jpg");

        URL startURL = new URL((String)dyski.get("Selected Ambient Works, Vol. 2"));
        menuBar = new JMenuBar();
        menu = new JMenu("Ulubione CD");
        menuBar.add(menu);
        frame.setJMenuBar(menuBar);
```

```
        for(Enumeration e = dyski.keys(); e.hasMoreElements();) {
            String nazwa = (String)e.nextElement();
            JMenuItem menuItem = new JMenuItem(nazwa);
            menu.add(menuItem);
            menuItem.addActionListener(new ActionListener() {
                    public void actionPerformed(ActionEvent event) {
                        imageComponent.setIcon(new PośrednikObrazków(pobierzUrlCD(event.
getActionCommand())));
                    frame.repaint();
                     }
            });
        }

        // przygotowanie ramki i menu

        Icon icon = new PośrednikObrazków(startURL);
        imageComponent = new ImageComponent(icon);
        frame.getContentPane().add(imageComponent);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(800,600);
        frame.setVisible(true);

    }

    URL pobierzUrlCD(String nazwa) {
        try {
            return new URL((String)dyski.get(nazwa));
        } catch (MalformedURLException e) {
            e.printStackTrace();
            return null;
        }
    }
}
```



Kod gotowy do użycia

Kod przeglądarki okładek CD — ciąg dalszy

```
package headfirst.proxy.virtualproxy;

import java.net.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class PośrednikObrazków implements Icon {
    ImageIcon imageIcon;
    URL obrazekURL;
    Thread pobieranieThread;
    boolean pobieranie = false;

    public PośrednikObrazków(URL url) { obrazekURL = url; }

    public int getIconWidth() {
        if (imageIcon != null) {
            return imageIcon.getIconWidth();
        } else {
            return 800;
        }
    }

    public int getIconHeight() {
        if (imageIcon != null) {
            return imageIcon.getIconHeight();
        } else {
            return 600;
        }
    }

    public void paintIcon(final Component c, Graphics  g, int x,  int y) {
        if (imageIcon != null) {
            imageIcon.paintIcon(c, g, x, y);
        } else {
            g.drawString("Trwa ładowanie okładki, proszę czekać...", x+300, y+190);
            if (!pobieranie) {
                pobieranie = true;

                pobieranieThread = new Thread(new Runnable() {
                    public void run() {
                        try {
                            imageIcon = new ImageIcon(obrazekURL, "Okładka CD");
                            c.repaint();
```

```
                        } catch (Exception e) {

                            e.printStackTrace();
                        }
                    }
                });
                pobieranieThread.start();
            }
        }
    }
}
```

```
package headfirst.proxy.virtualproxy;

import java.awt.*;
import javax.swing.*;

class ImageComponent extends JComponent {
    private Icon icon;

    public ImageComponent(Icon icon) {
        this.icon = icon;
    }

    public void setIcon(Icon icon) {
        this.icon = icon;
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        int w = icon.getIconWidth();
        int h = icon.getIconHeight();
        int x = (800 - w)/2;
        int y = (600 - h)/2;
        icon.paintIcon(this, g, x, y);
    }
}
```

Łączenie wzorców



Przyszłoby Ci do głowy, że wzorce mogą pracować razem? Byliśmy już świadkami wielu niespokojnych „Pogawędek przy kominku" (a ominął Cię „Death Match" wzorców, który wydawca kazał wyrzucić) — czy wsłuchując się w ich ton, można jeszcze liczyć na to, że wzorce będą ze sobą współpracować? Możesz wierzyć lub nie, ale najbardziej wyszukane projekty obiektowe wykorzystują wiele wzorców jednocześnie. Przygotuj się na kolejny poziom wiedzy o wzorcach projektowych. Czas na wzorce złożone.

Wzorce razem



Jednym z najlepszych sposobów wykorzystania wzorców jest zaprzęgnięcie ich do współpracy z innymi wzorcami. W praktyce, im więcej korzystamy z tych standardowych schematów projektowania, tym częściej widzimy, jak współgrają one z innymi, podobnie typowymi rozwiązaniami. Istnieje nawet specjalna nazwa oznaczająca zespół wzorców, które współpracują ze sobą w jednym projekcie i razem tworzą rozwiązanie nadające się do wielokrotnego stosowania w odniesieniu do wielu problemów. Chodzi o *wzorzec złożony* (ang. *compound pattern*). Tak właśnie, będziemy teraz mówić o wzorcach powstałych z połączenia kilku innych wzorców!

Opracowano już bardzo wiele wzorców złożonych. Teraz, gdy dysponujemy podstawami wiedzy o wzorcach, szybko zauważymy, że są to połączenia znanych schematów. Na pewno uprości to analizowanie ich pracy i znaczenia w projekcie.

Rozdział niniejszy rozpoczniemy od kaczek — tych, które poznaliśmy podczas budowania symulatora kaczki. Kaczki są tu nieuniknione. Towarzyszyły nam przez całą książkę i posłużyły do zilustrowania niejednego wzorca. Teraz pomogą nam zapoznać się z projektami, w których stosowanych jest wiele wzorców jednocześnie. Oczywiście, sam fakt, że wiele wzorców znalazło się w jednym rozwiązaniu, nie oznacza jeszcze, że mamy do czynienia z wzorcem złożonym. Aby tak było, użyte wzorce muszą tworzyć rozwiązanie o charakterze ogólnym, które może zostać wykorzystane w wielu różnych projektach. W drugiej części rozdziału przedstawimy więc *prawdziwy* wzorzec złożony: MVC (Model-Widok-Kontroler. Jeżeli jeszcze o nim nie słyszałeś, przygotuj się na poznanie wzorca o ogromnych możliwościach, jednego z najistotniejszych, które omawiamy w tej książce.

W jednym rozwiązaniu stosuje się często wiele różnych wzorców.

Wzorzec złożony to połączenie dwóch lub większej liczby wzorców służące rozwiązaniu ogólnego lub powtarzającego się problemu.

Powrót kaczek

Jak już zaznaczyliśmy, w tym rozdziale ponownie zajmiemy się kaczkami. Tym razem posłużą nam one do zilustrowania tego, jak wiele wzorców może koegzystować, a nawet współpracować w jednym rozwiązaniu.

Symulator kaczki napiszemy jeszcze raz, od podstaw, zwracając uwagę na interesujące możliwości, jakie daje łączenie wzorców. A więc, czas... start!

1 Rozpoczniemy od utworzenia interfejsu Kwacząca.

Jak powiedzieliśmy, rozpoczynamy pisanie symulatora od nowa. Tym razem kaczki będą implementacjami interfejsu Kwacząca. Będzie to informacja o tym, które części symulatora potrafią kwakać — DzikaKaczka, PłaskonosKaczka, WabikKaczka, a może nawet GumowaKaczka.

```
public interface Kwacząca {
    public void kwacz();
}
```

Klasa Kwacząca musi robić dobrze tylko jedną rzecz: kwakać!

2 A teraz parę Kaczek, które implementują ten interfejs.

Cóż znaczy interfejs bez implementujących go klas? Czas utworzyć kilka kaczek rzeczywistych (ale nie idziemy nad żaden staw).

Standardowa dzika kaczka.

```
public class DzikaKaczka implements Kwacząca {
    public void kwacz() {
        System.out.println("Kwa! Kwa!");
    }
}
```

```
public class PłaskonosKaczka implements Kwacząca
{
    public void kwacz() {
        System.out.println("Kwa! Kwa!");
    }
}
```

Jeżeli symulator ma być interesujący, musimy mieć więcej niż jeden gatunek kaczki.

Nie byłoby tyle zabawy, gdybyśmy nie dodali jeszcze większej liczby kaczek.

Pamiętasz, jak to było? Mieliśmy jeszcze wabiki (używane przez myśliwych, więc niewątpliwie kwaczące) i kaczki gumowe.

```
public class WabikKaczka implements Kwacząca {
    public void kwacz() {
        System.out.println("Kwak");
    }
}
```

Wabik kwacze, ale nie do końca brzmi jak prawdziwa kaczka.

```
public class GumowaKaczka implements Kwacząca {
    public void kwacz() {
        System.out.println("Piszczę!");
    }
}
```

Kaczka gumowa, która piszczy, a nie kwacze.

3 Mamy już kaczki, czas na symulator.

Napiszmy teraz kod symulatora, który tworzy kilka kaczek i sprawdza, czy ich narządy do kwakania są sprawne...

Metoda główna.

```
public class SymulatorKaczek {
    public static void main(String[] args) {
        SymulatorKaczek symulator = new SymulatorKaczek();
        symulator.uruchom();
    }

    void uruchom() {
        Kwacząca dzikaKaczka = new DzikaKaczka();
        Kwacząca płaskonosKaczka = new PłaskonosKaczka();
        Kwacząca wabikKaczka = new WabikKaczka();
        Kwacząca gumowaKaczka = new GumowaKaczka();

        System.out.println("\nSymulator Kaczek");

        uruchom(dzikaKaczka);
        uruchom(płaskonosKaczka);
        uruchom(wabikKaczka);
        uruchom(gumowaKaczka);
    }

    void uruchom(Kwacząca kaczka) {
        kaczka.kwacz();
    }
}
```

Tworzymy obiekt symulatora i wywołujemy metodę uruchom().

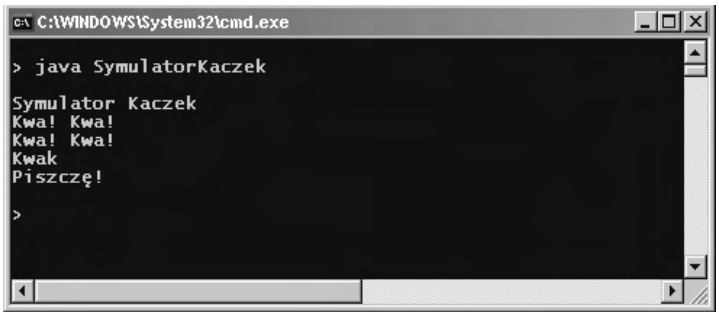
Potrzebujemy kilku kaczek, więc tworzymy po jednym egzemplarzu każdego kwaczącego obiektu...

... a następnie uruchamiamy je.

Tutaj przeciążamy metodę uruchom(), aby użyć jej do uruchamiania pojedynczej kaczki.

Tu wykorzystujemy magię polimorfizmu: bez względu na to, jaki kwaczący obiekt zostanie metodzie przekazany, rozpocznie ona kwakanie.

Niezbyt imponujące na tym etapie, ale nie użyliśmy jeszcze wzorców!



Wszystkie kaczki implementują interfejs Kwacząca, ale implementacje są różne, więc kwakanie nie zawsze jest takie samo.

Wygląda na to, że wszystko działa. Jak dotąd jest nieźle.

4 Gdzie kaczki, tam i gęsi.

Tam, gdzie znaleźliśmy jednego ptaka wodnego, będą też zapewne inne. Oto klasa Gęś, która od dawna czekała na swoje miejsce w symulatorze.

Gęś nie kwacze, ale gęga.

```
public class Gęś {
    public void gęgaj() {
        System.out.println("Gę! Gę!");
    }
}
```



WYTĘŻ UMYSŁ

Załóżmy, że potrzebna jest możliwość wprowadzenia gęsi wszędzie tam, gdzie używamy kaczek. Było nie było, gęsi też mają głos, gęsi latają i pływają. Dlaczego nie miałyby znaleźć swojego miejsca w symulatorze?

Jaki wzorzec pozwoli na łatwe mieszanie kaczek i gęsi?

5 Potrzebujemy adaptera gęsi.

Nasz symulator wymaga interfejsu Kwacząca. Ponieważ gęsi nie kwaczą, ale gęgają, możemy zastosować adapter, który przystosuje gęś do bycia kaczką.

```
public class GęśAdapter implements Kwacząca {
    Gęś gęś;

    public GęśAdapter(Gęś gęś) {
        this.gęś = gęś;
    }

    public void kwacz() {
        gęś.gęgaj();
    }
}
```

Pamiętajmy, że adapter jest implementacją interfejsu docelowego. W tym przypadku jest to interfejs Kwacząca.

Konstruktor pobiera adaptowaną gęś.

Wywołanie metody kwacz() powoduje delegację do metody gęgaj() gęsi.

6 Teraz gęsi powinny znaleźć swoje miejsce w symulatorze.

Wszystko, czego potrzebujemy, to stworzyć gęś i opakować ją adapterem, który implementuje interfejs Kwacząca.

```
public class SymulatorKaczek {
    public static void main(String[] args) {
        SymulatorKaczek symulator = new SymulatorKaczek();
        symulator.uruchom();
    }

    void uruchom() {
        Kwacząca dzikaKaczka = new DzikaKaczka();
        Kwacząca płaskonosKaczka = new PłaskonosKaczka();
        Kwacząca wabikKaczka = new WabikKaczka();
        Kwacząca gumowaKaczka = new GumowaKaczka();
        Kwacząca gęśKaczka = new GęśAdapter(new Gęś());

        System.out.println("\nSymulator Kaczek: z adapterem gęsi");

        uruchom(dzikaKaczka);
        uruchom(płaskonosKaczka);
        uruchom(wabikKaczka);
        uruchom(gumowaKaczka);
        uruchom(gęśKaczka);
    }

    void uruchom(Kwacząca kaczka) {
        kaczka.kwacz();
    }
}
```

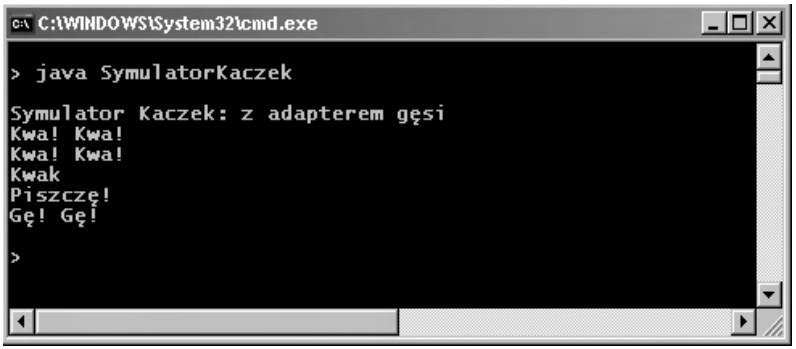
Opakowując gęś adapterem, tworzymy gęś, która zachowuje się jak kaczka.

Po osłonięciu gęsi możemy ją traktować podobnie jak wszystkie inne kwaczące.

7 Uruchamiamy nowy symulator...

Gdy tym razem uruchomimy symulator, lista obiektów przekazywanych metodzie uruchom() obejmie Gęś opakowaną adapterem. Wynik? Gęganie!

Oto gęś! Teraz gęś kwacze razem z kaczkami.





Kwakologia

Kwakolodzy zajmują się wszelkimi aspektami zachowania Kwacząca. Jedną z rzeczy, które zawsze chcieli zbadać, jest całkowita liczba kwaknięć stada kaczek.

Jak wprowadzić możliwość zliczania kwaknięć bez zmieniania klas kaczek?

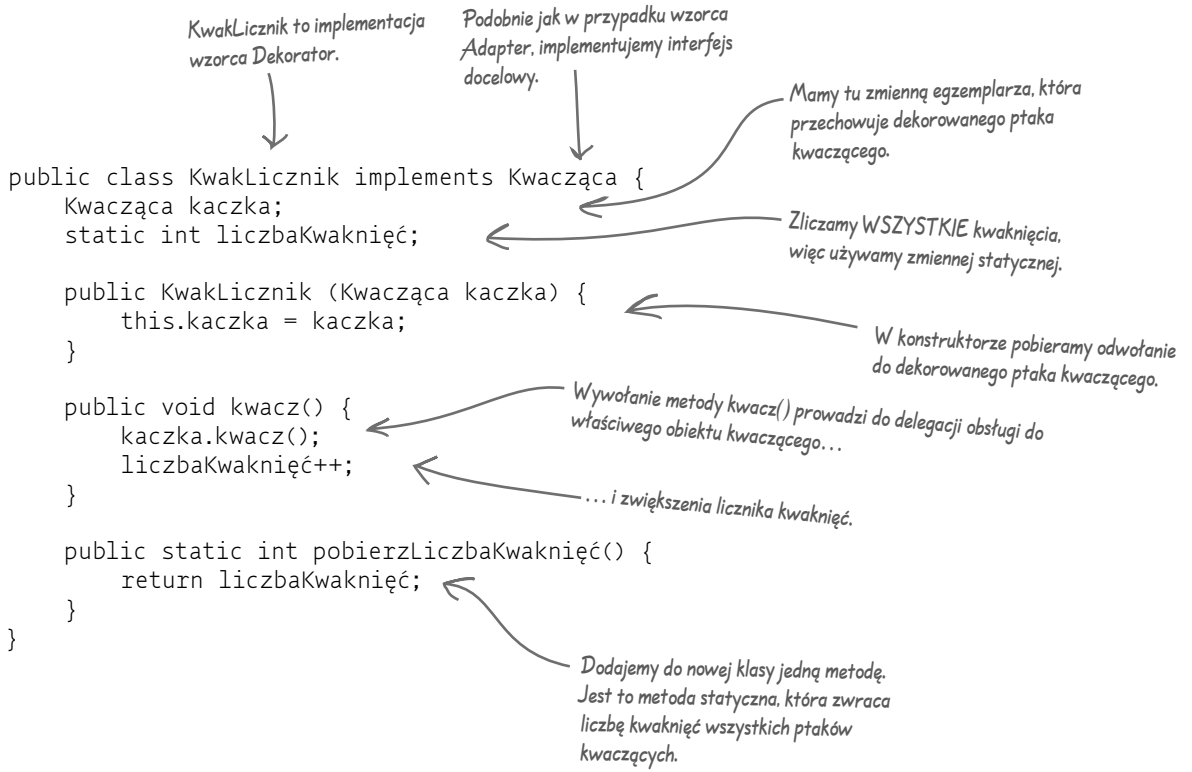
Czy masz pomysł, który wzorzec możemy tu zastosować?

J. Piwowarski, strażnik przyrody i kwakolog



⑧ Uszczęśliwimy kwakologów i wprowadzimy zliczanie kwaknięć.

W jaki sposób? Utworzymy klasę Dekorator, która zapewni kaczkom nowe zachowanie (zachowanie zliczania) poprzez osłonięcie odpowiednim obiektem. Kodu kaczek w ogóle nie będziemy zmieniać.




```
public class KwakLicznik implements Kwacząca {
    Kwacząca kaczka;
    static int liczbaKwaknięć;

    public KwakLicznik (Kwacząca kaczka) {
        this.kaczka = kaczka;
    }

    public void kwacz() {
        kaczka.kwacz();
        liczbaKwaknięć++;
    }

    public static int pobierzLiczbaKwaknięć() {
        return liczbaKwaknięć;
    }
}
```

9 Aktualizujemy symulator tak, aby korzystał z udekorowanych kaczek.

Musimy zapewnić opakowanie każdego tworzonego egzemplarza ptaka kwaczącego naszym nowym obiektem. Jeżeli o tym zapomnimy, zliczanie kwaknięć nie będzie działać.

```
public class SymulatorKaczek {
    public static void main(String[] args) {
        SymulatorKaczek symulator = new SymulatorKaczek();
        symulator.uruchom();
    }

    void uruchom() {
        Kwacząca dzikaKaczka = new KwakLicznik (new DzikaKaczka());
        Kwacząca płaskonosKaczka = new KwakLicznik (new PłaskonosKaczka());
        Kwacząca wabikKaczka = new KwakLicznik (new WabikKaczka());
        Kwacząca gumowaKaczka = new KwakLicznik (new GumowaKaczka());
        Kwacząca gęśKaczka = new GęśAdapter(new Gęś());

        System.out.println("\nSymulator Kaczek: z wzorcem Dekorator");

        uruchom(dzikaKaczka);
        uruchom(płaskonosKaczka);
        uruchom(wabikKaczka);
        uruchom(gumowaKaczka);
        uruchom(gęśKaczka);

        System.out.println("Kaczki kwaknęły " +
                           KwakLicznik.pobierzLiczbaKwaknięć() + " razy");
    }

    void uruchom(Kwacząca kaczka) {
        kaczka.kwacz();
    }
}
```

Za każdym razem, gdy tworzymy ptaka kwaczącego, osłaniamy go egzemplarzem nowej klasy.

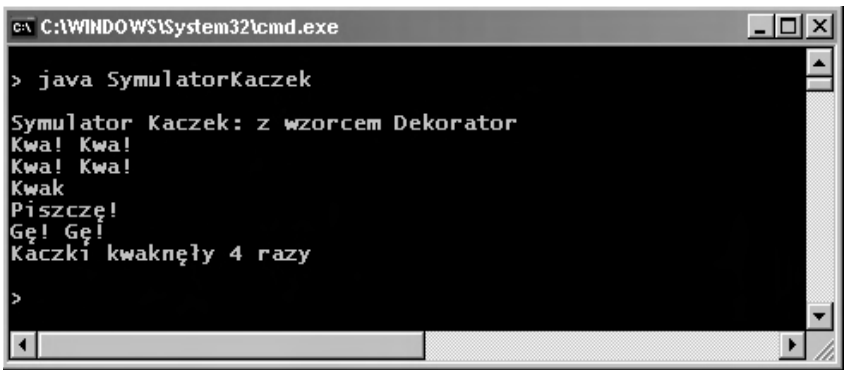
Strażnik przyrody powiedział, że nie chce zliczać gęgania, więc nie wprowadzamy opakowania.

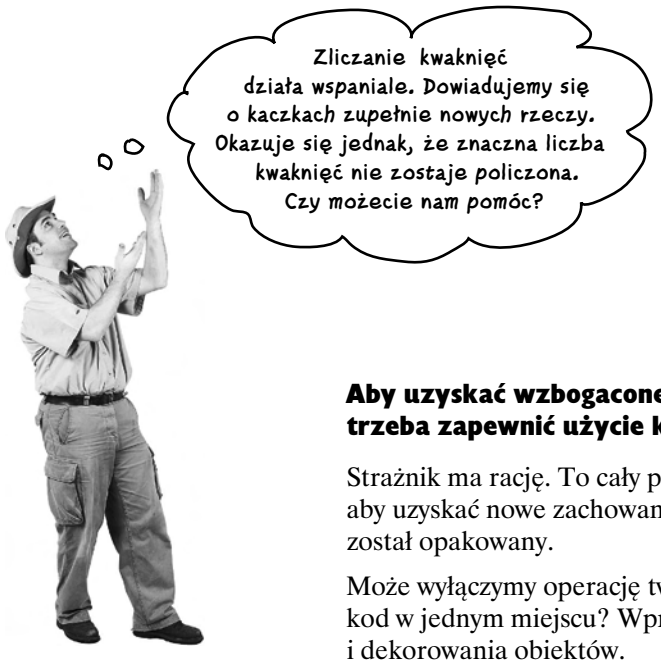
Tutaj wyświetlamy dane dla kwakologów.

Tutaj nic się nie zmienia, udekorowane obiekty to wciąż ptaki kwaczące.

Oto dane wyjściowe symulatora!

Nie liczymy gęsi.





Aby uzyskać wzbogacone zachowanie, trzeba zapewnić użycie klasy Dekorator.

Strażnik ma rację. To cały problem z osłanianiem obiektów: aby uzyskać nowe zachowania, trzeba dbać o to, aby obiekt został opakowany.

Może wyłączymy operację tworzenia kaczek i zgromadzimy jej kod w jednym miejscu? Wprowadźmy hermetyzację tworzenia i dekorowania obiektów.

Z jakim wzorcem to się kojarzy?

10 Potrzebujemy fabryki produkującej kaczki!

Wygląda na to, że potrzebujemy „kontroli jakości" zapewniającej, że każda kaczka zostanie opakowana. Zbudujemy w tym celu specjalną fabrykę. Fabryka powinna wytwarzać rodzinę produktów, która obejmuje różne typy kaczek. Użyjemy więc wzorca Fabryka Abstrakcyjna.

Rozpocznijmy od klasy AbstrakcyjnaFabrykaKaczek:

Definiujemy abstrakcyjną fabrykę implementowaną przez podklasy generujące różne rodziny obiektów.

```
public abstract class AbstrakcyjnaFabrykaKaczek {

    public abstract Kwacząca utwórzDzikaKaczka();
    public abstract Kwacząca utwórzPłaskonosKaczka();
    public abstract Kwacząca utwórzWabikKaczka();
    public abstract Kwacząca utwórzGumowaKaczka();
}
```

Każda metoda tworzy jeden rodzaj kaczki.

Najpierw stworzymy fabrykę, która tworzy kaczki bez opakowań, w zasadzie tylko po to, aby przypomnieć sobie, na czym polega zasada działania wzorca Fabryka:

Klasa FabrykaKaczek jest rozszerzeniem fabryki abstrakcyjnej.

```
public class FabrykaKaczek extends AbstrakcyjnaFabrykaKaczek {

    public Kwacząca utwórzDzikaKaczka() {
        return new DzikaKaczka();
    }

    public Kwacząca utwórzPłaskonosKaczka() {
        return new PłaskonosKaczka();
    }

    public Kwacząca utwórzWabikKaczka() {
        return new WabikKaczka();
    }

    public Kwacząca utwórzGumowaKaczka() {
        return new GumowaKaczka();
    }
}
```

Każda metoda tworzy pewien produkt fabryki: jeden rodzaj ptaka kwaczącego. Twór ów pozostaje nieznany dla symulatora, bo zadowala się on samą wiedzą, że produkt jest implementacją interfejsu Kwacząca.

Teraz tworzymy fabrykę, o którą nam naprawdę chodzi, a więc klasę FabrykaKaczekZLicznikiem:

FabrykaKaczekZLicznikiem również jest rozszerzeniem fabryki abstrakcyjnej.

```
public class FabrykaKaczekZLicznikiem extends AbstrakcyjnaFabrykaKaczek {

    public Kwacząca utwórzDzikaKaczka() {
        return new KwakLicznik(new DzikaKaczka());
    }

    public Kwacząca utwórzPłaskonosKaczka() {
        return new KwakLicznik(new PłaskonosKaczka());
    }

    public Kwacząca utwórzWabikKaczka() {
        return new KwakLicznik(new WabikKaczka());
    }

    public Kwacząca utwórzGumowaKaczka() {
        return new KwakLicznik(new GumowaKaczka());
    }
}
```

Każda metoda osłania ptaka kwaczącego obiektem zapewniającym zliczanie. Symulator nie musi wiedzieć, że coś się zmieniło. W dalszym ciągu operuje ptakami kwaczącymi. Różnica polega na tym, że tym razem strażnicy przyrody mogą mieć pewność, że wszystkie kwaknięcia zostaną policzone.

⑪ Przystosowujemy symulator do korzystania z fabryki.

Pamiętasz, na czym polega wzorzec Fabryka Abstrakcyjna? Używając go, tworzymy polimorficzną metodę, która pobiera fabrykę i wykorzystuje ją do tworzenia obiektów. Gdy przekazujemy różne fabryki, uzyskujemy różne rodziny produktów.

Zmienimy metodę uruchom() tak, aby pobierała fabrykę i wykorzystywała ją do tworzenia kaczek.

```
public class SymulatorKaczek {
    public static void main(String[] args) {
        SymulatorKaczek symulator = new SymulatorKaczek();
        AbstrakcyjnaFabrykaKaczek kaczkaFabryka = new FabrykaKaczekZLicznikiem();

        symulator.uruchom(kaczkaFabryka);
    }

    void uruchom(AbstrakcyjnaFabrykaKaczek kaczkaFabryka) {
        Kwacząca dzikaKaczka = kaczkaFabryka.utwórzDzikaKaczka();
        Kwacząca płaskonosKaczka = kaczkaFabryka.utwórzPłaskonosKaczka();
        Kwacząca wabikKaczka = kaczkaFabryka.utwórzWabikKaczka();
        Kwacząca gumowaKaczka = kaczkaFabryka.utwórzGumowaKaczka();
        Kwacząca gęśKaczka = new GęśAdapter(new Gęś());

        System.out.println("\nSymulator Kaczek: z wzorcem Fabryka Abstrakcyjna");

        uruchom(dzikaKaczka);
        uruchom(płaskonosKaczka);
        uruchom(wabikKaczka);
        uruchom(gumowaKaczka);
        uruchom(gęśKaczka);

        System.out.println("Kaczki kwaknęły " +
                           KwakLicznik.pobierzLiczbaKwaknięć() +
                           " razy");
    }

    void uruchom(Kwacząca kaczka) {
        kaczka.kwacz();
    }
}
```

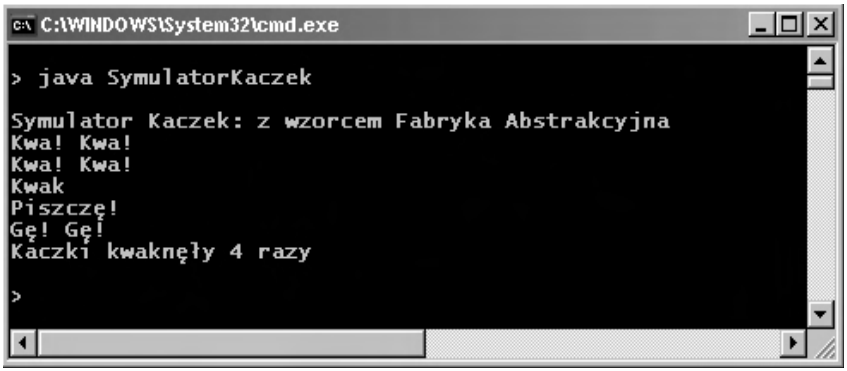
Najpierw tworzymy fabrykę, którą przekażemy metodzie uruchom().

Metoda uruchom() pobiera obiekt typu AbstrakcyjnaFabrykaKaczek i wykorzystuje go do tworzenia kaczek (już nie tworzy egzemplarzy samodzielnie).

Tutaj nic się nie zmienia! Jakieś starocie...

Oto wynik kolejnego uruchomienia symulatora...

Wynik jest podobny jak ostatnim razem, ale teraz zapewniamy udekorowanie wszystkich kaczek. Po to właśnie wprowadziliśmy fabrykę.





Zaostrz ołówek

Gęsi wciąż tworzymy bezpośrednio, odwołując się do klas konkretnych. Czy potrafisz napisać abstrakcyjną fabrykę gęsi? W jaki sposób powinna ona obsługiwać tworzenie „kwaczącej gęsi"?



Aha, teraz mamy stado kaczek.

Strażnik Piwowarski poruszył ciekawą kwestię: jak to się stało, że zarządzamy każdą kaczką niezależnie?

To nie jest zbyt poręczne!

```
Kwacząca dzikaKaczka = kaczkaFabryka.utwórzDzikaKaczka();
Kwacząca płaskonosKaczka = kaczkaFabryka.utwórzPłaskonosKaczka();
Kwacząca wabikKaczka = kaczkaFabryka.utwórzWabikKaczka();
Kwacząca gumowaKaczka = kaczkaFabryka.utwórzGumowaKaczka();
Kwacząca gęśKaczka = new GęśAdapter(new Gęś());

uruchom(dzikaKaczka);
uruchom(płaskonosKaczka);
uruchom(wabikKaczka);
uruchom(gumowaKaczka);
uruchom(gęśKaczka);
```

Tym, czego potrzebujemy, jest pewien sposób operowania kolekcjami i może nawet „podkolekcjami" kaczek (podkolekcja byłaby jedną rodziną). Wygodna byłaby też możliwość wywołania pewnej operacji dla całej grupy.

Jakiego wzorca użyjemy?

⑫ Utwórzmy stado kaczek (a właściwie - ptaków Kwaczących).

Czy pamiętasz wzorzec Kompozyt, umożliwiający traktowanie kolekcji obiektów w taki sam sposób, jak obiektów pojedynczych? Cóż może być tu lepszym przykładem niż stado ptaków kwaczących?!

Powoli przeanalizujemy, jak taki mechanizm ma działać:

Jak zapewne pamiętasz, obiekt złożony musi implementować ten sam interfejs, co obiekty-liście. Liśćmi będą w naszym przypadku ptaki kwaczące.

W każdym stadzie jest kolekcja ArrayList, która przechowuje listę należących do niego ptaków.

Metoda dodaj() dołącza kolejnego ptaka do stada.

```
public class Stado implements Kwacząca {
    ArrayList ptaki = new ArrayList();

    public void dodaj(Kwacząca ptak) {
        ptaki.add(ptak);
    }

    public void kwacz() {
        Iterator iterator = ptaki.iterator();
        while (iterator.hasNext()) {
            Kwacząca ptak = (Kwacząca)iterator.next();
            ptak.kwacz();
        }
    }
}
```

A teraz metoda kwacz() — ostatecznie, Stado też implementuje interfejs Kwacząca. Metoda kwacz() stada powinna operować na całym stadzie. Przeprowadzamy więc iterowanie kolekcji i wywołujemy dla każdego elementu metodę kwacz().



Bliskie spotkania z kodem aplikacji

Czy zauważyłeś, że próbowaliśmy „przemycić" jeden wzorzec, nie wspominając nawet o tym?

Tu jest! Wzorzec Iterator w akcji!

```
public void kwacz() {
    Iterator iterator = ptaki.iterator();
    while (iterator.hasNext()) {
        Kwacząca ptak = (Kwacząca)iterator.next();
        ptak.kwacz();
    }
}
```

⑬ Teraz musimy zmodyfikować symulator.

Obiekt złożony jest gotowy. Pozostaje tylko kilka wierszy kodu niezbędnych do tego, aby utworzyć nową strukturę.

```
public class SymulatorKaczek {
    // metoda main

    void uruchom(AbstrakcyjnaFabrykaKaczek kaczkaFabryka) {
        Kwacząca płaskonosKaczka = kaczkaFabryka.utwórzPłaskonosKaczka();
        Kwacząca wabikKaczka = kaczkaFabryka.utwórzWabikKaczka();
        Kwacząca gumowaKaczka = kaczkaFabryka.utwórzGumowaKaczka();
        Kwacząca gęśKaczka = new GęśAdapter(new Gęś());
        System.out.println("\nSymulator Kaczek: ze wzorcem Kompozyt - stada");

        Stado stadoKaczek = new Stado();

        stadoKaczek.dodaj(płaskonosKaczka);
        stadoKaczek.dodaj(wabikKaczka);
        stadoKaczek.dodaj(gumowaKaczka);
        stadoKaczek.dodaj(gęśKaczka);

        Stado stadoDzikichKaczek = new Stado();

        Kwacząca dzikaKaczkaJeden = kaczkaFabryka.utwórzDzikaKaczka();
        Kwacząca dzikaKaczkaDwa = kaczkaFabryka.utwórzDzikaKaczka();
        Kwacząca dzikaKaczkaTrzy = kaczkaFabryka.utwórzDzikaKaczka();
        Kwacząca dzikaKaczkaCztery = kaczkaFabryka.utwórzDzikaKaczka();

        stadoDzikichKaczek.dodaj(dzikaKaczkaJeden);
        stadoDzikichKaczek.dodaj(dzikaKaczkaDwa);
        stadoDzikichKaczek.dodaj(dzikaKaczkaTrzy);
        stadoDzikichKaczek.dodaj(dzikaKaczkaCztery);

        stadoKaczek.dodaj(stadoDzikichKaczek);

        System.out.println("\nSymulator Kaczek: Symulacja całego stada");
        uruchom(stadoKaczek);

        System.out.println("\nSymulator Kaczek: Symulacja stada dzikich kaczek");
        uruchom(stadoDzikichKaczek);

        System.out.println("\nKaczki kwaknęły " +
                           KwakLicznik.pobierzLiczbaKwaknięć() +
                           " razy");
    }

    void uruchom(Kwacząca kaczka) {
        kaczka.kwacz();
    }
}
```

Podobnie jak wcześniej, tworzymy wszystkie obiekty-liście.

Najpierw tworzymy stado, następnie „rejestrujemy" w nim wszystkie ptaki.

Potem jeszcze tworzymy stado dzikich kaczek.

Tutaj tworzymy małą rodzinkę dzikich kaczek...

... które włączamy do drugiego stada.

Następnie dodajemy stado dzikich kaczek do stada głównego.

Zobaczmy, jak działa całe stado!

Potem jeszcze sprawdźmy stado dzikich kaczek.

Na końcu wyświetlimy informację dla kwakologa.

Tutaj nic się nie zmienia, stado też ma interfejs Kwacząca!

Ruszamy...



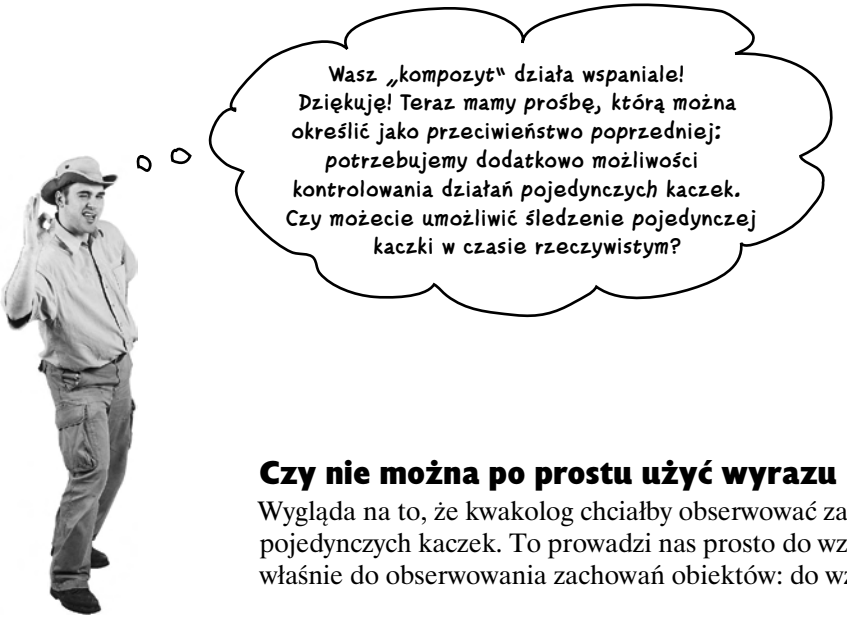


Bezpieczeństwo a przezroczystość

Być może pamiętasz, że w rozdziale o wzorcu Kompozyt elementy złożone (Menu) i elementy-liście (PozycjaMenu) miały dokładnie *ten sam* zbiór metod, łącznie z metodą dodaj(). Ten sam zestaw metod sprawiał, że część metod obiektu-liścia nie miała sensu — przede wszystkim metoda dodaj() dołączająca nową pozycję do menu. Uzyskiwaną korzyścią było to, że różnica między liśćmi i obiektami złożonymi była *niewidoczna*: klient nie musiał wiedzieć, z którym z obiektów ma do czynienia, i wywoływał te same metody.

Tutaj zdecydowaliśmy się utrzymać metody zarządzania potomkami poza węzłami-liśćmi: tylko klasa Stado ma metodę dodaj(). Nie ma sensu dodawanie czegokolwiek do obiektu Kaczka i w tej implementacji nie jest to możliwe. Kaczka nie ma metody dodaj(). Taki projekt jest *bezpieczniejszy* — nie można wywoływać nieadekwatnych dla danego typu składnika metod. Jest to zrazem projekt mniej *przezroczysty*. Aby dodać obiekt Kwacząca, klient musi wiedzieć, że dodaje do obiektu Stado.

Jest to jeden z typowych dla projektowania obiektowego wyborów. Kwestię tę należy zawsze rozważyć przy projektowaniu własnych struktur złożonych.



Czy nie można po prostu użyć wyrazu „obserwować"?

Wygląda na to, że kwakolog chciałby obserwować zachowanie pojedynczych kaczek. To prowadzi nas prosto do wzorca przeznaczonego właśnie do obserwowania zachowań obiektów: do wzorca Obserwator.

14 Najpierw musimy przygotować interfejs Obserwowany.

Pamiętamy, że obiekt Obserwowany to obiekt obserwowany. Wymaga on metod do rejestrowania i powiadamiania obserwatorów. Możemy też mieć metodę do usuwania obserwatorów. Dla uproszczenia pominiemy ten element.

KwakObserwowany to interfejs, który muszą implementować wszystkie śledzone obiekty Kwacząca.

```
public interface KwakObserwowany {
    public void zarejestrujObserwatora(Obserwator obserwator);
    public void powiadomObserwatorów();
}
```

Interfejs ma metodę do rejestrowania obiektów Obserwator. Każdy obiekt implementujący interfejs Obserwator może oczekiwać na kwaknięcia. Definicją tego drugiego interfejsu zajmiemy się za chwilę.

Ma też metodę do powiadamiania obiektów Obserwator.

Musimy zadbać o to, aby wszystkie kaczki implementowały ten interfejs...

```
public interface Kwacząca extends KwakObserwowany {
    public void kwacz();
}
```

A więc deklarujemy, że interfejs Kwacząca będzie rozszerzeniem KwakObserwowany.



15 Musimy teraz zadbać o to, żeby wszystkie klasy rzeczywiste, które implementują interfejs Kwacząca, faktycznie implementowały metody interfejsu KwakObserwowany.

Możemy rozwiązać ten problem, implementując rejestrowanie i powiadamianie w każdej kolejnej klasie (tak jak to zrobiliśmy w rozdziale 2.). Tym razem jednak pójdziemy nieco inną drogą — drogą hermetyzacji rejestracji i powiadamiania w osobnej klasie nazwanej standardowo Obserwowany. Będzie to implementacja interfejsu KwakObserwowany. W ten sposób odpowiedni kod napiszemy tylko raz. Implementacje interfejsu KwakObserwowany będą wymagały minimum uzupełnień — tylko tyle, żeby zapewnić delegowanie do klasy pomocniczej Obserwowany.

Spójrzmy na kod klasy pomocniczej, Obserwowany…

Klasa Obserwowany implementuje wszystkie funkcje, których wymaga obiekt Kwacząca, aby być implementacją klasy KwakObserwowany. Później powiążemy to z klasą delegującą.

Klasa Obserwowany musi implementować interfejs KwakObserwowany, ponieważ zawiera on te same metody, które będziemy tutaj delegować.

```
public class Obserwowany implements KwakObserwowany {
    ArrayList obserwatorzy = new ArrayList();
    KwakObserwowany kaczka;

    public Obserwowany(KwakObserwowany kaczka) {
        this.kaczka = kaczka;
    }

    public void zarejestrujObserwatora(Obserwator obserwator) {
        obserwatorzy.add(obserwator);
    }

    public void powiadomObserwatorów() {
        Iterator iterator = obserwatorzy.iterator();
        while (iterator.hasNext()) {
            Obserwator obserwator = (Observer)iterator.next();
            obserwator.aktualizuj(kaczka);
        }
    }
}
```

W konstruktorze pobieramy obiekt KwakObserwowany, który będzie wykorzystywał tworzony obiekt Obserwowany do obsługi części zachowań. Zwróć uwagę na metodę powiadamiającą, widoczną poniżej. Jak widać, przy powiadamianiu obserwatorów metoda powiadamiająca przekazuje obiekt obserwowany. Dzięki temu obserwator wie, który obiekt kwacze.

Tu mamy kod rejestracji obserwatora.

I kod powiadamiania.

A teraz zobaczmy, w jaki sposób klasa Kwacząca używa zdefiniowanej tu klasy pomocniczej…

16 Integrujemy klasy Kwacząca i Obserwowany.

To nie powinno być takie straszne. Wszystko, czego potrzebujemy, to zapewnić, że klasy Kwacząca będą wykorzystywały klasę Obserwowany. To sprawi, że będą poprawnymi implementacjami interfejsu KwakObserwowany. Oto implementacja klasy DzikaKaczka. Pozostałe są podobne.

```
public class DzikaKaczka implements Kwacząca {
    Obserwowany obserwowany;

    public DzikaKaczka() {
        obserwowany = new Obserwowany(this);
    }

    public void kwacz() {
        System.out.println("Kwa! Kwa!");
        powiadomObserwatorów();
    }

    public void zarejestrujObserwatora(Obserwator obserwator) {
        obserwowany.zarejestrujObserwatora(obserwator);
    }

    public void powiadomObserwatorów() {
        obserwowany.powiadomObserwatorów();
    }
}
```

Każda implementacja interfejsu Kwacząca będzie miała zmienną egzemplarza typu Obserwowany.

W konstruktorze, tworzymy obiekt Obserwowany i przekazujemy mu odwołanie do obiektu DzikaKaczka.

W trakcie wykonywania metody kwacz(), powiadamiamy obserwatorów.

To są dwie metody wymagane przez interfejs KwakObserwowany. Zwróć uwagę, że korzystają one z delegacji do klasy Obserwowany.



Zaostrz ołówek

Nie zmieniliśmy jeszcze implementacji ostatniej klasy typu Kwacząca, KwakLicznik. Ona również musi implementować interfejs KwakObserwowany. Może napiszesz nową wersję samodzielnie:

⑰ Prawie skończyliśmy! Pozostaje strona obserwatora.

Zaimplementowaliśmy już wszystkie elementy niezbędne dla klas obserwowanych. Czas zająć się obserwatorami. Rozpoczniemy od interfejsu Obserwator:

Interfejs Obserwator ma tylko jedną metodę, aktualizuj(), która pobiera kwaczący w danym momencie obiekt KwakObserwowany.

```
public interface Obserwator {
    public void aktualizuj(KwakObserwowany kaczka);
}
```

Teraz potrzebujemy obserwatora. Gdzie są ci kwakolodzy?!

Musimy zaimplementować interfejs Obserwowany, aby móc zarejestrować obserwatora w obiekcie obserwowanym, KwakObserwowany.

```
public class Kwakolog implements Obserwator {

    public void aktualizuj(KwakObserwowany kaczka) {
        System.out.println("Kwakolog: " + kaczka + " kwaknęła.");
    }
}
```

Kwakolog jest prosty. Ma tylko jedną metodę, aktualizuj(), która wyświetla informację o ptaku, który kwaknął.



Zaostrz ołówek

A gdyby kwakolog chciał śledzić całe stado? Co to dla nas oznacza? Możemy to potraktować w następujący sposób: jeżeli obserwujemy obiekt złożony, obserwujemy wszystkie *jego* elementy składowe. A więc po zarejestrowaniu obserwatora stada obiekt Stado dba o rejestrację tego obserwatora u wszystkich potomków. Potomkami są kaczki lub mniejsze stada (rodziny kaczek).

Zanim przejdziemy do kolejnego etapu naszych rozważań, spróbuj napisać kod obserwatora stada...

18 **Jesteśmy już gotowi do obserwowania kaczek. Zaktualizujmy więc symulator, aby uruchomić go w nowej konfiguracji:**

```
public class SymulatorKaczek {
    public static void main(String[] args) {
        SymulatorKaczek symulator = new SymulatorKaczek();
        AbstrakcyjnaFabrykaKaczek kaczkaFabryka = new FabrykaKaczekZLicznikiem();

        symulator.uruchom(kaczkaFabryka);
    }

    void uruchom(AbstrakcyjnaFabrykaKaczek kaczkaFabryka) {

        // tworzenie fabryk i kaczek

        // tworzenie stad

        System.out.println("\nSymulator Kaczek: z wzorcem Obserwator");

        Kwakolog kwakolog = new Kwakolog();
        stadoKaczek.zarejestrujObserwatora(kwakolog);

        uruchom(stadoKaczek);

        System.out.println("\nKaczki kwaknęły " +
                           KwakLicznik.pobierzLiczbaKwaknięć() +
                           " razy");
    }

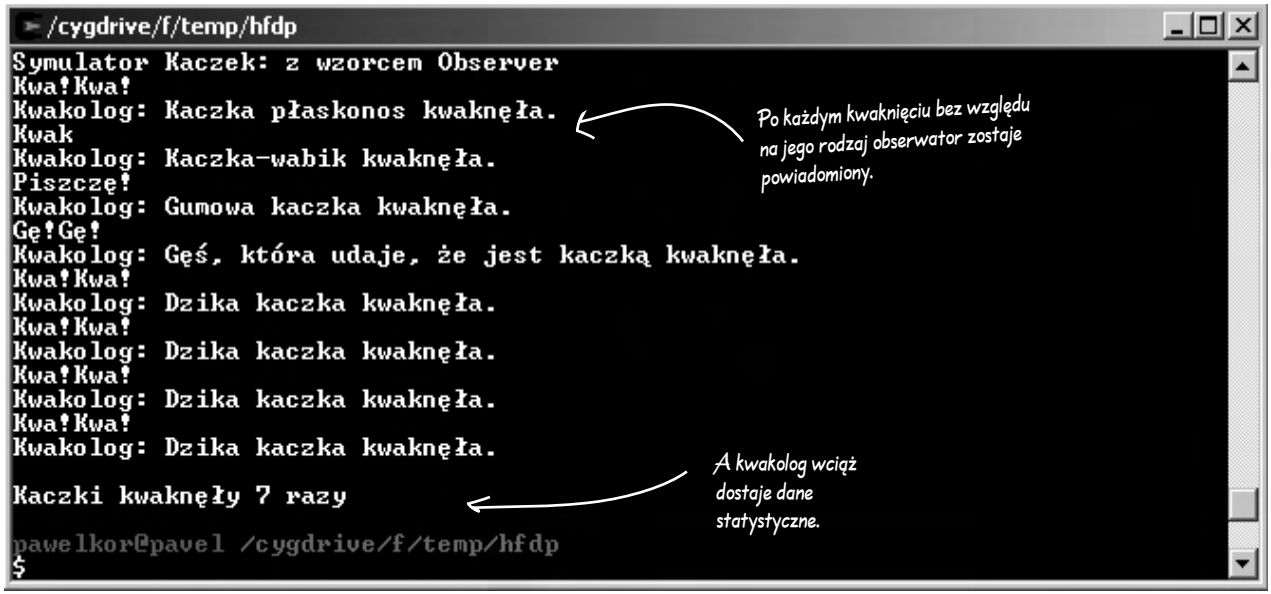
    void uruchom(Kwacząca kaczka) {
        kaczka.kwacz();
    }
}
```

Tutaj tworzymy egzemplarz kwakologa i rejestrujemy go w charakterze obserwatora stada.

Tym razem uruchomimy symulację tylko raz, dla całego stada.

Przekonajmy się, jak działa nowy kod!

Na tym skończymy zabawę z kaczkami. Połączyliśmy pięć… nie, sześć wzorców we wspaniały symulator. Nie zwlekając już ani chwili dłużej, prezentujemy SymulatorKaczek!



Nie istnieją głupie pytania

P: Czy to był wzorzec złożony?

O: Nie, to był przykład wykorzystania wielu różnych wzorców w jednym projekcie. Wzorzec złożony to zespół kilku wzorców połączonych w taki sposób, aby rozwiązywać pewien ogólny problem. Już za chwilę zapoznamy się z wzorcem złożonym Model-Widok-Kontroler. Jest to właśnie połączenie kilku wzorców wykorzystywane w wielu różnych projektach.

P: A więc prawdziwy urok wzorców projektowych polega na tym, że można stosować je kolejno w trakcie tworzenia aplikacji aż do momentu uzyskania gotowego rozwiązania, tak?

O: Nie. Przykład z kaczkami miał zademonstrować, że wzorce *mogą* być wykorzystywane razem. W praktyce rozwiązywanie problemu nigdy nie będzie wyglądać tak, jak to właśnie przedstawiliśmy. Istnieje wiele możliwości implementacji symulatora kaczek. Można wręcz stwierdzić, że w porównaniu z nimi wprowadzenie niektórych z użytych przez nas wzorców było tylko niepotrzebną stratą czasu.

W pewnych przypadkach same zasady projektowania obiektowego mogą doprowadzić nas do bardzo dobrego rozwiązania problemu.

Tego rodzaju dylematami zajmiemy się jeszcze szerzej w kolejnym rozdziale. Podstawową zasadą jest to, że wzorce projektowe stosujemy wtedy i tylko wtedy, gdy jest to uzasadnione i korzystne. Nie możemy rozpoczynać pracy z myślą o używaniu wzorców dla nich samych. Przedstawiony projekt symulatora kaczek jest „naciągany" i nienaturalny. Miał być zabawny i posłużyć do zilustrowania możliwości stosowania wielu wzorców w jednym rozwiązaniu.

Co zrobiliśmy?

Rozpoczęliśmy od kilku obiektów Kwacząca...

Potem pojawiła się Gęś i też chciała być Kwacząca. Użyliśmy wzorca *Adapter*, aby zaadaptować ją do interfejsu Kwacząca. Dzięki temu mogliśmy wywoływać metodę kwacz() osłony gęsi (adaptera) i uzyskiwaliśmy gęganie!

Potem kwakolodzy zdecydowali, że wymagają liczenia kwaknięć. Użyliśmy więc wzorca *Dekorator* i stworzyliśmy klasę KwakLicznik, która zlicza wywołania metody kwacz() i deleguje jej wykonanie do osłanianego obiektu Kwacząca.

Kwakolodzy ciągle zapominali o dodawaniu osłony zliczającej kwaknięcia. Użyliśmy więc wzorca *Fabryka Abstrakcyjna*, aby ukryć operacje tworzenia ptaków z osłonami. Po takiej zmianie, jeśli kiedykolwiek potrzebna jest kaczka, pobieramy ją z fabryki. Fabryka zwraca obiekt odpowiednio osłonięty. Wciąż możemy korzystać z innej fabryki, zwracającej kaczki, których kwaknięcia nie są zliczane.

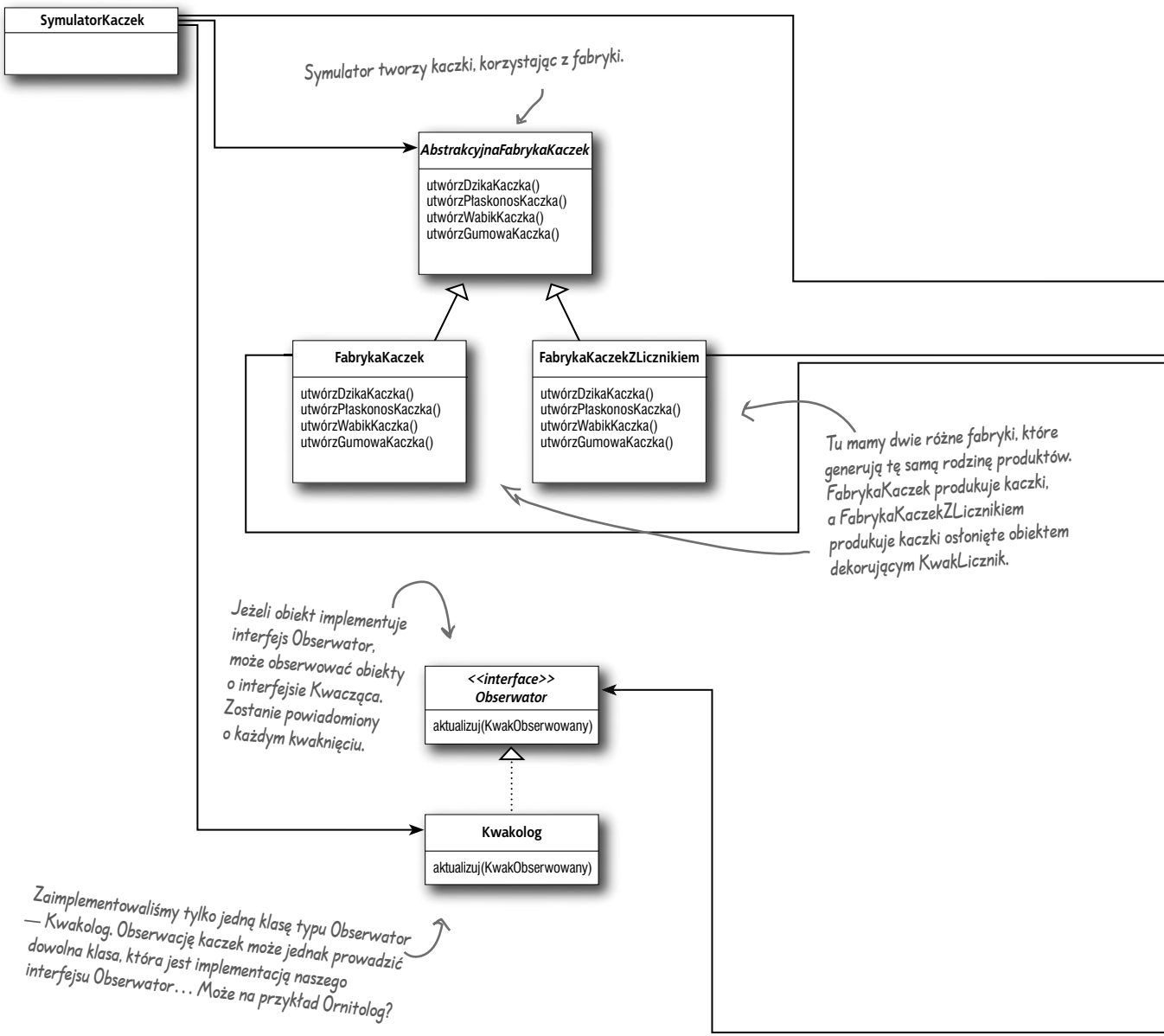
Pojawiły się problemy w zarządzaniu wszystkimi kaczkami, gęsiami i obiektami Kwacząca. Użyliśmy więc wzorca *Kompozyt*, aby połączyć ptaki w stada. Ten wzorzec umożliwił dodatkowo utworzenie rodzin kaczek, czyli mniejszych grup w stadzie. Użyliśmy też wzorca *Iterator*, pobierając standardowy iterator kolekcji ArrayList.

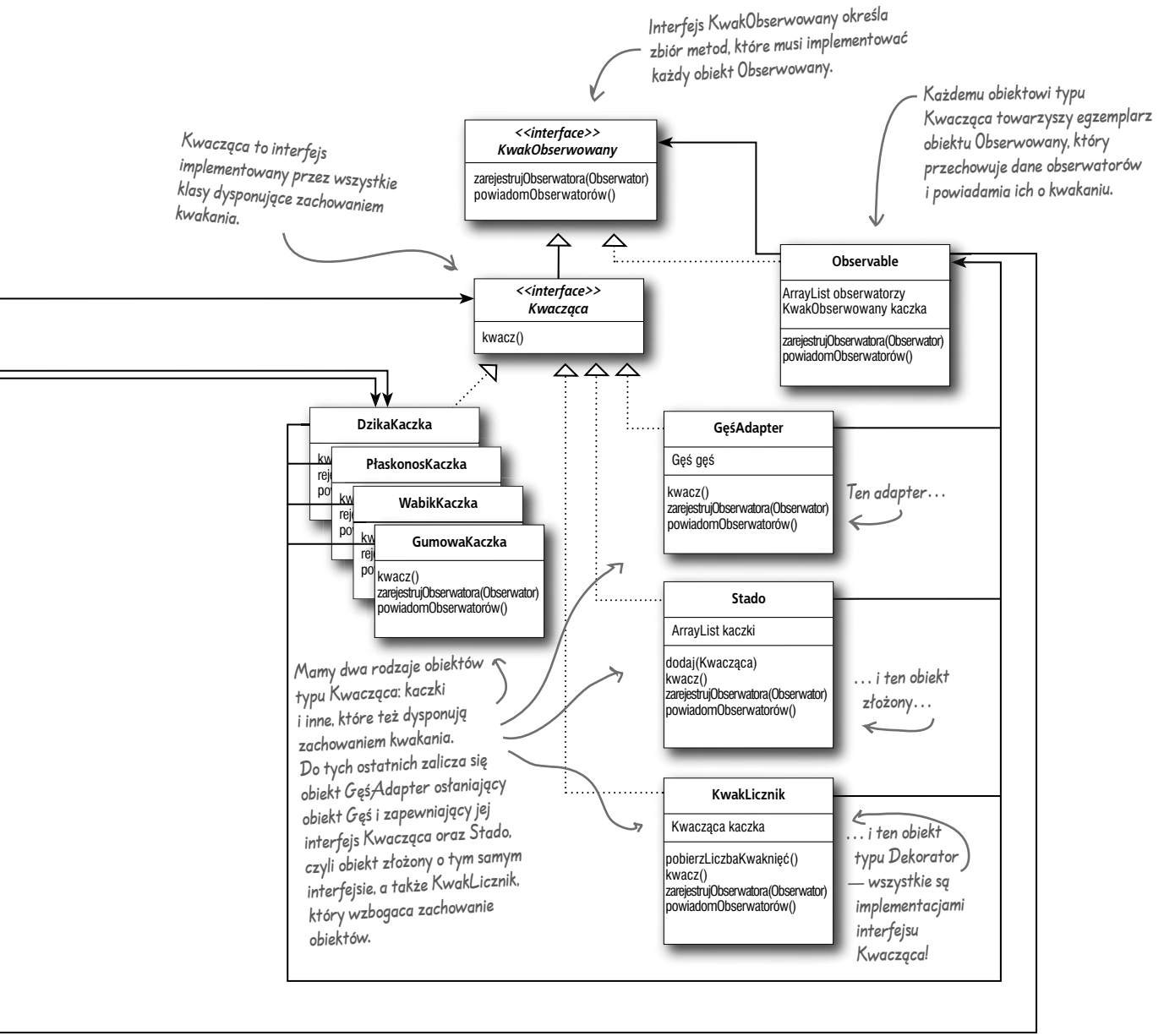
Kwakolodzy wymagali też powiadamiania o każdym kwaknięciu. Użyliśmy wzorca *Obserwator*, aby umożliwić rejestrowanie obiektów Kwakolog (jako obiektów Obserwator) w obiektach Kwacząca. Dzięki temu Kwakolog jest powiadamiany o każdym kwaknięciu. Ponownie skorzystaliśmy z iteratora. Kwakolog może również skorzystać z możliwości jednorazowej rejestracji w obiekcie złożonym (stadzie).



Widok z lotu ~~ptaka~~ kaczki — diagram klas

W niewielkim symulatorze kaczek wykorzystaliśmy mnóstwo wzorców!
Oto kompletny diagram naszej aplikacji:




Interfejs KwakObserwowany określa zbiór metod, które musi implementować każdy obiekt Obserwowany.
Każdemu obiektowi typu Kwacząca towarzyszy egzemplarz obiektu Obserwowany, który przechowuje dane obserwatorów i powiadamia ich o kwakaniu.
Kwacząca to interfejs implementowany przez wszystkie klasy dysponujące zachowaniem kwakania.
<<interface>>
KwakObserwowany
zarejestrujObserwatora(Obserwator)
powiadomObserwatorów()
Observable
ArrayList obserwatorzy
KwakObserwowany kaczka
zarejestrujObserwatora(Obserwator)
powiadomObserwatorów()
<<interface>>
Kwacząca
kwacz()
DzikaKaczka
PłaskonosKaczka
WabikKaczka
GumowaKaczka
kwacz()
zarejestrujObserwatora(Obserwator)
powiadomObserwatorów()
GęśAdapter
Gęś gęś
kwacz()
zarejestrujObserwatora(Obserwator)
powiadomObserwatorów()
Ten adapter...
Stado
ArrayList kaczki
dodaj(Kwacząca)
kwacz()
zarejestrujObserwatora(Obserwator)
powiadomObserwatorów()
... i ten obiekt złożony...
KwakLicznik
Kwacząca kaczka
pobierzLiczbaKwaknięć()
kwacz()
zarejestrujObserwatora(Obserwator)
powiadomObserwatorów()
... i ten obiekt typu Dekorator — wszystkie są implementacjami interfejsu Kwacząca!
Mamy dwa rodzaje obiektów typu Kwacząca: kaczki i inne, które też dysponują zachowaniem kwakania. Do tych ostatnich zalicza się obiekt GęśAdapter osłaniający obiekt Gęś i zapewniający jej interfejs Kwacząca oraz Stado, czyli obiekt złożony o tym samym interfejsie, a także KwakLicznik, który wzbogaca zachowanie obiektów.

Król wzorców złożonych

Gdyby Elvis był wzorcem złożonym, nazywałby się Model-Widok-Kontroler i śpiewałby taką piosenkę...

Model-Widok-Kontroler

Słowa i muzyka: James Dempsey

MVC to paradygmat porządkowania kodu według

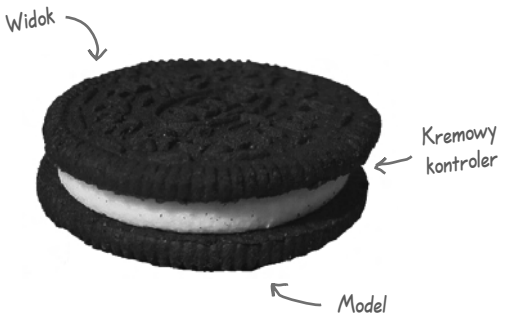
jego funkcji w celu uzyskania większej przejrzystości.

Aby uzyskać możliwość wielokrotnego wykorzystania tego samego kodu,

musisz zachować wyraźne granice między tymi częściami.

Model po jednej stronie, widok (ang. view) po drugiej,

a kontroler (ang. controller) w środku.



Model-Widok-Kontroler ma trzy warstwy, jak markizy

Model-Widok-Kontroler

Model-Widok Model-Widok Model-Widok-Kontroler

Obiekty modelu to racja bytu aplikacji

Różne obiekty z danymi, logiką i tak dalej

Tworzysz te klasy w dziedzinie problemu aplikacji;

możesz użyć ich z wszystkimi widokami.

Obiekty modelu nie zmienią się.

Możesz modelować przepustnicę albo rozmaitość

Modelować dreptanie dwulatka

Modelować butelkę świetnego Chardonnay

Modelować wszystkie wymawiane głoski

Modelować ruch gotujących się jajek

Możesz modelować drżenie nóg Hexleya

Model-Widok, możesz modelować wszystkie modele, które pozują w FTV

Model-Widok-Kontroler

I Java też!

Obiekty widoku to często elementy sterujące używane do wyświetlania i modyfikowania danych

Cocoa ma ich dużo, znanych i uznanych.

Weź NSTextView, daj mu dowolny ciąg Unicode

Użytkownik może robić z nim, co chce, może przechowywać niemal wszystko

Ale widok nie wie o modelu

Ten ciąg może być numerem telefonu albo pracami Arystotelesa

Utrzymuj separację

a uzyskasz niezwykłą swobodę wielokrotnego użycia

Model-Widok, wszystko pięknie w „aqua-blue"

Model-Widok-Kontroler

Zastanawiasz się pewnie teraz, jak

Zastanawiasz się pewnie teraz, jak

Dane przepływają między modelem a widokiem

Kontroler w tym pośredniczy

Pomiędzy zmiennymi stanami warstw

Synchronizuje dane obu
Pobiera i przekazuje każdą zmienioną wartość

Model-Widok, szalone podpory załogi SmallTalka!
Model-Widok-Kontroler
Model-Widok, wymawiane oooo-ooo-jeee-ooo
Model-Widok-Kontroler

Historia już się prawie kończy
Zostało tylko kilka chwil
Nikt chyba nie zyskuje sławy
Głowiąc się nad kodem kontrolera

Bo model jest najważniejszy
A widok najładniejszy
Mogę być leniwy, ale czasem to wariactwo
Ile kodu nie napiszę, to tylko klej
I nie byłoby to tragedią
Ale ten kod nie robi nic wielkiego
On po prostu przenosi wartości

I nie chcę być złośliwy
Ale to zaczyna być monotonne
Robienie tego wszystkiego, co robią kontrolery

Szkoda, że nie dostaję miedziaka
Za każdy jeden raz
Kiedy przesyłam StringValue pola tekstowego.

Model-Widok
Jak sobie poradzimy z tym klejem
Model-Widok-Kontroler

Kontrolery znają model i widok bardzo dobrze
Często opierają się na klasach rzeczywistych
co nie ułatwia ponownego użycia
Ale dzięki nim możesz połączyć dowolny klucz modelu
z dowolną właściwością widoku

A kiedy już zaczniesz wiązanie
Zacznie się kodu ubywanie

Tak, wiem, że teraz to automatyzują
i wszystko dostajesz za darmo

Ale myślę, że warto powtarzać
cały ten kod, którego nie będziesz potrzebować
gdy powiążesz to z ~~JB~~. Używając biblioteki Swing.
Model-Widok, obsługuje nawet wybieranie wielu elementów
Model-Widok-Kontroler
Model-Widok, założę się, że moja aplikacja powstanie szybciej
Model-Widok-Kontroler



Posłuchajmy

Nie ograniczaj się do czytania! Było nie było, to książka z serii *Rusz głową!*...
Weź swojego iPoda i uderz pod ten adres:
http://www.wickedlysmart.com/headfirstdesignpatterns/media.html
Usiądź, odpręż się i posłuchaj.



Nie. Kluczem do schematu MVC będą wzorce projektowe.

Próbujemy tylko zaostrzyć Twój apetyt. Właściwie, jak skończysz czytać ten rozdział, wróć do początku i posłuchaj piosenki jeszcze raz — będzie znacznie więcej zabawy.

Wygląda na to, że masz już złe doświadczenia z wzorcem Model-Widok-Kontroler? Większość ludzi je miała. Inni programiści pewnie opowiadali Ci o tym, jak zmienił ich życie i mógłby sprawić, by zapanował pokój na Ziemi. To faktycznie wzorzec złożony o wielkich możliwościach i choć nie możemy zagwarantować, że przyniesie światu pokój, na pewno zaoszczędzi wielu godzin pracy z programem.

Ale najpierw trzeba się go nauczyć, prawda? Tym razem będzie to wyglądało zupełnie inaczej — *teraz znasz wzorce projektowe*!

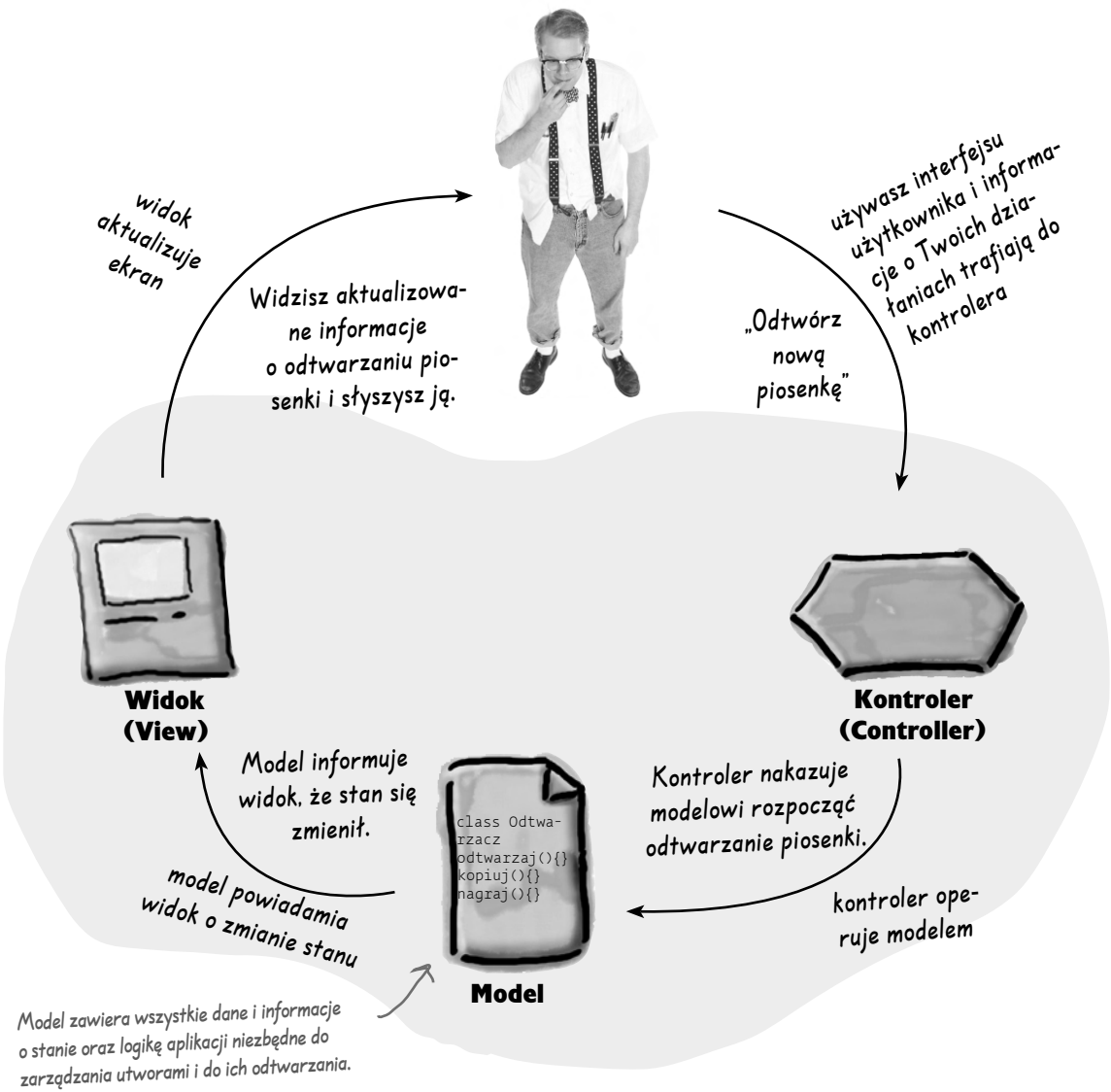
To prawda, wzorce są kluczem do zrozumienia tego schematu. Nauka metodą ”góra-dół”, czyli stopniowe zagłębianie się w temat nie sprawdza się w tym przypadku. Oto klucz: *Model-Widok-Kontroler to po prostu kilka zestawionych razem wzorców*. Gdy rozpoczynamy naukę o nim, mając cały czas na uwadze poznane wzorce, wszystko od razu układa się w zrozumiałą całość.

Czas zaczynać. Tym razem przygwoździmy MVC!

Poznajemy wzorzec Model-Widok-Kontroler

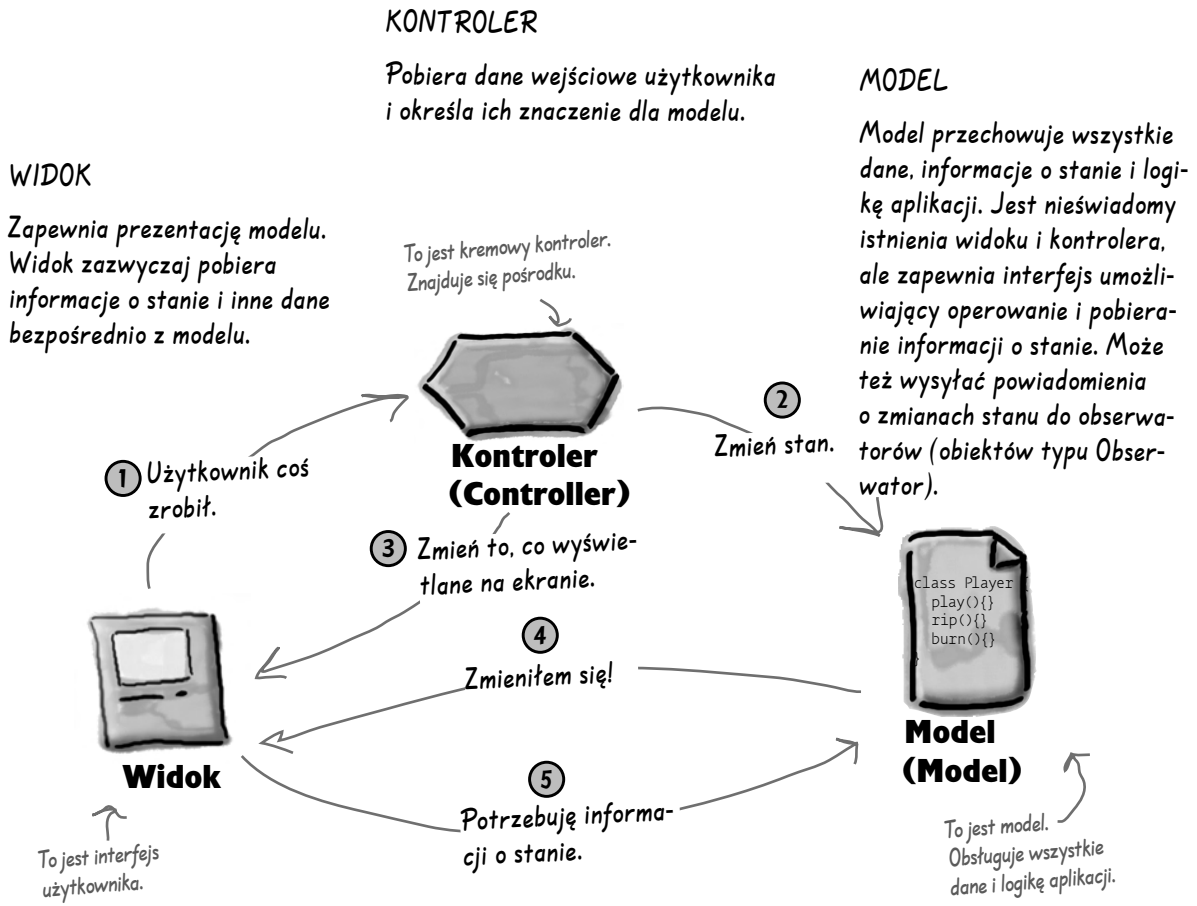
Wyobraź sobie, że używasz odtwarzacza MP3, takiego jak iTunes. Jego interfejs pozwala dodawać nowe piosenki, zarządzać listami odtwarzania i zmieniać nazwy ścieżek. Odtwarzacz dysponuje małą bazą danych, w której zapisane są wszystkie informacje o piosenkach. Umożliwia też odtwarzanie utworów. W trakcie odtwarzania dane interfejsu użytkownika są nieustannie aktualizowane — wyświetlana jest nazwa piosenki, czas od jej początku (albo pozostały do końca) i inne podobne rzeczy.

A za tym wszystkim stoi system Model-Widok-Kontroler…



Przyjrzyjmy się bliżej...

Opis odtwarzacza MP3 prezentuje nam ogólny sposób działania schematu MVC, ale nie pomaga zrozumieć, na czym dokładnie polega wzorzec złożony, jak taki wzorzec stworzyć ani dlaczego jest to tak dobre rozwiązanie. Przyjmy się więc dokładniej relacjom między modelem, widokiem i kontrolerem. Dopiero w kolejnym kroku zobaczymy, jak to wszystko wygląda od strony wzorców projektowych.



1. **Jesteś użytkownikiem — wchodzisz w interakcje z widokiem.**
Widok to okno na model. Gdy wchodzimy w jakiekolwiek interakcje z widokiem (na przykład wciskamy przycisk „Play"), widok informuje kontroler o tym, co zrobił użytkownik. Zadaniem kontrolera jest zapewnić odpowiednią obsługę.

2. **Kontroler żąda od modelu zmiany stanu.**
Kontroler przyjmuje informacje o czynnościach użytkownika i interpretuje je. Jeżeli klikniemy przycisk, określenie znaczenia takiej czynności jest właśnie zadaniem kontrolera. To on określa, jakie operacje powinny zostać wykonane na modelu.

3. **Kontroler może zażądać zmiany widoku.**
Jednym z elementów obsługi czynności użytkownika może być przekazanie do widoku żądania zmiany. Przykładowo, kontroler może uaktywnić lub zdezaktywować pewne przyciski lub elementy menu.

4. **Model powiadamia widok, gdy jego stan ulegnie zmianie.**
Gdy coś zmienia się w modelu w wyniku akcji użytkownika (takiej jak kliknięcie przycisku) lub innej zmiany wewnętrznej (takiej jak rozpoczęcie odtwarzania kolejnego utworu), model powiadamia widok o zmianie stanu.

5. **Widok żąda od modelu informacji o stanie.**
Widok pobiera wyświetlane informacje o stanie bezpośrednio z modelu. Przykładowo, gdy model powiadamia widok o tym, że rozpoczęto odtwarzanie nowego utworu, widok pobiera z modelu nazwę piosenki i wyświetla ją. Widok może też żądać od modelu informacji o stanie, gdy kontroler żąda od niego wprowadzenia pewnej zmiany w wyświetlanych danych.

Nie istnieją głupie pytania

P: Czy kontroler kiedykolwiek staje się obserwatorem modelu?

O: Oczywiście. W niektórych projektach kontroler rejestruje się w modelu i jest powiadamiany o zmianach. Z takim przypadkiem będziemy mieli do czynienia między innymi wtedy, gdy pewien element modelu bezpośrednio wpływa na elementy sterujące interfejsu użytkownika. Przykładowo, pewne stany modelu mogą narzucać aktywację lub deaktywację różnych elementów interfejsu. Gdy tak się dzieje, przekazanie widokowi żądania uaktualnienia wyświetlanych danych będzie zadaniem kontrolera.

P: Wszystko, co robi kontroler, to pobranie danych wejściowych użytkownika od widoku i przesłanie ich do modelu, tak? Po co w ogóle wprowadzać taki element, jeżeli to jest jego jedyne zadanie? Dlaczego nie umieścić odpowiedniego kodu w samym widoku? Czy w większości przypadków zadanie kontrolera nie ogranicza się do wywołania metody modelu?

O: Rola kontrolera to nie tylko „przesłanie wszystkiego do modelu". Kontroler odpowiada za interpretację danych wejściowych i wykonanie odpowiednich operacji na modelu.

Właściwym pytaniem jest zapewne: „Dlaczego nie mogę zrobić tego w kodzie widoku?".

Możesz. Są jednak dwa powody, dla których warto tego uniknąć: po pierwsze, kod widoku stanie się bardziej złożony, ponieważ będzie miał dwa zakresy odpowiedzialności: zarządzanie interfejsem użytkownika i zajmowanie się logiką sterowania modelem. Po drugie, doprowadzimy do ścisłego powiązania widoku z modelem. Gdy pojawi się potrzeba wykorzystania widoku z innym modelem, znaczną ilość kodu trzeba będzie napisać od nowa. Kontroler oddziela logikę sterowania od widoku i separuje oba elementy: widok i model. Utrzymując luźne powiązanie widoku z kontrolerem tworzymy projekt elastyczny i łatwy w rozbudowie — taki, który można przystosować do przyszłych zmian.

Spojrzenie na schemat Model-Widok-Kontroler przez pryzmat wzorców



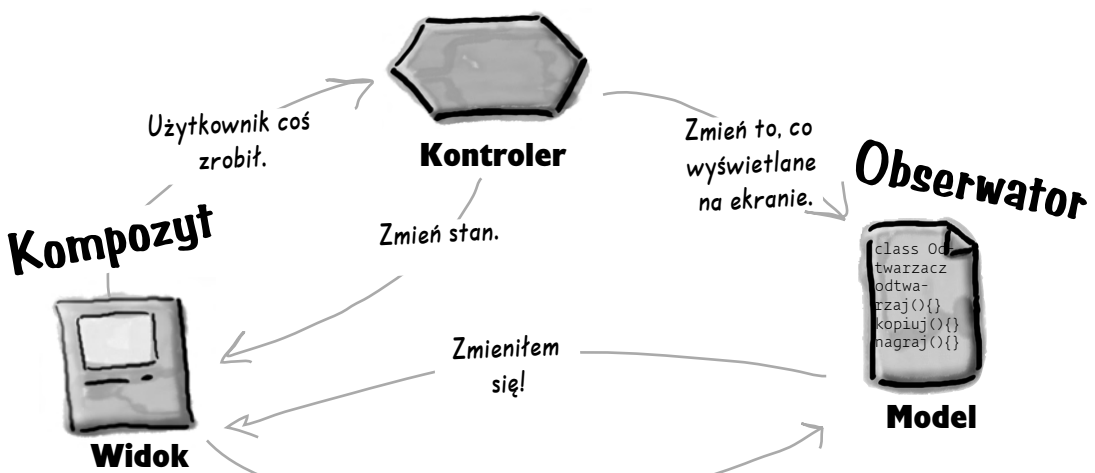
Jak powiedzieliśmy na początku, najlepszą drogą do poznania schematu MVC jest zobaczenie w nim tego, czym tak naprawdę jest: zbioru współpracujących w jednym projekcie wzorców.

Rozpocznijmy od modelu. Jak może się domyśliłeś, model wykorzystuje wzorzec Obserwator do aktualizowania widoków i kontrolerów odpowiednio do zmian swojego stanu. Z drugiej strony, widok i kontroler stosują wzorzec Strategia. Kontroler jest zachowaniem widoku. Można je łatwo zmienić, stosując inny kontroler. Sam widok również stosuje wewnętrznie pewien wzorzec ułatwiający mu zarządzanie oknami, przyciskami i innymi elementami wyświetlanymi na ekranie — jest to wzorzec Kompozyt.

Przyjrzyjmy się temu dokładnie:

Strategia

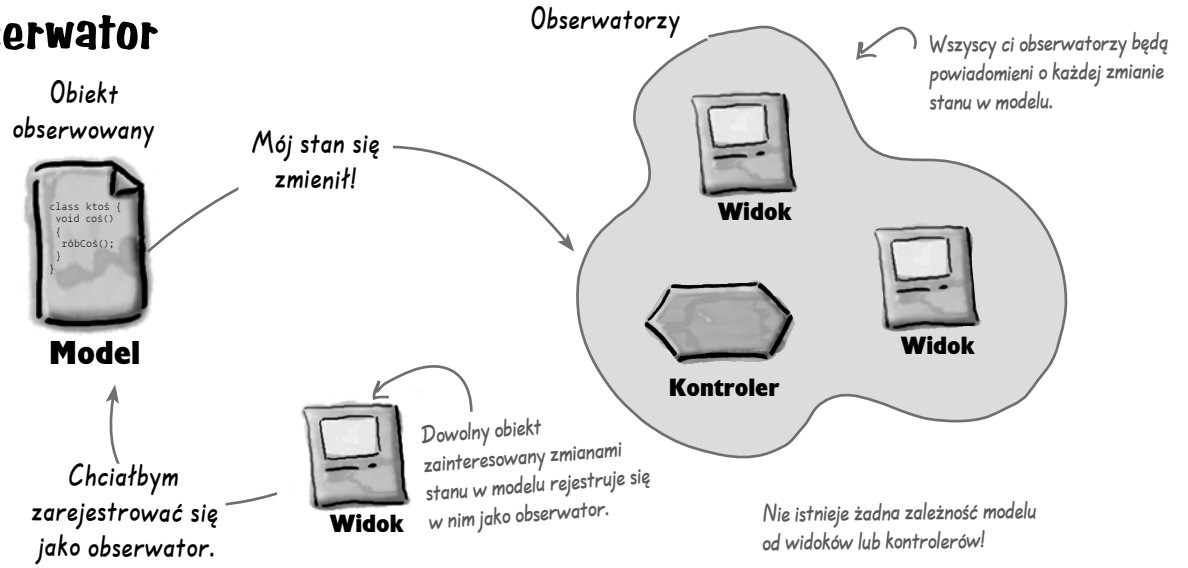
Widok i kontroler stosują klasyczny wzorzec Strategia: widok jest obiektem konfigurowanym przy użyciu strategii. Kontroler zapewnia tę strategię. Widok zajmuje się wyłącznie wizualnym aspektem aplikacji i deleguje do kontrolera wszystkie decyzje związane z zachowaniem interfejsu. Wykorzystanie wzorca Strategia zapewnia też separację widoku i modelu, ponieważ to kontroler odpowiada za interakcje z modelem w trakcie wykonywania żądań użytkownika. Widok nie wie nic o tym, w jaki sposób są one realizowane.



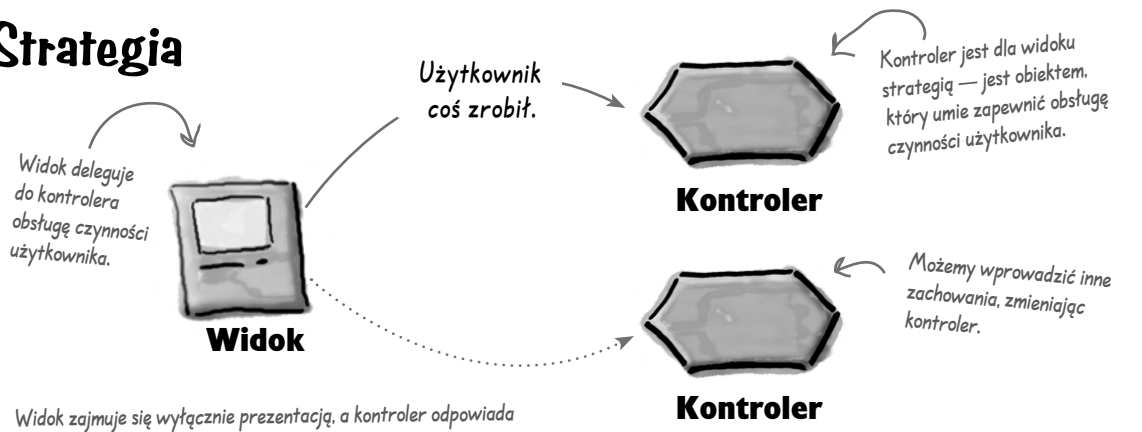
Wyświetlane na ekranie dane to zagnieżdżone zbiory okien, paneli, przycisków, etykiet i innych elementów. Każdy wyświetlany element to obiekt złożony (jak okno) lub obiekt-liść (jak przycisk). Gdy kontroler nakazuje widokowi aktualizację, informuje o niej wyłącznie główny element, a on, jako obiekt złożony, dba o szczegóły operacji.

Aby zapewnić informowanie zainteresowanych obiektów o zmianach stanu, model wykorzystuje wzorzec Obserwator. Zastosowanie tego wzorca zapewnia całkowitą niezależność modelu od widoków i kontrolerów. Umożliwia wykorzystywanie różnych widoków z tym samym modelem, a nawet wielu widoków jednocześnie.

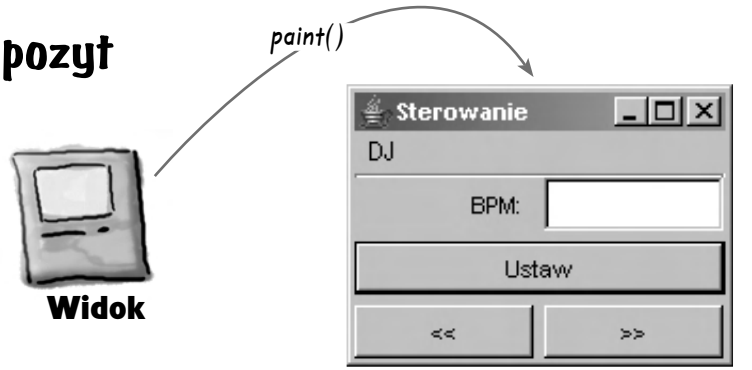
Obserwator



Strategia



Kompozyt



Widok jest kompozytem elementów graficznego interfejsu użytkownika (etykiet, przycisków, pól tekstowych itp.). Element główny zawiera inne elementy, które zawierają kolejne elementy i tak dalej, aż do elementów-liści.

Wykorzystujemy MVC do sterowania beatem...

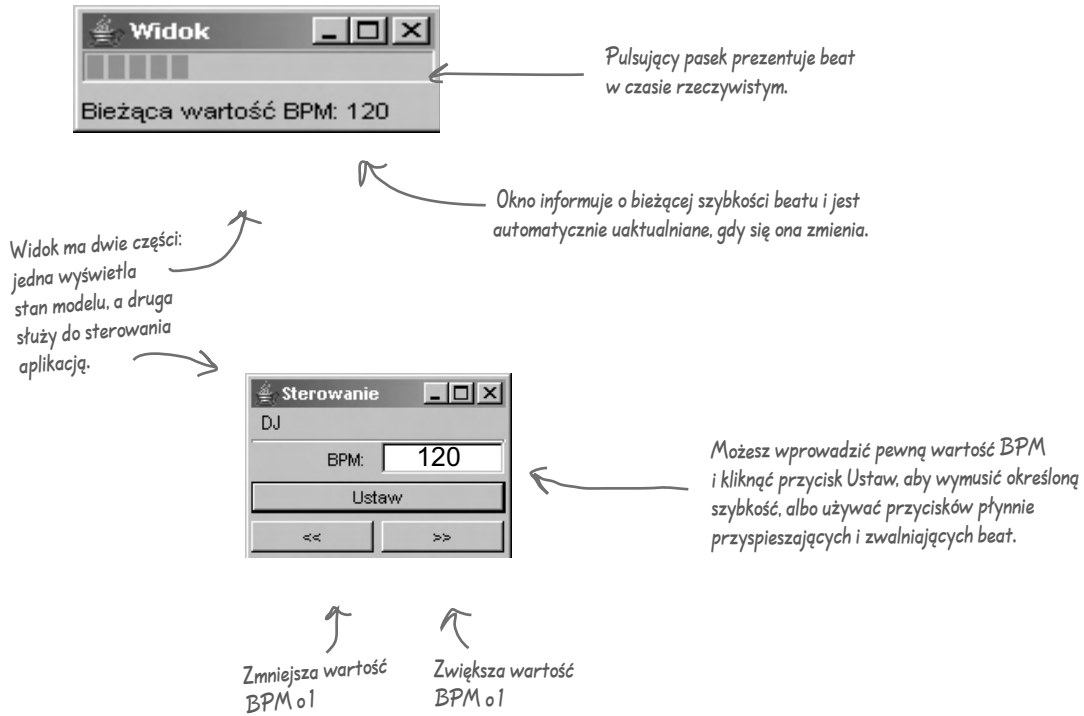


Czas zostać didżejem. Gdy jesteś didżejem, beat jest najważniejszy. Można rozpocząć miks od powolnego 95 BPM (ang. *beats per minute*, liczba uderzeń na minutę) i wznosić się do frenetycznego, transowego 140 BPM. Set może zakończyć się ambientowym miksem o szybkości 80 BPM.

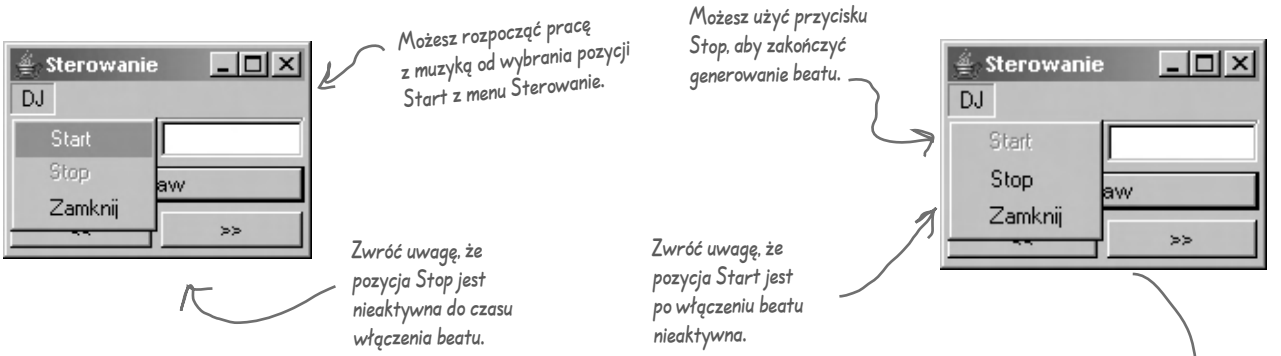
Jak to zrobić? Musisz mieć możliwość sterowania beatem. Zbudujemy teraz narzędzie, które da nam tę możliwość.

Java DJ View

Rozpocznijmy od **widoku** narzędzia. Widok umożliwi stworzenie wciągającego beatu i sterowanie jego szybkością, czyli BPM…

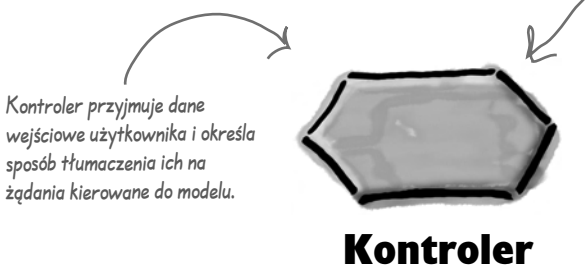


Istnieje jeszcze kilka innych elementów sterujących widokiem didżeja…



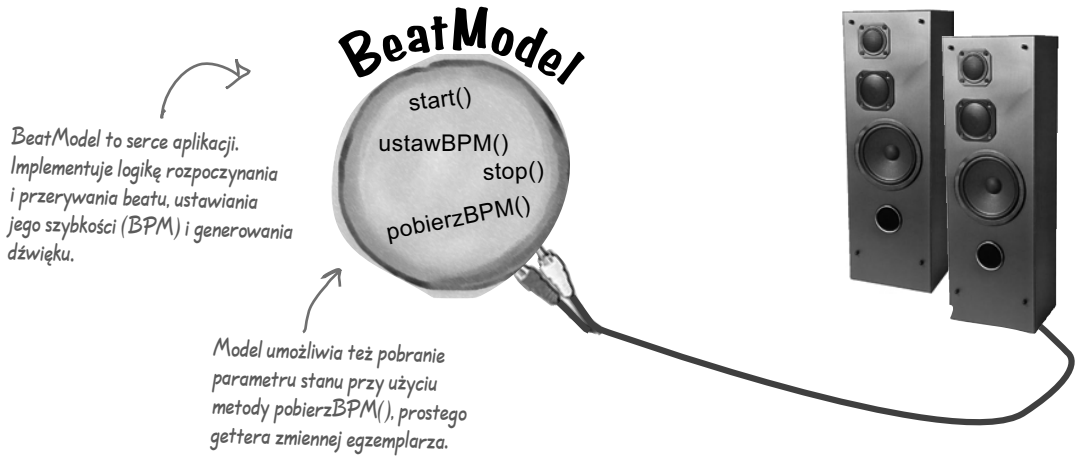
Kontroler jest w środku...

Kontroler pracuje pomiędzy widokiem a modelem. Pobiera dane wejściowe użytkownika, takie jak informacja o kliknięciu przycisku Start w menu Sterowanie, i przekształca je na taką operację na modelu, która na przykład rozpoczyna generowanie beatu.

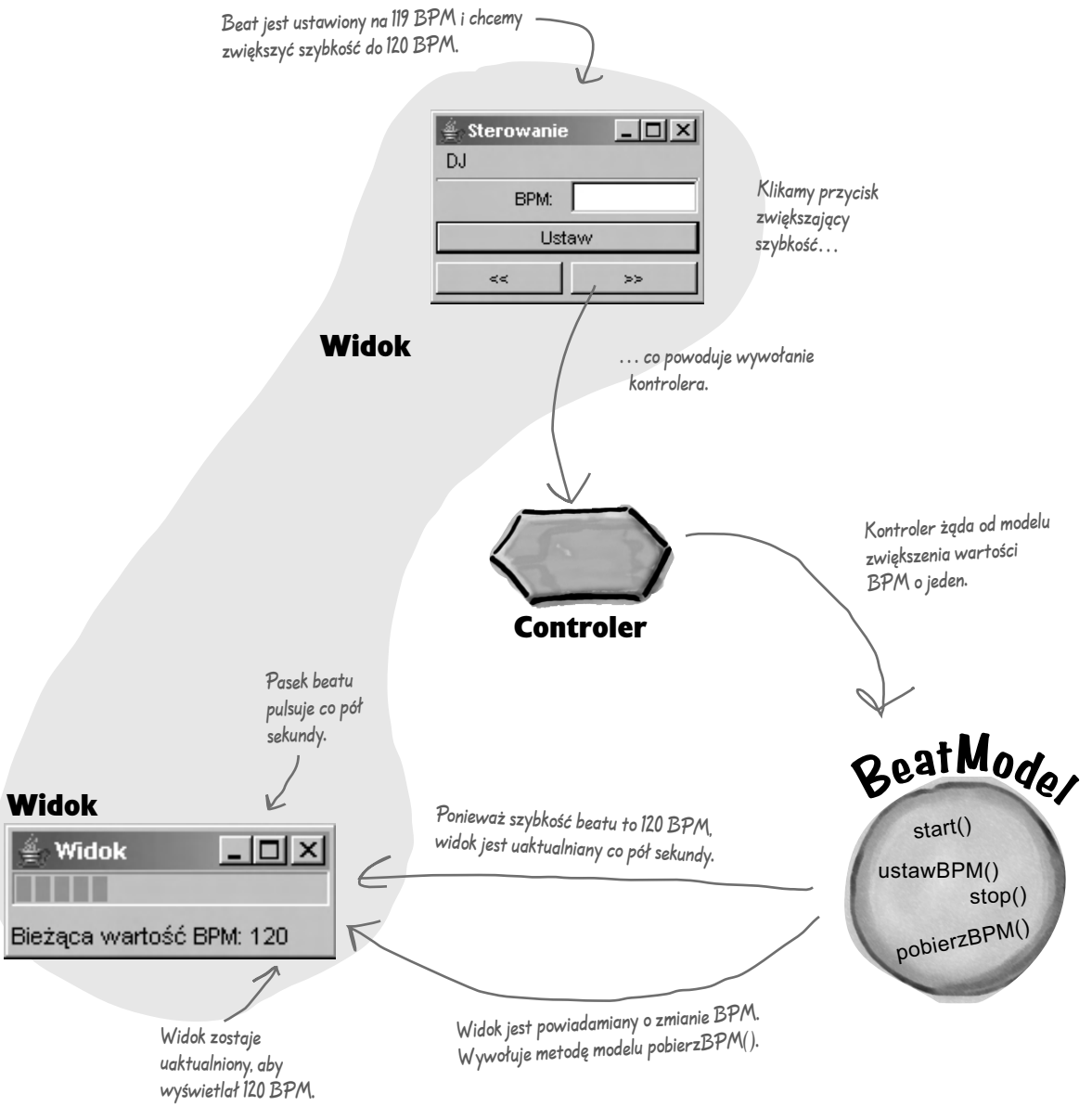


Nie zapominajmy o modelu, który jest sednem całości...

Model nie jest widoczny, ale można go usłyszeć. Model leży u podstaw aplikacji — zarządza beatem i przekazuje dźwięk do głośników.



Łączymy elementy w całość



Piszemy kod elementów

Wiemy już, że model odpowiada za zarządzanie danymi, stanem i wszelką logiką aplikacji. Co więc znajdziemy w klasie BeatModel? Jej głównym zadaniem jest zarządzanie beatem, więc wymaga ona informacji o stanie (czyli o liczbie uderzeń na minutę) oraz znacznej ilości kodu, który generuje zdarzenia MIDI odpowiedzialne za to, co słyszymy w głośnikach. Udostępnia też interfejs, który umożliwia kontrolerowi sterowanie beatem oraz pozwala widokowi i kontrolerowi pobierać informacje o stanie modelu. Nie można też zapomnieć o tym, że model korzysta z wzorca Obserwator. Potrzebne więc będą metody umożliwiające obiektom rejestrowanie się w charakterze obserwatorów i wysyłanie powiadomień.

Zanim zajmiemy się implementacją, przyjrzyjmy się interfejsowi BeatModelInterfejs:

```
public interface BeatModelInterfejs {
    void inicjuj();

    void start();

    void stop();

    void ustawBPM(int bpm);

    int pobierzBPM();

    void zarejestrujObserwatora(ObserwatorUderzeń o);

    void wyrejestrujObserwatora(ObserwatorUderzeń o);

    void zarejestrujObserwatora(ObserwatorBPM o);

    void wyrejestrujObserwatora(ObserwatorBPM o);
}
```

Ta metoda zostaje wywołana po utworzeniu egzemplarza klasy.

Te metody włączają i wyłączają generator beatu.

Ta metoda ustala liczbę uderzeń na minutę. Po jej wywołaniu szybkość beatu zostaje natychmiast zmieniona.

Metoda pobierzBPM() zwraca bieżącą wartość BPM lub 0, gdy generator jest wyłączony.

To są metody, których kontroler będzie używał do zarządzania beatem odpowiednio do interakcji z użytkownikiem.

Te metody umożliwiają widokowi i kontrolerowi pobieranie informacji o stanie oraz rejestrowanie się w charakterze obserwatorów.

Wyróżniliśmy tu dwa rodzaje obserwatorów: takich, którzy będą powiadamiani o każdym uderzeniu, i takich, którzy będą powiadamiani tylko wtedy, gdy zmieni się wartość BPM.

To powinno wyglądać znajomo — te metody umożliwiają rejestrowanie się w charakterze obserwatorów zmian stanu.

A teraz spójrzmy na rzeczywistą klasę BeatModel:

```
public class BeatModel implements BeatModelInterfejs, MetaEventListener {
    Sequencer sequencer;
    ArrayList obserwatorzyUderzeń = new ArrayList();
    ArrayList obserwatorzyBPM = new ArrayList();
    int bpm = 90;
    // inne zmienne egzemplarza

    public void inicjuj() {
        konfigurujMidi();
        zbudujŚcieżkęIRozpocznij();
    }

    public void start() {
        sequencer.start();
        ustawBPM(90);
    }

    public void stop() {
        ustawBPM(0);
        sequencer.stop();
    }

    public void ustawBPM(int bpm) {
        this.bpm = bpm;
        sequencer.setTempoInBPM(pobierzBPM());
        powiadomObserwatorówBPM();
    }

    public int pobierzBPM() {
        return bpm;
    }

    void uderzenieEvent() {
        powiadomObserwatorówUderzeń();
    }

    // kod rejestrowania i powiadamiania obserwatorów

    // dużo kodu MIDI do obsługi beatu
}
```

Implementujemy interfejs BeatModelInterfejs.

Tego potrzebuje kod MIDI.

Sekwenser to obiekt, który umie generować rzeczywisty beat (który możesz usłyszeć!).

Te kolekcje ArrayList przechowują dwa typy obserwatorów (beatu i BPM).

Zmienna egzemplarza bpm przechowuje szybkość beatu — standardowo 90 BPM.

Ta metoda konfiguruje sekwenser i przygotowuje ścieżki.

Metoda start() uruchamia sekwenser i ustawia BPM na wartość domyślną, czyli 90.

A metoda stop() kończy pracę sekwensera, ustawiając BPM na 0 i wywołując metodę stop() obiektu Sequencer.

Metoda ustawBPM() umożliwia kontrolerowi sterowanie beatem. Robi trzy rzeczy:

(1) Ustawia zmienną egzemplarza bpm.

(2) Żąda od sekwensera zmiany BPM.

(3) Powiadamia obserwatorów BPM o zmianie BPM.

Metoda pobierzBPM() zwraca wartość zmiennej egzemplarza bpm, odpowiadającą bieżącej szybkości beatu.

Metoda uderzenieEvent(), której nie ma w interfejsie BeatModelInterfejs, jest wywoływana przez kod MIDI, kiedy rozpoczyna się nowe uderzenie. Powiadamia ona obiekty BeatObserver o tym uderzeniu.



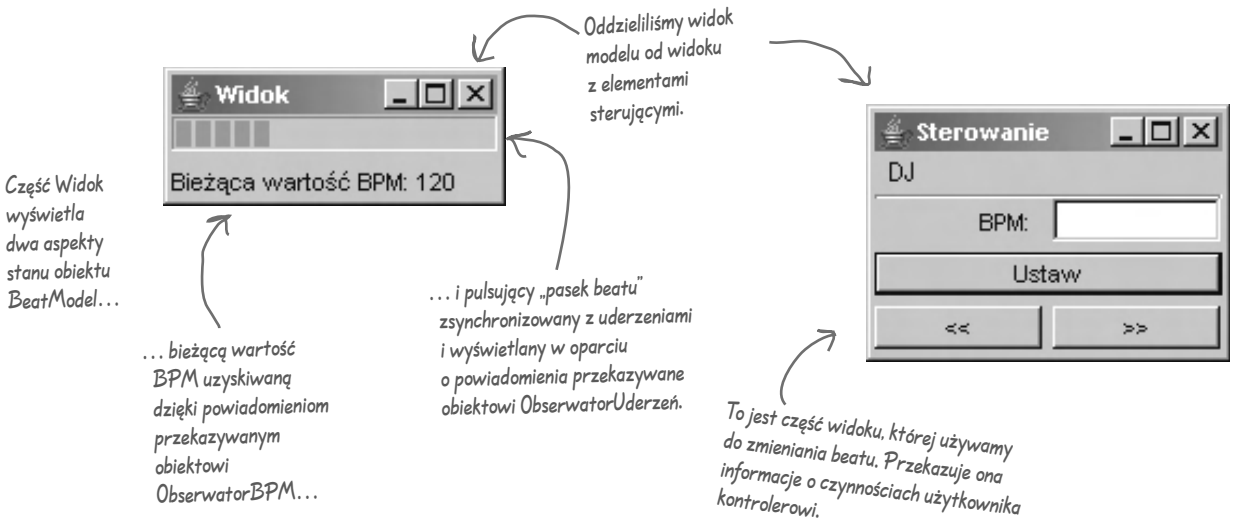
Kod gotowy do użycia

W tym modelu generujemy beat, korzystając ze standardowej obsługi MIDI w języku Java. Kompletną implementację klas tej aplikacji możesz pobrać z serwera FTP wydawnictwa Helion: *ftp://ftp.helion.pl/przyklady/wzorrg.zip*. Przedstawiamy ją też na końcu rozdziału.

Widok

Teraz zaczyna się zabawa; musimy uzupełnić model widokiem i zapewnić odpowiednią wizualizację!

Pierwszą ważną rzeczą dotyczącą widoku jest to, że będziemy korzystać z dwóch osobnych okien. Jedno okno zawiera informację o bieżącej wartości BPM i pulsu. W drugim są elementy sterujące. Dlaczego tak? Ma to podkreślić różnicę między częścią interfejsu, która zawiera widok modelu, a tą, w której umieszczone są elementy sterujące użytkownika. Przyjrzyjmy się dokładnie obu częściom:





WYTĘŻ UMYSŁ

Nasza klasa BeatModel nie opiera się na żadnych założeniach dotyczących widoku. Model jest zaimplementowany zgodnie z wzorcem Obserwator, więc ogranicza się do powiadamiania każdego widoku zarejestrowanego w charakterze obserwatora o zmianach swojego stanu. Widok uzyskuje dane o stanie modelu w oparciu o udostępniany przez ten model interfejs. Tutaj zaimplementowaliśmy jeden rodzaj widoku. Czy możesz wymyślić jeszcze inne, które mogłyby skorzystać z powiadomień i informacji o stanie przekazywanych przez klasę BeatModel?

System oświetlenia zsynchronizowany z beatem.

Widok tekstowy, który wyświetla informację o gatunku muzyki w oparciu o BPM

(ambient, downbeat, techno itd.).

Implementowanie widoku

Dwie części widoku — podgląd modelu i zbiór elementów sterujących — są wyświetlane w dwóch oknach, ale stanowią w rzeczywistości jedną klasę. Najpierw przedstawimy tutaj kod, który tworzy podgląd modelu i wyświetla bieżącą wartość BPM oraz pasek beatu. Na kolejnej stronie przedstawimy zaś kod, który tworzy elementy sterujące interfejsu użytkownika — pole tekstowe do wprowadzania BPM i przyciski.



Uwaga!

Kod na tych dwóch stronach to tylko ogólny schemat!

Podzieliliśmy tutaj JEDNĄ klasę na DWIE, przedstawiając jedną część widoku na tej stronie, a drugą na następnej. Całość jest w rzeczywistości JEDNĄ klasą — DJWidok.java. Klasa ta została przedstawiona w całości na końcu rozdziału.

DJWidok to obserwator zarówno beatu, jak i informacji o jego szybkości.

```
public class DJWidok implements ActionListener, ObserwatorUderzeń, ObserwatorBPM {
    BeatModelInterfejs model;
    KontrolerInterfejs kontroler;
    JFrame widokFrame;
    JPanel widokPanel;
    BeatBar beatBar;
    JLabel bpmWyjścieLabel;

    public DJWidok(KontrolerInterfejs kontroler, BeatModelInterfejs model) {
        this.kontroler = kontroler;
        this.model = model;
        model.zarejestrujObserwatora((ObserwatorUderzeń)this);
        model.zarejestrujObserwatora((ObserwatorBPM)this);
    }

    public void utwórzWidok() {
        // tworzenie elementów sterujących (Swing)
    }

    public void aktualizujBPM() {
            int bpm = model.getBPM();
            if (bpm == 0) {
                    bpmWyjścieLabel.setText("offline");
            } else {
                    bpmWyjścieLabel.setText("Bieżąca wartość BPM: " + model.pobierzBPM());
            }
    }

    public void aktualizujBeat() {
            beatBar.setValue(100);
    }
}
```

Widok przechowuje odwołania do modelu i kontrolera. Kontroler jest wykorzystywany wyłącznie przez część sterującą, którą zajmiemy się za chwilę…

Tutaj tworzymy kilka wyświetlanych elementów.

Konstruktor pobiera odwołania do kontrolera i modelu. Zapisujemy je w zmiennych egzemplarza.

Rejestrujemy widok jako obserwatora beatu i jako obserwatora BPM.

Metoda aktualizujBPM() jest wywoływana przy zmianie stanu modelu. Gdy taka zmiana nastąpi, aktualizujemy informację o bieżącej szybkości beatu. Możemy pobrać tę wartość bezpośrednio z modelu.

Podobnie metoda aktualizujBeat() jest wywoływana, gdy model rozpoczyna nowe uderzenie. Gdy tak się dzieje, musimy zapewnić jeden impuls naszego „paska beatu". Uzyskujemy ten efekt przez ustawienie jego maksymalnej wartości (100). Pasek sam dba o animację impulsu.

Implementowanie widoku — ciąg dalszy

Teraz zajmiemy się kodem elementów sterujących. Ta część widoku umożliwia sterowanie modelem — najpierw widok przekazuje kontrolerowi informacje o czynnościach użytkownika, następnie kontroler inicjuje operacje modelu. Pamiętajmy, że jest to kod tej samej klasy, która jest przedstawiona na poprzedniej stronie.

```
public class DJWidok implements ActionListener,  ObserwatorUderzeń, ObserwatorBPM
{
    BeatModelInterfejs model;
    KontrolerInterfejs kontroler;
    JLabel bpmLabel;
    JTextField bpmTextField;
    JButton ustawBPMButton;
    JButton zwiększBPMButton;
    JButton zmniejszBPMButton;
    JMenuBar menuBar;
    JMenu menu;
    JMenuItem startMenuItem;
    JMenuItem stopMenuItem;

    public void utwórzElementySterujące() {
        // tworzenie elementów sterujących (Swing)
    }

    public void aktywujStopMenuItem() {
        stopMenuItem.setEnabled(true);
    }

    public void dezaktywujStopMenuItem() {
        stopMenuItem.setEnabled(false);
    }

    public void aktywujStartMenuItem() {
        startMenuItem.setEnabled(true);
    }

    public void dezaktywujStartMenuItem() {
        startMenuItem.setEnabled(false);
    }

    public void actionPerformed(ActionEvent event) {
        if (event.getSource() == ustawBPMButton) {
            int bpm = Integer.parseInt(bpmTextField.getText());
            kontroler.ustawBPM(bpm);
        } else if (event.getSource() == zwiększBPMButton) {
            kontroler.zwiększBPM();
        } else if (event.getSource() == zmniejszBPMButton) {
            kontroler.zmniejszBPM();
        }
    }
}
```



Ta metoda tworzy wszystkie elementy sterujące i umieszcza je w interfejsie użytkownika. Buduje też menu. Wybranie pozycji Stop lub Start powoduje wywołanie odpowiednich metod kontrolera.

Te metody umożliwiają aktywowanie i dezaktywowanie pozycji menu Start i Stop. Jak się przekonamy, kontroler wykorzystuje je do modyfikowania interfejsu użytkownika.

Ta metoda jest wywoływana po kliknięciu przycisku.

Wciśnięcie przycisku Ustaw powoduje wywołanie metody kontrolera, której przekazywana jest wartość BPM z pola tekstowego.

Podobnie kliknięcie przycisków zwiększania lub zmniejszania szybkości powoduje wywołanie odpowiedniej metody kontrolera.

A teraz kontroler

Czas napisać ostatnią część kodu: kontroler. Pamiętajmy, że kontroler jest strategią, którą wpinamy do widoku, aby zapewnić mu inteligencję.

Ponieważ implementujemy wzorzec Strategia, rozpoczniemy od interfejsu, z którego widok będzie korzystał. Nazwiemy go KontrolerInterfejs.

```
public interface KontrolerInterfejs {
    void start();
    void stop();
    void zwiększszBPM();
    void zmniejszBPM();
    void ustawBPM(int bpm);
}
```

Tutaj mamy wszystkie metody kontrolera, z których będzie korzystał widok.

Gdy już znamy interfejs modelu, ten zestaw powinien wyglądać znajomo. Możemy rozpoczynać i kończyć generowanie beatu i zmieniać jego szybkość. Ten interfejs jest „bogatszy" niż interfejs klasy BeatModel, ponieważ umożliwia zwiększanie i zmniejszanie wartości BPM.



Łamigłówka projektowa

Wiemy już, że widok i kontroler opierają swoją współpracę na wzorcu Strategia. Czy potrafisz narysować diagram klas, który ilustruje, w jaki sposób ten wzorzec został zastosowany?

A oto implementacja kontrolera:

```
public class BeatKontroler implements KontrolerInterfejs {
    BeatModelInterfejs model;
    DJWidok widok;

    public BeatKontroler(BeatModelInterfejs model) {
        this.model = model;
        widok = new DJWidok(this, model);
        widok.utwórzWidok();
        widok.utwórzElementySterujące();
        widok.dezaktywujStopMenuItem();
        widok.aktywujStartMenuItem();
        model.inicjuj();
    }

    public void start() {
        model.start();
        widok.dezaktywujStartMenuItem();
        widok.aktywujStopMenuItem();
    }

    public void stop() {
        model.stop();
        widok.dezaktywujStopMenuItem();
        widok.aktywujStartMenuItem();
    }

    public void zwiększBPM() {
        int bpm = model.pobierzBPM();
        model.ustawBPM(bpm + 1);
    }

    public void zmniejszBPM() {
        int bpm = model.pobierzBPM();
        model.ustawBPM(bpm - 1);
      }

     public void ustawBPM(int bpm) {
        model.ustawBPM(bpm);
    }
}
```

Kontroler implementuje interfejs KontrolerInterfejs.

Kontroler to kremowe nadzienie w centrum markizy MVC — obiekt, który łączy się zarówno z widokiem, jak i z modelem, zapewniając, że tworzą jedną całość.

Konstruktor kontrolera pobiera model i tworzy widok.

Po wybraniu polecenia Start z interfejsu użytkownika kontroler uruchamia model i modyfikuje interfejs tak, że pozycja menu Start jest nieaktywna, a pozycja Stop staje się dostępna.

Podobnie, gdy wybieramy z menu polecenie Stop, kontroler wyłącza model i zmienia interfejs użytkownika tak, że pozycja Stop zostaje zdezaktywowana, a Start staje się dostępna.

UWAGA: Kontroler wyręcza widok w podejmowaniu decyzji. Widok umie tylko włączać i wyłączać elementy menu. Nie potrafi zdecydować, kiedy ma to nastąpić.

Po kliknięciu przycisku zwiększenia szybkości kontroler pobiera z modelu bieżącą wartość BPM, dodaje jeden i ustawia nową szybkość beatu.

To samo dzieje się tutaj, ale wartość BPM zostaje zmniejszona o jeden

Jeżeli użyjemy interfejsu użytkownika do ustawienia swobodnie wybranej wartości BPM, kontroler przekazuje ją modelowi.

Uruchamiamy aplikację...



Mamy już wszystko, czego potrzebujemy: model, widok i kontroler. Czas połączyć je razem! Zobaczymy i usłyszymy, jak sprawnie współpracują.

Uruchomienie naszej aplikacji nie wymaga już wiele kodu:

```
public class DJTest {

    public static void main (String[] args) {
        BeatModelInterfejs model = new BeatModel();
        KontrolerInterfejs kontroler = new BeatKontroler(model);
    }
}
```

Najpierw tworzymy model...

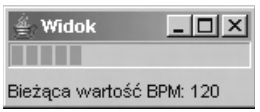
a następnie kontroler, któremu przekazujemy model. Pamiętamy, że kontroler sam tworzy widok, więc nie musimy już tego robić.

I sprawdzamy...



Uruchamiamy to...

... i widzimy to.





Do zrobienia

1. **Uruchom generowanie beatu przy użyciu pozycji menu Start. Zwróć uwagę, że kontroler dba o dezaktywację tej pozycji.**
2. **Użyj pola tekstowego oraz przycisków zmniejszania i zwiększania szybkości w celu zmiany wartości BPM. Zwróć uwagę, jak widok reaguje na zmiany pomimo faktu, że nie ma logicznego połączenia między elementami sterującymi.**
3. **Zwróć uwagę, jak pasek beatu nieustannie reaguje na beat dzięki temu, że jest obserwatorem modelu.**
4. **Włącz ulubioną piosenkę i zobacz, czy możesz dopasować tempo kliknięciami przycisków zmniejszania i zwiększania szybkości.**
5. **Zatrzymaj generator. Zwróć uwagę, że kontroler dezaktywuje pozycję menu Stop i ponownie aktywuje pozycję Start.**

Eksplorujemy możliwości wzorca Strategia



Skoncentrujemy się teraz na wzorcu Strategia, żeby lepiej oswoić się z jego rolą w mechanizmie MVC. Niejako przy okazji znajdziemy też miejsce dla kolejnego wzorca, który dość często towarzyszy układom Model-Widok-Kontroler — wzorca Adapter.

Zastanówmy się przez chwilę nad tym, co robi nasza przykładowa aplikacja: wyświetla informację o szybkości uderzeń i pulsujący wskaźnik beatu. Do czego jeszcze można tego rodzaju interfejs użytkownika wykorzystać? Pomyśl o uderzeniach serca i o pulsie… Tak się zdarzyło, że mamy „pod ręką” klasę monitora serca. Oto jej diagram:





WYTĘŻ UMYSŁ

Niewątpliwie byłoby bardzo wygodnie wykorzystać nasz gotowy widok do prezentacji modelu serca. Potrzebujemy jednak odpowiedniego kontrolera. Dodatkowo, interfejs klasy SerceModel nie odpowiada oczekiwaniom widoku — zamiast metody pobierzBPM() mamy metodę pobierzSzybkośćUderzeń(). Jak mógłby wyglądać projekt zespołu klas, który umożliwi wykorzystanie widoku z nowym modelem?

Adaptowanie modelu

Na początek będziemy musieli zaadaptować klasę SerceModel do interfejsu klasy BeatModel. Jest to niezbędne, aby widok mógł współpracować z nowym modelem. Wymaga on przecież metody pobierzBPM(), a nie pobierzSzybkośćUderzeń(). Jak to zrobimy? Oczywiście, użyjemy wzorca Adapter! Okazuje się, że jest to dość typowa towarzysząca układom MVC technika: wprowadzenie adaptera, który umożliwi zaadaptowanie modelu do pracy z istniejącymi kontrolerami i widokami.

Oto kod adaptera:

Implementujemy interfejs docelowy, w tym przypadku BeatModelInterfejs.

```
public class SerceAdapter implements BeatModelInterfejs {
    SerceModelInterfejs serce;

    public SerceAdapter(SerceModelInterfejs serce) {
        this.serce = serce;
    }

    public void inicjuj() {}

    public void start() {}

    public void stop() {}

    public int pobierzBPM() {
        return serce.pobierzSzybkośćUderzeń();
    }

    public void ustawBPM(int bpm) {}

    public void zarejestrujObserwatora(ObserwatorUderzeń o) {
        serce.zarejestrujObserwatora(o);
    }

    public void wyrejestrujObserwatora(ObserwatorUderzeń o) {
        serce.wyrejestrujObserwatora(o);
    }

    public void zarejestrujObserwatora(ObserwatorBPM o) {
        serce.zarejestrujObserwatora(o);
    }

    public void wyrejestrujObserwatora(ObserwatorBPM o) {
        serce.wyrejestrujObserwatora(o);
    }
}
```

Tutaj przechowujemy odwołanie do modelu serca.

Nie wiemy, co te metody miałyby robić w przypadku serca, ale wyglądają dość przerażająco. Niech więc pozostaną puste.

Wywołanie metody pobierzBPM() tłumaczymy na wywołanie pobierzSzybkośćUderzeń().

Takich rzeczy z sercem nie będziemy robić! Kolejna metoda pozostaje pusta.

Tu mamy nasze metody obsługujące obserwatorów. Delegujemy je do osłanianego modelu serca.

Nowy kontroler — SerceKontroler

Gdy już dysponujemy klasą SerceAdapter, nic nie stoi na przeszkodzie, aby przygotować nowy kontroler i umożliwić widokowi pracę z modelem SerceModel. To dopiero przykład ponownego wykorzystania kodu!

```
public class SerceKontroler implements KontrolerInterfejs {
    SerceModelInterfejs model;
    DJWidok widok;

    public SerceKontroler(SerceModelInterfejs model) {
        this.model = model;
        widok = new DJWidok(this, new SerceAdapter(model));
        widok.utwórzWidok();
        widok.utwórzElementySterujące();
        widok.dezaktywujStopMenuItem();
        widok.dezaktywujStartMenuItem();
    }

    public void start() {}

    public void stop() {}

    public void zwiększBPM() {}

    public void zmniejszBPM() {}

    public void ustawBPM(int bpm) {}
}
```

Klasa SerceKontroler jest implementacją interfejsu KontrolerInterfejs, podobnie jak wcześniejszy kontroler — BeatKontroler.

Podobnie jak wcześniej, kontroler tworzy widok i łączy wszystkie elementy w całość.

Jest jedna istotna zmiana — konstruktor pobiera obiekt SerceModel, a nie BeatModel...

... i zanim zostanie on przekazany widokowi, musi zostać osłonięty adapterem.

Nowy kontroler dezaktywuje nieużywane pozycje menu.

Tu nie ma wiele do zrobienia. Ostatecznie, serce nie daje takich możliwości sterowania jak urządzenia MIDI.

I to wszystko! Czas na kod testu...

```
public class SerceTest {

    public static void main (String[] args) {
        SerceModel serceModel = new SerceModel();
        KontrolerInterfejs model = new SerceKontroler(serceModel);
    }
}
```

Musimy tylko utworzyć kontroler i przekazać go monitorowi serca.

Testujemy...

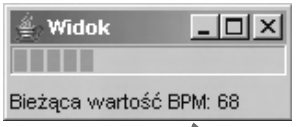


Uruchamiamy to...

... i widzimy to.



Do zrobienia



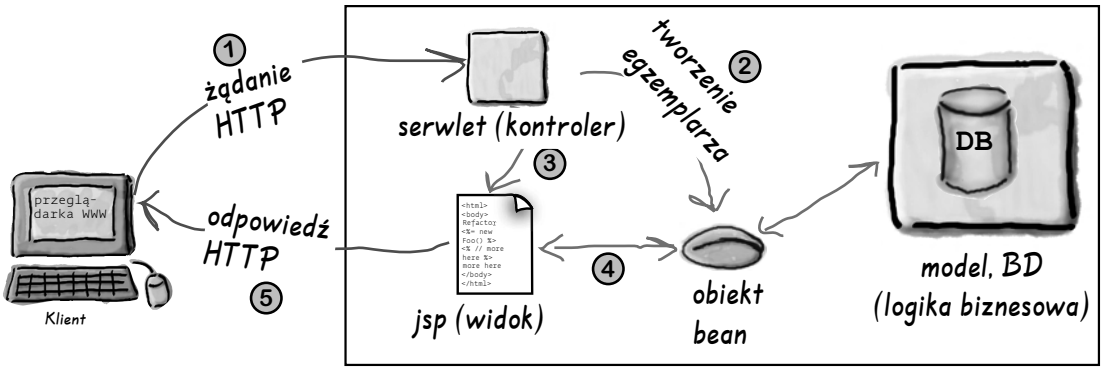
Ładny, zdrowy puls.

1. **Zwróć uwagę, że interfejs użytkownika doskonale współpracuje z nowym modelem! Pasek beatu wygląda jak prawdziwy wskaźnik pulsu. Ponieważ klasa SerceModel zapewnia obsługę obserwatorów BPM i uderzeń, nie ma żadnego problemu z aktualizacjami wyświetlanych danych.**
2. **Ponieważ szybkość bicia serca w naturalny sposób ulega zmianom, informacja o liczbie uderzeń na minutę jest stale uaktualniana.**
3. **Za każdym razem, gdy pobierasz nowe dane o liczbie uderzeń, adapter zapewnia translację wywołań pobierzBPM() na pobierzSzybkośćUderzeń().**
4. **Pozycje menu Start i Stop nie są aktywne, bo kontroler zadbał o ich wyłączenie.**
5. **Inne przyciski wciąż są dostępne, ale korzystanie z nich nie wywołuje żadnych efektów, ponieważ w implementacji kontrolera odpowiadają im metody puste. Widok można jeszcze poprawić tak, aby umożliwiał dezaktywowanie tych elementów.**

Wzorzec MVC i sieć WWW

Od boomu na WWW do rozpowszechnienia adaptacji wzorca MVC do rozwiązań typu przeglądarka-serwer upłynęło nieco czasu. Obecnie typowy schemat określa się krótko jako „Model 2” — jest to połączenie technologii serwletów i JSP, które prowadzi do takiego samego podziału na model, widok i kontroler, jaki spotykamy w tradycyjnych interfejsach GUI.

Model 2 wygląda tak:

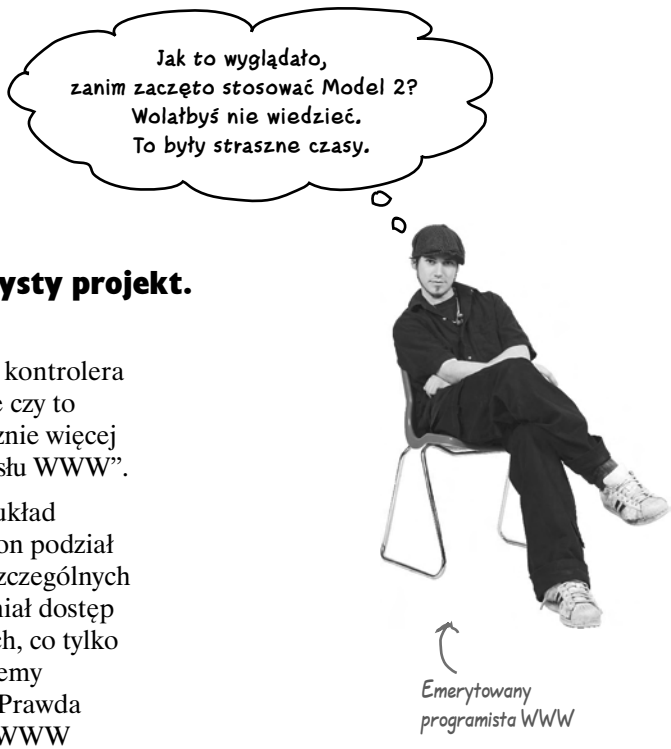


1 Przeglądarka użytkownika wysyła żądanie HTTP odbierane przez serwlet.
Użytkownik wysyła żądanie HTTP. Towarzyszą mu zazwyczaj pewne dane formularza, jak na przykład nazwa użytkownika i hasło. Serwlet odbiera te informacje i rozpoczyna ich przetwarzanie.

2 Serwlet pełni funkcję kontrolera.
Serwlet działa jak kontroler i przetwarza żądanie, w większości przypadków generując pewne żądania kierowane do modelu (którym zazwyczaj jest baza danych). Wynik przetwarzania żądania przyjmuje zwykle postać obiektu JavaBean.

3 Kontroler przekazuje sterowanie widokowi.
Funkcję widoku pełni kod JSP. Jedynym jego zadaniem jest generowanie strony WWW, która jest tutaj widokiem modelu ((4) uzyskiwanym w oparciu o dane udostępniane przez obiekt JavaBean) uzupełnionym elementami sterującymi niezbędnymi do dalszych interakcji z użytkownikiem.

5 Widok zwraca stronę przeglądarce (jako odpowiedź HTTP).
Strona jest przesyłana do przeglądarki, w której — jak każdy widok — zostaje wyświetlona użytkownikowi. W ten sam sposób można przetwarzać kolejne żądania użytkownika.



Model 2 to nie tylko przejrzysty projekt.

Zalety rozdzielenia widoku, modelu i kontrolera są już chyba dla Ciebie oczywiste. Ale czy to wszystko? Historia Modelu 2 to znacznie więcej — to podstawa współczesnego „przemysłu WWW".

Co to oznacza? Model 2 to nie tylko układ elementów w projekcie. Wprowadza on podział *odpowiedzialności* za *wytwarzanie* poszczególnych części aplikacji. Dawniej każdy, kto miał dostęp do plików JSP, mógł dopisywać do nich, co tylko chciał. Wiele takich osób miała problemy z odróżnieniem Javy od JavaScriptu. Prawda jest taka, że większość projektantów WWW zna się na HTML i projektowaniu stron internetowych, ale nie na programowaniu.

Model 2 okazał się jednym z najlepszych rozwiązań tego problemu. Dzięki niemu programowaniem mogą zajmować się autorzy serwletów, a projektantom WWW pozostają bardzo uproszczone strony JSP zawierające tylko HTML i nieskomplikowane „fasolki" (obiekty JavaBean).

Model 2 — Aplikacja didżeja w telefonie



Nie sądziłeś chyba, że zakończymy tę książkę bez przeniesienia naszego techno-modelu do środowiska WWW? W dzisiejszych czasach tak prosty interfejs WWW można wyświetlić w telefonie komórkowym. Wyobraźmy sobie didżeja, który wychodzi ze swojej budki i kontynuuje pracę zatopiony gdzieś w tłumie uczestników imprezy… Na co więc czekać? Zakodujmy to…

Plan

1. **Poprawiamy model.**
   Właściwie to nie ma w nim nic do poprawiania. Jest wystarczająco dobry w swojej obecnej postaci.

2. **Piszemy kontroler w postaci serwletu.**
   Potrzebujemy prostego serwletu, który będzie odbierał żądania HTTP i wykonywał operacje na modelu. W naszym przypadku potrzebujemy tylko uruchamiania i zatrzymywania beatu oraz modyfikowania jego szybkości.

3. **Piszemy widok HTML.**
   Utworzymy prosty widok w kodzie JSP. Będzie on korzystał z przekazanego przez kontroler obiektu JavaBean zawierającego wszystkie wyświetlane dane. Kod JSP wyświetli interfejs HTML.



Tylko dla wytrwałych

Przygotowywanie środowiska serwleta

Szczegółowy opis czynności niezbędnych do skonfigurowania środowiska serwletów to, jak na książkę o wzorcach projektowych, trochę zbyt obszerny temat (książka nie powinna ważyć więcej niż jej Czytelnik).

Otwórz swoją przeglądarkę i wyświetl stronę *http://jakarta.apache.org/tomcat/*. Znajdziesz tam aplikację Tomcat Servlet Container projektu Apache Jakarta, wraz z dokładnymi wskazówkami, co robić, aby ów „kontener serwletów" rozpoczął pracę.

Warto również zainteresować się książką *Head First Servlets & JSP. Edycja polska. Wydanie II, 2008* (jej autorzy to Bryan Basham, Kathy Sierra i Bert Bates).



Krok pierwszy — model

Jak pamiętamy, w schemacie MVC model nie dysponuje żadną wiedzą o widokach i kontrolerach. Innymi słowy, jest całkowicie odseparowany. Jedyne, czego oczekuje, to rejestracje obserwatorów, którzy będą powiadamiani o różnych zdarzeniach. Na tym polega urok wzorca Obserwator. Model zapewnia też interfejs, który widoki i kontrolery mogą wykorzystywać do pobierania i ustawiania parametrów stanu.

Jedyne, czego teraz wymagamy, to adaptacja modelu do środowiska WWW. Zważywszy, że nie jest on zależny od żadnych klas zewnętrznych, nie mamy w rzeczywistości nic do zrobienia. Klasę BeatModel będziemy mogli wykorzystać bez wprowadzania w niej ani jednej zmiany. To wszystko na ten temat. Przejdźmy do kroku drugiego!

Krok drugi — serwlet kontrolera

Kontroler ma teraz być serwletem. Będzie otrzymywał dane wejściowe w postaci przesyłanych przez przeglądarkę WWW żądań HTTP. Jego zadaniem pozostaje translacja tych żądań na operacje na modelu.

Drugim krokiem będzie przekazanie nowego widoku do przeglądarki. W tym celu przekażemy sterowanie widokowi, czyli kodowi JSP. Tym zajmiemy się w kroku trzecim.

Oto podstawowe fragmenty kodu naszego serwleta. Pełną implementację przedstawiamy na następnej stronie.

```
public class DJ extends HttpServlet {

    public void inicjuj() throws ServletException {
        BeatModel beatModel = new BeatModel();
        beatModel.inicjuj();
        getServletContext().setAttribute("beatModel", beatModel);
    }

    // metoda doPost

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
            throws IOException, ServletException
    {
        // implementacja
    }
}
```

Rozszerzamy klasę HttpServlet. To umożliwi naszej klasie realizowanie zachowań serwleta, takich jak odbieranie żądań HTTP.

To jest metoda inicjująca wywoływana przy tworzeniu serwleta.

Najpierw tworzymy obiekt BeatModel…

… i osadzamy odwołanie do niego w kontekście serwleta, aby zapewnić łatwy dostęp.

Metoda doGet() wykonuje właściwą pracę. Jej implementację znajdziesz na następnej stronie.

Oto kompletny listing implementacji metody doGet():

```
public void doGet(HttpServletRequest request,
                  HttpServletResponse, response)
        throws IOException, ServletException
{
    BeatModel beatModel =
        (BeatModel)getServletContext().getAttribute("beatModel");

    String bpm = request.getParameter("bpm");
    if (bpm == null) {
        bpm = beatModel.pobierzBPM() +"";
    }

    String set = request.getParameter("ustaw");
    if (set != null) {
        int bpmInt = 90;
        bpmInt = Integer.parseInt(bpm);
        beatModel.ustawBPM(bpmInt);
    }

    String zmniejsz = request.getParameter("zmniejsz");
    if (zmniejsz != null) {
        beatModel.ustawBPM(beatModel.getBPM() - 1);
    }
    String zwiększ = request.getParameter("zwiększ");
    if (zwiększ != null) {
        beatModel.ustawBPM(beatModel.getBPM() + 1);
    }
    String start = request.getParameter("start");
    if (start != null) {
        beatModel.start();
    }
    String stop = request.getParameter("stop");
    if (stop != null) {
        beatModel.stop();
    }

    request.setAttribute("beatModel", beatModel);

    RequestDispatcher dispatcher =
        request.getRequestDispatcher("/jsp/DJWidok.jsp");
    dispatcher.forward(request, response);
}
```

Najpierw pobieramy model z danych kontekstu serwleta. To odwołanie wykorzystamy do wywoływani a funkcji modelu.

Następnie pobieramy polecenia i parametry HTTP…

Jeżeli użytkownik użył polecenia „Ustaw", pobieramy wartość parametru ustaw i przekazujemy ją do modelu.

Aby zwiększyć lub zmniejszyć szybkość, pobieramy bieżącą wartość BPM z modelu i zwiększamy lub zmniejszamy ją o jeden.

Polecenia start i stop są w prosty sposób przekazywane do modelu.

Na tym kończy się praca kontrolera. Pozostaje przekazać sterowanie do widoku, aby ten utworzył prezentację HTML.

Zgodnie z zasadami Modelu 2 przekazujemy kodowi JSP obiekt bean z danymi stanu modelu. W tym przypadku przekazujemy sam model, co jest dopuszczalne, bo on też ma postać obiektu bean.

Teraz jeszcze widok...

Pozostało jeszcze zaimplementować widok i generator beatu z interfejsem WWW będzie gotowy! W Modelu 2 widok to strona JSP. Odwołania JSP wykorzystują wyłącznie obiekt bean przekazany przez kontroler. W naszym przypadku tym obiektem jest model. Jedyną właściwością wykorzystywaną przez stronę jest BPM. Korzystając z tego symbolicznego zbioru danych, budujemy stronę WWW zawierającą też elementy sterujące interfejsu użytkownika.

To nasz obiekt bean przekazany przez serwlet.

```
<jsp:useBean id="beatModel" scope="request" class="headfirst.mvc.dj.BeatModel" />

<html>
<head>
<title>DJ</title>
</head>
<body>

<h1>DJ</h1>
BPM = <jsp:getProperty name="beatModel" property="BPM" />
<br />
<hr>
<br />

<form method="post" action="/dj/servlet/DJ">
Nowa wartość BPM: <input type=text name="bpm"
                         value="<jsp:getProperty name="beatModel"
                         property"BPM" />">
 

<input type="submit" name="ustaw" value="set"><br />
<input type="submit" name="zmniejsz" value="<<">
<input type="submit" name="zwiększ" value=">>"><br />
<input type="submit" name="start" value="start">
<input type="submit" name="stop" value="stop"><br />
</form>

</body>
</html>
```

Początek kodu HTML.

Tutaj pobieramy właściwość BPM modelu (obiektu typu bean).

Teraz generujemy widok — informację o liczbie uderzeń na minutę.

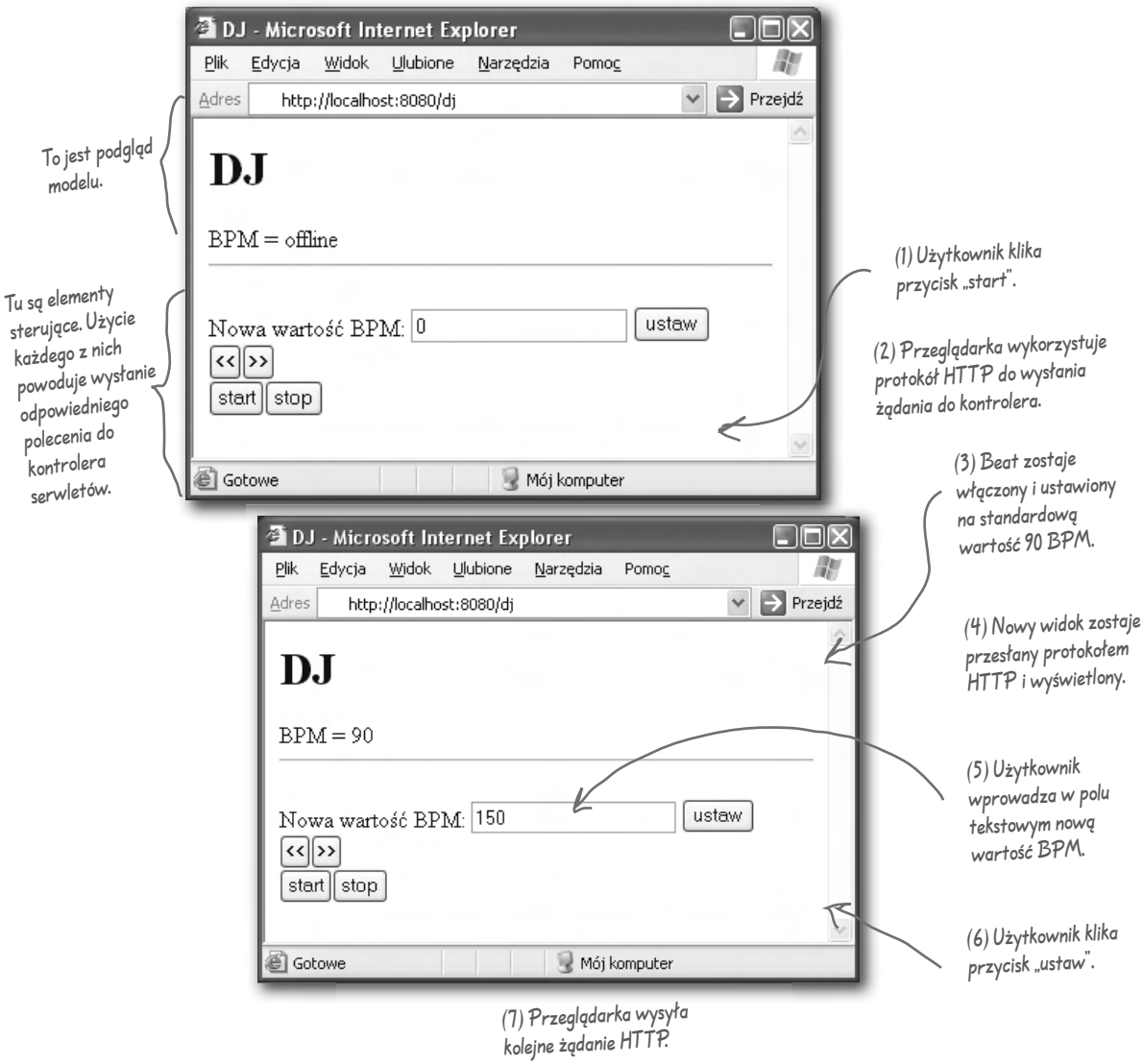
A tutaj mamy aktywną część widoku. Jest tu pole tekstowe do wprowadzania nowej wartości BPM oraz przyciski „zwiększ", „zmniejsz", „start" i „stop".

Tutaj kończy się kod HTML.

ZWRÓĆ UWAGĘ, że podobnie jak w MVC, w Modelu 2 widok nie zmienia modelu, ale jedynie wykorzystuje jego dane stanu! Zmienianie stanu modelu to zadanie kontrolera.

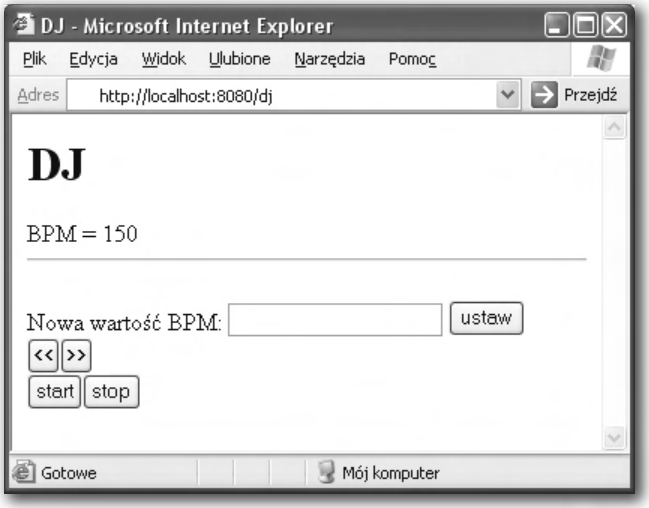
Testowanie aplikacji opartej na Modelu 2...

Czas otworzyć okno przeglądarki, wywołać serwlet DJ i rozpocząć zabawę…



(8) Kontroler zmienia wartość BPM w modelu na 150.

(9) Widok zwraca stronę HTML, której zawartość odpowiada bieżącemu stanowi modelu.



Do zrobienia

1. **Najpierw otwórz stronę WWW. Zobaczysz, że początkowa wartość BPM to 0. Kliknij przycisk „start".**
2. **Teraz na ekranie powinna pojawić się informacja, że wartość BPM to standardowe 90. Z komputera serwera zaczyna wydobywać się dźwięk.**
3. **Wprowadzamy pewną wartość BPM, na przykład 120, i klikamy przycisk „ustaw". Strona zostaje odświeżona, a beat z głośników serwera jest szybszy. Na ekranie widzimy „BPM = 120".**
4. **Teraz kliknij przyciski zwiększania i zmniejszania tempa.**
5. **Zwróć uwagę na to, jak przebiega wykonywanie kolejnych operacji. Interfejs HTML przesyła żądanie do serwleta (kontrolera), a serwlet przetwarza polecenia użytkownika i generuje żądania kierowane do modelu. Następnie serwlet przekazuje kontrolę podsystemowi JSP (widokowi), który generuje widok HTML zwracany do przeglądarki i wyświetlany.**

Model 2 a wzorce projektowe

Po zaimplementowaniu naszej aplikacji w środowisku WWW możesz zastanawiać się, co się stało z wcześniejszymi wzorcami. Mamy teraz widok HTML utworzony z JSP. Nie jest on obserwatorem modelu. Mamy kontroler, który jest serwletem i odbiera żądania HTTP. Czy wciąż możemy mówić o wzorcu Strategia? A co się stało z wzorcem Kompozyt? Nasz widok to wyświetlany w przeglądarce kod HTML.

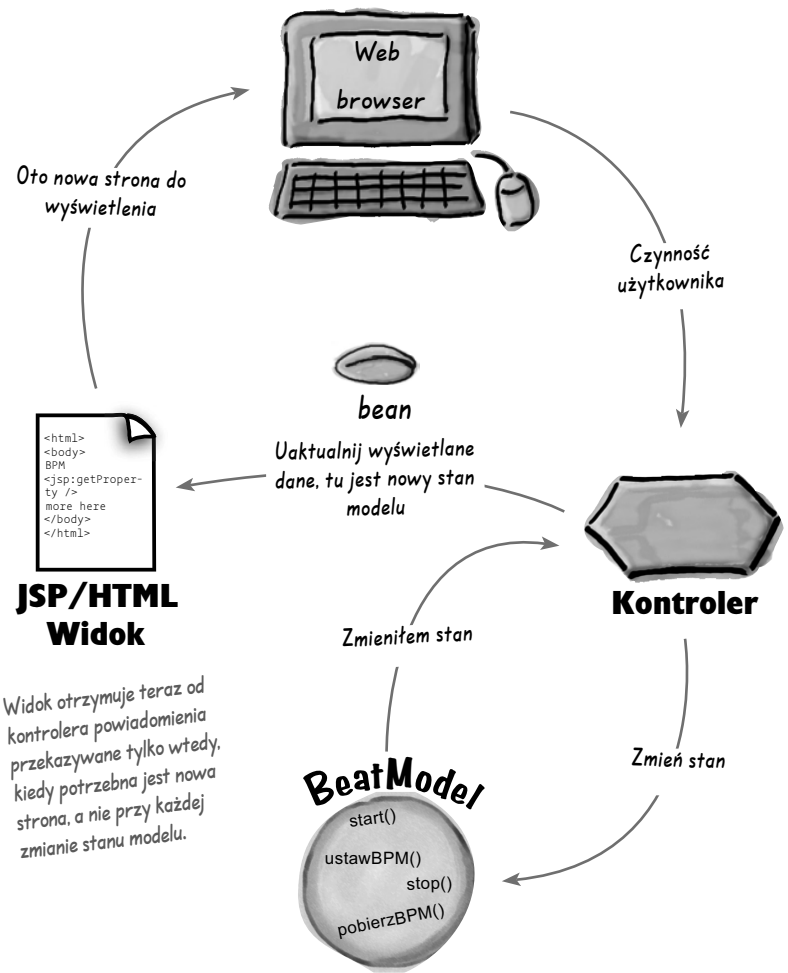
Model 2 to adaptacja wzorca MVC do środowiska WWW.
Nawet jeżeli Model 2 nie wygląda jak podręcznikowy przykład wzorca MVC, wszystkie jego elementy zostają zachowane. Idiosynkrazja środowiska WWW zmusiła jednak do ich modyfikacji. Spójrzmy teraz na Model 2 inaczej…

Obserwator

Widok nie jest już klasycznym obserwatorem modelu. Oznacza to, że nie rejestruje się w modelu jako obiekt odbierający informacje o zmianach stanu.

Widok otrzymuje powiadomienia pośrednio, od kontrolera. I w tym przypadku są to powiadomienia o każdej zmianie modelu. Kontroler dba również o przekazanie obiektu typu bean zawierającego niezbędne do utworzenia widoku dane.

Specyfiką pracy z przeglądarką WWW jest to, że widok wymaga aktualizacji tylko wtedy, kiedy serwer wysyła odpowiedź HTTP. Wysyłanie dodatkowych powiadomień jest bezcelowe. Pobieranie danych stanu i generowanie widoku należą więc do operacji wykonywanych w trakcie tworzenia i wysyłania strony.

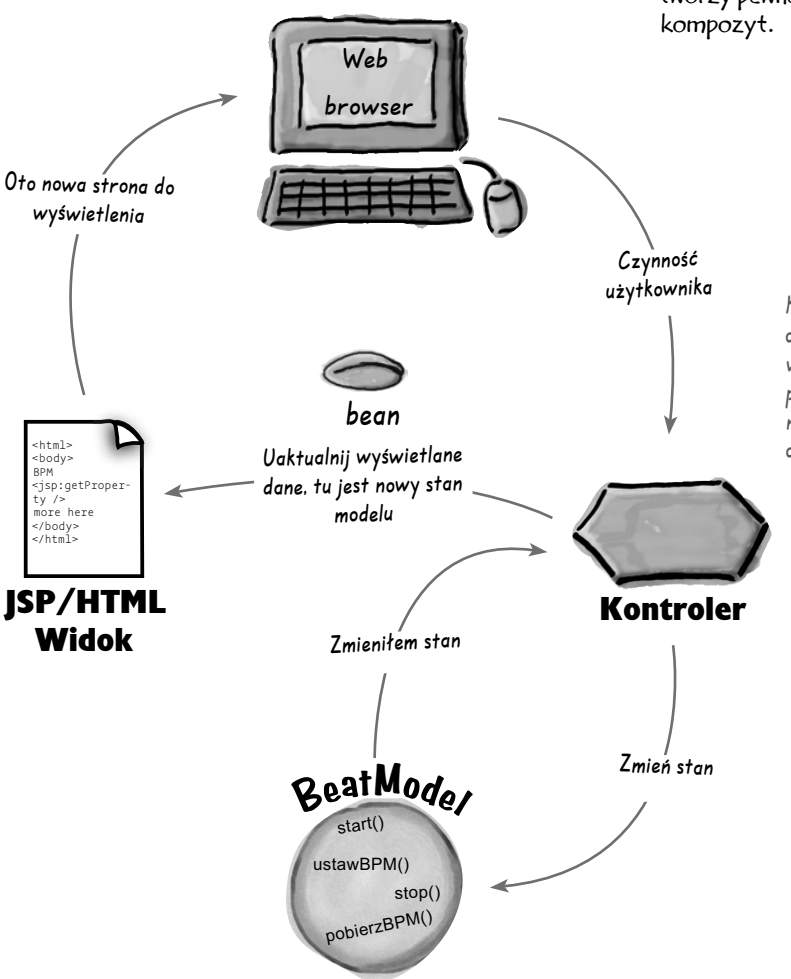


Strategia

W Modelu 2 obiektem strategii jest wciąż serwlet kontrolera. Nie tworzy on klasycznej kompozycji z widokiem. Jest to jednak obiekt, który implementuje zachowania widoku i może zostać zastąpiony innym, gdy pojawia się potrzeba zmiany zachowań.

Kompozyt

Podobnie jak biblioteka GUI Swing, widok jest zawsze hierarchicznym zespołem elementów graficznych. W tym przypadku są one renderowane przez przeglądarkę WWW na podstawie kodu HTML. U podstaw takiego mechanizmu pozostaje jednak system obiektów, który w większości przypadków tworzy pewnego rodzaju kompozyt.



Kontroler wciąż odpowiada za zachowania widoku, nawet jeżeli jego połączenie z widokiem nie jest już kompozycją obiektów.

Nie istnieją głupie pytania

P: Można mieć pewne wątpliwości, czy w rozwiązaniach typu MVC faktycznie znajdziemy wzorzec kompozyt. Czy on naprawdę w nich jest?

O: Tak, wzorzec Kompozyt jest standardowym elementem MVC. Jest to jednak dobre pytanie. Współczesne pakiety GUI, takie jak Swing, są już na tyle zaawansowane, że prawie nie zauważamy ich wewnętrznej struktury i faktu stosowania w nich struktury kompozytu, gdy budujemy i konfigurujemy mechanizm aktualizowania wyświetlanych danych. Jest to jeszcze trudniejsze, gdy dysponujemy przeglądarką WWW, która buduje interfejs użytkownika na podstawie kodu w języku takim, jak HTML.

Nie należy zapominać, że wzorzec MVC powstał w czasach, kiedy utworzenie graficznego interfejsu użytkownika wymagało znacznie więcej samodzielnej pracy programisty. Sprawiało to, że obecność wzorca Kompozyt w aplikacji opartej na schemacie MVC była znacznie wyraźniejsza.

P: Czy kontroler nigdy nie implementuje żadnej logiki aplikacji?

O: Nie, kontroler implementuje zachowania widoku. Jego zadaniem jest tłumaczenie czynności użytkownika na operacje modelu. Model reaguje na wywołania operacji i stosuje logikę aplikacji do określenia właściwych działań. Kontroler może wykonywać pewną pracę związaną z określaniem, jakie metody modelu mają zostać wywołane, ale nigdy nie należy to do funkcji określanych jako „logika aplikacji". Logika aplikacji to kod, który zarządza i operuje danymi. Jest on w całości zawarty w modelu.

P: Zawsze zastanawiałam się, co właściwie oznacza słowo „model". Rozumiem teraz, że to centralny mechanizm aplikacji. Ale dlaczego taki ogólny, mało oczywisty termin został wybrany do określenia tej części wzorca MVC?

O: Gdy powstawał wzorzec MVC, potrzebne było słowo na literę „M". W przeciwnym razie nie pasowałoby to do skrótu MVC.

Mówiąc poważniej, zgadzam się z Tobą w tym aspekcie. Niejeden już drapał się w głowę i zastanawiał, czym jest model. Wcześniej czy później jednak wszyscy zdają sobie sprawę, że nie znają lepszego słowa.

P: Wielokrotnie pojawiało się określenie „stan modelu". Czy to ma jakiś związek z wzorcem Stan?

O: Nie, chodzi o stan w ogólnym znaczeniu tego słowa. Choć trudno zaprzeczyć, że pewne modele wykorzystują do zarządzania stanem wzorzec Stan.

P: Widziałam opisy schematu MVC, w których kontroler jest opisywany jako „mediator" pracujący pomiędzy widokiem a modelem. Czy pojawia się tu wzorzec Mediator?

O: Nie zajmowaliśmy się w tej książce wzorcem Mediator (choć ogólny jego opis można znaleźć w dodatku), więc nie będziemy rozważać tego zagadnienia szczegółowo, ale celem wzorca Mediator jest hermetyzacja interakcji i zapewnienie separacji dwóch obiektów, które wówczas nie odwołują się do siebie w sposób jawny. Do pewnego stopnia kontroler może być postrzegany jako mediator — widok nigdy nie modyfikuje stanu modelu bezpośrednio. Pamiętajmy jednak, że widok dysponuje odwołaniem do modelu i wykorzystuje je do pobierania danych stanu. Gdyby kontroler był klasycznym mediatorem, pośredniczyłby również w pobieraniu danych stanu modelu.

P: Czy widok koniecznie musi pobierać dane stanu z modelu? Czy nie można przekazywać tych informacji razem z powiadomieniem o aktualizacji?

O: Owszem, model może przesyłać dane stanu razem z powiadomieniem. W rzeczywistości, gdy spojrzymy uważnie na widok JSP/HTML, zauważymy, że właśnie tak się dzieje. Przesyłanym obiektem bean jest cały model. Widok używa go jako źródła danych stanu, korzystając z mechanizmu właściwości obiektów bean. Podobny efekt moglibyśmy uzyskać, przekazując wyłącznie te dane stanu, którymi widok jest faktycznie zainteresowany. Jeżeli jednak pamiętasz omówienie wzorca Obserwator, rozwiązanie tego rodzaju ma kilka istotnych wad. Piszemy o nich w rozdziale drugim.

P: Jeżeli mam więcej niż jeden widok, czy zawsze potrzebuję więcej niż jednego kontrolera?

O: W typowych rozwiązaniach w czasie wykonywania każdy widok wymaga istnienia własnego kontrolera. Jednak różne widoki mogą być obsługiwane przez tę samą klasę.

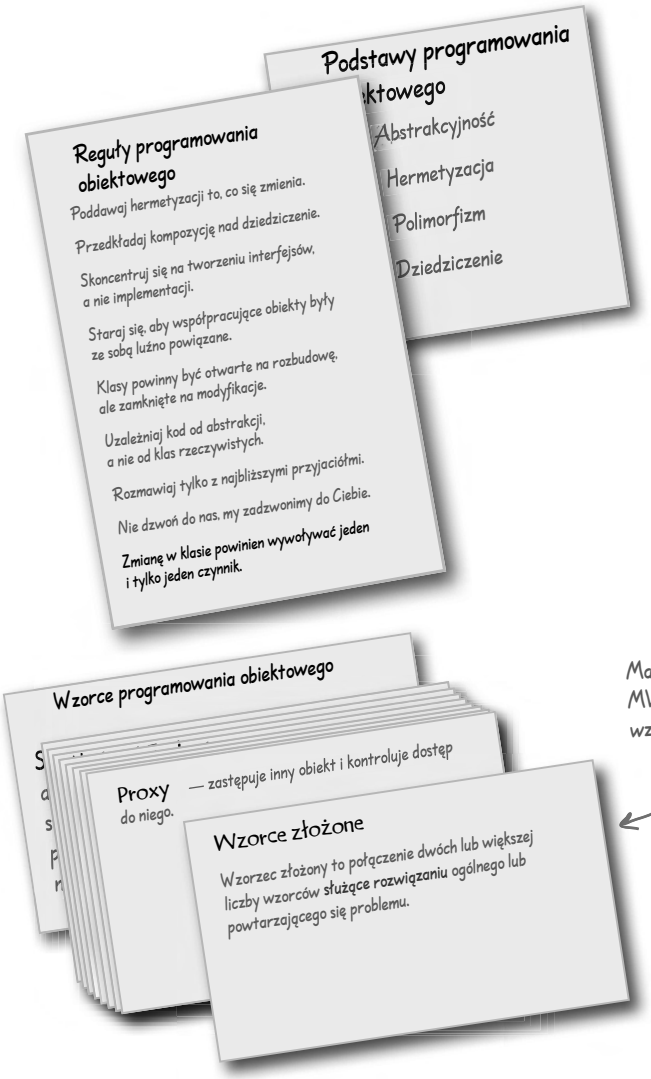
P: Widok nie powinien wykonywać operacji na modelu. Natomiast w przedstawionych przykładach widok ma pełny dostęp do metod zmieniających jego stan. Czy to nie jest niebezpieczne?

O: Masz całkowitą rację. Daliśmy widokowi pełny dostęp do zestawu metod modelu. Zrobiliśmy tak, aby uniknąć dodatkowych komplikacji. Dostęp widoku wyłącznie do pewnej części interfejsu API modelu jest w pewnych przypadkach uzasadniony. Istnieje wzorzec przeznaczony specjalnie do wyróżniania udostępnianej części interfejsu. Czy pamiętasz, jaki?



Twoja skrzynka narzędziowa

Nasza skrzynka narzędziowa niewątpliwie robi wrażenie. Spójrzmy teraz na te wszystkie reguły, wzorce, a teraz jeszcze wzorce złożone!



CELNE SPOSTRZEŻENIA

- Wzorzec Model-Widok-Kontroler (MVC) to wzorzec złożony, na który składają się wzorce Obserwator, Strategia i Kompozyt.
- Model wykorzystuje wzorzec Obserwator, aby powiadamiać o zmianie swojego stanu, a zarazem zachować separację od powiadamianych obiektów.
- Kontroler to strategia widoku. Widok może korzystać z różnych implementacji kontrolera, aby uzyskać inne zachowanie.
- Widok wykorzystuje wzorzec Kompozyt jako podstawę struktury interfejsu użytkownika, na którą składa się zazwyczaj wiele elementów złożonych, takich jak panele i ramki.
- Trzy wzorce składowe MVC zapewniają separację trzech elementów projektu, który jest dzięki temu przejrzysty i elastyczny.
- Wzorzec Adapter może posłużyć do przystosowania nowego modelu do starszego widoku i kontrolera.
- Model 2 to adaptacja wzorca MVC na potrzeby aplikacji WWW.
- W Modelu 2 kontroler jest serwletem, a widok ma postać kodu JSP i HTML.



Rozwiązania ćwiczeń



Zaostrz ołówek

Klasa KwakLicznik to jedna z klas implementujących interfejs Kwacząca. Gdy zmieniamy interfejs Kwacząca i staje się on rozszerzeniem interfejsu KwakObserwowany, musimy zmienić *każdą* klasę, która implementuje nową wersję interfejsu, a więc i KwakLicznik.

KwakLicznik jest implementacją interfejsu Kwacząca, a więc i interfejsu KwakObserwowany.

```
public class KwakLicznik implements Kwacząca {
    Kwacząca kaczka;
    static int liczbaKwaknięć;

    public KwakLicznik(Kwacząca kaczka) {
        this.kaczka = kaczka;
    }

    public void kwacz() {
        kaczka.kwacz();
        liczbaKwaknięć++;
    }

    public static int pobierzLiczbaKwaknięć() {
        return liczbaKwaknięć;
    }

    public void zarejestrujObserwatora(Obserwator obserwator) {
        kaczka.zarejestrujObserwatora(obserwator);
    }

    public void powiadomObserwatorów() {
        kaczka.powiadomObserwatorów();
    }
}
```

Tutaj jest kaczka dekorowana licznikiem. To ona jest obiektem, który powinien zapewnić obsługę metod interfejsu KwakObserwowany.

Cały ten kod pozostaje taki sam, jak poprzedniej wersji klasy.

Oto dwie metody wymagane przez interfejs KwakObserwowany. Zwróćmy uwagę, że ograniczamy się do delegacji obu wywołań do dekorowanej kaczki.



Zaostrz ołówek

A gdyby kwakolog chciał śledzić całe stado? Co to dla nas oznacza? Możemy to potraktować w następujący sposób: jeżeli obserwujemy obiekt złożony, obserwujemy wszystkie jego elementy składowe. A więc po zarejestrowaniu obserwatora stada obiekt Stado dba o rejestrację tego obserwatora u wszystkich potomków. Potomkami są kaczki lub mniejsze stada (rodziny kaczek).

Stado jest implementacją interfejsu Kwacząca, więc teraz musi implementować również metody interfejsu KwakObserwowana.

```
public class Stado implements Kwacząca {
    ArrayList ptaki = new ArrayList();

    public void dalej(Kwacząca ptak) {
        ptaki.dalej(ptak);
    }

    public void kwak() {
        Iterator iterator = ptaki.iterator();
        while (iterator.hasNext()) {
            Kwacząca ptak = (Kwacząca)iterator.next();
            ptak.kwak();
        }
    }

    public void zarejestrujObserwator(Obserwator obserwator) {
        Iterator iterator = ptaki.iterator();
        while (iterator.hasNext()) {
            Kwacząca ptak = (Kwacząca)iterator.next();
            ptak.zarejestrujObserwatora(obserwator);
        }
    }

    public void powiadomObserwatorów() { }
}
```

To są obiekty Kwacząca, które tworzą stado.

Gdy rejestrujemy obserwatora stada, w rzeczywistości przeprowadzamy rejestrację w każdym z jego obiektów składowych. Obejmuje to wszystkie obiekty typu Kwacząca, a więc zarówno kaczki, jak i inne stada.

Iterujemy wszystkie obiekty Kwacząca w stadzie i delegujemy wywołanie do każdego z nich. Jeżeli obiektem jest Stado, robimy dokładnie to samo.

Każdy obiekt Kwacząca generuje własne powiadomienie, więc obiekt typu Stado nie musi robić już nic. Dzieje się tak dzięki temu, że każde stado w podobny sposób deleguje obsługę metody kwacz().



Zaostrz ołówek

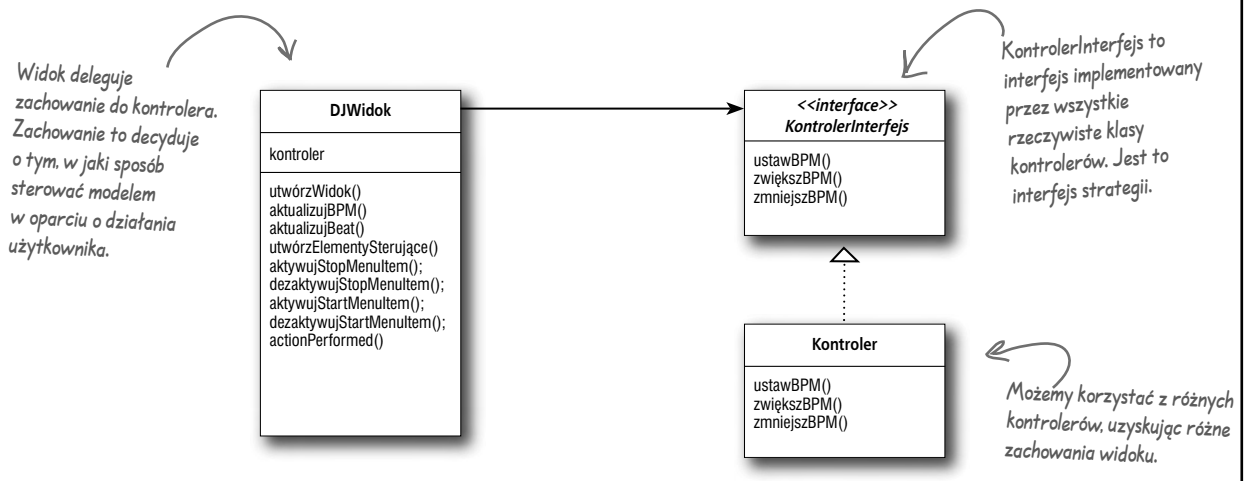
Gęsi wciąż tworzymy bezpośrednio, odwołując się do klas konkretnych. Czy potrafisz napisać abstrakcyjną fabrykę gęsi? W jaki sposób powinna ona obsługiwać tworzenie „kwaczącej gęsi"?

Możemy dodać do naszych fabryk kaczek metodę utwórzGęśKaczka(). Alternatywnym rozwiązaniem jest utworzenie osobnej fabryki produkującej rodziny gęsi.



Rozwiązanie łamigłówki projektowej

Wiemy już, że widok i kontroler opierają swoją współpracę na wzorcu Strategia. Czy potrafisz narysować diagram klas, który ilustruje, w jaki sposób ten wzorzec został zastosowany?





Kod gotowy do użycia

Oto kompletna implementacja aplikacji DJ. Zawiera ona obszerny kod MIDI, który generuje dźwięk, oraz wszystkie elementy graficzne niezbędne do utworzenia widoku. Kod można też pobrać z serwera FTP wydawnictwa Helion: *ftp://ftp.helion.pl/przyklady/wzorrg.zip*. Miłej zabawy!

```
package headfirst.mvc.dj;

public class DJTest {

    public static void main (String[] args) {
        BeatModelInterfejs model = new BeatModel();
        KontrolerInterfejs kontroler = new BeatKontroler(model);
    }
}
```

Model beatu

```
package headfirst.mvc.dj;

public interface BeatModelInterfejs {
    void inicjuj();

    void start();

    void stop();

    void ustawBPM(int bpm);

    int pobierzBPM();

    void zarejestrujObserwatora(ObserwatorUderzeń o);

    void wyrejestrujObserwatora(ObserwatorUderzeń o);

    void zarejestrujObserwatora(ObserwatorBPM o);

    void wyrejestrujObserwatora(ObserwatorBPM o);
}
```

```
package headfirst.mvc.dj;

import javax.sound.midi.*;
import java.util.*;

public class BeatModel implements BeatModelInterfejs, MetaEventListener {
    Sequencer sequencer;
    ArrayList obserwatorzyUderzeń = new ArrayList();
    ArrayList obserwatorzyBPM = new ArrayList();
    int bpm = 90;
// inne zmienne egzemplarza
    Sequence sequence;
    Track track;

    public void inicjuj() {
        konfigurujMidi();
        zbudujŚcieżkęIRozpocznij();
    }

    public void start() {
        sequencer.start();
        ustawBPM(90);
    }

    public void stop() {
        ustawBPM(0);
        sequencer.stop();
    }

    public void ustawBPM(int bpm) {
        this.bpm = bpm;
        sequencer.setTempoInBPM(pobierzBPM());
        powiadomObserwatorówBPM();
    }

    public int pobierzBPM() {
        return bpm;
    }

    void uderzenieEvent() {
        powiadomObserwatorówUderzeń();
    }


    public void zarejestrujObserwatora(ObserwatorUderzeń o) {
        obserwatorzyUderzeń.add(o);
    }

    public void powiadomObserwatorówUderzeń() {
        for(int i = 0; i < obserwatorzyUderzeń.size(); i++) {
```



Kod gotowy do użycia

```
            ObserwatorUderzeń obserwator = (ObserwatorUderzeń)obserwatorzyUderzeń.get(i);
            obserwator.aktualizujBeat();
        }
    }

    public void zarejestrujObserwatora(ObserwatorBPM o) {
        obserwatorzyBPM.add(o);
    }

    public void powiadomObserwatorówBPM() {
        for(int i = 0; i < obserwatorzyBPM.size(); i++) {
            ObserwatorBPM obserwator = (ObserwatorBPM)obserwatorzyBPM.get(i);
            obserwator.aktualizujBPM();
        }
    }


    public void wyrejestrujObserwatora(ObserwatorUderzeń o) {
        int i = obserwatorzyUderzeń.indexOf(o);
        if (i >= 0) {
            obserwatorzyUderzeń.remove(i);
        }
    }


    public void wyrejestrujObserwatora(ObserwatorBPM o) {
        int i = obserwatorzyBPM.indexOf(o);
        if (i >= 0) {
            obserwatorzyBPM.remove(i);
        }
    }

    public void meta(MetaMessage message) {
        if (message.getType() == 47) {
            uderzenieEvent();
            sequencer.start();
            ustawBPM(getBPM());
        }
    }

    public void konfigurujMidi() {
        try {
            sequencer = MidiSystem.getSequencer();
```

```
            sequencer.open();
            sequencer.addMetaEventListener(this);
            sequence = new Sequence(Sequence.PPQ,4);
            track = sequence.createTrack();
            sequencer.setTempoInBPM(getBPM());
        } catch(Exception e) {
                e.printStackTrace();
        }
    }

     public void zbudujŚcieżkęIRozpocznij() {
        int[] trackList = {35, 0, 46, 0};

        sequence.deleteTrack(null);
        track = sequence.createTrack();

          zbudujŚcieżki(trackList);
        track.add(zbudujMidiEvent(192,9,1,0,4));
         try {
            sequencer.setSequence(sequence);
        } catch(Exception e) {
            e.printStackTrace();
        }
    }

    public void zbudujŚcieżki(int[] lista) {

       for (int i = 0; i < lista.length; i++) {
          int key = lista[i];

          if (key != 0) {
             track.add(zbudujMidiEvent(144,9,key, 100, i));
             track.add(zbudujMidiEvent(128,9,key, 100, i+1));
          }
       }
    }

    public  MidiEvent zbudujMidiEvent(int polecenie, int kanał, int dane1, int dane2, int
czas) {
        MidiEvent event = null;
        try {
            ShortMessage a = new ShortMessage();
            a.setMessage(polecenie, kanał, dane1, dane2);
            event = new MidiEvent(a, czas);

        } catch(Exception e) {
            e.printStackTrace();
        }
        return event;
    }
}
```

Widok



Kod gotowy do użycia

```
package headfirst.mvc.dj;

public interface ObserwatorUderzeń {
    void aktualizujBeat();
}

package headfirst.mvc.dj;

public interface ObserwatorBPM {
    void aktualizujBPM();
}



package headfirst.mvc.dj;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class DJWidok implements ActionListener,  ObserwatorUderzeń, ObserwatorBPM {
    BeatModelInterfejs model;
    KontrolerInterfejs kontroler;
    JFrame widokFrame;
    JPanel widokPanel;
    BeatBar beatBar;
    JLabel bpmWyjścieLabel;
    JFrame sterowanieFrame;
    JPanel sterowaniePanel;
    JLabel bpmLabel;
    JTextField bpmTextField;
    JButton ustawBPMButton;
    JButton zwiększBPMButton;
    JButton zmniejszBPMButton;
    JMenuBar menuBar;
    JMenu menu;
    JMenuItem startMenuItem;
    JMenuItem stopMenuItem;

    public DJWidok(KontrolerInterfejs kontroler, BeatModelInterfejs model) {
        this.kontroler = kontroler;
        this.model = model;
        model.zarejestrujObserwatora((ObserwatorUderzeń)this);
        model.zarejestrujObserwatora((ObserwatorBPM)this);
    }

    public void utwórzWidok() {
```

```
    // Generowanie elementów sterujących (Swing)
    widokPanel = new JPanel(new GridLayout(1, 2));
    widokFrame = new JFrame("Widok");
    widokFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    widokFrame.setSize(new Dimension(100, 80));
    bpmWyjścieLabel = new JLabel("offline", SwingConstants.CENTER);
    beatBar = new BeatBar();
    beatBar.setValue(0);
    JPanel bpmPanel = new JPanel(new GridLayout(2, 1));
    bpmPanel.add(beatBar);
    bpmPanel.add(bpmWyjścieLabel);
    widokPanel.add(bpmPanel);
    widokFrame.getContentPane().add(widokPanel, BorderLayout.CENTER);
    widokFrame.pack();
    widokFrame.setVisible(true);
}


public void utwórzElementySterujące() {
    // Generowanie elementów sterujących (Swing)
    JFrame.setDefaultLookAndFeelDecorated(true);
    sterowanieFrame = new JFrame("Sterowanie");
    sterowanieFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    sterowanieFrame.setSize(new Dimension(100, 80));

    sterowaniePanel = new JPanel(new GridLayout(1, 2));

    menuBar = new JMenuBar();
    menu = new JMenu("DJ");
    startMenuItem = new JMenuItem("Start");
    menu.add(startMenuItem);
    startMenuItem.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent event) {
            kontroler.start();
        }
    });
    stopMenuItem = new JMenuItem("Stop");
    menu.add(stopMenuItem);
    stopMenuItem.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent event) {
            kontroler.stop();
            // bpmWyjścieLabel.setText("offline");
        }
    });
    JMenuItem exit = new JMenuItem("Zamknij");
    exit.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent event) {
            System.exit(0);
        }
    });
```



Kod gotowy do użycia

```
        menu.add(exit);
        menuBar.add(menu);
        sterowanieFrame.setJMenuBar(menuBar);

        bpmTextField = new JTextField(2);
        bpmLabel = new JLabel("BPM:", SwingConstants.RIGHT);
        ustawBPMButton = new JButton("Ustaw");
        ustawBPMButton.setSize(new Dimension(10,40));
        zwiększBPMButton = new JButton(">>");
        zmniejszBPMButton = new JButton("<<");
        ustawBPMButton.addActionListener(this);
        zwiększBPMButton.addActionListener(this);
        zmniejszBPMButton.addActionListener(this);

        JPanel przyciskiPanel = new JPanel(new GridLayout(1, 2));

        przyciskiPanel.add(zmniejszBPMButton);
        przyciskiPanel.add(zwiększBPMButton);

        JPanel wprowadzanieDanychPanel = new JPanel(new GridLayout(1, 2));
        wprowadzanieDanychPanel.add(bpmLabel);
        wprowadzanieDanychPanel.add(bpmTextField);
        JPanel sterowanieWewnPanel = new JPanel(new GridLayout(3, 1));
        sterowanieWewnPanel.add(wprowadzanieDanychPanel);
        sterowanieWewnPanel.add(ustawBPMButton);
        sterowanieWewnPanel.add(przyciskiPanel);
        sterowaniePanel.add(sterowanieWewnPanel);

        bpmLabel.setBorder(BorderFactory.createEmptyBorder(5,5,5,5));
        bpmWyjścieLabel.setBorder(BorderFactory.createEmptyBorder(5,5,5,5));

        sterowanieFrame.getRootPane().setDefaultButton(ustawBPMButton);
        sterowanieFrame.getContentPane().add(sterowaniePanel, BorderLayout.CENTER);

        sterowanieFrame.pack();
        sterowanieFrame.setVisible(true);
    }

    public void aktywujStopMenuItem() {
        stopMenuItem.setEnabled(true);
    }

    public void dezaktywujStopMenuItem() {
        stopMenuItem.setEnabled(false);
    }

    public void aktywujStartMenuItem() {
        startMenuItem.setEnabled(true);
    }
```

```
    public void dezaktywujStartMenuItem() {
        startMenuItem.setEnabled(false);
    }

    public void actionPerformed(ActionEvent event) {
        if (event.getSource() == ustawBPMButton) {
            int bpm = Integer.parseInt(bpmTextField.getText());
            kontroler.ustawBPM(bpm);
        } else if (event.getSource() == zwiększBPMButton) {
            kontroler.zwiększBPM();
        } else if (event.getSource() == zmniejszBPMButton) {
            kontroler.zmniejszBPM();
        }
    }

    public void aktualizujBPM() {
        if (model != null) {
            int bpm = model.getBPM();
            if (bpm == 0) {
                if (bpmWyjścieLabel != null) {
                    bpmWyjścieLabel.setText("offline");
                }
            } else {
                if (bpmWyjścieLabel != null) {
                    bpmWyjścieLabel.setText("Bieżąca wartość BPM: " + model.getBPM());
                }
            }
        }
    }

    public void aktualizujBeat() {
        if (beatBar != null) {
             beatBar.setValue(100);
        }
    }
}
```

Kontroler

```
package headfirst.mvc.dj;

public interface KontrolerInterfejs {
    void start();
    void stop();
    void zwiększBPM();
    void zmniejszBPM();
    void ustawBPM(int bpm);
}
```



Kod gotowy do użycia

```
package headfirst.mvc.dj;

public class BeatKontroler implements KontrolerInterfejs {
    BeatModelInterfejs model;
    DJWidok widok;

    public BeatKontroler(BeatModelInterfejs model) {
        this.model = model;
        widok = new DJWidok(this, model);
        widok.utwórzWidok();
        widok.utwórzElementySterujące();
        widok.dezaktywujStopMenuItem();
        widok.aktywujStartMenuItem();
        model.inicjuj();
    }

    public void start() {
        model.start();
        widok.dezaktywujStartMenuItem();
        widok.aktywujStopMenuItem();
    }

    public void stop() {
        model.stop();
        widok.dezaktywujStopMenuItem();
        widok.aktywujStartMenuItem();
    }

    public void zwiększBPM() {
        int bpm = model.pobierzBPM();
        model.ustawBPM(bpm + 1);
    }

    public void zmniejszBPM() {
        int bpm = model.pobierzBPM();
        model.ustawBPM(bpm - 1);
      }

     public void ustawBPM(int bpm) {
        model.ustawBPM(bpm);
    }
}
```

Model serca

```
package headfirst.mvc.dj;

public class SerceTest {

    public static void main (String[] args) {
        SerceModel serceModel = new SerceModel();
        KontrolerInterfejs model = new SerceKontroler(serceModel);
    }
}

package headfirst.mvc.dj;

public interface SerceModelInterfejs {
    int pobierzSzybkośćUderzeń();
    void zarejestrujObserwatora(ObserwatorUderzeń o);
    void wyrejestrujObserwatora(ObserwatorUderzeń o);
    void zarejestrujObserwatora(ObserwatorBPM o);
    void wyrejestrujObserwatora(ObserwatorBPM o);
}

package headfirst.mvc.dj;

import java.util.*;

public class SerceModel implements SerceModelInterfejs, Runnable {
    ArrayList obserwatorzyUderzeń = new ArrayList();
    ArrayList obserwatorzyBPM = new ArrayList();
    int czas = 1000;
    int bpm = 90;
    Random random = new Random(System.currentTimeMillis());
    Thread thread;

    public SerceModel() {
        thread = new Thread(this);
        thread.start();
    }

    public void run() {
        int ost_częstotliwość = -1;

        for(;;) {
            int zmiana = random.nextInt(10);
            if (random.nextInt(2) == 0) {
                zmiana = 0 - zmiana;
            }
            int częstotliwość = 60000/(czas + zmiana);
            if (częstotliwość < 120 && częstotliwość > 50) {
                czas += zmiana;
```



Kod gotowy do użycia

```
                powiadomObserwatorówUderzeń();
                if (częstotliwość != ost_częstotliwość) {
                    ost_częstotliwość = częstotliwość;
                    powiadomObserwatorówBPM();
                }
            }
            try {
                Thread.sleep(czas);
            } catch (Exception e) {}
        }
    }
    public int pobierzSzybkośćUderzeń() {
        return 60000/czas;
    }

    public void zarejestrujObserwatora(ObserwatorUderzeń o) {
        obserwatorzyUderzeń.add(o);
    }

    public void wyrejestrujObserwatora(ObserwatorUderzeń o) {
        int i = obserwatorzyUderzeń.indexOf(o);
        if (i >= 0) {
            obserwatorzyUderzeń.remove(i);
        }
    }

    public void powiadomObserwatorówUderzeń() {
        for(int i = 0; i < obserwatorzyUderzeń.size(); i++) {
            ObserwatorUderzeń obserwator = (ObserwatorUderzeń)obserwatorzyUderzeń.get(i);
            obserwator.aktualizujBeat();
        }
    }

    public void zarejestrujObserwatora(ObserwatorBPM o) {
        obserwatorzyBPM.add(o);
    }

    public void wyrejestrujObserwatora(ObserwatorBPM o) {
        int i = obserwatorzyBPM.indexOf(o);
        if (i >= 0) {
            obserwatorzyBPM.remove(i);
        }
    }

    public void powiadomObserwatorówBPM() {
        for(int i = 0; i < obserwatorzyBPM.size(); i++) {
            ObserwatorBPM obserwator = (ObserwatorBPM)obserwatorzyBPM.get(i);
            obserwator.aktualizujBPM();
        }
    }
}
```

Adapter klasy SerceModel

```
package headfirst.mvc.dj;

public class SerceAdapter implements BeatModelInterfejs {
    SerceModelInterfejs serce;

    public SerceAdapter(SerceModelInterfejs serce) {
        this.serce = serce;
    }

    public void inicjuj() {}

    public void start() {}

    public void stop() {}

    public int pobierzBPM() {
        return serce.pobierzSzybkośćUderzeń();
    }

    public void ustawBPM(int bpm) {}

    public void zarejestrujObserwatora(ObserwatorUderzeń o) {
        serce.zarejestrujObserwatora(o);
    }

    public void wyrejestrujObserwatora(ObserwatorUderzeń o) {
        serce.wyrejestrujObserwatora(o);
    }

    public void zarejestrujObserwatora(ObserwatorBPM o) {
        serce.zarejestrujObserwatora(o);
    }

    public void wyrejestrujObserwatora(ObserwatorBPM o) {
        serce.wyrejestrujObserwatora(o);
    }
}
```

Kontroler



Kod gotowy do użycia

```
package headfirst.mvc.dj;

public class SerceKontroler implements KontrolerInterfejs {
    SerceModelInterfejs model;
    DJWidok widok;

    public SerceKontroler(SerceModelInterfejs model) {
        this.model = model;
        widok = new DJWidok(this, new SerceAdapter(model));
        widok.utwórzWidok();
        widok.utwórzElementySterujące();
        widok.dezaktywujStopMenuItem();
        widok.dezaktywujStartMenuItem();
    }

    public void start() {}

    public void stop() {}

    public void zwiększBPM() {}

    public void zmniejszBPM() {}

    public void ustawBPM(int bpm) {}
}
```

Wzorce projektowe w praktyce



Ach, jesteś już gotowy na spotkanie z nowym wspaniałym światem pełnym wzorców projektowych... Ale zanim rozpoczniesz wędrówkę ku nowym horyzontom, poświęć chwilę na przeczytanie rozdziału poświęconego pewnym szczególnym kwestiom, które pojawiają się, gdy rozpoczynasz stosowanie wzorców w codziennej pracy. Nie wszędzie jest tak pięknie, jak w Obiektowie. Przygotowaliśmy więc mały przewodnik, który pomoże Ci odnaleźć się w twardej rzeczywistości...

Przewodnik na nowe życie z wzorcami



Przyjmij, proszę, nasz podręczny zbiór porad, które pomogą Ci żyć z wzorcami projektowymi w codziennej pracy i zmaganiach.

- Będziemy w nim mówić o:
  - nieporozumieniach związanych z terminem „wzorzec projektowy";
  - katalogach wzorców projektowych i o tym, jak bardzo mogą Ci się przydać;
  - unikaniu stosowania wzorców tam, gdzie nie jest to potrzebne;
  - właściwym klasyfikowaniu wzorców;
  - definiowaniu wzorców projektowych i o tym, że nie jest to zajęcie zarezerwowane dla wybranych. Dowiesz się, jak zostać autorem wzorców;
  - ćwiczeniach umysłu, które traktujemy z równą powagą jak mistrzowie Zen.
- Ujawnimy tożsamość Gangu Czterech — tajemniczego Gang of Four.
- Zdradzimy, jakie książki powinien mieć każdy użytkownik wzorców projektowych.
- Opowiemy, jak zdobyć przyjaciół i wpływać na programistów, stosując precyzyjną terminologię wzorców projektowych.

Wzorzec projektowy — definicja

Skoro czytasz przedostatni rozdział tej książki, masz już zapewne ukształtowany obraz tego, czym jest wzorzec projektowy. Warto jednak zwrócić uwagę, że nie przedstawiliśmy dotąd żadnej ściślejszej definicji tego pojęcia. Być może będziesz zaskoczony, gdy podamy ją w tak zwięzłej postaci:

> **Wzorzec projektowy** to rozwiązanie problemu w pewnym kontekście.

Bardzo jasna definicja, prawda? Przyjrzyjmy się jednak jej poszczególnym elementom, kontekstowi, problemowi i rozwiązaniu:

Przykład: masz kolekcję obiektów.

Kontekst to powtarzająca się sytuacja, w której można wzorzec zastosować.

Problem to cel, który chcemy osiągnąć w danym kontekście; w tym pojęciu mieszczą się również narzucane przez kontekst ograniczenia.

Potrzebujesz iterowania obiektów bez ujawniania implementacji tej kolekcji.

Rozwiązanie to jest to, czego szukamy: uogólniony projekt, który może zastosować każdy, kto dąży do tego samego celu, przy tych samych ograniczeniach.

Hermetyzujesz iterowanie w osobnej klasie.

Ta definicja należy do tych, z którymi należy się nieco oswoić. Nie śpiesz się. Oto wersja do zapamiętania:

> *„Jeżeli znajdziesz się w pewnym kontekście z problemem zdefiniowanym jako cel, który pokonuje określone ograniczenia, możesz zastosować projekt, który realizuje cel, uwzględnia ograniczenia i prowadzi do rozwiązania".*

W tym momencie może się już wydawać, że określenie, czym jest wzorzec projektowy, wymaga istotnego wysiłku. Ostatecznie, wiemy już, że wzorzec to rozwiązanie typowego, powtarzającego się problemu. Co daje cały ten formalizm? Okazuje się, że zastosowanie formalnych zasad opisywania wzorców pozwala utworzyć bardzo wygodny *katalog*.



Być może masz rację; zastanówmy się nad tym… Potrzebujemy *problemu*, *rozwiązania* i *kontekstu*:

Problem: Jak wykonać pracę na czas?

Kontekst: Zamknąłem kluczyki w samochodzie.

Rozwiązanie: Wybić szybę, dostać się do samochodu, uruchomić silnik i jechać do pracy.

Mamy wszystkie składniki definicji: mamy problem, a więc przede wszystkim cel (dojazd do pracy), ograniczenia czasowe, przestrzenne i zapewne jeszcze inne czynniki. Mamy też kontekst, charakteryzujący się tym, że nie możesz się dostać do kluczyków. Jest też rozwiązanie, prowadzące do odzyskania kluczyków i osiągnięcia celu mimo uwarunkowań czasowych i przestrzennych. To musi być wzorzec! Prawda?



WYTĘŻ UMYSŁ

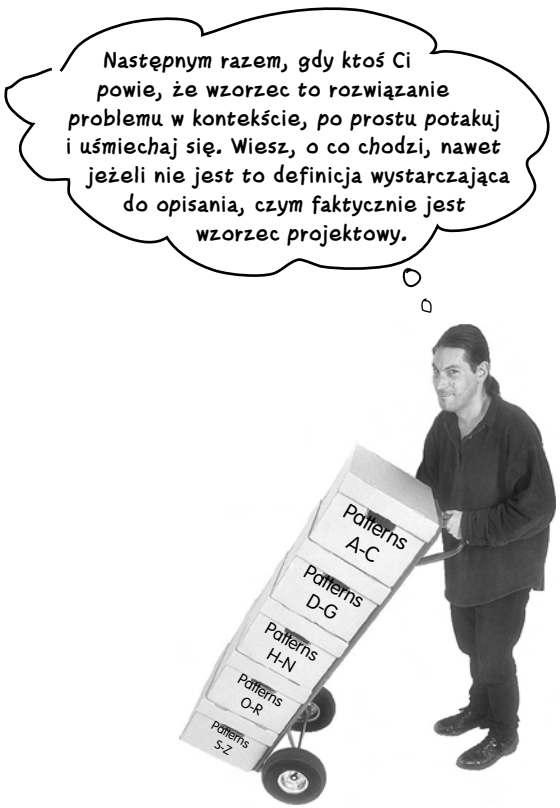
Podążając za definicją wzorca projektowego, zdefiniowaliśmy problem, kontekst i rozwiązanie (które działa!). Czy to jest wzorzec? Jeżeli nie, dlaczego? Czy definicja wzorca projektowania obiektowego ma jakieś luki?

Drugie spojrzenie na definicję wzorca

Nasz przykład wydaje się pasować do definicji wzorca projektowego, a jednak nim nie jest. Dlaczego? Przed wszystkim wiemy, że wzorzec ma mieć zastosowanie w przypadku powtarzającego się problemu. O ile roztargniona osoba może zamykać kluczyki w aucie z pewną regularnością, rozbijanie szyby nie kwalifikuje się jako rozwiązanie, które może być wielokrotnie stosowane (szczególnie, gdy weźmiemy pod uwagę ograniczenie, jakim jest koszt).

Przedstawiony przykład miał jeszcze inne braki: trudno wziąć taki opis, dać go komuś i polecić zastosowanie do rozwiązania jego własnego, niepowtarzalnego problemu. Pominęliśmy też prosty, a ważny element wzorca: nazwę! Bez nazwy wzorzec nie staje się częścią pewnego zasobu słownictwa, które może być używane w komunikacji z innymi programistami.

Jednak tak się szczęśliwie składa, że wzorce nie są opisywane w prostych ramach problemu, kontekstu i rozwiązania. Istnieją znacznie lepsze sposoby. Dodatkowo umożliwiają one gromadzenie wzorców w *katalogi*.



Nie istnieją głupie pytania

P: Czy znajdę gdzieś opisy wzorców z takim właśnie podziałem na problem, kontekst i rozwiązanie?

O: Opisy wzorców, które znajdziesz w katalogach, są zazwyczaj bardziej dopracowane. Katalogami zajmiemy się już za chwilę — mówią one przede wszystkim o celu i przeznaczeniu wzorca oraz o jego zastosowaniach. Zawierają też oczywiście projekt rozwiązania i informacje o konsekwencjach (pozytywnych i negatywnych) jego użycia.

P: Czy mogę nieco zmienić strukturę wzorca tak, aby pasowała do mojego projektu? Czy muszę ściśle trzymać się definicji?

O: Oczywiście, możesz zmienić strukturę. Podobnie jak zasady projektowania, wzorce to nie prawa czy sztywne reguły. Są to jedynie *zalecenia*, które można stosować odpowiednio do potrzeb. Jak się już przekonałeś, wiele przykładów z życia codziennego nie podąża za klasycznymi schematami zarysowanymi we wzorcach.

Gdy adaptujemy wzorzec, warto pamiętać o udokumentowaniu, w jaki sposób stosowane rozwiązanie różni się od standardowego — ułatwi to innym programistom rozpoznanie stosowanego wzorca razem z wprowadzonymi zmianami.

P: Gdzie znajdę katalog wzorców?

O: Najważniejszy katalog wzorców to zdecydowanie *Design Patterns: Elements of Rensable Object — Oriented Software* (Gamma, Helm, Johnson i Vlissides, wydane przez Addison Wesley). Zawiera on opisy dwudziestu trzech podstawowych schematów. Będziemy jeszcze pisać o tej książce.

Publikowanie katalogów skupionych wokół węższych dziedzin programowania — jak oprogramowanie dla przedsiębiorstw, systemy przetwarzania współbieżnego i systemy biznesowe — jest obecnie bardzo popularne.



Tylko dla wytrwałych

Niech moc będzie z Tobą

Definicja wzorca projektowego mówi nam, że *problem* składa się z *celu* i *zbioru ograniczeń*. Specjaliści od wzorców określają te elementy specjalną nazwą: siły (ang. *forces*). Dlaczego? No cóż, na pewno mają swoje powody, ale — jak może pamiętasz z filmu — „siła (moc) kształtuje i kontroluje Wszechświat". Podobnie siły, o których mowa w definicji wzorca, kształtują i wymuszają pewne rozwiązanie. Tylko zachowanie równowagi pomiędzy nimi (jasna strona mocy — Twój cel; ciemna strona — ograniczenia) sprawia, że rozwiązanie jest użytecznym wzorcem.

Mówienie o „siłach" może nieco zbijać z tropu, gdy pierwszy raz spotkamy się z nimi w literaturze. Pamiętajmy tylko, że istnieją dwa podstawowe ich kierunki (cele i ograniczenia). Wszystkie muszą być zrównoważone i uwzględnione w budowanym rozwiązaniu. Nie pozwól, żeby żargon zakłócił Twój osąd, i niech moc będzie z Tobą!



Franek: Powiedz nam, Janusz, o co chodzi. Uczyłem się o wzorcach, czytając po prostu różne artykuły.

Janusz: Jasne. Katalog wzorców to zbiór szczegółowych opisów, w których są też informacje o powiązaniach z innymi wzorcami.

Jacek: Więc takich katalogów jest wiele?

Janusz: Oczywiście. Są katalogi podstawowych wzorców projektowych i katalogi wzorców specyficznych dla pewnych dziedzin, takich jak na przykład EJB.

Franek: Który katalog tu masz?

Janusz: To tradycyjny katalog grupy GoF. Zawiera opisy dwudziestu trzech wzorców podstawowych.

Franek: GoF?

Janusz: Zgadza się. To skrót od Gang Of Four, co można tłumaczyć jako Gang Czterech. To autorzy pierwszego w historii katalogu wzorców.

Jacek: Co jest w tym katalogu?

Janusz: Zbiór powiązanych wzorców. Każdy wzorzec jest opisany według tego samego szablonu. Pierwszą ważną rzeczą jest *nazwa* każdego z nich.

Franek: To niesamowite — nazwa! Kto by pomyślał…

Janusz: Moment, Franek, nazwa naprawdę jest ważna. Gdy mamy nazwę, możemy rozmawiać o wzorcu. To kwestia wspólnego języka.

Franek: W porządku, tylko żartowałem. A więc, co tam jeszcze jest?

Janusz: A więc, tak jak mówiłem, opisy wzorców trzymają się pewnego szablonu. Każdy z nich zawiera nazwę i obejmuje kilka kolejnych działów z informacjami o danym wzorcu. Jest na przykład dział „Cel", w którym jest napisane, czym jest dany wzorzec. To coś w rodzaju definicji. Są też działy „Motywacja" i „Zastosowania", gdzie znajdziemy wskazówki, kiedy i gdzie można użyć tego właśnie wzorca.

Jacek: A co z właściwym projektem?

Janusz: Jest kilka działów, które opisują układ klas, poszczególne klasy z osobna i ich role. Jest też dział poświęcony temu, jak implementować wzorzec, a często również przykładowy kod.

Franek: Wygląda, jakby pomyśleli o wszystkim.

Janusz: Nawet więcej. Są też przykłady użycia wzorców w prawdziwych systemach oraz część, która wydaje mi się jedną z najbardziej przydatnych — mówiąca o tym, jak dany wzorzec jest powiązany z *innymi*.

Franek: Tam piszą takie rzeczy, jak to, czym różnią się wzorce *Stan* i *Strategia*?

Janusz: Dokładnie!

Jacek: A więc, Janusz, jak właściwie korzystać z takiego katalogu? Gdy masz pewien problem, po prostu otwierasz katalog i szukasz rozwiązania?

Janusz: Przede wszystkim próbuję zapoznać się z wszystkimi wzorcami i ich wzajemnymi relacjami. Potem, kiedy potrzebuję pewnego wzorca, mam pojęcie o tym, czym dysponuję. Wracam do katalogu i przeglądam działy „Motywacja" i „Zastosowania", aby upewnić się, czy wszystko dobrze rozumiem. Jest tam też jeszcze jeden bardzo ważny dział: „Konsekwencje". Przeglądam go, aby upewnić się, czy zastosowanie wzorca nie doprowadzi do niepożądanych efektów.

Franek: To ma sens. Więc kiedy już wiesz, że dany wzorzec jest właściwy w danej sytuacji, jak wprowadzasz go do projektu?

Janusz: Tu przydaje się diagram klas. Rozpoczynam od działu „Struktura" i przeglądania diagramu. Kolejnym krokiem jest dział „Uczestnicy", gdzie mogę upewnić się, czy dobrze rozumiem role poszczególnych klas. Dopiero wówczas rozpoczynam wprowadzanie wzorca do projektu, uwzględniając niezbędne do jego dopasowania zmiany. Przeglądam jeszcze działy „Implementacja" i „Przykładowy kod", aby upewnić się, że zapoznałem się z dobrymi metodami implementacji i typowymi niuansami tego etapu.

Jacek: Teraz widzę, w jaki sposób katalog może przyspieszyć moją pracę z wzorcami!

Franek: Bardzo. Janusz, możesz w sposób systematyczny omówić opis wzorca?

Opis każdego wzorca rozpoczyna nazwa. Jest to bardzo ważny element — bez odpowiedniej nazwy wzorzec nie może stać się częścią słownictwa, którego używamy w komunikacji z innymi programistami.

Dział Motywacja zawiera pewien opis sytuacji — opis problemu i zasady jego rozwiązania.

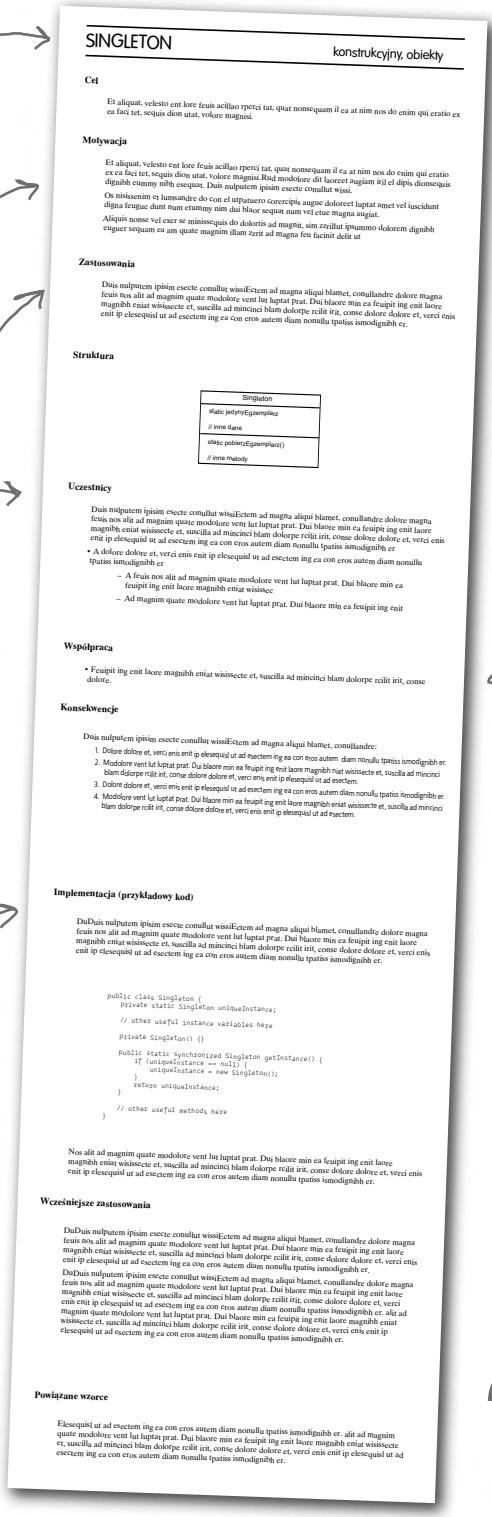
Dział Zastosowania zawiera bardziej szczegółowe omówienie sytuacji, w których wzorzec można zastosować.

Dział Uczestnicy zawiera klasy i obiekty tworzące dany wzorzec. Tu znajdziemy opis ich zakresów odpowiedzialności i ról.

Dział Konsekwencje omawia potencjalne skutki użycia wzorca — pożądane i niepożądane.

Dział Implementacja zawiera informacje o tym, jak może wyglądać kod rozwiązania; mamy tu też omówienie zagadnień, na które trzeba zwrócić szczególną uwagę.

Dział Wcześniejsze zastosowania zawiera przykłady użycia wzorca w istniejących systemach.



SINGLETON

konstrukcyjny, obiekty

Cel

Et aliquat, velesto ent lore feuis acillao rperci tat, quat nonsequam il ea at nim nos do enim qui eratio ex ea faci tet, sequis dion utat, volore magnisi.

Motywacja

Et aliquat, velesto eat lore feuis acillao rperci tat, quat nonsequam il ea at nim nos do enim qui eratio ex ea faci tet, sequis dion utat, volore magnisi.Riuq modolore dit laoreet augiam iril el dipis dionsequis dignibh eiusmy nibh esequat. Duis nulputem ipisim esecte comullut wissi.

Os nisissenim et lumsandre do con el utpatuero corercipis augue doloreet luptat amet vel luscidunt digna feuigue dunt num etuminy nim dui blaor sequat num vel etue magna augiat.

Aliquis nonse vel exer se minissequis do dolortis ad magna, sim zzrillut ipsummo dolorem dignibh euguer sequam ea am quate magnim illam zzrit ad magna feu facinit delit ut

Zastosowania

Duis nulputem ipisim esecte conullut wissiEctem ad magna aliqui blamet, conullandre dolore magna feuis nos alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er.

Struktura

Uczestnicy

Duis nulputem ipisim esecte conullut wissiEctem ad magna aliqui blamet, conullandre dolore magna feuis nos alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er

- A dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er
  - A feuis nos alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissec
  - Ad magnim quate modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit

Współpraca

- Feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore.

Konsekwencje

Duis nulputem ipisim esecte conullut wissiEctem ad magna aliqui blamet, conullandre:

1. Dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er.
2. Modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh niat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er.
3. Dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er.
4. Modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem.

Implementacja (przykładowy kod)

DuDuis nulputem ipisim esecte conullut wissiEctem ad magna aliqui blamet, conullandre dolore magna feuis nos alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er.

```
public class Singleton {
    Private static Singleton uniqueInstance;

    // other useful instance variables here

    Private Singleton() {}

    Public static synchronized Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }

    // other useful methods here
}
```

Nos alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er.

Wcześniejsze zastosowania

DuDuis nulputem ipisim esecte conullut wissiEctem ad magna aliqui blamet, conullandre dolore magna feuis nos alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh et.

DuDuis nulputem ipisim esecte conullut wissiEctem ad magna aliqui blamet, conullandre dolore magna feuis nos alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er. alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er.

Powiązane wzorce

Elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er. alit ad magnim quate modolore vent lut luptat prat. Dui blaore min ea feuipit ing enit laore magnibh eniat wisissecte et, suscilla ad mincinci blam dolorpe rcilit irit, conse dolore dolore et, verci enis enit ip elesequisl ut ad esectem ing ea con eros autem diam nonullu tpatiss ismodignibh er.

To jest dział klasyfikacja, wskazujący kategorię, do której wzorzec należy. Tym elementem zajmiemy się jeszcze dokładniej.

Dział Cel zawiera krótki opis funkcji wzorca. Inaczej mówiąc, jest to jego definicja (tak jak definicje przedstawiane w tej książce).

Dział Struktura zawiera diagram ilustrujący związki między klasami wzorca.

Dział Współpraca zawiera omówienie współpracy obiektów lub (i) klas wzorca.

Dział Przykładowy kod zawiera fragmenty kodu, które mogą pomóc w tworzeniu własnej implementacji.

Dział Powiązane wzorce zawiera opis związków i zależności między opisywanym właśnie wzorcem a innymi wzorcami.

Nie istnieją głupie pytania

P: Czy można tworzyć własne, nowe wzorce projektowe? Albo co trzeba zrobić, żeby zostać specjalistą od wzorców?

O: Przede wszystkim pamiętaj, że wzorce się *odkrywa*, a nie tworzy. Każdy może więc odkryć nowy wzorzec i zostać *autorem* jego opisu. Nie jest to jednak proste, wymaga czasu i nie zdarza się często. Aby zostać „autorem wzorców", niezbędne jest mocne zaangażowanie.

Powinieneś zastanowić się nad tym, dlaczego chciałbyś to robić. Większość ludzi nie opisuje wzorców, ale *używa* ich. Może się okazać, że pracujesz w pewnej specyficznej dziedzinie, w której opracowania nowych wzorców są bardzo przydatne. Być może po prostu natrafiłeś na rozwiązanie problemu, który, jak Ci się wydaje, jest dość typowy. A może po prostu chcesz dołączyć do społeczności programistów, którzy zajmują się wzorcami i wziąć udział w jej pracach?

P: Jestem zupełnie nowy w tym temacie. Od czego zacząć?

O: Podobnie jak w każdej innej dyscyplinie, im więcej wiesz, tym lepiej. Studiowanie dostępnych opisów wzorców, tego, co robią i jak można je odnieść do innych, powinno być Twoim podstawowym zajęciem. Nie tylko pozwoli to zapoznać się z charakterystyką opisów i samych wzorców, ale uchroni przed przysłowiowym „odkrywaniem Ameryki" na nowo. W pewnym momencie możesz rozpocząć spisywanie swoich wzorców i wykorzystywanie ich w pracy z innymi programistami. Za chwilę zajmiemy się jeszcze zagadnieniem przekazywania własnej wiedzy o wzorcach innym programistom.

P: Skąd będę wiedział, że naprawdę mam do czynienia z nowym wzorcem?

O: Dobre pytanie. Wzorzec nie istnieje do chwili, gdy inni skorzystają z niego i potwierdzą jego działanie. Ogólnie, obowiązuje tzw. „reguła trzech". Mówi ona, że pewien schemat można uznać za wzorzec projektowy, jeżeli został użyty w praktyce co najmniej trzy razy.

A więc chcesz zostać autorem wzorców?

No to posłuchaj.

Weź katalog wzorców.

Naucz się wszystkiego, co w nim napisane.

Weź się do opisów.

Jeżeli trzech stwierdzi, że wszystko jest dograne.

To wzorzec gotowy.

Na melodię „So you wanna be a Rock'n'Roll Star".

Zostać autorem wzorców projektowych

Odrób pracę domową. Zanim stworzysz własny wzorzec, musisz dobrze znać te, które zostały opisane wcześniej. Większość wzorców, które sprawiają wrażenie nowych, to w rzeczywistości odmiany tych, które zostały opracowane wcześniej. Studiując różne opisy, nabierzesz wprawy w rozpoznawaniu takich powtórzeń i nauczysz się szukać powiązań.

Daj sobie czas na zastanowienie i testy. Twoje doświadczenie — rozwiązane problemy, wykorzystane rozwiązania — jest właściwym miejscem narodzin nowych wzorców. To skłania do uważnego analizowania własnej pracy i wyszukiwania uogólnionych, wielokrotnie powtarzających się schematów. Pamiętajmy, że większość projektów nie jest niczym nowym, ale jedynie modyfikacją znanych już wzorców. Nawet, gdy znajdziemy układ, który mógłby być nowym wzorcem, jego zakres zastosowań może być zbyt wąski, aby potraktować go jako prawdziwy wzorzec projektowy.

Zapisz swoje pomysły w taki sposób, aby inni mogli je zrozumieć. Wyszukiwanie nowych wzorców nie miałoby sensu, gdyby inni nie mogli z nich korzystać. Potencjalne wzorce muszą zostać udokumentowane, aby inni mogli je przeczytać, zrozumieć i zastosować we własnych rozwiązaniach. Zapewni to niezbędne informacje zwrotne. Szczęśliwie się składa, że nie trzeba samodzielnie wymyślać metody opisywania wzorców. Jak widzieliśmy na przykładzie szablonu GoF, zasady opisywania wzorców zostały już bardzo dobrze przemyślane.

Niech inni spróbują zastosować Twoje wzorce; na pewno przyczynią się do ich dopracowania. Nie oczekuj, że Twój wzorzec będzie idealny i gotowy już przy „pierwszym podejściu". Będzie to raczej długotrwała praca, prowadząca do kolejnych usprawnień. Niech inni programiści ocenią wzorzec, wypróbują go, przekażą swoje komentarze. Włącz uzyskane informacje do opisu i znowu przekaż go innym. Opis nigdy nie będzie idealny, ale w pewnym momencie uzyska wystarczającą spójność i kompletność, aby różni programiści z łatwością mogli zapoznać się z nim i wykorzystać wzorzec w praktyce.

Nie zapominaj o „regule trzech". Pamiętaj, że Twój opis można uznać za wzorzec dopiero, gdy zostanie zastosowany w trzech prawdziwych aplikacjach. To kolejny powód, aby przekazywać jego opis innym programistom — aby go sprawdzić, uzyskać nowe komentarze i stopniowo posuwać się w kierunku „formalnego" uznania naszego schematu za wzorzec projektowy.





Dopasuj wzorce do opisów:

Wzorzec	Opis
Dekorator	Osłania inny obiekt i zapewnia inny interfejs.
Stan	Klasy podrzędne decydują o tym, w jaki sposób zostaną zaimplementowane poszczególne kroki algorytmu.
Iterator	Klasy podrzędne decydują o tym, która klasa rzeczywista zostanie użyta do utworzenia obiektu.
Fasada	Zapewnia utworzenie dokładnie jednego egzemplarza obiektu.
Strategia	Hermetyzuje wymienne zachowania i wykorzystuje mechanizm delegacji do wyboru, które zachowanie zostanie użyte.
Proxy	Klienty dysponują ujednoliconym dostępem do kolekcji obiektów i pojedynczych obiektów.
Metoda Fabrykująca	Hermetyzuje zachowania stanowe i wykorzystuje delegację do określania, które z nich powinno zostać zastosowane.
Adapter	Zapewnia możliwość przeglądania kolekcji obiektów bez wiedzy o jej implementacji.
Obserwator	Upraszcza interfejs grupy klas.
Metoda Szablonowa	Osłania inny obiekt i zapewnia dodatkowe zachowania.
Kompozyt	Umożliwia klientowi tworzenie rodzin obiektów bez określania ich klas rzeczywistych.
Singleton	Umożliwia powiadamianie grupy obiektów o zmianie stanu.
Fabryka Abstrakcyjna	Osłania inny obiekt, aby kontrolować dostęp do niego.
Polecenie	Hermetyzuje żądanie w postaci obiektu.

Porządkowanie wzorców projektowych

W miarę, jak rośnie liczba odkrytych wzorców projektowych, wagi nabiera potrzeba wprowadzenia pewnej klasyfikacji, która umożliwi ich uporządkowanie, pozwoli zawęzić zakres poszukiwań i ułatwi porównania przeprowadzane w granicach pewnej kategorii.

Katalogi wzorców operują kilkoma różnymi klasyfikacjami. Najbardziej znana pochodzi z pierwszego katalogu. Wyróżnia on trzy kategorie, a podział opiera na przeznaczeniu wzorca. Trzy grupy to wzorce konstrukcyjne (ang. *creational*), operacyjne (ang. *behavioral*) i strukturalne (ang. *structural*).

Zaostrz ołówek

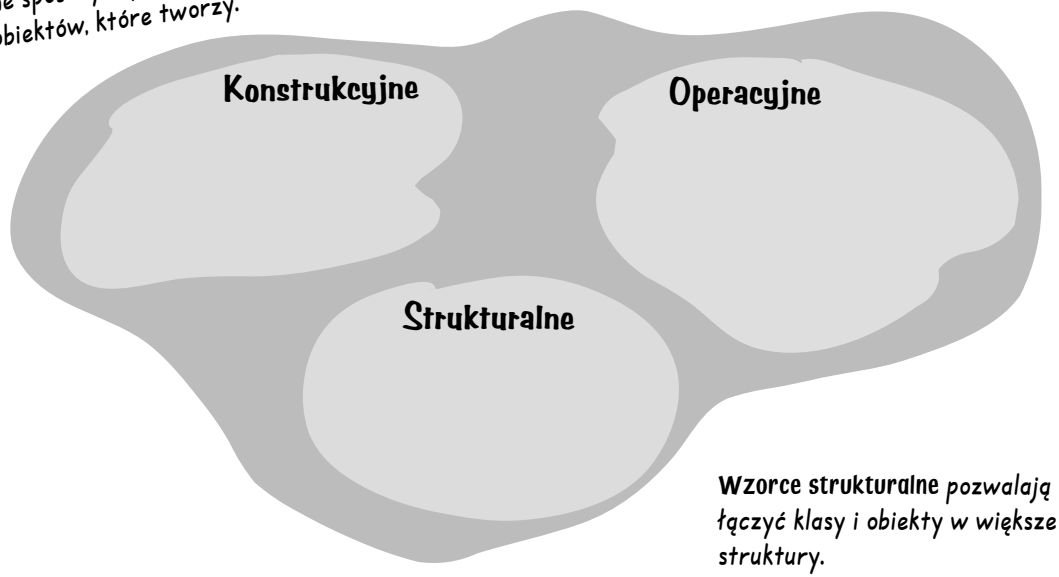
Przeczytaj opis każdej kategorii i sprawdź, czy potrafisz przypisać do nich wszystkie wzorce. To nie jest takie proste! Postaraj się zrobić to najlepiej, jak umiesz, i dopiero wtedy obejrzyj właściwe rozwiązanie łamigłówki, przedstawione na następnej stronie.

Fabryka Abstrakcyjna
Kompozyt
Obserwator
Strategia
Dekorator
Stan
Adapter
Singleton
Metoda Fabrykująca
Proxy
Metoda Szablonowa
Polecenie
Iterator
Fasada

Każdy z tych wzorców należy do jednej z kategorii.

Wzorce konstrukcyjne prowadzą do utworzenia egzemplarza obiektu. Wszystkie można określić jako różne sposoby separowania klienta od obiektów, które tworzy.

Wzorce operacyjne *dotyczą interakcji klas i obiektów oraz podziału odpowiedzialności.*



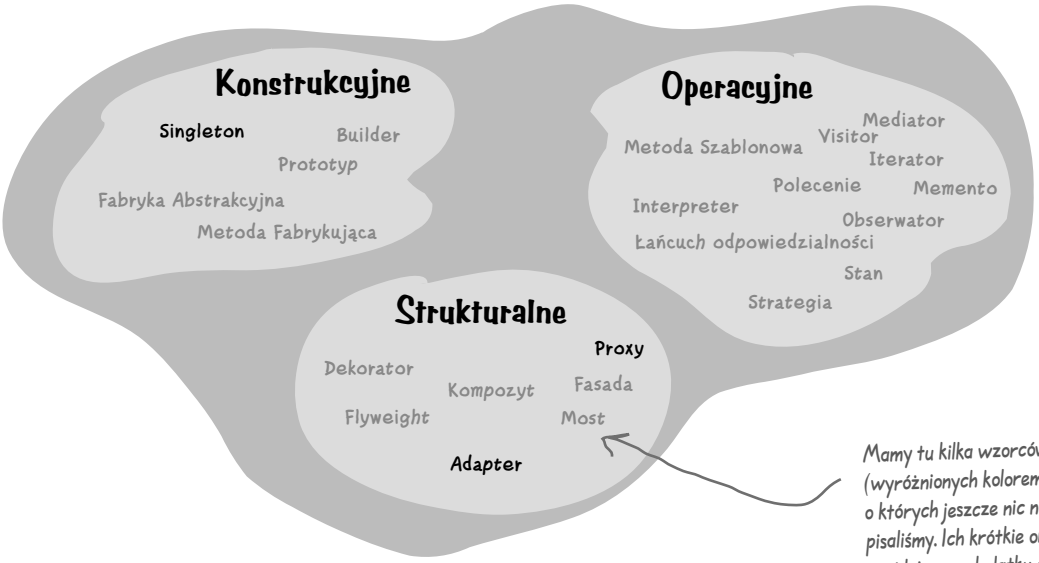
Wzorce strukturalne *pozwalają łączyć klasy i obiekty w większe struktury.*

Rozwiązanie — kategorie wzorców

Oto nasze wzorce pogrupowane w kategorie. Ćwiczenie z poprzedniej strony było zapewne trudne. Wiele wzorców wydaje się pasować do więcej niż jednej kategorii. Porażką nie powinniśmy się jednak przejmować. Wszyscy mają problemy z dopasowaniem wzorców do właściwych kategorii.

Wzorce konstrukcyjne *dotyczą interakcji klas i obiektów oraz podziału odpowiedzialności.*

Wzorce operacyjne *dotyczą interakcji klas i obiektów oraz podziału odpowiedzialności.*



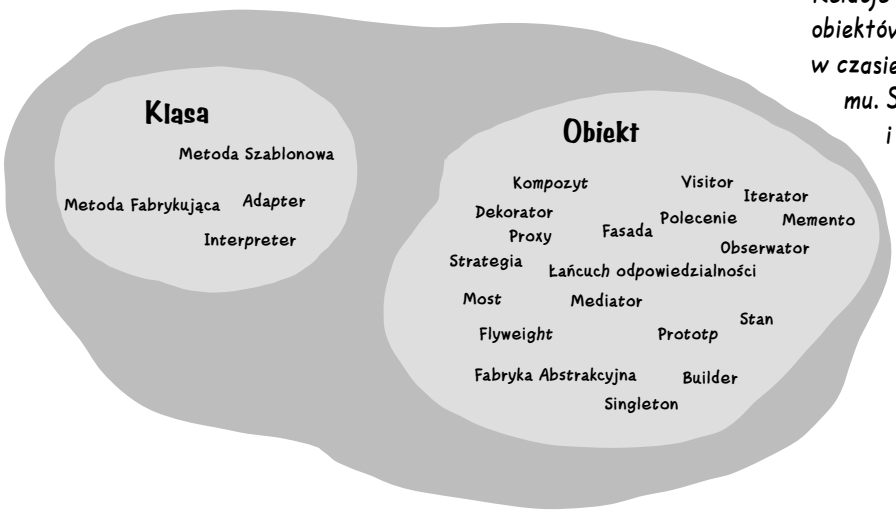
Mamy tu kilka wzorców (wyróżnionych kolorem szarym), o których jeszcze nic nie pisaliśmy. Ich krótkie omówienie znajdziesz w dodatku do książki.

Wzorce strukturalne *pozwalają łączyć klasy i obiekty w większe struktury.*

Wzorce klasyfikuje się też często w oparciu o ich drugi atrybut: o to, czy operują klasami, czy obiektami:

Wzorce klas *opisują, w jaki sposób dziedziczenie definiuje związki między klasami. Relacje wzorców klas ustanawiane są w czasie kompilowania kodu.*

Wzorce obiektów *opisują związki między obiektami i są definiowane głównie przez kompozycję. Relacje opisane we wzorcach obiektów powstają najczęściej w czasie wykonywania programu. Są bardziej dynamiczne i elastyczne.*



Zwróć uwagę, że jest znacznie więcej wzorców obiektów niż klas!

Nie istnieją głupie pytania

P: Czy to jedyne systemy klasyfikacji?

O: Nie, można spotkać się z innymi. Część z nich mówi o tych samych trzech kategoriach głównych i wprowadza podkategorie, takie jak „wzorce separacji". Warto zapoznać się z kilkoma takimi uporządkowaniami. Nic, oczywiście, nie stoi na przeszkodzie, żeby stworzyć swoją własną, inną klasyfikację, o ile tylko pomoże ona w nauce i praktycznym stosowaniu wzorców.

P: Czy porządkowanie wzorców w kategorie naprawdę ułatwia ich zapamiętanie?

O: Niewątpliwie jest to pewien schemat ułatwiający porównania. Trzeba jednak przyznać, że podział na wzorce konstrukcyjne, strukturalne i operacyjne nie zawsze jest oczywisty. Jeden wzorzec wydaje się często pasować do więcej niż jednej kategorii. Najważniejsze jest to, aby znać same wzorce i ich wzajemne relacje. Kategorii używamy wtedy, gdy są faktycznie pomocne!

P: Jak to się stało, że wzorzec Dekorator trafił do kategorii wzorców strukturalnych? Powiedziałbym raczej, że to wzorzec operacyjny; ostatecznie, jego funkcją jest wzbogacanie zachowań!

O: Tak, wielu programistów tak mówi! A oto, co stoi za klasyfikacją GoF: wzorce strukturalne opisują, w jaki sposób klasy i obiekty zostają połączone w celu utworzenia nowych struktur i funkcji. Wzorzec Dekorator prowadzi do układu obiektów, w którym jeden obiekt osłania inny, dostarczając nowych funkcji. Jego osią jest więc dynamiczne tworzenie układu obiektów, który dysponuje nowymi funkcjami. Nie chodzi tu o komunikację czy połączenia między obiektami, a więc zagadnienia charakterystyczne dla wzorców operacyjnych. Różnica jest dość subtelna, zwłaszcza gdy zwrócimy uwagę na strukturalne podobieństwa wzorców Dekorator (strukturalny) i Proxy (operacyjny). Pamiętajmy jednak, że różni się cel tych wzorców, a to on jest najczęściej kluczem do właściwej klasyfikacji.



Mistrz i uczeń...

Mistrz: Młody człowieku, wyglądasz na zafrasowanego.

Uczeń: Tak, dowiedziałem się właśnie o klasyfikacjach wzorców i nie bardzo to rozumiem.

Mistrz: Ach, adepcie, powiedz mi coś więcej...

Uczeń: Po tym, jak dowiedziałem się już tyle o wzorcach, słyszę teraz, że każdy z nich należy do jednej z kategorii: strukturalne, operacyjne i konstrukcyjne. Po co w ogóle te klasyfikacje?

Mistrz: Młody człowieku, gdy operujemy dużym zbiorem czegoś, to naturalne, że próbujemy stworzyć pewną klasyfikację. To pomaga rozważać te obiekty na bardziej abstrakcyjnym poziomie.

Uczeń: Mistrzu, możesz dać mi jakiś przykład?

Mistrz: Oczywiście. Weźmy samochody. Jest wiele różnych modeli samochodów. Niemal automatycznie łączymy je w kategorie: samochody ekonomiczne, sportowe, dostawcze, luksusowe...

Mistrz: Młody człowieku, wyglądasz na zdziwionego: czy to, co mówię, jest mało zrozumiałe?

Uczeń: Mistrzu, rozumiem, oczywiście, ale nie wiedziałem, że tyle wiesz o samochodach!

Mistrz: Młody człowieku, nie mogę porównywać wszystkiego do kwiatu lotosu i miseczek ryżu. Mogę kontynuować?

Uczeń: Tak, tak, przepraszam, kontynuuj.

Mistrz: Gdy mamy w pamięci pewną klasyfikację rzeczy albo ich podział na kategorie, możemy łatwiej rozmawiać o różnych grupach: „Jeśli jedziemy górami z Doliny Krzemowej do Santa Cruz, samochód sportowy będzie najlepszy" lub „Pogarszająca się sytuacja przemysłu paliwowego sprawia, że coraz więcej ludzi decyduje się na zakup samochodów ekonomicznych, o mniejszym zużyciu paliwa".

Uczeń: Więc gdy mamy kategorie, możemy rozmawiać o zbiorze wzorców jako o pewnej grupie. Może się zdarzyć, że wiemy już, że potrzebujemy wzorca konstrukcyjnego, ale nie wiemy jeszcze, jakiego. Umożliwia to jednak rozpoczęcie rozmowy o wzorcach konstrukcyjnych.

Mistrz: Tak, umożliwia to też porównywanie jednego elementu z pozostałymi w tej kategorii. Przykładowo: „Mini to najbardziej stylowy z dostępnych samochodów kompaktowych". A gdy chcemy znaleźć najlepszy model i próbujemy zawęzić zakres poszukiwań, stwierdzamy: „Potrzebuję samochodu ekonomicznego".

Uczeń: Rozumiem, więc mogę powiedzieć, że wzorzec Adapter to najlepszy wzorzec strukturalny do zmieniania interfejsu obiektu.

Mistrz: Tak. Możemy też wykorzystać kategorie do innego celu: aby stworzyć nowy obszar. Dajmy na to, że chcemy stworzyć samochód sportowy o możliwościach Ferrari, a cenie Daewoo.

Uczeń: To brzmi jak śmiertelna pułapka.

Mistrz: Przepraszam, nie dosłyszałem.

Uczeń: Eee... powiedziałem, że rozumiem.

Uczeń: A więc kategorie to pewna metoda ułatwiająca rozważania o relacjach między grupami wzorców i wewnątrz tych grup. Służy też ekstrapolacji do nowych wzorców. A dlaczego są tylko trzy kategorie, a nie cztery lub pięć?

Mistrz: Ach, tak jak gwiazd na niebie, kategorii będzie tyle, ile zechcesz zobaczyć. Trzy to poręczna liczba i wiele osób stwierdziło, że taki podział jest dobry. Oczywiście, byli też i tacy, którzy sugerowali operowanie czterema, pięcioma lub większą ich liczbą.



Myślenie wzorcami

Konteksty, ograniczenia, siły, katalogi, klasyfikacje… człowieku, to zaczyna brzmieć strasznie akademicko. W porządku, wszystko to jest ważne, a wiedza to potęga. Z drugiej strony, nie można zapominać, że sama wiedza akademicka bez *doświadczenia* i praktyki w stosowaniu wzorców niewiele zmieni w Twoim życiu.



Twój mózg pełen wzorców

Oto krótki przewodnik, który pomoże Ci *myśleć wzorcami*. Co przez to rozumiemy? Chodzi o umiejętność spojrzenia na projekt i rozpoznania miejsc, do których wzorce w naturalny sposób pasują.

Zasada prostoty

Przede wszystkim, w całym procesie projektowania powinniśmy trzymać się zasady stosowania najprostszych możliwych rozwiązań. Prostota powinna być celem zasadniczym. Nie można podążać ścieżką zastanawiania się „jak zastosować wzorzec do rozwiązania tego problemu?". Nie należy myśleć, że człowiek nie jest poważnym projektantem, jeżeli nie zastosował żadnego wzorca. Każdy programista doceni prostotę rozwiązań. To powiedziawszy, musimy stwierdzić, że często zastosowanie wzorca jest najlepszym sposobem zachowania prostoty i elastyczności.

Wzorce projektowe to nie magiczna różdżka

Wzorce, jak wiemy, to ogólne rozwiązania powtarzających się problemów. Jedną z ich zalet jest to, że zostały wielokrotnie sprawdzone przez innych programistów. Gdy więc widzimy potrzebę zastosowania jednego z nich, możemy spać spokojnie — wielu programistów już znalazło się w tym punkcie i rozwiązało problem w podobny sposób.

Z drugiej jednak strony, wzorce nie czynią cudów. Można ich użyć, skompilować program i wyjść wcześniej do domu. Nigdy nie należy jednak zapominać, że gdy korzystamy ze wzorców, musimy rozważyć konsekwencje ich wprowadzenia dla pozostałych elementów projektu.

Potrzebujesz wzorca, kiedy...

Ach… najważniejsze pytanie: kiedy powinniśmy użyć wzorca projektowego? Należy go użyć wtedy, gdy jesteśmy pewni, że rozwiązuje problem, na który natrafiliśmy w projekcie. Jeżeli prostsze rozwiązanie będzie wystarczające, rozważ je dobrze, zanim uciekniesz się do implementowania wzorców.

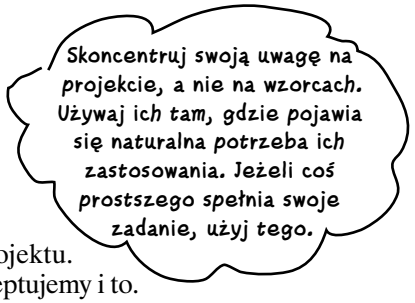
Wiedza o tym, w jakich sytuacjach dany wzorzec ma zastosowanie, to kwestia doświadczenia i umiejętności. Gdy już wiesz na pewno, że proste rozwiązanie nie wystarczy, powinieneś rozważyć sytuację w kategoriach problemu i zbioru ograniczeń, które trzeba uwzględnić w rozwiązaniu — to pomoże dopasować zagadnienie do wzorca. Jeśli dobrze znasz wzorce, pomysł użycia jednego z nich może nasunąć się sam. W innych przypadkach przeglądasz wzorce, skupiając się na tych, które dają nadzieję rozwiązania problemu. Wówczas czytasz tę część ich opisu, która mówi o celu i zastosowaniach. Gdy już znajdziesz wzorzec, który wydaje się pasować, upewnij się, że efekty jego użycia są akceptowalne. Zwróć też uwagę na wpływ takiego rozwiązania na inne elementy projektu. Jeżeli wszystko wygląda dobrze, możesz przejść do implementacji!

Istnieje jeden typ sytuacji, kiedy warto użyć wzorca nawet, gdy prostsze rozwiązanie również spełnia swoje zadanie: gdy oczekujemy, że różne aspekty systemu będą się zmieniać. Jak mieliśmy okazję się przekonać, identyfikowanie obszarów zmian w projekcie to często sygnał, że warto zastosować wzorzec. Dbajmy tylko o to, aby wprowadzać wzorce, które przygotują nas na *zmiany praktyczne*, których można oczekiwać, a nie na *zmiany hipotetyczne*, które po prostu mogą się pojawić.

Etap projektowania nie jest jedynym, podczas którego rozważamy wprowadzanie wzorców. Są one równie istotne na etapie refactoringu.

Czas refactoringu to czas wzorców projektowych!

Refactoring to procedura wprowadzania w kodzie takich zmian, które mają na celu zwiększenie jego uporządkowania. Celem jest poprawa struktury, a nie zmiana zachowań. To doskonały moment, aby ponownie przeanalizować projekt i wyszukać możliwości jego lepszej strukturalizacji poprzez wprowadzenie wzorców projektowych. Przykładowo, w kodzie jest wiele instrukcji warunkowych, co może sygnalizować potrzebę wprowadzenia wzorca Stan. Być może warto uporządkować kwestię zależności od klas rzeczywistych poprzez wprowadzenie wzorca Fabryka. Na temat stosowania wzorców w trakcie refactoringu napisano już całe książki. Jeśli znamy już podstawy stosowania wzorców, warto zainteresować się tym tematem bliżej.



Usuń wszystko, co nie jest niezbędne
Nie obawiaj się usuwania wzorców

W źródłach pisanych brak jakichkolwiek informacji o usuwaniu wzorców z projektu. Można by pomyśleć, że to bluźnierstwo! Ale nie, wszyscy jesteśmy dorośli; zaakceptujemy i to.

A więc, kiedy usunąć wzorzec? Kiedy system stał się złożony, a planowana elastyczność okazuje się niepotrzebna. Innymi słowy, gdy prostsze, nieuwzględniające wzorca rozwiązanie jest lepsze.



Jeżeli nie potrzebujesz tego dziś, odłóż to na jutro

Wzorce projektowe mają ogromne możliwości i łatwo znaleźć dla nich wiele zastosowań. Programiści kochają, oczywiście, dopracowane, gotowe na każdą zmianę architektury.

Oprzyj się pokusie. Jeżeli dziś istnieje praktyczna potrzeba przygotowania projektu do wprowadzenia zmian, zaakceptuj to i zastosuj odpowiedni wzorzec. Jeżeli jednak przyczyna zmiany pozostaje hipotetyczna, nie wprowadzaj nowego schematu. Zwiększy on złożoność rozwiązania, a może okazać się w ogóle niepotrzebny!



Mistrz i uczeń...

Mistrz: Młody człowieku, Twoje szkolenie, właściwie jego pierwszy etap, niemal się zakończył. Co planujesz?

Uczeń: Jadę do Disneylandu! A potem zacznę wytwarzać mnóstwo kodu z wzorcami!

Mistrz: Hej, powoli. Nigdy nie używaj swoich armat, jeżeli nie musisz.

Uczeń: Jak mam to rozumieć, Mistrzu? Teraz, gdy nauczyłem się tak wiele o wzorcach projektowych, nie powinienem używać ich we wszystkich moich projektach, aby zapewnić im jak największą skuteczność, elastyczność i przejrzystość?

Mistrz: Nie. Wzorce są pewnym narzędziem, które, jak każde inne narzędzie, powinno być stosowane tylko wtedy, gdy jest faktycznie potrzebne. Spędziłeś też wiele czasu, ucząc się zasad projektowania. Zawsze zaczynaj od tych zasad i buduj możliwie najprostszy kod, który wykonuje dane zadanie. Wzorców używaj wtedy, gdy pojawia się taka potrzeba.

Uczeń: Więc nie powinienem budować swojego projektu w oparciu o wzorce?

Mistrz: To nie powinno być Twoim celem, gdy rozpoczynasz pracę z projektem. Wzorce pojawiają się w tracie tego procesu w sposób naturalny.

Uczeń: Jeżeli wzorce są tak dobre, skąd ta ostrożność?

Mistrz: Wzorce mogą wprowadzić złożoność, a ta jest niepożądana wszędzie tam, gdzie nie ma uzasadnienia. Wzorce to narzędzie o dużych możliwościach, jeżeli jest odpowiednio stosowane. Jak już wiesz, wzorce to sprawdzone schematy rozwiązań, które pozwalają uniknąć typowych błędów. Zapewniają też dobrze zdefiniowane słownictwo, które pozwala opisać projekt innym.

Uczeń: To właściwie jak stwierdzić, że użycie wzorca jest wskazane?

Mistrz: Wprowadzamy wzorce, kiedy jesteśmy pewni, że jest to konieczne do rozwiązania pewnego problemu lub gdy jesteśmy prawie pewni, że będzie to wymagane, aby uwzględnić przyszłą zmianę w stawianych aplikacji wymaganiach.

Uczeń: Domyślam się, że chociaż wiele się już o wzorcach nauczyłem, moja nauka będzie trwała dalej.

Mistrz: Tak jest, adepcie. Uczenie się zarządzania złożonością i zmianami w oprogramowaniu to długa droga. Ale kiedy już znasz porządny zestaw wzorców, czas rozpocząć stosowanie ich we własnych projektach i zdobywanie wiedzy o innych.

Uczeń: Momencik, czy Ty nie znasz ich WSZYSTKICH?

Mistrz: Młody człowieku, poznałeś tylko podstawowe wzorce. Wkrótce stwierdzisz, że jest ich znacznie więcej, że istnieją też wzorce, które mają zastosowanie w wybranych dziedzinach, takich jak systemy przetwarzania współbieżnego czy systemy klasy enterprise. Teraz, kiedy znasz podstawy, masz wszelkie warunki ku temu, aby poznać je wszystkie!

Głowa pełna wzorców



UMYSŁ POCZĄTKUJĄCEGO

„Potrzebuję wzorca programu „Hello World".

Początkujący używa wzorców wszędzie. To jest dobre. Zdobywa dzięki temu dużo doświadczenia i praktyki w ich stosowaniu. Początkujący myśli: „Im więcej wzorców użyję, tym lepszy będzie mój projekt". Z czasem nauczy się, że wcale tak nie jest, że każdy projekt powinien być tak prosty, jak tylko jest to możliwe. Złożoność i wzorce powinny pojawiać się tylko tam, gdzie istnieje praktyczna potrzeba zapewnienia możliwości rozbudowy kodu.

W miarę postępów w nauce umysł zaawansowanego zaczyna rozpoznawać, gdzie wzorce są potrzebne, a gdzie nie. Wciąż utrzymuje się tendencja do wpychania wzorców „gdzie popadnie", ale zaczynamy widzieć, że mogą one być tak zaadaptowane, aby pasowały do tych sytuacji, gdzie podejście kanoniczne nie jest właściwe.



UMYSŁ ZAAWANSOWANEGO

„Może przydałby się tu Singleton".



UMYSŁ ZEN

„To naturalne, że tu użyjemy Deckratora".

Umysł Zen widzi wzorce tam, gdzie w naturalny sposób pasują. Umysłu Zen nie trapi dążenie do stosowania wzorców. Szuka on najprostszych rozwiązań, które najlepiej odpowiadają danemu problemowi. Umysł Zen myśli kategoriami zasad projektowania obiektowego i wyważonych decyzji. Gdy pojawia się naturalna potrzeba użycia wzorca, umysł Zen stosuje go z pełną świadomością, że może jeszcze być potrzebna pewna adaptacja. Umysł Zen widzi też powiązania z podobnymi wzorcami i rozumie subtelne różnice w ich przeznaczeniu.

Umysł Zen to też umysł początkującego — nie pozwala on, aby szeroka wiedza o wzorcach nadmiernie wpływała na kształt projektu.

UWAGA: Pochopne stosowanie wzorców prowadzi do kodu, który jest nadmiernie rozbudowany. Zawsze stawiaj na najprostsze rozwiązanie, które spełnia swoje zadanie, a wzorce wprowadzaj tam, gdzie występuje taka potrzeba.



Oczywiście, chcemy, żeby wszyscy używali wzorców!

Ale przede wszystkim chcemy, żeby byli to dobrzy projektanci.

Gdy problem aż prosi się o użycie wzorca, zyskujemy możliwość użycia rozwiązania, które zostało sprawdzone przez wielu programistów wcześniej. Korzystamy też z techniki, która jest dobrze udokumentowana i którą szybko rozpoznają inni programiści (znowu pojawia się kwestia jednolitego słownictwa).

Niemniej jednak, korzystanie z wzorców ma też swoją ciemną stronę. Wzorzec projektowy często wprowadza dodatkowe klasy i obiekty, a to oznacza wzrost złożoności projektu. Wzorce projektowe mogą też sprawić, że pojawią się całe nowe warstwy projektu, co oznacza nie tylko wzrost złożoności, ale i mniejszą wydajność.

Co więcej, stosowanie wzorców może być zwykłym marnotrawieniem czasu. W wielu przypadkach wystarczy trzymać się prostych zasad projektowania, aby znaleźć znacznie mniej skomplikowane rozwiązanie tego samego problemu. Jeśli tak się dzieje, nie walczmy z tym. Użyjmy prostszego rozwiązania.

Mimo to nie daj się zniechęcić. Wtedy, gdy wzorzec jest właściwym narzędziem, zalet jego zastosowania nie brakuje.

Nie zapominaj o potędze jednolitego słownictwa

Tyle uwagi już poświęciliśmy na opisywanie najróżniejszych niuansów technik programowania obiektowego, że łatwo byłoby zapomnieć o ludzkiej stronie wzorców projektowych — służą one nie tylko do wypełniania głowy początkującego gotowymi rozwiązaniami, ale mają przyczynić się do budowania jednolitego słownictwa, z którego korzystają wszyscy programiści. Nie należy lekceważyć tego elementu. Jest to jedna z *największych zalet* systemu wzorców projektowych.

Tylko pomyśl, już coś się zmieniło od ostatniego czasu, kiedy poruszaliśmy temat wspólnego słownictwa. Zacząłeś budować swoje własne! Oczywiście, nauczyłeś się też całego zbioru zasad projektowania obiektowego, które pozwalają z łatwością zrozumieć cel i zasady działania coraz to nowych wzorców.

Teraz, gdy masz za sobą podstawowy kurs wzorców projektowych, czas ruszyć w świat i nieść dobrą nowinę. Dlaczego? Bo gdy inni programiści poznają wzorce i zaczną korzystać z tego samego słownictwa, Wasze projekty będą lepsze, komunikacja będzie łatwa i, co najważniejsze, oszczędzisz wiele czasu, który możesz poświęcić na ciekawsze rzeczy.



Tak więc stworzyłam w końcu tę klasę rozgłoszeniową. Śledzi ona wszystkie obiekty, które w danej chwili nasłuchują, i za każdym razem, kiedy otrzymuje nowy zestaw danych, wysyła odpowiedni komunikat do wszystkich odbiorców. Fajną właściwością jest to, że obiekty nasłuchujące mogą w każdej chwili dołączyć do grona odbiorców komunikatów lub nawet mogą się same stamtąd usunąć. Sama klasa rozgłoszeniowa nie wie nic o obiektach nasłuchujących. Dopuszczalne jest rejestrowanie dowolnych obiektów, o ile tylko implementują one odpowiedni interfejs.

Niekompletne

Niejasne

Czasochłonne

Pięć podstawowych sposobów promowania Twojego słownictwa

1. **Na spotkaniach.** Gdy spotykasz się ze swoim zespołem, aby omawiać projekt oprogramowania, użyj wzorców projektowych w celu podtrzymania dyskusji o właściwych zagadnieniach projektowych. Omawianie rozwiązania z perspektywy wzorców projektowych i zasad programowania obiektowego powstrzyma uczestników od przedwczesnego zagłębienia się w detale implementacji i pozwoli uniknąć wielu nieporozumień.
2. **W rozmowach z innymi programistami.** Używaj wzorców w trakcie dyskusji z innymi. Pomoże im to dowiedzieć się więcej o wzorcach i przyczyni się do tworzenia poczucia wspólnoty. Najlepszym momentem podczas dzielenia się swoją wiedzą z innymi jest to wspaniałe uczucie, kiedy druga osoba w końcu „łapie", o co chodzi.
3. **W dokumentacji architektury.** Gdy piszemy dokumentację architektury aplikacji, użycie wzorców ograniczy jej rozmiary, zapewniając zarazem czytelnikowi przejrzysty obraz rozwiązania.
4. **W komentarzach i nazwach w kodzie.** Gdy piszesz kod aplikacji, jasno opisuj stosowane wzorce w komentarzach. Wybieraj też takie nazwy klas i metod, które nawiązują do stosowanych schematów. Gdy inni programiści będą musieli zapoznać się z Twoim kodem, podziękują Ci za jasną implementację i przejrzysty opis.
5. **Znajdź zainteresowanych.** Dziel się swoją wiedzą. Wielu programistów słyszało o wzorcach, ale mają słabą orientację w tym, czym są one naprawdę. Zaproponuj krótkie omówienie w porze lunchu lub zacznij udzielać się na lokalnym forum użytkowników.

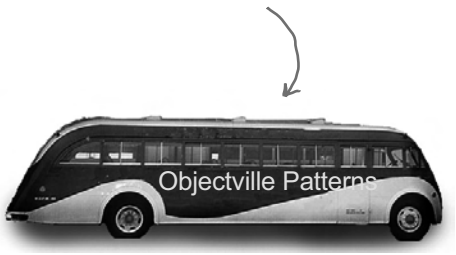


Gang Czterech w Obiektowie

Nie znajdziesz w Obiektowie gwiazd futbolu czy muzyki, ale jest w nim Gang Czterech. Jak zapewne zauważyłeś, trudno zrobić krok w świecie wzorców, nie natrafiając na słynny Gang of Four. Cóż to za tajemnicza grupa? Mówiąc prosto, Gang of Four (Gang Czterech) to Erich Gamma, Richard Helm, Ralph Johnson i John Vlissides — autorzy pierwszego katalogu wzorców projektowych, odpowiedzialni za stworzenie całkiem nowego trendu w dziedzinie programowania!

Skąd nazwa? Nikt nie wie tego na pewno. To etykietka, która przylgnęła do nich w pewnym momencie. Teraz pomyśl: jeżeli już w Obiektowie natykasz się na gang, czy mógłby on być bardziej sympatyczny? Co więcej, cały gang zgodził się na wizytę u nas…

Grupa GoF rozpoczęła popularyzowanie wzorców projektowych. Trudno jednak nie wspomnieć o innych zasłużonych. Wymienimy tylko kilku z nich: Ward Cunningham, Kent Beck, Jim Coplien, Grady Booch, Bruce Anderson, Richard Gabriel, Doug Lea, Peter Coad i Doug Schmidt.





Podróż dopiero się zaczyna...

Teraz, gdy znasz już wzorce projektowe i jesteś gotowy do dalszej eksploracji, czas przedstawić trzy podręczniki, które muszą znaleźć się na Twojej półce…



Podstawowy podręcznik wzorców projektowych

To książka, która zapoczątkowała całe zamieszanie wokół wzorców projektowych. Wydana została w 1995 roku. Znajdziesz w niej wszystkie podstawowe wzorce. Na niej wzorowany jest zbiór wzorców, które przedstawiliśmy w niniejszej publikacji.

Książka ta nie jest ostatnim słowem w dziedzinie wzorców. Wręcz przeciwnie, nowa dziedzina rozrosła się od czasu jej wydania znacząco. Jest to jednak pierwszy opis tego rodzaju, a zgromadzono w nim najważniejsze schematy.

Będzie doskonałą kontynuacją po „zabawie” z *Rusz głową!*.

Autorzy książki pod tytułem „Wzorce projektowe. Elementy oprogramowania obiektowego wielokrotnego użytku" są najlepiej znani jako Gang of Four, czyli Gang Czterech.

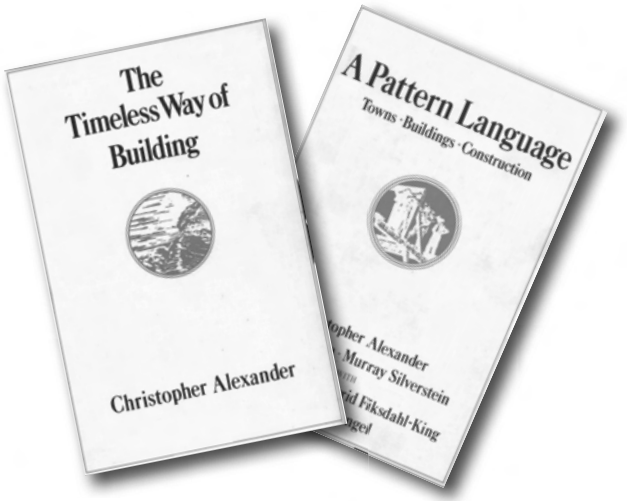
Christopher Alexander wymyślił pojęcie wzorca, co było inspiracją do wprowadzenia podobnego systemu w programowaniu.

Podstawowe podręczniki wzorców

Czwórka wielkich projektantów oprogramowania nie stworzyła pojęcia wzorca. Odpowiada za to Christopher Alexander, profesor architektury na Uniwersytecie w Berkeley. Tak właśnie! *Architekt*, a nie informatyk. Alexander zaczął tworzyć wzorce dla architektury — wzorce budowy domów i miast.

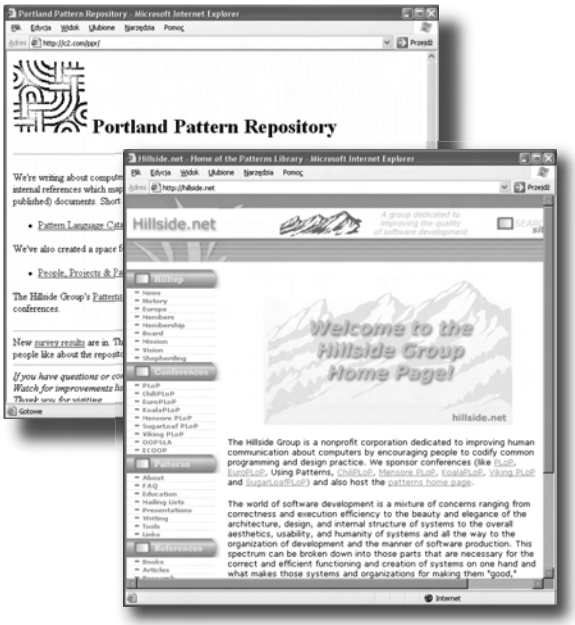
Kiedy będziesz w nastroju na wciągającą, bardziej ogólną w swojej naturze lekturę, sięgnij po *The Timeless Way of Building* i *A Pattern Language*. Poznasz prawdziwy początek systemu wzorców projektowych i odnajdziesz bezpośrednie analogie między tworzeniem architektury mieszkalnej a elastycznym i łatwym w rozbudowie oprogramowaniem.

Zaparz sobie kawę „Star Café”, usiądź w fotelu i baw się dobrze…



Inne źródła informacji o wzorcach

Wkrótce stwierdzisz, że na świecie istnieje już żywa i przyjazna społeczność użytkowników i autorów wzorców. Wszyscy będą szczęśliwi, gdy się do nich przyłączysz. Oto kilka źródeł, które pomogą Ci zacząć…



Witryny WWW

Portland Patterns Repository prowadzone przez Warda Cunninghama to Wiki poświęcona wszystkiemu, co jest związane z wzorcami. Każdy może wziąć udział w jej tworzeniu. Znajdziesz tu dyskusje na każdy możliwy temat, związany oczywiście z wzorcami i systemami obiektowymi.

`http://c2.com/ppr`

Hillside Group propaguje dobre praktyki programowania i projektowania zespołowego. Jest też obszernym źródłem informacji o wzorcach. Znajdziemy tu informacje o wielu innych publikacjach, listach mailingowych i narzędziach.

`http://hillside.net/`



Konferencje i warsztaty

Jeżeli chcesz poznać członków społeczności skupionej wokół wzorców osobiście, wybierz się na jedno z wielu seminariów i warsztatów. W zasobach witryny Hillside znajdziesz ich pełną listę. Najciekawszym wydarzeniem jest OOPSLA, konferencja ACM poświęcona systemom, językom i aplikacjom obiektowym (ang. *Object-Oriented Systems, Languages and Applications*).

ZOO pełne wzorców



Jak już wiemy, historia wzorców sięga poza dziedzinę programowania. Rozpoczęła się od projektowania budynków i miast. Tę samą koncepcję można dopasować też do innych dziedzin. Pospacerujmy po ogrodzie zoologicznym, w którym zgromadziliśmy najciekawsze okazy…



Wzorce w architekturze są używane do tworzenia przyjaznej i praktycznej architektury budynków i miast. Od tego zaczęła się kariera wzorców.

Środowisko naturalne: spotykane w budynkach, w których lubisz przebywać i mieszkać.

Środowisko naturalne: spotykane w okolicach architektur trójwarstwowych, systemów klient-serwer i sieci WWW.

Wzorce aplikacji to wzorce wspomagające tworzenie architektury systemowej. W tej kategorii znajdziemy wiele schematów architektury wielowarstwowej.



Notatka: MVC traktuje się często jako wzorzec aplikacji.



Wzorce specyficzne dla różnych dziedzin to wzorce, które rozwiązują problemy spotykane w węższych dziedzinach projektowania, takich jak systemy przetwarzania współbieżnego i w czasie rzeczywistym.

Pomóż określić środowisko naturalne

J2EE

Wzorce procesów biznesowych opisują interakcje między firmami, klientami i danymi. Mogą być stosowane do rozwiązywania problemów takich, jak efektywne podejmowanie decyzji i właściwy sposób informowania o nich.



Spotykane w salach konferencyjnych zarządów i na spotkaniach zespołów zadaniowych.

Pomóż znaleźć środowisko naturalne

Zespół programistów

Biuro obsługi klienta

Wzorce organizacyjne opisują struktury i praktyki stosowane w organizacjach. Jak dotąd, najwięcej uwagi poświęcono organizacjom, które produkują i zapewniają obsługę techniczną oprogramowania.





Wzorce projektowe interfejsu użytkownika koncentrują się na problemach projektowania oprogramowania interakcyjnego.

Środowisko naturalne: widziane w pobliżu projektantów gier wideo, graficznych interfejsów użytkownika, a także ich producentów.

Notatki: uzupełnij nasze zbiory własnymi obserwacjami dziedzin, w których stosuje się różne wzorce rozwiązań:

Walka ze złem przy użyciu antywzorców



Wszechświat nie byłby zrównoważony, gdyby wzorcom nie towarzyszyły antywzorce, prawda?

Jeżeli wzorzec zapewnia ogólne rozwiązanie powtarzającego się problemu w pewnym kontekście, co daje antywzorzec?

> **Antywzorzec** pokazuje, jak dojść od problemu do ZŁEGO rozwiązania.

Zapewne zadajesz sobie teraz pytanie: „Po co ktokolwiek miałby tracić czas, opisując złe rozwiązania?”.

Spójrz na zagadnienie z tej strony: jeżeli widzimy powtarzające się, złe rozwiązanie typowego problemu, stworzenie jego dokumentacji może uchronić innych programistów przed popełnieniem tego samego błędu. Ostatecznie, unikanie złych rozwiązań jest równie wartościowe, jak poszukiwanie dobrych!

Rozważmy teraz elementy antywzorca:

Antywzorzec informuje o tym, dlaczego złe rozwiązanie wydaje się korzystne. Spójrzmy prawdzie w oczy, nikt nie wybiera złego rozwiązania, jeżeli nie widzi w nim czegoś, co na pierwszy rzut oka wygląda bardzo atrakcyjnie. Jednym z najważniejszych zadań antywzorca jest ostrzeżenie programisty o takim zagrożeniu.

Antywzorzec wyjaśnia, dlaczego rozwiązanie na dłuższą metę okazuje się niewłaściwe. Aby zrozumieć, po co masz mieć do czynienia z antywzorcem, musisz wiedzieć, na czym polega jego niekorzystny wpływ. Antywzorzec wskazuje, w którym momencie rozwiązanie okaże się porażką. Wymienia on też inne wzorce, których zastosowanie może doprowadzić do dobrego rozwiązania.

Antywzorzec wskazuje na inne wzorce, których zastosowanie może doprowadzić do dobrego rozwiązania. Aby być prawdziwie użytecznym, antywzorzec musi wskazywać właściwy kierunek. Powinno to być odniesienie do innych dostępnych możliwości, które mogą doprowadzić do dobrych rozwiązań.

Obejrzyjmy przykład antywzorca.

Antywzorzec zawsze wygląda jak dobre rozwiązanie. Okazuje się ono niewłaściwe dopiero, gdy zostanie użyte.

Dokumentowanie antywzorców ułatwia innym rozpoznawanie złych rozwiązań jeszcze przed ich implementacją.

Podobnie jak wzorców, jest wiele typów antywzorców, w tym wzorce programowania, projektowania obiektowego, organizacyjne oraz wzorce specyficzne dla różnych węższych dziedzin.

Oto przykład antywzorca osadzonego w dziedzinie programowania.



Antywzorzec

Podobnie jak typowy wzorzec projektowy, antywzorzec posiada nazwę, która umożliwia stosowanie jednolitego, zwięzłego słownictwa.

Nazwa: Złoty młotek.

Problem i kontekst — podobnie jak w opisach wzorców.

Problem: Musisz wybrać technologie do realizacji projektu i uważasz, że jedna z nich powinna dominować w całej architekturze.

Kontekst: Musisz stworzyć nowy system lub element oprogramowania, który nie pasuje zbyt dobrze do technologii znanych zespołowi.

Mówi o tym, dlaczego rozwiązanie wydaje się atrakcyjne.

Siły:

- Zespół programistów jest silnie związany z technologią, którą stosuje od lat.
- Zespół programistów słabo zna inne technologie.
- Nieznane technologie są postrzegane jako ryzykowne.
- Planowanie i szacowanie ilości pracy jest łatwe, gdy trzymamy się znanej technologii.

Złe, pozornie atrakcyjne rozwiązanie.

Rzekome rozwiązanie: Użycie znanej technologii, nawet jeżeli nie jest idealna. Trzymamy się jej w każdym przypadku, nawet jeżeli jest w miarę oczywiste, że należałoby użyć innej.

Jak uzyskać dobre rozwiązanie.

Proponowane rozwiązanie: Poszerzenie wiedzy programistów przez edukację, szkolenia i grupy nauki własnej, aby oswoić ich z możliwościami innych technologii.

Przykład zastosowania antywzorca.

Przykłady:

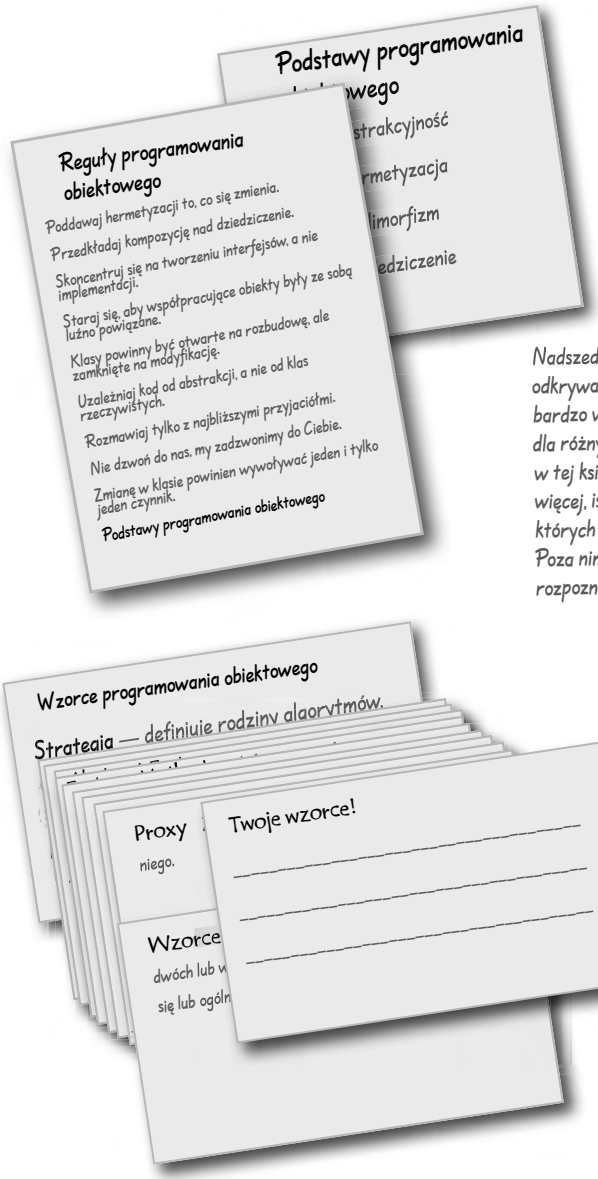
Firmy związane z WWW uporczywie trzymają się wewnętrznych, samodzielnie napisanych systemów buforowania, podczas gdy dostępne są dopracowane rozwiązania alternatywne, należące do kategorii open source.

Adaptacja na podstawie Wiki Portland Pattern Repository (http://c2.com/). Znajdziesz tam wiele antywzorców i dyskusji na ich temat.



Twoja skrzynka narzędziowa

Nadchodzi czas, gdy uczeń przerasta mistrza... Wówczas musi ruszać w świat i poznawać dalsze wzorce samodzielnie...



Nadszedł czas, abyś samodzielnie zaczął odkrywać wzorce projektowe. Jest bardzo wiele wzorców specyficznych dla różnych dziedzin, o których w ogóle w tej książce nie wspomnieliśmy. Co więcej, istnieją wzorce fundamentalne, których również tu nie omawialiśmy. Poza nimi są jeszcze te, które sam rozpoznasz i opiszesz.

Zapoznaj się z dodatkiem do książki. Krótko omawiamy tam inne wzorce podstawowe, którymi powinieneś się zainteresować.

CELNE SPOSTRZEŻENIA

- Pozwól, aby wzorce projektowe same wyłaniały się w Twoich projektach. Nie wprowadzaj ich tylko po to, aby projekt wyglądał poważniej.
- Wzorce projektowe można zmieniać; adaptuj je i modyfikuj odpowiednio do potrzeb.
- Zawsze używaj najprostszego rozwiązania, które odpowiada stawianym wymaganiom, nawet jeżeli nie wymaga ono użycia wzorców.
- Przeglądaj różne katalogi wzorców, aby zapoznać się z różnymi propozycjami i zależnościami między nimi.
- Klasyfikacje (kategorie) wzorców służą do ich grupowania. Używaj ich wtedy, gdy są pomocne.
- Aby samodzielnie napisać opis wzorca, musisz się zaangażować. Wymaga to czasu i cierpliwości, niezbędnych do wprowadzania kolejnych ulepszeń.
- Pamiętaj, że większość wzorców, które spotkamy w praktyce, to adaptacje tych, które zostały opracowane wcześniej, a nie zupełnie nowe techniki.
- Pracuj nad wspólnym słownictwem swoje zespołu. To jedna z największych korzyści ze stosowania wzorców.
- Podobnie jak każda inna społeczność, społeczność związana z wzorcami operuje specyficznym językiem. Nie pozwól, aby spotkanie z nim podziałało na Ciebie zniechęcająco. Po przeczytaniu tej książki jesteś już wystarczająco zaawansowany, aby sobie poradzić.

Opuszczamy Obiektowo...



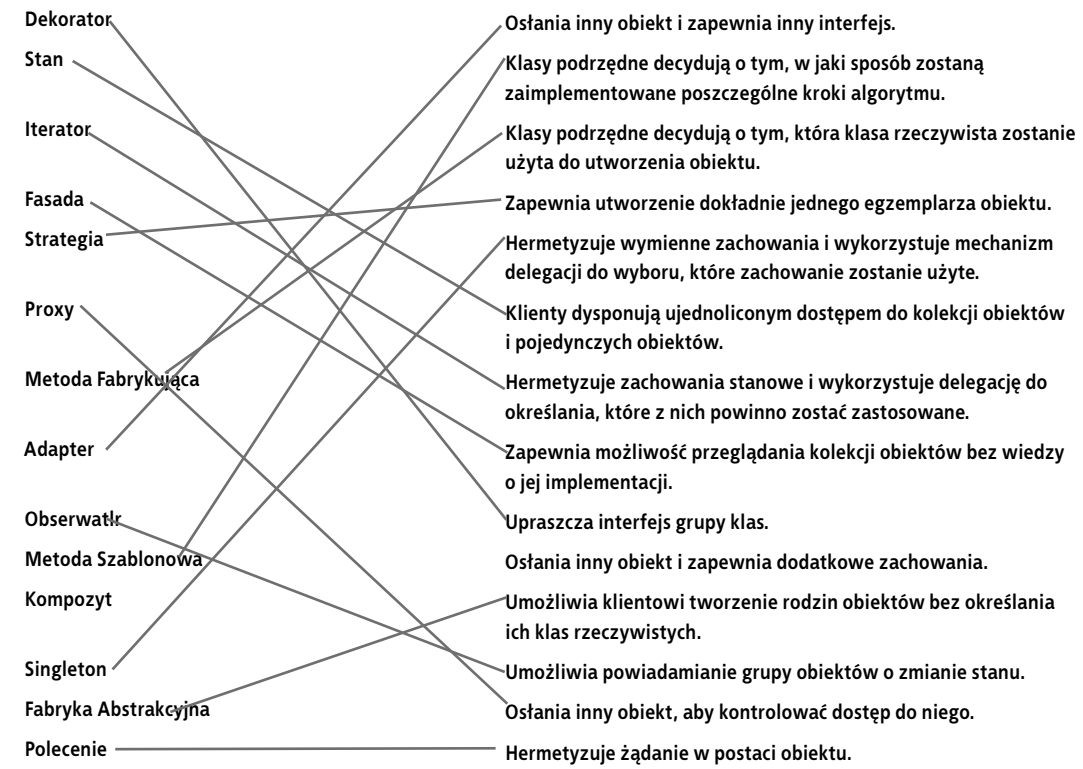
Kolego, Twoja wizyta była dla nas wielką przyjemnością

Będziemy tęsknić. Ale nie martw się — zanim się zorientujesz, wydamy kolejną książkę z serii *Rusz głową!* i spotkamy się ponownie. Jaka będzie następna książka? Dobre pytanie! Może pomożesz nam zdecydować? Wyślij nam e-mail na adres: *booksuggestions@wickedlysmart.com*

Rozwiązania ćwiczeń

Kto co robi?

Dopasuj wzorce do opisów:



Dodatek — inne wzorce



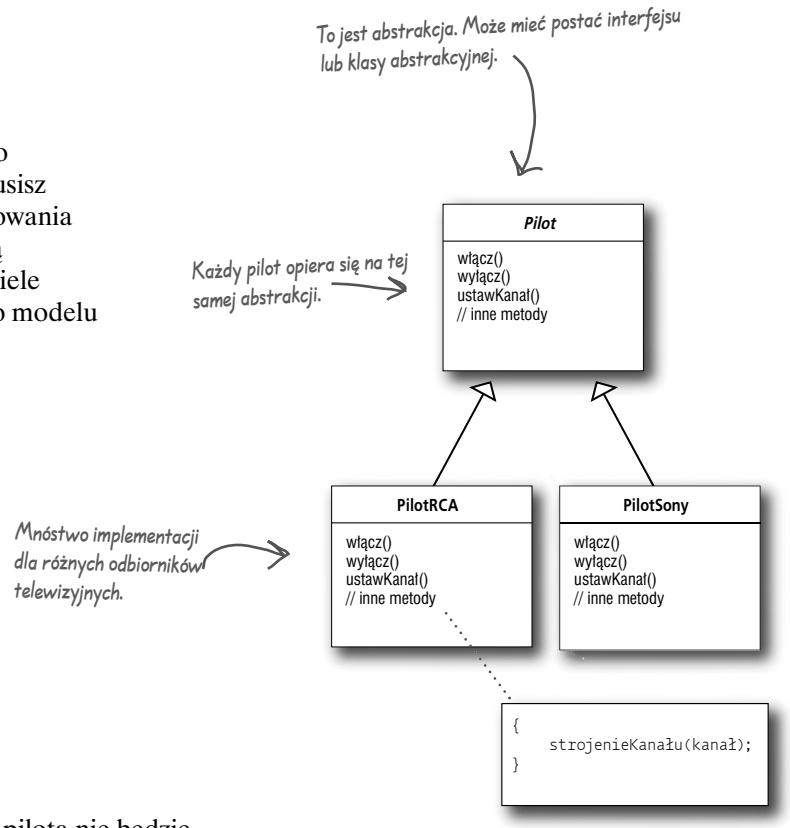
Nie wszyscy mogą być sławni. Przez ostatnie dziesięć lat wiele się w świecie wzorców zmieniło. Od czasu pierwszego wydania książki *Design Patterns: Elements of Reusable Object-Oriented Software* (*Wzorce projektowe*) programiści wykorzystali opisane w niej schematy tysiące razy. Wzorce, które zebraliśmy w tym dodatku, to dopracowane, kompletne, „oficjalne" wzorce grupy GoF. Różnią się od wcześniej opisanych tylko tym, że nie spotkamy ich tak często jak tych, którym poświęciliśmy całe rozdziały. Nie umniejsza to ich zalet i nie powinno zniechęcać do ich stosowania tam, gdzie wymaga tego sytuacja. Celem niniejszego dodatku jest zapewnienie Ci szerszej orientacji w najłatwiej dostępnych zasobach zgromadzonej przez lata wiedzy.

Most (ang. Bridge)

Wzorzec Bridge zapewnia możliwość modyfikowania nie tylko implementacji, ale i stosowanych abstrakcji.

Scenariusz

Wyobraź sobie, że Twoim zadaniem jest zrewolucjonizowanie zwyczajów miłośników telewizji. Piszesz kod dla nowego, ergonomicznego i przyjaznego użytkownikowi pilota. Wiesz już, że musisz użyć dopracowanych technik programowania obiektowego, bo choć piloty operują tą samą *abstrakcją*, będzie potrzebnych wiele *implementacji* — po jednej dla każdego modelu odbiornika telewizyjnego.



Dylemat projektanta

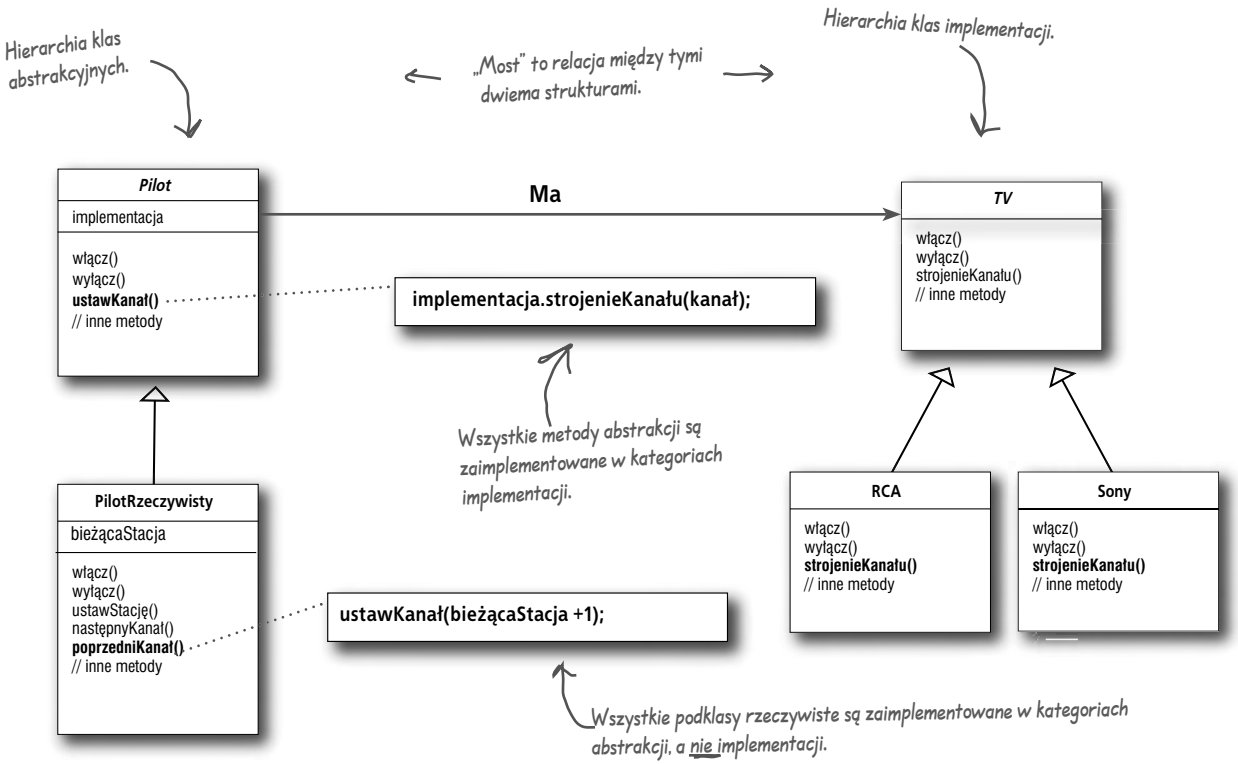
Jesteś pewien, że interfejs użytkownika pilota nie będzie od razu doskonały. Co więcej, oczekujesz tego, że nowy produkt będzie wielokrotnie usprawniany, w miarę napływu danych ze środowiska testowego.

Twoim problemem jest więc to, że zmieniać się będą zarówno odbiorniki telewizyjne, jak i piloty. Wprowadziłeś już *abstrakcję* interfejsu użytkownika, co pozwala na różnicowanie *implementacji* odpowiednio do różnych modeli telewizorów. Będziesz jednak potrzebował też *zmian abstrakcji*, ponieważ poprawki w pilocie będą wymuszały z czasem nowe modyfikacje.

Jak więc stworzyć projekt obiektowy, który pozwoli różnicować implementację i abstrakcję?

Dlaczego warto użyć wzorca Most?

Wzorzec Most umożliwia różnicowanie implementacji i abstrakcji poprzez umieszczenie obu elementów w osobnych hierarchiach klas.



Masz teraz dwie hierarchie, jedną dla pilotów i drugą dla tych implementacji telewizorów, które są specyficzne dla modelu. Most umożliwia modyfikowanie każdej ze stron niezależnie.

Zalety wzorca Most

- Separuje implementację, usuwając jej trwałe powiązanie z interfejsem.
- Abstrakcja i implementacja mogą być rozszerzane niezależnie.
- Zmiany w klasach rzeczywistych abstrakcji nie wpływają na klienta.

Zastosowania i wady wzorca Most

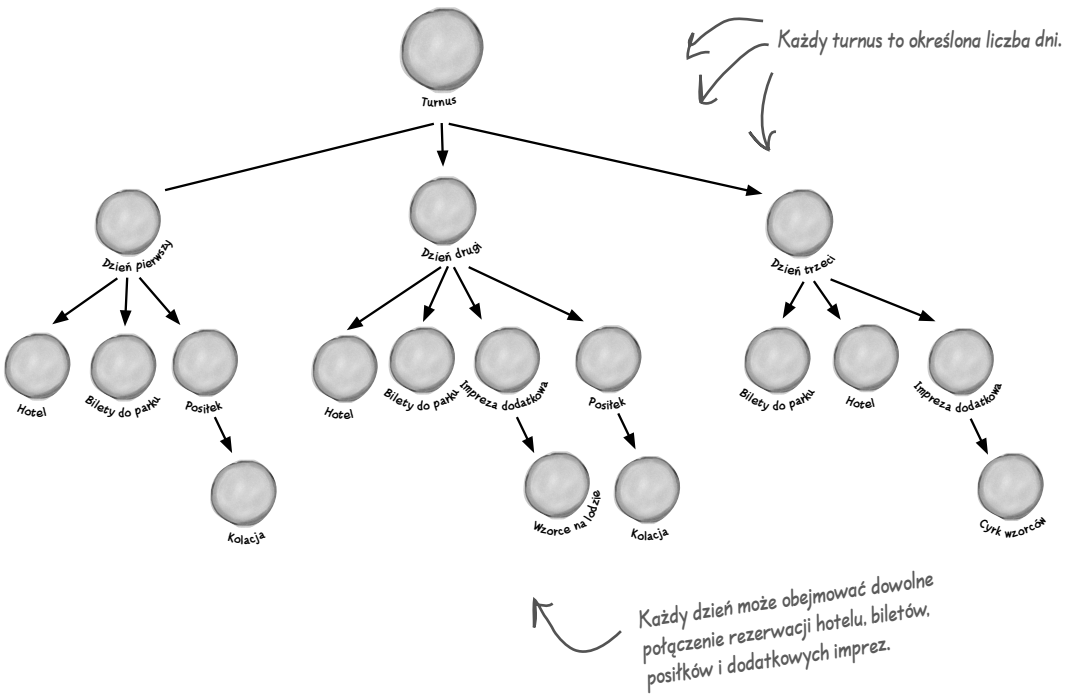
- Przydatny w systemach graficznych i okienkowych, które muszą pracować na różnych platformach.
- Przydatny w każdej sytuacji, gdy oczekujemy innych zmian w interfejsie i innych w implementacji.
- Zwiększa złożoność.

Builder (budowniczy)

Wzorca Builder używamy do hermetyzowania tworzenia produktu i w celu umożliwienia jego wieloetapowego inicjowania.

Scenariusz

Dostałeś zlecenie na stworzenie aplikacji wspomagającej sprzedaż turnusów w Krainie Wzorców, nowym parku tematycznym niedaleko Obiektowa. Klienci mogą wybrać hotel i różne typy biletów, rezerwować miejsca w restauracjach, a także zamówić bilety na dodatkowe imprezy. Podstawą aplikacji będą struktury takie, jak przedstawiona na rysunku:



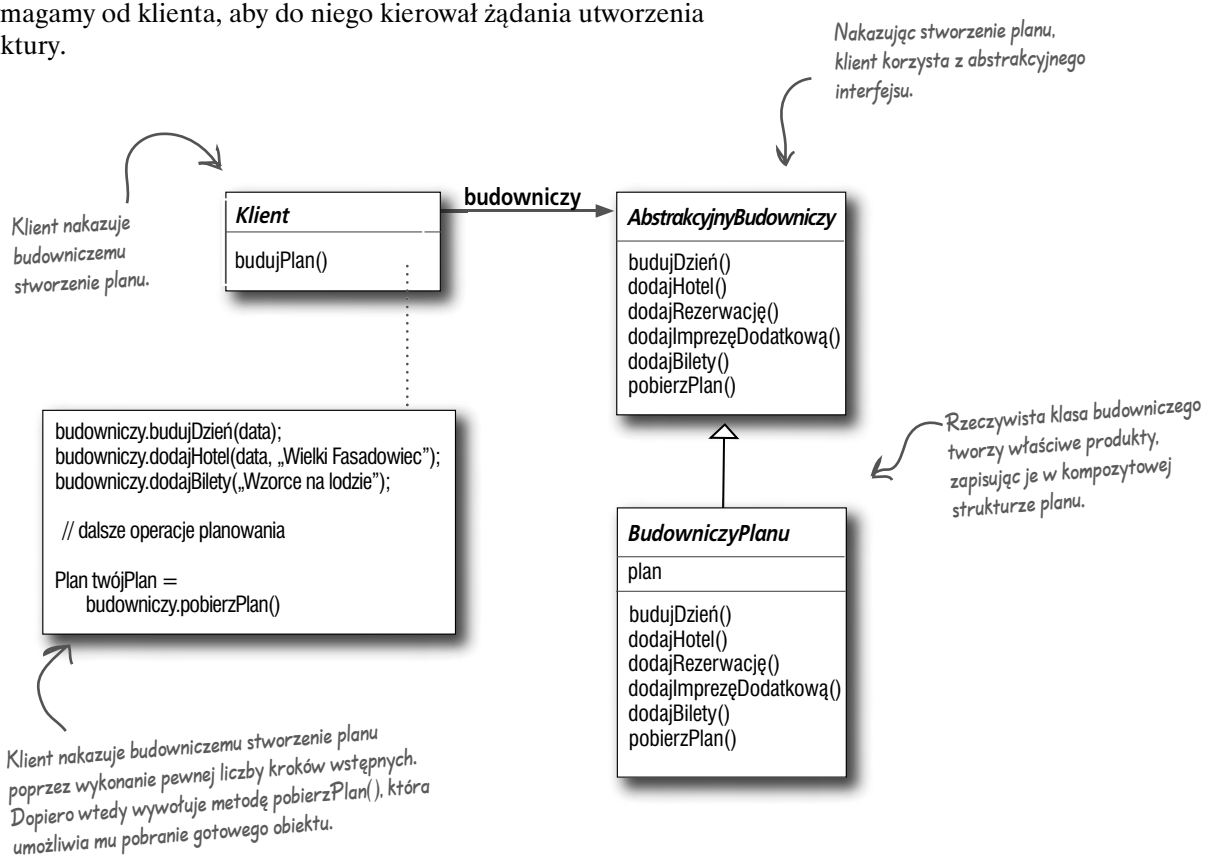
Potrzebujesz elastycznego projektu

Program każdego gościa parku może obejmować inną liczbę dni i inne opcje. Przykładowo, mieszkaniec Obiektowa może nie potrzebować hotelu, a więc zamawia tylko obiad i dodatkowe imprezy. Inny klient może przyjechać z daleka, wówczas będzie potrzebował hotelu, obiadów i biletów wstępu.

Potrzebna jest więc elastyczna struktura danych, która może reprezentować plan pojedynczego klienta w każdej potencjalnej odmianie. Pojawi się również sekwencja czynności prowadzących do utworzenia takiego planu (czynności te mogą być bardzo złożone). Jak zapewnić możliwość tworzenia tak złożonej struktury, nie łącząc tej operacji ze skomplikowanymi czynnościami, które prowadzą do końcowego efektu?

Dlaczego warto użyć wzorca Builder?

Pamiętasz wzorzec Iterator? Hermetyzowaliśmy iteracje w osobnym obiekcie i ukrywaliśmy wewnętrzną reprezentację kolekcji przed klientem. Podobny pomysł pojawia się w tym przypadku: hermetyzujemy tworzenie planu wycieczki w obiekcie (będzie to nasz „budowniczy") i wymagamy od klienta, aby do niego kierował żądania utworzenia struktury.



Zalety wzorca Builder

- Hermetyzuje operacje niezbędne do stworzenia złożonego obiektu.
- Umożliwia tworzenie obiektów w procedurze wielokrokowej i nie narzuca tej procedury (w przeciwieństwie do jednokrokowych wzorców z grupy Fabryka).
- Ukrywa wewnętrzną reprezentację produktu przed klientem.
- Implementacje produktów mogą być wymieniane, ponieważ klient korzysta wyłącznie z abstrakcyjnego interfejsu.

Zastosowania i wady wzorca Builder

- Często stosowany do budowania struktur kompozytowych.
- Tworzenie obiektów wymaga od klienta większej wiedzy z zakresu dziedziny problemu niż w przypadku stosowania wzorca Fabryka.

Łańcuch Odpowiedzialności (ang. *chain of responsibility*)

Wzorzec Łańcuch Odpowiedzialności stosujemy wtedy, gdy kolejne żądania mają być obsługiwane przez różne obiekty.

Scenariusz

Od czasu wprowadzenia na rynek nowego automatu sprzedającego z oprogramowaniem w języku Java firma Automaty Sprzedające SA odbiera coraz więcej e-maili. Skłoniło to jej szefów do przeprowadzenia analizy, w której wyróżniono cztery grupy listów: (1) pochwalne — wysyłane przez fanów promocji „1 z 10”, (2) skargi — składane przez rodziców, których dzieci uzależniły się od hazardu, i (3) wskazujące zapotrzebowanie — pochodzące od tych, którzy chcą, żeby w ich firmie postawić automat. Czwarta kategoria to spam.

Listy pochwalne mają trafiać prosto do prezesa, skargi do działu prawnego, a zapotrzebowania do działu rozwoju. Spam ma zostać usunięty.

Twoje zadanie

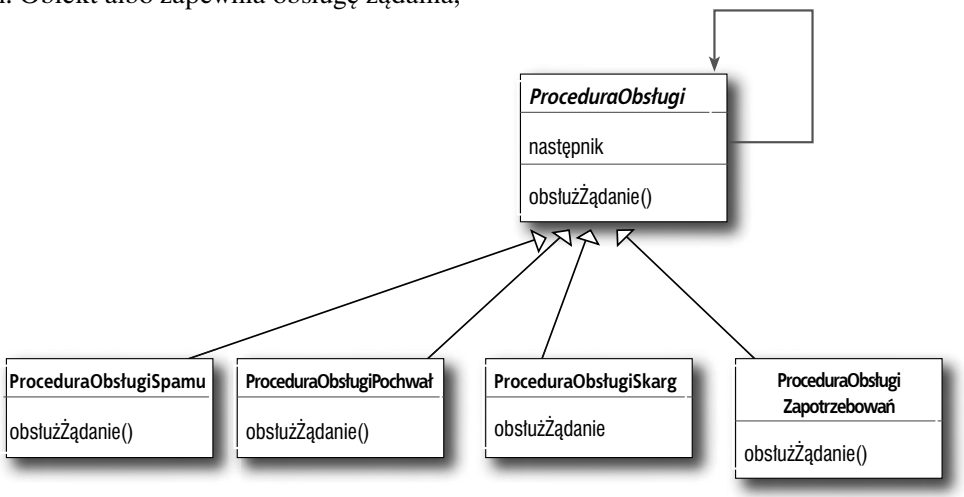
Firma dysponuje już inteligentnymi wykrywaczami spamu, listów pochwalnych, skarg i zapotrzebowań. Twoim zadaniem jest zaprojektowanie rozwiązania, które pozwoli wykorzystać te detektory do obsługi odbieranej poczty.



Jak używać wzorca Łańcuch Odpowiedzialności

Wzorzec Łańcuch Odpowiedzialności prowadzi do utworzenia łańcucha obiektów, które analizują żądanie. Analiza przeprowadzana jest kolejno przez każdy z nich. Obiekt albo zapewnia obsługę żądania, albo przekazuje je dalej.

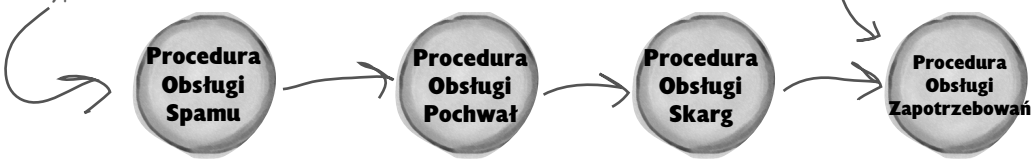
Każdy obiekt w łańcuchu działa jak procedura obsługi żądania i ma pewien następnik. Jeżeli dany obiekt potrafi obsłużyć żądanie, przetwarzanie zostaje zakończone. W innych przypadkach przekazuje żądanie następnikowi.



Po odebraniu list jest przekazywany do pierwszej procedury obsługi — tej, która radzi sobie ze spamem. Jeżeli nie pasuje ona do złożonego żądania, jego przetwarzanie jest przekazywane do procedury obsługi pochwał. I tak dalej…

Każdy list jest przekazywany do pierwszej procedury.

List nie zostanie odpowiednio potraktowany, jeżeli żaden z obiektów nie zapewni jego obsługi. Jednak ostatni element łańcucha może inicjować pewnego rodzaju przetwarzanie niezależnie od charakterystyki żądania.



Zalety wzorca Łańcuch Odpowiedzialności

- Separuje nadawcę żądania od jego odbiorców.
- Upraszcza obiekt, ponieważ nie wymaga utrzymywania informacji o strukturze łańcucha i operowania bezpośrednimi odwołaniami do jego elementów.
- Umożliwia dynamiczne dodawanie i usuwanie procedur obsługi poprzez zmianę elementów lub kolejności łańcucha.

Zastosowania i wady wzorca Łańcuch Odpowiedzialności...

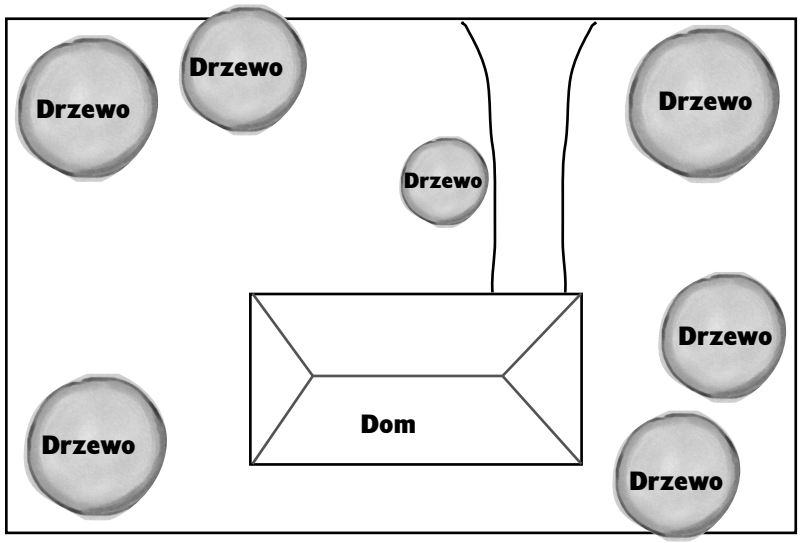
- Powszechnie używany w systemach okienkowych do obsługi zdarzeń takich, jak kliknięcia myszy i wciśnięcia klawiszy.
- Wykonanie żądania nie jest gwarantowane. Jeżeli żaden obiekt nie zapewni jego obsługi, żądanie opuści łańcuch, nie inicjując żadnych operacji (co może być tak wadą, jak i zaletą).
- Obserwowanie pracy łańcucha w czasie wykonywania aplikacji oraz jego debugowanie mogą być trudne.

Flyweight (waga piórkowa)

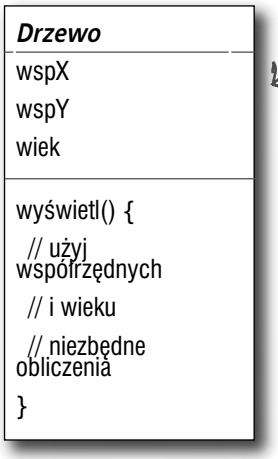
Wzorzec Flyweight stosujemy wtedy, gdy jeden egzemplarz klasy może zostać wykorzystany do stworzenia wielu „egzemplarzy wirtualnych”.

Scenariusz

W swojej nowej aplikacji wspomagającej pracę architekta musisz wprowadzić obiekty reprezentujące drzewa. Nie będą one rozbudowane: będą zawierały informację o położeniu i umiejętność wyświetlenia dopasowanego do wieku piktogramu. Problem stwarza to, że użytkownik aplikacji może wymagać bardzo wielu drzew. Może to wyglądać na przykład tak:



Każdy egzemplarz klasy Drzewo ma własne dane stanu.

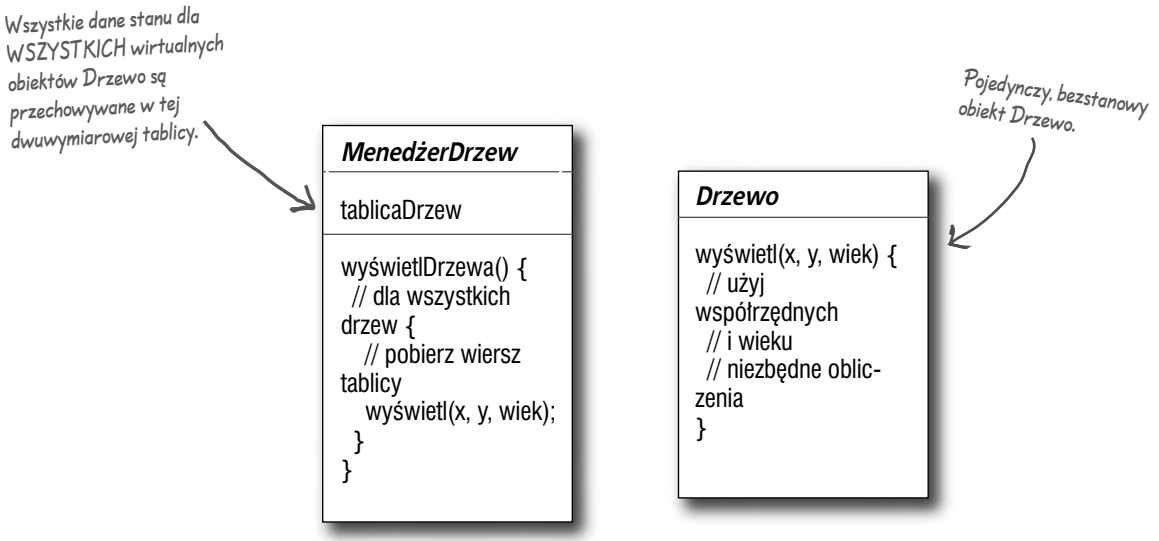


Problem dużego klienta

Właśnie złapałeś swojego najważniejszego klienta. Masz zapewniony stały dopływ gotówki przez wiele miesięcy. Kupi on 1000 egzemplarzy Twojej aplikacji. Chce wykorzystać ją do projektowania przestrzennego dużych osiedli. Po tygodniowych testach pojawiają się skargi, że gdy architekt planuje duży kompleks zieleni, aplikacja zaczyna pracować bardzo powoli…

Dlaczego warto użyć wzorca Flyweight?

Co by się stało, jeżeli mógłbyś wprowadzić taką zmianę w projekcie, aby w miejsce tysięcy obiektów Drzewo tworzyć tylko jeden taki obiekt? Niezbędnym uzupełnieniem będzie obiekt kliencki, który przechowuje stan WSZYSTKICH drzew. To właśnie wzorzec Flyweight!



Zalety wzorca Flyweight

- Ogranicza liczbę egzemplarzy obiektów w czasie wykonywania programu, oszczędzając w ten sposób pamięć aplikacji.
- Skupia składowanie danych stanu „wirtualnych" obiektów w jednej lokalizacji.

Zastosowania i wady wzorca Flyweight

- Wzorzec Flyweight stosuje się tam, gdzie klasa ma wiele egzemplarzy, a wszystkie mogą być sterowane w identyczny sposób.
- Wadą wzorca Flyweight jest to, że po jego zaimplementowaniu pojedyncze, logiczne egzemplarze klasy nie będą mogły dysponować zachowaniami niezależnymi od pozostałych egzemplarzy.

Interpreter (interpreter)

Wzorca Interpreter używamy do budowania interpretera pewnego języka.



Wzorzec Interpreter wymaga pewnej znajomości gramatyk formalnych. Jeżeli nigdy nie zajmowałeś się nimi, po prostu przeczytaj, co napisaliśmy o wzorcu. Uzyskasz przynajmniej ogólną orientację.

Scenariusz

Czy pamiętasz symulator stawu z kaczkami? Mógłby on stać się świetnym narzędziem edukacyjnym, ułatwiającym dzieciom naukę podstaw programowania. Pomysł polega na tym, że w trakcie korzystania z symulatora każde dziecko może sterować jedną kaczką przy użyciu prostego języka. Oto jego przykład:

```
prawo;
dopóki (dzień) lataj;
kwacz;
```

Kaczka skręca w prawo.

Lata cały dzień…

… a potem kwacze.

Teraz, mając w pamięci zasady budowania gramatyk formalnych z jednego z kursów wprowadzających do programowania, rozpisujesz gramatykę tego języka:

Program to wyrażenie składające się z sekwencji poleceń i powtórzeń, czyli instrukcji takich, jak „dopóki" („while").

```
wyrażenie ::= <polecenie> | <sekwencja> | <powtórzenie>
sekwencja ::= <wyrażenie> ';' <wyrażenie>
polecenie ::= prawo | kwacz | lataj
powtórzenie ::= dopóki '(' <zmienna> ')' <wyrażenie>
zmienna ::= [A-z,a-z]+
```

Sekwencja to zbiór wyrażeń rozdzielanych średnikami.

Mamy trzy polecenia: prawo, kwacz i lataj.

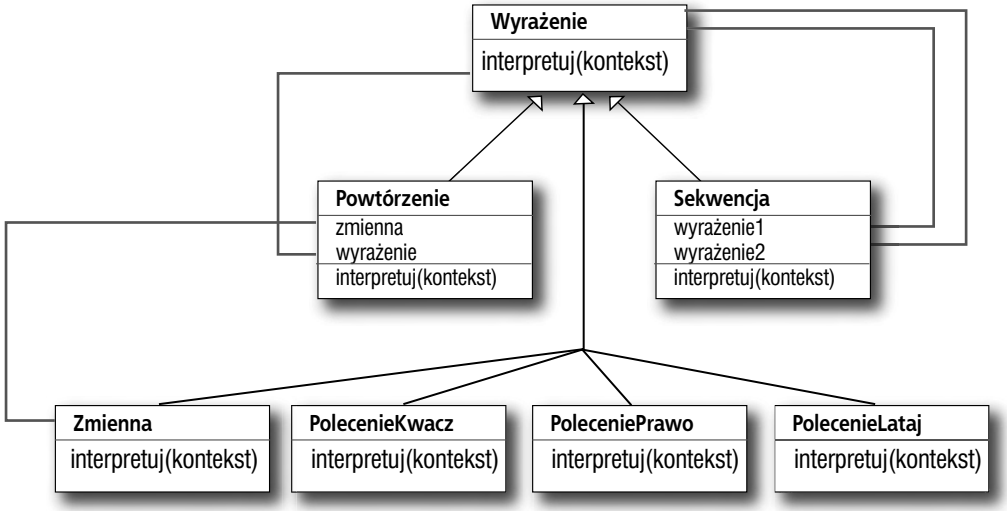
Instrukcja „dopóki" obejmuje zmienną, która określa warunek, oraz wyrażenie.

I co teraz?

Masz gramatykę. Potrzebujesz teraz systemu do reprezentowania i interpretowania formułowanych w niej zdań w taki sposób, aby uczniowie zobaczyli, jak wprowadzony przez nich program wpływa na symulowane kaczki.

Jak zbudować interpreter

Gdy pojawia się potrzeba zaimplementowania prostego języka, wzorzec Interpreter definiuje reprezentację gramatyki opartą na klasach i kod, który interpretuje wprowadzone zdania. Każda klasa reprezentuje jedną regułę języka. Oto język kaczek przekształcony na klasy. Zwróć uwagę na bezpośrednie mapowanie do gramatyki.



Aby zinterpretować kod języka, wywołujemy polecenie interpretuj dla każdego typu wyrażenia. Metoda ta wymaga podania kontekstu. Ten zawiera strumień wejściowy przetwarzanego programu, który jest analizowany i wykonywany.

Zalety wzorca Interpreter

- Reprezentowanie każdej reguły gramatycznej jako klasy znacznie ułatwia implementowanie języka.
- Ponieważ gramatykę reprezentują klasy, język można łatwo modyfikować i rozszerzać.
- Dodając nowe metody klas, możesz wprowadzać nowe zachowania wykraczające poza samą interpretację, jak ładne wydruki lub wyszukane sprawdzanie poprawności programu.

Zastosowania i wady wzorca Interpreter

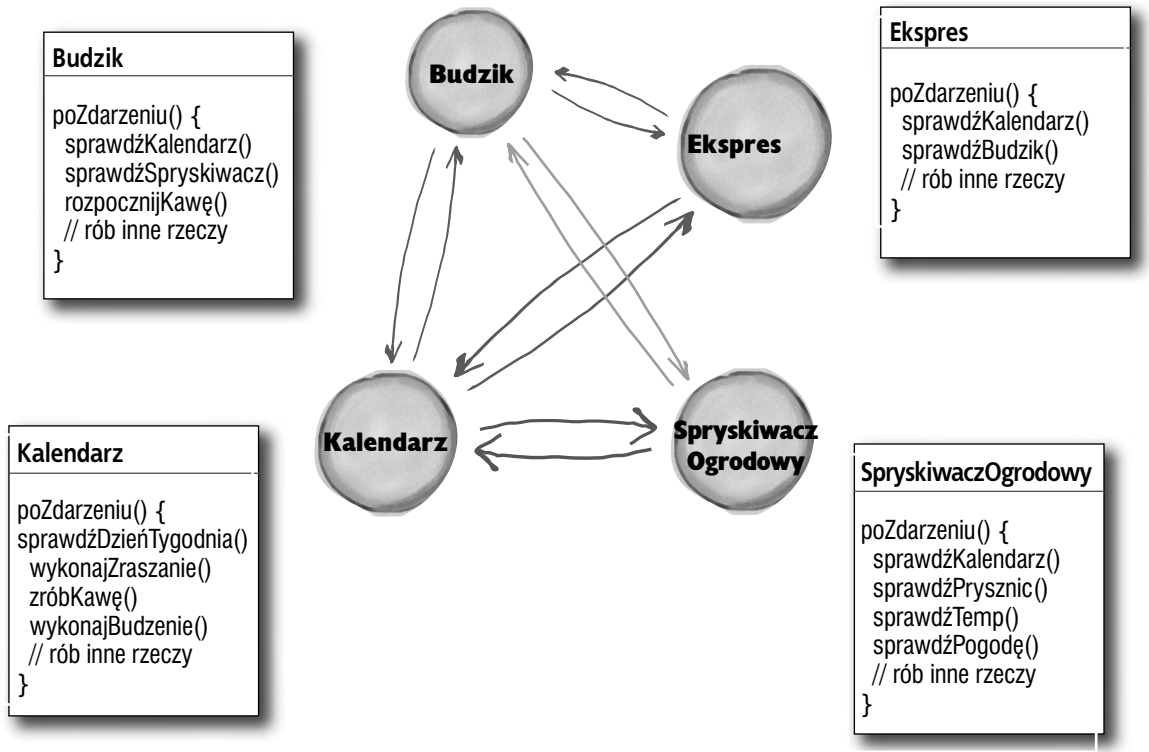
- Wzorca Interpreter używamy tam, gdzie pojawia się potrzeba zaimplementowania prostego języka.
- Wzorzec jest odpowiedni tam, gdzie gramatyka jest prosta i przejrzystość rozwiązania jest ważniejsza od wydajności.
- Używany do obsługi języków skryptowych i innych języków programowania.
- Wzorzec może stać się nieporęczny, gdy liczba reguł gramatycznych jest znaczna. W takich przypadkach generator kompilatorów będzie lepszym rozwiązaniem.

Mediator

Wzorca Mediator używamy do skupiania złożonych procedur komunikacji i sterowania w środowisku powiązanych obiektów.

Scenariusz

Robert to szczęśliwy użytkownik „domu inteligentnego” zbudowanego przez firmę DomPrzyszłości. Głównym językiem programowania jest tu Java. Wszystkie urządzenia domowe są zaprojektowane tak, aby życie mieszkańca było jak najwygodniejsze. Gdy odrywa on palec od przycisku budzika, budzik powiadamia ekspres do kawy, że czas zacząć parzenie. Mimo takich ułatwień Robert i inni klienci wciąż zgłaszają zapotrzebowanie na nowe funkcje: nie ma kawy w weekendy… wyłączcie spryskiwacz ogrodowy 15 minut przed prysznicem… ustawcie budzik na wcześniejszą godzinę w dniach wywozu śmieci…



Dylemat firmy DomPrzyszłości

Śledzenie tego, w których obiektach zaimplementowane zostały poszczególne reguły, zaczyna być skomplikowane. Podobnie jest z relacjami między obiektami.

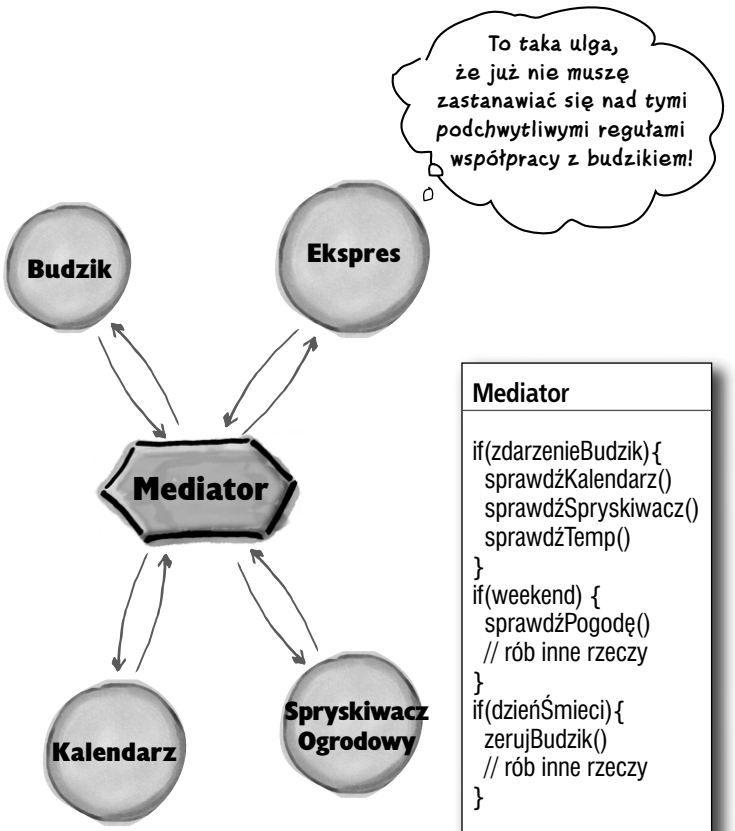
Mediator w akcji

Po wprowadzeniu do systemu wzorca Mediator wszystkie obiekty reprezentujące urządzenia mogą być znacznie uproszczone:

- Informują one mediatora o zmianach stanu.
- Odpowiadają na żądania mediatora.

Przed wprowadzeniem wzorca do projektu wszystkie obiektu urządzeń musiały wiedzieć o sobie nawzajem… obowiązywała zasada ścisłych powiązań. Mediator zapewnia im *całkowitą separację*.

Mediator zawiera całość logiki sterowania systemu. Gdy jedno z urządzeń wymaga nowej reguły albo pojawia się całkiem nowe urządzenie, jest jasne, że wszelka logika zostanie zaimplementowana w klasie mediatora.



Zalety wzorca Mediator

- Zwiększa szanse ponownego wykorzystania obiektów, z którymi współpracuje mediator, bo zapewnia ich separację od systemu.
- Upraszcza rozwijanie systemu, bo skupia w jednym miejscu logikę sterowania.
- Upraszcza i zmniejsza liczbę komunikatów przesyłanych pomiędzy obiektami w systemie.

Zastosowania i wady wzorca Mediator

- Wzorzec Mediator jest powszechnie stosowany do koordynowania powiązanych składników graficznego interfejsu użytkownika.
- Wadą wzorca jest to, że gdy projekt nie jest dopracowany, sam obiekt mediatora może stać się za bardzo skomplikowany.

Memento

Wzorca Memento używamy wtedy, gdy potrzebujemy możliwości przywrócenia obiektu do jednego z wcześniejszych stanów. Przykładem może być operacja użytkownika „Cofnij”.

Scenariusz

Twoja interakcyjna gra RPG osiągnęła wielki sukces i ma już swoje grono nałogowców walczących o to, aby dostać się na słynny „poziom 13”. Wraz z postępami użytkowników, którzy wspinają się na coraz trudniejsze poziomy gry, szanse ich zetknięcia ze smutnym komunikatem „gra skończona” rosną. Najwięksi fani, którzy spędzili całe dnie, żeby uzyskać swoją moc i potęgę, są oczywiście bardzo niezadowoleni, gdy wszystko to znika i muszą zaczynać od nowa. Żądania wprowadzenia polecenia „zapisz postępy” są trudne do zignorowania. Gracze chcą mieć możliwość zapisywania danych swojej postaci, a przynajmniej odtworzenia większości pracy w sytuacji, gdy postać ta zostanie niesprawiedliwie zgładzona. Funkcja „zapisz postępy” powinna być tak zaprojektowana, aby umożliwić powrót zmartwychwstałego gracza do ostatniego ukończonego poziomu.

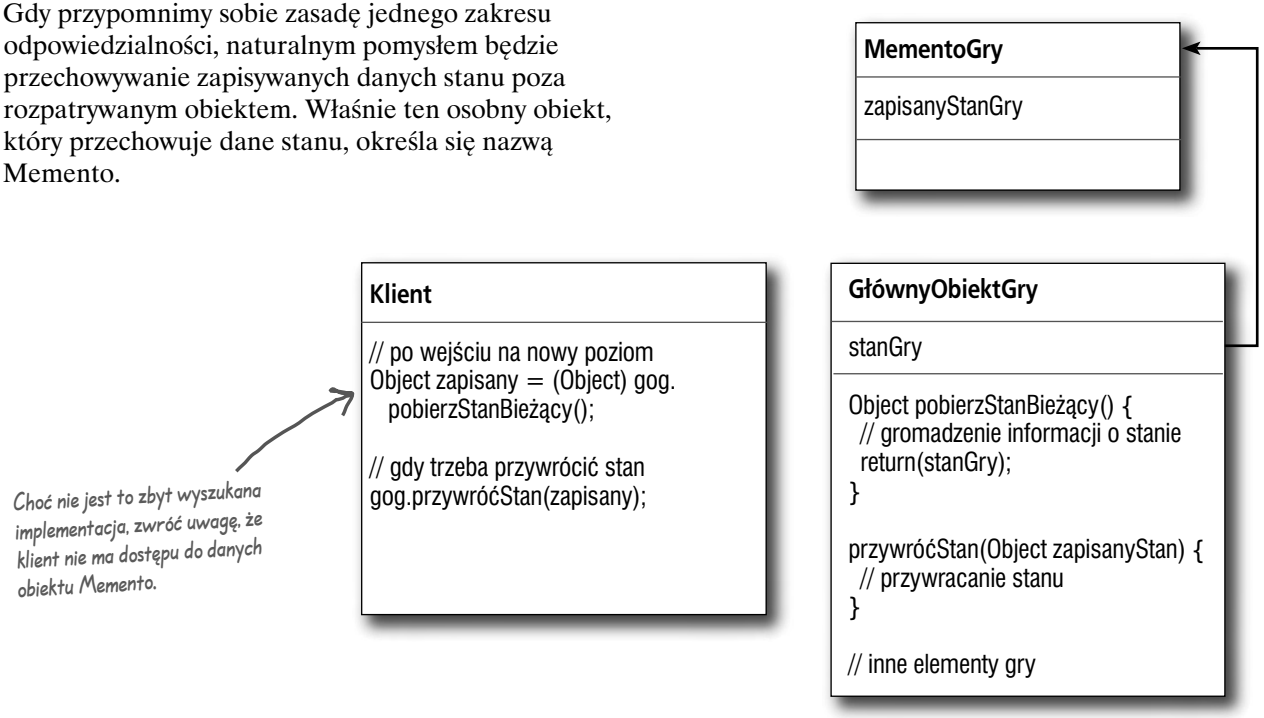


Wzorzec Memento w akcji

Wzorzec Memento ma dwa cele:

- Zapisanie istotnych danych stanu kluczowego obiektu systemu.
- Hermetyzacja tego obiektu.

Gdy przypomnimy sobie zasadę jednego zakresu odpowiedzialności, naturalnym pomysłem będzie przechowywanie zapisywanych danych stanu poza rozpatrywanym obiektem. Właśnie ten osobny obiekt, który przechowuje dane stanu, określa się nazwą Memento.



Zalety wzorca Memento

- Pozostawienie zapisywanego stanu poza głównym obiektem pomaga zachować spójność.
- Zapewnia hermetyzację danych głównego obiektu.
- Zapewnia funkcję przywracania, która jest stosunkowo łatwa do implementacji.

Zastosowania i wady wzorca Memento

- Wzorzec Memento służy do zapisywania stanu.
- Wadą wzorca Memento jest to, że zapisywanie i przywracanie danych może być czasochłonne.
- W systemach opartych na języku Java warto rozważyć zastosowanie serializacji do zapisywania stanu systemu.

Prototyp

Wzorca Prototyp używamy wtedy, gdy tworzenie egzemplarza danej klasy jest w pewien sposób kosztowne.

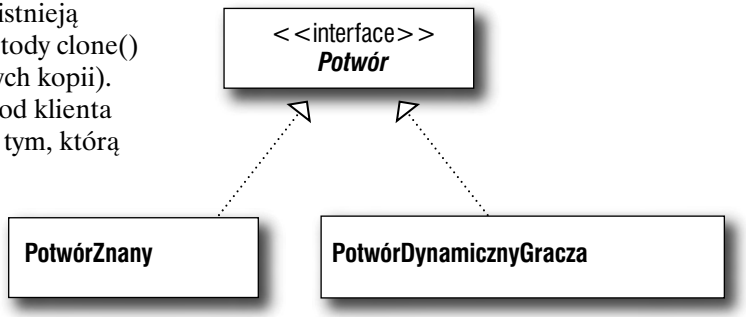
Scenariusz

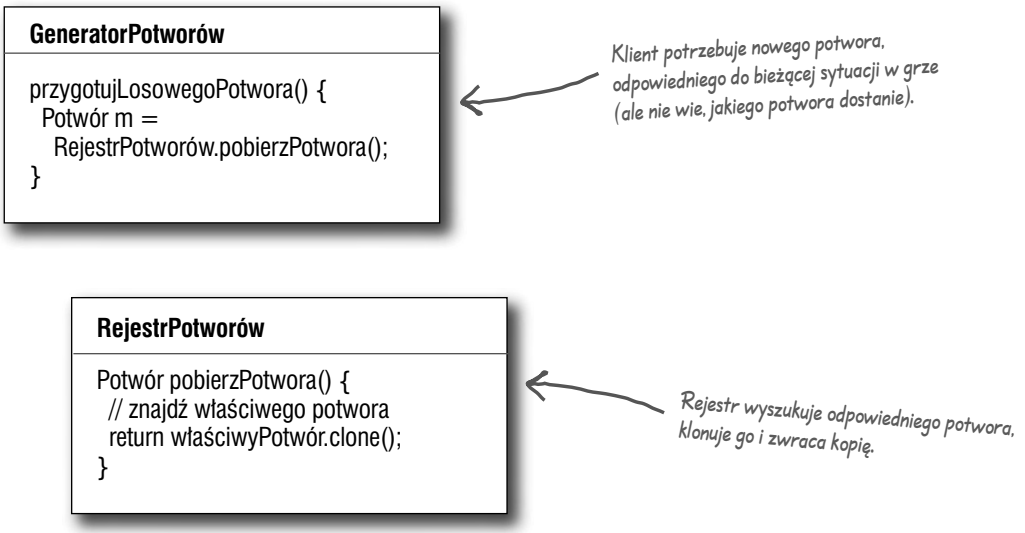
W Twojej interakcyjnej grze RPG wciąż pojawiają się najróżniejsze potwory. Kiedy bohaterowie wędrują przez dynamicznie generowany krajobraz, napotykają niekończący się łańcuch wrogów. Chciałbyś, żeby charakterystyka potworów ewoluowała wraz ze zmianami krajobrazu. Pozwoli to uniknąć takich sytuacji, jak wyświetlanie ptaków w krainach podwodnych. Co więcej, dalszym etapem będzie umożliwienie zaawansowanym graczom samodzielnego tworzenia potworów.



Wzorzec Prototyp rozwiązuje problem

Wzorzec Prototyp umożliwia tworzenie nowych egzemplarzy przez kopiowanie tych, które już istnieją (w języku Java oznacza to zazwyczaj użycie metody clone() lub deserializację, gdy potrzebujemy pełniejszych kopii). Kluczowym aspektem tego wzorca jest to, że kod klienta może tworzyć nowe egzemplarze bez wiedzy o tym, którą klasą rzeczywistą operuje.





Zalety wzorca Prototyp

- Ukrywa złożoność operacji tworzenia nowych egzemplarzy przed klientem.
- Zapewnia klientowi możliwość generowania obiektów, których typ nie jest znany.
- W pewnych warunkach kopiowanie obiektów może być wydajniejsze niż generowanie nowych.

Zastosowania i wady wzorca Prototyp

- Warto rozważyć zastosowanie wzorca Prototp, gdy system wymaga tworzenia nowych obiektów różnych typów w złożonej hierarchii klas.
- Wadą wzorca jest to, że utworzenie kopii obiektu może być niekiedy bardzo skomplikowane.

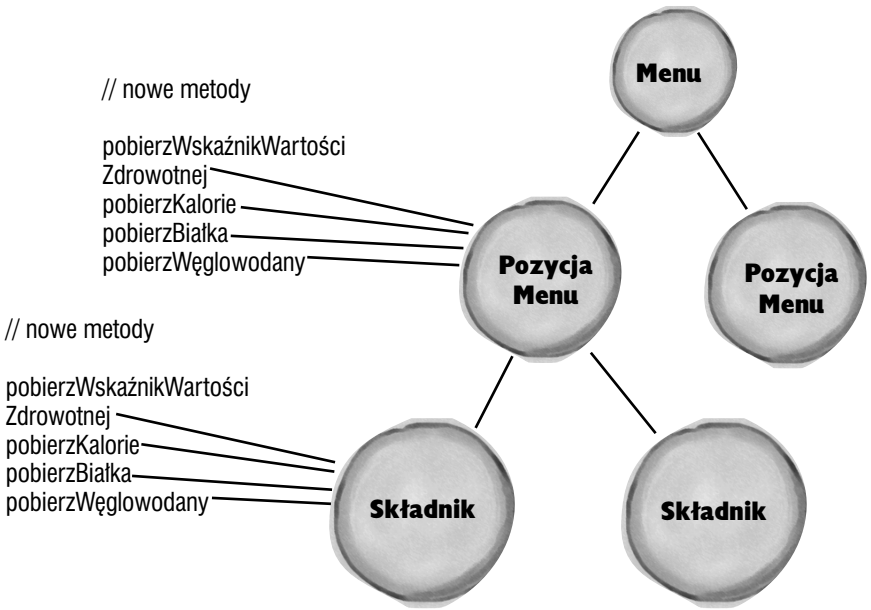
Visitor (gość)

Wzorca Visitor używamy, aby rozbudować zakres funkcji obiektów kompozytowych, gdy hermetyzacja nie jest ważna.

Scenariusz

Klienci odwiedzający restauracje Diner i Pancake House w Obiektowie zaczęli w ostatnich latach bardziej dbać o swoje zdrowie. Przed zamówieniem posiłku żądają teraz informacji o jego wartościach odżywczych. Ponieważ oba kierownictwa bardzo chętnie przyjmują zamówienia specjalne, niektórym klientom trzeba przedstawiać informacje obliczone na podstawie składników zamówionych potraw.

Łukasz zaproponował takie rozwiązanie:

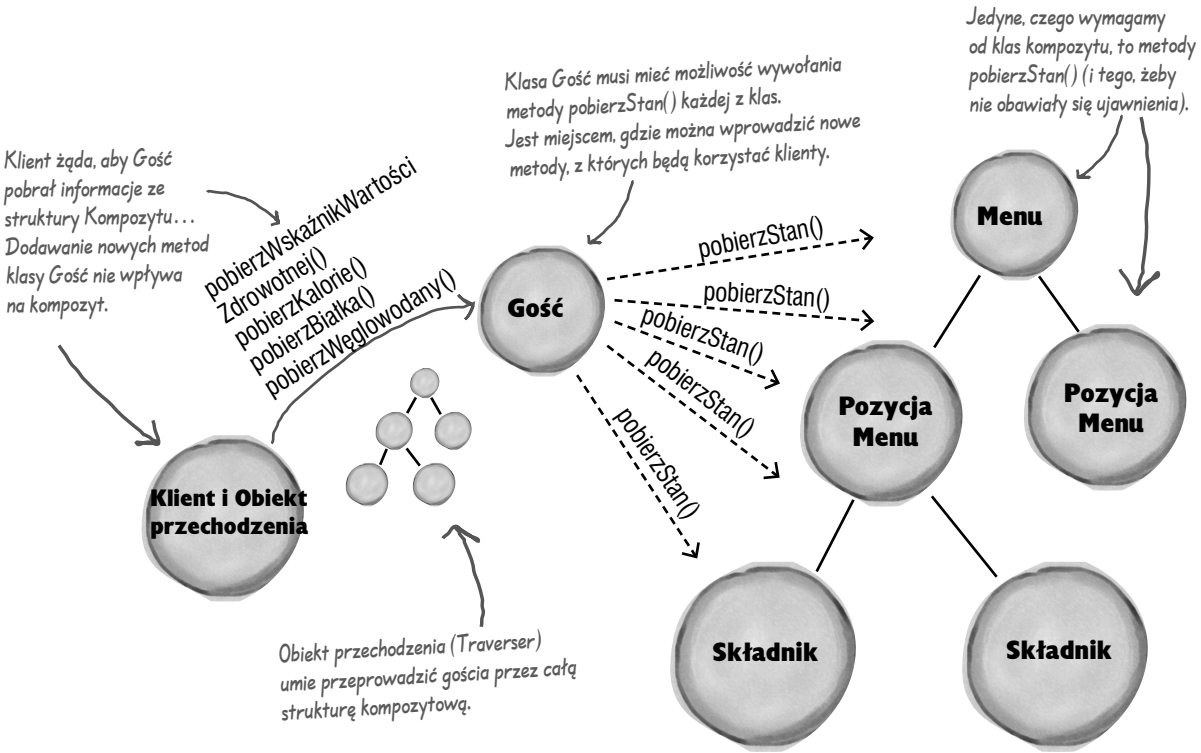


Miłosz zastanawia się...

„Człowieku, to wygląda jak puszka Pandory. Bóg jeden wie, jakie dalsze metody będziemy musieli jeszcze dodawać, a za każdym razem, gdy będziemy to robić, musimy dbać o to w dwóch miejscach. Więcej, co stanie się, gdy zapragniemy rozszerzyć naszą aplikację o, powiedzmy, klasę przepisów? Wówczas będziemy musieli robić zmiany w trzech różnych miejscach…”.

Gość w dom...

Obiekt Visitor („gość") odwiedza każdy element kompozytu. Ta funkcja jest częścią obiektu Traverser, czyli „obiektu przechodzenia". Zgodnie z jego nakazem obiekt gościa gromadzi dane stanu wszystkich elementów kompozytu. Po wykonaniu takiej operacji klient może żądać od klasy gościa wykonywania różnorodnych operacji na pobranych danych. Gdy pojawia się potrzeba dodania nowych funkcji, jest to jedyna modyfikowana klasa.



Zalety wzorca Visitor

- Umożliwia dodawanie nowych operacji do struktury kompozytowej bez modyfikowania samej struktury.
- Dodawanie nowych operacji jest stosunkowo proste.
- Kod operacji wykonywanych przez Gościa jest skupiony w jednym miejscu.

Zastosowania i wady wzorca Visitor

- Wprowadzenie wzorca Visitor łamie hermetyzację uzyskaną dzięki zastosowaniu struktury kompozytowej.
- Ponieważ wykorzystywana jest funkcja przechodzenia między elementami kompozytu, wprowadzanie w niej zmian jest utrudnione.

Skorowidz

A

Abstract Factory, 183
AbstractButton, 102
abstrakcja interfejsu, 630
abstrakcyjna metoda fabrykująca, 161
ActionListener, 102
Adapter, 259, 260, 265, 267, 276, 283, 284, 294, 489, 522, 541
  definicja, 266, 267
  diagram klas, 267
  EnumerationIterator, 274
  implementacja, 274
  klasy, 268, 271
  klient, 265
  obiektowość, 261
  obiekty, 268, 271
  projektowanie, 273
  remove(), 274
  testowanie, 264
addObserver(), 95
adres URL, 466, 482
agregat, 345, 356
  przechodzenie przez elementy, 356
aktualizacja(), 85, 87, 89
algorytmy, 56, 297, 303
  sortowanie, 324
  szkielet, 311
  wydobywanie, 304
antywzorce, 624
API, 61
aplety Java, 329
Array, 322, 335
ArrayList, 344, 358
automat, 405
  implementacja, 408
  klient monitora, 472
  monitor, 449
  obiekty stanowe, 420
  promocja 1 z 10, 431
  reprezentacja stanu, 417
  stany, 405
  test wewnętrzny, 410
  testowanie, 434
  zdalne żądania, 469
  zdalny interfejs, 470
  zdalny pośrednik, 468
  zmiany stanu, 426

B

bezpieczeństwo, 533
  typologiczne, 369
biblioteki, 61
  Swing, 102
błędy czasu wykonania, 165
BPM, 552
Bridge, 630
Budowniczy, 632
BufferedInputStream, 130
Builder, 632, 633

C

Caching Proxy, 489, 506, 508
Chain Of Responsibility, 634
ClassLoader, 212, 213
Cofnij, 217
Collection, 368
Collections, 272
Command, 222
Comparable, 323
  implementacja, 324
compareTo(), 323, 326
Complexity Hiding Proxy, 507
Composite, 376
compound pattern, 518
Copy-On-Write Proxy, 507, 508
CopyOnWriteArrayList, 507
currentItem(), 358
część-całość, 376

D

definicja wzorców projektowych, 598
definiowanie wzorców, 605
deklaracja metody typu Fabryka, 155
Dekorator, 109, 118, 121, 134, 276, 489, 490, 508, 524, 541
  abstrakcyjny, 127
  charakterystyka, 120
  definicja, 121
  diagram klas, 121, 122
  implementacja, 121, 125, 126
  java.io, 131
  łańcuch przetwarzania, 129
  obsługa wejścia-wyjścia, 130, 132
  przeznaczenie, 121
  testowanie, 128, 133
  tworzenie, 123
dekorowanie obiektów, 109
delegacja, 115, 118, 135
deleteObserver(), 95
diagram
  stanów, 405, 413
  ukrytych zachowań, 54
DIP, 169, 170, 192
doGet(), 571
dokumentacja, 241
doPost(), 570
drzewo obiektów, 376
  składniki, 378
duplikacja kodu, 39
dynamic class downloading, 466
Dynamic Proxy, 492, 504
dynamiczna zmiana zachowania, 52, 53
dynamiczne pobieranie klas, 466

dynamiczne przydzielanie nowych zachowań obiektowi, 121
dynamiczny pośrednik, 497
dziedziczenie, 37, 112, 115, 123, 135
  wielokrotne, 268

E

efekty uboczne, 36
EJB, 601
enum, 165
Enumeration, 272, 273, 358
EnumerationIterator, 274
enumeratory, 272

F

Fabryka, 139, 165, 526, 541
  deklaracja metod, 155
  implementacja, 176
  interfejs, 176
  metody, 154
  składniki, 175
  użycie, 182
  zastosowanie, 144
Fabryka Abstrakcyjna, 183, 186, 188, 190, 192, 528, 541
  definicja, 186
fabryka statyczna, 145
Facade, 282
Factory, 144
Factory Method, 161
Fasada, 259, 278, 282, 284, 288, 294
  definicja, 288
  implementacja, 285, 286
  reguła ograniczania interakcji, 293
  zastosowanie, 284
fasolki, 568
FilterInputStream, 130, 132
Firewall Proxy, 506, 508
Flyweight, 636
for-in, 369

G

Gang Czterech, 619
Gang of Four, 601, 619
garbage collector, 212
generics, 369
gniazda, 455
GoF, 609
Gość, 646
grupowanie obiektów, 335
GUI, 104, 567

H

haczyk, 314, 317, 328, 332
  aplety Java, 329
  zastosowanie, 315
Hashtable, 335, 358, 363, 368
hasMoreElements(), 358
hasNext(), 358, 390
hermetyzacja, 41, 141, 166
  algorytmy, 297
  iteracja, 343, 344
  stan obiektu, 428
  tworzenie obiektów, 144, 161
  wywołania, 217
  żądania, 232
hierarcha część-całość, 376
Hollywood, 318, 319
HTTP, 567
  Get, 466

I

Icon, 481
identyfikacja elementów zmiennych, 142
ImageIcon, 489
implementacja, 43
  interfejsu zdalnego, 461
  usługi zdalnej, 463
  zachowań, 45
IncovationHandler, 497
instanceof, 387
integracja zachowania, 47
inteligentny pośrednik odwołań, 506
interakcje między obiektami, 290
interfejs, 38, 43, 44, 259
  Collection, 368
  Comparable, 323
  Enumeration, 272, 358
  InvocationHandler, 492, 498
  Iterator, 272
  konwersja, 267
  Observer, 94
  Serializable, 460
  tworzenie, 44
  usługi zdalnej, 463
  zdalny, 459, 460
interfejs użytkownika, 566
internet, 481
Interpreter, 638
InvocationHandler, 492, 498, 499, 504, 508
  implementacja, 498
invoke(), 498
isDone(), 358
iteracje, 343
Iterator, 272, 273, 335, 343, 344, 345, 352, 353, 388, 394, 399, 541
  definicja, 356
  diagram klas, 357
  implementacja, 345, 346, 358
  kolejność iterowania, 358
  zakres odpowiedzialności, 359
  zastosowanie, 356
IteratorKompozytu, 389
IteratorPusty, 392
iteratory, 272, 345, 388
  budowanie, 346
  Java 5, 369
  kolekcje, 368
  kompozyt, 389
  puste, 392
  wewnętrzne, 358, 390
  zewnętrzne, 358, 390

J

Java, 29, 61
Java 5, 210, 369
Java Collection Framework, 358, 368
Java DJ View, 552
Java Remote Method Invocation, 454
java.io, 130
java.lang.reflect, 492
java.lang.reflect.Proxy, 504
java.rmi.Naming, 461
java.rmi.Remote, 460
java.util.Iterator, 353
java.util.Observable, 97, 101
JavaBean, 567, 568
JavaScript, 568
JavaSpace, 507
JButton, 102, 103
JDK, 67, 459
jeden obiekt, 198
jeden zakres odpowiedzialności, 359, 387
jeden-do-wielu, 67, 81
jednolite słownictwo, 617
JEST, 55
język Java, 29
JFrame, 103, 328
JSP, 567, 572
JVM, 452

K

katalog wzorców, 597, 602
kategorie wzorców, 607, 608, 610
klasy, 36, 46
  abstrakcyjne, 302
  adapter, 268
  Arrays, 322
  BufferedInputStream, 130
  Collections, 272
  dynamiczna zmiana zachowania, 52
  FilterInputStream, 130, 132
  Hashtable, 368
  InvocationHandler, 498, 504
  IteratorPusty, 392
  JFrame, 328
  nadrzędne, 42
  Observable, 94, 95, 97
  ochrona, 494
  podrzędne, 101, 123, 151
  Proxy, 504
  równoległa hierarchia, 162
  testowe, 51
  UnicastRemoteObject, 461
  wewnętrzne, 103
  zachowania, 46
  zakres odpowiedzialności, 359
klasyfikacja wzorców, 609
klient, 455, 464, 466
kod
  JSP, 567
  polimorficzny, 358
kohezja, 359
kolejkowanie żądań, 232, 254
kolekcje, 335, 345, 358
  iteratory, 345, 368
  Java 5, 369
  kolejność iterowania, 358
  przechodzenie między elementami, 356
  usuwanie elementów, 352
kompozycja, 55, 101, 115, 123, 135
Kompozyt, 335, 343, 376, 388, 394, 396, 399, 531, 541, 576
  definicja, 376
  implementacja, 379, 380
  iterator, 389
  jeden zakres odpowiedzialności, 387
  liście, 378
  menu, 379
  składniki, 378
  struktura, 382
  testowanie, 384
  zastosowanie, 376
komunikacja sieciowa, 452, 475
kontekst, 597, 598
kontrola dostępu do obiektu, 447
kontroler, 544, 549, 553, 560, 589, 590
  implementacja, 561
  serwlet, 569, 570
konwersja interfejsu, 266

L

LinkedList, 368
List, 335
liście, 378
logika aplikacji, 577
lokalna aktualizacja kodu, 36
luźne zależności, 82, 83

Ł

ładowanie obrazu z sieci, 481
Łańcuch Odpowiedzialności, 634, 635
łączenie wzorców, 517

M

MA, 55
makropolecenia, 250, 251
maszyna stanowa, 406
mechanizm
  Cofnij, 217
  generics, 369
  wywołań zdalnych, 453
Mediator, 577, 640
Memento, 642
menu, 338, 340, 362
  Kompozyt, 379
Menu XML, 342
Meta Polecenie, 232
metapoznawanie, 25
Metoda Fabrykująca, 161, 164, 165, 188, 190, 192
  definicja, 164
  równoległa hierarchia klas, 162
Metoda Szablonowa, 297, 301, 308, 311, 321, 327, 330, 332
  aplety Java, 329
  definicja, 311

Metoda Szablonowa
haczyk, 314, 317, 328
implementacja, 309
Java API, 327
reguła Hollywood, 318, 319
sort(), 323
sortowanie, 322
zastosowania, 310
metody
abstrakcyjne, 150
compareTo(), 323
Fabryka, 154, 155
hermetyzacja wywołań, 217
klasowe, 200
statyczne, 200
swap(), 326
model, 544, 549, 553, 564, 569, 570, 577, 584
Model 2, 575, 576, 578
Model-Widok-Kontroler, 89, 518, 540, 544, 547, 578
implementacja, 555
implementacja kontrolera, 561
implementacja widoku, 558
Kompozyt, 551, 576
kontroler, 553, 560, 589, 590
model, 553, 564, 570, 584
Model 2, 575, 576
Obserwator, 551, 575
sieć WWW, 567
Strategia, 550, 551, 563, 576
widok, 552, 557, 586, 588
wzorce projektowe, 546
monitor automatu, 450
Most, 630, 631
motywacja, 603
MVC, 518, 577
MXML, 342

N

Naming, 461
new, 139, 141, 173
newProxyInstance(), 504
next(), 358, 390
nextElement(), 358
niska kohezja, 359
notifyObserver(), 95
notifyObservers(), 95, 96
null, 240

O

obiekty
aktualizacja danych, 77
dekorowanie, 109, 118
grupowanie, 335
hermetyzacja tworzenia, 161
interakcje, 290
JavaBean, 567
kolekcje, 335
kontrola dostępu, 447
luźne zależności, 83
nasłuchujące, 103
null, 240
obserwowany, 75, 78, 92, 104
obserwujący, 75, 78, 92, 104
polecenie, 222
pomocniczy, 457, 458, 464
pomocniczy klienta, 504
pośrednik, 501
pośrednik klienta, 461
Proxy, 447
przechowywanie, 345
puste, 240
relacje jeden-do -wielu, 81
stanowe, 420
tworzenie, 140, 144
zależności, 168
zdalny pośrednik, 452
zmiana stanu, 81, 403
zmiana zachowania, 428
Observable, 94, 95, 97, 101, 104
addObserver(), 95
deleteObserver(), 95
metody, 101
notifyObservers(), 95, 96
setChanged(), 95, 96
Observer, 94, 117
Obserwator, 67, 73, 74, 104, 534, 541
aktualizacja(), 85, 87, 89
biblioteka Swing, 102
definicja, 80, 81
diagram klas, 82
dodawanie obserwatorów, 83
działanie, 76
implementacja, 81, 87
JDK, 102
luźne zależności, 83
obsługa, 87
powiadomienia o zmianie stanu, 95
testowanie, 90
wzorzec wbudowany, 94, 95, 97
zastosowanie, 94
obsługa wejścia-wyjścia, 130, 132
testowanie dekoratora, 133
oddzielanie składników, 180
odpowiedź HTTP, 567, 575
odwracanie zależności, 169, 170
ograniczanie interakcji, 289
OOPSLA, 621
operatory
instanceof, 387
new, 140, 141
otwarte-zamknięte, 116

P

parametryzacja obiektów, 232
pętla for-in, 369
podklasy, 152
podwójne blokowanie, 210
Polecenie, 217, 222, 227, 232, 256
definicja, 232, 233
diagram klas, 233
implementacja, 229, 236, 237
kolejkowanie żądań, 254
makropolecenia, 251
model, 226
obiekt null, 240
obiekt wywołujący, 235
testowanie, 238

wycofywanie operacji, 242, 244
zastosowanie, 230, 254
żądania rejestracji, 255
polimorfizm, 44, 49, 141
połączenie z internetem, 481
pośrednik, 447, 451, 458, 462
buforujący, 489, 506, 508
chroniący, 489, 492, 508
dynamiczny, 492, 497, 504
kopiujący, 507, 508
synchronizujący, 507, 508
ukrywający złożoność, 507
wirtualny, 480, 507
zapora sieciowa, 506
zdalny, 480
powiązania międzyobiektowe, 139
PriorityQueue, 368
problem, 597, 598
procesy biznesowe, 623
programowanie zorientowane obiektowo, 35, 117
projektowanie, 41
klas, 302
wirtualnego pośrednika, 482
zachowania, 43
zorientowane obiektowo, 39, 59
promowanie słownictwa, 618
PropertyChangelistener, 102
Prosta Fabryka, 147, 165, 192
Protection Proxy, 489, 492, 508
Prototyp, 644
proxy, 451
Proxy, 447, 478, 490, 501, 508
definicja, 478
diagram klas, 479
kontrola dostępu, 478
pośrednik buforujący, 489
pośrednik chroniący, 489, 492
pośrednik dynamiczny, 492
pośrednik wirtualny, 480
pośrednik zdalny, 480
zastosowanie, 478
przechowywanie danych, 335, 338, 345
przekazywanie żądań, 455
przezroczystość, 387, 533

R

ramki, 328
read(), 327
rebind(), 461
refactoring, 374, 613
reguły
DIP, 169, 170
Hollywood, 318, 319
odwracania zależności, 169
ograniczania interakcji, 289, 290, 292
otwarte-zamknięte, 116, 117
projektowania, 41, 43, 55
trzech, 605
rejestr RMI, 459, 461
wprowadzanie wpisu, 471
rejestracja, 232, 255
relacje, 54
jeden-do-wielu, 67, 81
JEST, 55
MA, 55
Remote, 460, 461
Remote Proxy, 480, 482, 489, 508
RemoteException, 460, 461
remove(), 274, 352, 358
reprezentacja stanu, 416, 417
RMI, 104, 453, 454, 458, 467, 470, 504
klient, 464, 466
obiekt pomocniczy, 464, 465
pośrednik, 462
rejestr, 461
rmiregistry, 462
serwer, 463
szkielet, 462
tworzenie usługi zdalnej, 459
wprowadzanie wpisu w rejestrze, 471
wyszukiwanie w rejestrze, 465
rmic, 459
rmiregistry, 459, 462
rodzina algorytmów, 54
rozbudowa funkcjonalności kodu, 117
rozwiązanie, 597, 598
równoległa hierarchia klas, 162

S

Serializable, 460
serializacja, 460, 466, 470
serwer, 455
serwlet, 567
kontroler, 570
setChanged(), 95, 96
sieć WWW, 567
Simple Factory, 147
Singleton, 197, 199, 202, 205, 214
ClassLoader, 213
definicja, 205
diagram klas, 205
globalne odwołanie, 212
implementacja, 201
podwójne blokowanie, 210
tworzenie wzorca, 200
wątki, 206
wielowątkowość, 208
skeleton, 458
składniki, 175, 378
niskiego poziomu, 169, 320
wysokiego poziomu, 169, 320
Smart Reference Proxy, 506
sort(), 323
sortowanie, 322, 324
Stack, 335, 368
Stan, 403, 430, 436, 441
definicja, 428
diagram klas, 428
implementacja, 422
reprezentacja, 416, 417
zmiany, 426
stan modelu, 577
static factory, 145
Strategia, 56, 330, 429, 436, 563, 576
Strategy, 56
strony
JSP, 568, 572
WWW, 329
struktury drzewiaste, 376
strumienie, 131
stub, 458
superklasy, 42, 123

superkolekcje obiektów, 335
swap(), 326
Swing, 102, 104, 481
synchronizacja dostępu do metody, 208
Synchronization Proxy, 507, 508
system
  zamówień, 110
  zarządzania zawartością, 506
  zdalnego sterowania, 221, 232
szkielet, 458, 462, 504
  algorytmu, 311
  aplikacji, 61

T

tablice, 322
terminal, 462
testowanie
  kodu klasy, 51
  zachowania, 50
tworzenie
  dynamicznego pośrednika, 497
  egzemplarzy wirtualnych, 636
  implementacji zdalnej, 459
  interfejsu, 43, 44
  interfejsu zdalnego, 459, 460
  iteratora, 346
  jednego obiektu, 198
  obiektów, 140, 144
  oprogramowania, 40
  pośrednika chroniącego, 492
  usługi zdalnej, 459
  zależnych obiektów, 186

U

Undo, 217
UnicastRemoteObject, 461
update(), 95
uproszczony interfejs, 286
uruchamianie usługi zdalnej, 459
usługi zdalne, 455, 459, 469
usuwanie elementów kolekcji, 352

V

Vector, 358, 368
Virtual Proxy, 480, 489, 508
Visitor, 646, 647
volatile, 210

W

Waga Piórkowa, 636
wątki, 206, 207
wbudowany wzorzec Obserwator, 94, 95
wejście-wyjście, 130
widok, 544, 549, 552, 557, 577, 586, 588
  HTML, 569
  implementacja, 558
  strony JSP, 572
wieloetapowe inicjowanie, 632
wielokrotne dziedziczenie, 268
wielokrotne wykorzystanie kodu, 331, 332
wielowątkowość, 206, 208, 209, 507
wirtualna maszyna Javy, 207
wirtualny pośrednik, 508
wspólny słownik, 58, 59, 60
współużytkowanie wyników, 506
WWW, 567
wycofaj(), 242
wycofywanie operacji, 232, 242
  implementacja, 244, 246
  przechowywanie stanu, 246
wydobywanie algorytmów, 304
wysoka kohezja, 359
wyświetlanie danych, 89
wywołania metod, 291
  zdalne, 452, 453
wzorce
  aplikacji, 622
  klas, 609
  konstrukcyjne, 607
  obiektów, 609
  operacyjne, 607, 608
  organizacyjne, 623
  procesów biznesowych, 623
  projektowe interfejsu, 623
  specyficzne, 622
  strukturalne, 607, 608
  w architekturze, 622
wzorce projektowe, 33, 39, 60, 64, 597, 626
  Adapter, 260, 265, 267
  Bridge, 630
  Budowniczy, 632
  Builder, 632
  Chain Of Responsibility, 634
  definicja, 597, 598, 599
  Dekorator, 109, 118
  Fabryka, 139
  Fabryka Abstrakcyjna, 183, 186
  Fasada, 278, 282, 288
  Flyweight, 636
  Gość, 646
  implementacja, 603
  Interpreter, 638
  Iterator, 345, 356
  katalog, 597
  kategorie, 608
  klasyfikacja, 609
  Kompozyt, 376
  konsekwencje, 603
  kontekst, 597
  Łańcuch Odpowiedzialności, 634, 635
  łączenie, 517
  Mediator, 640
  Memento, 642
  Meta Polecenie, 232
  Metoda Fabrykująca, 161, 164
  Metoda Szablonowa, 297, 308, 311
  Model-Widok-Kontroler, 544, 547
  Most, 630
  motywacja, 603
  MVC, 518
  nazwy, 603
  Obserwator, 67, 74
  Polecenie, 217, 222, 232

problem, 597
Prosta Fabryka, 147
Protection Proxy, 492
Prototyp, 644
Proxy, 447, 478
Remote Proxy, 482
rozwiązanie, 597
Singleton, 197, 205
Stan, 403, 428
Strategia, 56
struktura, 599
tworzenie, 604
uczestnicy, 603
Visitor, 646
Waga Piórkowa, 636
zastosowania, 61, 603
wzorce złożone, 517, 518, 540, 541
Model-Widok-Kontroler, 544

Z

zachowania klasy, 43, 46
zakres odpowiedzialności, 225, 359
Kompozyt, 387
zależności międzyobiektowe, 168
zarządzanie
agregatem, 359
kolekcjami, 335
zasada jednego zakresu odpowiedzialności, 359, 387
zbiór zachowań, 54
zdalne wywołania metod, 454, 455
wywołanie metody, 456
zdalne wywołania sieciowe, 475
zdalny pośrednik, 452, 468, 508
zgodność typów, 123, 165
zliczanie odwołań, 506
zmiany stanu, 426
zmienne
globalne, 213
obiektowe, 47
zmienne elementy aplikacji, 142

Ż

żądania
HTTP, 567, 569, 575
rejestracji, 255

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

helion kopia dla: Marek Stawarczyk mafikrpg@gmail.com