



Rusz głową! Wzorce projektowe

Jak unikać
niebezpiecznych
związków



Dlaczego powszechna
opinia o wzorcu Factory
mija się z rzeczywistością



Wzorce projektowe
i nowoczesne
metody nauki

Odkrywamy sekrety Mistrza
Wzorów



Jak sieć Star Cafe podwoiła
swoją wartość dzięki
wzorcowi Decorator



Jak unikanie
dziedziczenia
zmieni Twoje życie



Tytuł oryginału: Head First Design Patterns

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

ISBN: 978-83-283-3355-0

© 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>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:

<ftp://ftp.helion.pl/przyklady/wzorvv.zip>

Printed in Poland.

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

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

Spis treści (skrócony)

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

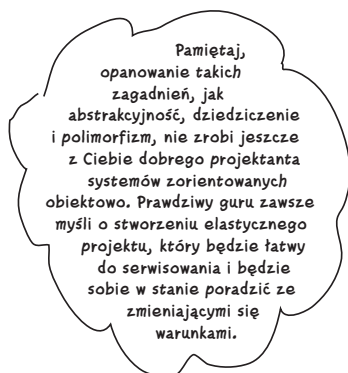
Dla kogo przeznaczona jest ta książka?	22
Wiemy także, co sobie myśli Twój mózg	23
Metapoznanie	25
Zmuś swój mózg do posłuszeństwa	27
Zespół recenzentów technicznych	30
Podziękowania	31

Wprowadzenie do wzorców projektowych

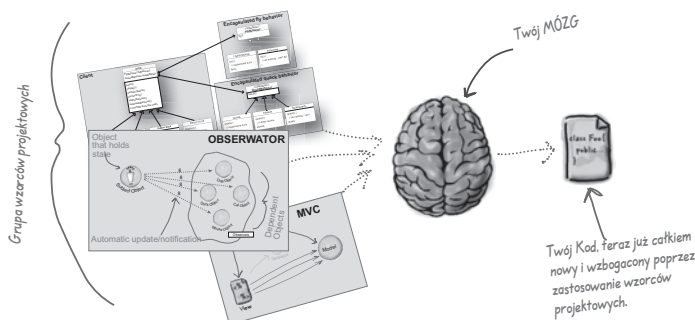
1

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 dojrniemy 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.

Podstawy programowania obiektowego

Abstrakcyjność

Hermetyzacja

Polimorfizm

Encapsulacja

Reguły programowania obiektowego

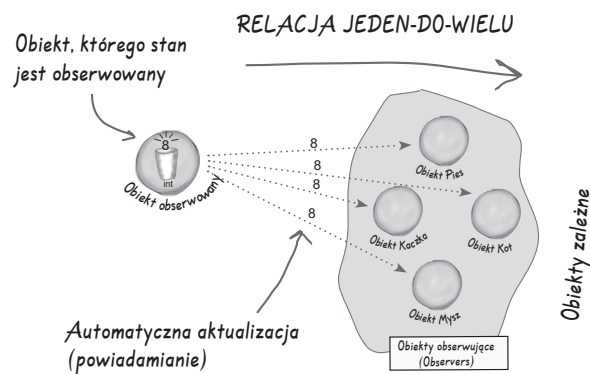
Poddawaj hermetyzacji to, co się zmienia.

Przedkładaj kompozycję nad dziedziczenie.

Skoncentruj się na tworzeniu interfejsów, a nie implementacji.

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

Aplikacja sprawdzająca warunki pogodowe	69
Spotkanie z wzorcem Obserwator	74
Wydawca + Prenumerat = 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 przyrzeć 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”	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 Café”	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

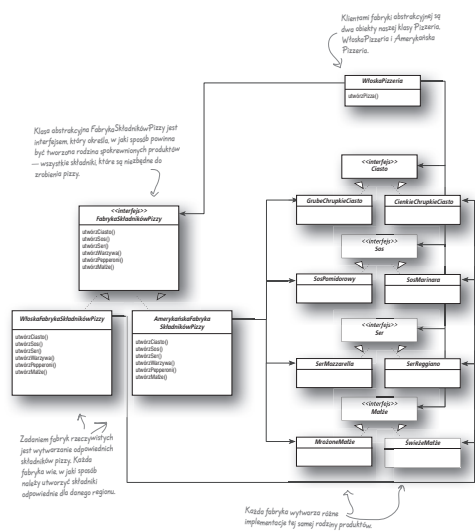
Zawsze sądziłem,
że prawdziwi mężczyźni tworzą
podklasy dla wszystkiego, co się tylko
do tego nadaje. Tak było do czasu,
gdy dowiedziałem się o korzyściach,
jakie daje rozszerzanie możliwości
aplikacji na poziomie działania,
a nie kompilacji. A teraz
— spójrzcie tylko na mnie!



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 Obiekcie	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

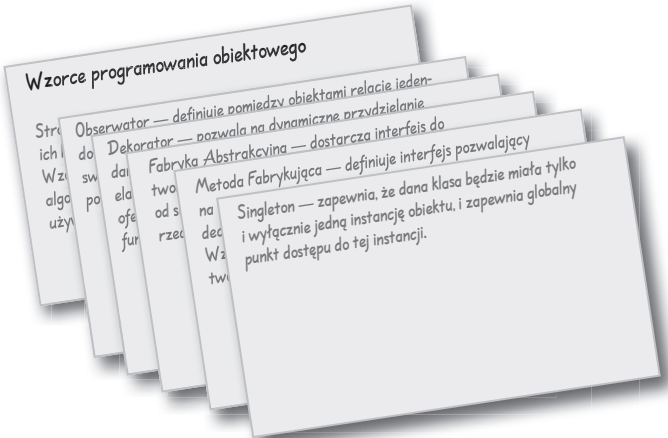
Obiekty jedyne w swoim rodzaju

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

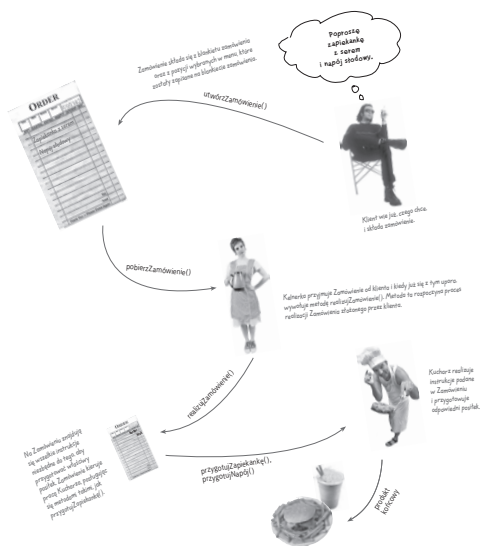
Jeden i tylko jeden	198
Mały Singleton	199
Analiza klasycznej implementacji wzorca Singleton	201
Wyznania obiektu Singleton	202
Fabryka czekolady	203
Definicja wzorca Singleton	205
Ups, mamy problem...	206
Zostań wirtualną maszyną Java	207
Jak sobie radzić z wielowątkowością?	208
Wzorzec Singleton —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



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.



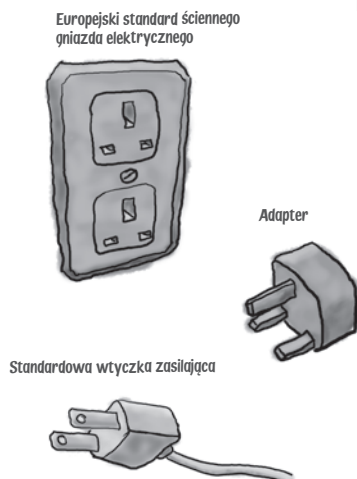
Automatyka w domu i zagrodzie	218
Mamy nową zabawkę! Sprawdzamy, jak działa SuperPilot...	219
Co zawiera otrzymany dysk CD-R?	220
A w międzyczasie w naszym barze szybkiej obsługi...	223
Przyjrzyjmy się nieco dokładniej wzajemnym interakcjom...	224
Zadania i zakresy odpowiedzialności	225
Od Baru do wzorca 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 — kolejkovanie żądań	254
Kolejne zastosowania wzorca Polecenie — żądania rejestracji	255
Twoja skrzynka narzędziowa	256
Rozwiązania ćwiczeń	258

Wzorce Adapter oraz Fasada

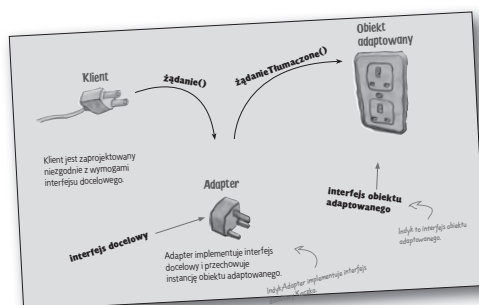
7

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. Pamiętajsz 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.



Adaptory są wśród nas	260
Adaptory zorientowane obiektowo	261
Wzorzec Adapter bez tajemnic	265
Definicja wzorca Adapter	267
Adaptory obiektów i klas	268
Temat dzisiejszej wieczornej pogawędki: Adapter obiektów i Adapter klas	271
Adaptory 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

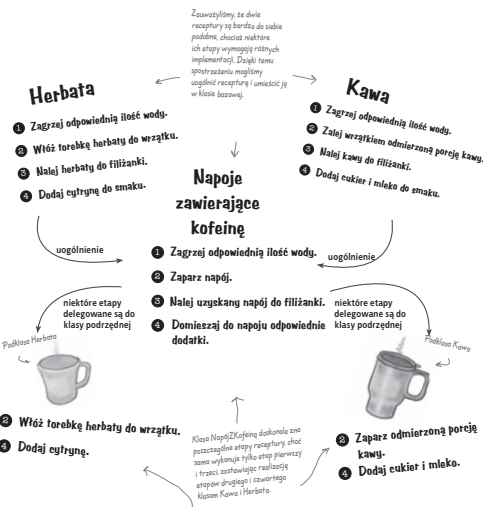


8

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

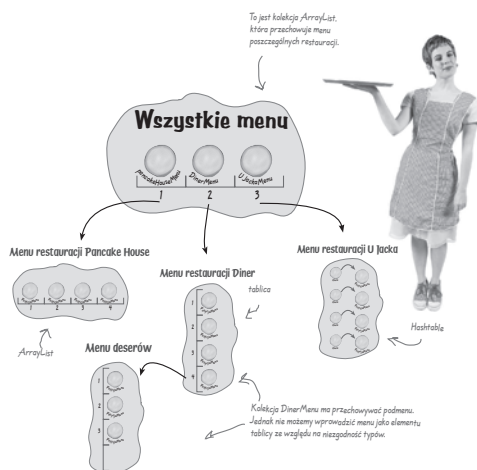


Wzorce Iterator i Kompozyt

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
Wzorec 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

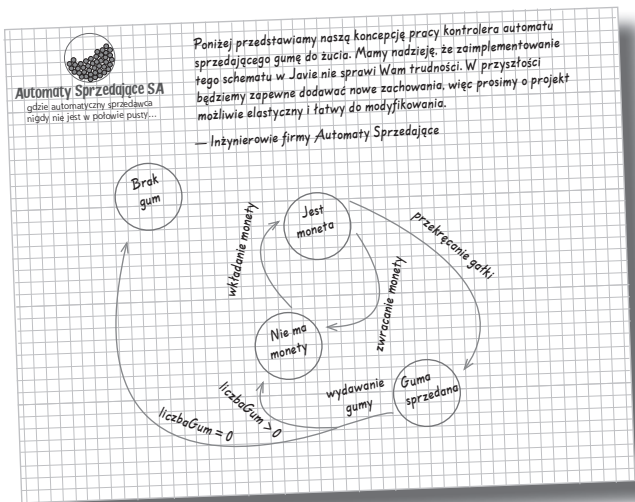
10

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...”.



Krótką 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

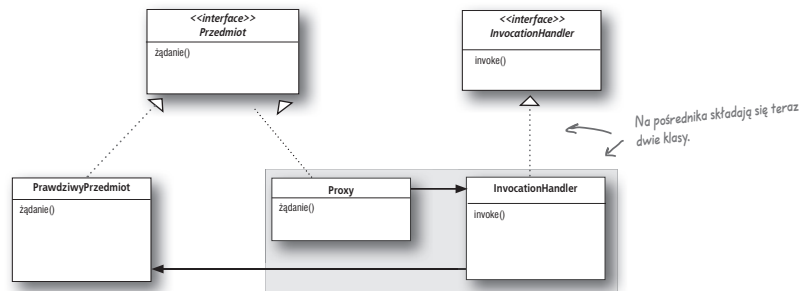
11

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 wywołanie metody obiektowi w innym węźle internetu; bywa też, że zastępują wyjątkowo leniwe obiekty.



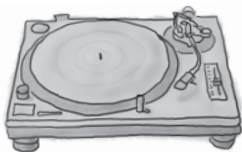
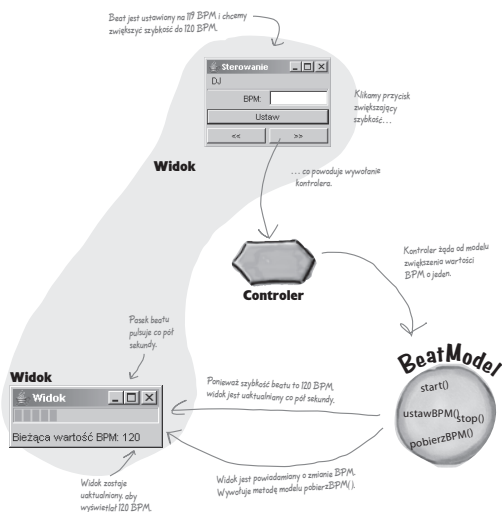
Kontrolowanie stanu automatów sprzedających	448
Rola „zdalnego pośrednika”	452
RMI — wycieczka z przewodnikiem	455
Zdalny pośrednik automatu sprzedającego	468
Pośrednik zdalny, za kulisami	476
Definicja wzorca Proxy	478
Pośrednik wirtualny	480
Projektowanie wirtualnego pośrednika do wyświetlania okładek	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



12

Łączenie wzorców

Przyszedłoby 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.



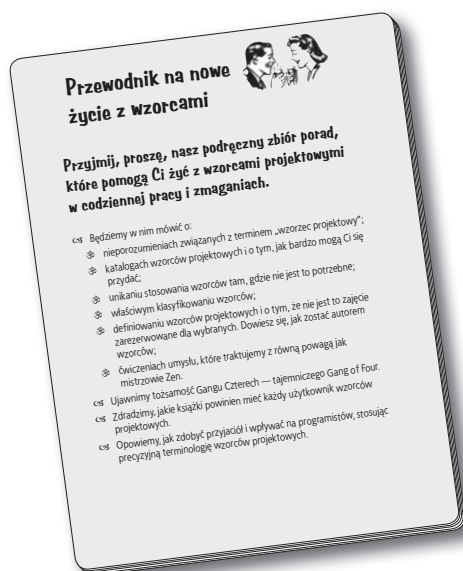
Wzorec złożony	518
Powrót kaczek	519
Potrzebujemy adaptera gęsi	522
Wprowadzamy zliczanie kwaknięć	524
Fabryka produkująca kaczki	526
Tworzymy stado kaczek	531
Przygotowanie interfejsu Observable	534
Co zrobiliśmy?	541
Widok z lotu kaczki — diagram klas	542
Model-Widok-Kontroler — piosenka	544
Kluczem 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
Wzorec MVC i sieć WWW	567
Model 2 a wzorce projektowe	575
Twoja skrzynka narzędziowa	578
Rozwiązania ćwiczeń	579

Nowe życie z wzorcami

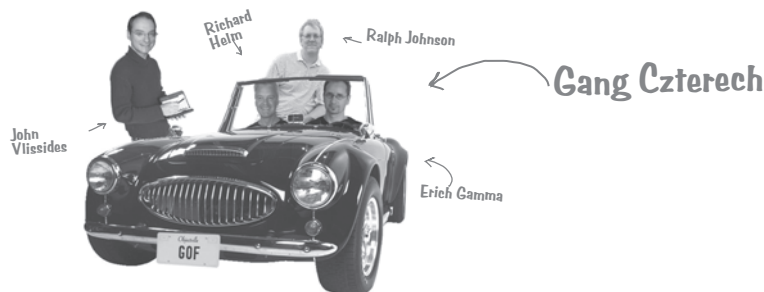
13

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 Obiekcie. 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 Obiekcie	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

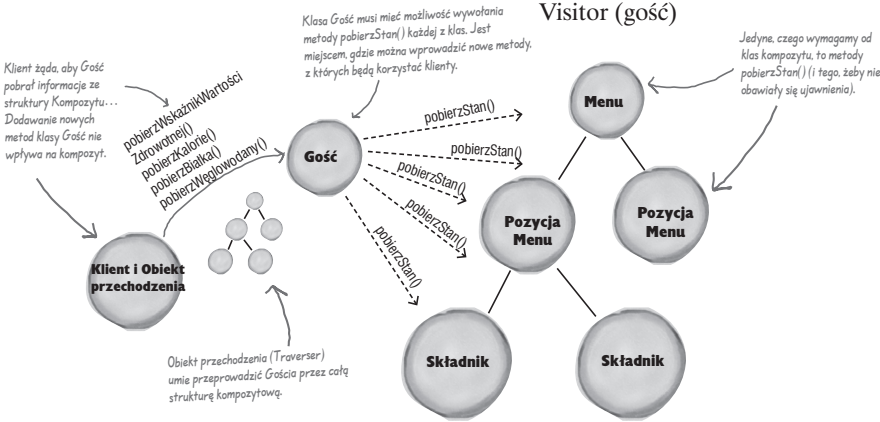


14

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

2. Wzorzec Obserwator

Jak sprawić, by Twoje obiekty były zawsze dobrze poinformowane



Cześć, Jarek, właśnie dzwonię po kolei do wszystkich z informacją, że spotkanie naszej grupy użytkowników wzorców zostało przeniesione na sobotni wieczór. Będziemy omawiać wzorzec Obserwator. Tak, ten wzorzec jest najlepszy. Jarek, on jest po prostu NAJLEPSZY!

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.



Pogodyнка 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 Pogodyнка 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 wspólnie firmy!

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

Z poważaniem,

Janusz Monsun

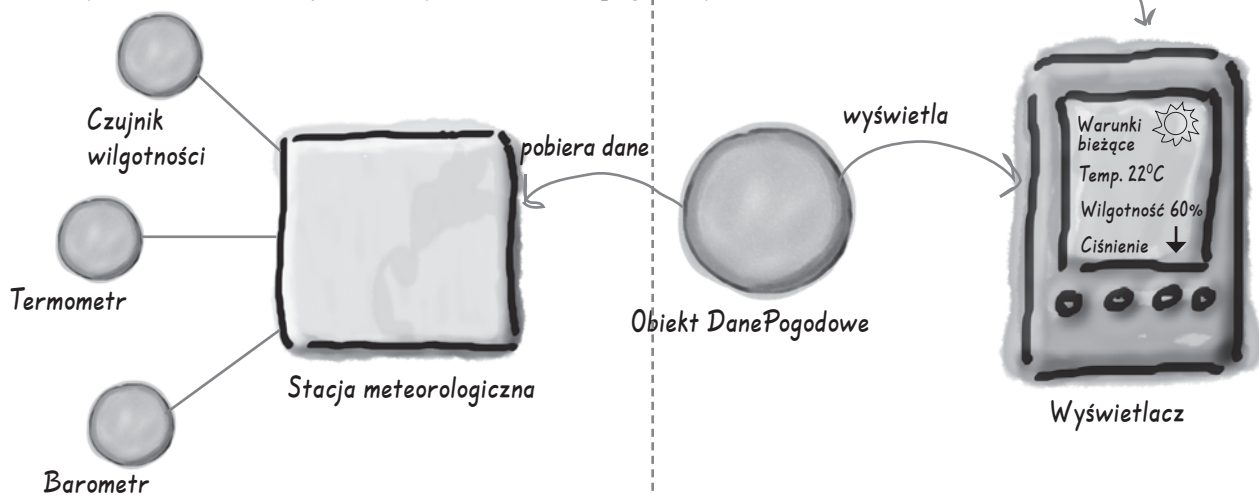
Janusz Monsun, Prezes

P.S. Pliki źródłowe zawierające kod naszego opatentowanego obiektu DanePogodowe wysłamy 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.

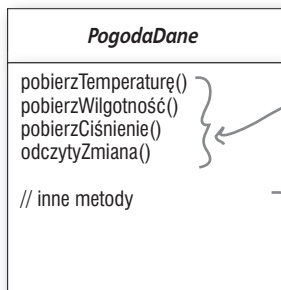
Elementy, które musimy zaimplementować

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 obiecano, 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 tryb wyświetlania



Drugi tryb wyświetlania



Trzeci tryb wyświetlania



Przyszłe tryby wyświetlania

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 Pogodyнка 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.

Hmmmm, ja
wiem, że jestem tutaj nowy,
ale ponieważ znajdujemy się właśnie
w rozdziale dotyczącym wzorca
Obserwator, może w końcu
zaczęlibyśmy z niego korzystać?



**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.

Tęsknisz za nowymi informacjami z Obiektowa? Nie ma innego wyjścia, musimy zaprenumerować tamtejszą gazetę codzienną!

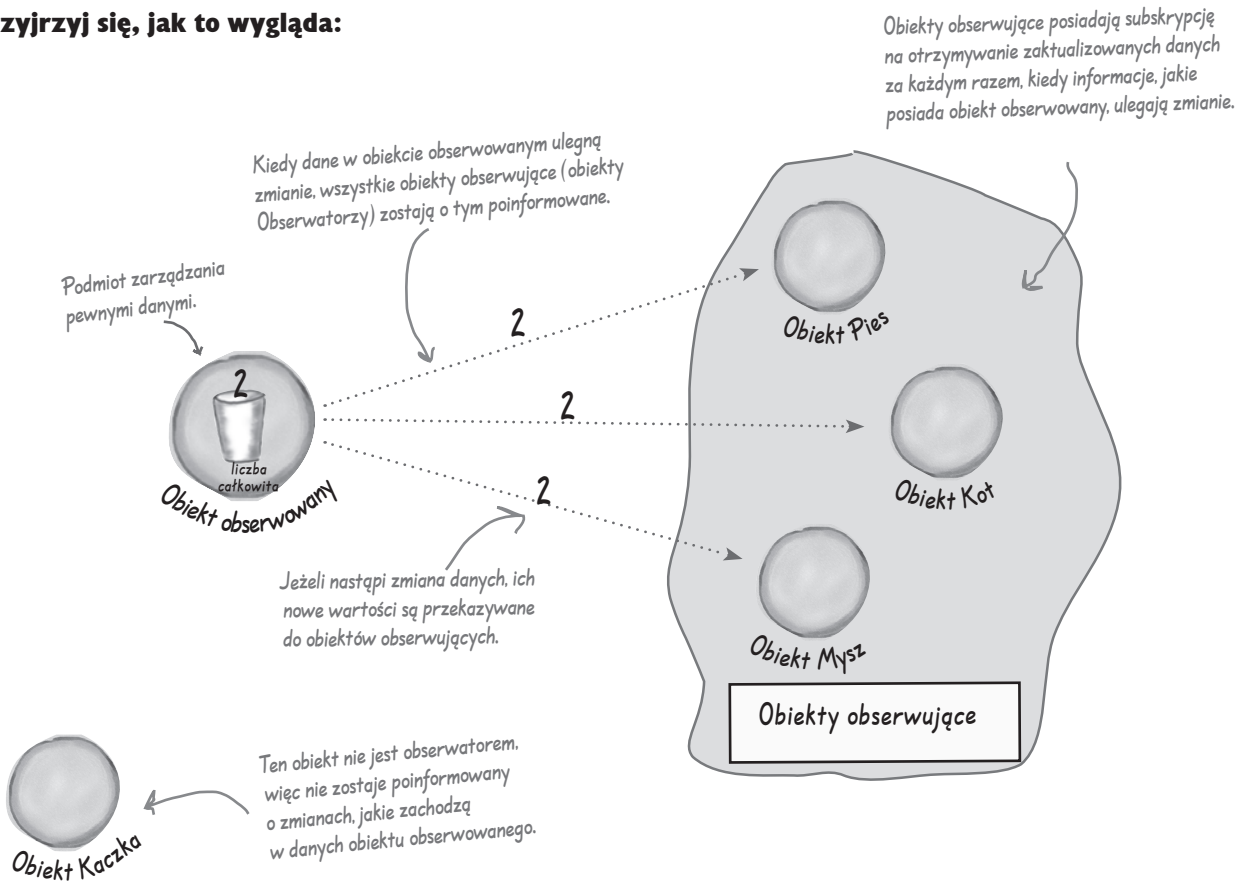


Wydawca + Prenumerator

= 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).

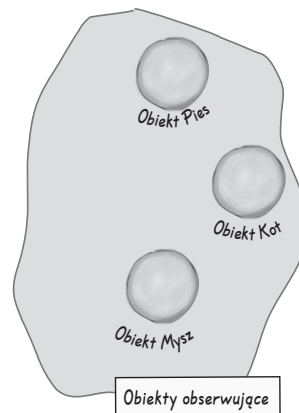
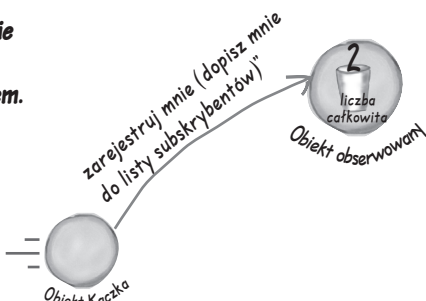
Przyjrzyj się, jak to wygląda:



Dzień z życia wzorca Obserwator

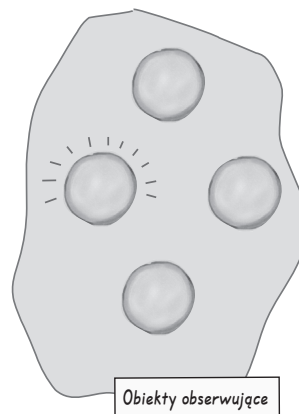
Obiekt Kaczka pojawia się i następnie informuje obiekt obserwowany, że chciałby zostać jego obserwatorem.

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 uznany za obserwatora.

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

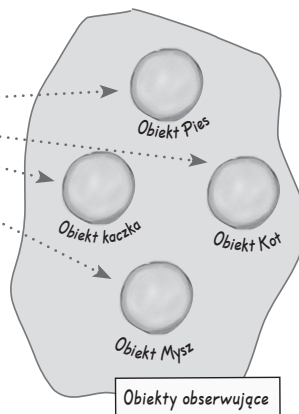


Obiekt obserwowany otrzymuje nowe dane!

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

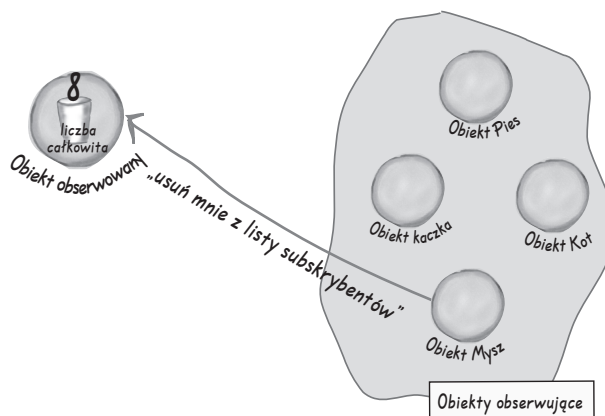


8
8
8
8



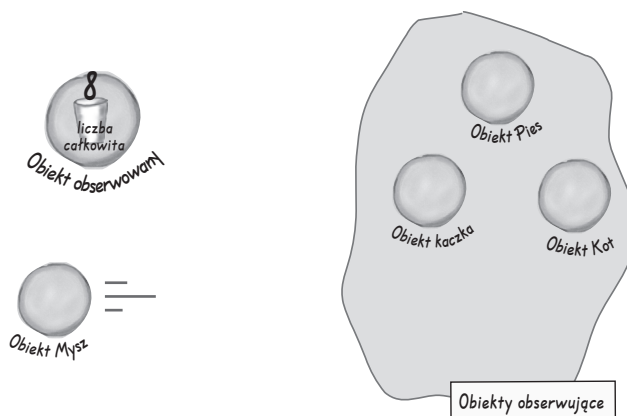
Obiekt Mysz prosi o usunięcie z listy obserwatorów.

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



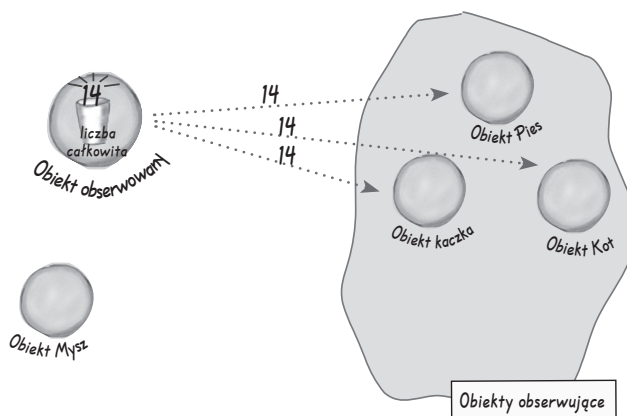
Myszy już tu nie ma!

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

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

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)

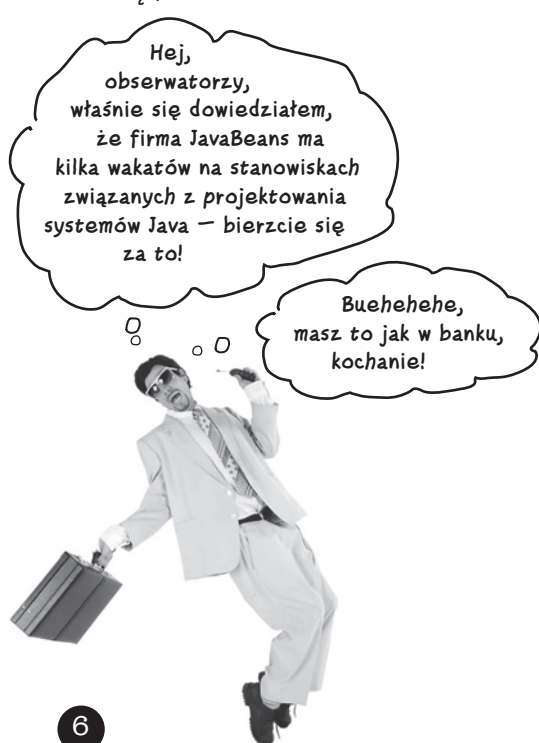
Umieszczę
Twoje nazwisko na
mojej liście. Dzięki temu
będziesz informowana
na bieżąco, razem
z pozostałymi
obserwatorami.



4

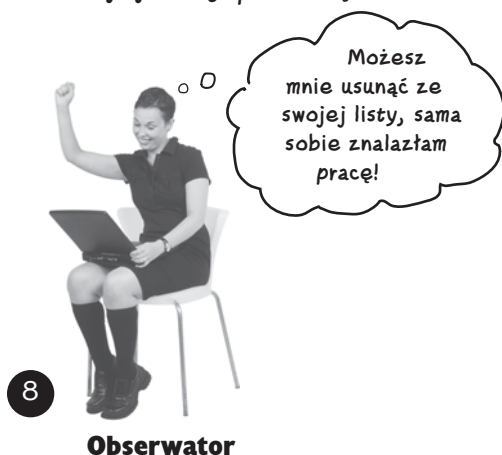
**Obiekt obserwowany
(Podmiot)**

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**

Joanna znajduje nowego pracodawcę.



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.



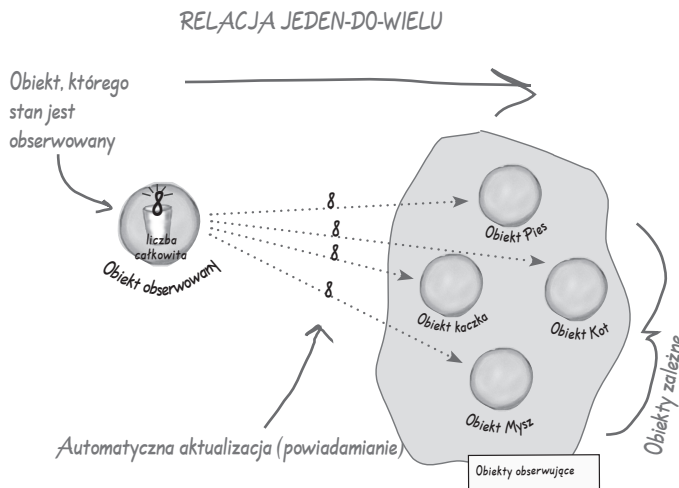
Definicja wzorca Obserwator

Jeżeli będziesz próbował w jakiś sposób zobrazować sobie wzorzec Obserwator, to zapewne szybko dojdiesz 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.

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 powiadomianie, 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.

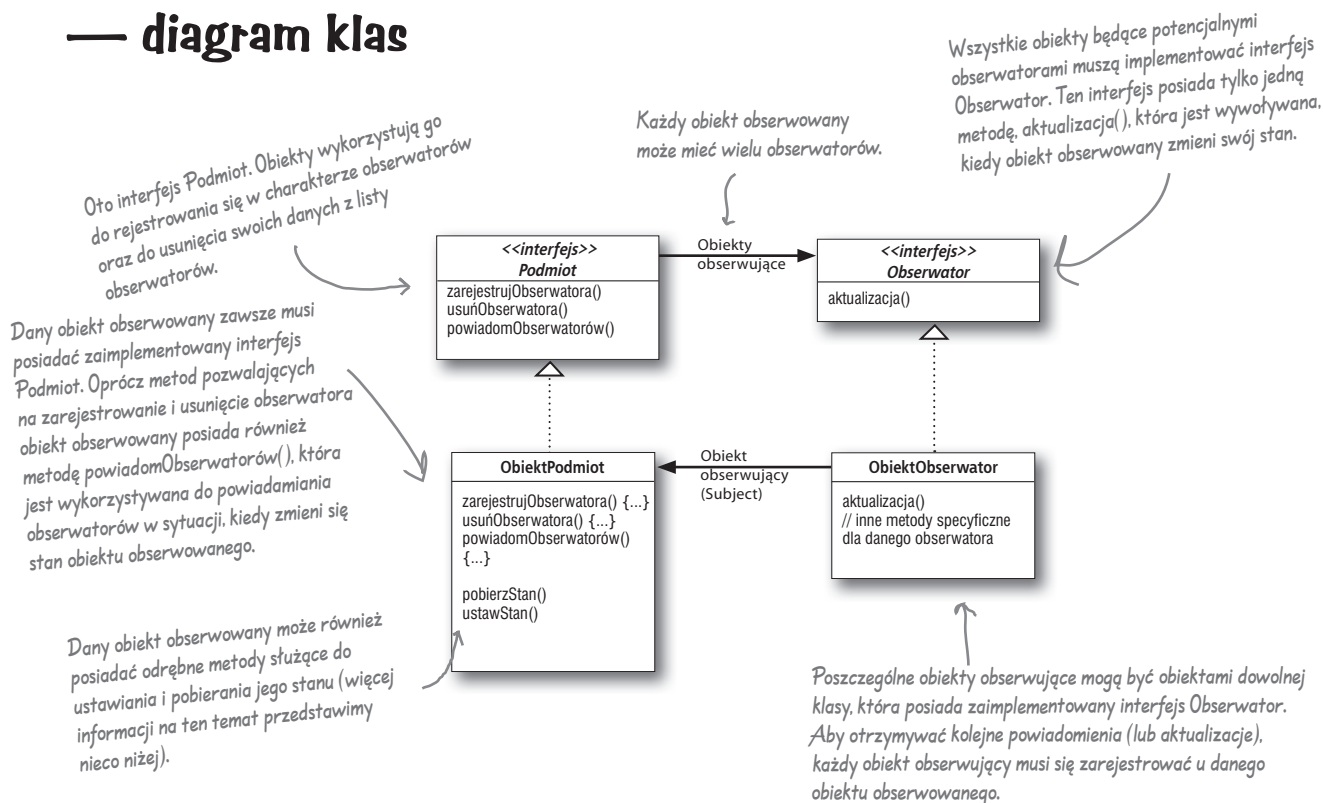
Rzućmy na to okiem...

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.

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 miałyby 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 wnিকamy, 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.



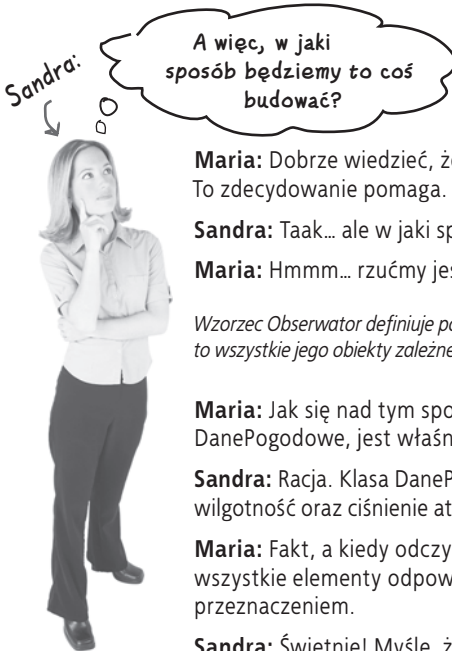
Zaostrz ołówek

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 boku obok już rozpoczęły dysputę na temat tego projektu.

Zastępowane w sąsiednim boksie...

Wracając do tematu projektu stacji meteorologicznej... wygląda na to, że Twoje koleżanki z boks 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: Hmm... 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.

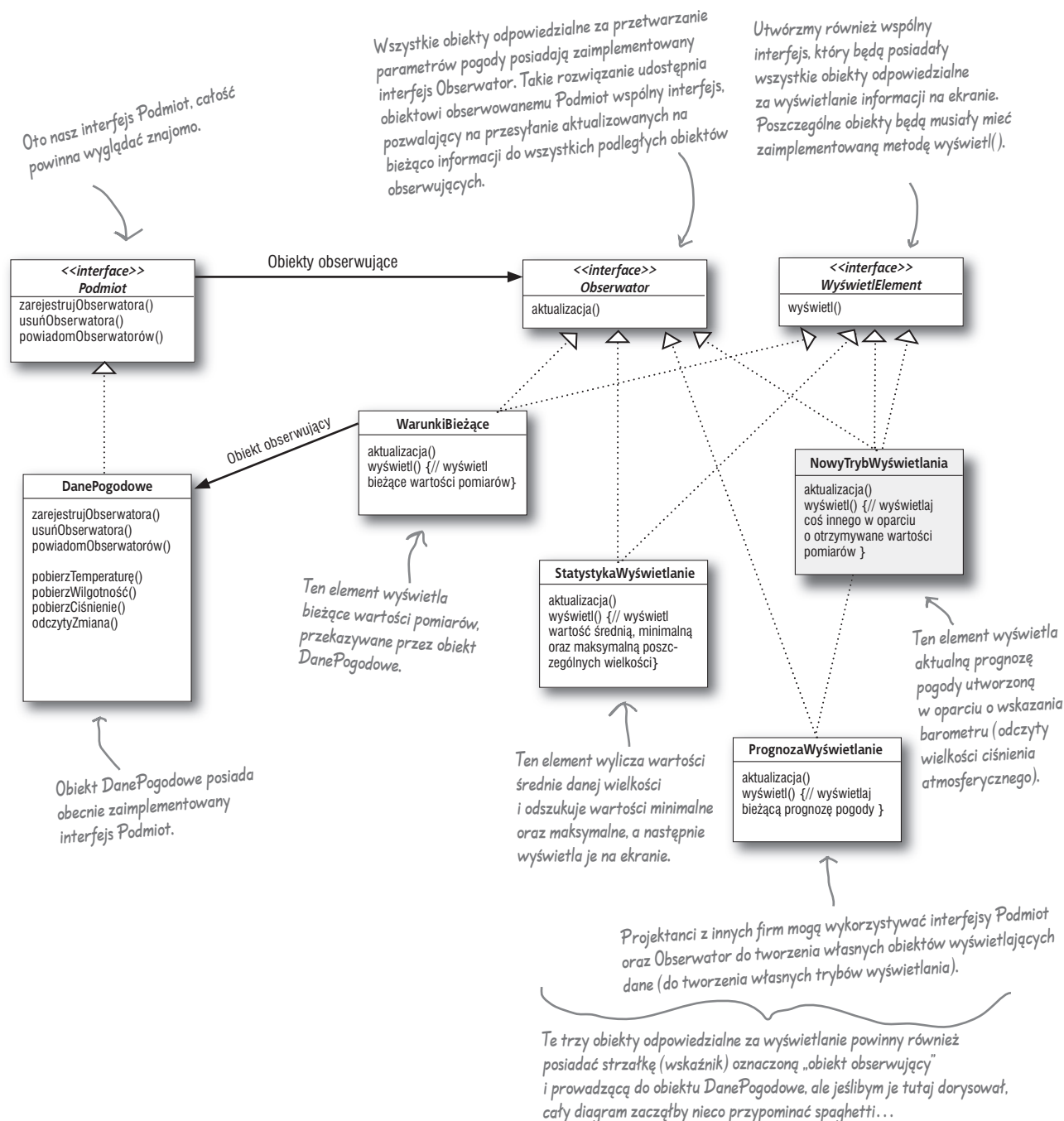
Sandra: Musimy jednak pamiętać, że każdy wyświetlany element może być zupełnie inny... Więc tak sobie myślę, gdzie jest miejsce na wprowadzenie wspólnego interfejsu? Pomimo, że poszczególne elementy mogą być zupełnie odmiennych typów, wszystkie powinny posiadać taki sam interfejs, tak aby obiekt DanePogodowe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

Rozpocniemy 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 szczerzy 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.

```
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.

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.



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

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...

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>

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.

Dodałiśmy 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 Pogodyńka 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?

U: 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?

U: 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

1 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);
    }
}
```

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

Najpierw utworzymy obiekt klasy DanePogodowe.

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

Symulacja kolejnych wartości odczytów z czujników pogody.

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

```
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!
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
Warunki bieżące: 25.5 stopni C oraz 90 % wilgotność
Temperatura średnia / Maksymalna / Minimalna = 26.6 / 27.7 / 25.5
Prognoza: Bieżąca pogoda się utrzyma
>
```



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 =

$$16.923 + 1.85212 * 10^{-1} * T + 5.37941 * RH - 1.00254 * 10^{-1} * T * RH + 9.41695 * 10^{-3} * T^2 + 7.28898 * 10^{-3} * RH^2 + 3.45372 * 10^{-4} * T^2 * RH - 8.14971 * 10^{-4} * T * RH^2 + 1.02102 * 10^{-5} * T^2 * RH^2 - 3.8646 * 10^{-5} * T^3 + 2.91583 * 10^{-5} * RH^3 + 1.42721 * 10^{-6} * T^3 * RH + 1.97483 * 10^{-7} * T * RH^3 - 2.18429 * 10^{-8} * T^3 * RH^2 + 8.43296 * 10^{-10} * T^2 * RH^3 - 4.81975 * 10^{-11} * T^3 * RH^3$$

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

Oczywiście, żartowaliśmy! Nie martw się, nie będziesz musiał pracowniczce 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>

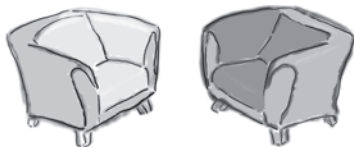
Jak działa ta magiczna formuła? Aby się tego dowiedzieć, powinieneś zajrzeć do książki *Meteorologia. Rusz głową!* (o ile kiedyś takowa się ukaże), zapytać kogoś pracującego w Instytucie Meteorologii i Gospodarki Wodnej lub po prostu zapytać Wielkiego Google'a (<http://www.google.pl>).

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 indeks 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 indeks 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 indeks ciepła ma wartość 28.69426
>
```

Pogawędki przy kominku



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. Wysłałam 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 mała 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 jesteście 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...

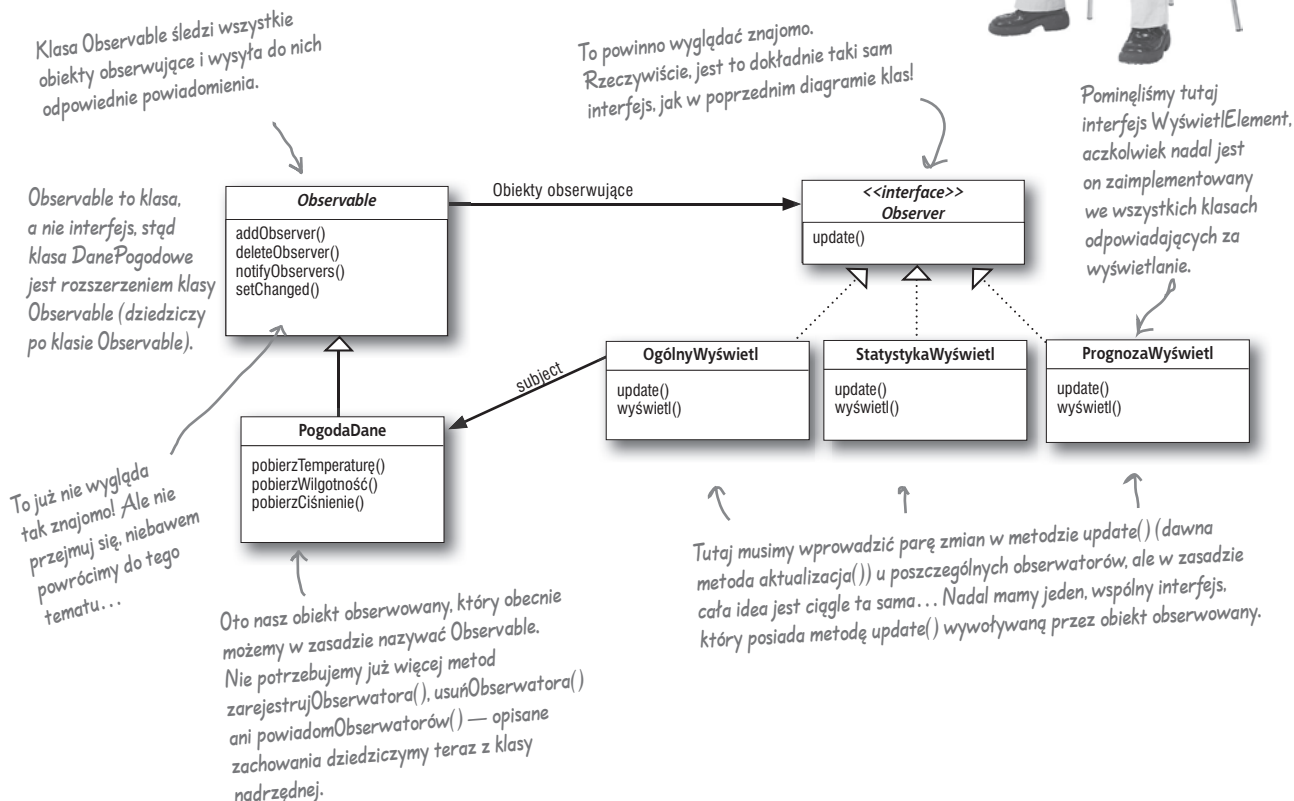
Niemożliwe, czyżbyśmy jednak doszliśmy do jakiegoś porozumienia? Czyli jednak zawsze jest jakaś nadzieja...?

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:

Dzięki wbudowanej w język Java obsłudze tego wzorca wszystko, co musisz zrobić, to tylko rozszerzyć klasę `Observable` i poinformować ją, kiedy ma powiadamiać swoich obserwatorów. Całą resztę wykona za Ciebie API.



Java — jak funkcjonuje wbudowany wzorec 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` wysłał 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:

- ❶ Musisz wywołać metodę `setChanged()`, aby zasygnalizować, że zmienił się stan obiektu obserwowanego.
- ❷ 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:

`update(Observable o, Object arg)`

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 wysłać 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.

Czekaj, zanim się za to weźmiemy, wyjaśnij, proszę, do czego będzie nam potrzebna metoda `setChanged()`? Poprzednio jej nie potrzebowaliśmy.



Fragment pseudokodu klasy `Observable`.

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()` 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()`.

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

```

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 odczytyZmiana() {
        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;
    }
}

```

Modyfikacja trybu „warunki bieżące”

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().



Ćwiczenia



Magnesiki z kodem

Jak widać, definicja klasy PrognozaWyświetl znajduje się w kompletnej rozsypance. 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;
```

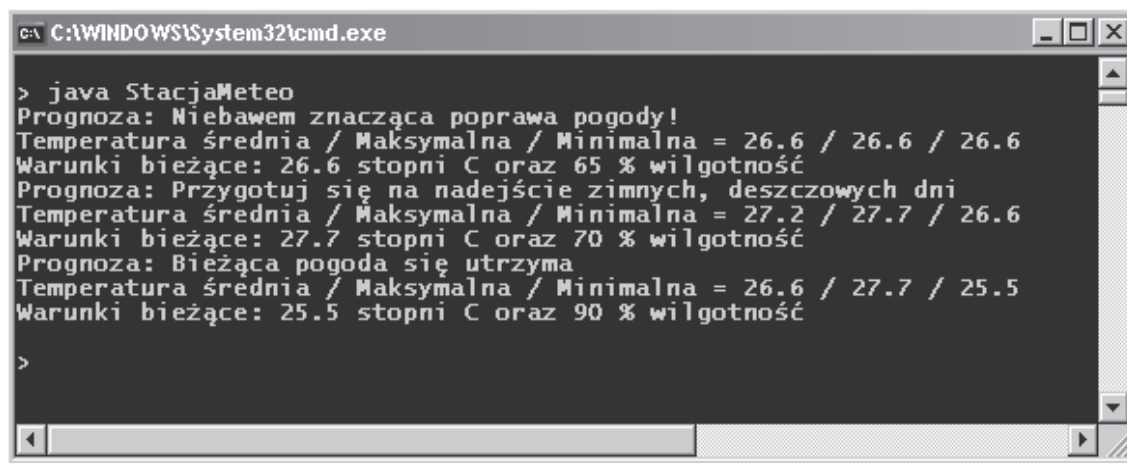
```
DanePogodowe danePogodowe =  
(DanePogodowe) observable;
```

```
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...



```
C:\WINDOWS\System32\cmd.exe

> java StacjaMeteo
Prognoza: Niebawem znacząca poprawa pogody!
Temperatura średnia / Maksymalna / Minimalna = 26.6 / 26.6 / 26.6
Warunki bieżące: 26.6 stopni C oraz 65 % wilgotność
Prognoza: Przygotuj się na nadejście zimnych, deszczowych dni
Temperatura średnia / Maksymalna / Minimalna = 27.2 / 27.7 / 26.6
Warunki bieżące: 27.7 stopni C oraz 70 % wilgotność
Prognoza: Bieżąca pogoda się utrzyma
Temperatura średnia / Maksymalna / Minimalna = 26.6 / 27.7 / 25.5
Warunki bieżące: 25.5 stopni C oraz 90 % wilgotność

>
```

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”.

Czy
definicja klasy `java.util.
Observable` nie stanowi przypadkiem
pogwałcenia jednej z naszych reguł
projektowania zorientowanego obiektowo,
głoszącej, że należy skupić się na tworzeniu
interfejsów, a nie na implementacji?



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 przyjrzyj 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 wyposażać 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 wzorec Obserwator

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

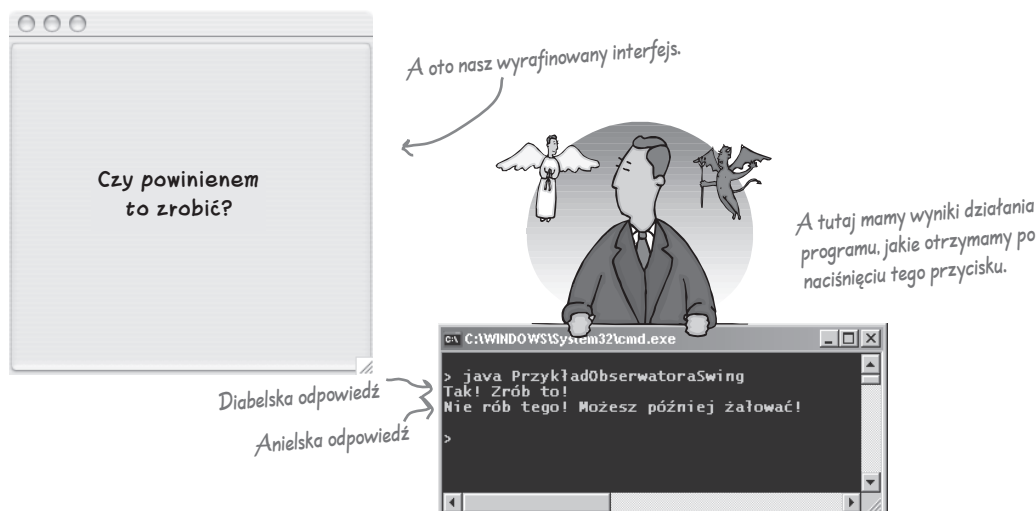
`java.util.Observer` oraz `java.util.Observable` nie są jedynymi miejscami w JDK (ang. Java Development Kit), w których możesz odnaleźć implementację wzorca `Observer`; zarówno komponenty `JavaBean`, jak i `Swing` mogą się pochwalić swoimi własnymi implementacjami tego wzorca. Na tym etapie wiedzy, na 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!");
        }
    }

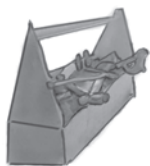
    class DiabełSłuchacz implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            System.out.println("Tak! Zrób to!");
        }
    }
}
```

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...

Podstawy programowania obiektowego

Reguły programowania obiektowego

Poddawaj hermetyzacji to, co się zmienia.

Przedkładaj kompozycję nad dziedziczenie.

Skoncentruj się na tworzeniu interfejsów, a nie implementacji.

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

Oto Twoja najnowsza reguła projektowania. Pamiętaj, że struktury wykorzystujące luźne związki są o wiele bardziej elastyczne i łatwiej adaptują się do zmian.

Wzorce programowania obiektowego

Strategie
ich her
Wzorce
algory
używa

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

Nowy wzorec projektowy, który pozwala na powiadamianie zbioru obiektów o zmianach stanu, wykorzystujący luźne związki pomiędzy poszczególnymi obiektami. Nie omawialiśmy jeszcze ostatniego typu wzorca Obserwator — poczekajmy z tym jednak do czasu, aż przedstawimy wzorec MVC!

CELNE

SPOSTRZEŻENIA



- Wzorec 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 baczność 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.
- Wzorec Obserwator jest wykorzystywany w wielu innych miejscach, jak np. `JavaBeans` czy `RMI`.



Reguły projektowania — wyzwanie

Napisz, w jaki sposób wzorec 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

Przedkładać kompozycję nad dziedziczenie.

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

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



Rozwiązania



Reguły projektowania — wyzwanie

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.



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.

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!

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.

Wzorec 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()
    }
}

```




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
 kolejkovanie 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
 metapoznajwanie, 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
- wzorec 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
 zaporą sieciową, 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 międzyobiektywne, 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
 obiektywne, 47
 zmienne elementy aplikacji, 142

Ż

żądania
 HTTP, 567, 569, 575
 rejestracji, 255

PROGRAM PARTNERSKI

— GRUPY HELION —



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

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

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

GRUPA
Helion 

Przekonaj się, że nowoczesne metody nauczania mogą zmienić również sposób poznawania nowoczesnych technik programistycznych.

Rusz głową! Wzorce projektowe

Otwórz swój umysł. Poznaj wszystko, co jest związane z wzorcami projektowymi, w sposób gwarantujący szybkie i skuteczne opanowanie zasad ich stosowania. Zapomnij o listingach liczących tysiące wierszy, długich i nużących opisach teoretycznych oraz rozbudowanych schematach zależności. Wzorce projektowe to gotowe opisy rozwiązań najczęstszych problemów napotykanych przy tworzeniu oprogramowania. Aby je prawidłowo stosować, należy poznać założenia, na podstawie których zostały stworzone, oraz nauczyć się implementować je we właściwy sposób.

Naprzód, głowo!

Nikt nie potrafił Ci tego wytłumaczyć? Wydaje Ci się, że to problem nie na Twoją głowę? Nie potrzebujesz elektrowstrząsów, żeby pobudzić swój mózg do aktywnego działania. Tylko żadnych gwałtownych ruchów! Usiądź wygodnie, otwórz książkę, dopiero teraz się zacznij. Na początek – rusz głową!

Precz z nudnymi wykładami i zakuwaniem bez zrozumienia!

Nauka to znacznie więcej niż tylko czytanie suchego tekstu. Twój mózg jest niczym głodny rekin, cały czas pracy naprzód w poszukiwaniu nowej, apetycznej przekąski.

Jak karmimy Twój wygłodniały umysł?

Używamy rysunków, bo obraz wart jest 1024 słów. Stosujemy powtórzenia, by zakodować na stałe dane w Twojej chłonnej głowie. Oddziałujemy na emocje, jesteśmy nieprzewidywalni, zaskakujący i zabawni. Stawiamy przed Tobą wyzwania i zadajemy pytania, które angażują Cię w proces studiowania przedstawianych zagadnień. Cały czas pobudzamy Twój umysł do aktywnego działania, zmuszamy go do posłuszeństwa... a za ciężką pracę nagrodzimy go smakowitym ciasteczkiem w postaci wiedzy – wisienka gratis!

Rozgrzyź to sam!

▼
Cele stosowania wzorców projektowych

▼
Założenia, na których opierają się wzorce projektowe

▼
Najważniejsze i najczęściej wykorzystywane wzorce projektowe

▼
Przechowywanie i prezentacja danych

▼
Mechanizm RMI

▼
Wzorzec MVC

▼
Implementacja wzorców projektowych w aplikacjach

sięgnij po WIĘCEJ



KOD KORZYŚCI

ISBN 978-83-283-3355-0



cena: 89,00 zł

Helion

księgarnia internetowa



<http://helion.pl>

zamówienia telefoniczne



0 801 339900



0 601 339900

Informatyka w najlepszym wydaniu

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
<http://helion.pl>

Sprawdź najnowsze promocje:
• <http://helion.pl/promocje>
Książki najchętniej czytane:
• <http://helion.pl/bestsellery>
Zamów informacje o nowościach:
• <http://helion.pl/nowości>

