helion.pl
KSIĘGARNIA INTERNETOWA WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

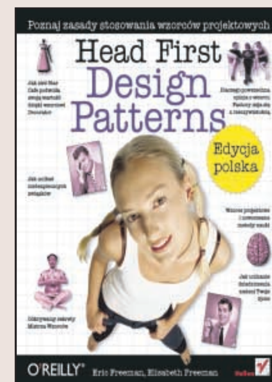
KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

Head First Design Patterns. Edycja polska



Autorzy: Eric Freeman, Elisabeth Freeman
Tłumaczenie: Paweł Koronkiewicz (wprowadzenie, rozdz. 1–8), Grzegorz Kowalczyk (rozdz. 9–14)
ISBN: 83-7361-792-2
Tytuł oryginału: Head First Design Patterns
Format: 200×230, stron: 656

Poznaj w niekonwencjonalny sposób zasady stosowania wzorców projektowych

- Dowiedz się, czym są wzorce projektowe
- Poznaj typy wzorców projektowych
- Zastosuj wzorce projektowe w praktyce
- Naucz się projektować aplikacje w oparciu o 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 linii, długich i nużących opisach teoretycznych oraz rozbudowanych schematach zależności. Czytając książkę „Head First Design Patterns. Edycja polska", poznasz wzorce projektowe w inny sposób. Wzorce projektowe to gotowe opisy rozwiązań najczęściej spotykanych zagadnień związanych z tworzeniem oprogramowania. Aby je prawidłowo stosować, należy poznać założenia, na podstawie których zostały stworzone, oraz nauczyć się implementować je we właściwy sposób.

Dzięki książce „Head First Design Pattern. Edycja polska" wszystkie pojęcia związane ze wzorcami projektowymi przestaną być dla Ciebie wiedzą tajemną. Autorzy książki, wykorzystując najnowsze elementy teorii uczenia, przedstawią Ci wszystkie zagadnienia niezbędne do rozpoczęcia projektowania i tworzenia aplikacji w oparciu o wzorce projektowe. Poznasz najczęściej stosowane wzorce projektowe, metody ich implementacji i zadania, do jakich są przeznaczone. Jednak, co najważniejsze, nauczysz się stosować tę wiedzę w praktyce.

- 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

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

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści (skrócony)

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

Spis treści (na serio)

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

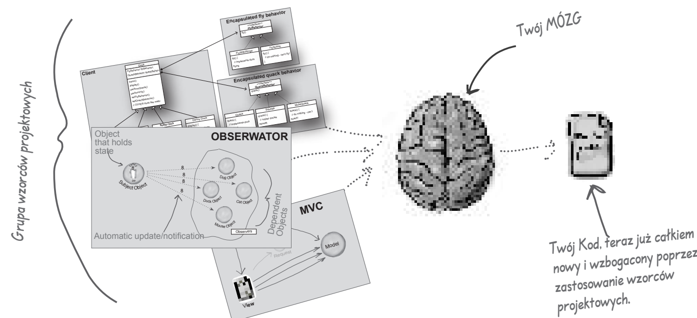
1

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 taką wyprawę. Zanim dobrniemy do końca rozdziału, rzucimy okiem na sposoby wykorzystywania wzorców projektowych i przedstawimy ich zalety, poznamy kilka podstawowych zasad projektowania zorientowanego obiektowo, a także omówimy sposób działania przykładowego wzorca. Najlepszą metodą zastosowania wzorca jest *załadowanie go bezpośrednio do Twojego mózgu*, a następnie *zlokalizowanie* obszarów w obrębie projektowanych rozwiązań oraz istniejących aplikacji, w których możesz je *zastosować*. Pracując z wzorcami projektowymi, zamiast wielokrotnego wykorzystywania tych samych fragmentów kodu, wielokrotnie wykorzystujesz swoje *doświadczenia*.

Prosta aplikacji 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

Pamiętaj, opanowanie takich zagadnień, jak abstrakcyjność, dziedziczenie i polimorfizm, nie zrobi jeszcze z Ciebie dobrego projektanta systemów zorientowanych obiektowo. Prawdziwy guru zawsze myśli o stworzeniu elastycznego projektu, który będzie łatwy do serwisowania i będzie sobie w stanie poradzić ze zmieniającymi się warunkami.



2 Wzorzec Obserwator

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

...iczenie

Reguły programowania obiektowego

Reguły programowania obiektowego

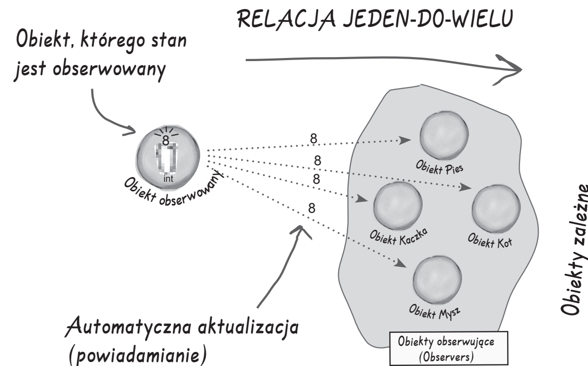
Poddawaj hermetyzacji to, co się zmienia.

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.

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

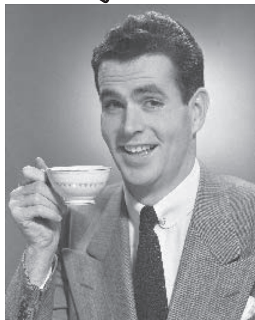


Wzorzec Dekorator

3 Dekorowanie zachowania obiektów

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

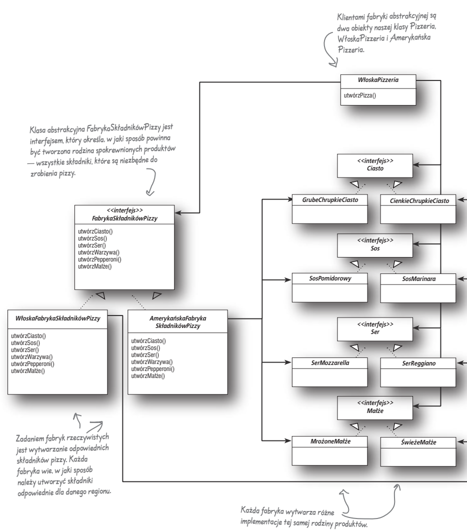
Witamy w „Star Caf ”	110
Reguła otwarte-zamknięte	116
Spotkanie z wzorcem Decorator	118
Konstruowanie zamówienia przy użyciu Dekoratorów	119
Definicja wzorca Decorator	121
Dekorujemy nasze Napoje	122
Tworzymy kod aplikacji „Star Caf ”	125
Dekoratory w świecie rzeczywistym: obsługa wejścia-wyjścia w języku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



Wzorzec Fabryka

4 Pizzeria zorientowana obiektowo

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



Kiedy widzisz „nowy” obiekt, myśl o nim jako o „konkretnym”	140
Pizza w Obiektowie	142
Hermetyzacja procesu tworzenia obiektów	144
Budujemy prostą fabrykę pizzy	145
Tworzymy definicję „wzorca” Simple Factory	147
Nowa struktura Pizzerii	150
Zezwalamy klasom podrzędnym na podejmowanie decyzji	151
Tworzymy Pizzerię	153
Deklarowanie metody typu Factory (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

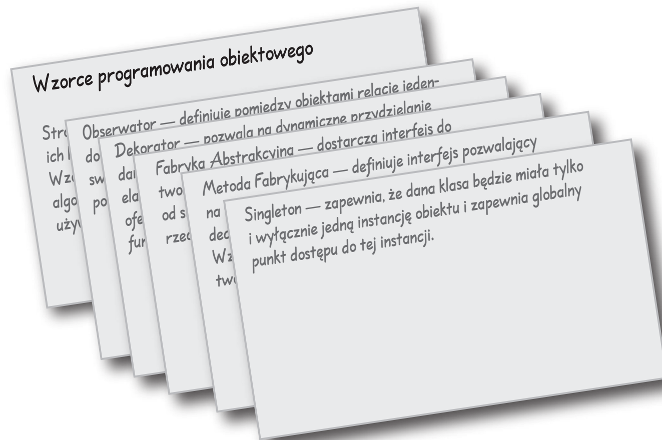
Wzorzec Singleton

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 jakby mogło się wydawać.

Hershey, PA

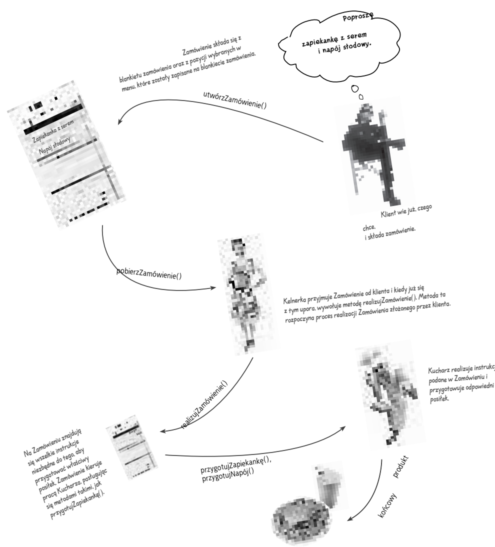
Jeden i tylko jeden 198
Mały Singleton 199
Analiza klasycznej implementacji wzorca Singleton 201
Wyznania obiektu Singleton 202
Fabryka czekolady 203
Definicja wzorca Singleton 205
Uuups, mamy problem… 206
Zostań wirtualną maszyną Java 207
Jak sobie radzić z wielowątkowością? 208
Wzorzec Singleton —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



Wzorzec Polecenie

6 Hermetyzacja wywołań

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



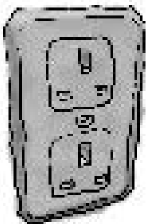
Automatyka w domu i zagrodzie 218
Mamy nową zabawkę! Sprawdzamy, jak działa SuperPilot… 219
Co zawiera otrzymany dysk CD-R 220
A w międzyczasie w naszym barze szybkiej obsługi… 223
Przyjrzyjmy się nieco dokładniej wzajemnym interakcjom… 224
Zadania i zakresy odpowiedzialności 225
Od Baru do wzorca Polecenie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

Wzorce Adapter oraz Fasada

7 Zdolność do adaptacji

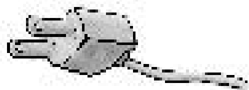
W niniejszym rozdziale mamy zamiar dokonać paru niesamowitych wyczynów z dziedziny rzeczy niemożliwych, takich jak na przykład włożenie kwadratowego kołka do okrągłego otworu. Brzmi nierealnie? Nie wtedy, kiedy mamy pod ręką odpowiednie wzorce projektowe. Pamiętasz wzorzec Dekorator? Podczas pracy z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

Europejski standard ściennego gniazda elektrycznego

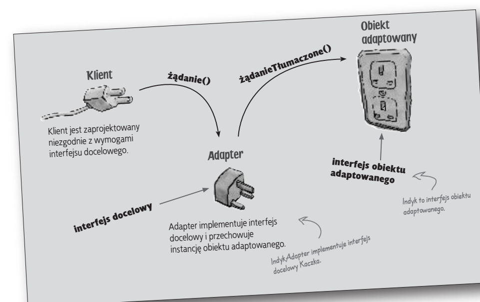


Adapter

Standardowa wtyczka zasilająca



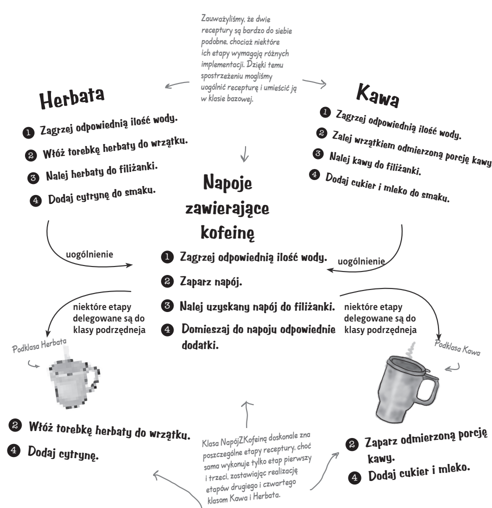
Adaptery są wśród nas	260
Adaptery zorientowane obiektowo	261
Wzorzec Adapter bez tajemnic	265
Definicja wzorca Adapter	267
Adaptery obiektów i klas	268
Temat dzisiejszej wieczornej pogawędki: Adapter obiektów i Adapter klas	271
Adaptery w świecie rzeczywistym	272
Adaptujemy interfejs Enumeration do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



Wzorzec Metoda Szablonowa

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.

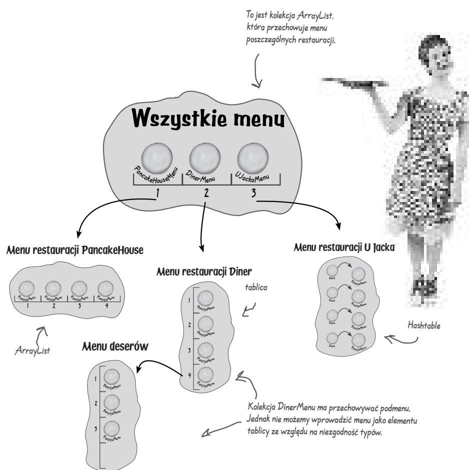


Tworzymy klasy reprezentujące kawę i herbatę (w języku Java)	299
Kawa i herbata, czyli klasy abstrakcyjne	302
Ciągniemy nasz projekt o krok dalej…	303
Wydobywanie metody recepturaParzenia()	304
Czego już dokonaliśmy?	307
Spotkanie z wzorcem Metoda Szablonowa	308
Zróbmy sobie herbatę…	309
Co nam daje 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

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 też pisać o odpowiedzialności obiektów.



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

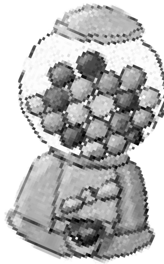
Wzorzec Stan

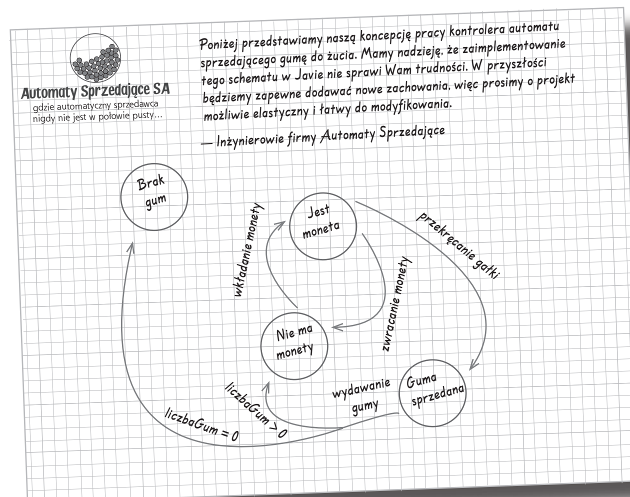
10

Stan obiektu

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

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



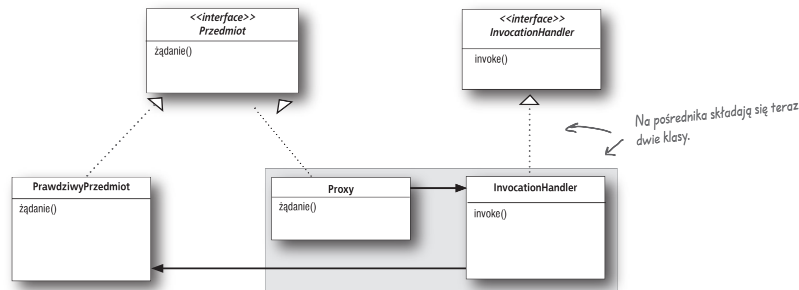


Wzorzec Proxy

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 i zarządzanie dostępem. Jak się przekonamy, istnieje *bardzo wiele* schematów takiego pośrednictwa. Obiekty Proxy mogą przekazywać wywoływanie metody obiektowi w innym węźle internetu; bywa też, że zastępują wyjątkowo leniwe obiekty.

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

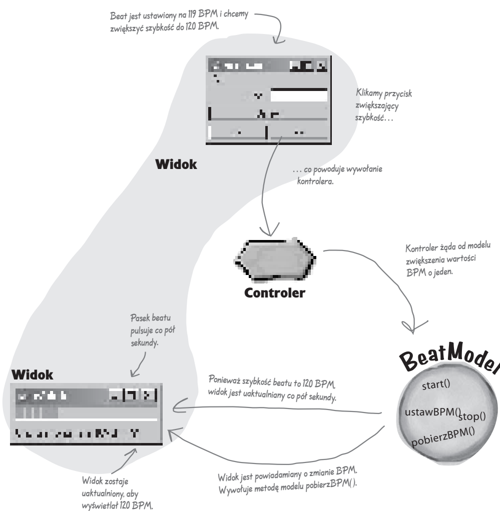


Wzorce złożone

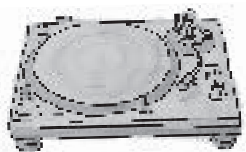
12

Łączenie wzorców

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



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

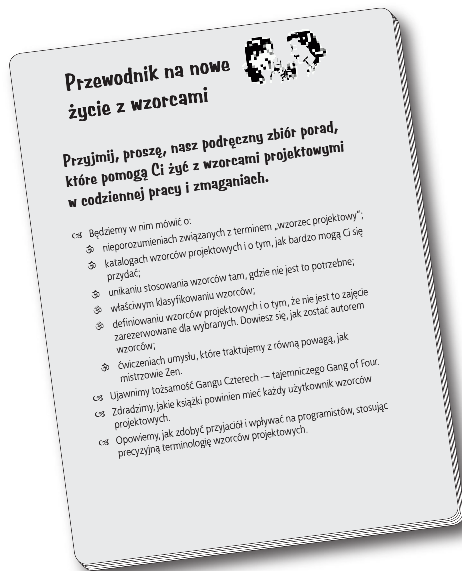


Nowe życie z wzorcami

13

Wzorce projektowe w praktyce

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

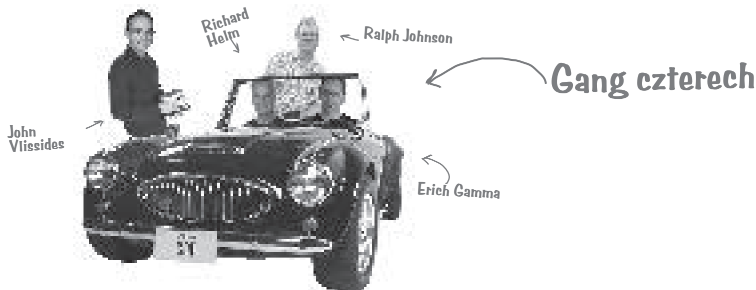


Przewodnik na nowe życie z wzorcami

Przyjmij, proszę, nasz podręczny zbiór porad, które pomogą Ci żyć z wzorcami projektowymi w codziennej pracy i zmaganiach.

- Będziemy w nim mówić o:
  - nieporozumieniach związanych z terminem „wzorzec projektowy";
  - katalogach wzorców projektowych i o tym, jak bardzo mogą Ci się przydać;
  - unikaniu stosowania wzorców tam, gdzie nie jest to potrzebne;
  - właściwym klasyfikowaniu wzorców;
  - definiowaniu wzorców projektowych i o tym, że nie jest to zajęcie zarezerwowane dla wybranych. Dowiesz się, jak zostać autorem wzorców;
  - ćwiczeniach umysłu, które traktujemy z równą powagą, jak mistrzowie Zen.
- Ujawnimy tożsamość Gangu Czterech — tajemniczego Gang of Four.
- Zdradzimy, jakie książki powinien mieć każdy użytkownik wzorców projektowych.
- Opowiemy, jak zdobyć przyjaciół i wpływać na programistów, stosując precyzyjną terminologię wzorców projektowych.

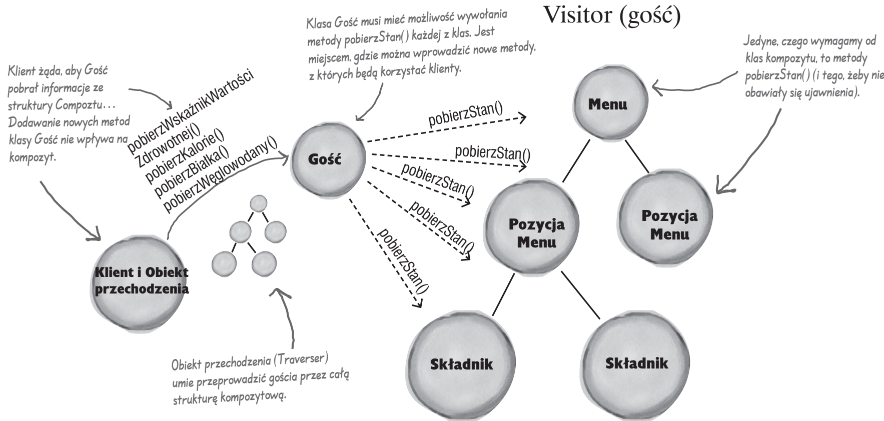
Przewodnik na nowe życie z wzorcami	578
Definicja wzorca projektowego	597
Drugie spojrzenie na definicję wzorca	599
Niech moc będzie z Tobą	600
Katalog wzorców	601
Jak tworzyć wzorce	604
Zostać autorem wzorców projektowych	605
Porządkowanie wzorców projektowych	607
Myślenie wzorcami	612
Głowa pełna wzorców	615
Nie zapominaj o potędze jednolitego słownictwa	617
Pięć podstawowych sposobów promowania Twojego słownictwa	618
Gang Czterech w Obiektowie	619
Podróż dopiero się zaczyna…	620
Inne źródła informacji o wzorcach	621
ZOO pełne wzorców	622
Walka ze złem przy użyciu antywzorców	624
Twoja skrzynka narzędziowa	626
Opuszczamy Obiektowo…	627



14 Dodatek — inne wzorce

Nie wszyscy mogą być sławni. Przez ostatnie dziesięć lat wiele się w świecie wzorców zmieniło. Od czasu pierwszego wydania książki *Design Patterns: Elements of Reusable Object-Oriented Software (Wzorce projektowe)* programiści wykorzystali opisane w nich 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



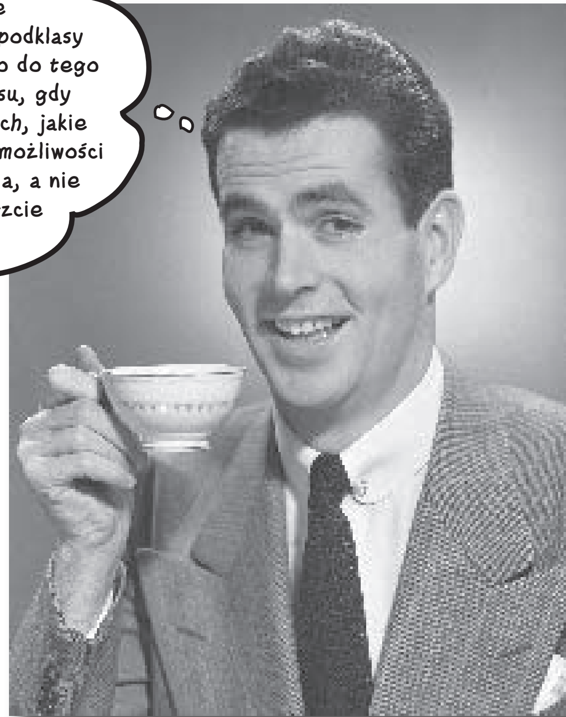
S Skorowidz

Skorowidz 649

Rozdział 3. Wzorzec Dekorator

Dekorowanie zachowania obiektów

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 możliwość rozszerzania możliwości aplikacji na poziomie działania, a nie kompilacji. A teraz — spójrzcie tylko na mnie!



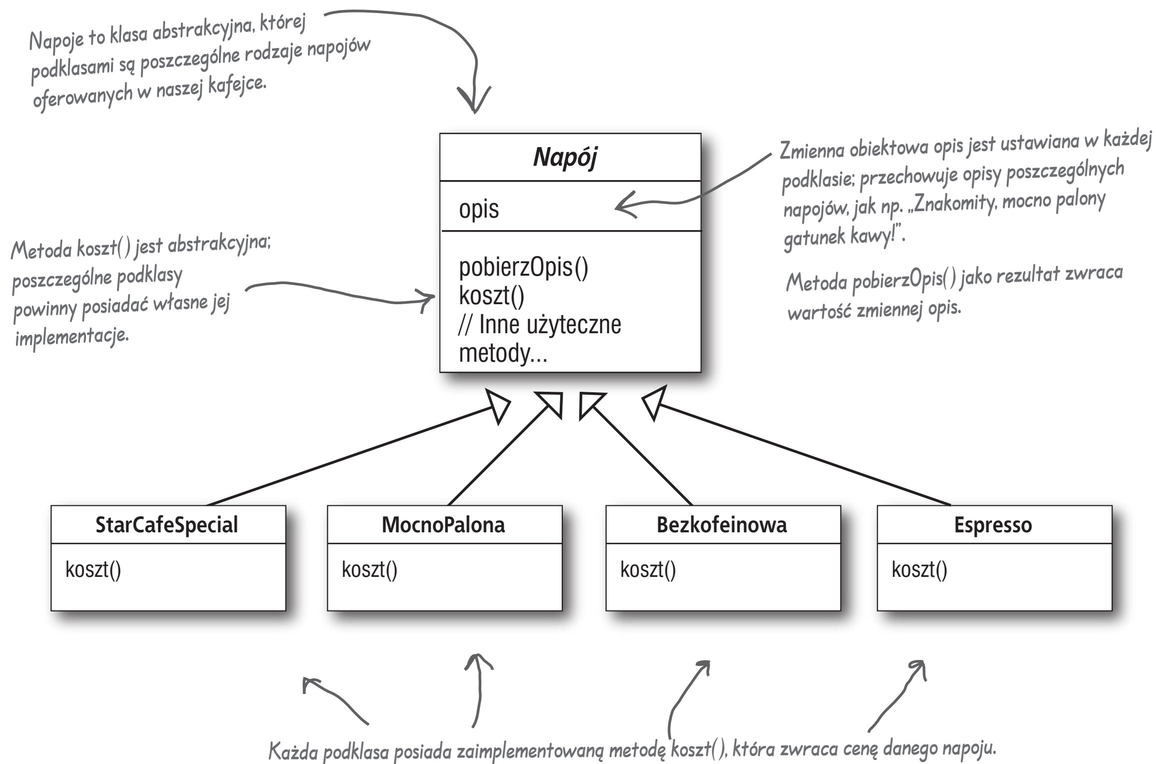
W zasadzie niniejszy rozdział możemy równie dobrze zatytułować „Otwieranie oczu programistom z nadmiernymi skłonnościami do nadużywania dziedziczenia”. W tym rozdziale spróbujemy krytycznie przyjrzeć się zwyczajowym skłonnościom do nadużywania mechanizmu dziedziczenia oraz nauczymy Cię sposobów dekorowania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

Witamy w „Star Café”

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

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

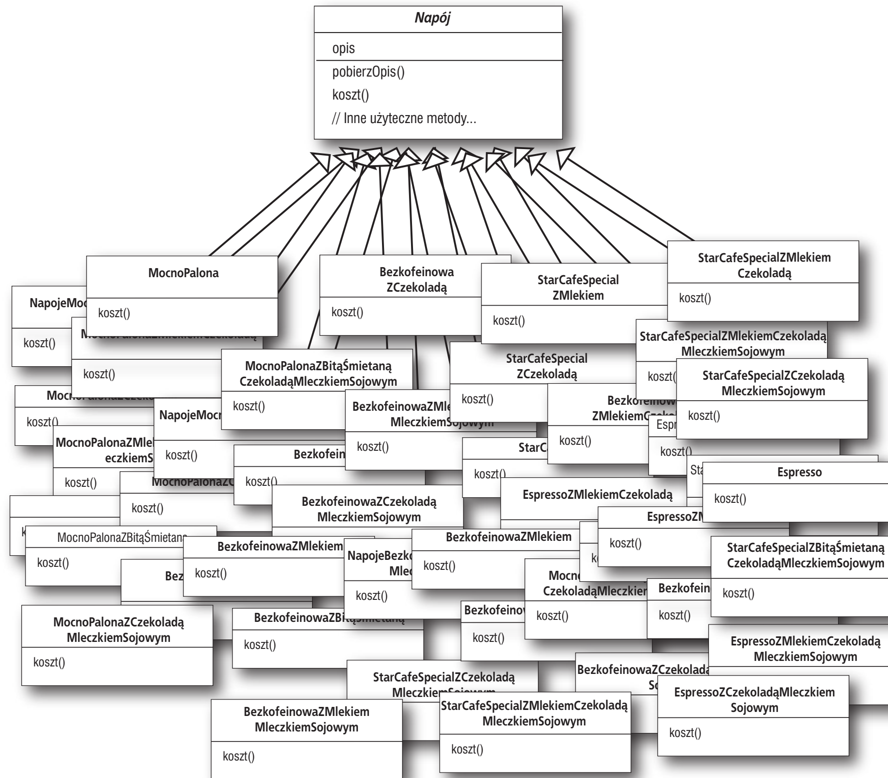
Kiedy pierwsza kawiarenka „Star Café” otwarła swe podwoje dla klientów, diagram klas ich systemu zamówień wyglądał mniej więcej tak:



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

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

Ich pierwsza próba wyglądała mniej więcej tak...



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

Uuuaaa! To się dopiero nazywa „eksplozja klas"!

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

Wysil szare komórki

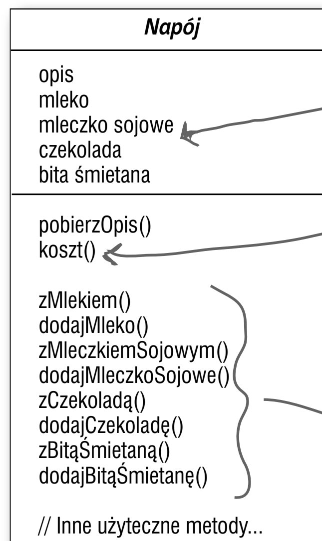
Chyba dla wszystkich jest oczywiste, że projektanci ze „Star Café" sami sobie narobili kłopotów, tworząc strukturę aplikacji, która z punktu widzenia serwisowania, modyfikacji i rozbudowy jest po prostu koszmarnym snem szalonego programisty. Wyobraź sobie tylko, co oni powinni zrobić, jeśli cena mleka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

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

Wskazówka: dwie z tych reguł zostały tutaj kompletnie podeptane!

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

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

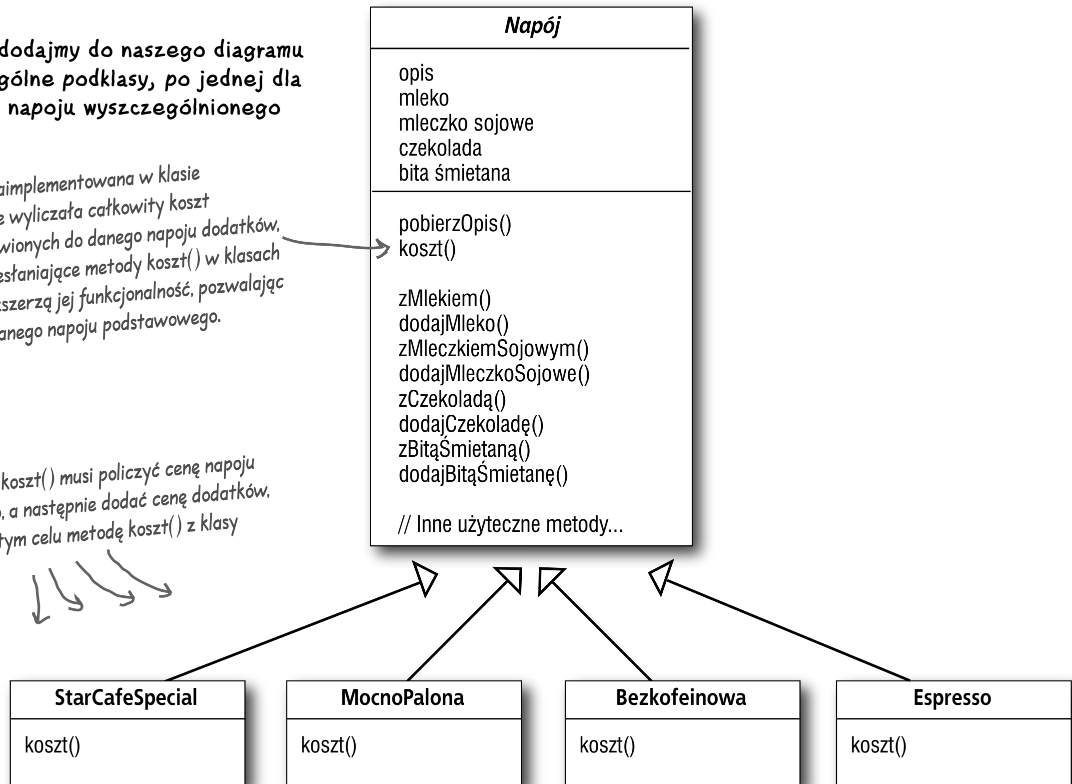


typu boolean dla każdego z dodatków.

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

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

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



Zaostrz ołówek

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

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



  }
}
```

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


  }
}
```

Zaostrz ołówek

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

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

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

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

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

Twoja kolej:

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

Mistrz i uczeń...

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

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

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

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

Mistrz: *Proszę, kontynuuj…*

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

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

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

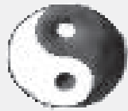
Mistrz: *A czego nauczyłeś się o wpływie, jaki zastosowanie kompozycji wywiera na możliwość zarządzania i modyfikacji kodu programu?*

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

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

Reguła otwarte-zamknięte

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



Reguła projektowania

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

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

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

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

Nie ma niemądrych pytań

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

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

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

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

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

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

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

P: Będę wiedział, które obszary zmian są najważniejsze?

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

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

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

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

Spotkanie z wzorcem Dekorator

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

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

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

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

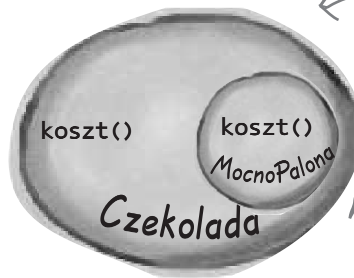
Konstruowanie zamówienia przy użyciu Dekoratorów

❶ **Rozpoczynamy od naszego obiektu MocnoPalona.**

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

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

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



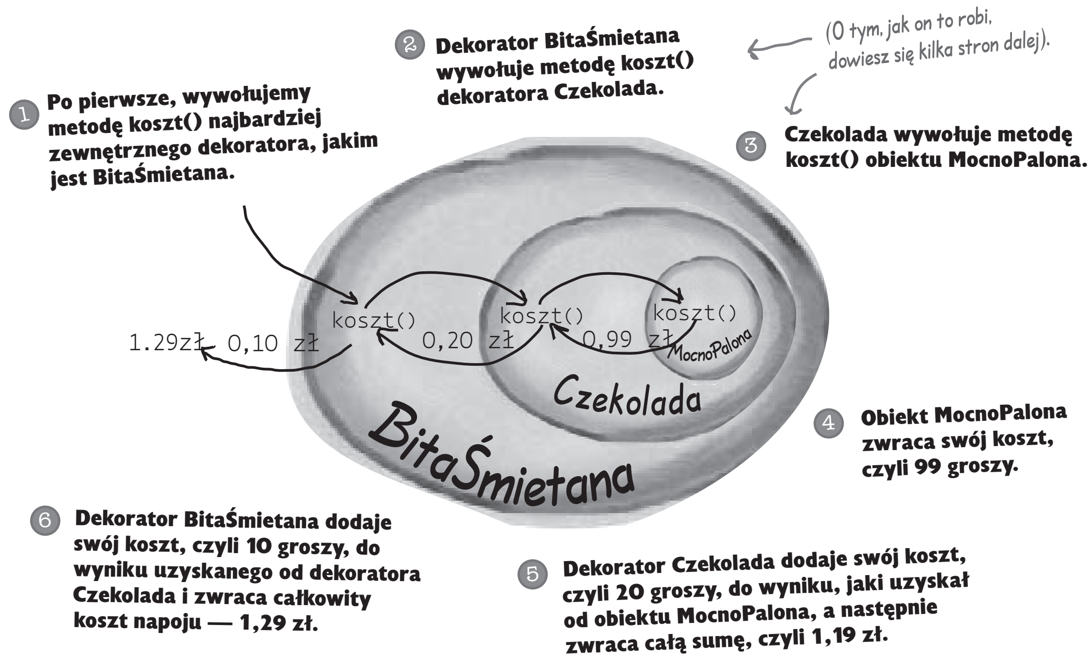
Jak widać, Czekolada również posiada swoją metodę koszt() i dzięki własnościom polimorfizmu możemy traktować każdy napój, do którego dodamy Czekoladę, jako kolejny napój (ponieważ Czekolada jest podtypem klasy Napoje).

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

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

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

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



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

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

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

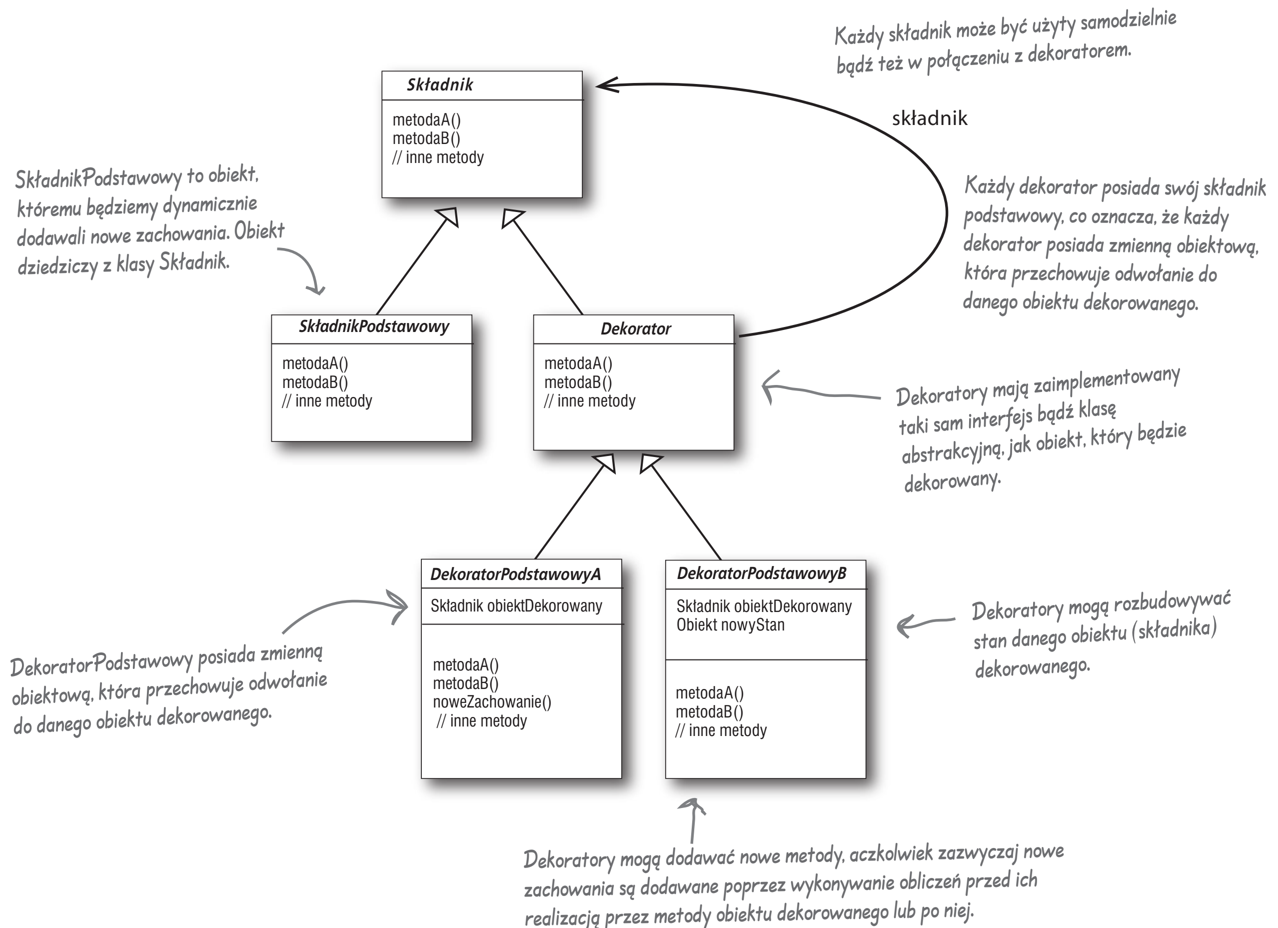
A teraz zobaczymy, jak to naprawdę działa — przyjrzymy się definicji wzorca Dekorator, a także napiszemy nieco kodu.

Definicja wzorca Dekorator

Przede wszystkim przyjrzyjmy się definicji wzorca Dekorator:

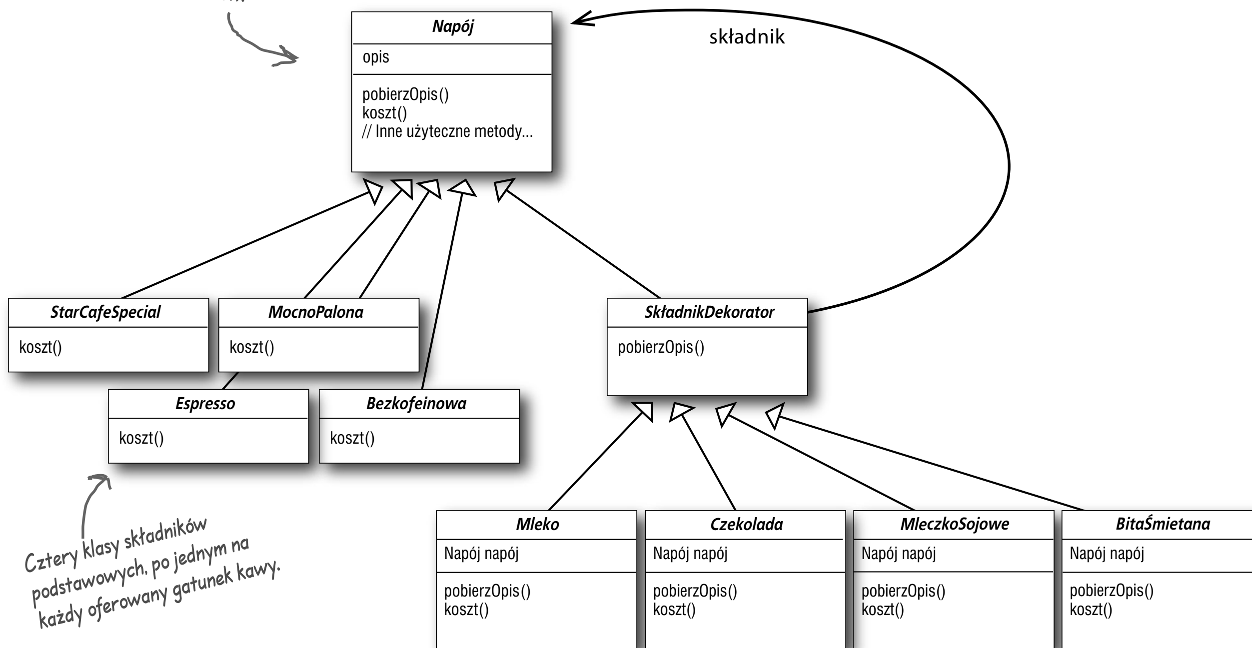
> **Wzorzec Dekorator** pozwala na dynamiczne przydzielanie danemu obiektowi nowych zachowań. Dekoratory dają elastyczność podobną do tej, jaką daje dziedziczenie, oferując jednak w zamian znacznie rozszerzoną funkcjonalność.

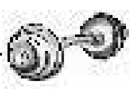
Powyższa definicja opisuje nam jedynie *przeznaczenie* wzorca Dekorator, ale nie daje nam niemal żadnych wskazówek, w jaki sposób moglibyśmy *zaimplementować* ten wzorzec w naszej aplikacji. Rzućmy okiem na diagram klas, który nieco rozjaśni nam całą sytuację (na następnej stronie zobaczymy tę samą strukturę,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



Dekorujemy nasze Napoje

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



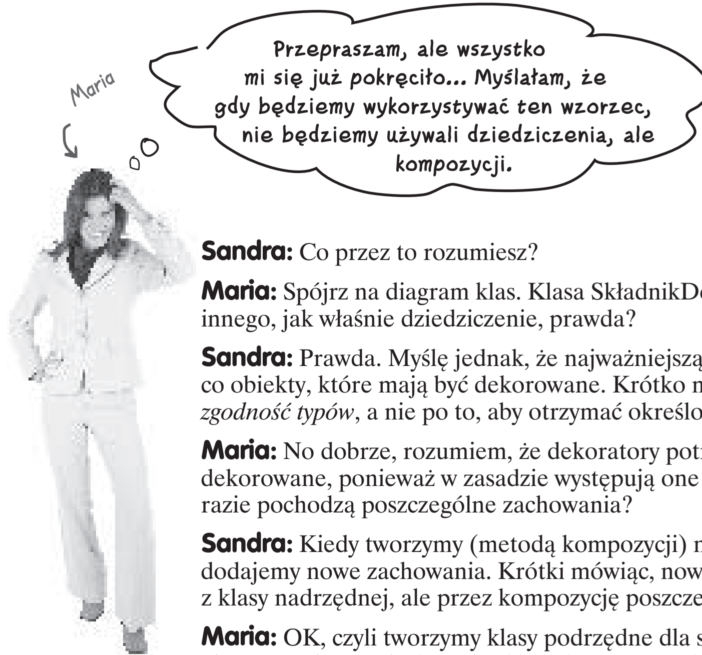


Wysil szare komórki

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

Zasłyszane w sąsiednim boksie...

Zamieszanie z Dziedziczeniem i Kompozycją



Sandra: Co przez to rozumiesz?

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

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

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

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

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

Sandra: Zgadza się.

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

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

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

Sandra: Dokładnie tak.

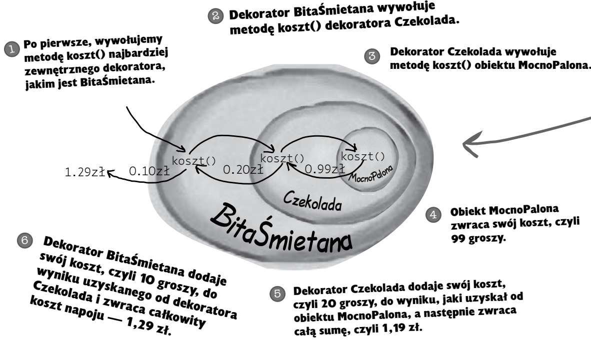
Maria: Mam jeszcze tylko jedno pytanie. Jeżeli wszystko, czego potrzebujemy od dziedziczenia, to tylko prawidłowy typ składnika, dlaczego nie użyć zamiast abstrakcyjnej klasy Napój po prostu interfejsu?

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

Szkolenie nowego barmana

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

OK, poproszę kawę z podwójną czekoladą, mleczkiem sojowym i bitą śmietaną.



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

„Star Café"

Kawy

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

Dodatki

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

Zaostrz ołówek

Schemat narysuj tutaj.

„Star Café" „Star Café" „Star Café" „Star Café" „Star Café"

Wskazówka: kawę z podwójną czekoladą, mleczkiem sojowym i bitą śmietaną możesz przyrządzić, biorąc jedną porcję kawy Star Cafe Special, jedną porcję mleczka sojowego, dwie porcje czekolady oraz jedną porcję bitej śmietany.

Tworzymy kod aplikacji „Star Café”

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

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

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

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

    public String pobierzOpis() {
        return opis;
    }

    public abstract double koszt();
}
```

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

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

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

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

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

Tworzenie kodu klas opisujących napoje

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

```
public class Espresso extends Napój {

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

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

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

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

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

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

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

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

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

„Star Café"

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

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

Tworzenie kodu klas opisujących dodatki

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

Czekolada jest dekoratorem, więc dziedziczy po klasie SkładnikDekorator.

Pamiętaj, klasa SkładnikiDekorator dziedziczy po klasie Napoje.

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

(1) Zmienną obiektową przechowującą dekorowany obiekt.

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

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

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

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

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

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

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

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

Zaostrz ołówek

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

Podajemy kawę

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

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

```
public class StarCafe {

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

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

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

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

Utwórz obiekt MocnoPalona.

Udekoruj go Czekoladą.

Udekoruj drugą porcją Czekolady.

Udekoruj BitąŚmietaną.

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

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

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

% java StarCafe
Kawa Espresso 1.99 zł
Kawa Mocno Palona, Czekolada, Czekolada, Bita Śmietana 1.49 zł
Kawa Star Cafe Special, Mleczko Sojowe, Czekolada, Bita Śmietana 1.34 zł

%
```

Nie ma niemądrych pytań

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

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

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

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

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

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

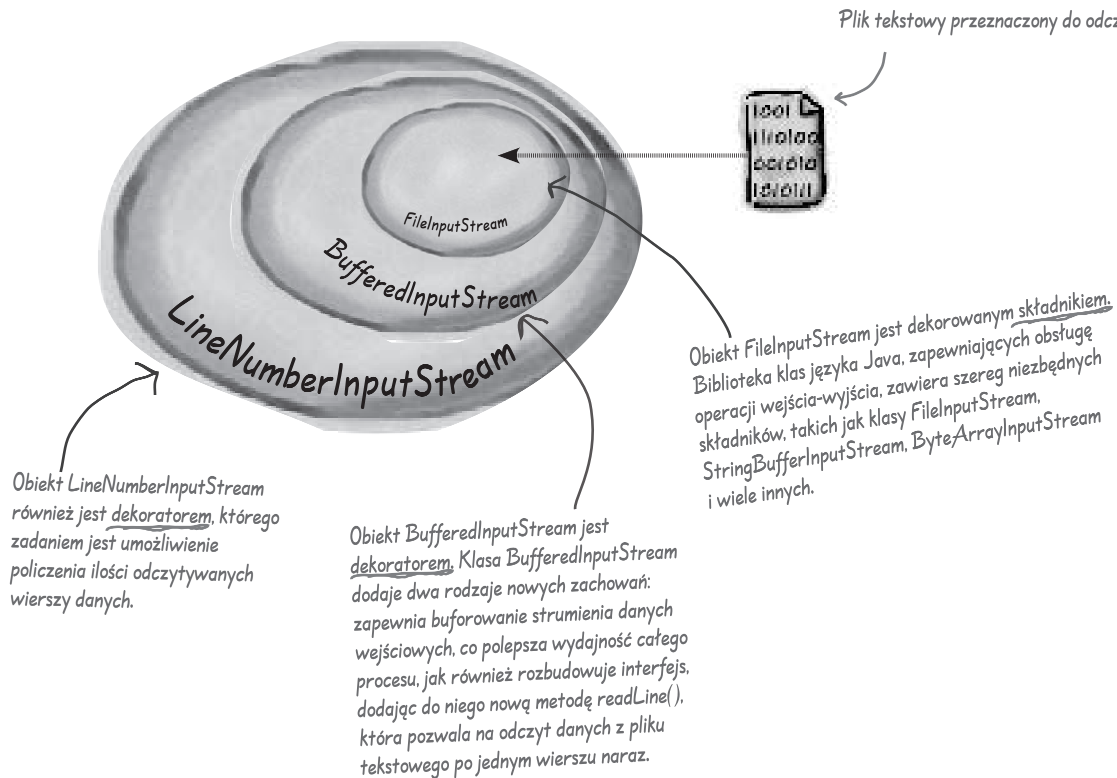
Zaostrz ołówek

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

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

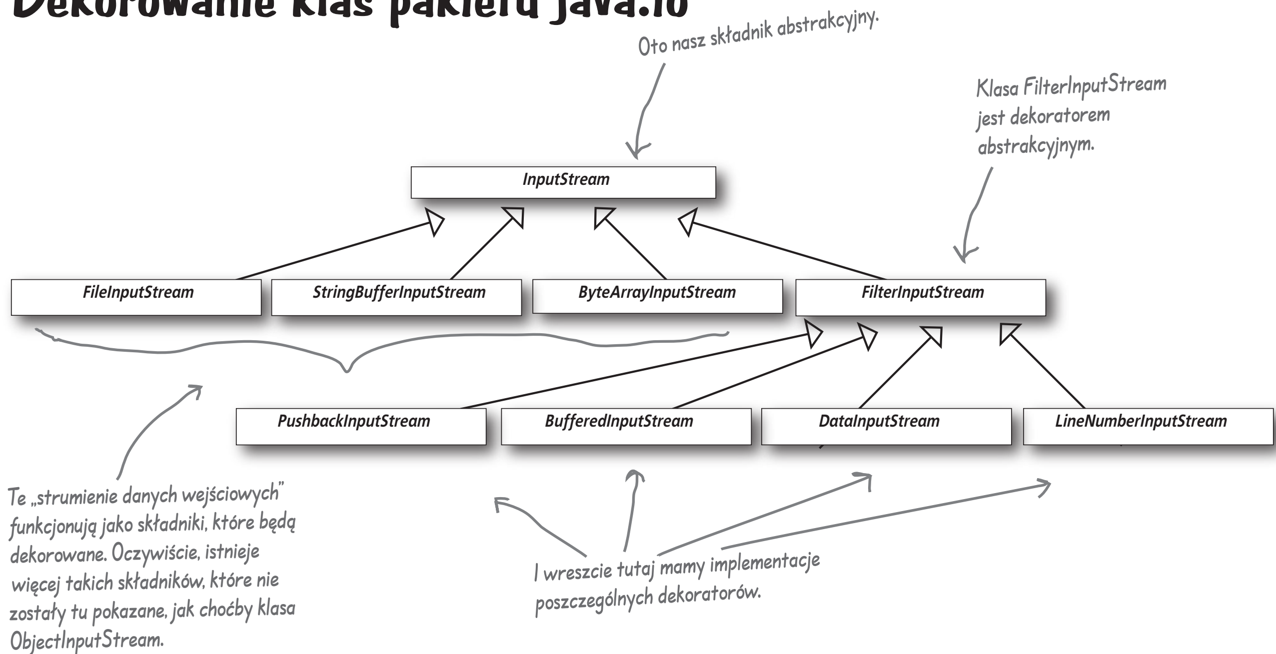
Dekoratory w świecie rzeczywistym: obsługa wejścia-wyjścia w języku Java

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



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

Dekorowanie klas pakietu java.io



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

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

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

Tworzenie własnych dekoratorów obsługi wejścia-wyjścia

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

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

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

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

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

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

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

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

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

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

Testowanie nowego dekoratora obsługującego wejście-wyjście

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

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

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

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

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

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

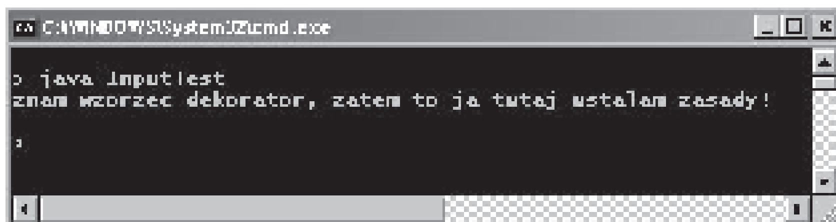
Korzystając ze strumienia danych, odczytujemy dane z pliku i na bieżąco drukujemy je na ekranie.

```
Znam wzorzec Dekorator, zatem to Ja tutaj USTALAM ZASADY!
```

plik test.txt

Musisz utworzyć ten plik samodzielnie.

Odpalamy program:



Wzorce bez tajemnic

Wywiad tygodnia:
Wyznania Dekoratora

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

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

HeadFirst: To może podzielisz się swoimi problemami również z nami?

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

HeadFirst: Czy możesz nam zaprezentować jakiś konkretny przykład?

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

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

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

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

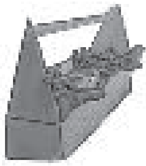
Dekorator: No wiem, wiem, i mimo wszystko staram się tego nie robić. Innym problemem, z którym się muszę borykać, jest to, że wprowadzenie dekoratorów do aplikacji może spowodować wzrost stopnia złożoności kodu niezbędnego do prawidłowego tworzenia obiektów poszczególnych klas składników. Kiedy korzystasz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

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

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

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

Dekorator: Postaram się, dziękuję bardzo.



Twoja skrzynka narzędziowa

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

Podstawy programowania obiektowego

Abstrakcyjność

Hermetyzacja

Polimorfizm

Dziedziczenie

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.

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

Od tej pory będziemy się również kierowali regułą otwarte-zamknięte. Będziemy się starali realizować projekty naszych systemów w taki sposób, aby gotowe, zamknięte elementy były skutecznie izolowane od wprowadzanych przez nas rozszerzeń.

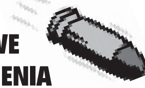
Wzorce programowania obiektowego

Str... algo... tyz... one

Obserwator — definiuje pomiędzy... obie... spos... nia s... zale... auto...

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

A tutaj mamy nasz pierwszy wzorzec projektowy, który spełniał wymogi reguły otwarte-zamknięte. Czy jednak naprawdę był pierwszy? Czy jest jakiś inny wzorzec, którego już używaliśmy, a który również był zgodny z tą regułą?



KLUCZOWE ZAGADNIENIA

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

Rozwiązania ćwiczeń

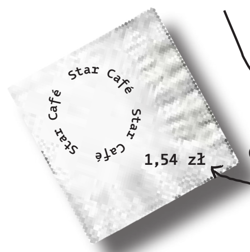
```
public class Napój {

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

public float koszt() {

        float dodatekKoszt = 0,0;
      if (zMlekiem()) {
         dodatekKoszt += mlekoKoszt;
      }
      if (zMleczkiemSojowym()) {
         dodatekKoszt += mleczkoSojoweKoszt;
      }
      if (zCzekoladą()) {
         dodatekKoszt += czekoladaKoszt;
      }
      if (zBitąŚmietaną()) {
         dodatekKoszt += bitaŚmietanaKoszt;
      }
      return dodatekKoszt;
   }
}
```

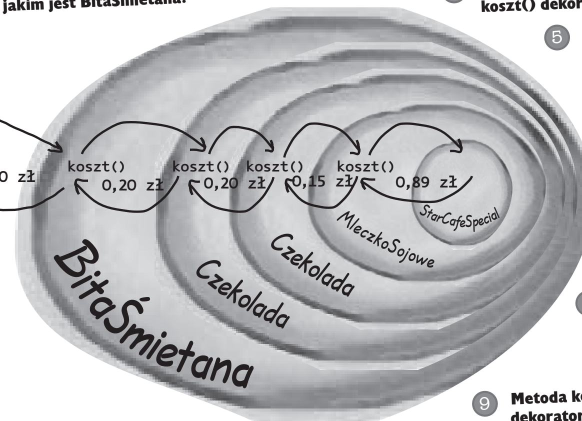
```
public class MocnoPalona extends Napój {

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

   public float koszt() {

      return 1,99 + super.koszt();

   }
}
```

Szkolenie nowego barmana

kawa z podwójną czekoladą, mleczkiem sojowym i bitą śmietaną

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





Rozwiązania ćwiczeń

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

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

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

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

    public pobierzWielkość() {
        return napój.pobierzWielkość();
    }

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

    public double koszt() {
        double koszt = napój.koszt();
        if (pobierzWielkość() == Napój.MAŁY) {
            koszt += 0,10;
        } else if (pobierzWielkość() == Napój.ŚREDNI) {
            koszt += 0,15;
        } else if (pobierzWielkość() == Napój.DUŻY) {
            koszt += 0,20;
        }
        return.koszt;
        }
}
```

Teraz musimy zaimplementować metodę pobierzWielkość() do dekorowanego napoju. Musimy również zaimplementować tę metodę w klasie abstrakcyjnej, ponieważ będzie używana przez wszystkie dekoratory.

Tutaj pobieramy wielkość napoju, a następnie dodajemy do ceny napoju odpowiedni koszt dodatku.