

Matt Weisfeld



Myślenie obiektowe w programowaniu

Wydanie IV

Poznaj świat programowania obiektowego!



Tytuł oryginału: The Object-Oriented Thought Process (4th Edition)

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-246-8120-4

Authorized translation from the English language edition, entitled: THE OBJECT-ORIENTED THOUGHT PROCESS, Fourth Edition; ISBN 0321861272; by Matt Weisfeld; published by Pearson Education, Inc, publishing as Addison Wesley.
Copyright © 2013 Pearson Education, Inc.

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 Pearson Education, Inc.
Polish language edition published by HELION S.A. Copyright © 2014.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek 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/myobp4>
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

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

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

Spis treści

O autorze	13
Wstęp	15
Tematyka książki	15
Nowości w czwartym wydaniu	17
Adresaci książki	17
Metodyka	18
Konwencje	19
Kod źródłowy	19
Rozdział 1 Podstawowe pojęcia obiektowości	21
Podstawowe pojęcia	22
Obiekty a stare systemy	22
Programowanie obiektowe a proceduralne	24
Zamiana podejścia proceduralnego na obiektowe	27
Programowanie proceduralne	27
Programowanie obiektowe	28
Definicja obiektu	28
Dane obiektu	29
Zachowania obiektu	29
Definicja klasy	33
Tworzenie obiektów	34
Atrybuty	35
Metody	36
Komunikaty	36
Modelowanie klas przy użyciu diagramów UML	36
Hermetyzacja i ukrywanie danych	37
Interfejsy	37
Implementacje	38
Realistyczna ilustracja paradygmatu interfejsu i implementacji	39
Model paradygmatu interfejs – implementacja	39

Dziedziczenie	40
Nadklasy i podklasy	42
Abstrakcja	42
Związek typu „jest”	42
Polimorfizm	44
Kompozycja	47
Abstrakcja	47
Związek typu „ma”	47
Podsumowanie	48
Listingi	48
TestPerson	48
TestShape	49
Rozdział 2 Myślenie w kategoriach obiektowych	51
Różnica między interfejsem a implementacją	52
Interfejs	54
Implementacja	54
Przykład implementacji i interfejsu	55
Zastosowanie myślenia abstrakcyjnego w projektowaniu interfejsów	59
Minimalizowanie interfejsu	61
Określanie grupy docelowej	62
Zachowania obiektu	63
Ograniczenia środowiska	63
Identyfikowanie publicznych interfejsów	63
Identyfikowanie implementacji	64
Podsumowanie	65
Źródła	65
Rozdział 3 Zaawansowane pojęcia z zakresu obiektowości	67
Konstruktory	67
Kiedy wywoływany jest konstruktor	68
Zawartość konstruktora	68
Konstruktor domyślny	69
Zastosowanie wielu konstruktorów	70
Projektowanie konstruktorów	73
Obsługa błędów	74
Ignorowanie problemu	74
Szukanie błędów i kończenie działania programu	75
Szukanie błędów i próba ich naprawienia	75
Zgłaszanie wyjątków	76
Pojęcie zakresu	78
Atrybuty lokalne	78
Atrybuty obiektowe	79
Atrybuty klasowe	81

Przeciążanie operatorów	82
Wielokrotne dziedziczenie	83
Operacje obiektów	84
Podsumowanie	85
Źródła	85
Listingi	86
TestNumber	86
Rozdział 4 Anatomia klasy	87
Nazwa klasy	87
Komentarze	89
Atrybuty	89
Konstruktory	91
Metody dostępne	93
Metody interfejsu publicznego	95
Prywatne metody implementacyjne	95
Podsumowanie	96
Źródła	96
Listingi	96
TestCab	96
Rozdział 5 Wytyczne dotyczące projektowania klas	99
Modelowanie systemów świata rzeczywistego	99
Identyfikowanie interfejsów publicznych	100
Minimalizacja interfejsu publicznego	100
Ukrywanie implementacji	101
Projektowanie niezawodnych konstruktorów i destruktorów	102
Projektowanie mechanizmu obsługi błędów w klasie	103
Pisanie dokumentacji i stosowanie komentarzy	103
Tworzenie obiektów nadających się do kooperacji	104
Wielokrotne użycie kodu	104
Rozszerzalność	105
Tworzenie opisowych nazw	105
Wyodrębnianie nieprzenośnego kodu	106
Umożliwianie kopiowania i porównywania obiektów	107
Ograniczanie zakresu	107
Klasa powinna odpowiadać sama za siebie	108
Konservacja kodu	109
Iteracja	110
Testowanie interfejsu	110
Wykorzystanie trwałości obiektów	112
Serializacja i szeregowanie obiektów	113
Podsumowanie	113
Źródła	114

Listingi	114
TestMath	114
Rozdział 6 Wytyczne dotyczące projektowania klas	115
Wytyczne dotyczące projektowania	115
Wykonanie odpowiedniej analizy	119
Określanie zakresu planowanych prac	119
Gromadzenie wymagań	120
Opracowywanie prototypu interfejsu użytkownika	120
Identyfikowanie klas	120
Definiowanie wymagań wobec każdej z klas	121
Określenie warunków współpracy między klasami	121
Tworzenie modelu klas opisującego system	121
Tworzenie prototypu interfejsu użytkownika	121
Obiekty opakowujące	122
Kod strukturalny	122
Opakowywanie kodu strukturalnego	124
Opakowywanie nieprzenośnego kodu	125
Opakowywanie istniejących klas	126
Podsumowanie	127
Źródła	128
Rozdział 7 Dziedziczenie i kompozycja	129
Wielokrotne wykorzystywanie obiektów	129
Dziedziczenie	131
Generalizacja i specjalizacja	133
Decyzje projektowe	134
Kompozycja	136
Reprezentowanie kompozycji na diagramach UML	137
Czemu hermetyzacja jest podstawą technologii obiektowej	138
Jak dziedziczenie osłabia hermetyzację	139
Szczegółowy przykład wykorzystania polimorfizmu	141
Odpowiedzialność obiektów	141
Klasy abstrakcyjne, metody wirtualne i protokoły	145
Podsumowanie	146
Źródła	147
Listingi	147
TestShape	147
Rozdział 8 Wielokrotne wykorzystanie kodu	
— interfejsy i klasy abstrakcyjne	149
Wielokrotne wykorzystanie kodu	149
Infrastruktura programistyczna	150
Co to jest kontrakt	152
Klasy abstrakcyjne	153
Interfejsy	156

Wnioski	158
Dowód kompilatora	160
Zawieranie kontraktu	161
Punkty dostępowe do systemu	163
Przykład biznesu elektronicznego	163
Biznes elektroniczny	164
Podejście niezakładające wielokrotnego wykorzystania kodu	165
Rozwiązanie dla aplikacji biznesu elektronicznego	167
Model obiektowy UML	167
Podsumowanie	170
Źródła	170
Listingi	170
TestShop	171
Rozdział 9 Tworzenie obiektów	175
Relacje kompozycji	175
Podział procesu budowy na etapy	177
Rodzaje kompozycji	179
Agregacja	179
Asocjacja	180
Łączne wykorzystanie asocjacji i agregacji	181
Unikanie zależności	182
Licznosc	183
Kilka asocjacji	184
Asocjacje opcjonalne	186
Praktyczny przykład	186
Podsumowanie	187
Źródła	188
Rozdział 10 Tworzenie modeli obiektowych	189
Co to jest UML	189
Struktura diagramu klasy	190
Atrybuty i metody	192
Atrybuty	192
Metody	192
Określanie dostępności	193
Dziedziczenie	194
Interfejsy	195
Kompozycja	196
Agregacja	196
Asocjacja	197
Licznosc	199
Podsumowanie	200
Źródła	201

Rozdział 11	Obiekty i dane przenośne — XML	203
	Przenośność danych	204
	Rozszerzalny język znaczników — XML	205
	XML a HTML	206
	XML a języki obiektowe	207
	Wymiana danych między firmami	208
	Sprawdzanie poprawności dokumentu względem DTD	208
	Integrowanie DTD z dokumentem XML	210
	Kaskadowe arkusze stylów	216
	Notacja obiektowa języka JavaScript (JSON)	217
	Podsumowanie	222
	Źródła	223
Rozdział 12	Obiekty trwałe — serializacja i relacyjne bazy danych	225
	Podstawy trwałości obiektów	225
	Zapisywanie obiektu w pliku płaskim	226
	Serializacja pliku	227
	Jeszcze raz o implementacji i interfejsach	229
	Serializacja metod	231
	Serializacja przy użyciu języka XML	231
	Zapisywanie danych w relacyjnej bazie danych	233
	Dostęp do relacyjnej bazy danych	235
	Podsumowanie	237
	Źródła	237
	Listingi	238
	Klasa Person	238
Rozdział 13	Obiekty w usługach sieciowych, aplikacjach mobilnych i aplikacjach hybrydowych	241
	Ewolucja technik przetwarzania rozproszonego	241
	Obiektowe skryptowe języki programowania	242
	Weryfikacja danych za pomocą języka JavaScript	245
	Obiekty na stronach internetowych	248
	Obiekty JavaScript	248
	Kontrolki na stronach internetowych	250
	Odtwarzacze dźwięku	250
	Odtwarzacze filmów	251
	Animacje Flash	252
	Obiekty rozproszone i systemy przedsiębiorstw	252
	CORBA	254
	Definicja usługi sieciowej	257
	Kod usług sieciowych	261
	Representational State Transfer (ReST)	263
	Podsumowanie	264
	Źródła	264

Rozdział 14	Obiekty w aplikacjach typu klient-serwer	265
	Model klient-serwer	265
	Rozwiązanie własnościowe	266
	Kod obiektu do serializacji	266
	Kod klienta	267
	Kod serwera	269
	Uruchamianie aplikacji	270
	Technika z wykorzystaniem XML	271
	Definicja obiektu	272
	Kod klienta	273
	Kod serwera	274
	Uruchamianie programu	276
	Podsumowanie	276
	Źródła	276
	Listingi	277
Rozdział 15	Wzorce projektowe	279
	Historia wzorców projektowych	280
	Wzorzec MVC języka Smalltalk	280
	Rodzaje wzorców projektowych	283
	Wzorce konstrukcyjne	283
	Wzorce strukturalne	288
	Wzorce czynnościowe	290
	Antywzorce	291
	Podsumowanie	292
	Źródła	292
	Listingi	293
	Counter.cs	293
	Singleton.cs	293
	MailTool.cs	294
	MailInterface.cs	294
	MyMailTool.cs	295
	Adapter.cs	295
	Iterator.cs	296
	Skorowidz	297

Dziedziczenie i kompozycja

Dziedziczenie i kompozycja odgrywają w projektowaniu systemów obiektowych bardzo ważną rolę. W istocie wiele najtrudniejszych i najciekawszych decyzji można sprowadzić do wyboru między tymi dwoma.

Decyzje te w miarę ewoluowania technik obiektowych stawały się coraz ciekawsze. Do najciekawszych dyskusji, jakie toczą się na tematy związane z obiektowością, należy spór dotyczący dziedziczenia. Mimo że dziedziczenie jest jednym z fundamentów programowania obiektowego (aby język uznano za obiektowy, musi umożliwiać korzystanie z tej techniki), wielu programistów unika go, wybierając inne metody projektowe.

Zarówno dziedziczenie, jak i kompozycja to techniki umożliwiające wielokrotne wykorzystanie kodu. **Dziedziczenie**, jak sama nazwa wskazuje, to technika polegająca na dziedziczeniu atrybutów i zachowań przez klasy po innych klasach. Wytwarzają się prawdziwe relacje typu rodzic – dziecko. Dziecko (czyli podklasa) dziedziczy bezpośrednio po swoim rodzicu (czyli nadklasie).

Kompozycja, również jak nazwa wskazuje, oznacza tworzenie obiektów przy użyciu innych obiektów. W rozdziale tym opiszę wyraźne i bardziej subtelne różnice między tymi dwiema technikami. Zacznę od tego, kiedy w ogóle należy stosować każdą z nich.

Wielokrotne wykorzystywanie obiektów

Najważniejszym powodem, dla którego powstały techniki dziedziczenia i kompozycji, jest chęć wielokrotnego wykorzystania obiektów. Mówiąc krótko, klasy (które służą do tworzenia obiektów) można tworzyć, wykorzystując inne klasy za pomocą dziedziczenia lub kompozycji. Oznacza to, że techniki te są jedynymi, za pomocą których można wielokrotnie wykorzystywać kod istniejących klas.

Dziedziczenie reprezentuje związek typu „jest”, którego definicję przedstawiłem w rozdziale 1. „Wstęp do obiektowości”. Na przykład pies **jest** ssakiem.

Kompozycja to technika polegająca na budowaniu skomplikowanych klas przy użyciu innych klas — to tworzenie pewnego rodzaju kolekcji. Nie implikuje relacji typu

rodzic – dziecko. Zasadniczo obiekty złożone składają się z innych obiektów. Kompozycja reprezentuje związek typu „ma”. Na przykład samochód **ma** silnik. Zarówno samochód, jak i silnik są odrębnymi obiektami. Jednak ten pierwszy jest obiektem złożonym, który zawiera (ma) obiekt silnika. W istocie nawet obiekt potomny może być złożony z innych obiektów. Na przykład silnik może zawierać cylindry. Wówczas można powiedzieć, że samochód **ma** cylinder, a nawet kilka.

Gdy technologia obiektowa trafiła do szerokiego grona odbiorców, dziedziczenie stało się wielkim hitem. Możliwość pisania klas, których funkcjonalność można było później wykorzystywać w wielu innych klasach, uznawano za najważniejszą zaletę obiektowości. Nazwano to wielokrotnym wykorzystaniem kodu — wyrazem tej techniki było właśnie dziedziczenie.

Po pewnym jednak czasie blask dziedziczenia nieco przygasł. Niektórzy nawet kwestionują zasadność stosowania tej techniki. Peter Coad i Mark Mayfield w swojej książce *Java Design* zamieścili nawet cały rozdział zatytułowany „Design with Composition Rather Than Inheritance” (Wykorzystywanie w projektowaniu kompozycji zamiast dziedziczenia). Wiele wczesnych platform obiektowych nie obsługiwało nawet tej techniki w czystej postaci. W języku Visual Basic rzeczywistą obsługę dziedziczenia wprowadzono dopiero w wersji .NET. Platformy takie jak Microsoft COM zostały oparte na dziedziczeniu interfejsów. Temat ten szczegółowo opisuję w rozdziale 8. „Modele i wielokrotne wykorzystanie kodu: projektowanie z wykorzystaniem interfejsów i klas abstrakcyjnych”.

Dziś zastosowanie dziedziczenia wciąż jest przedmiotem ożywionych dyskusji. Będące częścią technik dziedziczenia klasy abstrakcyjne w niektórych językach programowania, takich jak Objective-C, nie są bezpośrednio dostępne. Używa się interfejsów, mimo że nie mają one wszystkich cech dostępnych dzięki klasom abstrakcyjnym.

Dobrze zdać sobie sprawę z tego, że wynikiem całej tej dyskusji na temat używania lub nieużywania dziedziczenia i kompozycji będzie w końcu opracowanie jakiegoś kompromisowego stanowiska. Tak jak we wszystkich debatach natury filozoficznej i tu po obu stronach stoją fanatyczni zwolennicy. Na szczęście, jak zawsze w takich sytuacjach, gorąca dyskusja doprowadziła do lepszego zrozumienia sposobów wykorzystania technologii, które jej podlegają.

W dalszej części rozdziału wyjaśnię, dlaczego niektórzy uważają, iż dziedziczenia należy unikać, a w jego miejsce stosować kompozycję. Jest to delikatny i skomplikowany problem. W rzeczywistości przy projektowaniu klas można wykorzystywać zarówno dziedziczenie, jak i kompozycję, które mają swoje ustalone miejsca w obiektowości. Trzeba przynajmniej rozumieć obie te techniki, aby móc dokonać rozsądnego wyboru jednej z nich.

To, że dziedziczenie jest często źle stosowane i nadużywane, jest wynikiem braku zrozumienia tej techniki, a nie jakąś jej fundamentalną usterką.

Należy pamiętać, że zarówno dziedziczenie, jak i kompozycja to ważne techniki służące do budowy systemów obiektowych. Projektanci i programiści powinni odpowiednio się przygotować od strony merytorycznej, aby zrozumieć ich mocne i słabe strony. To pozwoli im stosować obie te techniki w odpowiedni sposób.

Dziedziczenie

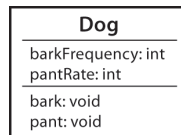
W rozdziale 1. dziedziczenie zdefiniowane zostało jako system, w którym klasy potomne dziedziczą atrybuty i zachowania po klasach rodzicach. Jednak na temat dziedziczenia można powiedzieć znacznie więcej, co zrobię w tym rozdziale.

W rozdziale 1. napisałem, że dziedziczenie można rozpoznać bardzo prostą metodą. Jeśli stwierdzenie „Klasa B jest rodzajem klasy A” jest prawdziwe, związek taki może być określony jako dziedziczenie.

Związek typu „jest”

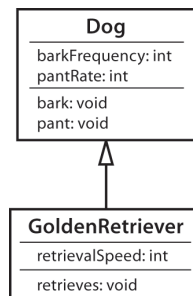
Jedna z najważniejszych zasad projektowania obiektowego głosi, że interfejs publiczny powinien być reprezentowany przez związek typu „jest”.

Skorzystam z przedstawionego w rozdziale 1. przykładu rodziny ssaków. Załóżmy, że utworzono klasę `Dog`. Psy mają kilka charakterystycznych dla siebie zachowań, które odróżniają je od np. kotów. Na potrzeby tego przykładu niech będą to szczekanie (ang. *bark*) i sapanie (ang. *pant*). Na tej podstawie można utworzyć klasę `Dog` z dwoma zachowaniami i dwoma atrybutami (rysunek 7.1).



Rysunek 7.1. Diagram klasy `Dog`

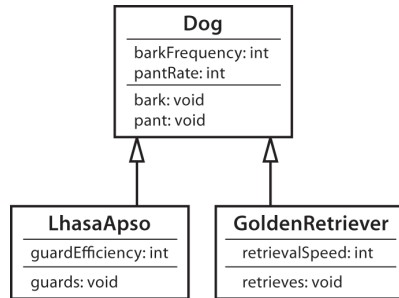
Teraz utworzymy klasę `GoldenRetriever`. Można by było utworzyć całkiem nową klasę z takimi samymi zachowaniami jak w klasie `Dog`. Można jednak też dojść do następującego rozsądnego wniosku: golden retriever to **jest** pies. Można zatem wykorzystać dziedziczenie i utworzyć klasę `GoldenRetriever` jako podklasę klasy `Dog`, z której zostaną odziedziczone wszystkie atrybuty i zachowania (rysunek 7.2).



Rysunek 7.2. Klasa `GoldenRetriever` dziedziczy po `Dog`

Dzięki temu klasa `GoldenRetriever` poza własnymi zachowaniami będzie zawierała wszystkie te, które są charakterystyczne ogólnie dla psów. Jest to korzystne z kilku powodów. Po pierwsze, przy tworzeniu tej klasy nie trzeba było od nowa wynajdywać koła, pisząc po raz drugi metody `bark` i `pant`. To nie tylko pozwala zaoszczędzić na czasie, jeśli chodzi o pisanie kodu, ale również umożliwia ograniczenie czynności związanych z testowaniem i konserwacją. Metody `bark` i `pant` zostały napisane jeden raz, przy założeniu, że odpowiednio je przetestowano przy okazji pisania klasy `Dog`, i nie trzeba już ich szczegółowo testować. Ale pewne testy trzeba przeprowadzić jeszcze raz, bo pojawiły się nowe interfejsy itd.

Spróbuję w pełni wykorzystać zalety dziedziczenia. W tym celu utworzę drugą podklasę klasy `Dog`, o nazwie `LhasaApso`. Podczas gdy retrievery są hodowane, aby aportowały, lhasa apso mogą służyć jako strażnicy. Mają wyczulone zmysły i kiedy wyczują coś podejrzanego, od razu zaczynają szczekać. Nie są jednak agresywne. Można więc utworzyć klasę `LhasaApso`, która, podobnie jak `GoldenRetriever`, będzie dziedziczyła po klasie `Dog` (rysunek 7.3).



Rysunek 7.3. Klasa `LhasaApso` dziedziczy po klasie `Dog`

Testowanie nowego kodu

W opisywanym tu przypadku klasy `GoldenRetriever` metody `bark` i `pant` powinny zostać napisane, przetestowane i oczyszczone z błędów już na etapie prac nad klasą `Dog`. Teoretycznie kod ich powinien być już niezawodny i gotowy do użytku w różnych sytuacjach. Jednak to, że kod można wykorzystać wielokrotnie, nie oznacza, że nie trzeba go testować. Mimo iż wydaje się to mało prawdopodobne, rasa golden retriever może mieć pewne cechy, które w jakiś sposób będą zakłócać działanie tego kodu. Główna zasada jest taka, że zawsze należy testować nowy kod. Każdy związek dziedziczenia wytwarza nowy kontekst, w którym mogą być używane odziedziczone metody. W kompletnej strategii testowania wszystkie takie konteksty powinny być uwzględnione.

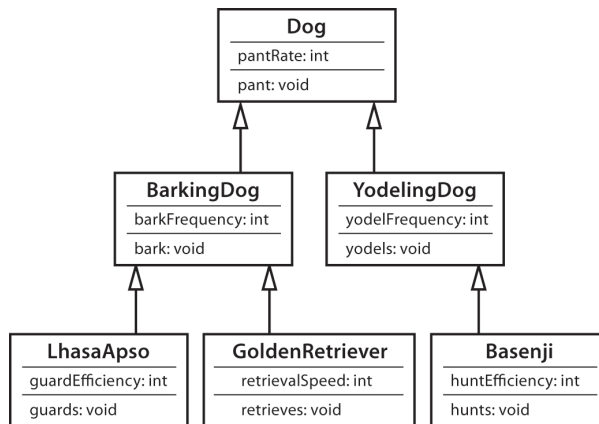
Kolejną zaletą dziedziczenia jest to, że kod metod `bark()` i `pant()` znajduje się tylko w jednym miejscu. Załóżmy, że trzeba zmodyfikować kod pierwszej z nich. Wówczas wystarczy tylko zmiana w klasie `Dog`, a będzie ona od razu przejęta przez klasy `LhasaApso` i `GoldenRetriever`.

Czy jest tu jakiś problem? Na tym poziomie wydaje się, że model dziedziczenia sprawdza się doskonale. Czy można jednak mieć pewność, że wszystkie psy mają takie zachowania, które zdefiniowano w klasie Dog?

W swojej książce *Effective C++* Scott Meyers podał świetny przykład ilustrujący dylemat dotyczący wykorzystania dziedziczenia w projektowaniu. Weźmy na przykład klasę reprezentującą ptaka. Jedną z najbardziej charakterystycznych cech ptaków jest to, że potrafią latać. Tworzymy zatem klasę Bird z metodą fly. Od razu nasuwa się pytanie, co zrobić z pingwinami i strusiami? Są ptakami, a nie potrafią fruwać. Można by było to zachowanie lokalnie przesłonić, ale nazwa metody pozostałaby taka sama. A przecież nie byłoby sensu tworzyć metody fly dla ptaka, który nie umie latać, a tylko chodzić jak kaczka.

To może prowadzić do powstawania potencjalnie groźnych problemów. Gdyby na przykład pingwin miał metodę fly, zrozumiałe, że zechciałby ją kiedyś wypróbować. Gdyby jednak metoda ta została przesłonięta i odpowiadające jej zachowanie w rzeczywistości nie istniałoby, pingwin byłby bardzo zdziwiony po wywołaniu metody fly. Wyobraź sobie rozgoryczenie pingwina, który wywołał metodę fly, a uzyskał jedynie kaczyczą chód.

W przypadku klasy psów założono, że wszystkie psy szczekają. Są jednak takie rasy, które tego nie robią, np. basenji. Psy tej rasy nie szczekają, tylko jodłują. Czy należy zatem przeprowadzić rewizję pierwotnego projektu? Jak by po niej wyglądał? Na rysunku 7.4 został przedstawiony lepszy model hierarchii klasy Dog.



Rysunek 7.4. Hierarchia klasy Dog

Generalizacja i specjalizacja

Jako przykładem posłużę się hierarchią klasy Dog. Na początku utworzono jedną klasę o nazwie Dog, w której zdefiniowano pewne cechy wspólne wszystkich psów. Koncepcja ta jest czasami nazywana *generalizacją-specjalizacją* i stanowi kolejną rzecz, którą należy wziąć pod uwagę przy wykorzystywaniu dziedziczenia. Chodzi o to, że im niższy poziom

drzewa dziedziczenia, tym bardziej specyficzna klasa. Najbardziej ogólna znajduje się na samym wierzchołku. W przedstawionym przykładzie jest to klasa `Dog`. Najbardziej specyficzne są klasy odpowiadające poszczególnym rasom psów — `GoldenRetriever`, `LhasaApso` i `Basenji`. Zasadą dziedziczenia jest przechodzenie od najbardziej ogólnego przypadku do najbardziej specyficznego poprzez wyodrębnianie cech wspólnych.

W modelu dziedziczenia klasy `Dog` wyszliśmy z założenia, że mimo iż golden retrievery mają pewne zachowania, których nie mają psy lhasa apso, rasy te mają także pewne cechy wspólne — na przykład jedne i drugie szczekają i sapią. Następnie zdaliśmy sobie sprawę, że są takie psy, które nie szczekają, tylko jodłują. Zmusiło nas to do wydzielenia szczekania do osobnej klasy o nazwie `BarkingDog`. Jodłowanie znalazło się w klasie `YodelingDog`. Wiemy też, że mimo pewnych różnic jedne i drugie psy mają ze sobą coś wspólnego — sapanie. Dlatego klasy `YodelingDog` i `BarkingDog` dziedziczą po klasie `Dog`. Teraz klasa `Basenji` może dziedziczyć po `YodelingDog`, a klasy `GoldenRetriever` i `LhasaApso` po `BarkingDog`.

Można by było uniknąć tworzenia dwóch osobnych klas dla psów jodłujących i szczekających. Wówczas szczekanie i jodłowanie można by było zaimplementować jako część klasy reprezentującej każdą rasę — odgłosy wydawane przez każdego psa mogą brzmieć inaczej. Jest to tylko jeden przykład decyzji projektowych, jakie czasami trzeba podejmować. Prawdopodobnie najlepszym rozwiązaniem byłoby zaimplementować szczekanie i jodłowanie jako interfejsy, które zostaną opisane w rozdziale 8. „Modele i wielokrotne wykorzystanie kodu: projektowanie z wykorzystaniem interfejsów i klas abstrakcyjnych”.

Decyzje projektowe

Teoretycznie wyodrębnienie jak największej liczby cech wspólnych jest bardzo dobrym pomysłem. Jednak jak to zwykle bywa w projektowaniu, czasami można przedobrzyć. Podczas gdy znajdowanie cech wspólnych kilku klas jest dobrym sposobem na jak najwierniejsze odwzorowanie świata rzeczywistego, może już nie tak dobrze reprezentować sam model. Im więcej tego typu działań, tym bardziej skomplikowany robi się system. Powstaje dylemat: czy bardziej potrzebny jest precyzyjny model, czy mniej skomplikowany system? Decyzję należy podjąć w zależności od sytuacji. Nie ma sztywnego zestawu wskazówek, które pomogą w jej podjęciu.

W czym komputery nie są dobre

Oczywiście system komputerowy może tylko w przybliżeniu modelować świat rzeczywisty. Komputery doskonale sprawdzają się w wykonywaniu obliczeń, ale gorzej radzą sobie z operacjami abstrakcyjnymi.

Na przykład rozbitcie klasy `Dog` na `YodelingDog` i `BarkingDog` lepiej pozwala odwzorować świat rzeczywisty niż przyjęcie założenia, że wszystkie psy szczekają, ale wiąże się z tym dodatkowa komplikacja systemu.

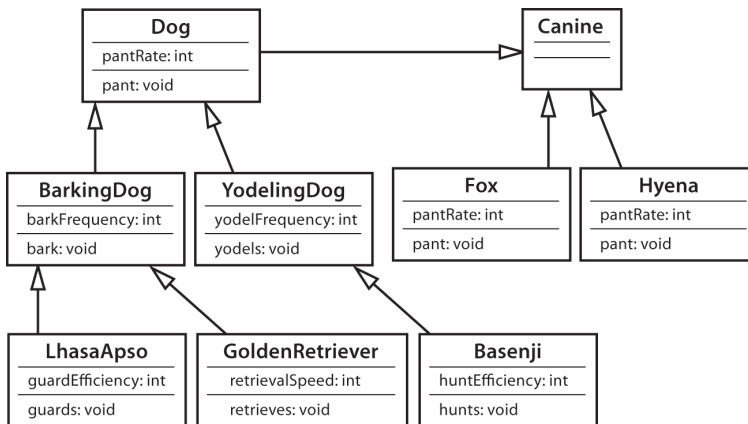
Złożoność modelu

Na takim poziomie jak w przedstawionym przykładzie dodanie dwóch klas nie komplikuje systemu aż tak, żeby go uczynić trudnym do ogarnięcia. Natomiast w dużych systemach tego rodzaju decyzje, jeśli są podejmowane wielokrotnie, mogą szybko doprowadzić do dużej komplikacji. W takich systemach najlepszą strategią jest bronienie za wszelką cenę prostoty.

Zdarzają się sytuacje, w których doprecyzowanie modelu nie powoduje jego dodatkowego skomplikowania. Wyobraźmy sobie hodowcę psów, który podpisał kontrakt na system informacji o wszystkich swoich psach. Model podzielony na psy szczekające i jodłujące działa bardzo dobrze. Wyobraźmy sobie, że hodowca ten nie hoduje psów, które jodłują — nigdy tego nie robił i nie planuje zacząć. Wówczas nie ma raczej sensu komplikować systemu niepotrzebnym podziałem. System będzie prostszy, a jego funkcjonalność w ogóle nie zmaleje.

Podjęcie decyzji, czy zaprojektować prostszy, czy bardziej funkcjonalny system, jest bardzo trudne. Główną zasadą jest to, aby tworzyć systemy elastyczne i na tyle mało skomplikowane, że nie zawalą się pod własnym ciężarem.

Ważnym czynnikiem mającym wpływ na decyzję są bieżące i przyszłe koszty. Mimo iż czasami może się wydawać, że właściwym podejściem jest zaprojektowanie kompletnego i elastycznego systemu, dodatkowa funkcjonalność może tylko w niewielkim stopniu przynosić korzyści — po prostu inwestycja może się nie opłacić. Czy warto by było rozszerzyć projekt klasy Dog na dodatkowe zwierzęta z tej rodziny, np. hieny i lisy (rysunek 7.5)?



Rysunek 7.5. Rozszerzony model rodziny psowatych

Projekt taki mógłby być przydatny dla systemu obsługi zoo, ale dla hodowcy psów udomowionych nie jest już raczej odpowiedni.

Jak widać, każdy projekt wymaga jakichś kompromisów.

Podęjmuj decyzje projektowe, myśląc przyszłościowo

W tym momencie można by było powiedzieć „Nigdy nie mów nigdy”. Mimo że teraz hodowca nie ma u siebie jodłujących psów, w przyszłości może zacząć je hodować. Jeśli nie przygotuje się projektu na taką ewentualność od razu, zmiana systemu w przyszłości może być znacznie bardziej kosztowna. Jest to jeszcze jedna z decyzji, które należy podjąć. Metodę bark() można by było przesłonić, aby odpowiadała jodłowaniu. Jest to jednak rozwiązanie sprzeczne z intuicją, ponieważ większość ludzi spodziewa się, że metoda o nazwie bark() oznacza szczekanie.

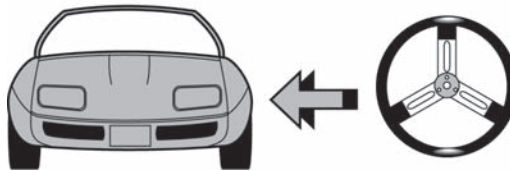
Kompozycja

Traktowanie obiektów jako zbiorów innych obiektów jest naturalnym tokiem rozumowania. Odbiornik telewizyjny zawiera urządzenie odbiorcze i ekran. Komputer zawiera kartę graficzną, klawiaturę i napędy. Komputer można traktować jako jeden duży obiekt, ale podłączana do niego pamięć Flash również jest obiektem. Komputer można otworzyć i wyjąć z niego dysk twardy. W zasadzie dysk ten można by było nawet przenieść do innego komputera. O tym, że dysk jest samodzielnym obiektem, przekonuje fakt, że można go użyć w wielu różnych komputerach.

Klasycznym przykładem kompozycji obiektów jest samochód. Przykład ten jest powtarzany jak mantra w wielu książkach, na kursach i na wykładach. Mimo że części zamienne wcześniej stosowano do naprawy karabinów, dla większości ludzi kwintesencją montażu z gotowych komponentów jest wynaleziona przez Henry’ego Forda linia montażowa. Dlatego wydaje się naturalną kolejną rzeczą, że samochód stał się najważniejszym punktem odniesienia w projektowaniu systemów obiektowych.

Dla większości ludzi oczywiste jest, że samochód ma silnik. Oczywiście poza nim w samochodzie można znaleźć wiele innych rzeczy, np. koła, kierownicę i radio samochodowe. Każdy obiekt składający się z innych obiektów, które są jego polami, nazywa się *agregatem* lub *obiektem złożonym* (rysunek 7.6).

Samochód ma kierownicę



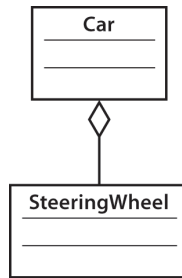
Rysunek 7.6. Przykład kompozycji

Agregacja, asocjacja i kompozycja

Moim zdaniem są tylko dwa sposoby na wielokrotne wykorzystanie kodu klasy — dziedziczenie i kompozycja. Szczegółowy opis tej drugiej metody znajduje się w rozdziale 9. „Tworzenie obiektów” — a mówiąc dokładniej: agregacji i asocjacji. W książce tej przyjąłem, że agregacja i asocjacja to rodzaje kompozycji, chociaż opinie na ten temat są podzielone.

Reprezentowanie kompozycji na diagramach UML

Do zaznaczenia na diagramie UML, że samochód ma kierownicę, służy notacja przedstawiona na rysunku 7.7.



Rysunek 7.7. Reprezentacja kompozycji w UML

Agregacja, asocjacja i UML

W tej książce agregacje (np. silnik jest częścią samochodu) są na diagramach UML oznaczane linią ciągłą zakończoną rombem. Asocjację oznacza sama linia bez rombu na końcu (np. samodzielna klawiatura obsługująca samodzielny komputer).

Zależy zauważyć, że romb na linii łączącej klasę Car z klasą SteeringWheel znajduje się po stronie tej pierwszej. Oznacza to, że samochód (Car) *zawiera* (ma) kierownicę (SteeringWheel).

Rozwinę ten przykład. Załóżmy, że żaden z użytych w tym projekcie obiektów nie wykorzystuje dziedziczenia. Wszystkie relacje między obiektami ograniczają się wyłącznie do kompozycji, która jest wielopoziomowa. Jest to oczywiście uproszczony przykład, ponieważ w każdym samochodzie jest nieporównywalnie więcej relacji zachodzących między różnymi obiektami. Jednak tutaj chodzi tylko o zilustrowanie, na czym polega kompozycja.

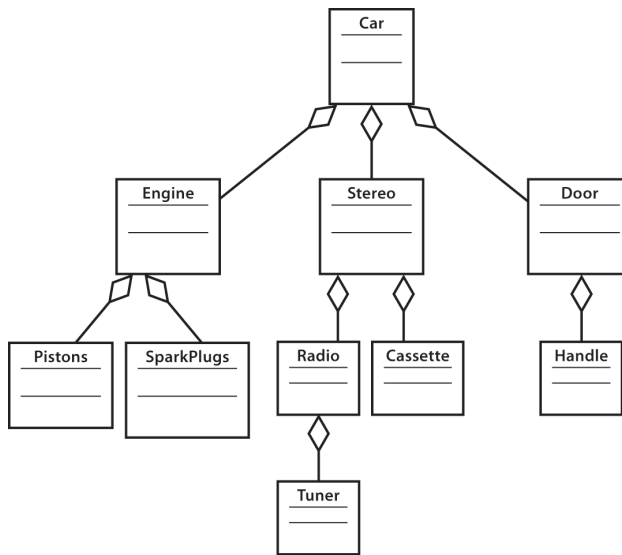
Założmy, że samochód składa się z silnika, radia i drzwi.

Przyjąć do wiadomości fakt, że samochód składa się z silnika, radia i drzwi, jest łatwo, ponieważ większość osób tak właśnie wyobraża sobie samochody. Jednak przy projektowaniu systemów oprogramowania należy pamiętać, że podobnie jak samochody, tak i pozostałe obiekty składają się z innych obiektów. W zasadzie liczba węzłów i gałęzi, które można dodać do drzewa klas, jest nieograniczona.

Ile drzwi i odbiorników radiowych

Należy zauważyć, że każdy normalny samochód ma więcej niż jedno drzwi. Niektóre są dwu-, a inne czterodrzwiowe. Czasami wyróżnia się nawet pięć drzwi. Analogicznie wcale nie jest powiedziane, że każdy samochód ma radio. Może mieć jedno lub wcale. Raz nawet widziałem samochód z dwoma osobnymi zestawami nagłośnienia. Tego rodzaju sytuacje szczegółowo opiszę w rozdziale 9. Dla uproszczenia przyjmijmy, że samochód ma tylko jedno drzwi (np. wyścigowy) i jedno radio.

Na rysunku 7.8 pokazano model obiektowy samochodu z silnikiem, radiem i drzwiami.



Rysunek 7.8. Hierarchia klasy Car

Należy zauważyć, że wszystkie trzy obiekty składające się na ten samochód same składają się z innych obiektów. Silnik zawiera tłoki i świece zapłonowe. Radio zawiera odbiornik radiowy i odtwarzacz płyt CD. Drzwi mają klamkę. Ponadto jest jeszcze jeden dodatkowy poziom. Radio zawiera urządzenie odbiorcze. Można by było jeszcze zaznaczyć fakt, że klamka ma zamek. CD ma przycisk do szybkiego przewijania w przód itp. Można by też było wyjść o jeden poziom za urządzenie odbiorcze i utworzyć obiekt reprezentujący antenę. Poziom złożoności modelu obiektowego całkowicie zależy od projektującego.

Złożoność modelu

Zbyt intensywne wykorzystanie kompozycji w przypadku dziedziczenia — analogicznie jak w problemie ze szczekającymi i jodłującymi psami — może również doprowadzić do skomplikowania systemu. Granica między takim modelem obiektowym, który jest wystarczająco precyzyjny, aby wszystko było jasne, a takim, w którym zbyt duża szczegółowość zaciemnia obraz, jest trudna do zrozumienia i wycucia.

Czemu hermetyzacja jest podstawą technologii obiektowej

Hermetyzacja naprawdę jest podstawą obiektowości. Każdy opis interfejsu i implementacji dotyczy w rzeczywistości hermetyzacji. Podstawowe pytanie dotyczy tego, co w klasie powinno być udostępnione na zewnątrz, a co ukryte. Kwestia hermetyzacji ma się tak

samo do danych jak i zachowań. W przypadku klas najważniejsza decyzja projektowa dotyczy hermetyzacji zarówno danych, jak i zachowań.

Stephen Gilbert i Bill McCarty definiują hermetyzację jako „pakowanie programu — dzielenie klas na dwie części: interfejs i implementację”. Jest to przesłanie, które zostało już wielokrotnie w tej książce powtórzone.

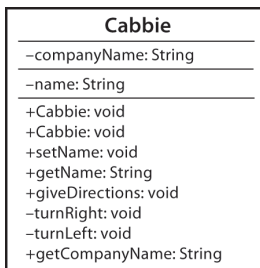
Jak ma się hermetyzacja do dziedziczenia i jaki ma to związek z tematem tego rozdziału? Chodzi o pewien paradoks paradygmatu obiektowego. Hermetyzacja jest tak ważna w obiektowości, że stanowi jedną z jej kardynalnych zasad. Dziedziczenie także jest uznawane za jeden z trzech filarów obiektowości. Jednak dziedziczenie w pewnym sensie łamie zasadę hermetyzacji! Jak to możliwe? Czy dwie najważniejsze koncepcje obiektowości mogą stać ze sobą w sprzeczności? Sprawdźmy to.

Jak dziedziczenie osłabia hermetyzację

Jak wiadomo, hermetyzacja polega na dzieleniu pakietów, jakimi są klasy, na publiczny interfejs i prywatną implementację. Zasadniczo wszystko, co nie jest potrzebne innym klasom, powinno być ukryte.

Peter Coad i Mark Mayfield dowodzą, że dziedziczenie z natury rzeczy osłabia hermetyzację w obrębie hierarchii klas. Zwracają uwagę na specyficzne ryzyko: dziedziczenie implikuje silną hermetyzację między niezwiązanymi klasami, ale słabą między podklasami i nadklasami.

Problem polega na tym, że jeśli podklasy odziedziczą po nadklasie implementację, która zostanie później zmieniona, zmiany te będą widoczne także we wszystkich podklasach. Ten *efekt odbicia* może potencjalnie mieć wpływ na wszystkie podklasy. Na pierwszy rzut oka może się wydawać, że to niewielki problem. Ale wiadomo jednak, że czasami efekty tego mogą być trudne do przewidzenia. Dobrym przykładem jest testowanie, które z tego powodu może zamienić się w koszmar. W rozdziale 6. „Projektowanie z wykorzystaniem obiektów” objaśniłem, w jaki sposób hermetyzacja ułatwia testowanie systemów. Teoretycznie, jeśli klasa *Cabbie* (rysunek 7.9) będzie miała poprawnie zaprojektowane interfejsy publiczne, żadne zmiany ich implementacji nie powinny być widoczne dla innych klas. Jednak zmiana w nadklasie nigdy nie spowoduje takich samych zmian w podklasach. Czy już widać, na czym polega problem?



Rysunek 7.9. Diagram UML klasy Cabbie

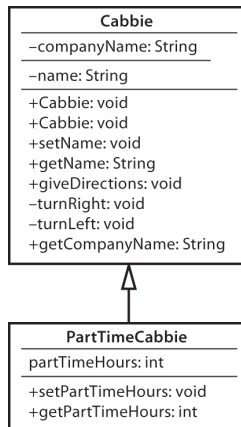
Gdyby inne klasy były bezpośrednio zależne od implementacji klasy `Cabbie`, testowanie stałoby się znacznie trudniejsze, a może nawet niemożliwe.

Pamiętaj o testowaniu

Nawet mimo wykorzystywania hermetyzacji nadal należy testować klasy korzystające z klasy `Cabbie`, aby sprawdzić, czy jej modyfikacje nie wywołały jakichś niepożądanych skutków.

Jeśli później zostanie utworzona podklasa klasy `Cabbie` o nazwie `PartTimeCabbie` (która odziedziczy po niej implementację), zmiana implementacji tej pierwszej będzie miała bezpośredni wpływ na drugą.

Jako przykład niech posłuży diagram UML widoczny na rysunku 7.10. `PartTimeCabbie` to podklasa klasy `Cabbie`. Zatem dziedziczy ona implementację swojej nadklasy, z metodą `giveDirections()` włącznie. Gdyby metoda ta zmieniała się w klasie `Cabbie`, miałoby to bezpośredni wpływ na klasę `PartTimeCabbie` i wszystkie inne jej podklasy. W takim przypadku zmiany implementacji klasy `Cabbie` niekoniecznie muszą się ograniczać tylko do niej.



Rysunek 7.10. Diagram UML klas `Cabbie` i `PartTimeCabbie`

Aby zmniejszyć ryzyko związane z tym zjawiskiem, należy ściśle trzymać się zasady relacji „jest” w dziedziczeniu. Jeśli podklasa rzeczywiście jest wyspecjalizowaną wersją swojej nadklasy, zmiany w tej nadklasie powinny mieć taki wpływ na podklasę, jakiego należałoby się spodziewać i jaki jest naturalny. Rozważmy taki przykład. Jeśli klasa `Circle` dziedziczy implementację po `Shape` i zmiana w implementacji tej drugiej powoduje uszkodzenia pierwszej, należy uznać, że `Circle` nie jest właściwą podklasą klasy `Shape`.

Jak można wykorzystać dziedziczenie w niewłaściwy sposób? Wyobraźmy sobie, że trzeba utworzyć okno na potrzeby graficznego interfejsu użytkownika (GUI). Można by było w pierwszej chwili uznać, że dobrym rozwiązaniem będzie utworzenie takiego okna jako podklasy klasy reprezentującej prostokąt:

```
public class Rectangle {  
}  
public class Window extends Rectangle {  
}
```

W rzeczywistości jednak okno GUI to coś znacznie więcej niż prostokąt. Nie jest to tak naprawdę specjalny rodzaj prostokąta, jak np. kwadrat. Prawdziwe okno może zawierać prostokąt (a nawet wiele prostokątów). Nie jest to jednak prostokąt sam w sobie. W takim przypadku klasa `Window` nie powinna dziedziczyć po `Rectangle`, tylko zawierać obiekty tej klasy.

```
public class Window {  
  
    Rectangle menubar;  
    Rectangle statusbar;  
    Rectangle mainview;  
  
}
```

Szczegółowy przykład wykorzystania polimorfizmu

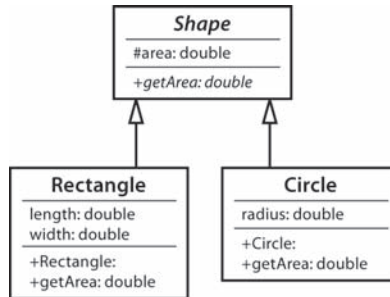
Dla wielu polimorfizm jest podstawą obiektowości. W projektowaniu systemów obiektowych chodzi przede wszystkim o utworzenie całkowicie niezależnych obiektów. W dobrze zaprojektowanym systemie obiekt powinien potrafić odpowiedzieć na wszystkie ważne pytania, które go dotyczą. Z zasady każdy obiekt powinien odpowiadać sam za siebie. Niezależność jest jednym z podstawowych warunków, których spełnienie umożliwia wielokrotne wykorzystanie kodu.

Przypominam z rozdziału 1., że polimorfizm dosłownie oznacza *wiele kształtów*. Jeśli obiekt odbiera komunikaty, musi mieć odpowiednie metody, aby na nie odpowiedzieć. W hierarchii dziedziczenia podklasy *dziedziczą* po swoich nadklasach interfejsy. Ponieważ jednak każda klasa jest w zasadzie samodzielną jednostką, może na komunikaty odpowiadać we właściwy sobie sposób.

Wróć na chwilę do przykładu z rozdziału 1., w którym opisana została klasa `Shape`. Ma ona zachowanie o nazwie `Draw`. Jeśli poprosi się kogoś o narysowanie figury geometrycznej, najprawdopodobniej padnie pytanie, jakiej. Instrukcja, aby narysować figurę geometryczną, jest zbyt abstrakcyjna (w istocie metoda `Draw` w klasie `Shape` nie ma implementacji). Należy więc sprecyzować, o jaką figurę chodzi. W tym celu w klasie `Circle` i innych podobnych powinna znaleźć się odpowiednia implementacja metody `Draw`. Mimo że klasa `Shape` zawiera metodę `Draw`, klasa `Circle` przesyła ją własną wersją. Przesyłanie to, mówiąc najkrócej, zastępowanie implementacji odziedziczonej po nadklasie własnym kodem.

Odpowiedzialność obiektów

Jeszcze raz wróć do przykładu klasy `Shape` z rozdziału 1. (rysunek 7.11).



Rysunek 7.11. Hierarchia klasy Shape

Polimorfizm jest jednym z najbardziej eleganckich sposobów wykorzystania dziedziczenia. Przypomnę, że nie można utworzyć egzemplarza klasy Shape. Jest to klasa abstrakcyjna, co można stwierdzić dzięki obecności w niej abstrakcyjnej metody `getArea()`. Szczegółowy opis klas abstrakcyjnych znajduje się w rozdziale 8.

Natomiast w przypadku klas `Rectangle` i `Circle` tworzenie egzemplarzy jest możliwe, ponieważ są to klasy konkretne. Mimo że klasy `Rectangle` i `Circle` dziedziczą po tej samej klasie Shape, są między nimi pewne oczywiste różnice. Ponieważ są to figury geometryczne, można obliczyć ich pole powierzchni. Ale wzór na to pole dla każdej wygląda inaczej. Dlatego implementacja tej metody nie może znajdować się w klasie Shape.

Jest to idealna sytuacja, w której należy wykorzystać polimorfizm. Polimorfizm polega na tym, że taki sam komunikat można wysłać do wielu różnych obiektów, a każdy z nich odpowie we właściwy sobie sposób. Jeśli na przykład zostanie wysłany komunikat `getArea` do obiektu klasy `Circle`, wykonane obliczenia będą znacznie się różniły od tych, które w odpowiedzi na ten sam komunikat wykonałby obiekt klasy `Rectangle`. Jest to możliwe dzięki temu, że obiekty te same odpowiadają za siebie. Jeśli zażąda się od obiektu `Circle`, aby obliczył swoje pole powierzchni, wykona polecenie. Jeśli zażąda się od tego obiektu, aby się narysował, również to nastąpi. Obiekt klasy Shape nie mógłby wykonać żadnej z tych czynności, nawet gdyby dało się go w ogóle utworzyć, ponieważ miałby zbyt mało informacji o sobie. Należy zauważyć, że nazwa metody `getArea` na diagramie klasy Shape (rysunek 7.11) jest napisana kursywą. To oznacza, że metoda ta jest abstrakcyjna.

Przedstawię bardzo prosty przykład. Są cztery klasy: jedna abstrakcyjna o nazwie Shape i trzy konkretne o nazwach `Circle`, `Rectangle` oraz `Star`. Oto ich kod źródłowy:

```

public abstract class Shape{
    public abstract void draw();
}
public class Circle extends Shape{
    public void draw() {
        System.out.println("Rysuję koło.");
    }
}
public class Rectangle extends Shape{
    public void draw() {
        System.out.println("Rysuję prostokąt.");
    }
}
  
```



```

    }
}
public class Star extends Shape{
    public void draw() {
        System.out.println("Rysuję gwiazdę.");
    }
}

```

Zwracam uwagę, że w każdej z tych klas jest tylko jedna metoda — `draw`. Ilustruje to ważną cechę polimorfizmu: przedstawione wyżej konkretne klasy same odpowiadają za rysowanie swoich obiektów. Klasa `Shape` nie dostarcza żadnego kodu rysującego. Klasy `Circle`, `Rectangle` i `Star` robią to we własnym zakresie. Oto przykładowy fragment kodu, który posłuży za dowód:

```

public class TestShape {

    public static void main(String args[]) {

        Circle circle = new Circle();
        Rectangle rectangle = new Rectangle();
        Star star = new Star();

        circle.draw();
        rectangle.draw();
        star.draw();
    }
}

```

Aplikacja testowa *TestShape* utworzy trzy klasy: `Circle`, `Rectangle` oraz `Star`. Aby narysować obiekty tych klas, `TestShape` wysyła do nich żądanie:

```

circle.draw();
rectangle.draw();
star.draw();

```

Wynik działania programu *TestShape* będzie następujący:

```

C:\>java TestShape
Rysuję koło.
Rysuję prostokąt.
Rysuję gwiazdę.

```

Tak działa polimorfizm. Co trzeba by było zrobić, gdyby wystąpiła konieczność utworzenia nowej klasy, np. `Triangle`? Wystarczy ją napisać, skompilować, przetestować i używać. W klasie bazowej `Shape` ani innych klasach nie trzeba niczego zmieniać:

```

public class Triangle extends Shape{
    public void draw() {
        System.out.println("Rysuję trójkąt.");
    }
}

```

Od tej pory można wysyłać komunikaty do obiektów klasy `Triangle`. Mimo że klasa `Shape` nie jest w stanie narysować trójkąta, `Triangle` zrobi to doskonale:

```
public class TestShape {
    public static void main(String args[]) {
        Circle circle = new Circle();
        Rectangle rectangle = new Rectangle();
        Star star = new Star();
        Triangle triangle = new Triangle ();

        circle.draw();
        rectangle.draw();
        star.draw();
        triangle.draw();
    }
}
```

```
C:\>java TestShape
Rysuję koło.
Rysuję prostokąt.
Rysuję gwiazdę.
Rysuję trójkąt.
```

Aby przekonać się o prawdziwej mocy polimorfizmu, można obiekt figury przekazać do metody, która absolutnie „nie ma pojęcia”, co to jest. Spójrzmy na poniższy kod zawierający informacje o konkretnych kształtach w parametrach.

```
public class TestShape {
    public static void main(String args[]) {
        Circle circle = new Circle();
        Rectangle rectangle = new Rectangle();
        Star star = new Star();

        drawMe(circle);
        drawMe(rectangle);
        drawMe(star);
    }

    static void drawMe(Shape s) {
        s.draw();
    }
}
```

W tym przypadku obiekt klasy Shape jest przekazywany metodzie drawMe, która potrafi obsłużyć każdy poprawny obiekt tego typu — nawet taki, który zostanie dodany później. Tę wersję programu *TestShape* można uruchomić w taki sam sposób jak poprzednią.

Klasy abstrakcyjne, metody wirtualne i protokoły

Klasy abstrakcyjne, takie jak zdefiniowane w języku Java, można także zaimplementować w językach .NET i C++. Nie jest zaskoczeniem, że kod źródłowy w języku C# wygląda podobnie do kodu Javy, o czym można się przekonać, patrząc na poniższy przykład.

```
public abstract class Shape
{
```

```
    Konstrukcja podobna do interfejsów w Javie, zwanych protokołami (opisane dalej).
    public abstract void draw();
```

```
}
```

Kod w języku Visual Basic .NET wygląda tak:

```
Public MustInherit Class Shape
```

```
    Public MustOverride Function draw()
```

```
End Class
```

Taką samą funkcjonalność można uzyskać w języku C++ przy użyciu metod wirtualnych:

```
class Shape
```

```
{
```

```
    public:
```

```
        virtual void draw() = 0;
```

```
}
```

Jak pisałem w poprzednich rozdziałach, język Objective-C nie implementuje w pełni klas abstrakcyjnych.

Weźmy na przykład pokazywany już kilka razy kod klasy Shape w języku Java:

```
public abstract class Shape{
```

```
    public abstract void draw();
```

```
}
```

Poniżej przedstawiony jest będący odpowiednikiem tego kodu protokół w języku Objective-C. Zwróć uwagę, że w obu przypadkach nie została zaimplementowana metoda draw():

```
@protocol Shape
```

```
@required
- (void) draw;
```

```
@end // Shape
```

W przedstawionych do tej pory przykładach funkcjonalność klas abstrakcyjnych pokrywa się z funkcjonalnością protokołów. Poniżej natomiast przedstawiam przykład, w którym interfejsy Javy i protokoły Objective-C znacząco od siebie odbiegają. Spójrzmy na poniższy kod w języku Java:

```
public abstract class Shape{

    public abstract void draw();
    public void print() {
        System.out.println("Drukuje");
    }
}
```

W tym kodzie metoda `print()` może zostać odziedziczona przez podklasę klasy `Shape`. Podobnie jest w językach C# .NET, VB .NET oraz C++, ale nie w przypadku protokołu języka Objective-C, który mógłby wyglądać tak:

```
@protocol Shape

@required
- (void) draw;
- (void) print;

@end // Shape
```

W tym protokole została podana sygnatura metody `print()`, co oznacza, że w podklasie musi znaleźć się jej implementacja. Nie ma jednak żadnego kodu implementacyjnego w protokole. Krótko mówiąc, podklasy nie mogą niczego dziedziczyć po protokołach. Dlatego protokoły nie są tym samym co klasy abstrakcyjne i ma to wpływ na proces projektowania modelu obiektowego.

Podsumowanie

W rozdziale tym przedstawiłem podstawowe wiadomości na temat dziedziczenia i kompozycji oraz opisałem dzielące je różnice. Wielu uznanych specjalistów od obiektowości twierdzi, że kompozycję należy wykorzystywać wszędzie, gdzie to możliwe, a dziedziczenie tylko wówczas, gdy jest to konieczne.

Jest to jednak pewne uproszczenie. Moim zdaniem twierdzenie, że kompozycji należy używać gdzie to tylko możliwe, ma drugie dno. Po prostu kompozycja ma więcej zastosowań niż dziedziczenie. To, że jest więcej przypadków, w których kompozycja jest najlepszym rozwiązaniem, nie oznacza, że dziedziczenie jest złe. Należy stosować obie techniki, ale w odpowiednich sytuacjach.

W dotychczasowych rozdziałach kilkakrotnie poruszany był temat klas abstrakcyjnych i interfejsów Javy. W rozdziale 8. opiszę pojęcie kontraktu w programowaniu oraz wyjaśnię, jak klasy abstrakcyjne i interfejsy służą do wywiązywania się z kontraktów.

Źródła

Steven Holzner, *Visual Quickstart Guide, Objective-C*, Peachpit Press, Berkeley 2010.

Booch, Grady i in., *Object-Oriented Analysis and Design with Applications*, 3rd ed., Addison-Wesley, Boston 2007.

Scott Meyers, *Effective C++*, 3rd ed., Addison-Wesley Professional, Boston 2005.

Peter Coad i Mark Mayfield, *Java Design*, Prentice-Hall, Upper Saddle River 1997.

Stephen Gilbert i Bill McCarty, *Object-Oriented Design in Java*, The Waite Group Press, Berkeley 1998.

Listingi

Poniżej przedstawiam kod źródłowy w języku C# .NET. Przykłady w innych językach, takich jak VB.NET i Objective-C, są dostępne na serwerze FTP wydawnictwa. Listingi te są odpowiednikami przykładów kodu w języku Java, które przedstawiono w tekście rozdziału.

TestShape

```
using System;

namespace TestShape
{
    public class TestShape
    {
        public static void Main()
        {
            Circle circle = new Circle();
            Rectangle rectangle = new Rectangle();

            circle.draw();
            rectangle.draw();
        }
    }

    public abstract class Shape
    {
        public abstract void draw();
    }

    public class Circle : Shape
    {
        public override void draw()
```

```
        {  
            Console.WriteLine("Rysuję koło.");  
        }  
    }  
  
    public class Rectangle : Shape  
    {  
        public override void draw()  
        {  
            Console.WriteLine("Rysuję prostokąt.");  
        }  
    }  
  
    public class Star : Shape  
    {  
        public override void draw()  
        {  
            Console.WriteLine("Rysuję gwiazdę.");  
        }  
    }  
  
    public class Triangle : Shape  
    {  
        public override void draw()  
        {  
            Console.WriteLine("Rysuję trójkąt.");  
        }  
    }  
}
```

Skorowidz

A

abstrakcja, 42, 47, 236
abstrakcyjny interfejs, 60
adres
 IP, 268
 URL, 252
agregacja, 136, 179, 181, 196
agregat, 136
akcesory, 30
analizator składni, 208
animacje Flash, 252
antywzorce, 291
API, application
 programming interface,
 55, 152
aplety, 28
aplikacje
 hybrydowe, 23, 241
 klient-serwer, 265
 przepływ danych, 266
 rozwiązanie
 własnościowe, 266
 uruchamianie, 270, 276
 XML, 271
 mobilne, 241
arkusze stylów, 216
asocjacja, 136, 180, 197
asocjacje opcjonalne, 186
atrybuty, 29, 35, 192
 inicjowanie, 92
 klasowe, 81
 lokalne, 78

 obiektywne, 79, 90
 prywatne, 38
 publiczne, 38
 statyczne, 90, 95, 105

B

baza danych
 Oracle, 204
 SQL Server, 204
bazy danych
 obiektywne, 112
 relacyjne, 112
bezpieczeństwo, 118
bezpieczeństwo klientów,
 244
biznes elektroniczny, 163
blok try-catch, 76
błędy, 74, 166

C

catch, 76
COM, 130
CORBA, Common Object
 Request Broker
 Architecture, 254
 elementy systemu, 257
CSS, cascading style sheets,
 216
cykl życia obiektu, 226
czarna skrzynka, 24

D

dane
 globalne, 25, 107
 historyczne, 200
 obiekty, 29
 prywatne, 38
 przenośne, 203
DCOM, 258
decyzje projektowe, 136
definicja
 atrybutu XML, 232
 klasy, 33
 obiekty, 28
 typu dokumentu, 206
 XML obiektu, 272
definiowanie wymagań, 121
destrukторы
 projektowanie, 102
diagram UML, 32, 36, 189
 hierarchii klas, 43
 interfejsu, 157
 klasy, 31, 190
 Cabbie, 139, 190
 DataBaseReader, 56, 71
 Dog, 131
 Person, 37
reprezentujący
 asocjacje, 198
 kompozycję, 196
 systemu Shop, 167
 wzorca Singleton, 284

dokument

- DTD, 209
- HTML, 244
 - z arkuszem stylów, 222
 - z obiektem JSON, 220
- opisujący system, 119
- SOW, 119
- wymagań, 120
- XML, 206, 210
- dokumentacja, 103
- dokumentacja API, 152
- dostęp do relacyjnej bazy
 - danych, 235
- drukowanie, 108
- drzewo
 - dziedziczenia, 42
 - obiektów JavaScript, 249
- DTD, Document Type
 - Definition, 206, 209
- dziedziczenie, inheritance,
 - 41, 47, 129, 131, 133, 194
 - abstrakcja, 42
 - decyzje projektowe, 134
 - definicji, 158
 - generalizacja, 133
 - implementacji, 84, 158
 - pojedyncze, 42
 - specjalizacja, 133
 - wielokrotne, 42, 83, 156, 194
 - zachowań, 84

E

- edytor
 - formatu JSON, 219
 - w3schools, 221
- egzemplarz, instantiation, 32
- egzemplarze klasy, 32
- elementy składowe wzorca, 281
- Enterprise JavaBeans, 258

F

- faza analizy, 119
- Financial products Markup
 - Language, 204
- Flash, 252
- format
 - JSON, 219
 - XML, 271
- formatowanie
 - danych, 214
 - elementów, 216
- framework, 150
- funkcje, 29

G

- generowanie dokumentacji, 89
- graficzny interfejs
 - użytkownika, GUI, 52
- gromadzenie wymagań, 120
- GUI, graphical user
 - interface, 52

H

- hermetyzacja, encapsulation,
 - 26, 37, 48, 138
 - dziedziczenie, 139
 - odpowiedzialność
 - obiektów, 141
 - polimorfizm, 141
- hierarchia
 - agregacji, 180
 - klasy, 41, 43
 - abstrakcyjnej, 154
 - Car, 138
 - Dog, 133
 - Shape, 142
- HTML, Hypertext Markup
 - Language, 205, 214, 245

I

- IDE, 120
- identyfikacja
 - implementacji, 64
 - interfejsów publicznych,
 - 63, 100
 - klas, 120
 - użytkownika, 54
- identyfikator abstract, 44
- IDL, Interface Definition
 - Language, 255
- IIOP, Internet Inter-ORB
 - Protocol, 257
- implementacja, 32, 38, 52, 229
 - interfejsu, 39
 - kontraktu, 153
 - prywatna, 40
 - sklepu, 168
- infrastruktura
 - programistyczna, 150
- inicjalizacja atrybutów, 69
- integracja DTD i XML, 210
- interfejs, 37, 52, 59, 156, 195, 229
 - Externalizable, 227
 - IDL, 255
 - ISerializable, 113
 - MailInterface, 289
 - Nameable, 161, 167
 - Serializable, 113, 227
- interfejsy
 - abstrakcyjne, 60
 - klasy, 38, 54
 - minimalizacja, 57, 61
 - określanie grupy
 - docelowej, 62
 - programistyczne, 152
 - projektowanie, 59
 - publiczne, 40, 56, 63, 100
- iteracja, 110

J

JDBC, 235
 język
 C, 16, 52
 C# .NET, 16, 30, 203
 C++, 16
 COBOL, 52
 FORTRAN, 52
 HTML, 205
 Java, 16, 246
 JavaScript, 217, 245
 Objective-C, 17
 SGML, 205
 Smalltalk, 16, 241, 280
 SQL, 235
 UML, 15, 36, 116
 VB .NET, 16
 Visual Basic, 16
 XML, 29, 203–205
 języki
 modelowania, 16
 obiektywne, 207
 programowania, 16
 skryptowe, 242
 znaczników, 204
 JSON, JavaScript Object Notation, 218

K

kaskadowe arkusze stylów, 216
 klasa, 35, 84
 Circle, 155
 Invoice, 262
 JMenuBar, 152
 Person, 238
 Shape, 142
 Shop, 169
 TestShape, 147
 TestShop, 171
 klasy
 abstrakcyjne, 44, 145, 153, 158
 atrybuty, 35, 81

definicja, 33
 definiowanie wymagań, 121
 diagramy UML, 32
 dziedziczenie, 40, 47, 131
 hermetyzacja, 37
 implementacja interfejsu, 39, 54
 interfejs, 38, 54
 kompozycja, 47
 konstruktor domyślny, 69
 konstruktory, 91
 ładowanie dynamiczne, 59
 metody dostępne, 93
 metody statyczne, 105
 modelowanie, 36, 71
 nadrzędne, 42
 nazwa, 87
 o wysokim stopniu sprzężenia, 109
 ograniczanie zakresu, 107
 określanie dostępu, 35
 polimorfizm, 44
 projektowanie, 26
 referencje, 84
 rozszerzalność, 105
 rozszerzanie interfejsu, 101
 ukrywanie danych, 37
 ukrywanie implementacji, 101
 wytyczne dotyczące projektowania, 99, 115
 klient, 267, 273
 kod
 JSON, 219
 klienta, 267, 273
 nieprzenośny, 106, 125
 obiektu do serializacji, 266
 pośredni, 56
 serwera, 269, 274
 strukturalny, 122, 124, 154
 usług sieciowych, 261
 komentarze, 89, 103
 kompozycja, composition, 47, 129, 136, 196
 agregacja, 179
 asocjacja, 180
 diagramy UML, 137
 unikanie zależności, 182
 komunikacja
 klient-serwer, 266, 272
 między obiektami, 27, 36
 komunikaty, 36
 konkatenacja łańcuchów, 82
 konkretyzacja, 32
 konserwacja kodu, 109
 konstruktory, 45, 67, 70, 91
 projektowanie, 73, 102
 wywoływanie, 68
 zawartość, 68
 kontrakt, 152, 260
 implementacja, 153
 interfejsy, 156
 klasy abstrakcyjne, 153
 zawieranie, 161
 kontrola dostępu do danych, 26
 kontroler, Controller, 281
 kontrolki, 250
 konwencja nazewnicza, 106
 kopiowanie
 głębokie, 84
 obiektów, 84, 107
 płytkie, 84

L

liczba obiektów, 183
 liczność, cardinality, 183, 199
 asocjacji klas, 184
 na diagramie, 185
 LIFO, last-in, first-out, 46

Ł

łączenie łańcuchów, 82

M

mapery relacyjno-
obiektywne, 127

mechanizm usuwania
nieużytków, 103

metadane, 33

metoda

get (), 93

getArea(), 44

set (), 93

open(), 58

metody, 29, 36, 192

abstrakcyjne, 44, 155

dostępowe, accessor, 30,

93, 232

interfejs, 30

interfejsu publicznego,
95

pobierające, getter, 29,
93

prywatne, 95

przeciążanie, 70

przesłanianie, 44

przydzielanie pamięci, 94

statyczne, 105

sygnatura, 70

ustawiające, setter, 29,
93

wirtualne, 145

współdzielone, 78

Microsoft COM, 130

minimalizacja interfejsu, 57,
61, 100

model, 135, 281

kaskadowy, 116, 117

klas opisujący system,
121

klient-serwer, 236, 243,
265

obiekty, 116

obiekty UML, 167

relacyjny, 234

wzorca singleton, 284

modelowanie

klas, 36, 71

obiektywne, 189

systemów świata

rzeczywistego, 99

modyfikator dostępu

internal, 194

private, 35, 193

protected, 35, 193

public, 35, 193

mutator, 30

MVC,

Model-View-Controller, 280

myślenie abstrakcyjne, 59

N

nadklasy, 42, 73

najlepsze praktyki, 279

namiastki, stub, 110

narzędzia do modelowania,
32

nazwa klasy, 87

nazwy, 105

notacja

obiektyowa, 217

UML, 189

O

obiekt singletonowy, 286

obiektyowa baza danych, 24

obiektywne skryptowe języki

programowania, 242

obiektyowy paradygmat

myślenia, 15

programowania, 52

obiekty, 24, 28, 35, 94, 203,

241, 255

atrybuty, 29, 79

cykl życia, 226

dane, 29

definicja, 28

JavaScript, 248

komunikacja, 36

kopiowanie, 84

liczność, 183

na stronach

internetowych, 248

nadające się do

kooperacji, 104

opakowujące, 122

operacje, 84

osłonowe, 23, 255

porównywanie, 84

rozproszone, 252

serializacja, 113

sieciowe, 28

szeregowanie, 113

trwałość, 57, 112, 225

w aplikacjach

klient-serwer, 265

hybrydowych, 241

mobilnych, 241

w usługach sieciowych,
241

wielokrotne

wykorzystanie, 129

zachowania, 29, 63

złożone, 136

obsługa błędów

ignorowanie problemu,
74

mieszanie technik, 75

precyzja

przechwytywania

wyjątków, 77

projektowanie

mechanizmu, 103

wyszukiwanie, 75, 166

zgłaszanie wyjątków, 76

ODBC, Open Database

Connectivity, 236

odpowiedzialność obiektów,
141

odtworzacze

dźwięku, 250

filmów, 251

ograniczanie
 liczności, 199
 zakresu, 107
 środowiska, 63

określanie
 dostępności, 193
 grupy docelowej, 62
 zakresu planowanych
 prac, 119

opakowywanie
 istniejących klas, 126
 kodu strukturalnego, 124
 nieprzenośnego kodu,
 125

operacje obiektów, 84

oprogramowanie
 pośredniczące, 254

ORB, Object Request
 Broker, 256

organizacja W3C, 205

P

paradygmat obiektowy, 16

parser, 208

pasek menu, 152

PCDATA, Parsed Character
 Data, 210

pionowe przenoszenie
 danych, 204

platforma .NET, 30

plik
 Invoice.xsd, 260
 mwssoap.xml, 260

pliki
 płaskie, 226
 wsadowe, 271, 287

plug-and-play, 150

podklasy, 42

podobiekty, 84

podprocedury, 29

polimorfizm, 44, 48, 141,
 142

ponowne kompilowanie
 kodu, 59

poprawność
 dokumentów, 206
 dokumentów XML, 212,
 214
 dokumentu względem
 DTD, 208

porównywanie obiektów, 84,
 107

powtórzenia, 123

prezentowanie danych, 214

procedura inicjująca, 69

procedury, 24, 29

program
 do odbierania poczty,
 289
 ORB, 256
 porządkowy, 69
 RestorePerson, 230
 XML Notepad, 212

programowanie
 obiektowe, 24, 28, 116
 proceduralne, 27, 99
 strukturalne, 25
 powtórzenia, 123
 sekwencyjność, 123
 warunki, 123

projektowanie, 25, 28, 115,
 199

analizy, 119

definiowanie wymagań,
 121

destruktorów, 102

dokument SOW, 119

gromadzenie wymagań,
 120

identyfikacja klas, 120

interfejsów, 59

klas, 26

konstruktorów, 73, 102

mechanizmu
 obsługi błędów, 103

model kaskadowy, 116

model klas opisujący
 system, 121

prototyp interfejsu
 użytkownika, 120, 121

szybkie prototypowanie,
 117

współpraca między
 klasami, 121

wytyczne, 115

zakres planowanych
 prac, 119

protokół, 145
 IIOP, 257
 SOAP, 257

prototyp interfejsu
 użytkownika, 120

prywatne metody, 95

przechowywanie danych
 historycznych, 200

przechwytywanie wyjątku,
 76, 77

przeciążanie
 metody, method
 overloading, 70
 operatorów, 82

przekazywanie referencji, 91

przenośność danych, 204

przepływ danych między
 klientem a serwerem, 266

przesłanianie, overriding, 44

przesyłanie danych, 223

przetwarzanie rozproszone,
 241, 252
 CORBA, 254
 DCOM, 258
 ReST, 263
 RPC, 258
 SOAP, 258, 259
 usługi sieciowe, 257

przewaga nad konkurencją,
 119

przdzielanie pamięci
 metodom, 94
 obiektom, 90

punkty dostępne do
 systemu, 163

R

RecipeML, 205
 referencje, 84, 286
 relacje kompozycji, 175
 relacyjne bazy danych, 225, 233
 JDBC, 235
 model klient-serwer, 236
 ODBC, 236
 SQL, 235
 zapisywanie danych, 233
 ReST, Representational State Transfer, 263
 RMI, remote method invocation, 258
 rodzaje
 interfejsów, 38
 kompozycji, 179
 wzorców projektowych, 283
 rozproszone działanie, 254
 rozszerzalność, 105
 rozszerzalny język znaczników, 205
 rozszerzanie interfejsu, 101
 RPC, Remote Procedure Call, 258

S

samodzielne oprogramowanie, 58
 sekwencyjność, 123
 serializacja, serialization, 113, 225
 język XML, 231
 metod, 231
 obiektu, 230
 pliku, 227
 serwer, 269, 274
 serwer WWW, 243
 serwis w3schools, 214

SGML, Standard Generalized Markup Language, 205
 sieć
 komórkowa, 23
 mobilna, 23
 składnia języka Java, 89
 słownictwo, vocabulary, 204
 słowo kluczowe
 class, 88
 init, 67
 interface, 56, 157
 new, 73
 New, 67
 private, 90
 static, 90
 this, 81
 SOAP, Simple Object Access Protocol, 257
 solidny artefakt, 292
 SOW, statement of work, 119
 specyfikacja formatowania elementów, 216
 sprawdzanie poprawności dokumentów XML, 214
 SQL, Structured Query Language, 235
 standaryzacja, 150
 stos
 wstawienie, push, 46
 zdejmnianie, pop, 46
 struktura diagramu klasy, 190
 strumień, 227
 sygnatura, 70
 system
 informatyczny przedsiębiorstwa, 252
 oprogramowania, 177
 plików płaskich, 112
 trzywarstwowy, 255
 wielowarstwowy, 254

szeregowanie, marshaling, 113
 szukanie błędów, 75
 szybkie prototypowanie, 117

Ś

śledzenie referencji, 85

T

tablica asocjacyjna, 219
 technika plug-and-play, 150
 techniki przetwarzania rozproszonego, 241
 testowanie
 interfejsu, 110
 klasy, 140
 kodu, 118, 132
 transfer danych między aplikacjami, 207
 transmisja danych przez sieć, 28
 trwałość obiektów, 57, 112, 225
 tworzenie
 diagramów klas, 32
 egzemplarza, 32, 46
 klas, 47
 konstruktora, 69
 modeli obiektowych, 189
 modelu klas opisującego system, 121
 obiektów, 32, 45, 67, 73, 159, 175–188
 prototypu interfejsu użytkownika, 121
 systemu, 177
 typ zwrotny, 72
 typy danych, 25, 33

U

ukrywanie
 danych, data hiding, 26,
 37, 90
 implementacji, 101, 107
 UML, Unified Modeling
 Language, 32, 116, 189
 agregacja, 137, 196
 asocjacja, 137, 197
 atrybuty, 192
 dziedziczenie, 194
 dziedziczenie
 interfejsów, 194
 dziedziczenie
 wielokrotne, 194
 interfejsy, 195
 kompozycja, 137, 196
 liczność, 199
 metody, 192
 określanie dostępności,
 193
 uruchamianie aplikacji, 270
 usługi sieciowe, 257
 usługi sieciowe ReSTful, 263
 usuwanie nieużytków, 103
 użytkownik, 54

W

walidator, 214
 wartość null, 91, 186
 warunki, 123
 weryfikacja
 danych, 244–246
 dokumentu, 215
 widok, View, 281
 wielokrotne
 dziedziczenie, 83, 156,
 194
 wykorzystanie kodu, 40,
 104, 149, 151
 wykorzystanie obiektów,
 129

własności, 30
 wskaźnik this, 248
 współpraca między klasami,
 121
 wyciek pamięci, 103
 wyjątek, exception, 76
 wykrywanie błędów, 166
 wymagania, 120
 wymiana danych, 208
 wytyczne dotyczące
 projektowania, 115
 wywołanie
 konstruktora, 68
 metody, 29
 wyznaczanie trasy, 257
 wzorce czynnościowe
 Interpretator, 290
 Iterator, 290
 Łańcuch
 Odpowiedzialności,
 290
 Mediator, 290
 Metoda Szablonowa, 290
 Obserwator, 290
 Pamiętka, 290
 Polecenie, 290
 Stan, 290
 Strategia, 290
 wzorce konstrukcyjne
 Budowniczy, 283
 Fabryka Abstrakcyjna,
 283
 Metoda Fabrykująca, 283
 Prototyp, 283
 Singleton, 283
 wzorce projektowe, 279
 konsekwencje, 281
 nazwa, 281
 problem, 281
 rozwiązanie, 281
 wzorce strukturalne
 Adapter, 288
 Dekorator, 288
 Fasada, 288

Kompozyt, 288
 Most, 288
 Pośrednik, 288
 Waga Piórkowa, 288
 wzorzec MVC, 280, 282

X

XML, Extensible Markup
 Language, 204–207, 214,
 271
 dokumenty, 206
 integracja z DTD, 210
 PCDATA, 210
 poprawność dokumentu,
 210
 serializacja, 231
 specyfikacja
 danych do wymiany,
 208
 formatowania
 elementów, 216
 sprawdzanie
 poprawności
 dokumentu, 208

Z

zachowania obiektu, 29, 63
 zadania
 klienta, 273
 serwera, 274
 zakres, scope, 78, 107
 zakres planowanych prac,
 119
 zapętlenie lokalne, 268
 zapisywanie
 liczności, 184
 obiektu, 229
 obiektu w pliku płaskim,
 226
 zastosowanie obiektów, 241
 zaślepki, 110, 112

zawieranie kontraktu, 149, 161	zintegrowane środowisko programistyczne, 120	związek
zaznaczanie zakresu, 78	złożoność modelu, 135, 138	dziedziczenia, 176
zdalne wywoływanie	znak	typu „jest”, 42, 129, 159
procedur, 258	-, 39	typu „ma”, 47, 130
zestawy, assembly, 203	+, 39	

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



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

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

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



Twoje kompendium wiedzy o programowaniu obiektowym!

Obiektowe podejście do programowania pojawiło się w latach 60. ubiegłego wieku. Simula 67 był pierwszym językiem, w którym je zastosowano. Dzięki temu życie programistów stało się zdecydowanie prostsze, a odwzorowanie świata rzeczywistego — możliwe. Jednak żeby skorzystać z zalet podejścia obiektowego, należy najpierw opanować nowy sposób myślenia.

Kolejne wydanie tej docenionej przez profesjonalistów książki szybko Ci w tym pomoże! W trakcie lektury poznasz podstawowe pojęcia oraz założenia programowania obiektowego. Dowiesz się, co to hermetyzacja, polimorfizm oraz dziedziczenie. Zobaczysz, jak obiekty powoływane są do życia oraz jak komunikują się między sobą. Ponadto nauczysz się korzystać z interfejsów, modelować klasy z wykorzystaniem diagramów UML oraz utrzymywać stan obiektów. To wydanie zostało uzupełnione o mnóstwo nowych informacji, dotyczących między innymi wykorzystania obiektów w usługach sieciowych oraz aplikacjach mobilnych. Książka ta jest obowiązkową lekturą dla każdego programisty chcącego w 100% wykorzystać potencjał programowania obiektowego.

Sięgnij po tę książkę i:

- modeluj klasy przy użyciu UML
- swobodnie poruszaj się w świecie klas, interfejsów i obiektów
- utrwalaj stan swoich obiektów poprzez serializację
- korzystaj z obiektów w komunikacji sieciowej
- zostań ekspertem w zakresie programowania obiektowego

helion.pl
księgarnia
internetowa

Nr katalogowy: 16328

Księgarnia internetowa
<http://helion.pl>

Zamówienia telefoniczne:
0 801 339900
0 601 339900

Informatyka w najlepszym wydaniu



Helion

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

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

sięgnij po WIĘCEJ



KOD KORZYŚCI

Addison
Wesley

cena: 49,90 zł

ISBN 978-83-246-8120-4



9 788324 681204