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.

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.

Spis treści

7

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 wykorzystać 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.

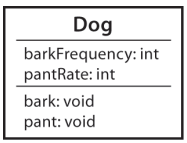
Dziedziczenie

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.

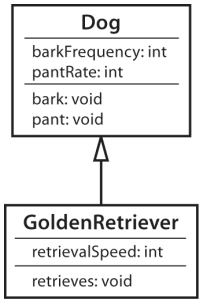
> **Związek typu „jest"**
>
> Jedna z najważniejszych zasad projektowania obiektowego głosi, że interfejs publiczny powinien być reprezentowany przez związek typu „jest".

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

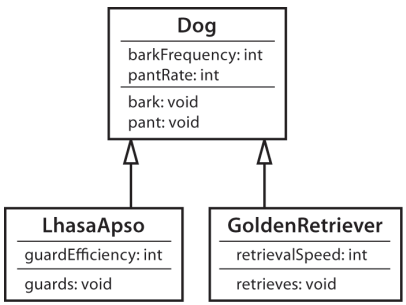
Teraz utworzymy klasę `GoldenRetriever`. Można by było utworzyć całkiem nową klasę z takimi samymi zachowaniami jak w klasie `Dog`. Można jednak też dojść do następującego rozsądnego wniosku: golden retriever to **jest** pies. Można zatem wykorzystać dziedziczenie i utworzyć klasę `GoldenRetriever` jako 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.

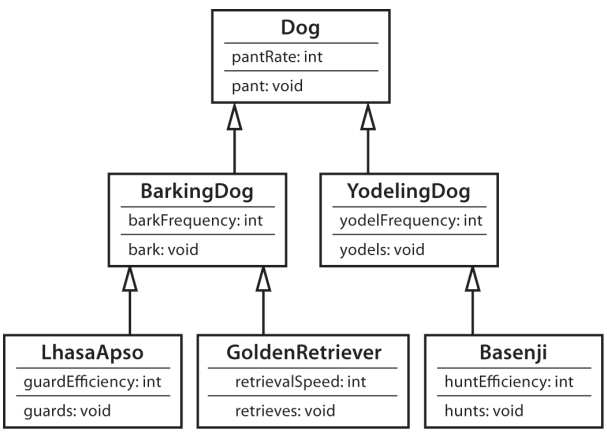
Kolejną zaletą dziedziczenia jest to, że kod metod `bark()` i `pant()` znajduje się tylko w jednym miejscu. Załóżmy, że trzeba zmodyfikować kod pierwszej z nich. Wówczas wystarczy tylko zmiana w klasie `Dog`, a będzie ona od razu przejęta przez klasy `LhasaApso` i `GoldenRetriever`.

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 kaczy 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 rozbicie 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.

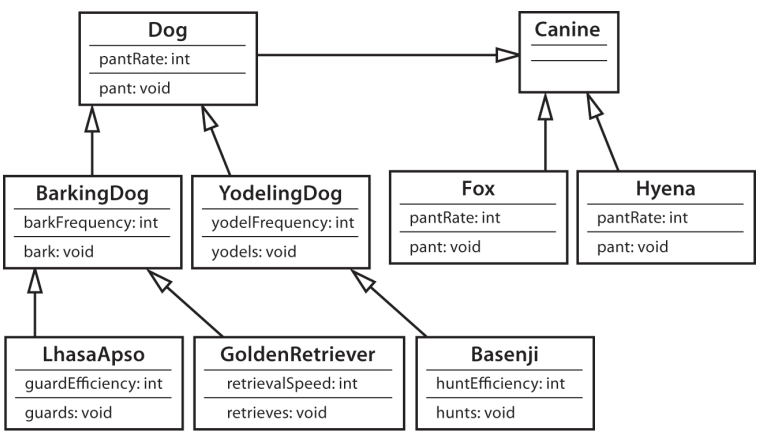
Złożoność modelu

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.

Podejmuj 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 koleją rzeczy, 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).

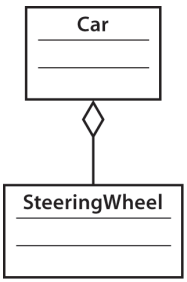
Rysunek 7.6. Przykład kompozycji

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.

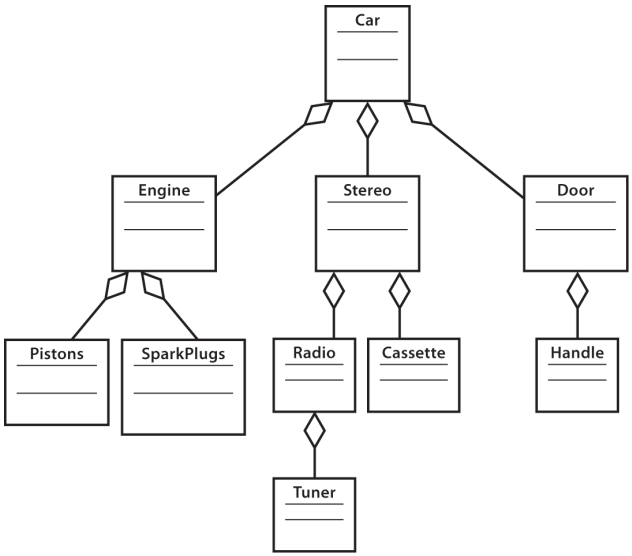
Załóżmy, że samochód składa się z silnika, radia i drzwi.

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 jedne drzwi. Niektóre są dwu-, a inne czterodrzwiowe. Czasami wyróżnia się nawet piąte 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 przyjmiemy, że samochód ma tylko jedne 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 zbytnia szczegółowość zaciemnia obraz, jest trudna do zrozumienia i wyczucia.

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.

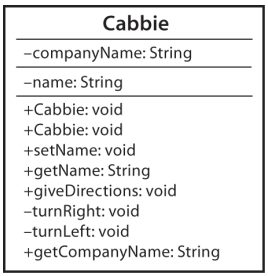
Jak ma się hermetyzacja do dziedziczenia i jaki ma to związek z tematem tego rozdziału? Chodzi o pewien paradoks paradygmatu obiektowego. Hermetyzacja jest tak ważna w obiektowości, że stanowi jedną z jej kardynalnych zasad. Dziedziczenie także jest uznawane za jeden z trzech filarów obiektowości. Jednak 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?



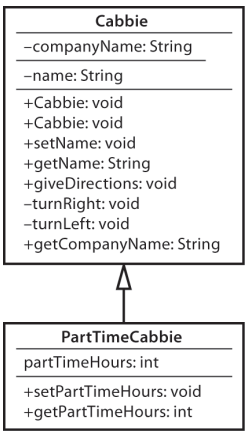
Rysunek 7.9. Diagram UML klasy Cabbie

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 zmieniła 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

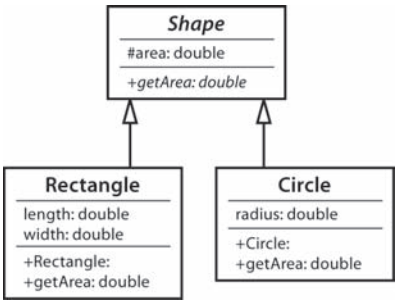
Dla wielu polimorfizm jest podstawą obiektowości. W projektowaniu systemów obiektowych chodzi przede wszystkim o utworzenie całkowicie niezależnych obiektów. W dobrze zaprojektowanym systemie obiekt powinien potrafić odpowiedzieć na wszystkie ważne pytania, które go dotyczą. Z zasady każdy obiekt powinien 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.

Wrócę 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` przesłania ją własną wersją. Przesłanianie to, mówiąc najkrócej, zastępowanie implementacji odziedziczonej po nadklasie własnym kodem.

Odpowiedzialność obiektów

Jeszcze raz wrócę 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("Drukuję");
    };
}
```

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

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

W

X

Z

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 utrwalać 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

Addison Wesley

cena: 49,90 zł

ISBN 978-83-246-8120-4

9 788324 681204