Matt Weisfeld

Myślenie obiektowe w programowaniu

Wydanie V

Tytuł oryginału: The Object-Oriented Thought Process (5th Edition)

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-283-6104-1

Helion SA
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/myobp5
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/myobp5.zip

Printed in Poland.

Spis treści

11

Jak uniknąć zależności i silnych powiązań między klasami

Jak napisałem w rozdziale 1. zawierającym wprowadzenie do pojęć obiektowości, programowanie obiektowe tradycyjnie wyróżnia zastosowanie takich technik jak hermetyzacja, dziedziczenie i polimorfizm. Aby język programowania teoretycznie można było nazwać obiektowym, musi spełniać warunki tych trzech technik, chociaż lubię jeszcze do nich dodawać kompozycję, którą także opisałem w rozdziale 1.

W związku z tym podstawowy zestaw pojęć, jaki przedstawiam na zajęciach z programowania obiektowego, przedstawia się następująco:

- Hermetyzacja
- Dziedziczenie
- Polimorfizm
- Kompozycja

Wskazówka

Do listy tej można by jeszcze dodać interfejsy, ale uważam je za specjalny rodzaj dziedziczenia.

Dziś ze względu na toczącą się w środowisku programistów dyskusję na temat poprawnego posługiwania się dziedziczeniem obecność kompozycji na tej liście wydaje się jeszcze ważniejsza. Wątpliwości wokół dziedziczenia nie są żadną nowością, ale w ostatnich latach debata na jego temat nabrała rumieńców. Wielu programistów, z którymi rozmawiam, opowiada się za kompozycją zamiast dziedziczenia. Niektórzy nawet starają się go w ogóle unikać albo przynajmniej ograniczyć je do jednego poziomu hierarchii.

Dyskusje dotyczące sposobów posługiwania się dziedziczeniem krążą wokół tematu powiązań. Argumenty za nim to przede wszystkim możliwość wielokrotnego wykorzystania kodu, rozszerzalność i polimorfizm. Należy też jednak pogodzić się z możliwością powstania zależności między klasami, czyli powiązaniem klas. Zależności te sprawiają problemy zarówno podczas konserwacji, jak i testowania kodu.

W rozdziale 7. „Dziedziczenie i kompozycja" opisałem, jak przez dziedziczenie może dojść do osłabienia hermetyzacji, co wydaje się bez sensu, ponieważ są to dwa podstawowe pojęcia. A jednak na tym polega zabawa i zmusza nas to do dokładnego zastanowienia się nad tym, w jaki sposób posługiwać się dziedziczeniem.

> **Ostrzeżenie**
>
> Podkreślę, że nie jestem przeciwnikiem posługiwania się dziedziczeniem. W rozdziale tym opisuję metody unikania zależności i ścisłych powiązań między klasami, a prawidłowe posługiwanie się dziedziczeniem stanowi ważny element tej układanki.

Powyższe rozważania nieuchronnie prowadzą do następującego pytania: jeśli nie dziedziczenie, to co? Odpowiedź, że kompozycja, nikogo nie zdziwi, ponieważ od początku tej książki podkreślam, że są tylko dwa sposoby na wielokrotne wykorzystanie klas — dziedziczenie i kompozycja właśnie. Programista może utworzyć klasę potomną za pomocą dziedziczenia lub może umieścić jedną klasę w innej klasie, korzystając z technik kompozycji.

Jeśli zgodnie z zaleceniami niektórych programistów dziedziczenia należy unikać, to po co w ogóle marnować czas na naukę posługiwania się nim? Odpowiedź jest prosta: technika ta jest wykorzystywana w wielu programach. Programiści bardzo szybko się przekonują, że ogromna większość istniejącego kodu wymaga poprawiania. Aby naprawiać, ulepszać i doskonalić kod napisany przy użyciu technik dziedziczenia, należy te techniki rozumieć. Czasami nawet pisze się własny kod z ich wykorzystaniem. Krótko mówiąc, programista musi znać wszystkie podstawowe techniki oraz powinien dysponować pełnym wachlarzem narzędzi programistycznych. To jednak zmusza do ciągłego poznawania nowych narzędzi i zmiany sposobu myślenia o tych, które już znamy.

Jeszcze raz podkreślę, że nie chodzi mi o jakiekolwiek osądzanie pod względem wartości. Nie twierdzę, że dziedziczenie sprawia problemy i że należy go unikać. Chcę tylko zaznaczyć, jak ważne jest dokładne zrozumienie dziedziczenia, zapoznanie się z innymi metodami projektowania oraz wybranie najbardziej odpowiedniej dla siebie. Dlatego też przykłady prezentowane w tym rozdziale niekoniecznie przedstawiają optymalny sposób zaprojektowania klas. To tylko ćwiczenia, które mają zachęcić do przemyślenia kwestii związanych z wyborem między dziedziczeniem i kompozycją. Pamiętajmy, że tak ewoluują wszystkie technologie — zachowują to, co w nich dobre, i doskonalą to, co ma wady.

Ponadto kompozycja także nie jest wolna od problemów dotyczących powiązań między klasami. W rozdziale 7. opisałem różne typy kompozycji: asocjację i agregację. Agregacja to obiekty umieszczone w innych obiektach (utworzonych za pomocą słowa kluczowego `new`), natomiast asocjacja to obiekty przekazywane do innych obiektów za pośrednictwem listy parametrów. Skoro agregacje są osadzone w obiektach, to znaczy, że są z nimi ściśle powiązane, a tego przecież chcemy uniknąć.

Dlatego, choć dziedziczenie może mieć opinię techniki sprzyjającej silnemu wiązaniu klas, kompozycja (przy użyciu agregacji) także może sprawiać takie same problemy. Wrócimy do przykładu ze sprzętem stereo z rozdziału 9. „Tworzenie obiektów i projektowanie obiektowe" i na jego podstawie połączymy wszystkie te koncepcje w jedną całość.

Tworzenie zestawu stereo przy użyciu agregacji można porównać do tworzenia szafy grającej, która jest sprzętem zawierającym wszystkie komponenty wewnątrz jednej obudowy. Rozwiązanie takie w wielu sytuacjach jest bardzo wygodne. Szafę można podnieść, przesunąć

i nie wymaga ona specjalnych umiejętności przy składaniu. Z drugiej strony taka konstrukcja może też nastręczać liczne problemy. Jeśli popsuje się któryś z komponentów, na przykład odtwarzacz MP3, to do naprawy trzeba oddać całe urządzenie. Co gorsza, może zdarzyć się tak, że jakaś awaria, na przykład zasilania, zniszczy całą szafę grającą za jednym zamachem.

Wiele z problemów wynikających z użycia agregacji można zlikwidować, posługując się asocjacją. Wyobraź sobie, że system stereo jest zestawem asocjacji połączonych przewodami (albo bezprzewodowo). W takiej konstrukcji występuje centralny obiekt zwany odbiornikiem, z którym łączą się inne obiekty, takie jak głośniki, odtwarzacze CD czy nawet konsole do miksowania i magnetofony. Można nawet pokusić się o stwierdzenie, że to rozwiązanie niezależne od dostawcy, ponieważ każdy komponent możemy kupić osobno, co jest wielką zaletą tego rozwiązania.

Jeśli w takim zestawie popsuje się odtwarzacz CD, wystarczy go odłączyć i oddać do naprawy (jednocześnie nadal korzystając z pozostałych części) lub wymienić na nowy. Taka jest zaleta asocjacji i ograniczenia do minimum powiązań między klasami.

> **Wskazówka**
>
> Jak napisałem w rozdziale 9., choć generalnie ścisłe powiązania między klasami są niemile widziane, czasami można zaakceptować tę niedogodność. Jednym z przykładów takiej sytuacji jest opisana wcześniej szafa grająca. Mimo ścisłego powiązania jej komponentów taka konstrukcja ma pewne zalety.

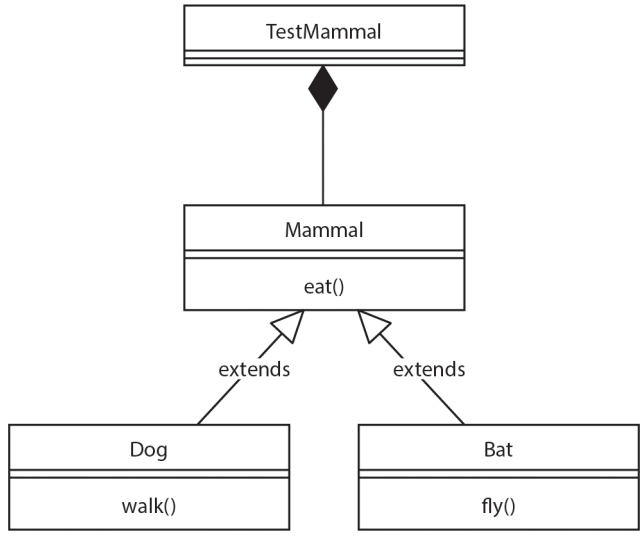
Wiemy już, jakie problemy sprawiają ścisłe powiązania komponentów zarówno w przypadku dziedziczenia, jak i kompozycji, więc teraz przyjrzymy się paru przykładom ściśle powiązanych projektów utworzonych przy użyciu tych technik. Na wzór zajęć, które prowadzę, będziemy stopniowo przebudowywać te przykłady, aż dojdziemy do techniki zwanej wstrzykiwaniem zależności, która pozwoli nam ograniczyć problemy związane z powiązaniem komponentów.

Kompozycja a dziedziczenie i wstrzykiwanie zależności

Na początek zastanowimy się, jak przeprojektować model dziedziczenia (typowy dla przykładów często używanych w tej książce) przy użyciu kompozycji. Drugi przykład pokazuje sposób zmiany projektu z wykorzystaniem kompozycji, choć przy użyciu agregacji, która nie zawsze jest optymalnym rozwiązaniem. Trzeci przykład pokazuje, jak uniknąć agregacji i zamiast nich zaprojektować system przy użyciu asocjacji — przedstawia zatem koncepcję **wstrzykiwania zależności**.

1. Dziedziczenie

Niezależnie od tego, czy uznajesz wyższość kompozycji nad dziedziczeniem, na początek przeanalizujemy prosty przykład dziedziczenia oraz sprawdzimy, jak zaimplementować go przy użyciu kompozycji, po drodze przypominając sobie przykład dotyczący ssaków, który towarzyszy nam od początku książki. Tym razem wprowadzimy nietoperza, czyli latającego ssaka — rysunek 11.1.



Rysunek 11.1. Tworzenie obiektów ssaków przy użyciu dziedziczenia

W tym konkretnym przykładzie dziedziczenie wydaje się oczywistym wyborem. Utworzenie klasy `Dog`, która dziedziczy po klasie `Mammal`, to doskonały pomysł, prawda? Spójrz na poniższy kod, w którym właśnie w taki sposób wykorzystano dziedziczenie:

```
class Mammal {
    public void eat () {System.out.println("Jem");};
}
class Bat extends Mammal {
    public void fly () {System.out.println("Latam");};
}
class Dog extends Mammal {
    public void walk () {System.out.println("Chodzę");};
}

public class TestMammal {

    public static void main(String args[]) {

        System.out.println("Kompozycja ponad dziedziczeniem");;

        System.out.println("\nDog");
        Dog fido = new Dog();
        fido.eat();
        fido.walk();
        System.out.println("\nBat");
        Bat brown = new Bat();
        brown.eat();
        brown.fly();
    }
}
```

W tym projekcie klasa `Mammal` ma jedno zachowanie — `eat()` — wynikające z założenia, że wszystkie ssaki muszą jeść. Jednak już po dodaniu dwóch podklas, `Bat` i `Dog`, zaczynamy dostrzegać problem. Choć psy potrafią chodzić, nie wszystkie ssaki mają tę umiejętność.

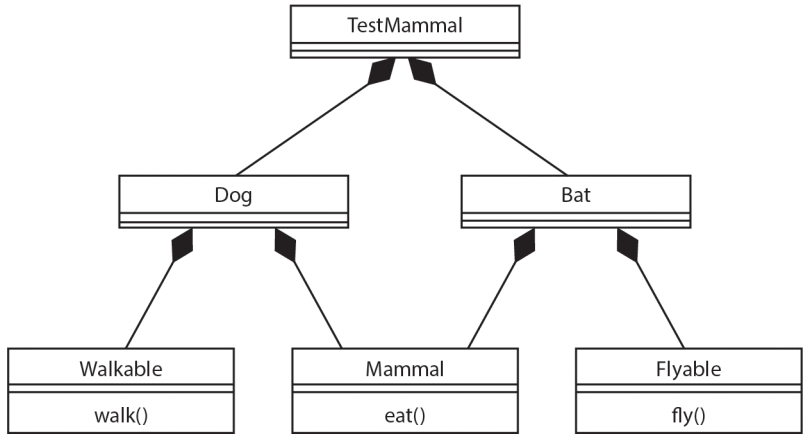
Z drugiej strony nietoperze latają, ale to też jest umiejętność tylko wybranych ssaków. Gdzie w takim razie umieścić metody odpowiadające tym zdolnościom? Podobnie jak w przypadku wcześniejszego przykładu z pingwinami, fakt, że nie wszystkie ptaki potrafią latać, sprawia, że w hierarchii dziedziczenia trudno jest znaleźć odpowiednie miejsce do umieszczenia niektórych metod.

Podział klasy `Mammal` na klasy `FlyingMammals` i `WalkingMammals` jest nieeleganckim rozwiązaniem, ponieważ to tylko wierzchołek góry lodowej. Niektóre ssaki potrafią pływać, a część z nich nawet składa jaja. Poza tym poszczególne gatunki ssaków wyróżniają się przeróżnymi wyjątkowymi zdolnościami i tworzenie osobnej klasy dla wszystkich tych zachowań byłoby wyjątkowo niepraktyczne. Dlatego w tym przypadku zamiast posługiwać się relację typu „jest", lepiej postawić na relację „ma".

2. Kompozycja

W tej strategii zamiast umieszczać zachowania w samych klasach, dla każdego zachowania tworzymy osobną klasę. W ten sposób zamiast umieszczać zachowania w hierarchii dziedziczenia, możemy utworzyć klasę dla każdego zachowania, po czym możemy tworzyć poszczególne ssaki przy użyciu odpowiednich dla nich umiejętności (poprzez agregację).

W związku z tym tworzymy klasy `Walkable` i `Flyable`, jak widać na rysunku 11.2.



Rysunek 11.2. Tworzenie ssaków za pomocą kompozycji

Spójrz na poniższy kod. Nadal mamy klasę `Mammal` z metodą `eat()` oraz nadal mamy klasy `Dog` i `Bat`. Główna różnica projektowa dotyczy tego, że klasy `Dog` i `Bat` uzyskują swoje zachowania przy użyciu kompozycji, poprzez agregację.

> **Ostrzeżenie**
>
> Pamiętaj, że w poprzednim akapicie wystąpiło pojęcie agregacji. Ten przykład ilustruje sposób użycia kompozycji zamiast dziedziczenia, ale w postaci agregacji, która także jest obciążona silnymi powiązaniami. Dlatego należy to traktować jako etap pośredni na drodze do następnego przykładu z wykorzystaniem interfejsów.

```
class Mammal {
    public void eat () {System.out.println("Jem");};
}
class Walkable {
    public void walk () {System.out.println("Chodzę");};
}
class Flyable {
    public void fly () {System.out.println("Latam");};
}
class Dog {
    Mammal dog = new Mammal();
    Walkable walker = new Walkable();
}
class Bat {
    Mammal bat = new Mammal();
    Flyable flyer = new Flyable();
}

public class TestMammal {

    public static void main(String args[]) {

        System.out.println("Kompozycja ponad dziedziczeniem");;
        System.out.println("\nDog");;
        Dog fido = new Dog();
        fido.dog.eat();
        fido.walker.walk();

        System.out.println("\nBat");;
        Bat brown = new Bat();
        brown.bat.eat();
        brown.flyer.fly();
    }
}
```

> **Uwaga**
>
> Celem tego przykładu jest pokazanie, jak zastąpić dziedziczenie kompozycją. To nie znaczy, że z dziedziczenia należy całkowicie zrezygnować. Jeśli stwierdzisz, że absolutnie wszystkie ssaki muszą jeść, to metodę `eat()` możesz umieścić w klasie `Mammal`, po której będą dziedziczyć klasy `Dog` i `Bat`. To typowa decyzja projektanta.

W tym przypadku najistotniejsze jest chyba to, o czym pisałem już wcześniej, że dziedziczenie narusza hermetyzację. Łatwo to zrozumieć, ponieważ zmiana w klasie `Mammal` wymagałaby ponownej kompilacji (i zapewne nawet nowego wdrożenia) wszystkich jej podklas. To oznacza, że klasy są silnie powiązane, co stoi w sprzeczności z naszym celem ograniczenia do minimum powiązań między klasami.

Gdybyśmy w naszym przykładzie kompozycji chcieli dodać klasę `Whale` (wieloryb), to żadna z poprzednich klas nie wymagałaby przepisania. Wystarczyłoby dodać klasy `Swimmable` i `Whale`. Następnie klasy `Swimmable` moglibyśmy też użyć do budowy na przykład klasy `Dolphin`.

```
class Swimmable {
    public void fly () {System.out.println("Pływam");};
}

class Whale {
    Mammal whale = new Mammal();
    Walkable swimmer = new Swimmable ();
}
```

Funkcjonalność tę można dodać do klasy głównej bez wprowadzania jakichkolwiek zmian we wcześniejszych klasach.

```
System.out.println("\nWhale");
Whale shamu = new Whale();
shamu.whale.eat();
shamu.swimmer.swim();
```

Jedną z podstawowych zasad jest korzystanie z dziedziczenia tylko w przypadkach prawdziwego polimorfizmu, a więc technika ta może świetnie sprawdzić się dla kół i prostokątów dziedziczących po klasie `Shape`. Z drugiej strony wybór dziedziczenia dla takich zachowań jak chodzenie i latanie może być nie najlepszy, ponieważ ich przesłanianie może sprawiać trudności. Gdybyśmy na przykład w klasie `Dog` mieli przesłonić metodę `fly()`, to jedynym sensownym wyjściem byłoby, aby nic ona nie robiła. Podobnie jak we wcześniejszym przykładzie z pingwinem, nie chcielibyśmy przecież, aby nasz pies skoczył z urwiska, po czym wykonał dostępną metodę `fly()` tylko po to, by ku swemu przerażeniu stwierdzić, że nic ona nie robi.

Choć w tej implementacji wykorzystaliśmy kompozycję, projekt ten ma jedną poważną wadę. Obiekty są ściśle powiązane, ponieważ użycie słowa kluczowego `new` jest tu oczywiste.

```
class Whale {
    Mammal whale = new Mammal();
    Walkable swimmer = new Swimmable ();
}
```

Aby naprawdę doprowadzić do rozdzielenia klas, wprowadzimy pojęcie **wstrzykiwania zależności**. Krótko mówiąc, zamiast tworzyć obiekty w innych obiektach, będziemy wstrzykiwać obiekty z zewnątrz za pośrednictwem list parametrów. Opis dotyczy wyłącznie koncepcji wstrzykiwania zależności.

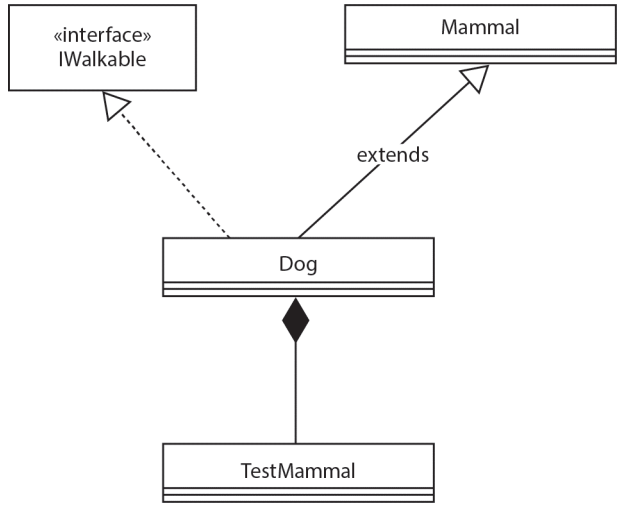
Wstrzykiwanie zależności

W poprzednim przykładzie posłużyliśmy się kompozycją (z agregacją), aby nadać psu (`Dog`) umiejętność chodzenia (`Walkable`). Klasa `Dog` dosłownie tworzyła nowy obiekt `Walkable` w swoim wnętrzu, jak ilustruje poniższy fragment kodu:

```
class Dog {
    Walkable walker = new Walkable();
}
```

Choć to wprawdzie działa, taka klasa jest ściśle powiązana z inną klasą. Aby pozbyć się tego powiązania z poprzedniego przykładu, posłużymy się wspominaną już wcześniej koncepcją wstrzykiwania zależności. Technice tej zazwyczaj towarzyszy pojęcie odwrócenia kontroli (ang. *inversion of control* — IOC). Polega ono na tym, że przerzucamy zadanie utworzenia obiektu na kogoś innego, po czym każemy sobie ten obiekt przekazać. Właśnie taki sposób działania zaimplementujemy teraz w tym przykładzie.

Ponieważ nie wszystkie ssaki chodzą, latają lub pływają, proces likwidacji powiązań rozpoczniemy od utworzenia interfejsów reprezentujących umiejętności różnych gatunków ssaków. W przykładzie skupiam się na umiejętności chodzenia, którą będzie reprezentować interfejs `IWalkable` (rysunek 11.3).



Rysunek 11.3. Tworzenie obiektów ssaków przy użyciu interfejsów

Kod źródłowy interfejsu `IWalkable` wygląda następująco:

```
interface IWalkable {
    public void walk();
}
```

Zawiera on tylko jedną metodę o nazwie `walk()`, której implementacja powinna się znaleźć w implementacji konkretnej klasy.

```
class Dog extends Mammal implements IWalkable{
    Walkable walker;
    public void setWalker (Walkable w) {
        this.walker=w;
    }
    public void walk () {System.out.println("Chodzę");};
}
```

Klasa `Dog` rozszerza klasę `Mammal` i implementuje interfejs `IWalkable`. Ponadto klasa `Dog` zawiera referencję i konstruktor umożliwiający wstrzykiwanie zależności.

```
Walkable walker;
public void setWalker (Walkable w) {
    this.walker=w;
}
```

Na tym w skrócie polega wstrzykiwanie zależności. Zachowanie `Walkable` nie zostało utworzone wewnątrz klasy `Dog` za pomocą słowa kluczowego `new`, tylko wstrzyknięto je przy użyciu listy parametrów.

Oto kompletny przykład:

```
class Mammal {
    public void eat () {System.out.println("Jem");};
}
interface IWalkable {
    public void walk();
}
```

```
class Dog extends Mammal implements IWalkable{
    Walkable walker;
    public void setWalker (Walkable w) {
        this.walker=w;
    }
    public void walk () {System.out.println("Chodzę");};
}
public class TestMammal {
    public static void main(String args[]) {
        System.out.println("Kompozycja ponad dziedziczeniem");
        System.out.println("\nDog");
        Walkable walker = new Walkable();
        Dog fido = new Dog();
        fido.setWalker(walker);
        fido.eat();
        fido.walker.walk();
    }
}
```

W tym przykładzie wstrzyknięcie zostało zrealizowane za pomocą konstruktora, ale nie jest to jedyny sposób na zastosowanie tej techniki.

Wstrzykiwanie przez konstruktor

Jednym ze sposobów na *wstrzyknięcie* zachowania `Walkable` jest utworzenie konstruktora w klasie `Dog`, który będzie przyjmował argument z aplikacji głównej:

```
class Dog {
    Walkable walker;
    public Dog (Walkable w) {
        this.walker=w;
    }
}
```

W tym przypadku aplikacja tworzy obiekt `Walkable` i wstawia go do `Dog` za pomocą konstruktora.

```
Walkable walker = new Walkable();
Dog fido = new Dog(walker);
```

Wstrzykiwanie za pomocą metody ustawiającej

Choć konstruktor *inicjalizuje* atrybuty w chwili tworzenia obiektu, w czasie istnienia obiektu często występuje potrzeba zresetowania wartości. Do tego wykorzystuje się metody ustawiające, czyli settery. Zachowanie `Walkable` można *wstawić* do klasy `Dog` za pomocą metody ustawiającej o nazwie `setWalker()`:

```
class Dog {
    Walkable walker;
    public void setWalker (Walkable w) {
        this.walker=w;
    }
}
```

Podobnie jak w przypadku techniki z wykorzystaniem konstruktora, aplikacja tworzy obiekt `Walkable`. Następnie obiekt ten jest wstawiany do klasy `Dog` za pomocą settera:

```
Walkable walker = new Walkable();
Dog fido = new Dog();
fido.setWalker(walker);
```

Podsumowanie

Wstrzykiwanie zależności oddziela proces tworzenia obiektu klasy od jej zależności. To czynność bardziej przypominająca branie rzeczy z półki (*od dostawcy*) niż samodzielne ich tworzenie za każdym razem.

Jest to kluczowy element dyskusji dotyczącej dziedziczenia i kompozycji, choć należy podkreślić, że to tylko dyskusja. W rozdziale tym moim celem nie było przedstawienie „optymalnego" sposobu projektowania klas, lecz zachęcenie Cię do przemyślenia kwestii związanych z podejmowaniem decyzji podczas wyboru między dziedziczeniem i kompozycją. W następnym rozdziale opisuję zasady SOLID projektowania obiektowego, które są powszechnie akceptowane i cenione w środowiskach programistów.

Źródła

Robert C. Martin, *Zwinne wytwarzanie oprogramowania. Najlepsze zasady, wzorce i praktyki*, Helion, Gliwice 2015.

Robert C. Martin, *Czysty kod. Podręcznik dobrego programisty*, Helion, Gliwice 2009.

Techniki obiektowe. Zrozum, zanim zaimplementujesz!

Burzliwy rozwój obiektowości nastąpił w połowie lat 90. wraz z upowszechnieniem się takich języków jak C++ czy Smalltalk. Obecnie programowanie zorientowane obiektowo wciąż jest jednym z najważniejszych paradygmatów projektowania. Łatwo zauważyć, że większość nowoczesnych języków programowania i technologii sieciowych opiera się na technikach obiektowych. Nieco trudniej jest dostrzec, że mimo szybkiego ewoluowania technik i języków programistycznych podstawowe pojęcia programowania obiektowego pozostają niezmienne i niezależne od jakiejkolwiek platformy. Początkujący programiści powinni więc poświęcić nieco czasu na zapoznanie się z tymi pojęciami i naukę czegoś, co można nazwać „myśleniem obiektowym w programowaniu".

Ta książka jest kolejnym, poprawionym i uzupełnionym wydaniem wyczerpującego wprowadzenia do programowania zorientowanego obiektowo. Jej głównym celem jest przedstawienie podstaw myślenia obiektowego i najważniejszych pojęć w tym zakresie. Wyjaśniono tu, w jaki sposób poprawnie posługiwać się dziedziczeniem i kompozycją, odróżniać agregację od asocjacji oraz zrozumieć różnice między interfejsem a implementacją. Szczególną uwagę zwrócono na technologie, które przetrwały próbę czasu ostatnich 20 lat i stały się rdzeniem koncepcji programowania obiektowego. Opisano też najważniejsze wzorce projektowe, wskazano techniki unikania zależności i zaprezentowano zasady zwane SOLID, których przestrzeganie pozwala tworzyć kod wysokiej jakości, zrozumiały i elastyczny.

Dzięki tej książce:

- 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

Matt Weisfeld jest wykładowcą, programistą i autorem książek. Niegdyś próbował też swoich sił w biznesie. Od ponad 20 lat jest związany zawodowo z branżą informatyczną oraz z zarządzaniem. Napisał kilka książek o programowaniu i opublikował szereg artykułów w wielu prestiżowych periodykach. Mieszka w Cleveland w stanie Ohio.

ISBN 978-83-283-6104-1
9 788328 361041
Cena: 59,00 zł
Addison Wesley
ALWAYS LEARNING
PEARSON