O'REILLY®

Wydanie II

Head First

Wzorce projektowe

Rusz głową!

Tworzenie rozszerzalnego i łatwego w utrzymaniu oprogramowania obiektowego

Eric Freeman
Elisabeth Robson

Helion

Tytuł oryginału: Head First Design Patterns: Building Extensible and Maintainable
Object-Oriented Software, 2nd Edition

Tłumaczenie: Piotr Rajca na podstawie „Wzorce projektowe. Rusz głową!”
w tłumaczeniu Grzegorza Kowalczyka i Pawła Koronkiewicza

ISBN: 978-83-283-7875-9

Helion S.A.
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)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
https://ftp.helion.pl/przyklady/wzorg2.zip

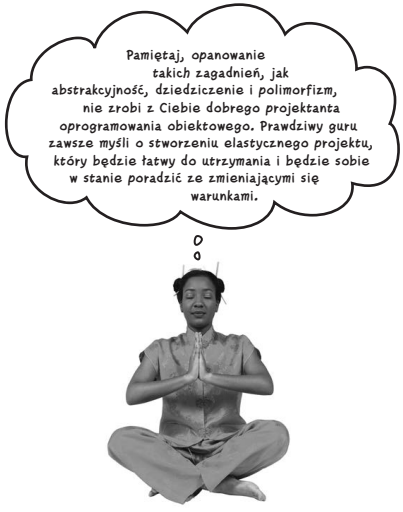
Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/wzorg2
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

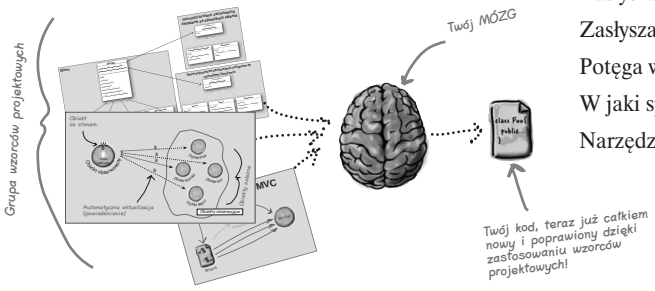
Printed in Poland.

Wprowadzenie do wzorców projektowych

1 Witamy w krainie wzorców projektowych

Ktoś rozwiązał już Twoje problemy. W tym rozdziale dowiesz się, dlaczego (i w jaki sposób) możesz wykorzystać wiedzę i doświadczenia zdobyte przez innych programistów, którzy podążali tą samą ścieżką projektową i — co najważniejsze — udało im się przeżyć taką wyprawę. Zanim dobrniemy do końca rozdziału, rzucimy okiem na sposoby wykorzystywania wzorców projektowych i przedstawimy ich zalety, poznamy kilka podstawowych zasad projektowania obiektowego, a także omówimy sposób działania przykładowego wzorca. Najlepszą metodą zastosowania wzorca jest *załadowanie go bezpośrednio do Twojego mózgu*, a następnie *zlokalizowanie obszarów* w obrębie projektowanych rozwiązań oraz istniejących aplikacji, w których możesz *je zastosować*. Pracując z wzorcami projektowymi, zamiast wielokrotnego wykorzystywania *kodu* wielokrotnie wykorzystujesz swoje *doświadczenia*.

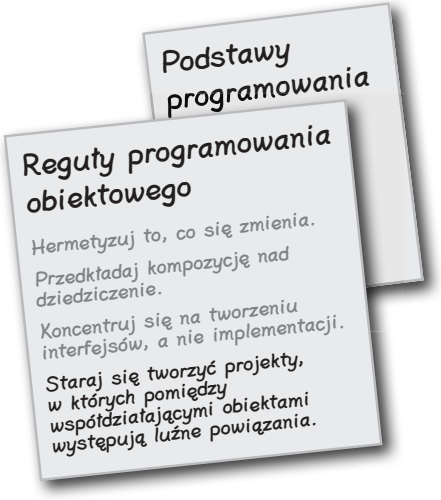


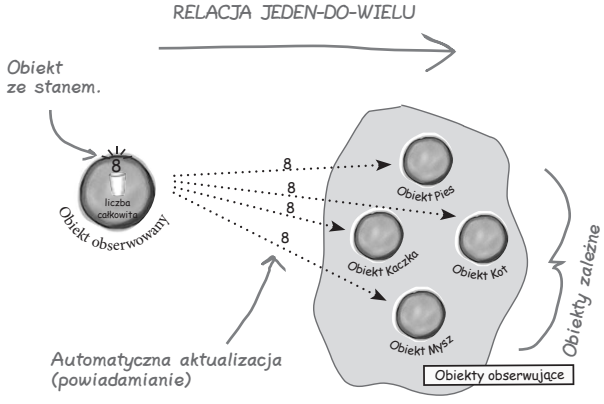


Wzorzec Obserwator

2 Jak sprawić, by Twoje obiekty były zawsze dobrze poinformowane

Nie chcesz przegapić żadnego momentu, kiedy dzieje się coś naprawdę ciekawego, prawda? Istnieje pewien wzorzec, który potrafi *poinformować* inne obiekty o tym, że wydarzyło się coś, co je *interesuje* — to wzorzec Obserwator (ang. *Observer*). Jest on jednym z najczęściej używanych wzorców projektowych i jest wręcz niewiarygodnie użyteczny. W niniejszym rozdziale przyjrzymy się wszystkim interesującym aspektom tego wzorca, takim jak *relacje typu jeden-do-wielu* oraz *luźne powiązania*. A kiedy już poznasz te wszystkie zagadnienia, będziemy się mogli zastanowić, w jaki sposób wzorzec Obserwator może nam ułatwić życie w kontekście organizacji spotkania grupy użytkowników wzorców projektowych.

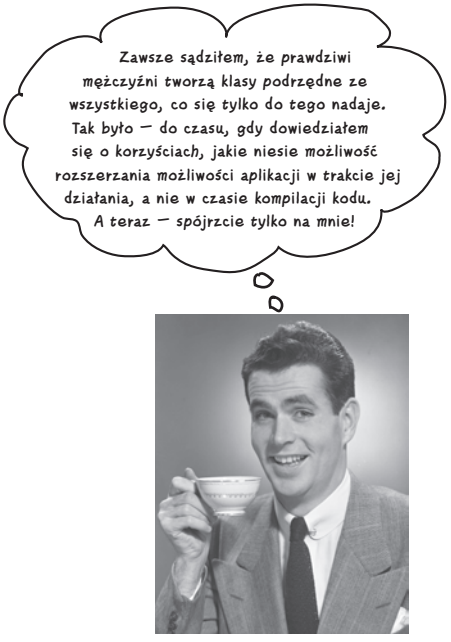




Wzorzec Dekorator

3 Dekorowanie obiektów

W zasadzie niniejszy rozdział moglibyśmy zatytułować „Otwieranie oczu programistom zapatrzonym w dziedziczenie". W tym rozdziale spróbujemy się krytycznie przyjrzeć zwyczajowym skłonnościom do nadużywania mechanizmu dziedziczenia oraz nauczymy Cię sposobów dekorowania klas w czasie działania programu przy użyciu pewnej formy kompozycji obiektów. Dlaczego? Po zapoznaniu się z technikami dekorowania klas będziesz mógł wyposażać swoje (i nie tylko) obiekty w nowe możliwości *bez konieczności wprowadzania jakichkolwiek modyfikacji w kodzie używanych klas.*

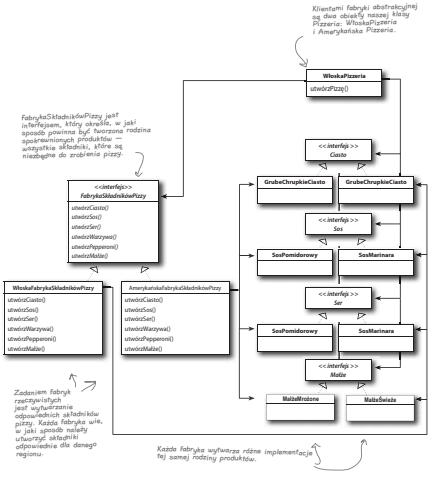


4

Wzorzec Fabryka

Wypieki obiektowe

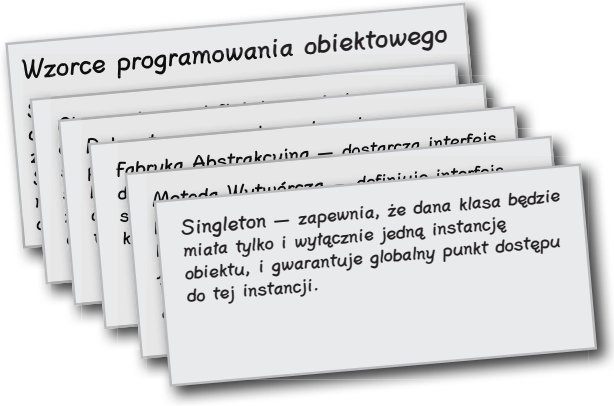
Przygotuj się do stworzenia kilku projektów, w których zastosujemy luźne powiązania pomiędzy poszczególnymi obiektami. Stworzenie nowego obiektu to dużo więcej niż tylko proste zastosowanie operatora **new**. Niebawem przekonasz się, że proces ten jest operacją, która nie zawsze powinna być publicznie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



Wzorzec Singleton

5 Obiekty jedyne w swoim rodzaju

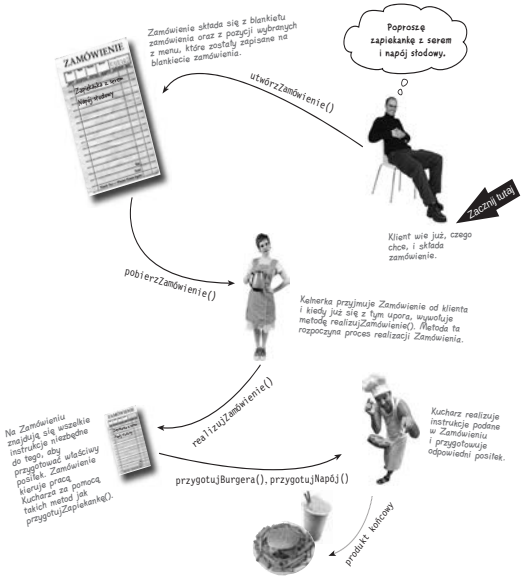
Kolejnym przystankiem w naszej podróży jest wzorzec Singleton, czyli nasza przepustka do tworzenia jedynych w swoim rodzaju obiektów, posiadających tylko jedną instancję. Być może ucieszysz się na wieść o tym, że Singleton jest najprostszym z istniejących wzorców projektowych (przynajmniej pod względem stopnia złożoności jego diagramu klas); jak by na to nie patrzeć, jego diagram składa się tylko z jednej klasy! Ale nie wpadaj w euforię; niezależnie od prostoty diagramu klas tego wzorca na drodze prowadzącej do jego implementacji napotkamy całkiem sporo wybojów i dziur, więc lepiej zapnij mocno pasy!



Wzorzec Polecenie

6 Hermetyzacja wywołań

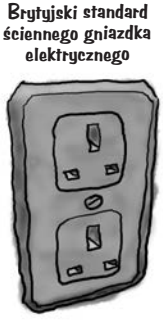
W niniejszym rozdziale przeniesiemy hermetyzację na zupełnie nowy poziom: mamy zamiar hermetyzować wywołania metod. Zgadza się, dzięki hermetyzacji wywołań metod możemy wykrystalizować pewne fragmenty obliczeń tak, by obiekt wywołujący obliczenia nie musiał się martwić, w jaki sposób je wykonać; po prostu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

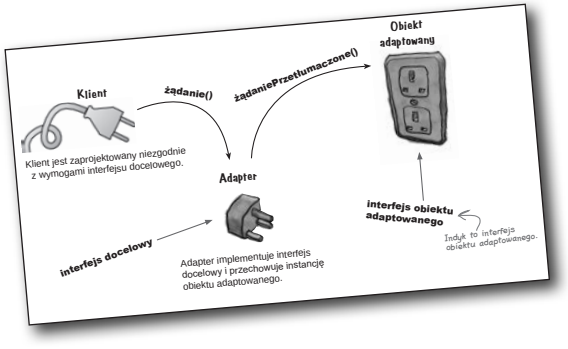


Wzorce Adapter i Fasada

7 Zdolność adaptacji

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

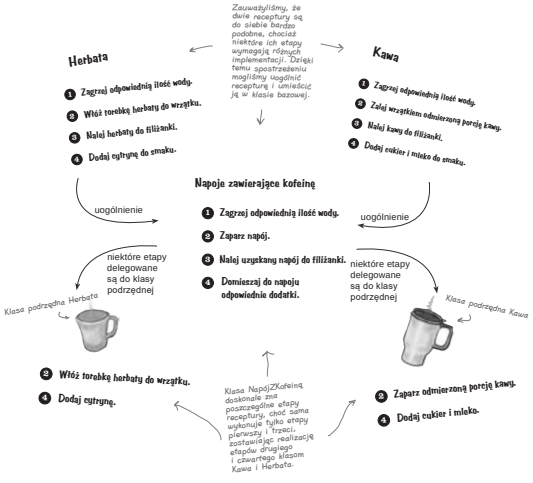




Wzorzec Metoda Szablonowa

8 Hermetyzacja algorytmów

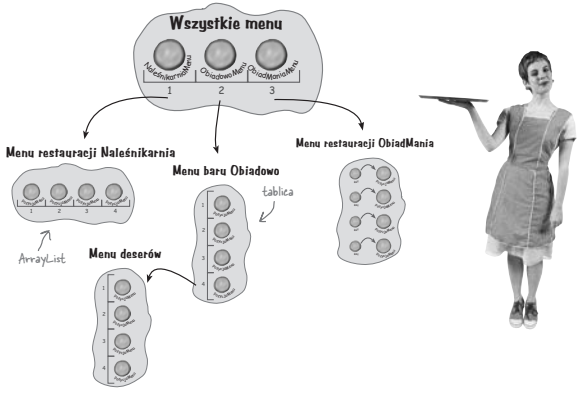
Jesteśmy jak w transie hermetyzacji: hermetyzowaliśmy już proces tworzenia obiektów, wywołania metod, złożone interfejsy, kaczki, indyki, pizze... Ciekawe, co będzie następne? Otóż teraz mamy zamiar zająć się hermetyzacją fragmentów algorytmów, tak aby klasy podrzędne mogły „się podczepiać" w różnych miejscach wykonywanych obliczeń. Co więcej, zajmiemy się również regułą projektowania, której korzenie sięgają w prostej linii... Hollywood



Wzorce Iterator i Kompozyt

9 Zarządzanie kolekcjami

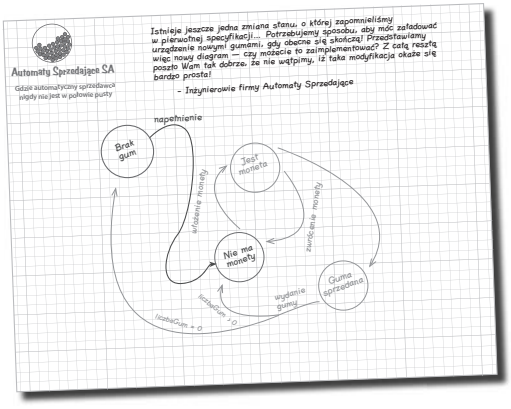
Jest wiele sposobów grupowania obiektów w kolekcje. Można utworzyć obiekty Array, Stack, List, HashMap. Każdy z nich ma swoje zalety i wady. Jednak w pewnym momencie klient rozpocznie iteracyjne przetwarzanie elementów kolekcji. Czy wtedy ujawnisz mu swoją implementację? Mam nadzieję, że nie. To nie byłoby profesjonalne. Nie musisz się jednak obawiać, Twoja kariera zawodowa nie jest zagrożona. W tym rozdziale przedstawimy metodę, która umożliwia klientom przeglądanie zawartości kolekcji bez wiedzy o tym, w jaki sposób obiekty są w nich przechowywane. Przedstawimy też technikę tworzenia „superkolekcji" obiektów, które pozwalają na obsługę bardzo rozbudowanych struktur danych. Wspomnimy także pokrótce o odpowiedzialności obiektów.



Wzorzec Stan

10 Stan obiektu

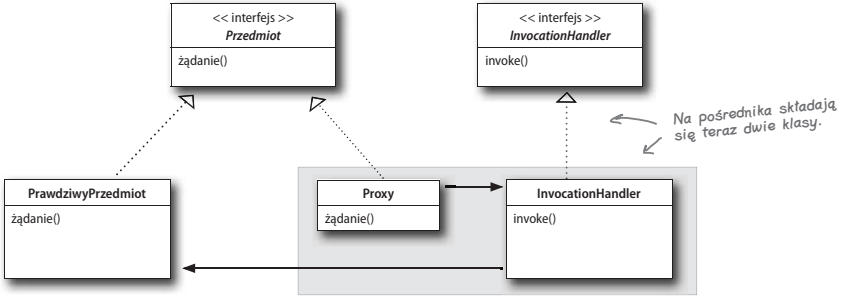
Mało znany fakt: wzorce Strategia i Stan to bliźniaki rozdzielone zaraz po narodzinach. Jak już wiemy, wzorzec Strategia umożliwił przeprowadzenie wielu niezwykle udanych przedsięwzięć opartych na zamiennie stosowanych algorytmach, z kolei wzorzec Stan podąża bardziej chwalebną ścieżką — pomaga obiektom kontrolować swoje działanie poprzez modyfikowanie swojego stanu. Choć losy obu tych wzorców są odmienne, to jednak projekt leżący u ich podstaw jest dokładnie taki sam. Jak to możliwe? Jak się przekonasz, wzorce Strategia i Stan służą do zupełnie innych celów. Zacznijmy od dokładniejszego przyjrzenia się, o co w ogóle chodzi we wzorcu Stan, a następnie, pod koniec rozdziału, wrócimy do zagadnienia wzajemnych związków pomiędzy wzorcami Strategia i Stan.



Wzorzec Pośrednik

11 Kontrola dostępu do obiektu

Próbowałeś kiedyś stosować metodę „dobrego i złego"? Ty jesteś tym dobrym, który zrobi wszystko, o co się go poprosi, który jest zawsze miły i uprzejmy. Nie chcesz jednak, żeby każdy mógł prosić o Twoje usługi. To jest miejsce dla „złego", który będzie kontrolował dostęp do Ciebie. Takie jest właśnie zadanie pośredników (ang. *proxy*) w modelu obiektowym — kontrolowanie dostępu i zarządzanie nim. Jak się przekonamy, istnieje bardzo wiele schematów takiego pośrednictwa. Na przykład obiekty pośredniczące mogą przekazywać kompletne wywołania metod na inne komputery podłączone do internetu; bywa też, że zastępują wyjątkowo leniwe obiekty.

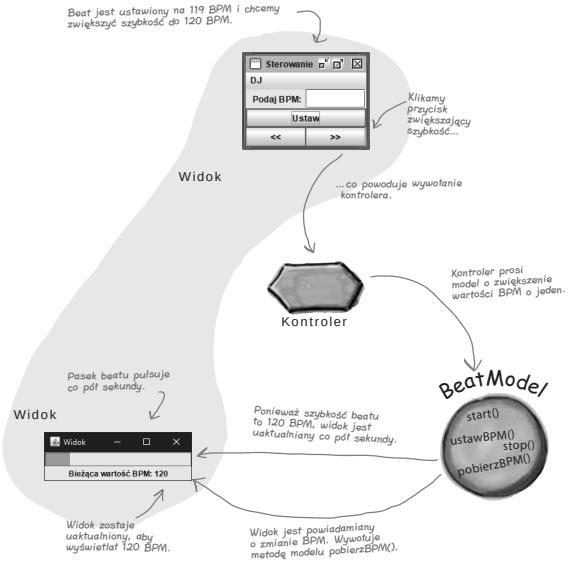


12

Wzorce złożone

Łączenie wzorców

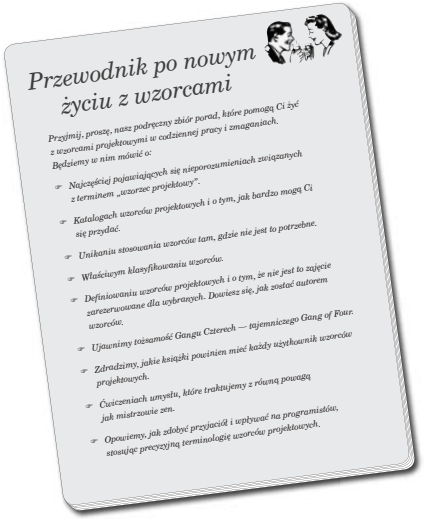
Przyszłoby Ci do głowy, że wzorce mogą pracować razem? Byliśmy już świadkami wielu niespokojnych „Pogawędek przy kominku" (a ominęły Cię „Pojedynki na śmierć i życie" wzorców, które wydawca kazał wyrzucić). Czy wsłuchując się w ich ton, można jeszcze liczyć na to, że wzorce będą ze sobą współpracować?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

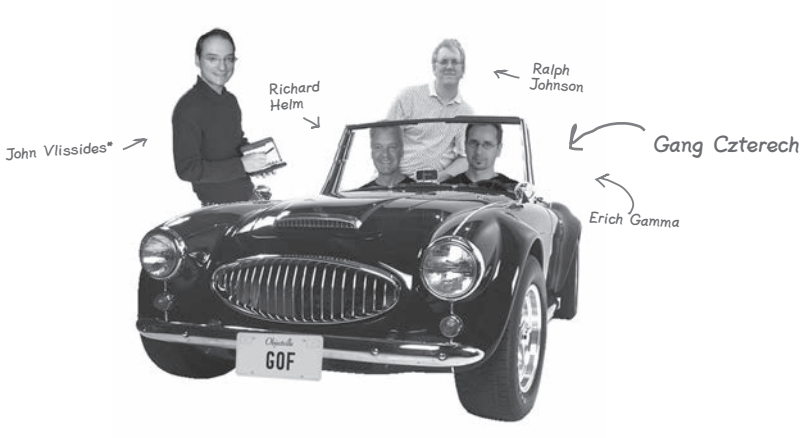


Lepsze życie dzięki wzorcom

13 Wzorce projektowe w praktyce

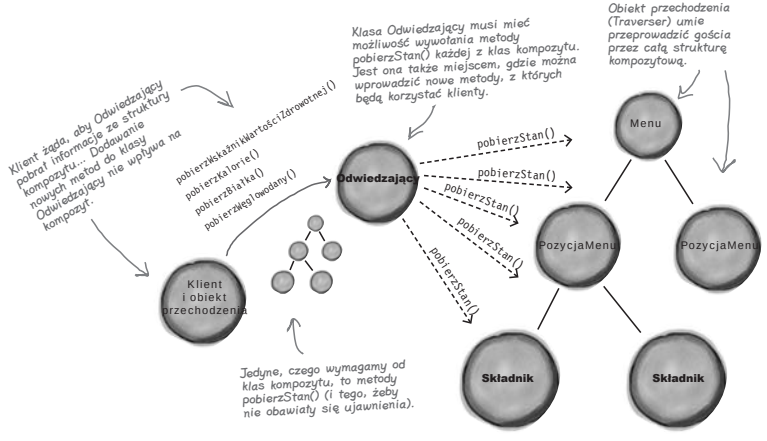
Ach, jesteś już gotowy na spotkanie z nowym wspaniałym światem pełnym wzorców projektowych... Ale zanim rozpoczniesz wędrówkę ku nowym horyzontom, musimy poświęcić nieco czasu na przedstawienie pewnych szczególnych zagadnień związanych ze stosowaniem wzorców projektowych w codziennej pracy — okazuje się bowiem, że rzeczywistość jest nieco bardziej złożona niż to, co zobaczyłeś w Obiektowie. Przygotowaliśmy więc mały przewodnik, który pomoże Ci odnaleźć się w tej twardej rzeczywistości...



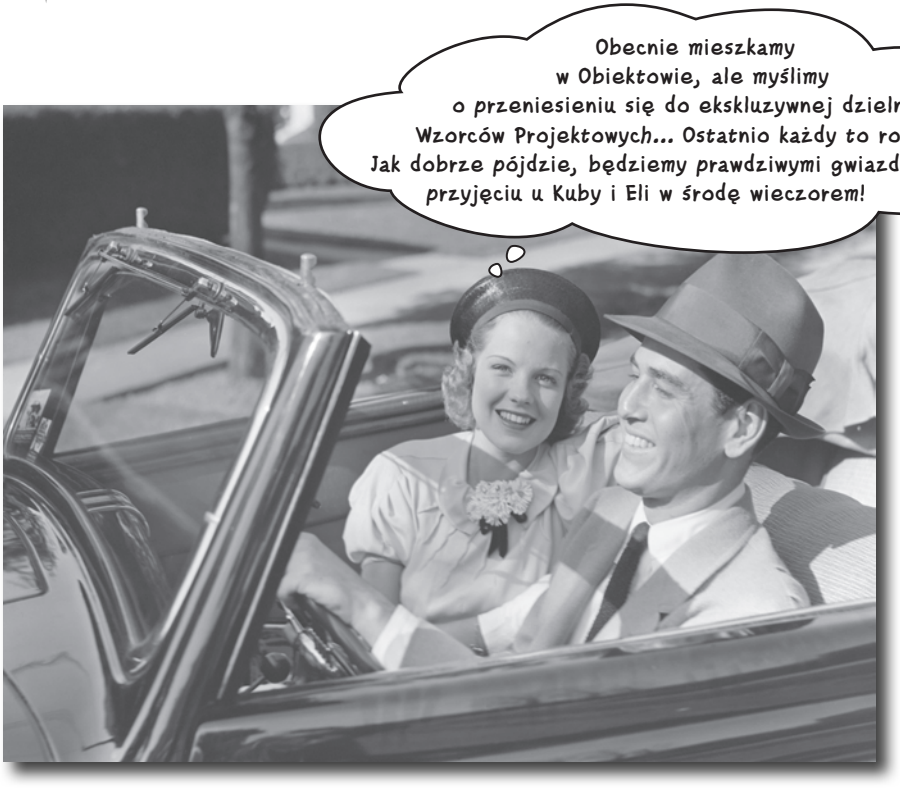


Dodatek: Pozostałe wzorce

Nie wszyscy mogą być sławni. W ciągu ostatniego ćwierćwiecza wiele się w świecie wzorców zmieniło. Od czasu pierwszego wydania książki *Wzorce projektowe. Elementy oprogramowania obiektowego wielokrotnego użytku* programiści wykorzystali opisane w nich wzorce tysiące razy. Wzorce, które zebraliśmy w tym dodatku, to dopracowane, kompletne, „oficjalne" wzorce grupy GoF. Różnią się od wcześniej opisanych tylko tym, że nie spotkamy ich tak często jak tych, którym poświęciliśmy całe rozdziały. Nie umniejsza to ich zalet i nie powinno zniechęcać do ich stosowania tam, gdzie wymaga tego sytuacja. Celem niniejszego dodatku jest zapewnienie Ci lepszej orientacji w najłatwiej dostępnych zasobach zgromadzonej przez lata wiedzy.

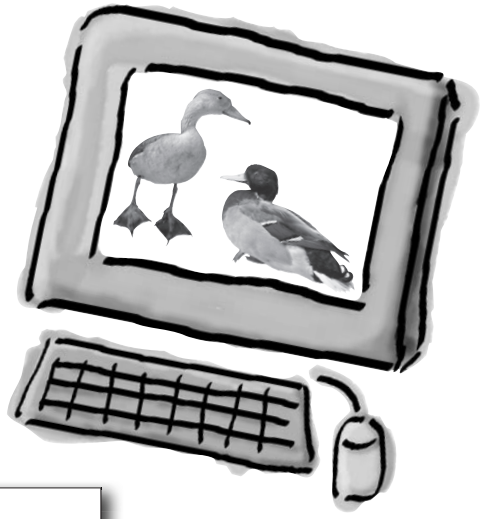


Witamy w krainie wzorców projektowych

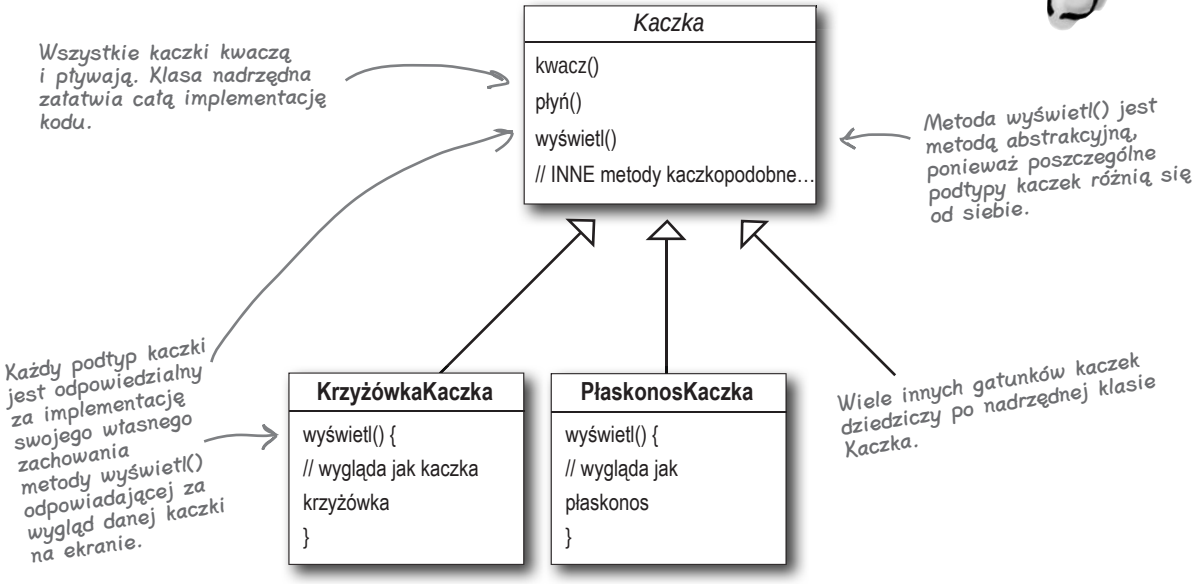


Ktoś rozwiązał już Twoje problemy. W tym rozdziale dowiesz się, dlaczego (i w jaki sposób) możesz wykorzystać wiedzę i doświadczenia zdobyte przez innych programistów, którzy podążali tą samą ścieżką projektową i — co najważniejsze — udało im się przeżyć taką wyprawę. Zanim dobrniemy do końca rozdziału, rzucimy okiem na sposoby wykorzystywania wzorców projektowych i przedstawimy ich zalety, poznamy kilka podstawowych zasad projektowania obiektowego, a także omówimy sposób działania przykładowego wzorca. Najlepszą metodą zastosowania wzorca jest *załadowanie go bezpośrednio do Twojego mózgu*, a następnie *zlokalizowanie obszarów* w obrębie projektowanych rozwiązań oraz istniejących aplikacji, w których możesz *je zastosować*. Pracując z wzorcami projektowymi, zamiast wielokrotnego wykorzystywania *kodu* wielokrotnie wykorzystujesz swoje *doświadczenia*.

Wszystko rozpoczęło się od prostej aplikacji o nazwie KaczySim



Józek pracuje w firmie będącej producentem odnoszącej ogromne sukcesy gry, której akcja toczy się w wirtualnym stawie z kaczkami. Gra *KaczySim* może wyświetlić wiele różnych gatunków kaczek, pływających oraz wydających odgłosy kwakania. Pierwszy zespół projektantów, który pracował nad powstaniem tej gry, wykorzystywał standardowe techniki programowania obiektowego i utworzył jedną klasę nadrzędną czy też, inaczej mówiąc, superklasę o nazwie Kaczka, po której wszystkie inne rodzaje kaczek dziedziczą właściwości.



W zeszłym roku firma znajdowała się pod ciągłą presją ze strony konkurentów. Po tygodniowej burzy mózgów, spędzonej na polu golfowym, zarząd firmy uznał, że nadszedł czas na wprowadzenie wielkich, innowacyjnych zmian. Potrzebował czegoś, co na nadchodzącym walnym zgromadzeniu akcjonariuszy, które miało się odbyć w *następnym tygodniu* na Maui, wywoła naprawdę *wielkie wrażenie*.

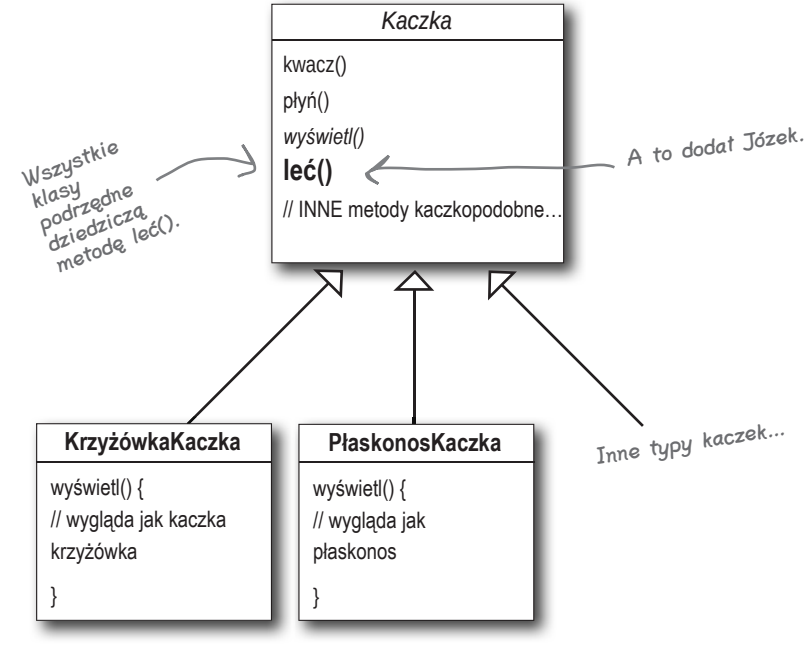
Ale teraz nasze kaczki muszą LATAĆ

Zarząd firmy uznał, że tym, czego symulator potrzebuje, by zetrzeć w proch konkurencję, są kaczki, które potrafią latać. No i oczywiście szef Józka uznał, że przygotowanie czegoś na następny tydzień nie powinno być dla niego problemem, bo jak stwierdził: „Józek jest przecież programistą obiektowym… *a to nie może być aż tak trudne*".

Oto czego chcemy.

Józek

Muszę po prostu dodać metodę leć() do klasy nadrzędnej Kaczka. Wtedy wszystkie inne kaczki będą ją dziedziczyły. Wreszcie będę mógł ujawnić swój prawdziwy geniusz programisty zorientowanego obiektowo.



Ale coś poszło strasznie nie tak...

Słuchaj, Józek, jestem na spotkaniu akcjonariuszy. Właśnie uruchomiliśmy wersję demonstracyjną tego nowego symulatora i na ekranie zaczęły latać **gumowe kaczuszki.** Czy to miał być żart?

Co się stało?

Józek nie zwrócił uwagi na fakt, że nie *wszystkie* podklasy klasy Kaczka powinny umieć *latać*. Kiedy dodawał nowe zachowanie do klasy nadrzędnej Kaczka, dodał je również do kilku klas podrzędnych, dla których takie zachowanie *nie jest* właściwe. W efekcie w programie KaczySim pojawiły się niepożądane obiekty nieożywione, które mimo to latają.

Lokalna aktualizacja kodu programu spowodowała niepożądany globalny efekt uboczny (pojawienie się latających gumowych kaczek)!

No dobrze, wychodzi na to, że do mojego projektu wkradła się drobna usterka. Nie rozumiem, czemu nie mogą tego uznać za „możliwość". Przecież to całkiem urocze...

To, o czym Józek myślał, że jest świetnym zastosowaniem dziedziczenia w celu ponownego wykorzystania istniejącego kodu, okazało się niezbyt fortunnym rozwiązaniem pod względem utrzymania kodu.

Poprzez umieszczenie metody leć() w klasie nadrzędnej Józek umożliwił latanie WSZYSTKIM kaczkom, nawet tym, które latać nie powinny.

Zauważ także, że gumowe kaczuszki nie kwaczą, zachowanie w metodzie kwacz() zostało więc zmienione na „piszcz".

Józek rozmyśla o dziedziczeniu...

Oto kolejna klasa w hierarchii kaczek; zwróć uwagę, że ta kaczka, podobnie jak klasa GumowaKaczka, nie lata, a dodatkowo także nie kwacze.

```
WabikKaczka
kwacz() {
    // przesłania, żeby nic nie robiła
}
wyświetl() { // wabik na kaczki}
leć () {
    // przesłania, żeby nic nie robiła
}
```

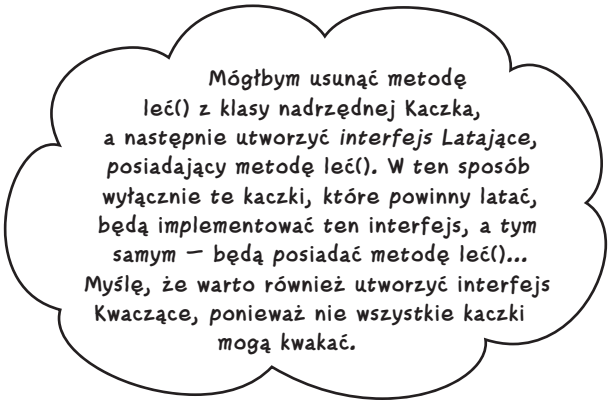
Zaostrz ołówek

Które z poniżej wymienionych negatywnych zjawisk będzie rezultatem zastosowania mechanizmu dziedziczenia do tworzenia poszczególnych zachowań kaczki? (Zaznacz wszystkie pasujące).

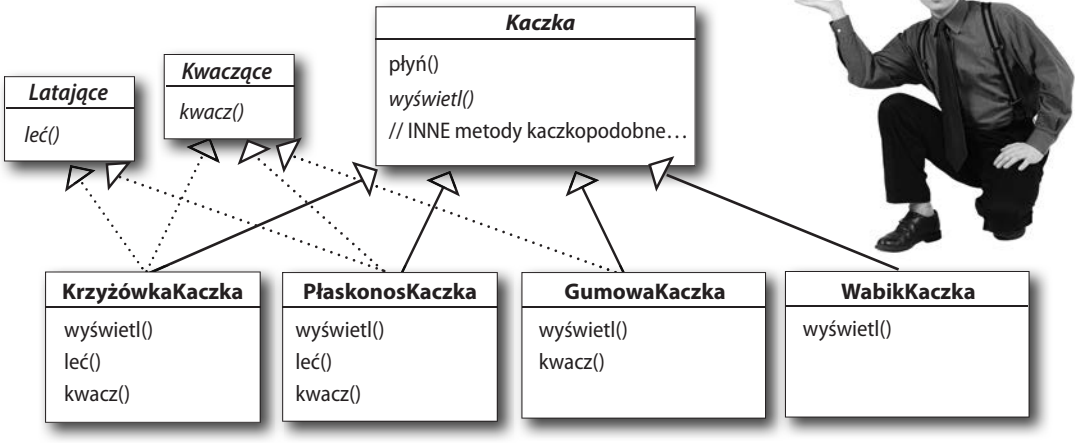
- ❑ A. Kod jest powielany w klasach podrzędnych.
- ❑ B. Wprowadzanie zmian w zachowaniu programu jest trudne.
- ❑ C. Nie możemy sprawić, by kaczki tańczyły.
- ❑ D. Trudno zebrać informacje o zachowaniach wszystkich kaczek.
- ❑ E. Kaczki nie mogą jednocześnie latać i kwakać.
- ❑ F. Wprowadzane zmiany mogą mieć niezamierzony wpływ na inne kaczki.

A może by tak interfejs?

Józek uzmysłowił sobie, że dziedziczenie nie jest odpowiednim rozwiązaniem problemu. Nastąpiło to w momencie, gdy otrzymał notatkę stwierdzającą, że zarząd firmy postanowił wprowadzać modyfikacje produktu co sześć miesięcy (co gorsza, sam zarząd jeszcze nie podjął decyzji, na czym te zmiany będą polegać). Józek już wie, że specyfikacja będzie się ciągle zmieniać i z tego powodu będzie zmuszony szczegółowo analizować oraz prawdopodobnie modyfikować działanie metod leć() oraz kwacz() dla każdej nowo dodanej klasy podrzędnej klasy Kaczka.



W takiej sytuacji Józek potrzebuje lepszego rozwiązania, które pozwoli latać lub kwakać tylko *wybranym* gatunkom kaczek (ale nie *wszystkim*).



A co TY sądzisz o takim projekcie?

To chyba najbardziej idiotyczny z Twoich pomysłów. Czy wiesz, czym jest „powielanie kodu"? Jeśli uważasz, że zmiana działania zaledwie kilku metod była złym pomysłem, to co powiesz o konieczności zmiany sposobu latania... we wszystkich 48 klasach podrzędnych klasy Kaczka, reprezentujących latające gatunki kaczek?

Co byś zrobił na miejscu Józka?

Wiemy, że nie wszystkie klasy podrzędne powinny mieć zachowania opisujące latanie czy kwakanie, jasne jest więc, że dziedziczenie nie jest tutaj dobrym rozwiązaniem. Z drugiej jednak strony, choć implementacja takich interfejsów jak Latające i (lub) Kwaczące *częściowo* rozwiązuje nasz problem (gumowe kaczki nie będą już latać wbrew swojej naturze), takie podejście kompletnie niszczy możliwość ponownego wykorzystywania tych samych fragmentów kodu opisującego takie zachowania, czyli krótko mówiąc, wprowadza *inny* koszmar, tym razem związany z utrzymaniem kodu. Oczywiście, nie warto już tutaj wspominać o różnych sposobach latania, spotykanych przecież wśród tych kaczek, które *potrafią* latać…

W tym momencie mógłbyś już w zasadzie po cichutku oczekiwać, że odpowiedni wzorzec projektowy nadjedzie z fasonem na białym rumaku i uratuje Cię z opresji. Ale czy to byłoby naprawdę zabawne? Absolutnie nie! Mamy więc zamiar poszukać rozwiązania w nieco staroświecki sposób — *wykorzystując dobre zasady programowania obiektowego*.

Czyż nie byłoby cudownie, gdyby istniała możliwość napisania oprogramowania w taki sposób, że jeżeli zaistnieje potrzeba wprowadzenia w nim zmian, można będzie ich dokonywać, wywierając możliwie jak najmniejszy wpływ na istniejący kod? Moglibyśmy spędzać znacznie mniej czasu na poprawianiu kodu, a za to poświęcić go więcej na zaimplementowanie w programach nowych, lepszych funkcji...

Jedyny pewny element procesu wytwarzania oprogramowania

No dobrze, jak sądzisz, co jest jedynym elementem, z którym na pewno zetkniesz się podczas wytwarzania oprogramowania?

Niezależnie od tego, gdzie pracujesz, co tworzysz i jakiego języka programowania używasz, co będzie jedynym pewnym elementem, który zawsze będzie Ci towarzyszył?

(aby przeczytać, użyj lusterka)

Niezależnie od tego, jak dobrze zaprojektujesz aplikację, z upływem czasu musi ona rosnąć i zmieniać się — w przeciwnym wypadku po prostu *umrze* śmiercią naturalną.

Zaostrz ołówek

Zmiany mogą być wywoływane przez szereg różnych czynników. Wypisz kilka powodów, dla których musiałeś dokonywać modyfikacji kodu w swoich aplikacjach (aby ułatwić Ci wykonanie ćwiczenia, na początku podaliśmy dwa własne przykłady). Zanim przejdziesz dalej, sprawdź odpowiedzi zamieszczone na końcu rozdziału.

Klienci lub użytkownicy zdecydowali, że chcą otrzymać coś zupełnie nowego bądź też dodać do oprogramowania nowe funkcjonalności.

Zarząd firmy podjął decyzję, że będziemy korzystać z bazy danych innego producenta, a same dane zostaną kupione od dostawcy, który oczywiście wykorzystuje zupełnie inny ich format. Auć!

Zerowanie problemu...

Wiemy już zatem, że w naszym przypadku mechanizm dziedziczenia nie sprawdził się najlepiej, gdyż zachowania kaczek w poszczególnych klasach podrzędnych zmieniają się, a co więcej, nie *wszystkie* klasy podrzędne kaczek powinny mieć te same zachowania. Na pierwszy rzut oka pomysł z implementacją interfejsów Latające i Kwaczące wygląda obiecująco — tylko kaczki, które potrafią latać, będą implementowały interfejs Latające itd. Jest tylko ten jeden problem, że interfejsy Javy zazwyczaj nie mają własnego kodu implementującego, nie można więc wielokrotnie wykorzystywać fragmentów takiego kodu. Oznacza to, że jeżeli przyjdzie Ci modyfikować jakieś określone zachowanie, zazwyczaj będziesz zmuszony prześledzić wszystkie klasy podrzędne danej klasy nadrzędnej, wyszukać w nich poszczególne implementacje danego zachowania, a następnie dokonać odpowiednich modyfikacji — o tym, że przy okazji zapewne „dorzucisz" do modyfikowanego kodu *nowe* błędy, nie warto już nawet wspominać!

Na szczęście istnieje odpowiednia na taką okazję reguła projektowania:

Reguła projektowania

Zidentyfikuj te aspekty aplikacji, które są zmienne, i oddziel je od tych, które pozostają stałe.

Jest to nasza pierwsza i na pewno nie ostatnia reguła projektowania. W dalszych częściach książki poświęcimy im więcej uwagi.

Innymi słowy, jeśli zidentyfikujesz jakiś aspekt kodu, który się zmienia, na przykład wraz z pojawieniem się każdego nowego wymagania, to wiesz, że znalazłeś zachowanie, które musi zostać wydzielone i odseparowane od kodu, który się nie zmienia.

O tej zasadzie można także myśleć w inny sposób: ***weź te elementy, które ulegają zmianom, i hermetyzuj je, tak byś później mógł je modyfikować lub rozszerzać bez konieczności wprowadzania modyfikacji w pozostałych fragmentach kodu***.

Pomimo swojej uderzającej prostoty powyższa zasada stanowi praktycznie podwaliny pod niemal każdy wzorzec projektowy. Wszystkie wzorce w taki czy inny sposób pozwalają na to, by *pewne fragmenty systemu mogły być modyfikowane niezależnie od innych*.

Okej, a zatem do dzieła! Czas wyciągnąć zachowania kaczek z poszczególnych klas podrzędnych klasy Kaczka.

Weź to, co się zmienia, i „hermetyzuj" to w taki sposób, by nie miało wpływu na resztę kodu.

Wynik? Mniej niezamierzonych efektów ubocznych będących konsekwencją wprowadzania zmian w kodzie i większa elastyczność systemu.

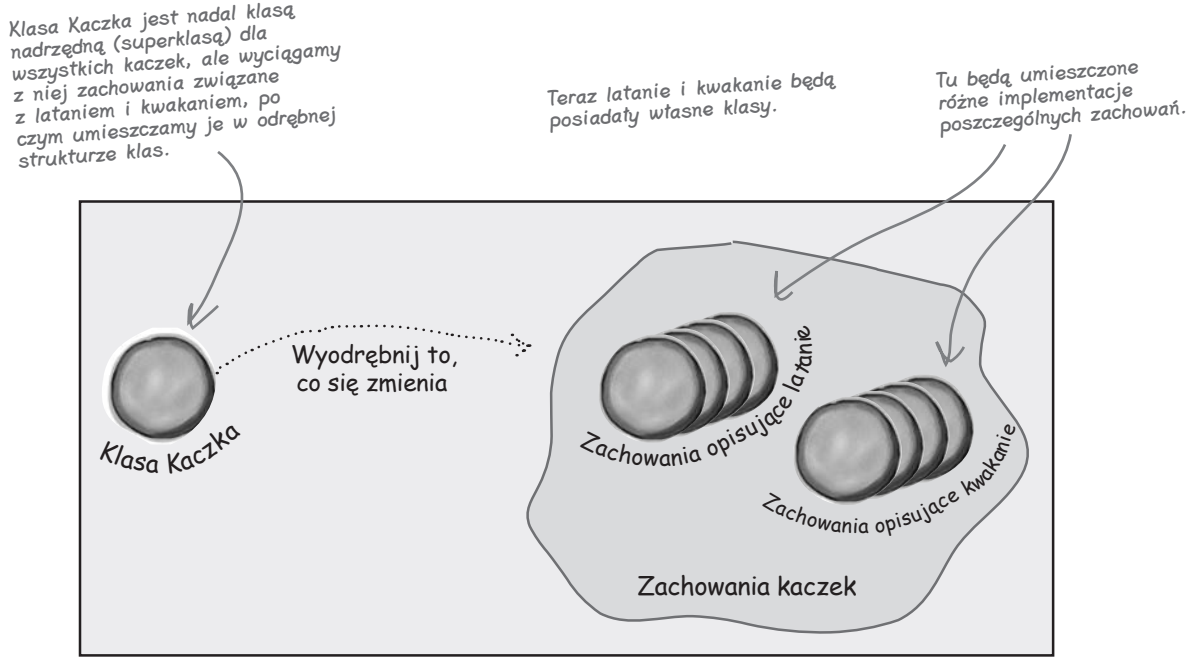
Oddzielanie tego, co się zmienia, od tego, co pozostaje niezmienione

Gdzie mamy rozpocząć? Z tego, co nam wiadomo, poza pewnymi problemami z metodami leć() i kwacz() klasa Kaczka działa całkiem dobrze i wydaje się, że inne jej elementy nie powinny się zbyt często zmieniać. Z tego względu poza wprowadzeniem kilku drobnych modyfikacji pozostawimy klasę Kaczka we względnym spokoju.

Teraz, aby oddzielić „elementy, które się zmieniają, od tych, które pozostają niezmienione", będziemy chcieli utworzyć dwa *zestawy* klas (całkowicie niezależnych od klasy nadrzędnej Kaczka) — jeden związany z *lataniem* i drugi z *kwakaniem*. Każdy z zestawów klas będzie posiadał swoją niezależną implementację odpowiednich zachowań. Może się na przykład zdarzyć tak, że będziesz chciał mieć *jedną* klasę odpowiadającą za *kwakanie*, inną klasę odpowiadającą za *piszczenie* i, wreszcie, jeszcze jedną, która będzie odpowiadała za *niewydawanie żadnych dźwięków*.

Wiemy, że metody leć() oraz kwacz() są tymi częściami klasy Kaczka, które podlegają zmianom „od kaczki do kaczki".

Aby wydzielić omawiane zachowania z klasy Kaczka, musimy wyciągnąć obydwie metody **z wnętrza tej klasy, a następnie utworzyć zestaw nowych klas reprezentujących poszczególne zachowania.**



Projektowanie zachowań Kaczki

W jaki sposób zaprojektujemy więc zestaw klas implementujących latanie i kwakanie?

Przede wszystkim chcielibyśmy, by nasze rozwiązanie było elastyczne — w końcu to właśnie brak elastyczności zachowań kaczek wpędził nas w kłopoty. Wiemy również, że będziemy chcieli *przypisywać* zachowania instancjom klasy Kaczka. Na przykład możemy chcieć utworzyć nową instancję klasy KrzyżówkaKaczka, a następnie zainicjować ją konkretnym *typem* latania. A skoro już o tym mowa, to niby dlaczego nie mielibyśmy zapewnić możliwości *dynamicznego* modyfikowania zachowań kaczek? Innymi słowy, w klasach kaczek powinniśmy przygotować metody pozwalające na dynamiczne przypisywanie zachowań, tak byśmy *podczas działania* programu mogli zmieniać sposób latania obiektu KrzyżówkaKaczka.

Mając na uwadze takie cele, przyjrzyjmy się naszej drugiej regule projektowania:

Reguła projektowania

Skoncentruj się na tworzeniu interfejsów, a nie implementacji.

Do reprezentacji poszczególnych zachowań będziemy wykorzystywać interfejsy — na przykład LatanieInterfejs i KwakanieInterfejs — a każda implementacja danego *zachowania* będzie implementowała jeden z tych interfejsów.

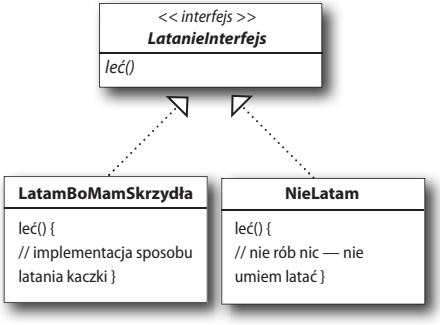
A zatem tym razem to nie klasa *Kaczka* będzie przechowywała implementacje wspomnianych interfejsów. Zamiast tego utworzymy szereg klas, których jedynym przeznaczeniem będzie reprezentowanie poszczególnych zachowań (takich jak „piszczenie") — i to właśnie te klasy *zachowań*, a nie klasa Kaczka, będą implementowały nasze interfejsy zachowań.

Takie podejście jest zupełnie inne od dotychczasowego. Wcześniej zachowanie było determinowane określoną implementacją w klasie nadrzędnej Kaczka lub wyspecjalizowaną implementacją w samej klasie podrzędnej. W obydwu przypadkach polegaliśmy na *implementacji*. W efekcie nie było innego wyjścia, jak tylko korzystać z istniejącej implementacji zachowania, i nie dysponowaliśmy możliwością zmiany danego zachowania (inną niż modyfikacja kodu programu).

W naszym nowym projekcie klasy podrzędne klasy Kaczka będą używały zachowań reprezentowanych przez *interfejsy* (LatanieInterfejs i KwakanieInterfejs), dzięki czemu właściwa *implementacja* tych zachowań (czyli konkretny kod definiujący to zachowanie w klasie implementującej odpowiedni interfejs — LatanieInterfejs lub KwakanieInterfejs) nie będzie zablokowana w klasie nadrzędnej Kaczka.

Od tej chwili zachowania klasy Kaczka będą umieszczone w osobnej klasie — klasie, która implementuje interfejs reprezentujący dane zachowanie.

W ten sposób klasy kaczek nie muszą znać żadnych szczegółów implementacji swoich własnych zachowań.

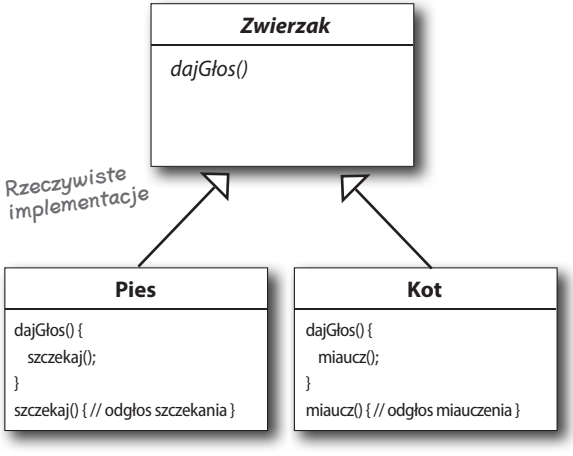


„Skoncentruj się na tworzeniu interfejsów” oznacza „skoncentruj się na tworzeniu typów nadrzędnych”.

Słowo *interfejs* jest tutaj zdecydowanie zbyt przeładowane. Z jednej strony mamy samą jego *ideę*, a z drugiej strony interfejs jest związany bezpośrednio ze *słowem kluczowym* `interface` języka Java. Warto zatem uświadomić sobie, że przedstawioną wcześniej regułę *koncentrowania się na tworzeniu interfejsów* można z powodzeniem wcielać w życie bez konieczności użycia słowa kluczowego `interface` i tworzenia interfejsu w ścisłym rozumieniu składni języka Java. Cała sztuka polega na tym, aby odpowiednio wykorzystywać polimorfizm i tworzyć odpowiednie typy nadrzędne, tak by zachowanie poszczególnych obiektów nie było ściśle związane z ich własnym kodem. W zasadzie moglibyśmy więc zamiast zalecać „skoncentruj się na tworzeniu typów nadrzędnych”, powiedzieć, że „deklarowanym typem zmiennych powinien być typ nadrzędny, czyli zazwyczaj klasa abstrakcyjna bądź interfejs — chodzi o to, aby obiekty przypisane do tych zmiennych mogły przyjmować postać dowolnej rzeczywistej implementacji tego typu nadrzędnego, co w praktyce oznacza, że klasa, która je deklaruje, nie musi znać prawdziwych typów poszczególnych obiektów!”.

Prawdopodobnie to, co teraz powiemy, nie będzie dla Ciebie niczym nowym, ale aby mieć pewność, że myślimy tak samo jak Ty, przedstawimy prosty przykład zastosowania typu polimorficznego. Wyobraź sobie klasę abstrakcyjną o nazwie Zwierzak, posiadającą dwie rzeczywiste implementacje: Pies i Kot.

Abstrakcyjny typ nadrzędny (może to być klasa abstrakcyjna LUB interfejs).



Bezpośrednia implementacja kodu może wyglądać następująco:

```
Pies p = new Pies();
p.szczekaj();
```

Deklaracja zmiennej „p” jako zmiennej typu Pies (rzeczywista implementacja Zwierzaka) wymusza tworzenie kodu odpowiadającego danej implementacji.

Ale za to **utworzenie odpowiedniego interfejsu (lub typu nadrzędnego)** może wyglądać następująco:

```
Zwierzak zwierzak = new Pies();
zwierzak.dajGłos();
```

Wiemy, że to jest Pies, ale możemy tutaj użyć polimorficznego odwołania do tego zwierzaka.

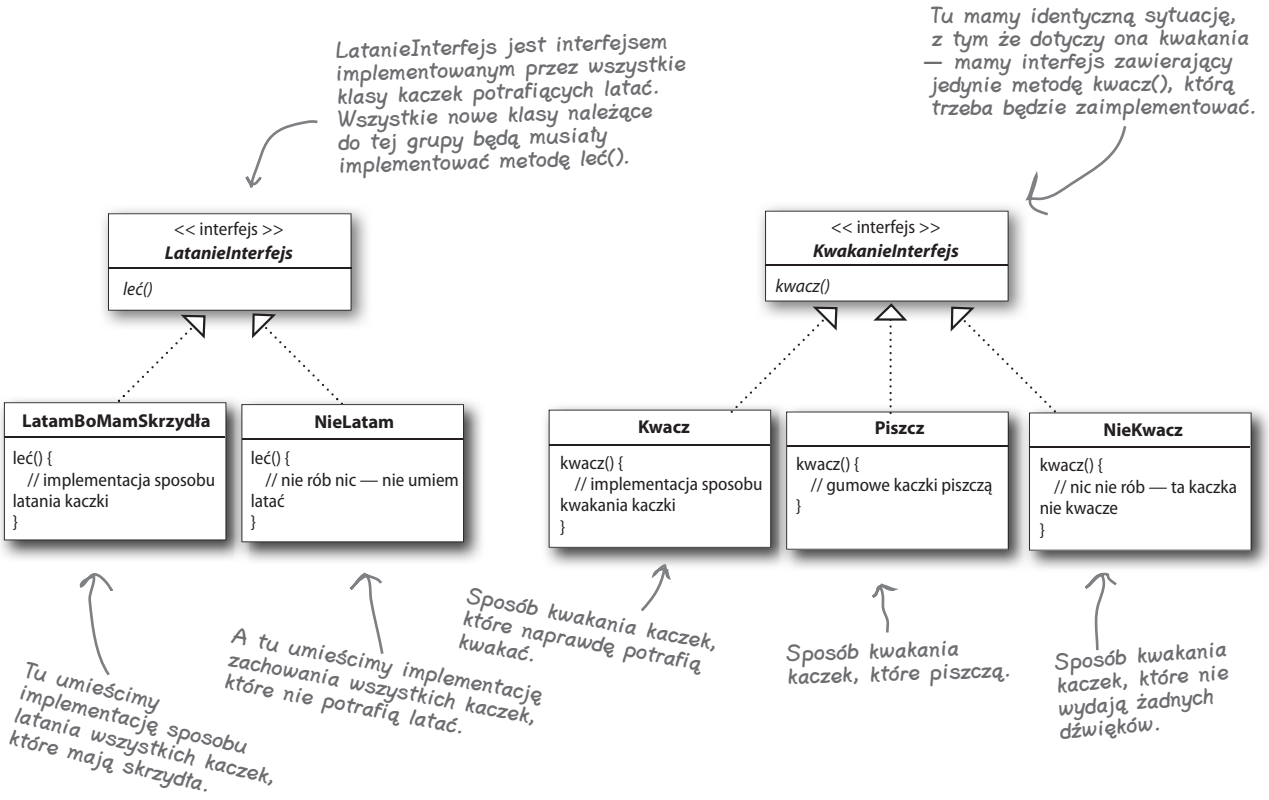
Sztywne umieszczenie w kodzie tworzenia nowego obiektu (jak new Pies()) jest niezłym rozwiązaniem, ale jeszcze lepszym będzie tworzenie rzeczywistej implementacji obiektu bezpośrednio w czasie działania programu:

```
z = getZwierzak();
z.dajGłos();
```

Nie wiemy, jaki jest FAKTYCZNY podtyp Zwierzaka w danej chwili... jednak wszystko, o co musimy zadbać, to żeby program „wiedział”, jak odpowiedzieć na polecenie dajGłos().

Implementacja zachowań klasy Kaczka

Poniżej przedstawiamy dwa interfejsy: LatanieInterfejs i KwakanieInterfejs, wraz z odpowiednimi klasami, które implementują zachowania:



Dzięki takiemu projektowi inne typy obiektów mogą wykorzystywać odpowiednie zachowania (latanie i kwakanie), ponieważ nie są one już dłużej ukryte w klasie nadrzędnej Kaczka!

W razie potrzeby możemy dodawać nowe zachowania bez konieczności modyfikacji jakichkolwiek istniejących klas opisujących dotychczasowe zachowania i bez modyfikacji tych klas opisujących kaczki, które wykorzystują zachowania związane z lataniem.

Jak widać, możemy swobodnie korzystać z zalet, jakie daje możliwość WIELOKROTNEGO WYKORZYSTYWANIA kodu, i to bez tego całego niepotrzebnego bagażu, jaki niesie ze sobą mechanizm dziedziczenia.

Nie istnieją głupie pytania

P: Czy zawsze powinienem najpierw zaimplementować aplikację, określić elementy, które się zmieniają, a dopiero potem wydzielić je i hermetyzować?

O: Nie zawsze. Często zdarza się tak, że już na etapie projektowania aplikacji można przewidzieć obszary, które będą podlegały zmianom, dzięki czemu można niejako z wyprzedzeniem zaprojektować i zaimplementować odpowiednio elastyczny kod. Z lektury naszej książki dowiesz się, że odpowiednie reguły i wzorce projektowe mogą być wykorzystywane na niemal każdym etapie tworzenia aplikacji.

P: Czy także klasę Kaczka powinniśmy przekształcić na interfejs?

O: Nie w tym przypadku. Jak się niebawem przekonasz, po złożeniu wszystkiego w jedną całość bardzo ułatwi nam życie fakt, że będziemy mieli Kaczkę jako rzeczywistą implementację klasy oraz poszczególne rodzaje kaczek, jak KrzyżówkaKaczka, które będą po niej dziedziczyły wspólne właściwości i metody. Dzięki temu, że teraz usunęliśmy elementy specyficzne dla dziedziczenia, możemy korzystać ze struktury, która posiada wszystkie zalety dziedziczenia, ale nie powiela jego wad i problemów.

P: Posiadanie klasy, która opisuje tylko zachowanie, wydaje się nieco dziwne. Czy klasy nie powinny przypadkiem reprezentować czegoś bardziej materialnego? Czy klasy nie powinny posiadać stanu ORAZ zachowania?

O: W systemach obiektowych — tak, klasy reprezentują elementy, które — ogólnie rzecz biorąc — posiadają zarówno stan (zapisany w zmiennych instancyjnych), jak i odpowiednie metody. W naszym przypadku elementy te charakteryzują się określonymi zachowaniami. Należy jednak pamiętać, że nawet dane zachowanie może posiadać stan i swoje metody; przykładowo zachowanie związane z lataniem może posiadać swoje zmienne instancyjne reprezentujące odpowiednie atrybuty latania, takie jak ilość uderzeń skrzydłami na minutę, maksymalna szybkość czy wysokość lotu itp.).

Zaostrz ołówek

1. W jaki sposób przy wykorzystaniu naszego nowego projektu możesz dodać możliwość latania z napędem rakietowym do symulatora KaczySim?

2. Czy potrafisz wymyślić przykład klasy, która mogłaby używać zachowania Kwacz, a nie reprezentowałaby kaczki?

Odpowiedzi:

1. Utwórz klasę
latamZNapędemRakietowym
implementującą interfejs LataniеInterfejs.

2. Przykładem takiej klasy może być klasa reprezentująca wabik (urządzenie wydające dźwięki przypominające odgłosy wydawane przez kaczki).

Integracja zachowań klasy Kaczka

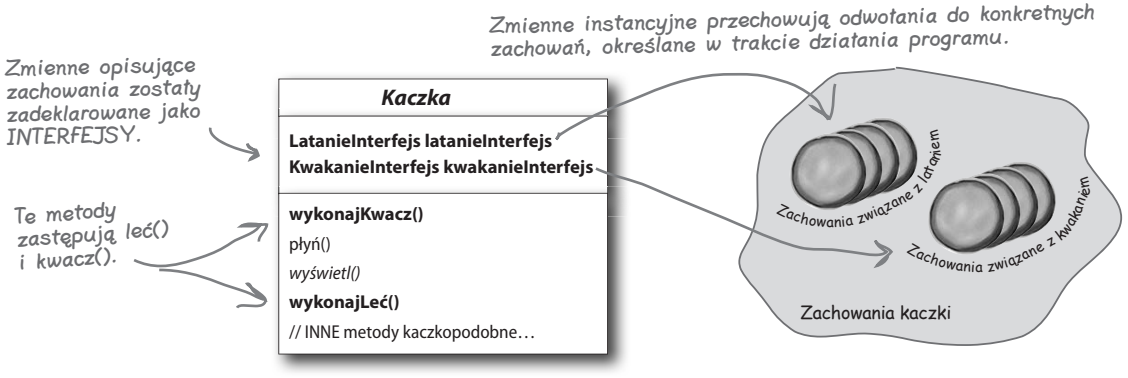
Elementem kluczowym jest teraz fakt, że obiekt Kaczka zamiast wykorzystywać metody opisujące latanie i kwakanie, zdefiniowane w klasie nadrzędnej Kaczka (bądź w klasach podrzędnych), będzie delegować te zachowania do innych obiektów.

Zaraz dowiesz się, jak tego dokonać:

1. **Po pierwsze, w definicji klasy Kaczka umieścimy dwie zmienne instancyjne typów LatanieInterfejs i KwakanieInterfejs**, którym nadamy odpowiednio nazwy latanieInterfejs i kwakanieInterfejs. Każdy obiekt reprezentujący kaczkę w trakcie działania programu będzie zapisywał w tych zmiennych *konkretne* zachowania, takie jak LatamBoMamSkrzydła w przypadku latania lub Piszcz w przypadku kwakania.

   Z klasy nadrzędnej Kaczka (jak również ze wszystkich jej klas podrzędnych) usuniemy metody leć() oraz kwacz(), co wynika z faktu, że odpowiadające im zachowania zostały przeniesione do klas implementujących interfejsy LatanieInterfejs oraz KwakanieInterfejs.

   Zamienimy metody leć() i kwacz() w klasie Kaczka na dwie podobne, o nazwach wykonajLeć() oraz wykonajKwacz() — niebawem dowiesz się, jakie jest ich przeznaczenie.



2. **A teraz zaimplementujemy metodę wykonajKwacz():**

```
public abstract class Kaczka {
    KwakanieInterfejs kwakanieInterfejs;
    // więcej

    public void wykonajKwacz() {
        kwakanieInterfejs.kwacz();
    }
```

Każdy obiekt Kaczka posiada odwołania do czegoś, co ma zaimplementowany interfejs KwakanieInterfejs.

Zamiast samodzielnie obsługiwać kwakanie, obiekt Kaczka deleguje obsługę tego zachowania do obiektu wskazywanego przez zmienną kwakanieInterfejs.

Całkiem proste, prawda? Aby zakwakać, kaczka po prostu pozwala obiektowi wskazywanemu przez zmienną kwakanieInterfejs na wydawanie odgłosów kwakania. W tym miejscu kodu zupełnie nie dbamy o to, jakiego konkretnego typu jest obiekt Kaczka. ***Tym, co ma dla nas znaczenie, jest to, że obiekt wie, jak wykonać metodę kwacz()***!

Więcej integracji...

3 No dobrze, nadszedł czas, by zastanowić się, **w jaki sposób** będą **ustawiane** wartości zmiennych instancyjnych latanieInterfejs i kwakanieInterfejs. Przyjrzyjmy się definicji klasy KrzyżówkaKaczka:

```
public class KrzyżówkaKaczka extends Kaczka {

    public KrzyżówkaKaczka() {
        kwakanieInterfejs = new Kwacz();
        latanieInterfejs = new LatamBoMamSkrzydła();
    }

    public void wyświetl() {
        System.out.println("Jestem prawdziwą kaczką krzyżówką");
    }
}
```

Klasa KrzyżówkaKaczka używa klasy Kwacz do obsługi czynności kwakania. Kiedy więc wywoływana jest metoda wykonajKwacz(), odpowiedzialność za wykonanie kwakania będzie delegowana do obiektu Kwacz, a my uzyskamy faktyczne kwakanie.

Do obsługi latania ta klasa używa klasy LatamBoMamSkrzydła.

Pamiętaj, że klasa KrzyżówkaKaczka dziedziczy zmienne kwakanieInterfejs i latanieInterfejs po klasie Kaczka.

Jak wiemy, w realnym świecie kaczka krzyżówka kwacze dokładnie tak, jak się tego możemy po kaczkach spodziewać — jest to prawdziwe **kwakanie**, a nie **piszczenie** czy też **milczenie**. Jak widać, kiedy jest tworzony nowy obiekt KrzyżówkaKaczka, jego konstruktor przypisuje odziedziczonej zmiennej instancyjnej kwakanieInterfejs nowy obiekt typu Kwacz (jest to konkretna klasa implementująca interfejs KwakanieInterfejs).

Opisany powyżej proces przebiega identycznie dla zachowania opisującego sposób latania — konstruktor klasy KrzyżówkaKaczka przypisuje do odziedziczonej zmiennej instancyjnej latanieInterfejs nowy obiekt typu LatamBoMamSkrzydła() (będącego konkretną klasą implementującą interfejs LatanieInterfejs).

Zaraz, zaraz! Czy przypadkiem nie wspominaliście kiedyś, że NIE powinniśmy się koncentrować na tworzeniu implementacji? A co w takim razie robimy w konstruktorze? Przecież — ni mniej, ni więcej — tworzymy w nim nowy obiekt klasy Kwacz!

Dobry strzał — to jest dokładnie to, co właśnie tutaj robimy… *przynajmniej w tej chwili*.

W dalszej części książki będziemy mieli jednak do dyspozycji więcej wzorców projektowych, które pozwolą nam poprawić zastosowane tu rozwiązanie.

Zwróć uwagę, że choć faktycznie *określamy* zachowania, używając konkretnych klas (poprzez tworzenie nowych obiektów takich klas jak Kwacz czy LatamBoMamSkrzydła i przypisywanie ich odpowiednim zmiennym instancyjnym), to jednak możemy je *łatwo* zmieniać podczas działania programu.

Zauważ, że choć nasz program nadal jest elastyczny, to nie radzimy sobie zbyt dobrze z inicjowaniem zmiennych instancyjnych w sposób elastyczny. Ale zwróć uwagę, że zmienna instancyjna kwakanieInterfejs jest typu interfejsu, a więc w czasie działania programu (wykorzystując magię polimorfizmu) możemy do niej dynamicznie przypisywać wybrane klasy implementujące poszczególne zachowania.

Znajdź chwilę, aby zastanowić się, w jaki sposób można zaimplementować kaczkę, której zachowania będą się mogły zmieniać dynamicznie w trakcie działania programu (przykładowy kod stanowiący rozwiązanie tego problemu znajdziesz kilka stron dalej).

Testowanie kodu klasy Kaczka

1 **Wpisz i skompiluj przedstawiony poniżej kod klasy Kaczka (plik Kaczka.java) oraz klasy KrzyżówkaKaczka (plik KrzyżówkaKaczka.java), przedstawiony kilka stron wcześniej.**

```
public abstract class Kaczka {

   LatanieInterfejs latanieInterfejs;
   KwakanieInterfejs kwakanieInterfejs;
   public Kaczka() { }

   public abstract void wyświetl();

   public void wykonajLeć() {
     latanieInterfejs.leć();
   }

   public void wykonajKwacz() {
      kwakanieInterfejs.kwacz();
   }

   public void pływ() {
      System.out.println("Wszystkie kaczki pływają, nawet te sztuczne!");
   }
}
```

Deklaruje dwie zmienne typu interfejsu dla poszczególnych zachowań. Wszystkie klasy podrzędne klasy Kaczka (w tym samym pakiecie) będą dziedziczyły te zmienne.

Delegowanie zachowania do odpowiedniej klasy.

2 **Wpisz i skompiluj kod interfejsu LatanieInterfejs (plik LatanieInterfejs.java) oraz kody dwóch klas implementujących zachowania (pliki LatamBoMamSkrzydła.java oraz NieLatam.java).**

```
public interface LatanieInterfejs {
   public void leć();
}
```

Interfejs, który implementują wszystkie klasy posiadające to zachowanie (latanie).

```
public class LatamBoMamSkrzydła implements LatanieInterfejs
{
   public void leć() {
      System.out.println("O rany! Latam!");
   }
}
```

Implementacja sposobu latania kaczek, które NAPRAWDĘ potrafią latać...

```
public class NieLatam implements LatanieInterfejs {
   public void leć() {
      System.out.println("Nie umiem latać!");
   }
}
```

Implementacja sposobu latania kaczek, które NIE POTRAFIĄ latać (takich jak gumowe kaczuszki lub kaczki wabiki).

Testowania kodu klasy Kaczka ciąg dalszy...

3 **Wpisz i skompiluj kod interfejsu KwakanieInterfejs (plik KwakanieInterfejs.java) oraz kody trzech klas implementujących zachowanie związane z kwakaniem (pliki Kwacz.java, NieKwacz.java oraz Piszcz.java).**

```
public interface KwakanieInterfejs {
   public void kwacz();
}
```

```
public class Kwacz implements KwakanieInterfejs {
   public void kwacz() {
      System.out.println("Kwa! Kwa!");
   }
}
```

```
public class NieKwacz implements KwakanieInterfejs {
   public void kwacz() {
      System.out.println("<<CISZA>>");
   }
}
```

```
public class Piszcz implements KwakanieInterfejs {
   public void kwacz() {
      System.out.println("Piszczę!");
   }
}
```

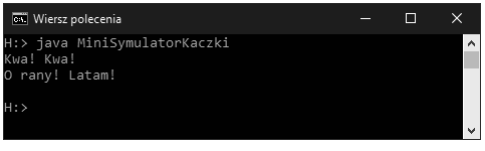
4 **Wpisz i skompiluj klasę testową (MiniSymulatorKaczki.java).**

```
public class MiniSymulatorKaczki {
   public static void main(String[] args) {
      Kaczka dzika = new KrzyżówkaKaczka();
      dzika.wykonajKwacz();
      dzika.wykonajLeć();
   }
}
```

To polecenie powoduje wywołanie dziedziczonej metody wykonajKwacz(), która następnie deleguje obsługę zachowania do właściwej metody interfejsu KwakanieInterfejs (przykładowo wywołuje metodę kwacz() właściwą dla rodzaju obiektu wskazywanego przez dziedziczoną zmienną kwakanieInterfejs).

Następnie robimy dokładnie to samo z dziedziczoną przez klasę KrzyżówkaKaczka metodą wykonajLeć().

Uruchom kod programu!

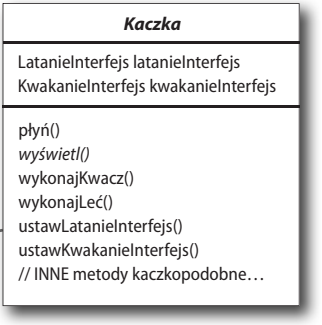


Dynamiczne ustawianie zachowania

To naprawdę wstyd dysponować tymi wszystkimi dynamicznymi możliwościami, którymi cechują się nasze kaczki, i w ogóle z nich nie korzystać! Wyobraź sobie, że chcesz określić typ zachowania kaczki przy użyciu metody ustawiającej zdefiniowanej w klasie Kaczka, zamiast podawać go w konstruktorze kaczki.

1 Dodaj dwie nowe metody do klasy Kaczka:

```
public void ustawLatanieInterfejs (LatanieInterfejs li) {
    latanieInterfejs = li;
}

public void ustawKwakanieInterfejs(KwakanieInterfejs ki) {
    kwakanieInterfejs = ki;
}
```



Te metody możemy wywoływać w dowolnym momencie, kiedy tylko będziemy chcieli zmieniać zachowania kaczek w locie.

Dopisek redaktora: niepotrzebne dwuznaczności — poprawić

2 Utwórz nową klasę kaczki (ModelowaKaczka.java)

```
public class ModelowaKaczka extends Kaczka {
    public ModelowaKaczka() {
        latanieInterfejs = new NieLatam();
        kwakanieInterfejs = new Kwacz();
    }

    public void wyświetl() {
        System.out.println("Jestem modelową kaczką!");
    }
}
```

Nasza modelowa kaczka zaczyna życie jako nielot... bez możliwości latania.

3 Utwórz nowy typ implementujący LatanieInterfejs (LatamZNapędemRakietowym.java)

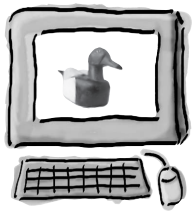
Tak właśnie ma być, tworzymy przecież zachowanie opisujące latanie z napędem rakietowym.

```
public class LatamZNapędemRakietowym implements LatanieInterfejs
{
    public void leć() {
        System.out.println("Mam rakietę! I jej używam...");
    }
}
```

4 **Zmodyfikuj kod klasy testowej (MiniSymulatorKaczki.java), dodaj do niego klasę ModelowaKaczka i spraw, by mogła ona korzystać z latania przy użyciu napędu rakietowego.**

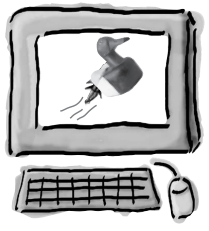
```
public class MiniSymulatorKaczki {
    public static void main(String[] args) {
        Kaczka dzika = new KrzyżówkaKaczka();
        dzika.wykonajKwacz();
        dzika.wykonajLeć();

        Kaczka modelowa = new ModelowaKaczka();
        modelowa.wykonajLeć();
        modelowa.ustawLatanieInterfejs(new LatamZNapędemRakietowym());
        modelowa.wykonajLeć();
    }
}
```

przed



Pierwsze wywołanie metody wykonajLeć() deleguje obsługę zachowania do metody obiektu ustawionego w konstruktorze klasy ModelowaKaczka, czyli do obiektu klasy NieLatam.

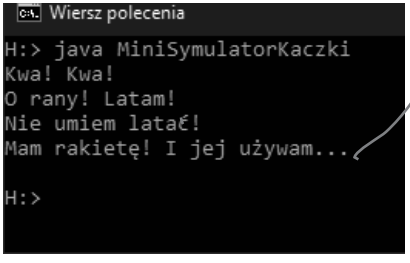
Ta instrukcja wywołuje odziedziczoną metodę ustawiającą i... proszę bardzo! Nasza modelowa kaczka nagle zyskała możliwość latania z użyciem napędu rakietowego.



po

Jeśli to zadziałało, oznacza to, że nasza modelowa kaczka dynamicznie zmieniła swoje zachowanie związane z lataniem! Nie mógłbyś TEGO zrobić, gdyby cała implementacja była umieszczona w klasie Kaczka.

5 **Uruchom program!**



Aby zmienić zachowanie kaczki w czasie działania programu, wystarczy wywołać odpowiednią metodę ustawiającą odpowiadającą za określanie danego zachowania.

Kompletny diagram hermetyzowanych zachowań

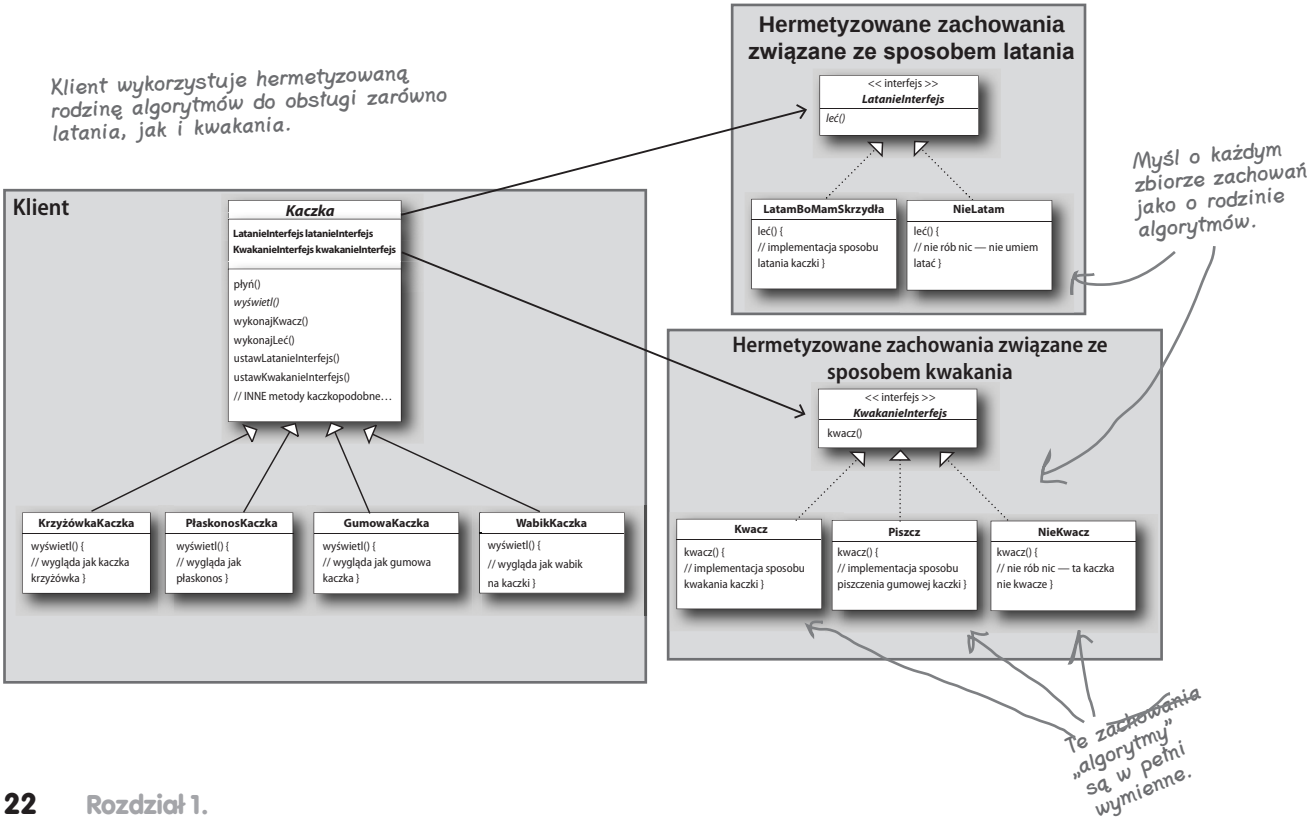
No dobrze, skoro wskoczyliśmy już na głęboką wodę z projektem symulatora kaczek, warto się zatrzymać, by złapać oddech i przyjrzeć się kompletnemu schematowi aktualnego rozwiązania.

Na poniższym diagramie przedstawiliśmy kompletną postać struktury klas. Mamy tu wszystko, czego mógłbyś oczekiwać: poszczególne kaczki rozszerzające klasę Kaczka, sposoby latania zdefiniowane w klasach implementujących interfejs LatanieInterfejs oraz sposoby kwakania zaimplementowane w klasach implementujących interfejs KwakanieInterfejs.

Zwróć także uwagę, że zaczęliśmy nieco inaczej opisywać pewne elementy. Zamiast myśleć o zachowaniach kaczek jako o *zbiorze zachowań*, będziemy o nich myśleć jako o *rodzinie algorytmów*. Pomyśl o tym, że w projekcie KaczySim algorytmy reprezentują co prawda poszczególne czynności, jakie kaczka może wykonać (różne sposoby latania lub kwakania), lecz równie dobrze moglibyśmy użyć takich samych technik do utworzenia zestawu klas, które implementują zasady obliczania podatku dochodowego przy uwzględnieniu rodzaju płatnika, różnych stawek podatkowych oraz obowiązujących ulg i zwolnień.

Zwróć szczególną uwagę na *relacje* pomiędzy poszczególnymi klasami. Najlepiej będzie, jeśli po prostu weźmiesz pióro i zapiszesz nazwy odpowiednich relacji (JEST, MA oraz IMPLEMENTUJE) na poszczególnych strzałkach diagramu klas.

Zrób to koniecznie!



Relacja MA może być lepsza od JEST

Relacja MA jest bardzo ciekawa: każda kaczka ma interfejsy LatanieInterfejs oraz KwakanieInterfejs, do których deleguje zachowania „latanie” i „kwakanie”.

Kiedy łączysz w ten sposób dwie klasy, używasz **kompozycji**. Zamiast *dziedziczyć* swoje zachowania, kaczki otrzymują je w ten sposób, że są *komponowane* z odpowiednich obiektów implementujących poszczególne zachowania.

Jest to bardzo ważna technika, która znajduje swoje odzwierciedlenie w naszej trzeciej regule projektowania:

Reguła projektowania

Przedkładaj kompozycję nad dziedziczenie.

Jak zapewne zdążyłeś zauważyć, tworzenie systemów metodą kompozycji zapewnia o wiele większą elastyczność. Nie tylko pozwala na hermetyzowanie całych rodzin algorytmów we własnych zestawach klas, ale również na ***zmianę zachowania w czasie działania programu***, o ile tylko obiekty, których używasz do kompozycji, implementują odpowiednie interfejsy zachowań.

Kompozycja jest wykorzystywana w wielu różnych wzorcach projektowych; w czasie lektury niniejszej książki poznasz jeszcze wiele jej zalet, ale dowiesz się również o jej wadach.

MOC UMYSŁU

Wabik to rodzaj specjalnego gwizdka, którego myśliwi używają do naśladowania odgłosów wydawanych przez kaczki (odgłosów kwakania). Zastanów się, w jaki sposób mógłbyś zaimplementować swój własny wabik, który nie dziedziczyłby zachowań po klasie Kaczka.

Mistrz i uczeń...

Mistrz: *Młody człowieku, powiedz mi, czego nauczyłeś się o filozofii programowania obiektowego.*

Uczeń: *Mistrzu, nauczyłem się, że obietnicą obiektowości jest możliwość wielokrotnego wykorzystywania tych samych elementów.*

Mistrz: *Kontynuuj, młody człowieku...*

Uczeń: *Mistrzu, dzięki dziedziczeniu wszystkie dobre rzeczy mogą być wykorzystywane po wielokroć, dzięki czemu czas tworzenia kodu możemy przyciąć równie łatwo jak łodygę bambusa...*

Mistrz: *Młody człowieku, czy więcej czasu poświęcamy na pisanie kodu **przed**, czy **po** zakończeniu prac nad projektem?*

Uczeń: *Odpowiedź brzmi: po, Mistrzu. Zawsze potrzebujemy więcej czasu na konserwację i wprowadzanie zmian w kodzie niż na jego początkowe projektowanie.*

Mistrz: *Czyżby zatem, młody człowieku, nasze wysiłki miały iść w stronę zapewnienia jak największych możliwości wielokrotnego wykorzystywania rzeczy — nawet **kosztem** pogorszenia możliwości utrzymania i rozbudowy?*

Uczeń: *Mistrzu, wierzę, że tak właśnie powinno być.*

Mistrz: *Widzę, że wciąż jeszcze musisz się sporo nauczyć, młody człowieku. Chciałbym, abyś teraz odszedł i oddał się dalszym medytacjom na temat właściwości dziedziczenia. Jak zdążyłeś zauważyć, dziedziczenie ma swoje wady, a istnieją także inne sposoby zapewniania możliwości ponownego stosowania rzeczy...*

Rozmawiając o wzorcach projektowych...

Gratulujemy poznania pierwszego wzorca projektowego!

Właśnie zastosowałeś swój pierwszy wzorzec projektowy — wzorzec o nazwie **Strategia** (ang. Strategy). Tak, wszystko się zgadza! Użyłeś tego wzorca do przeprojektowania aplikacji KaczySim.

Dzięki jego zastosowaniu nasz symulator jest gotowy na niemal dowolne zmiany, jakie zarząd firmy może wymyślić podczas kolejnego biznesowego wyjazdu na Maui.

Dopiero teraz, pomimo że wcześniej przeprowadziliśmy Cię przez długą drogę implementacji wzorca, przedstawimy jego formalną definicję:

Wzorzec Strategia definiuje rodzinę algorytmów, hermetyzuje je jako osobne klasy i sprawia, że są one w pełni wymienne. Zastosowanie tego wzorca pozwala na to, aby zmiany w implementacji tych algorytmów były całkowicie niezależne od klientów, które ich używają.

Jeśli chcesz zaimponować kolegom bądź zrobić dobre wrażenie na swoich przełożonych, to użyj właśnie TEJ definicji.

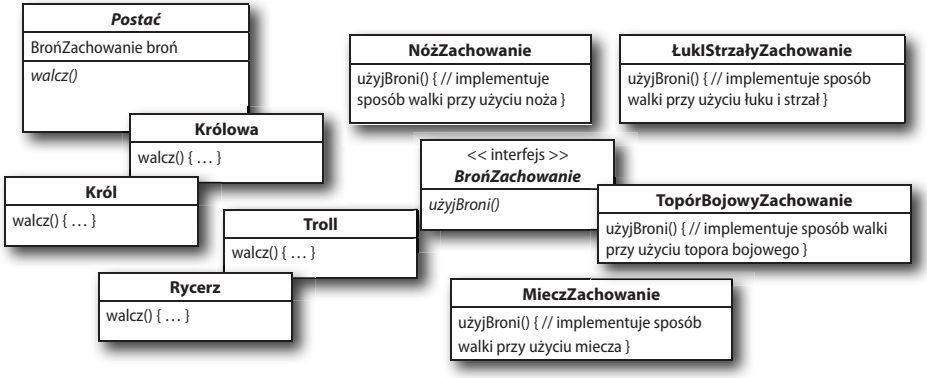
Łamigłówka projektowa

Poniżej znajdziesz całe mnóstwo przypadkowo rozrzuconych klas oraz interfejsów stanowiących hierarchię typów gry przygodowej. Znajdziesz zarówno klasy opisujące bohaterów gry, jak i klasy opisujące zachowania broni, jakich poszczególne postaci mogą używać. Każda postać może używać tylko jednej broni w danej chwili, jednak może zmieniać broń w dowolnym momencie gry. Twoim zadaniem jest uporządkowanie tego całego kramu…

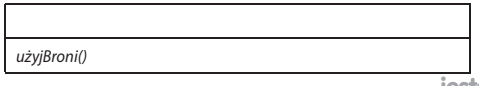
(Odpowiedzi znajdziesz na końcu rozdziału).

A oto Twoje zadanie:

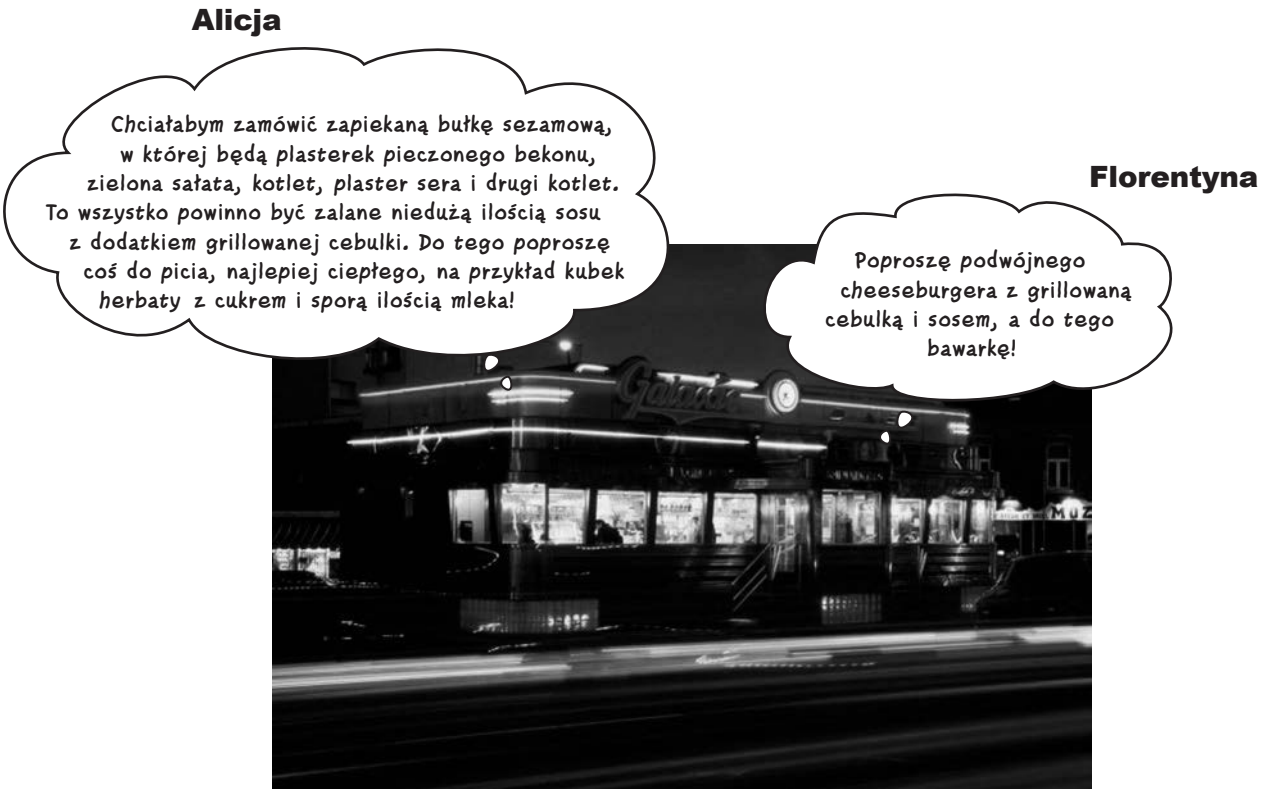
1. Poukładaj poszczególne klasy.
2. Zidentyfikuj jedną klasę abstrakcyjną, jeden interfejs oraz osiem klas.
3. Narysuj odpowiednie strzałki pomiędzy klasami.
   a. Takimi strzałkami oznacz dziedziczenie (słowo kluczowe „extends”).
   b. Takimi strzałkami oznacz implementację interfejsu (słowo kluczowe „implements”).
   c. Takimi strzałkami oznacz relację „MA”.
4. Umieść metodę ustawBroń() we właściwej klasie.




```
ustawBroń(BrońZachowanie b) {
    this.broń = b;
}
```



Zasłyszane w lokalnym barze szybkiej obsługi...



Jaka jest różnica pomiędzy tymi dwoma zamówieniami? Żadna! Obie dziewczyny zamówiły to samo, z tym że Alicja użyła przynajmniej dwa razy tyle słów oraz zdawała się testować cierpliwość gderliwego kelnera z baru szybkiej obsługi.

Co takiego zatem posiada Florentyna, czego nie ma Alicja? Otóż zarówno ona, jak i kelner z baru szybkiej obsługi używają **wspólnego słownika**. Dzięki temu nie tylko łatwiej jest się jej dogadać z kelnerem, ale również kelnerowi łatwiej zapamiętać całe zamówienie, ponieważ zapewne zna on na pamięć wszystkie zestawy obiadowe.

Wzorce projektowe stanowią swoisty wspólny słownik, którego możesz używać do porozumiewania się z innymi programistami. Kiedy dobrze poznasz to słownictwo, z pewnością będziesz mógł się łatwiej z nimi komunikować; a jednocześnie możesz zainspirować innych, nieznających wzorców projektowych, do zapoznania się z nim. Podniesie to również Twoje umiejętności projektowania architektury aplikacji, ponieważ będziesz w stanie **myśleć** na bardziej abstrakcyjnym **poziomie *wzorców projektowych***, a nie tylko na bardziej szczegółowym i konkretnym poziomie *obiektów*.

Zasłyszane w sąsiednim boksie

No to utworzyłem w końcu tę klasę rozgłoszeniową. Śledzi ona wszystkie obiekty, które w danej chwili nasłuchują, i za każdym razem, kiedy otrzymuje nowy zestaw danych, wysyła odpowiedni komunikat do wszystkich odbiorców. Fajną właściwością jest to, że obiekty nasłuchujące mogą w każdej chwili dołączyć do grona odbiorców nadawanych komunikatów lub nawet mogą się same stamtąd usunąć. Cały proces jest naprawdę dynamiczny i zapewnia luźne powiązania!

MOC UMYSŁU

Czy potrafisz sobie wyobrazić inne wspólne słowniki, które będą wykorzystywane w sytuacjach odmiennych niż tylko projektowanie obiektowe czy bar szybkiej obsługi? (Podpowiedź: pomyśl o mechanikach samochodowych, stolarzach, pracownikach kontroli lotów). Jakimi cechami szczególnymi charakteryzuje się przekazywanie informacji w takim żargonie?

Czy myślałeś o tych aspektach projektowania obiektowego, które są komunikowane poprzez nazwę danego wzorca projektowego? Jakie cechy szczególne niesie ze sobą nazwa „wzorzec Strategia"?

Rysiek, czy ty przypadkiem nie mówiłeś, że używałeś wzorca Obserwator?

Właśnie. Jeżeli komunikujesz się przy użyciu wzorców, inni projektanci będą od razu dokładnie i *precyzyjnie* orientowali się w szczegółach projektu, który opisujesz. Z drugiej strony uważaj, aby nie dostać „gorączki wzorcowej"... Łatwo się zorientujesz, że już ją masz, jeżeli zaczniesz używać wzorców do tworzenia aplikacji typu „Witaj, świecie!"...

Potęga wspólnego słownika wzorców

Kiedy komunikujesz się przy użyciu wzorców, robisz coś więcej, niż tylko posługujesz się ŻARGONEM.

Wspólne słowniki wzorców są naprawdę POTĘŻNYM narzędziem. Kiedy porozumiewasz się z innym projektantem czy całym zespołem przy użyciu wzorców, tak naprawdę przekazujesz nie tylko samą nazwę wzorca, ale jednocześnie cały zestaw jego cech szczególnych, jego charakterystyki i ograniczenia.

„Do implementacji różnych zachowań kaczek używamy wzorca Strategia". Taka informacja mówi Ci, że zachowania kaczek są hermetyzowane w postaci osobnego zestawu klas, który w razie potrzeby może być łatwo rozbudowywany i modyfikowany, nawet w trakcie działania programu.

Wzorce pozwalają Ci powiedzieć więcej przy użyciu mniejszej liczby słów. Jeżeli do opisywania swoich projektów używasz wzorców, inni projektanci szybko i precyzyjnie zorientują się, o co Ci chodzi.

Rozmowa na poziomie wzorców pozwoli Ci poświęcić większą ilość czasu na przemyślenie samego projektu niż na jego implementację. Rozmowa o rozwiązaniach wykorzystujących wzorce projektowe pozwala na utrzymywanie dyskusji na poziomie projektowania. Dzięki temu unikniesz konieczności zagłębiania się w szczegółowe rozważania na temat implementacji poszczególnych obiektów i klas.

Na ilu byłeś takich spotkaniach, na których dyskusja bardzo szybko podryfowała w stronę takiego czy innego szczegółu implementacyjnego?

Wspólne słowniki mogą być rodzajem turbodoładowania dla całego Twojego zespołu programistycznego. Zespół programistów, który ma duże doświadczenie w korzystaniu z wzorców, może iść naprzód o wiele szybciej i z mniejszym ryzykiem występowania nieporozumień.

W miarę jak cały zespół zaczyna wymieniać pomysły i doświadczenia w kategoriach wzorców, zaczynasz tworzyć całą społeczność użytkowników, która czerpie korzyści z posługiwania się wzorcami.

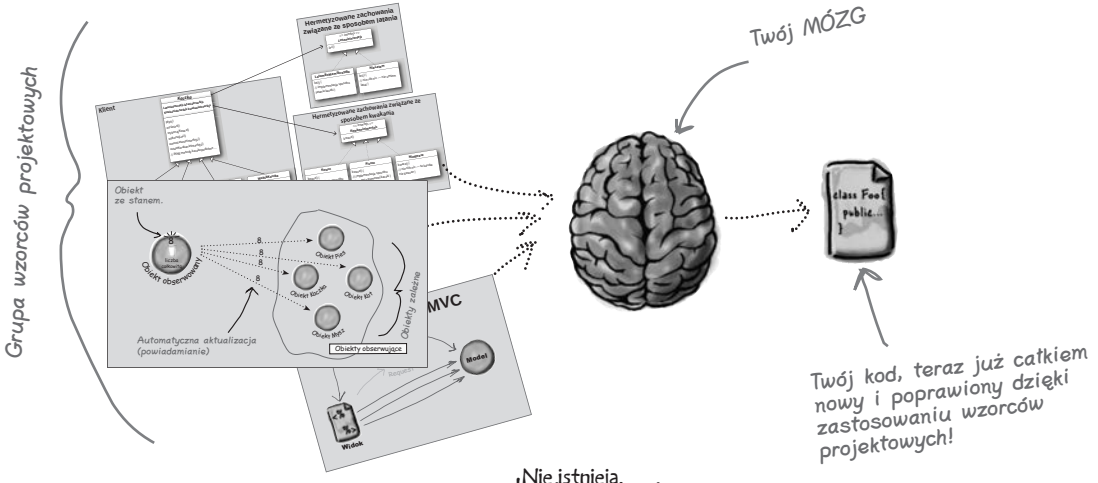
Wspólne słowniki zachęcają coraz więcej młodych programistów do większego wysiłku. Młodzi programiści często starają się naśladować starszych, bardziej doświadczonych kolegów. Jeżeli starszy programista korzysta z wzorców projektowych, to zazwyczaj młodsi programiści są bardziej zmotywowani, aby również się z nimi zapoznać. Spróbuj stworzyć w swojej firmie całe środowisko programistów posługujących się wzorcami.

Pomyśl o stworzeniu w swojej firmie grupy, która będzie się zajmowała studiowaniem, analizą i wdrażaniem wzorców — może nawet uda Ci się połączyć przyjemne z pożytecznym i za trudy nauki ktoś Ci jeszcze co nieco zapłaci...

W jaki sposób mogę używać wzorców projektowych?

Praktycznie każdy z nas korzystał z różnego rodzaju gotowych produktów, takich jak biblioteki, frameworki itp. Nie zastanawiając się nad tym, bierzemy po prostu taki produkt, korzystamy z możliwości jego interfejsu programowania aplikacji (ang. *Application Programming Interface*, w skrócie: *API*), kompilujemy, a potem pozostaje już tylko czerpać korzyści z dużej ilości kodu, który tak naprawdę napisał przecież ktoś inny. Pomyśl chociażby o API języka Java i tej całej funkcjonalności, jaką Ci ona daje: obsługa sieci, graficznego interfejsu użytkownika, obsługa operacji wejścia-wyjścia itd. Biblioteki i frameworki przeszły długą drogę, zanim wyewoluowały do obecnego modelu, w którym możesz wybrać odpowiednie komponenty i po prostu dołączyć je do aplikacji. Ale… mimo całej swojej doskonałości API nie pomoże nam w utworzeniu takiej struktury naszych własnych aplikacji, która będzie przejrzysta, łatwa w utrzymaniu i elastyczna. I właśnie w tym miejscu przychodzą nam z pomocą wzorce projektowe.

Wzorce projektowe nie są od razu implementowane w kodzie aplikacji — najpierw muszą odpowiednio zagnieździć się w Twoim MÓZGU. Dopiero po tym, gdy odpowiednia wiedza na temat praktycznego zastosowania wzorców znajdzie się na swoim miejscu, możesz spróbować użyć jej podczas tworzenia nowych projektów bądź reorganizacji kodu istniejących aplikacji, jeśli uznasz, że powoli zaczyna się doń wkradać chaos.



Nie istnieją głupie pytania

P: Skoro wzorce projektowe są tak wspaniałe, to dlaczego ktoś nie stworzy ich biblioteki, tak by inni już nie musieli tego robić?

O: Wzorce projektowe znajdują się na nieco wyższym poziomie abstrakcji niż zwykłe biblioteki. Mówią one, w jaki sposób możemy stworzyć taką strukturę klas i obiektów, która umożliwi nam rozwiązanie określonych problemów. Naszym zadaniem jest natomiast adaptacja takiego projektu do potrzeb konkretnej aplikacji.

P: Czy biblioteki i frameworki także są wzorcami projektowymi?

O: Biblioteki i frameworki nie są wzorcami projektowymi; one zapewniają tylko specyficzne implementacje, które możemy stosować w naszym kodzie. Jednak z drugiej strony biblioteki i frameworki korzystają z wzorców projektowych w swoich własnych implementacjach. To znakomite rozwiązanie, ponieważ jeżeli dobrze rozumiesz wzorce projektowe, to łatwiej będzie Ci zrozumieć API, które powstało w oparciu o nie.

P: Czyli nie ma żadnych bibliotek wzorców projektowych?

O: Nie, jednak nieco później dowiesz się o katalogach wzorców projektowych obejmujących zestawienia wzorców, których możesz używać podczas tworzenia własnych aplikacji.

Programista-sceptyk **Wzorcowy guru**

Programista: No dobrze, ale czy… hm, czy to wszystko nie jest jednak po prostu programowaniem obiektowym? Chodzi mi o to, że jeżeli postępuję zgodnie z regułami hermetyzacji klas i znam zagadnienia dotyczące abstrakcyjności oraz mechanizmów dziedziczenia i polimorfizmu, to czy naprawdę muszę myśleć o wzorcach projektowych? Czy to nie dlatego byłem na tych wszystkich szkoleniach i kursach programowania obiektowego? Myślę, że wzorce projektowe są użyteczne dla tych, którzy nie znają dobrze zasad programowania obiektowego.

Guru: Taaak, to jest właśnie jeden z mitów związanych z projektowaniem obiektowym: że dzięki dobrej znajomości podstaw takiego programowania będziemy automatycznie specjalistami w tworzeniu systemów elastycznych, nadających się do wielokrotnego wykorzystania oraz łatwych w utrzymaniu.

Programista: A nie?

Guru: Nie. Jak się okazuje, zasady tworzenia systemów obiektowych, które posiadają wymienione wcześniej cechy, nie zawsze są takie proste i oczywiste, a ich odkrycie często bywa okupione ciężką i wytężoną pracą.

Programista: Myślę, że zaczynam łapać, o co chodzi. Po pewnym czasie te wszystkie — czasami niezbyt oczywiste — zasady tworzenia systemów obiektowych zostały zebrane…

Guru: …tak, w postaci zestawu wzorców nazywanych powszechnie wzorcami projektowymi.

Programista: Czy zatem, znając odpowiednie wzorce, mogę pominąć tę całą ciężką pracę i będę w stanie od razu tworzyć projekty, które zawsze będą działać?

Guru: Tak, do pewnego stopnia, ale powinieneś pamiętać, że projektowanie jest sztuką. Zawsze będą istniały jakieś kompromisy. Jeżeli jednak będziesz korzystał z dobrze przemyślanych i przetestowanych wzorców projektowych, zaoszczędzisz mnóstwo czasu.

Programista: A co powinienem zrobić w sytuacji, kiedy nie uda mi się znaleźć odpowiedniego wzorca?

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

Guru: Pod „przykryciem" wzorców znajdziesz szereg dobrze znanych i rozumianych reguł projektowania obiektowego, a znając je, na pewno będziesz w stanie poradzić sobie w sytuacjach, gdy nie istnieje wzorzec pasujący do problemu, który musisz rozwiązać.

Programista: Reguły? Masz na myśli te wszystkie abstrakcyjności, hermetyzacje klas oraz…

Guru: Tak! Jednym z sekretów, które pozwalają na stworzenie łatwych w utrzymaniu systemów obiektowych, jest myślenie z wyprzedzeniem, jak mogą się one zmieniać w czasie — właśnie te reguły pozwalają radzić sobie w takich sytuacjach.

Narzędzia do Twojej projektowej skrzynki narzędziowej

Prawie skończyłeś już lekturę pierwszego rozdziału naszej książki! Do tej pory zdążyłeś umieścić kilka narzędzi w swojej skrzynce narzędziowej programisty obiektowego; zanim przejdziemy do rozdziału 2., zróbmy dla przypomnienia ich listę.

Podstawy programowania obiektowego

Abstrakcyjność

Hermetyzacja

Polimorfizm

Dziedziczenie

Zakładamy, że znasz już podstawy projektowania zorientowanego obiektowo i zasady działania polimorfizmu klas, dziedziczenia oraz hermetyzacji. Jeżeli jednak Twoja wiedza wymaga pewnego odświeżenia, mimo wszystko powinieneś teraz ściągnąć z półki książkę **Rusz głową! Java**, przypomnieć sobie co nieco i dopiero wtedy ponownie wrócić do tego rozdziału.

Reguły programowania obiektowego

Hermetyzuj to, co się zmienia.

Przedkładaj kompozycję nad dziedziczenie.

Koncentruj się na tworzeniu interfejsów, a nie implementacji.

W najbliższym czasie przyjrzymy się tym zagadnieniom nieco dokładniej; dotożymy również do tej listy kilka nowych pozycji.

Wzorce programowania obiektowego

Strategia — definiuje rodziny algorytmów, dokonuje ich hermetyzacji i powoduje, że stają się one wymienne. Wzorzec Strategia (Strategy) pozwala na modyfikację danego algorytmu niezależnie od klienta, który tego algorytmu używa.

W czasie pracy z książką myśl o tym, w jaki sposób wzorce są powiązane z podstawami i regułami projektowania obiektowego.

Jeden z głowy, ale jeszcze dużo przed nami!

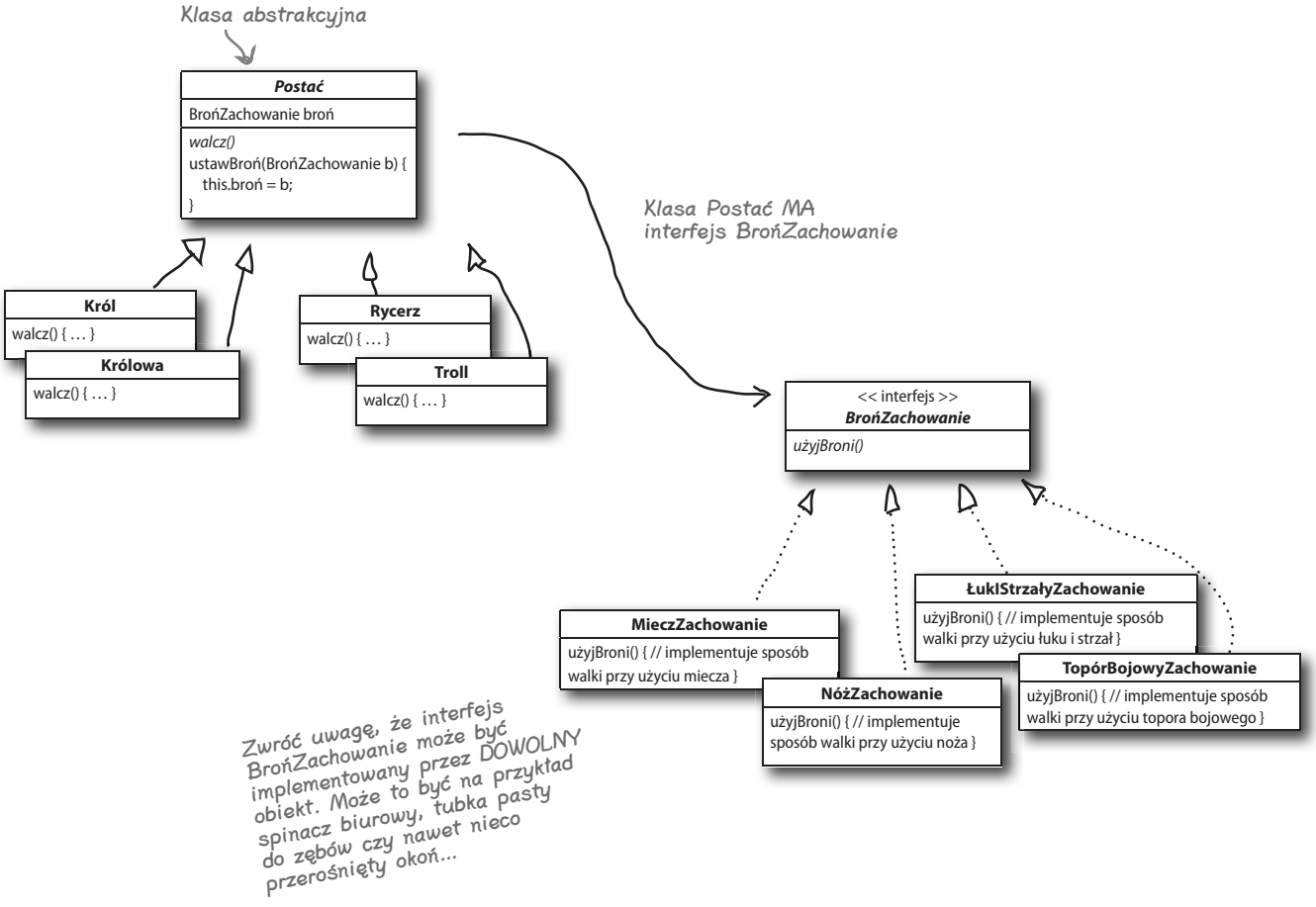
CELNE SPOSTRZEŻENIA

- Znajomość podstaw programowania obiektowego nie czyni Cię automatycznie dobrym projektantem obiektowym.
- Dobre projekty obiektowe nadają się do wielokrotnego użycia, zapewniają łatwość rozbudowy i utrzymania.
- Wzorce pokazują sposoby tworzenia całych systemów posiadających cechy charakterystyczne dobrych projektów obiektowych.
- Wzorce projektowe znakomicie sprawdziły się w wielu rzeczywistych aplikacjach systemów obiektowych.
- Wzorce nie udostępniają Ci gotowego kodu, a jedynie ogólne sposoby rozwiązywania problemów pojawiających się w fazie projektowania. Sam musisz zaadaptować je do konkretnych aplikacji.
- Wzorce nie są *wynalazkami*, wzorce się *odkrywa*.
- Większość wzorców i reguł odnosi się do sytuacji, w których w danym oprogramowaniu muszą zostać wprowadzone określone *zmiany*.
- Większość wzorców umożliwia modyfikowanie pewnych fragmentów systemu całkowicie niezależnie od jego pozostałych elementów.
- Bardzo często staramy się zidentyfikować elementy, które się w danym systemie zmieniają, a następnie próbujemy dokonać ich hermetyzacji.
- Wzorce zapewniają rodzaj wspólnego, jednolitego języka, który może maksymalizować efektywność komunikacji pomiędzy poszczególnymi członkami zespołu projektowego.

Rozwiązanie łamigłówki projektowej

Klasa Postać jest klasą abstrakcyjną dla wszystkich pozostałych klas (postaci), takich jak Król, Królowa, Rycerz oraz Troll. Z kolei BrońZachowanie jest interfejsem zaimplementowanym dla wszystkich klas opisujących poszczególne rodzaje broni. Jak łatwo się zorientować, wszystkie postaci oraz poszczególne rodzaje broni zostały zaimplementowane jako osobne klasy.

Aby zmienić rodzaj używanej broni, każda z postaci musi wywołać metodę ustawBroń() zdefiniowaną w klasie nadrzędnej Postać. Podczas walki poszczególne postaci wywołują metodę użyjBroni(), dzięki czemu mogą wykorzystywać aktualnie wybraną broń do zadawania śmiertelnych ciosów innym postaciom.



Zaostrz ołówek
Rozwiązanie

Które z poniżej wymienionych negatywnych zjawisk będzie rezultatem zastosowania mechanizmu dziedziczenia do tworzenia poszczególnych zachowań kaczki? (Zaznacz wszystkie pasujące). Poniżej przedstawiamy rozwiązanie:

- ☑ A. Kod jest powielany w klasach podrzędnych.
- ☑ B. Wprowadzanie zmian w zachowaniu programu jest trudne.
- ☐ C. Nie możemy sprawić, by kaczki tańczyły.
- ☑ D. Trudno zebrać informacje o zachowaniach wszystkich kaczek.
- ☑ E. Kaczki nie mogą jednocześnie latać i kwakać.
- ☐ F. Wprowadzane zmiany mogą mieć niezamierzony wpływ na inne kaczki.

Zaostrz ołówek
Rozwiązanie

Wypisz kilka powodów, dla których musiałeś dokonywać modyfikacji kodu w swoich aplikacjach. Poniżej podaliśmy kilka przykładów, choć Twoje powody na pewno będą inne. Wyglądają znajomo? Oto nasze powody:

Klienci lub użytkownicy zdecydowali, że chcą otrzymać coś zupełnie nowego bądź też rozszerzyć możliwości funkcjonalne istniejącego oprogramowania.

Zarząd firmy podjął decyzję, że będziemy korzystać z zupełnie nowej bazy danych innego producenta, a same dane zostaną dostarczone przez dostawcę, który oczywiście wykorzystuje zupełnie inny ich format. Auć!

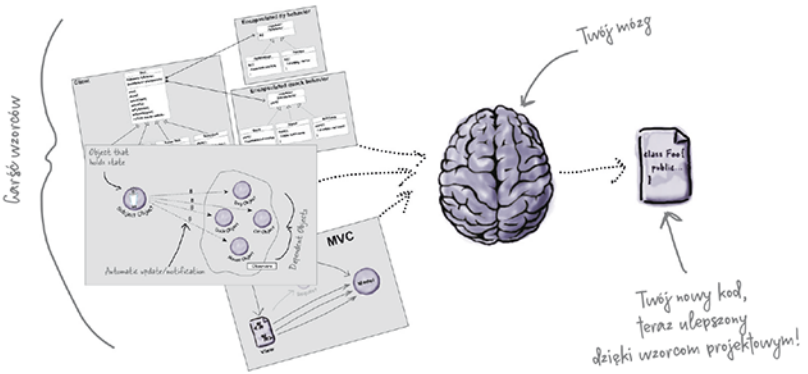
Taaak... nastąpiły zmiany w technologii i, niestety, musimy dokonać aktualizacji kodu, tak aby można było skorzystać z nowych protokołów.

Podczas budowania naszego systemu zdobyliśmy tak wiele cennych doświadczeń, że chcielibyśmy teraz cofnąć się do początku, rozpocząć wszystko od nowa i zrobić to trochę lepiej.

Skorowidz

Witaj w Obiektowie – i ciesz się każdą nową klasą!

Nie warto wyważać otwartych drzwi ani ponownie wynajdywać koła. Sprytniej jest skorzystać ze sprawdzonych rozwiązań, które ktoś już opracował i wdrożył. Dlatego właśnie mądrzy programiści lubią wzorce projektowe: to jest ich sekretny sposób, aby nie tracić sił na nudne drobiazgi, tylko zająć się trudniejszymi, ważniejszymi i ciekawszymi zadaniami. Nic dziwnego, że powstało naprawdę bardzo dużo wzorców projektowych. Przegląd ich wszystkich byłby niemożliwy. Które z nich więc są najpotrzebniejsze w programowaniu obiektowym i kiedy dokładnie z nich skorzystać?



Odpowiedź na to fundamentalne pytanie znajdziesz w tej książce – drugim wydaniu świetnego podręcznika dla przyszłych mistrzów kodu. Zostało ono, podobnie jak inne pozycje z serii *Rusz głową!*, przygotowane zgodnie z najnowszymi odkryciami nauk poznawczych, teorii uczenia się i neurofizjologii. I właśnie dzięki temu zaangażujesz swój mózg, wykorzystasz wiele zmysłów i niepostrzeżenie poznasz najprzydatniejsze i najciekawsze wzorce projektowe stosowane w programowaniu zorientowanym obiektowo. Tak złożone tematy jak klasy, interfejsy, kolekcje czy łączenie wzorców staną się jasne i zrozumiałe. Tego wszystkiego będziesz się uczyć, rozwiązując łamigłówki, wykonując praktyczne ćwiczenia, pisząc aplikacje – i wybuchając głośnym śmiechem!

Dostałem książkę wczoraj i zacząłem ją czytać w drodze do domu... nie mogłem przestać. Wziąłem ją ze sobą na siłownię i przypuszczam, że wszyscy widzieli, jak często się uśmiechałem podczas czytania. Naprawdę *cool*! To zabawna książka, a przy tym porusza bardzo wiele zagadnień i jest niezwykle celna. Jestem pod wielkim wrażeniem

– Erich Gamma
IBM Distinguished Engineer i współautor książki *Wzorce projektowe. Elementy oprogramowania obiektowego wielokrotnego użytku* napisanej przez Gang Czterech, wraz z Richardem Helmem, Ralphem Johnsonem i Johnem Vlissidesem

Czuję, jakby z mojej głowy usunięto ciężar tysiąca książek

– Ward Cunningham
twórca wiki i założyciel Hillside Group

Dr Eric Freeman – informatyk, badacz, autor książek technicznych. Wcześniej był dyrektorem technicznym w Walt Disney Company. Wraz z rodziną mieszka w Austin w stanie Teksas.

Elisabeth Robson – programistka, instruktorka i autorka książek. Współzałożycielka firmy WickedlySmart. Zajmuje się tworzeniem produktów edukacyjnych dla profesjonalnych programistów.

ISBN 978-83-283-7875-9
9 788328 378759
Cena: 109,00 zł