Poznaj zasady stosowania wzorców projektowych

Rusz głową!

Wzorce projektowe

Jak unikać niebezpiecznych związków

Odkrywamy sekrety Mistrza Wzorców

Jak sieć Star Cafe podwoiła swoją wartość dzięki wzorcowi Decorator

Dlaczego powszechna opinia o wzorcu Factory mija się z rzeczywistością

Wzorce projektowe i nowoczesne metody nauki

Jak unikanie dziedziczenia zmieni Twoje życie

Eric Freeman, Elisabeth Freeman
Bert Bates, Kathy Sierra

O'REILLY®

Helion

Wzorce projektowe. Rusz głową!

Autorzy: Elisabeth Freeman, Eric Freeman, Bert Bates, Kathy Sierra
Tłumaczenie: Adam Balcerzak, Marcin Jędrysiak, Tomasz Wasilewski
ISBN: 978-83-246-2803-2
Tytuł oryginału: Head First Design Patterns
Format: 200×230, stron: 656

Otwórz swój umysł. Poznaj wszystko, co związane z wzorcami projektowymi, w sposób gwarantujący szybkie i skuteczne opanowanie zasad ich stosowania. Zapomnij o listingach liczących tysiące wierszy, długich i nużących opisach teoretycznych oraz rozbudowanych schematach zależności. Wzorce projektowe to gotowe opisy rozwiązań najczęstszych problemów napotykanych przy tworzeniu oprogramowania. Aby je prawidłowo stosować, należy poznać założenia, na podstawie których zostały stworzone, oraz nauczyć się implementować je we właściwy sposób.

Naprzód, głowo!

Nikt ci tego nie potrafił wytłumaczyć? Wydaje Ci się, że to problem nie na Twoją głowę? Nie potrzebujesz elektrowstrząsów, żeby pobudzić swój mózg do aktywnego działania. Tylko żadnych gwałtownych gestów! Usiądź wygodnie, otwórz książkę, dopiero teraz się zacznie. Na początek – rusz głową!

Precz z nudnymi wykładami i zakuwaniem bez zrozumienia!

Nauka to znacznie więcej niż tylko czytanie suchego tekstu. Twój mózg jest niczym głodny rekin, cały czas pracy naprzód w poszukiwaniu nowej, apetycznej przekąski. Jak karmimy Twój wygłodniały umysł?

Używamy rysunków, bo obraz wart jest 1024 słów. Stosujemy powtórzenia, by zakodować na stałe dane w Twojej chłonnej głowie. Oddziałujemy na emocje, jesteśmy nieprzewidywalni, zaskakujący i zabawni. Stawiamy przed Tobą wyzwania i zadajemy pytania, które angażują Cię w proces studiowania przedstawianych zagadnień. Cały czas pobudzamy Twój umysł do aktywnego działania, zmuszamy go do posłuszeństwa... a za ciężką pracę nagrodzimy go smakowitym ciasteczkiem w postaci wiedzy – wisienka gratis!

Rozkmiń to sam!

- Cele stosowania wzorców projektowych
- Założenia, na których opierają się wzorce projektowe
- Najważniejsze i najczęściej wykorzystywane wzorce projektowe
- Przechowywanie i prezentacja danych
- Mechanizm RMI
- Wzorzec MVC
- Implementacja wzorców projektowych w aplikacjach

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

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

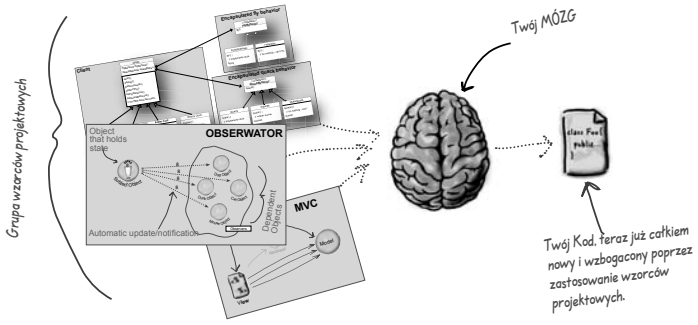
Spis treści (skrócony)

Spis treści (na serio)

Wprowadzenie do wzorców projektowych

1 Witamy w krainie wzorców projektowych

Ktoś rozwiązał już Twoje problemy. W tym rozdziale dowiesz się, dlaczego (i w jaki sposób) możesz wykorzystać wiedzę i doświadczenia zdobyte przez innych projektantów i programistów, którzy podczas pracy nad różnymi projektami zmuszeni byli wstąpić na pełną zdradliwych pułapek ścieżkę i — co najważniejsze — udało im się przeżyć taką wyprawę. Zanim dobrniemy do końca rozdziału, rzucimy okiem na sposoby wykorzystywania wzorców projektowych i przedstawimy ich zalety, poznamy kilka podstawowych zasad projektowania zorientowanego obiektowo, a także omówimy sposób działania przykładowego wzorca. Najlepszą metodą zastosowania wzorca jest *załadowanie go bezpośrednio do Twojego mózgu*, a następnie *zlokalizowanie* obszarów w obrębie projektowanych rozwiązań oraz istniejących aplikacji, w których możesz je *zastosować*. Pracując z wzorcami projektowymi, zamiast wielokrotnego wykorzystywania tych samych fragmentów kodu, wielokrotnie wykorzystujesz swoje *doświadczenia*.

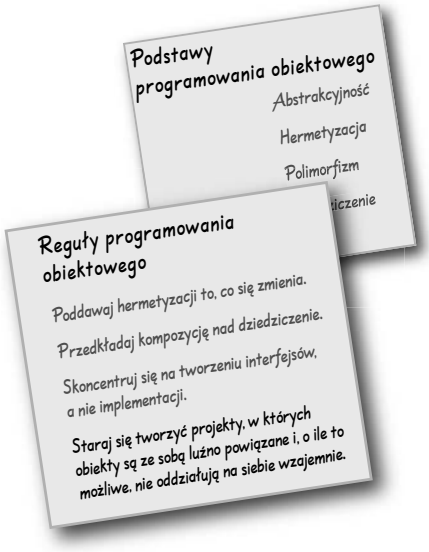


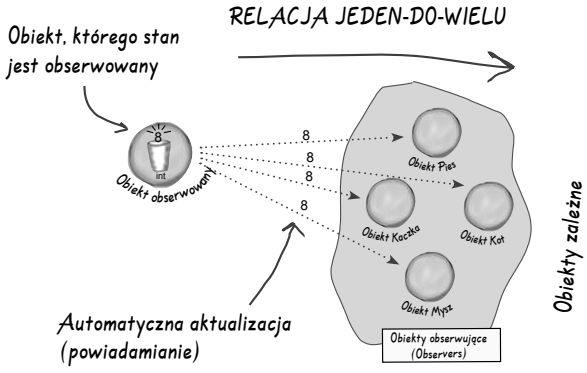
Wzorzec Obserwator

2 Jak sprawić, by Twoje obiekty były zawsze dobrze poinformowane

Nie przegap okazji, kiedy dzieje się coś naprawdę ciekawego!

Przedstawimy Ci wzorzec, który potrafi poinformować inne obiekty o tym, że wydarzyło się coś, czym powinny się zająć. Co ciekawe, obiekty mogą nawet samodzielnie decydować w czasie działania programu o tym, czy chcą być informowane o takich wydarzeniach. Wzorzec Obserwator jest jednym z najczęściej wykorzystywanych wzorców w pakiecie JDK (ang. *Java Development Kit*), a co najważniejsze — jest wręcz niewiarygodnie użyteczny. W niniejszym rozdziale rzucimy również okiem na relacje typu jeden-do-wielu oraz tzw. luźne związki (tak, to prawda, napisaliśmy „luźne związki"). Korzystając z wzorca Obserwator, z pewnością odmienisz swoje życie.





Wzorzec Dekorator

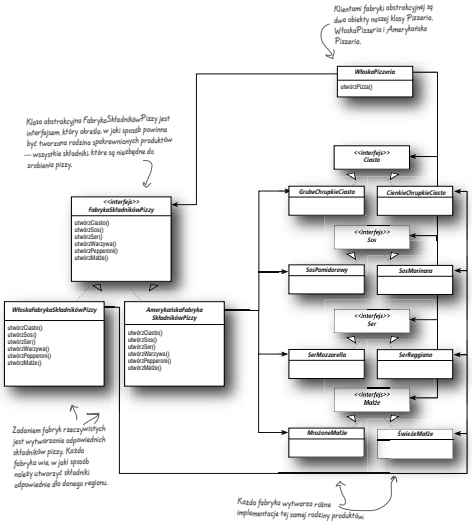
3 Dekorowanie zachowania obiektów

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

Wzorzec Fabryka

4 Pizzeria zorientowana obiektowo

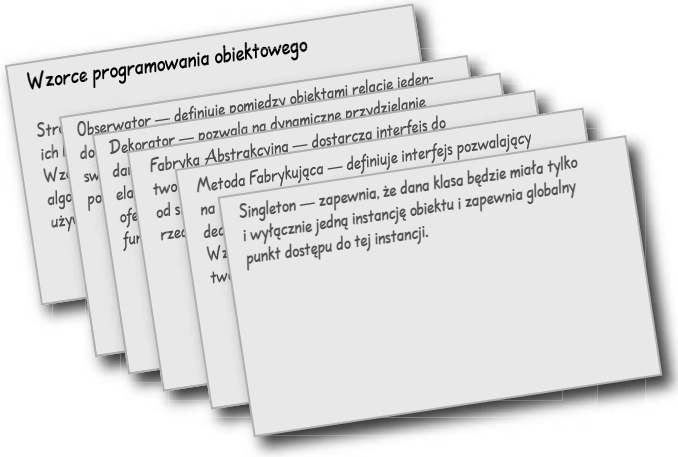
Przygotuj się do stworzenia kilku projektów, w których zastosujemy luźne powiązania pomiędzy poszczególnymi obiektami. Stworzenie nowego obiektu to dużo więcej niż tylko proste zastosowanie operatora new. Niebawem przekonasz się, że proces ten jest operacją, która nie zawsze powinna być publicznie dostępna,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



5 Obiekty jedyne w swoim rodzaju

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

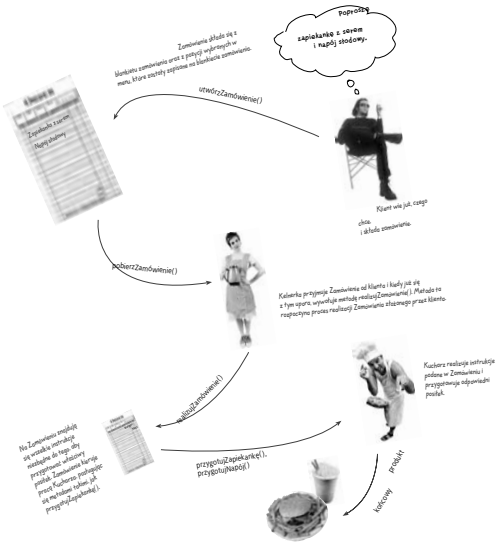
Hershey, PA



Wzorzec Polecenie

6 Hermetyzacja wywołań

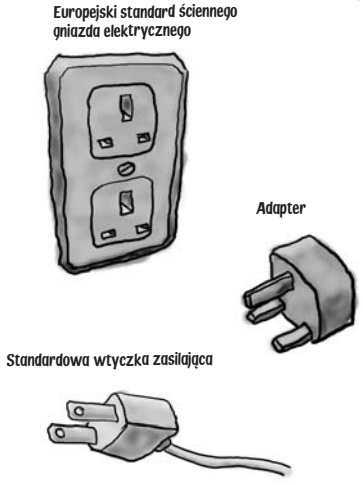
W niniejszym rozdziale przeniesiemy hermetyzację na zupełnie nowy poziom: mamy zamiar dokonać hermetyzacji wywołań metod. Zgadza się, dzięki hermetyzacji wywołań metod możemy wykrystalizować pewne fragmenty obliczeń tak, że obiekt wywołujący obliczenia nie musi się martwić, w jaki sposób je wykonać;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

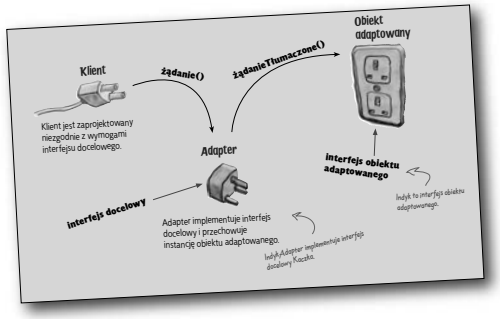


Wzorce Adapter oraz Fasada

7 Zdolność do adaptacji

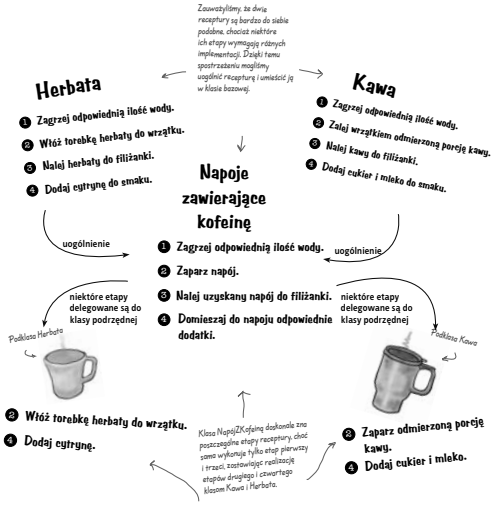
W niniejszym rozdziale mamy zamiar dokonać paru niesamowitych wyczynów z dziedziny rzeczy niemożliwych, takich jak na przykład włożenie kwadratowego kołka do okrągłego otworu. Brzmi nierealnie? Nie wtedy, kiedy mamy pod ręką odpowiednie wzorce projektowe. Pamiętasz wzorzec Dekorator? Podczas pracy z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





8 Hermetyzacja algorytmów

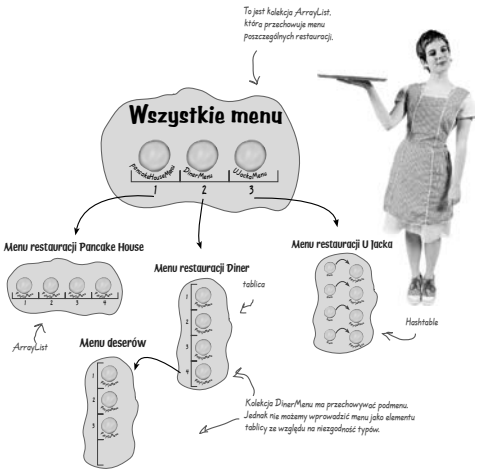
Jesteśmy jak w transie: hermetyzowaliśmy już proces tworzenia obiektów, wywołania metod, złożone interfejsy, kaczki, indyki, pizze… ciekawe, co będzie następne? Otóż, teraz mamy zamiar zająć się hermetyzacją fragmentów algorytmów, tak aby klasy podrzędne mogły „podczepiać się" w różnych miejscach wykonywanych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



Wzorce Iterator i Kompozyt

9 Zarządzanie kolekcjami

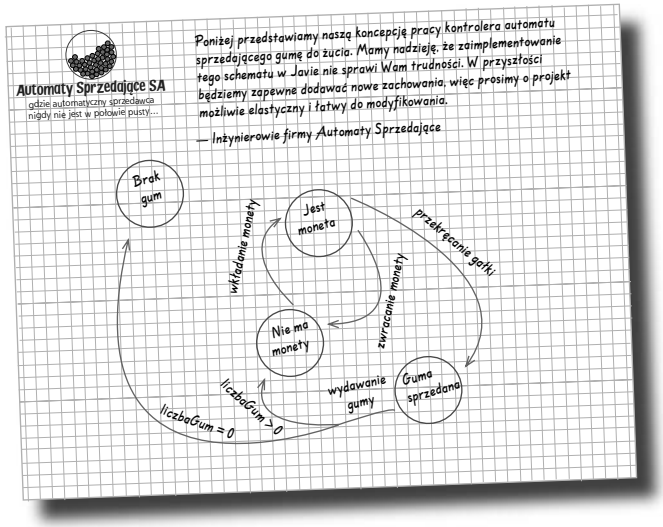
Jest wiele sposobów grupowania obiektów w kolekcje. Można utworzyć obiekty Array, Stack, List, Hashtable. Każdy z nich ma swoje zalety i wady. Jednak w pewnym momencie klient rozpocznie iteracyjne przetwarzanie elementów kolekcji. Czy wtedy ujawnisz mu swoją implementację? Mam nadzieję, że nie. To nie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



Wzorzec Stan

10 Stan obiektu

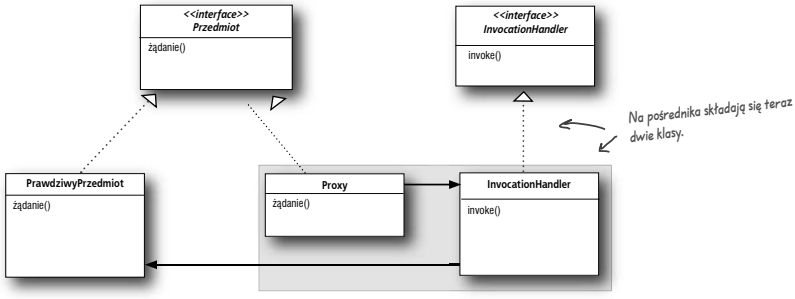
Mało znany fakt: wzorce Strategy i State to bliźniaki, rozdzielone zaraz po narodzinach. Jak już wiemy, wzorzec Strategy umożliwił przeprowadzenie wielu niezwykle udanych przedsięwzięć opartych na zamiennie stosowanych algorytmach. Wzorzec State ma inną rolę. Jest nią wspomaganie obiektów w kontrolowaniu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



Wzorzec Proxy

11 Kontrola dostępu do obiektu

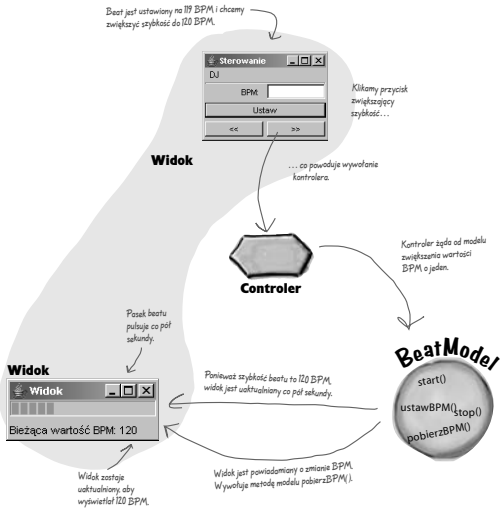
Próbowałeś kiedyś stosować metodę „dobrego i złego”? Ty jesteś tym dobrym, który zrobi wszystko, o co się go poprosi, który jest zawsze miły i uprzejmy. Nie chcesz jednak, żeby *każdy* mógł prosić o Twoje usługi. To jest miejsce dla „złego”, który będzie *kontrolował dostęp* do Ciebie. Takie jest właśnie zadanie pośredników (ang. *proxy*) w modelu obiektowym — kontrolowanie i zarządzanie dostępem. Jak się przekonamy, istnieje *bardzo wiele* schematów takiego pośrednictwa. Obiekty Proxy mogą przekazywać wywoływanie metody obiektowi w innym węźle internetu; bywa też, że zastępują wyjątkowo leniwe obiekty.



Wzorce złożone

12 Łączenie wzorców

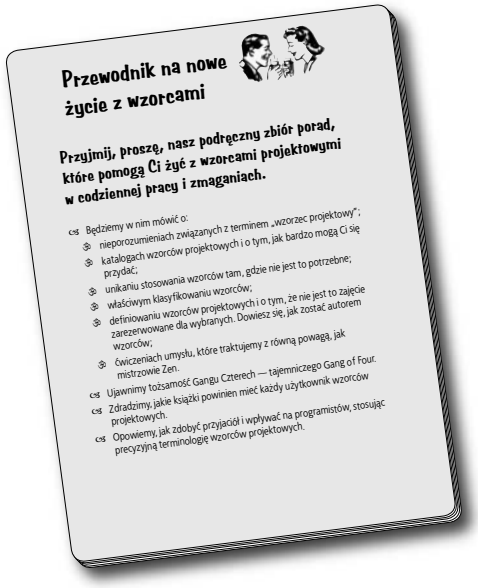
Przyszłoby Ci do głowy, że wzorce mogą pracować razem? Byliśmy już świadkami wielu niespokojnych „Pogawędek przy kominku" (a ominął Cię „Death Match" wzorców, który wydawca kazał wyrzucić) — czy wsłuchując się w ich ton, można jeszcze liczyć na to, że wzorce będą ze sobą współpracować? Możesz wierzyć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

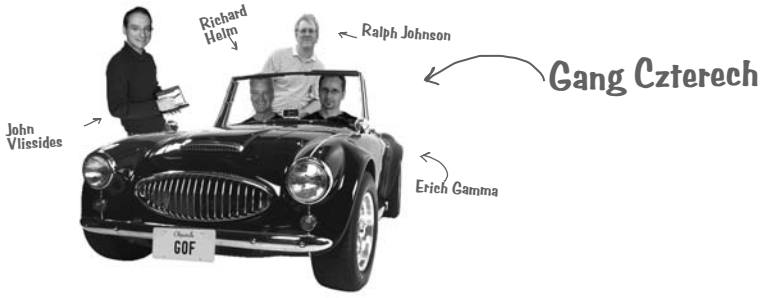


Nowe życie z wzorcami

13 Wzorce projektowe w praktyce

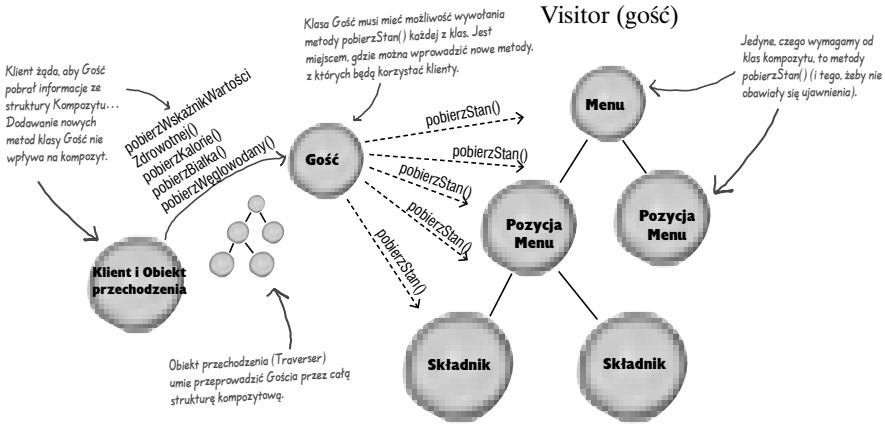
Ach, jesteś już gotowy na spotkanie z nowym wspaniałym światem pełnym wzorców projektowych… Ale zanim rozpoczniesz wędrówkę ku nowym horyzontom, poświęć chwilę na przeczytanie rozdziału poświęconego pewnym szczególnym kwestiom, które pojawiają się, gdy rozpoczynasz stosowanie wzorców w codziennej pracy.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





14 Dodatek — inne wzorce

Nie wszyscy mogą być sławni. Przez ostatnie dziesięć lat wiele się w świecie wzorców zmieniło. Od czasu pierwszego wydania książki *Design Patterns: Elements of Reusable Object-Oriented Software (Wzorce projektowe)* programiści wykorzystali opisane w nich schematy tysiące razy. Wzorce, które zebraliśmy w tym dodatku, to dopracowane, kompletne, „oficjalne" wzorce grupy GoF. Różnią się od wcześniej opisanych tylko tym, że nie spotkamy ich tak często, jak tych, którym poświęciliśmy całe rozdziały. Nie umniejsza to ich zalet i nie powinno zniechęcać do ich stosowania tam, gdzie wymaga tego sytuacja. Celem niniejszego dodatku jest zapewnienie Ci szerszej orientacji w najłatwiej dostępnych zasobach zgromadzonej przez lata wiedzy.



Jak sprawić, by Twoje obiekty były zawsze dobrze poinformowane

Nie przegap okazji, kiedy dzieje się coś naprawdę ciekawego! Przedstawimy Ci wzorzec, który potrafi poinformować inne obiekty o tym, że wydarzyło się coś, czym powinny się zająć. Co ciekawe, obiekty mogą nawet samodzielnie decydować w czasie działania programu o tym, czy chcą być informowane o takich wydarzeniach. Wzorzec Obserwator jest jednym z najczęściej wykorzystywanych wzorców w pakiecie JDK (ang. *Java Development Kit*), a co najważniejsze — jest wręcz niewiarygodnie użyteczny. W niniejszym rozdziale rzucimy również okiem na relacje typu jeden-do-wielu oraz tzw. luźne związki (tak, to prawda, napisaliśmy „luźne związki"). Korzystając z wzorca Obserwator, z pewnością odmienisz swoje życie.

Gratulacje!

Wasz zespół właśnie wygrał kontrakt na budowę Pogodynki 2, najnowszej generacji internetowej stacji sprawdzającej i podającej aktualną pogodę.

Pogodynka sp. z o.o.
ul. Deszczowa 25
00-000 Śniegowo

Oświadczenie o współpracy

Chcielibyśmy złożyć serdeczne gratulacje z okazji wygrania przez Państwa firmę kontraktu na budowę naszej najnowszej generacji internetowej stacji meteorologicznej!

Stacja meteorologiczna oparta będzie na naszym znakomitym, opatentowanym obiekcie DanePogodowe, który automatycznie śledzi bieżące warunki pogodowe (temperatura otoczenia, wilgotność oraz ciśnienie atmosferyczne). Chcielibyśmy, aby Wasza firma utworzyła aplikację, która początkowo będzie miała za zadanie wyświetlać na ekranie trzy główne elementy: informację o bieżących warunkach pogodowych, dane statystyczne o pogodzie oraz prostą prognozę pogody, wszystkie aktualizowane w czasie rzeczywistym w chwili, gdy obiekt DanePogodowe otrzyma najnowsze odczyty z urządzeń pomiarowych.

Musimy jednak pamiętać o tym, że jest to perspektywiczna wersja stacji meteorologicznej. Nasza firma chce w najbliższej przyszłości wypuścić na rynek wersję API do stacji Pogodynka 2, która umożliwi innym projektantom tworzenie własnych sposobów wyświetlania informacji o pogodzie i łatwe dołączanie takich modułów do głównego oprogramowania stacji. Chcielibyśmy, aby Wasza firma dostarczyła nam pierwszą wersję takiego API.

Uważamy, że nasza firma realizuje znakomity biznesplan: po podpisaniu umowy z klientem mamy zamiar obciążać go dodatkowymi kosztami za każdy panel wyświetlający informacje z naszej stacji. A teraz najlepsza część: oczekujemy, że akceptowaną przez Was formą płatności będą opcje na zakup akcji naszej wspaniałej firmy!

Z niecierpliwością oczekujemy na projekt stacji oraz wersję alfa aplikacji sterującej.

Z poważaniem,

Janusz Monsun

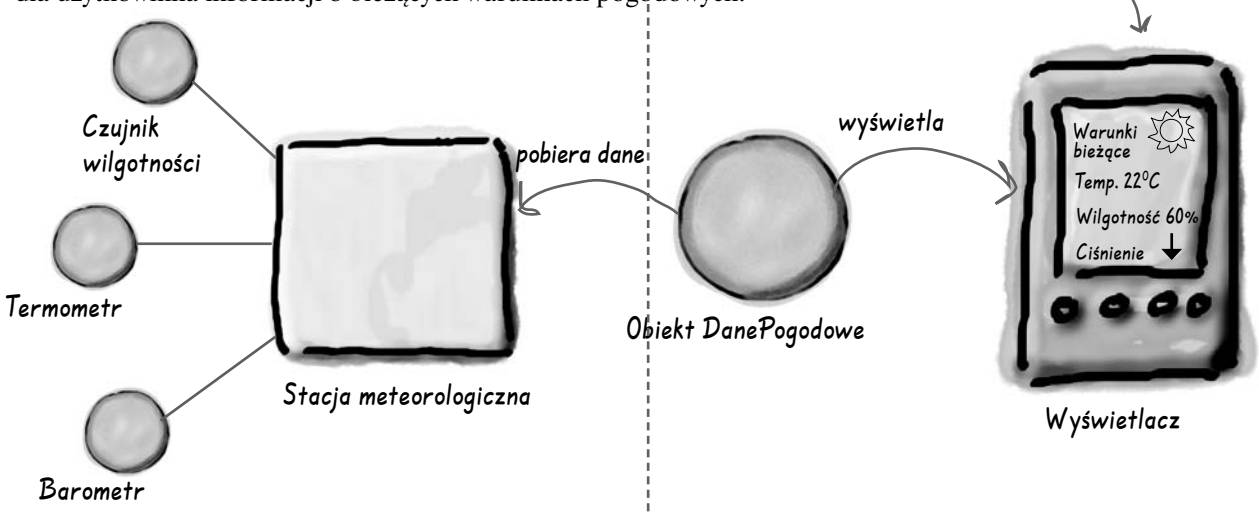
Janusz Monsun, Prezes

P.S. Pliki źródłowe zawierające kod naszego opatentowanego obiektu DanePogodowe wyślemy do Państwa kurierem jeszcze dziś w nocy.

Ogólne spojrzenie na aplikację sprawdzającą warunki pogodowe

Trzema głównymi graczami w naszym systemie są: stacja meteorologiczna (fizyczne urządzenie zbierające dane z czujników temperatury, ciśnienia i wilgotności), obiekt DanePogodowe (który zapewnia śledzenie danych nadchodzących z tej stacji oraz aktualizuje informacje wyświetlane na ekranie) oraz sam wyświetlacz, którego zadaniem jest zobrazowanie w postaci czytelnej dla użytkownika informacji o bieżących warunkach pogodowych.

„Warunki bieżące" to jeden z dostępnych trybów wyświetlania informacji. Użytkownik może również zażyczyć sobie wyświetlenia danych statystycznych o pogodzie dla naszego regionu (tryb „Statystyka") lub też prostej prognozy pogody (tryb „Prognoza").



Elementy dostarczane przez firmę Pogodynka sp. z o.o.

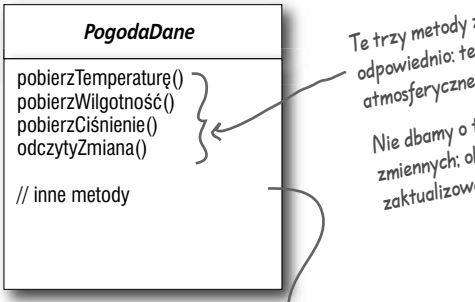
Elementy, które musimy zaimplementować

Obiekt DanePogodowe „wie", w jaki sposób należy komunikować się z fizyczną stacją meteorologiczną tak, aby otrzymywać aktualne odczyty danych. Po otrzymaniu danych obiekt aktualizuje wyświetlane informacje dla trzech głównych trybów pracy: „Warunki bieżące" (tryb ten pokazuje informacje na temat temperatury otoczenia, wilgotności oraz ciśnienia atmosferycznego), „Statystyka" oraz „Prognoza".

Jeżeli zdecydujemy się na zaakceptowanie tego kontraktu, naszym zadaniem będzie stworzenie aplikacji, która będzie wykorzystywała obiekt DanePogodowe do aktualizacji informacji wyświetlanych w trzech głównych trybach pracy.

Rozpakowywanie klasy DanePogodowe

Jak zostało obiecane, następnego dnia rano kurier dostarczył pliki źródłowe klasy DanePogodowe. Kiedy rzuciliśmy okiem na jej kod, wszystko stało się jasne:



Te trzy metody zwracają aktualne odczyty z czujników, odpowiednio: temperatury, wilgotności oraz ciśnienia atmosferycznego.

Nie dbamy o to, JAK ustawiane są wartości tych zmiennych; obiekt PogodaDane „wie", w jaki sposób otrzymać zaktualizowane informacje ze stacji meteorologicznej.

Jak widać, projektanci klasy DanePogodowe zostawili nam wskazówkę na temat tego, co musimy tutaj dołożyć...

```
/*
* Ta metoda jest wywoływana
* za każdym razem, kiedy zostaną
 zaktualizowane
* odczyty z czujników pogody
*
*/
 public void odczytyZmiana() {
     // tutaj umieść odpowiedni kod
}
```

plik DanePogodowe.java

Pamiętaj, że tryb „Warunki bieżące" jest tylko JEDNYM z trzech dostępnych trybów wyświetlania informacji.

Wyświetlacz

Nasze zadanie polega na implementacji metody odczytyZmiana(), tak aby była ona w stanie automatycznie aktualizować informacje wyświetlane w trybach „Warunki bieżące", „Statystyka" oraz „Prognoza".

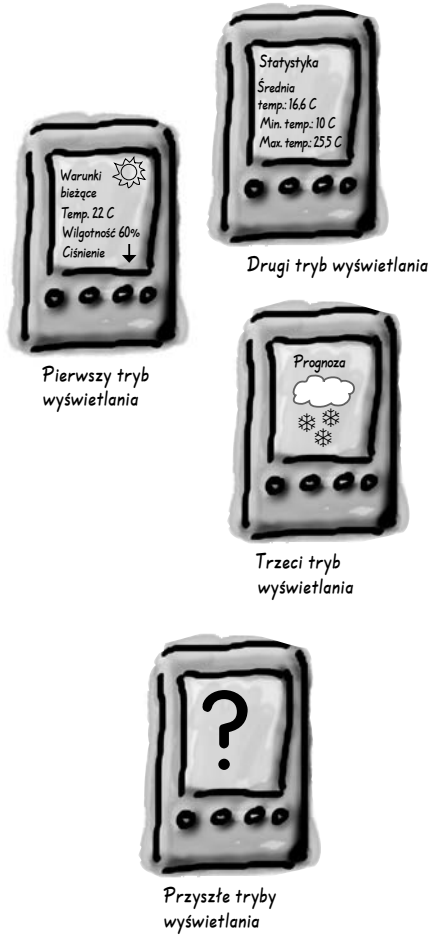
O czym wiemy do tej pory?

Specyfikacja nadesłana z firmy Pogodynka sp. z o.o. nie była do końca jasna, więc musieliśmy się zorientować, co tak naprawdę mamy zrobić. Podsumujmy zatem, czego dowiedzieliśmy się dotychczas:

- Klasa DanePogodowe posiada odpowiednie metody, umożliwiające pobieranie trzech mierzonych wartości (temperatury otoczenia, wilgotności oraz ciśnienia atmosferycznego) z czujników fizycznych.
- Metoda odczytyZmiana() jest wywoływana za każdym razem, kiedy pojawiają się nowe dane z czujników fizycznych (nie wiemy, a w zasadzie nawet nie dbamy o to, w jaki sposób ta metoda jest wywołana; wiemy tylko, że tak *po prostu działa*).
- Musimy zaimplementować obsługę trzech trybów wyświetlania wykorzystujących dane o pogodzie: trybu *Warunki bieżące*, trybu *Statystyka* oraz trybu *Prognoza*. Dane wyświetlane w poszczególnych trybach muszą być na bieżąco aktualizowane za każdym razem, kiedy obiekt DanePogodowe zyskuje odczyty nowych pomiarów.
- Cały system musi być łatwy do rozbudowy — inni projektanci mogą tworzyć swoje własne, nowe tryby wyświetlania danych, a użytkownicy mogą dodawać do aplikacji bądź usuwać z niej dowolną ilość trybów wyświetlania. W chwili obecnej znamy tylko *trzy* początkowe tryby wyświetlania danych („Warunki bieżące", „Statystyka" oraz „Prognoza").

```
pobierzTemperaturę()
pobierzWilgotność()
pobierzCiśnienie()


odczytyZmiana()
```



Pierwszy, pozorny sukces w walce ze stacją meteorologiczną

Poniżej przedstawiamy pierwszy wariant implementacji naszego systemu — skorzystamy tu z porady, jaką nam przekazali programiści z firmy Pogodynka sp. z o.o., i umieścimy nasz kod wewnątrz metody odczytyZmiana():

```
public class DanePogodowe {
    // deklaracje zmiennych obiektowych
    public void odczytyZmiana() {

        float temp = pobierzTemperaturę();
        float wilgotność = pobierzWilgotność();
        float pressure = pobierzCiśnienie();

        warunkiBieżąceWyświetl.aktualizacja(temp, wilgotność, ciśnienie);
        statystykaWyświetl.aktualizacja(temp, wilgotność, ciśnienie);
        prognozaWyświetl.aktualizacja(temp, wilgotność, ciśnienie);
    }
    // w tym miejscu można wstawić inne metody obiektu PogodaDane
}
```

Pobierz najbardziej aktualne odczyty, wywołując odpowiednie metody umożliwiające pobieranie trzech mierzonych wartości z czujników fizycznych (metody zostały już wcześniej zaimplementowane).

Aktualizacja wyświetlanych informacji…

Wywołuje każdy wyświetlany element i przekazuje mu do wyświetlania najbardziej aktualne wartości odczytów z odpowiedniego czujnika.

Zaostrz ołówek

Opierając się na pierwszej implementacji naszego systemu, określ, które z wymienionych zdarzeń są prawdziwe. (Zaznacz wszystkie poprawne odpowiedzi).

A. ❑ Tworzymy poszczególne implementacje, a nie interfejsy.

B. ❑ Dodanie nowego trybu wyświetlania będzie każdorazowo wymuszało modyfikację kodu programu.

C. ❑ Nie mamy żadnych możliwości dodawania (lub usuwania) wybranych trybów wyświetlania podczas działania programu.

D. ❑ Poszczególne wyświetlane elementy nie posiadają wspólnego interfejsu.

E. ❑ Nie dokonaliśmy hermetyzacji tych elementów aplikacji, które się zmieniają.

F. ❑ Naruszyliśmy hermetyzację klasy DanePogodowe.

Co jest nie tak z naszą implementacją?

Powróć na chwilę myślami do tych wszystkich pojęć i reguł, o których wspominaliśmy w rozdziale 1...

```
public class PogodaDane {
    // deklaracje zmiennych obiektowych
    public void odczytyZmiana() {
        float temp = pobierzTemperaturę();
        float wilgotność = pobierzWilgotność();
        float pressure = pobierzCiśnienie();

        warunkiBieżąceWyświetl.aktualizacja(temp, wilgotność, ciśnienie);
        statystykaWyświetl.aktualizacja(temp, wilgotność, ciśnienie);
        prognozaWyświetl.aktualizacja(temp, wilgotność, ciśnienie);
    }
```

Obszar zmian, powinniśmy zatem dokonać jego hermetyzacji.

W przypadku tworzenia poszczególnych implementacji nie mamy żadnych możliwości dodawania ani usuwania wyświetlanych elementów bez dokonywania modyfikacji w kodzie programu.

Wygląda na to, że przynajmniej staramy się używać wspólnego, jednolitego interfejsu dla wyświetlanych elementów... Wszystkie posiadają metodę aktualizacja(), której argumentami są wartości temperatury otoczenia, wilgotności oraz ciśnienia atmosferycznego.

Rzucimy teraz okiem na wzorzec Obserwator, a następnie powrócimy do naszej aplikacji i pokażemy, w jaki sposób można zastosować go do aplikacji obsługującej stację meteorologiczną.

Spotkanie z wzorcem Obserwator

Zapewne wiesz, w jaki sposób działa prenumerata gazet i czasopism:

1. Wydawca nowej gazety lub czasopisma rozpoczyna swoją działalność i na rynku pojawia się nowy tytuł prasowy.

2. Zamawiasz prenumeratę u danego wydawcy i od tego momentu za każdym razem, kiedy pojawia się nowe wydanie, zostaje Ci ono dostarczone. Całość działa tak długo, jak długo pozostajesz prenumeratorem danego tytułu prasowego.

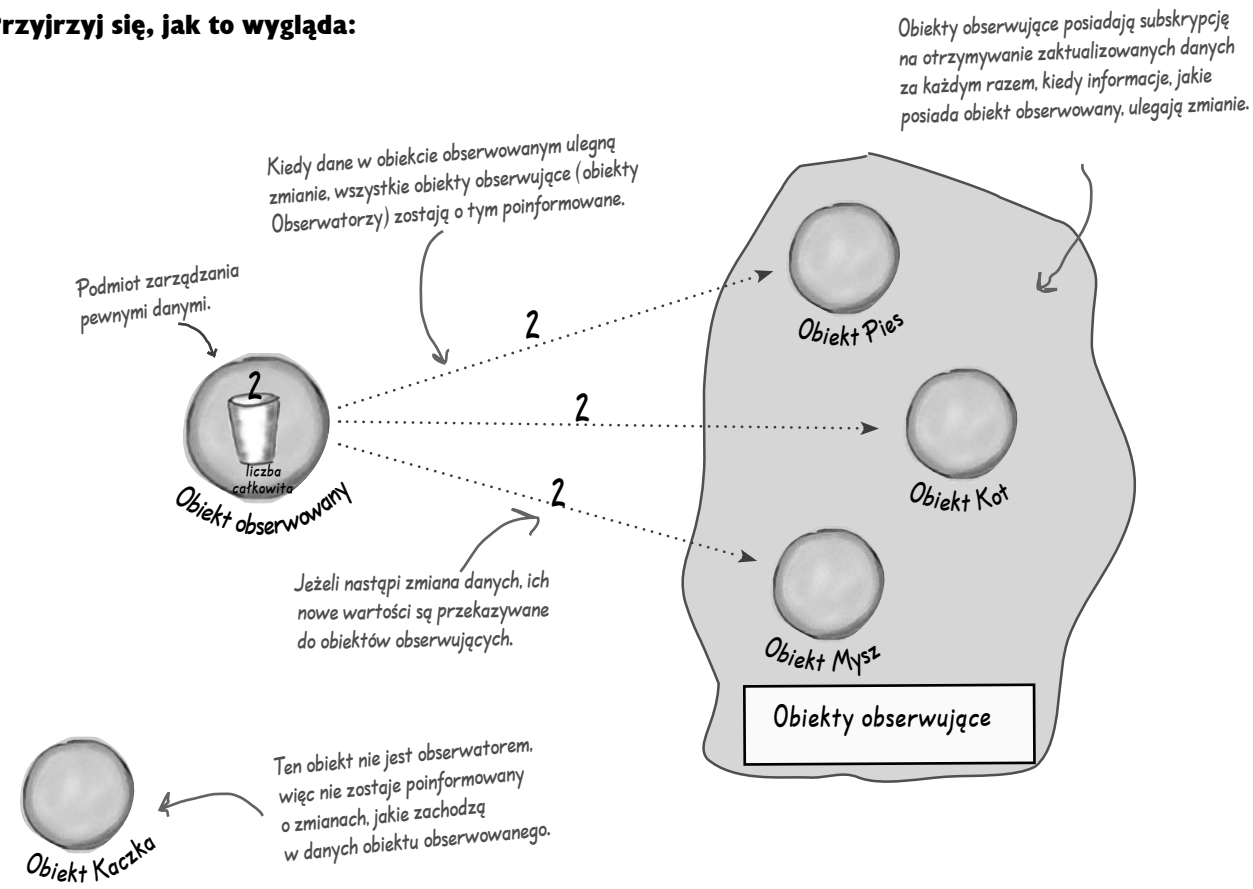
3. Jeżeli nie chcesz więcej otrzymywać danej gazety lub czasopisma, po prostu wypowiadasz prenumeratę i od tego momentu nowe wydania przestają do Ciebie docierać.

4. Jak długo dany wydawca (czy też dany tytuł prasowy) istnieje na rynku, tak długo różni ludzie, firmy, hotele, linie lotnicze itp. będą ustawicznie dokonywały nowych prenumerat i wypowiadały stare prenumeraty.

Wydawca + Prenumerator = wzorzec Obserwator

Jeżeli rozumiesz zasady prenumerowania gazet i czasopism, oznacza to, że całkiem dobrze rozumiesz zasady funkcjonowania wzorca Obserwator, z tym tylko, że we wzorcu zamiast wydawcy występuje PODMIOT (obiekt obserwowany), a zamiast prenumeratorów występują OBIEKTY OBSERWATORZY (obiekty obserwujące).

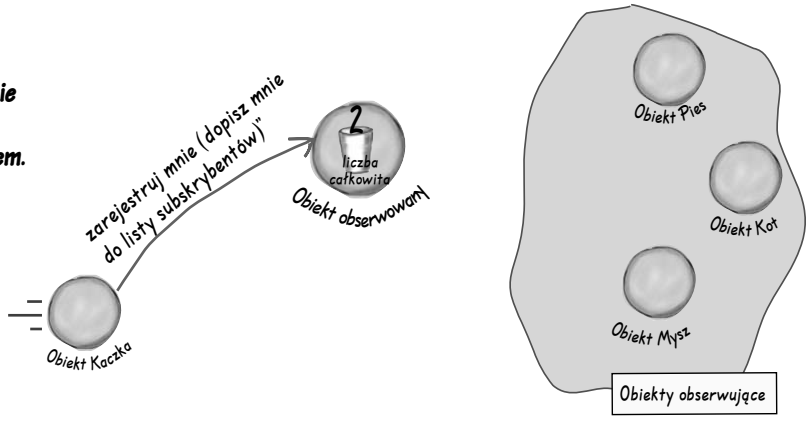
Przyjrzyj się, jak to wygląda:



Dzień z życia wzorca Obserwator

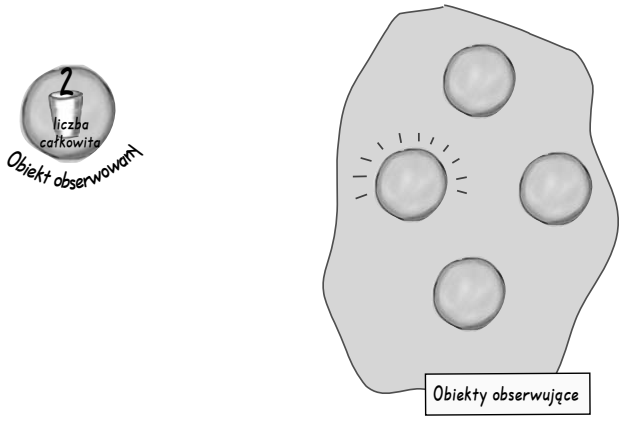
Obiekt Kaczka pojawia się i następnie informuje obiekt obserwowany, że chciałby zostać jego obserwatorem.

Obiekt Kaczka tak naprawdę chce tylko jednego: danych, jakie obiekt obserwowany wysyła za każdym razem, kiedy zmienia się jego stan.



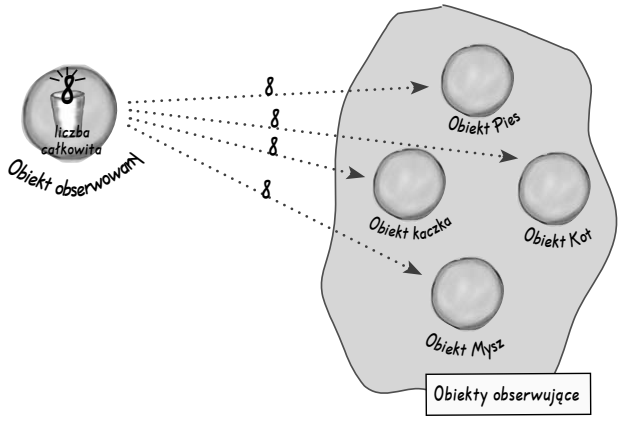
Obiekt Kaczka jest teraz oficjalnie uznanym obserwatorem.

Kaczka jest podekscytowana... znajduje się już na liście i z niecierpliwością oczekuje na nadejście pierwszych informacji od obiektu obserwowanego.



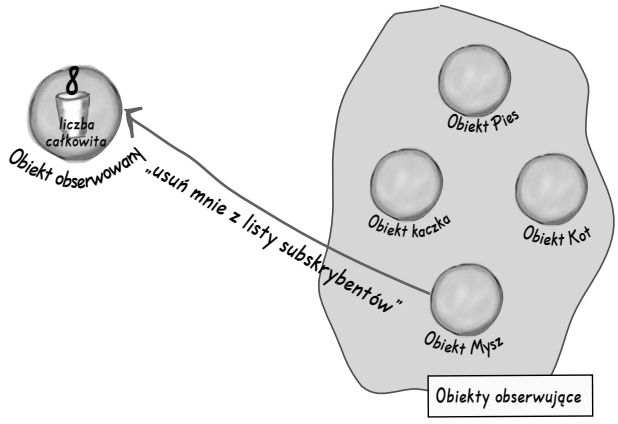
Obiekt obserwowany otrzymuje nowe dane!

W takiej sytuacji obiekt Kaczka oraz pozostałe obiekty obserwujące otrzymują powiadomienie, że obiekt obserwowany zmienił stan.



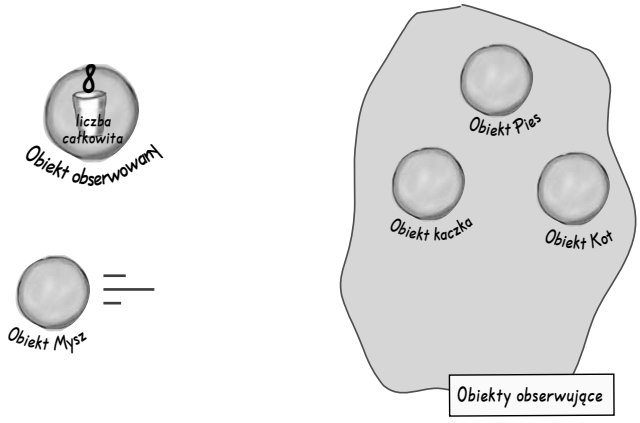
Obiekt Mysz prosi o usunięcie z listy obserwatorów.

Mysz otrzymywała aktualizacje danych z obiektu obserwowanego od wieków — nic dziwnego zatem, że w końcu jej się znudziło i postanowiła zrezygnować z bycia obserwatorem.



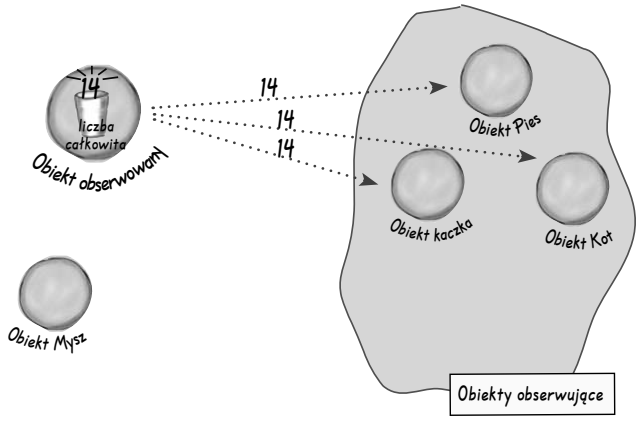
Myszy już tu nie ma!

Obiekt obserwowany potwierdza przyjęcia żądania Myszy i usuwa ją z listy obserwatorów.



Obiekt obserwowany ponownie zmienił stan.

Wszystkie obiekty obserwujące zostają o tym fakcie powiadomione; wyjątkiem jest tutaj Mysz, która już nie znajduje się na liście subskrybentów. Nie mów tego nikomu, ale tak naprawdę nasza Mysz w skrytości ducha tęskni za bieżącymi komunikatami… być może pewnego dnia wystąpi ona z prośbą o ponowne zaliczenie jej w poczet obserwatorów.



Pięciominutowe przedstawienie — obserwowany kontra obserwujący

W dzisiejszym skeczu dwóch zakręconych projektantów oprogramowania spotyka prawdziwego „łowcę talentów"...

Mówi Robert. Szukam nowego miejsca pracy w charakterze projektanta systemów opartych na Javie. Mam pięcioletnie doświadczenie oraz...

1

Projektant nr 1

Hmmm, oczywiście, Ty i kto tylko jeszcze będzie chciał, mój chłopcze. Właśnie umieszczam Cię na mojej liście projektantów systemów Java. Nie dzwoń do mnie, to ja do Ciebie oddzwonię!

2

Łowca talentów (Podmiot)

Cześć, mam na imię Joanna, napisałam już wiele systemów wykorzystujących EJB. Jestem zainteresowana dowolnym stanowiskiem pracy w dziale zajmującym się projektowaniem systemów Java.

3

Projektant nr 2

Umieszczę Twoje nazwisko na mojej liście. Dzięki temu będziesz informowana na bieżąco, razem z pozostałymi obserwatorami.

4

Obiekt obserwowany (Podmiot)

5 *W międzyczasie życie Roberta i Joanny biegło swoim zwykłym torem; jeżeli pojawiała się jakaś oferta pracy dla projektantów Javy, otrzymywali informacje — krótko mówiąc, zostali obserwatorami.*

6 **Obiekt obserwowany**

7 **Obserwator**

Obserwator

Joanna znajduje nowego pracodawcę.

8 **Obserwator**

9 **Obiekt obserwowany**

Dwa tygodnie później

Joanna wielce sobie ceni swoje obecne życie i nie musi już odgrywać roli obserwatora. Jest również bardzo zadowolona, że nowa firma zaproponowała jej bardzo dogodne warunki finansowe, tym lepsze, że nie musiała dodatkowo opłacić usług łowcy talentów.

Ale co się stało z naszym drogim Robertem? Słyszeliśmy, że pobił łowcę talentów na jego własnym podwórku. Jest teraz nie tylko obserwatorem, ale również posiada swoją własną listę subskrybentów (obserwatorów), których powiadamia o pojawieniu się nowych ofert na rynku pracy. Krótko mówiąc, Robert jest teraz zarówno obserwującym, jak i obserwowanym.

Definicja wzorca Obserwator

Jeżeli będziesz próbował w jakiś sposób zobrazować sobie wzorzec Obserwator, to zapewne szybko dojdziesz do wniosku, że przedstawiony wcześniej przykład z wydawnictwem i prenumeratą periodyków prasowych stanowi jego znakomitą wizualizację.

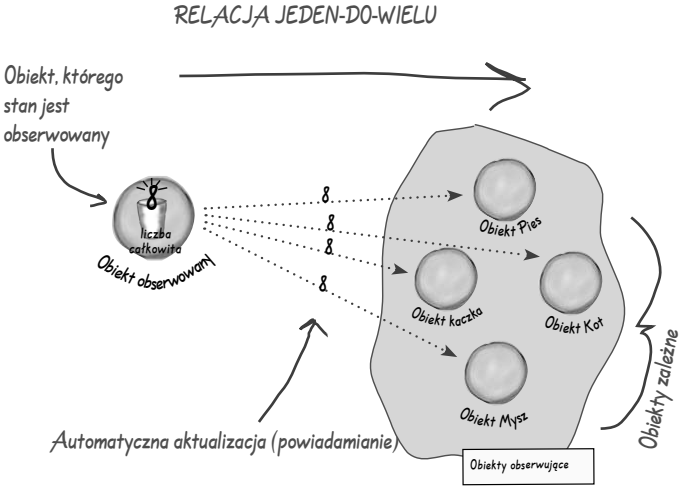
Jednak w świecie rzeczywistym wzorzec Obserwator jest zazwyczaj definiowany następująco:

> **Wzorzec Obserwator** definiuje pomiędzy obiektami relację jeden-do-wielu w taki sposób, że kiedy wybrany obiekt zmienia swój stan, to wszystkie jego obiekty zależne zostają o tym powiadomione i automatycznie zaktualizowane.

Wzorzec Obserwator definiuje relację jeden–do–wielu pomiędzy obiektami danego zbioru obiektów.

Kiedy stan jednego z obiektów zmienia się, wszystkie jego obiekty zależne zostają o tym powiadomione

Porównajmy zatem powyższą definicję z tym, o czym do tej pory mówiliśmy na temat wzorca:

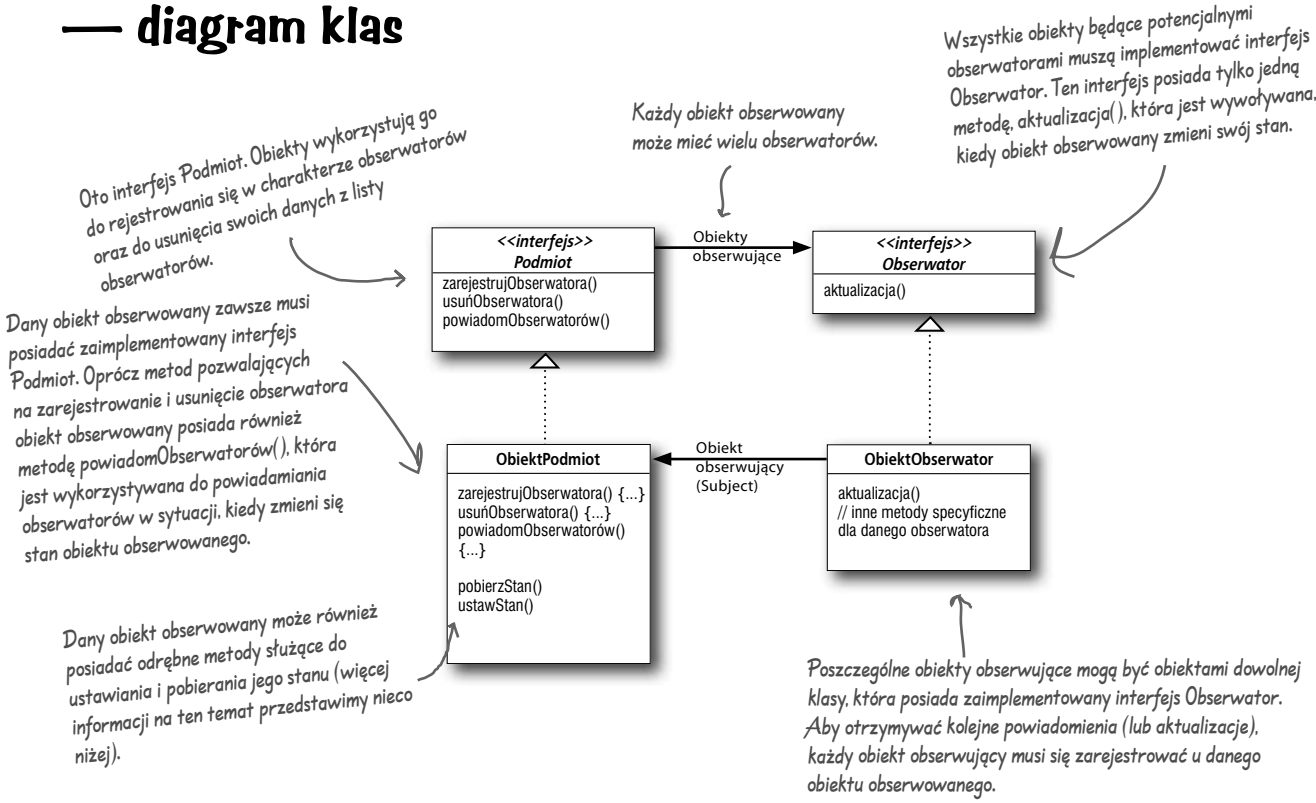


Obiekt obserwowany oraz obiekty obserwujące są ze sobą powiązane relacją jeden-do-wielu. Obiekty obserwujące są zależne od obiektu obserwowanego, co przejawia się w ten sposób, że jeżeli obiekt obserwowany zmienia swój stan, to wszystkie obiekty obserwujące zostają o tym powiadomione. W zależności od sposobu, w jaki realizowane jest powiadamianie, obiekty zależne mogą być również automatycznie aktualizowane (mogą otrzymywać nowe dane od obiektu obserwowanego).

Jak się niebawem sam przekonasz, wzorzec Obserwator może być implementowany na kilka różnych sposobów, aczkolwiek większość z nich obraca się dookoła tworzenia osobnych klas, które posiadają interfejsy Podmiot oraz Obserwator.

Rzućmy na to okiem...

Definicja wzorca Obserwator — diagram klas



Nie istnieją głupie pytania

P: Co to wszystko ma wspólnego z relacją jeden-do-wielu?

O: Zgodnie z założeniami wzorca Obserwaror to obiekt obserwowany jest obiektem, który posiada określony stan i kontroluje go. Zatem wynika stąd, że mamy tylko JEDEN obiekt, który posiada stan. Z kolei obiekty obserwujące wykorzystują ten stan, nawet jeżeli nie są jego właścicielami. Krótko mówiąc, istnieje wielu obserwatorów, którzy polegają na informacjach docierających z obiektu obserwowanego, przekazywanych w momencie, kiedy jego stan ulega zmianie. Możemy zatem powiedzieć, że jest to relacja pomiędzy JEDNYM obiektem obserwowanym, a WIELOMA obiektami obserwującymi.

P: Gdzie zatem w tym całym układzie znajduje się zależność?

O: Ponieważ to obiekt obserwowany jest jedynym i wyłącznym właścicielem danych, aktualizacja obiektów obserwujących jest zależna od zmiany stanu obiektu obserwowanego. Takie podejście pozwala na stosowanie bardziej przejrzystego procesu projektowania (zorientowanego obiektowo), niż miałoby to miejsce w sytuacji, kiedy wiele obiektów może sterować tym samym zestawem danych.

Siła luźnych zależności

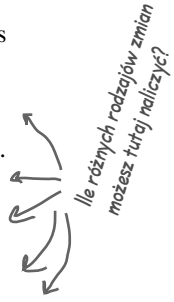
Kiedy dwa obiekty są ze sobą luźno powiązane, mogą ze sobą współpracować, ale z drugiej strony wzajemnie nie wiedzą o sobie zbyt wiele.

Wzorzec Obserwator zapewnia utworzenie takiej struktury obiektów, w której obiekty obserwowane są luźno powiązane ze swoimi obiektami obserwującymi.

Dlaczego?

Jedyną informacją, jaką obiekt obserwowany posiada o obiekcie obserwującym, jest to, że obiekt obserwujący posiada pewien określony interfejs (interfejs Obserwator). Obiekt obserwowany nie musi nawet wiedzieć, do jakiej klasy należy obiekt obserwujący i czym się on zajmuje. Nie musi też znać żadnej tego typu informacji.

Nowych obserwatorów możemy dodawać w dowolnym momencie. Ponieważ jedynym elementem, na którym opiera się działanie obiektu obserwowanego, jest lista zarejestrowanych obiektów implementujących interfejs Obserwator, nowe obiekty obserwujące mogą być dodawane w dowolnie wybranej chwili. W praktyce w czasie działania programu możemy wymienić dowolny, zarejestrowany obiekt obserwujący na inny — a obiekt obserwowany niezależnie od tego będzie sobie funkcjonował tak samo, jak do tej pory. Jak łatwo się domyślić, w analogiczny sposób możemy usunąć w dowolnej chwili dowolny obiekt obserwujący.



Dodawanie nowych typów obserwatorów nigdy nie pociąga za sobą konieczności modyfikacji obiektu obserwowanego. Powiedzmy, że pojawiła nam się (nie wnikamy, skąd) nowa klasa, której obiekty muszą być obserwatorami. Aby dodać je do listy subskrybentów, nie musimy modyfikować kodu obiektu obserwowanego — wszystko, czym musimy się zająć, to implementacja w nowym obiekcie interfejsu Obserwator i zarejestrowanie nowego obserwatora na liście obiektu obserwowanego. Ten ostatni zupełnie nie będzie się tym przejmował; jego zadaniem jest po prostu dostarczanie określonych informacji do wszystkich obiektów, które zostały zarejestrowane i implementują interfejs Obserwator.

Zarówno obiekty obserwowane, jak i obiekty obserwujące mogą być niezależnie od siebie wielokrotnie wykorzystywane. Jeżeli dla któregoś z obiektów obserwowanych lub obserwujących znajdziesz inne zastosowanie, możesz je łatwo wykorzystać, ponieważ są one ze sobą bardzo luźno powiązane.

Zarówno zmiany wprowadzane do obiektu obserwowanego, jak do obiektów obserwujących nie mają wzajemnie na siebie żadnego wpływu. Ponieważ wspomniane dwa rodzaje obiektów są ze sobą luźno powiązane, możemy je niemal dowolnie modyfikować tak długo, jak długo poszczególne obiekty będą spełniały założenie posiadania poprawnie zaimplementowanego interfejsu Podmiot bądź Obserwator.

Reguła projektowania

Staraj się tworzyć projekty, w których obiekty są ze sobą luźno powiązane i, o ile to możliwe, nie oddziałują na siebie wzajemnie.

Projekty charakteryzujące się luźnymi powiązaniami pomiędzy obiektami pozwalają na tworzenie bardzo elastycznych systemów, w łatwy sposób adaptujących się do wprowadzanych zmian, co zawdzięczają minimalizacji wzajemnych zależności wewnętrznych.

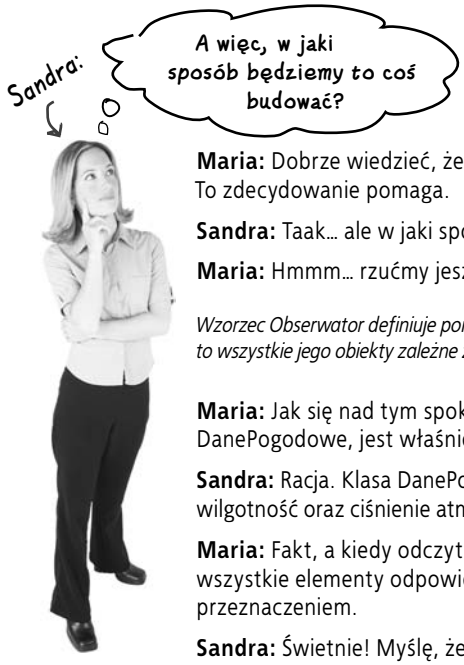
Zaostrz ołówek

Zanim przejdziesz do kolejnej sekcji, spróbuj naszkicować strukturę klas, jakiej będziesz potrzebował do zaimplementowania oprogramowania stacji meteorologicznej, włączając w to klasę DanePogodowe oraz jej elementy odpowiadające za wyświetlanie. Upewnij się, że rysowany diagram pokazuje, w jaki sposób poszczególne elementy są ze sobą połączone, a także wyjaśnia, w jaki sposób inny programista lub projektant może dołączyć swój własny, nowy tryb wyświetlania.

Jeżeli potrzebujesz małej pomocy, przeczytaj to, co zamieściliśmy na następnej stronie; wygląda na to, że Twoje koleżanki z boksu obok już rozpoczęły dysputę na temat tego projektu.

Zasłyszane w sąsiednim boksie...

Wracając do tematu projektu stacji meteorologicznej... wygląda na to, że Twoje koleżanki z boksu obok zaczęły już myśleć o rozwiązaniu tego zadania...



Maria: Dobrze wiedzieć, że będziemy się posługiwać wzorcem Obserwator. To zdecydowanie pomaga.

Sandra: Taak... ale w jaki sposób go zaimplementujemy?

Maria: Hmmm... rzućmy jeszcze raz okiem na definicję tego wzorca:

Wzorzec Obserwator definiuje pomiędzy obiektami relację jeden-do-wielu w taki sposób, że kiedy wybrany obiekt zmienia swój stan, to wszystkie jego obiekty zależne zostają o tym powiadomione i automatycznie zaktualizowane.

Maria: Jak się nad tym spokojnie zastanowić, wychodzi na to, że taka definicja ma sens. Nasza klasa, DanePogodowe, jest właśnie tym „jednym", a „wieloma" są poszczególne tryby wyświetlania informacji o pogodzie.

Sandra: Racja. Klasa DanePogodowe posiada również swoje stany... to znaczy, mam na myśli temperaturę otoczenia, wilgotność oraz ciśnienie atmosferyczne, a te wartości definitywnie będą się zmieniać na przestrzeni czasu.

Maria: Fakt, a kiedy odczyty tych parametrów będą się zmieniały, będziemy musieli poinformować o tym wszystkie elementy odpowiedzialne za wyświetlanie, tak aby mogły przetwarzać nowe wartości zgodnie ze swoim przeznaczeniem.

Sandra: Świetnie! Myślę, że już wiem, w jaki sposób możemy zastosować wzorzec Obserwator do naszego systemu śledzenia pogody według wskazań czujników stacji meteorologicznej.

Maria: Ale ciągle pozostaje kilka spraw, które musimy rozważyć, a co do których nie jestem całkowicie pewna, że je dobrze rozumiem.

Sandra: Na przykład?

Maria: Na przykład to, w jaki sposób dostarczymy wyniki pomiarów poszczególnych parametrów do elementów odpowiedzialnych za wyświetlanie danych.

Sandra: No cóż, przypomnij sobie, jak wygląda diagram wzorca Obserwator. Jeżeli naszym obiektem obserwowanym zostanie obiekt DanePogodowe, a obiektami obserwującymi będą elementy (obiekty) odpowiedzialne za wyświetlanie poszczególnych informacji, to aby otrzymywać aktualne informacje, obiekty takie będą musiały najpierw zostać zarejestrowane przez obiekt obserwowany, prawda?

Maria: Tak... a kiedy obiekt obserwowany będzie znał listę swoich subskrybentów, wystarczy, że będzie wywoływał odpowiednie metody, których zadaniem będzie przekazywanie im zaktualizowanych danych z pomiarów.

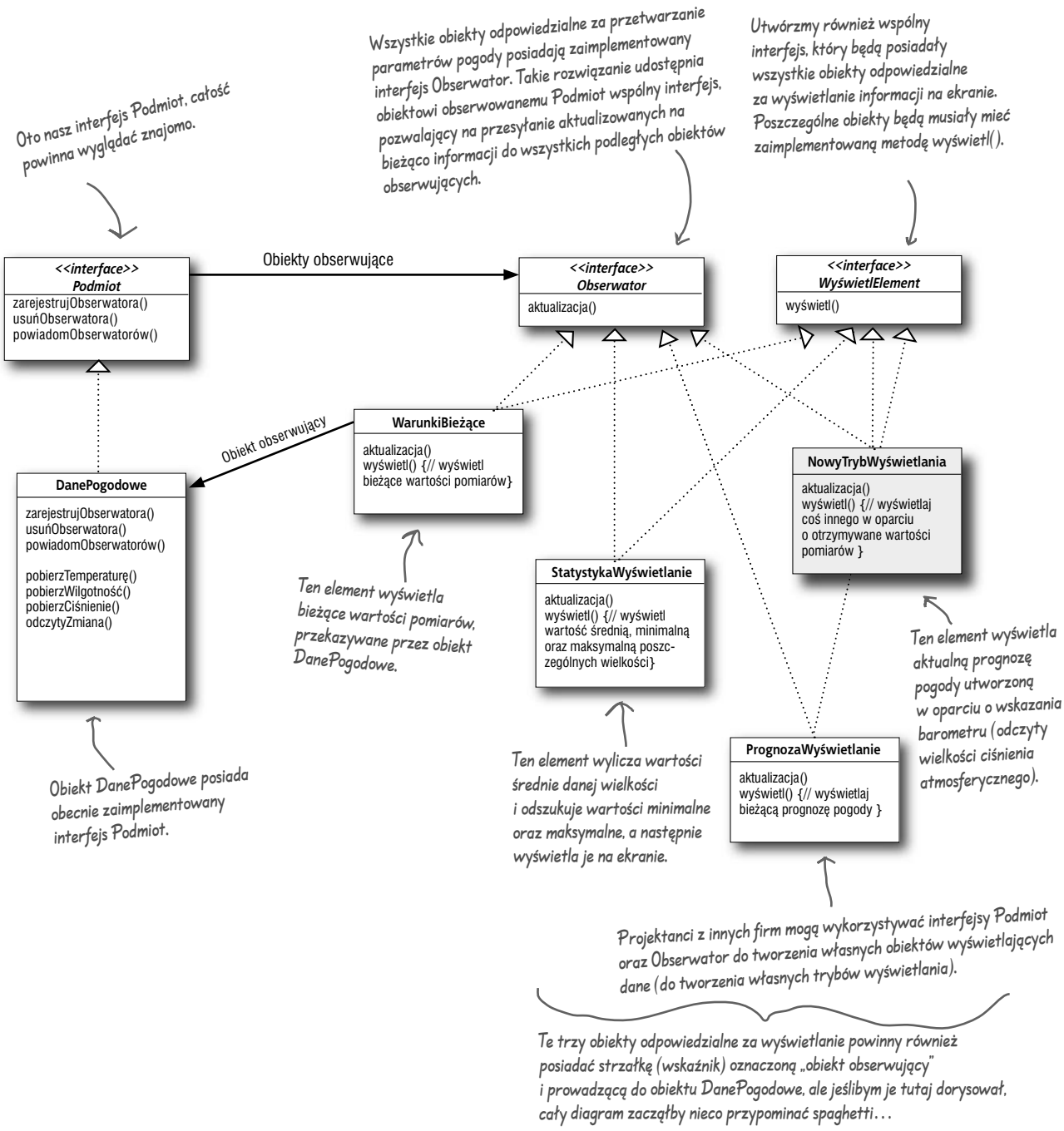
Sandra: Musimy jednak pamiętać, że każdy wyświetlany element może być zupełnie inny... Więc tak sobie myślę, gdzie jest miejsce na wprowadzenie wspólnego interfejsu? Pomimo, że poszczególne elementy mogą być zupełnie odmiennych typów, wszystkie powinny posiadać taki sam interfejs, tak aby obiekt DanePogodowe wiedział, w jaki sposób wysyłać im wyniki poszczególnych pomiarów.

Maria: Wiem, o co Ci chodzi. Wynika z tego, że każdy obiekt odpowiedzialny za wyświetlanie będzie posiadał metodę, powiedzmy, aktualizacja(), która będzie wywoływana przez obiekt DanePogodowe.

Sandra: ... i ta metoda aktualizacja() będzie zdefiniowana w wspólnym interfejsie, zaimplementowanym we wszystkich obiektach.

Projektowanie stacji meteorologicznej

Jak poniższy diagram wygląda w porównaniu z Twoim?



Implementacja stacji meteorologicznej

Rozpoczniemy teraz implementację aplikacji obsługującej stację meteorologiczną, wykorzystując do tego celu przedstawiony diagram klas oraz biorąc pod uwagę wnioski płynące z rozmowy Marii i Sandry (zapis rozmowy zamieściliśmy kilka stron wcześniej). W nieco dalszej części niniejszego rozdziału przekonasz się, że Java posiada pewne wbudowane mechanizmy obsługi wzorca Obserwator, aczkolwiek w chwili obecnej mamy szczery zamiar nieco pobrudzić sobie rączki, samemu zabrać się do roboty i stworzyć niezbędny kod. Co prawda, wbudowane mechanizmy Javy pozwalają na skorzystanie z nich w niektórych wypadkach, ale samodzielne stworzenie odpowiednich klas gwarantuje o wiele większą elastyczność całego systemu (a zaręczam, że nie jest to takie trudne). Dobrze, zatem zacznijmy od stworzenia odpowiednich interfejsów (pliki Podmiot.java, Obserwator.java oraz WyświetlElement.java):

```
public interface Podmiot {
    public void zarejestrujObserwatora (Obserwator o);
    public void usuńObserwatora(Obserwator o);
    public void powiadomObserwatorów();
}
```

Metody pobierają jako argument obiekt typu Obserwator, czyli konkretnie obiekt, który ma zostać zarejestrowany na liście zarejestrowanych bądź usunięty z tej listy.

Ta metoda jest wywoływana w celu powiadomienia wszystkich obserwatorów o tym, że stan obiektu obserwowanego zmienił się.

```
public interface Obserwator {
    public void aktualizacja(float temp, float wilgotność, float ciśnienie);
}
```

Te zmienne odpowiadają wartościom stanu, jakie obiekty obserwujące otrzymują od obiektu obserwowanego, kiedy zmieniają się odczyty parametrów pogody.

Wszystkie obiekty obserwujące posiadają zaimplementowany interfejs Obserwator, stąd wszystkie muszą mieć także zaimplementowaną metodę aktualizacja(). Właśnie w tym miejscu wykorzystamy sugestie, jakie padły podczas rozmowy Marii z Sandrą, i tutaj będziemy przekazywać wyniki kolejnych pomiarów do poszczególnych obserwatorów.

```
public interface WyświetlElement{
    public void wyświetl();
}
```

Interfejs WyświetlElement posiada tylko jedną metodę, wyświetl(), którą będziemy wywoływać w sytuacji, kiedy niezbędne będzie wyświetlanie danego elementu na ekranie.

WYTĘŻ UMYSŁ

Maria i Sandra myślały, że przekazywanie wyników pomiarów bezpośrednio do obiektów obserwujących jest najprostszym sposobem na ich uaktualnianie. Czy zgadzasz się z ich zdaniem? Wskazówka: czy w naszej aplikacji jest jakiś obszar, który może w przyszłości ulec zmianie? Jeżeli taka zmiana nastąpi, to czy będzie ona wystarczająco dobrze hermetyzowana, czy też będzie jednak wymagała dokonania modyfikacji wielu innych części kodu aplikacji?

Czy możesz pomyśleć nad opracowaniem nowych sposobów przekazywania uaktualnionych odczytów do obiektów obserwujących?

Nie musisz się przejmować — powrócimy jeszcze do problemu zatwierdzenia projektu aplikacji po zakończeniu wstępnej fazy jej implementacji.

Implementacja interfejsu Podmiot w klasie DanePogodowe

PAMIĘTAJ: w prezentowanych przykładach listingów nie umieszczamy poleceń „import" ani „package". Komplet kodów źródłowych poszczególnych przykładów znajdziesz na serwerze FTP wydawnictwa HELION: ftp://ftp.helion.pl/przyklady/wzorrg.zip

Czy pamiętasz naszą pierwszą próbę implementacji klasy DanePogodowe, którą przeprowadziliśmy na początku tego rozdziału? Być może będziesz chciał nieco odświeżyć sobie pamięć… Teraz właśnie nastał czas, aby do niej powrócić i trochę ją przeprojektować — mając na względzie wymagania, jakie narzuca nam wzorzec Obserwator…

```
public class DanePogodowe implements Podmiot {
    private ArrayList obserwatorzy;
    private float temperatura;
    private float wilgotność;
    private float ciśnienie;

    public DanePogodowe() {
        obserwatorzy = new ArrayList();
    }

    public void zarejestrujObserwatora(Obserwator o) {
        obserwatorzy.add(o);
    }

    public void usuńObserwatora(Obserwator o) {
        int i = obserwatorzy.indexOf(o);
        if (i >= 0) {
            obserwatorzy.remove(i);
        }
    }

    public void powiadomObserwatorów() {
        for (int i = 0; i < obserwatorzy.size(); i++) {
            Obserwator Obs = (Obserwator)obserwatorzy.get(i);
            Obs.aktualizacja(temperatura, wilgotność, ciśnienie);
        }
    }

    public void odczytyZmiana() {
        powiadomObserwatorów();
    }

    public void ustawOdczyty(float temperatura, float wilgotność, float ciśnienie) {
        this.temperatura = temperatura;
        this.wilgotność = wilgotność;
        this.ciśnienie = ciśnienie;
        odczytyZmiana();
    }

    // tutaj zamieść inne metody klasy DanePogodowe
}
```

Klasa DanePogodowe implementuje obecnie interfejs Podmiot.

Dodaliśmy zmienną typu ArrayList, której zadaniem jest przechowywanie listy obserwatorów, a tworzenie tej zmiennej umieściliśmy w definicji konstruktora obiektów klasy DanePogodowe.

Kiedy dany obserwator się rejestruje, dopisujemy go po prostu na końcu listy.

Analogicznie, jeżeli dany obserwator chce się wyrejestrować, po prostu usuwamy go z listy.

A tutaj mamy coś fajnego: to jest właśnie miejsce, w którym mówimy obiektom obserwującym (obserwatorom), że zmienił się stan obiektu obserwowanego. Ponieważ wszystkie obiekty obserwujące posiadają interfejs Obserwator, wiemy, że posiadają także zaimplementowaną metodę aktualizacja() i w związku z tym wiemy, jak możemy je powiadomić.

Tutaj dokonujemy implementacji interfejsu Subject.

Obserwatorów będziemy powiadamiali po otrzymaniu nowych wartości pomiarów ze stacji meteorologicznej.

No dobra… musimy się przyznać, że ze swojej strony chcieliśmy sprzedawać kopię naszej wspaniałej aplikacji do obsługi stacji meteorologicznej razem z każdym egzemplarzem niniejszej książki, ale, niestety, nasz wydawca nie chciał się zgodzić na takie rozwiązanie. Z tego zatem względu, zamiast odczytywać wyniki pomiarów z jakichś rzeczywistych urządzeń na stacji meteo, będziemy wykorzystywali tę metodę do testowania naszych obiektów odpowiedzialnych za poszczególne tryby wyświetlania. Jeżeli chcesz, dla rozrywki możesz napisać odpowiedni kod, który będzie pobierał odpowiednie informacje z jakiejś strony sieci WWW.

A teraz pora na zbudowanie elementów wyświetlających dane

Mamy już naszą klasę DanePogodowe zapiętą niemal na ostatni guzik, nadszedł więc czas na stworzenie elementów, które będą wyświetlały odpowiednie informacje w poszczególnych trybach pracy na ekranie. Zleceniodawca, firma Pogodynka sp. z o.o., poprosiła o przygotowanie trzech trybów wyświetlania: „Warunki bieżące", „Statystyka" oraz „Prognoza". Rzućmy okiem na element odpowiedzialny za wyświetlanie bieżących warunków pogodowych; kiedy dokładnie się z nim zapoznasz i zrozumiesz jego zasady działania, powinieneś sprawdzić kody pozostałych dwóch elementów, odpowiedzialnych za statystykę i prognozy (odpowiednie kody źródłowe znajdziesz na serwerze FTP wydawnictwa HELION: *ftp://ftp.helion.pl/przyklady/wzorrg.zip*). Szybko się zorientujesz, że ich struktura jest bardzo zbliżona.

Ta klasa, odpowiadająca za wyświetlanie warunków bieżących, posiada zaimplementowany interfejs Obserwator, dzięki czemu może otrzymywać informacje o zmieniających się danych od obiektu klasy DanePogodowe.

Ta klasa posiada również zaimplementowany interfejs WyświetlElement, ponieważ nasze API będzie wymagało, aby każdy element odpowiedzialny za wyświetlanie informacji na ekranie posiadał taki interfejs.

```
public class WarunkiBieżąceWyświetl implements Obserwator, WyświetlElement {
    private float temperatura;
    private float wilgotność;
    private Podmiot DanePogodowe;

    public WarunkiBieżąceWyświetl(Podmiot DanePogodowe) {
        this.DanePogodowe = DanePogodowe;
        DanePogodowe.zarejestrujObserwatora(this);
    }

    public void aktualizacja(float temperatura, float wilgotność, float ciśnienie) {
        this.temperatura = temperatura;
        this.wilgotność = wilgotność;
        wyświetl();
    }

    public void wyświetl() {
        System.out.println("Warunki bieżące " + temperatura
            + " stopni C oraz " + wilgotność + "% wilgotność");
    }
}
```

Konstruktor obiektów klasy WarunkiBieżąceWyświetl otrzymuje jako argument wywołania obiekt DanePogodowe (czyli obiekt obserwowany, Podmiot), a następnie wykorzystuje go do zarejestrowania swojego obiektu jako obserwatora.

Kiedy wywoływana jest metoda aktualizacja(), zachowujemy wartości temperatury otoczenia oraz wilgotności, a następnie wywołujemy metodę wyświetl().

Metoda wyświetl() powoduje po prostu wyświetlenie najbardziej aktualnych odczytów temperatury i wilgotności.

Nie istnieją głupie pytania

P: Czy metoda aktualizacja() to najlepsze miejsce do umieszczenia wywołania metody realizującej wyświetlanie?

O: W naszym prostym przykładzie wywoływanie metody wyświetl(), kiedy następowała aktualizacja danych, miało sens. Nie zmienia to jednak faktu, że masz rację, twierdząc, że istnieją o wiele lepsze sposoby wyświetlania przychodzących danych. Przekonasz się o tym przy okazji omawiania wzorca Model-Widok-Kontroler.

P: Dlaczego odwołanie do obiektu obserwowanego (Podmiot) jest zapamiętywane? Przecież nie wygląda na to, aby było ono używane gdziekolwiek poza konstruktorem tej klasy?

O: Prawda, aczkolwiek w przyszłości może zaistnieć potrzeba usunięcia tego obiektu z listy zarejestrowanych obserwatorów i wtedy posiadanie takiego gotowego, istniejącego odwołania może okazać się bardzo użyteczne.

Włączamy zasilanie naszej stacji meteorologicznej

❶ Po pierwsze, przygotujmy środowisko testowe.

Nasz aplikacja do obsługi stacji meteorologicznej jest w zasadzie gotowa do pracy. Jedyne, czego jeszcze będziemy potrzebować, to odrobina kodu, który połączy wszystkie stworzone do tej pory moduły w jedną, zgrabną i sprawnie działającą całość. Poniżej znajdziesz naszą pierwszą przymiarkę do tego zadania. W nieco dalszej części książki powrócimy jeszcze do tego zagadnienia i upewnimy się, że poszczególne elementy składowe całej aplikacji mogą być łatwo do niej dołączane za pośrednictwem odpowiedniego pliku konfiguracyjnego. W chwili obecnej nasz kod wygląda następująco:

```
public class StacjaMeteo {

    public static void main(String[] args) {
        DanePogodowe danePogodowe = new DanePogodowe();

        WarunkiBieżąceWyświetl warunkiBieżąceWyświetl =
            new WarunkiBieżąceWyświetl(danePogodowe);
        StatystykaWyświetl statystykaWyświetl = new StatystykaWyświetl(danePogodowe);
        PrognozaWyświetl prognozaWyświetl = new PrognozaWyświetl(danePogodowe);

        danePogodowe.ustawOdczyty(26.6, 65, 1013.1f);
        danePogodowe.ustawOdczyty(27.7, 70, 997.0f);
        danePogodowe.ustawOdczyty(25.5, 90, 997.0f);
    }
}
```

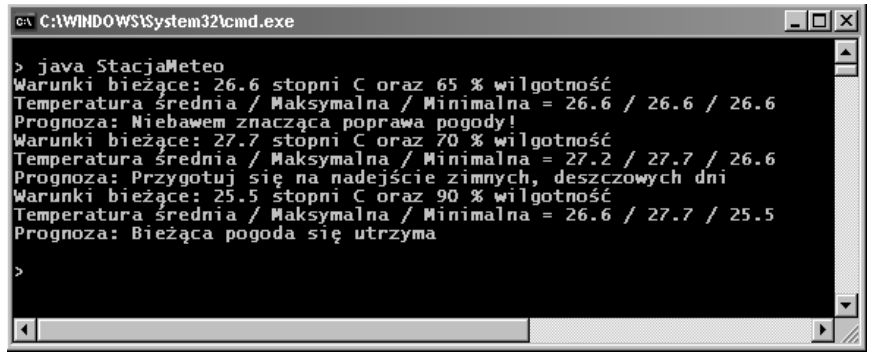
Najpierw utworzymy obiekt klasy DanePogodowe.

Jeżeli nie chcesz ściągać gotowych przykładów z serwera FTP wydawnictwa Helion, możesz po prostu wykomentować te dwa wiersze i uruchomić program bez nich.

Tworzymy trzy obiekty odpowiedzialne za poszczególne tryby wyświetlania i przekazujemy im stworzony obiekt klasy DanePogodowe.

Symulacja kolejnych wartości odczytów z czujników pogody.

❷ Uruchom program i pozwól, aby zaczęła działać magia wzorca Obserwator.



Zaostrz ołówek

Właśnie przed chwilą zadzwonił p. Janusz Monsun, prezes rady nadzorczej firmy Pogodynka sp. z o.o., i stwierdził, że sprzedaż całego systemu nie będzie możliwa, jeżeli nie zostanie zaimplementowany nowy tryb wyświetlania, prezentujący tzw. indeks ciepła. Poniżej załączamy definicję tego indeksu:

Indeks ciepła to wskaźnik, który łączy ze sobą wskazania temperatury i wilgotności powietrza i określa rzeczywistą temperaturę otoczenia (tzn. *odczuwaną temperaturę*). Aby obliczyć wartość indeksu ciepła, należy wziąć wartość temperatury `T*)` oraz wilgotności względnej `RH`, a następnie podstawić do poniższego wzoru:

indeksCiepła =

$$16.923 + 1.85212 * 10^{-1} * T + 5.37941 * RH - 1.00254 * 10^{-1} * T * RH + 9.41695 * 10^{-3} * T2 + 7.28898 * 10^{-3} * RH2 + 3.45372 * 10^{-4} * T2 * RH - 8.14971 * 10^{-4} * T * RH2 + 1.02102 * 10^{-5} * T2 * RH2 - 3.8646 * 10^{-5} * T3 + 2.91583 * 10^{-5} * RH3 + 1.42721 * 10^{-6} * T3 * RH + 1.97483 * 10^{-7} * T * RH3 - 2.18429 * 10^{-8} * T3 * RH2 + 8.43296 * 10^{-10} * T2 * RH3 - 4.81975 * 10^{-11} * T3 * RH3$$

Słowem, nie pozostało Ci nic innego, jak tylko rozpocząć „wklepywanie"!

Oczywiście, żartowaliśmy! Nie martw się, nie będziesz musiał pracowicie przepisywać całego wzoru; wystarczy, że stworzysz swój własny plik *IndeksCiepła.java* i skopiujesz do niego gotową formułę, którą znajdziesz w pliku *IndeksCiepła.txt*.

Plik IndeksCiepła.txt możesz odnaleźć na serwerze FTP wydawnictwa Helion: ftp://ftp.helion.pl/przyklady/wzorrg.zip

Jak działa ta magiczna formuła? Aby się tego dowiedzieć, powinieneś zajrzeć do książki *Meteorologia. Rusz głową!* (o ile kiedyś takowa się ukaże), zapytać kogoś pracującego w Instytucie Meteorologii i Gospodarki Wodnej lub po prostu zapytać Wielkiego Google'a (*http://www.google.pl*).

Po wprowadzeniu odpowiednich zmian i uruchomieniu programu wyniki jego działania powinny wyglądać następująco:

Oto zmiany, jakie powinieneś zauważyć w wynikach działania programu.

```
C:\WINDOWS\System32\cmd.exe
> java StacjaMeteo
Warunki bieżące: 26.6 stopni C oraz 65 % wilgotność
Temperatura średnia / Maksymalna / Minimalna = 26.6 / 26.6 / 26.6
Prognoza: Niebawem znacząca poprawa pogody!
Obliczony indek ciepła ma wartość 28.30852
Warunki bieżące: 27.7 stopni C oraz 70 % wilgotność
Temperatura średnia / Maksymalna / Minimalna = 27.2 / 27.7 / 26.6
Prognoza: Przygotuj się na nadejście zimnych, deszczowych dni
Obliczony indek ciepła ma wartość 30.50068
Warunki bieżące: 25.5 stopni C oraz 90 % wilgotność
Temperatura średnia / Maksymalna / Minimalna = 26.6 / 27.7 / 25.5
Prognoza: Bieżąca pogoda się utrzyma
Obliczony indek ciepła ma wartość 28.69426

>
```

Temat dzisiejszej wieczornej pogawędki: **Obserwowany (Podmiot) oraz Obserwujący (Obserwator) przekomarzają się na temat najlepszego sposobu przekazywania informacji o zmianach stanu do obiektów obserwujących.**

Obiekt obserwowany

Cieszę się, że w końcu mamy okazję porozmawiać osobiście.

Niby tak, ale przecież robię to, co do mnie należy. Zawsze mówię Wam, co w trawie piszczy… To, że nie wiem, kim tak naprawdę jesteście, nie oznacza wcale, że o Was nie dbam. A poza tym wiem o Was jedną, najważniejszą rzecz — macie zaimplementowany interfejs Obserwator.

Tak? Ciekawe, co?

Przepraszam BARDZO. Wysyłam powiadomienia wraz z informacjami o moim stanie tak, abyście Wy, leniwi obserwatorzy, wiedzieli, co tu się w końcu dzieje!

No dobrze…. myślę, że to mogłoby zadziałać. Musiałbym tylko bardziej otworzyć się na Wasze zakusy i pozwolić wszystkim Obserwatorom dostać się do środka tak, abyście mogli samodzielnie pobierać potrzebne Wam informacje o stanie. Niemniej jednak, takie rozwiązanie może być nieco niebezpieczne. Nie mogę przecież pozwolić na to, aby Obserwatorzy wchodzili na mój teren bez zaproszenia i przeglądali wszystko, co mam do zaoferowania.

Obiekt obserwujący

Naprawdę?? A ja myślałem, że Ty zupełnie nie dbasz o nas, Obserwatorów.

Tak, pewnie, ale to tylko malutka część tego, co posiadamy. Tak czy inaczej, z pewnością wiemy o Tobie znacznie więcej…

Na przykład to, że zawsze przekazujesz informacje o swoim stanie nam, Obserwatorom, abyśmy wiedzieli, co się wewnątrz Ciebie dzieje. A to od czasu do czasu może być nieco irytujące…

OK, poczekaj moment i nie gorączkuj się tak; po pierwsze, nie jesteśmy leniwi, mamy po prostu bardzo wiele innych zadań do wykonania pomiędzy tymi Twoimi jakże-bardzo-ważnymi-powiadomieniami, panie Obserwowany; po drugie, dlaczego nie pozwolisz nam po prostu wejść na Twój teren i wziąć potrzebnych informacji, zamiast wypychać je uparcie na siłę wszystkim Obserwatorom?

Obiekt obserwowany

Tak, mógłbym w zasadzie pozwolić Ci na **pobieranie** mojego stanu. Ale czy takie rozwiązanie nie będzie dla Ciebie mniej wygodne? Jeżeli musiałbyś przychodzić do mnie za każdym razem, kiedy będziesz czegoś potrzebował, to może się zdarzyć, że aby otrzymać niezbędne informacje, będziesz zmuszony do wykonywania wielokrotnych wywołań metod. Z tej prostej przyczyny wolę **wysyłać** Wam mój stan… dzięki temu otrzymujecie to wszystko, co niezbędne, w jednym, prostym powiadomieniu.

Hmmm, jak by na to nie patrzeć, faktycznie oba sposoby mają swoje zalety. Zauważyłem, że Java posiada wbudowaną obsługę wzorca Obserwator, który pozwala na przesyłanie informacji zarówno w postaci wysyłania, jak i pobierania danych.

Świetnie… być może będę miał zatem okazję zobaczyć dobry przykład takiego rozwiązania, który pozwoli mi zmienić zdanie…

Obiekt obserwujący

To dlaczego po prostu nie napiszesz jakiejś publicznie dostępnej metody pozwalającej na pobieranie danych i nie udostępnisz nam jej, abyśmy sami mogli pobierać niezbędne informacje o Twoim stanie?

Nie bądź taki natarczywy! Istnieje tak wiele różnych typów Obserwatorów, że z pewnością nie możesz zawsze przewidzieć wszystkich naszych potrzeb. Po prostu pozwól nam wejść na Twój teren i pobrać potrzebne informacje o Twoim stanie. W ten sposób, jeżeli któryś z nas będzie potrzebował tylko pewnej niewielkiej części informacji, nie będzie zmuszony do przyjmowania ich kompletu. Dzięki temu łatwiej później będzie dokonywać modyfikacji. Powiedzmy, na przykład, że trochę się rozwiniesz i Twój stan będzie posiadał kilka nowych informacji. Wtedy, hmmm... jeżeli korzystasz z metody pobierania danych, nie będziesz musiał biegać dookoła i modyfikować metody aktualizacja() każdego zarejestrowanego Obserwatora. Wystarczy, że zmienisz się sam, udostępniając nowe metody, dające dostęp do dodatkowych elementów Twojego stanu.

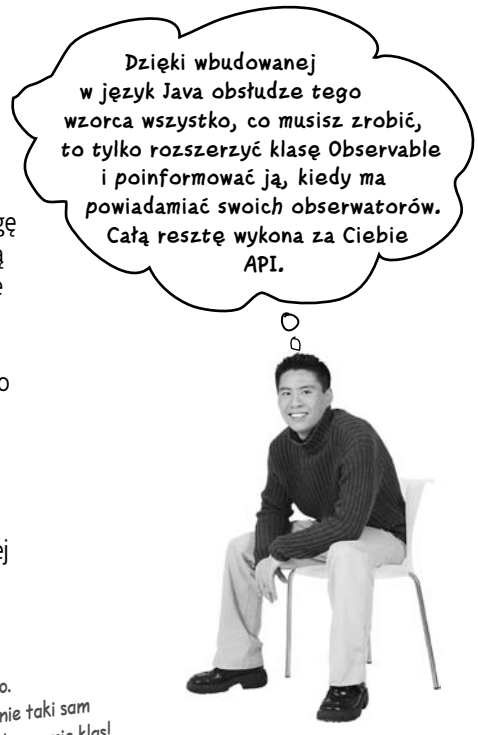
Och, naprawdę? Myślę zatem, że powinniśmy mu się przyjrzeć…

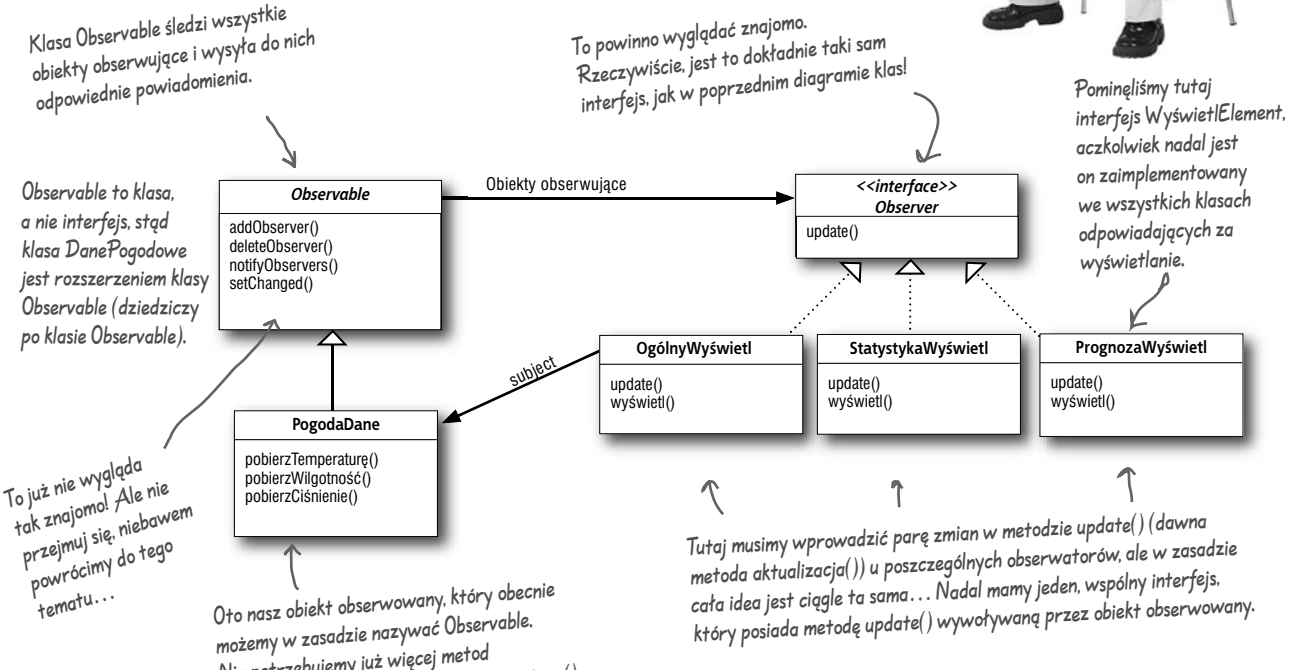
Niemożliwe, czyżbyśmy jednak doszliśmy do jakiegoś porozumienia? Czyli jednak zawsze jest jakaś nadzieja…?

Java — zastosowanie wbudowanego wzorca Obserwator

Jak pamiętasz, do tej pory tworzyliśmy swój kod, zgodny z wymogami wzorca Obserwator. Warto jednak wspomnieć, że język Java posiada także własną obsługę tego wzorca, wbudowaną w kilka różnych API. Najbardziej ogólną implementacją jest interfejs Observer i klasa Observable (oba elementy zlokalizowane w pakiecie java.util). Elementy te są dosyć podobne do naszych interfejsów Obserwator i Podmiot, ale już gotowe, i praktycznie natychmiast udostępniają Ci swoją dosyć rozbudowaną funkcjonalność. Dzięki temu pakietowi możesz skorzystać zarówno z metody wysyłania powiadomień przez obiekt obserwowany, jak i z metody pobierania stanu obiektu obserwowanego przez obiekty obserwujące, co zresztą postaramy się zademonstrować.

Aby lepiej zrozumieć interfejs java.util.Observer i klasę java.util.Observable, przeanalizuj przedstawiony poniżej nowy, zorientowany obiektowo projekt naszej stacji meteorologicznej:





Java — jak funkcjonuje wbudowany wzorzec Obserwator

Wbudowane mechanizmy obsługi wzorca Obserwator działają nieco inaczej niż implementacja, której używaliśmy w początkowej wersji projektu stacji meteorologicznej. Najbardziej oczywistą różnicę stanowi fakt, że klasa DanePogodowe (czyli nasz obiekt obserwowany) jest teraz rozszerzeniem klasy Observable i dziedziczy po niej metody addObserver(), deleteObserver() oraz notifyObservers() (a oprócz tego jeszcze kilka innych). Poniżej dowiesz się, w jaki sposób możemy skorzystać w języku Java z wbudowanego mechanizmu obsługi wzorca Obserwator.

Aby dowolny obiekt został obserwatorem...

Jak zwykle, musisz zaimplementować interfejs Obserwatora (teraz jest to interfejs java.util.Observer), a następnie wywołać metodę addObserver() dla dowolnego, wybranego obiektu klasy Observable. Analogicznie, aby usunąć dany obiekt z listy obserwatorów, musisz wywołać metodę deleteObserver().

Aby wybrany obiekt klasy Observable wysyłał powiadomienia o zmianie stanu...

Po pierwsze i najważniejsze, dany obiekt musi dać się obserwować, czyli musi dziedziczyć po klasie nadrzędnej java.util.Observable. Jeżeli ten warunek jest spełniony, trzeba wykonać jeszcze dwa kroki:

1. **Musisz wywołać metodę setChanged(), aby zasygnalizować, że zmienił się stan obiektu obserwowanego.**

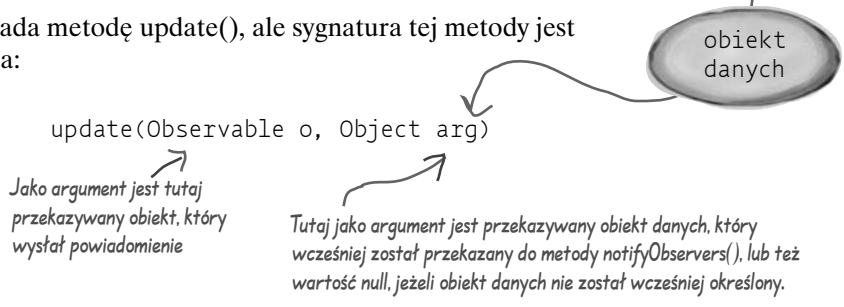
2. **Następnie musisz wywołać jedną z dwóch dostępnych metod notifyObservers():**

   `notifyObservers()` lub `notifyObservers(Object arg)`.

Ta wersja pozwala na pobranie dowolnego obiektu danych, który następnie zostanie przekazany podczas powiadamiania do wszystkich zarejestrowanych obiektów obserwujących.

Aby obserwator otrzymywał powiadomienia...

Obserwator posiada metodę update(), ale sygnatura tej metody jest jednak nieco inna:



Jeżeli chcesz wysyłać dane do obserwatorów, możesz przekazać odpowiednie dane jako obiekt danych do metody notifyObserver(arg). Jeżeli nie, obserwator musi „pobierać" potrzebne dane z obiektu (obserwowanego) klasy Observable, który zostaje do niego przekazany. Jak? Zmienimy teraz po raz kolejny projekt naszej stacji meteorologicznej i wtedy przekonasz się sam.

Za kulisami

Metoda setChanged() jest wykorzystywana do zasygnalizowania, że stan obiektu obserwowanego zmienił się i wywołanie metody notifyObservers() powinno przynieść aktualizację danych dla obserwatorów. Jeżeli metoda notifyObservers() zostanie wywołana bez uprzedniego wywołania metody setChanged(), obserwatorzy NIE ZOSTANĄ powiadomieni. Rzućmy teraz okiem za kulisy (czyli zajrzyjmy do kodu źródłowego klasy Observable) i zobaczmy, jak działa metoda setChanged().

Za kulisami

```
setChanged() {
    changed = true
}

notifyObservers(Object arg){
    if (changed) {
        dla każdego obserwatora na liście {
            call update(this, arg)
        }
        changed = false
    }
}

notifyObservers(){
    notifyObservers(null)
}
```

Metoda setChanged() ustawia flagę changed na wartość „prawda" (true).

Metoda notifyObservers() będzie wysyłała powiadomienia wyłącznie wtedy, jeśli wcześniej flaga changed zostanie ustawiona na wartość „prawda" (true).

Po rozesłaniu powiadomień metoda ponownie ustawia flagę changed na wartość „fałsz" (false).

Fragment pseudokodu klasy Observable.

Dlaczego takie rozwiązanie jest konieczne? Metoda setChanged() została pomyślana jako sposób na zapewnienie większej elastyczności w sposobie powiadamiania obserwatorów, sposób pozwalający na wprowadzenie optymalizacji powiadomień. Przykładowo, załóżmy, że na naszej stacji meteorologicznej znajdują się tak bardzo czułe przyrządy, że odczyty temperatury ciągle oscylują wokół pewnej temperatury bazowej z dokładnością do kilku setnych części stopnia. Taka sytuacja może spowodować, że obiekt klasy DanePogodowe będzie nieustannie wysyłał nowe powiadomienia. Zamiast do tego dopuścić, lepiej będzie wysyłać powiadomienia wyłącznie wtedy, kiedy różnica temperatury pomiędzy dwoma kolejnymi pomiarami wyniesie, dajmy na to, co najmniej pół stopnia (inaczej mówiąc, tylko w takiej sytuacji wywołana zostanie metoda setChanged()).

Być może nie będziesz zbyt często korzystał z opisanego przed chwilą mechanizmu, niemniej jednak, jeżeli już zaistnieje taka konieczność, warto pamiętać, że rozwiązanie masz gotowe. Pamiętaj również, że jeżeli chcesz, aby powiadomienia w ogóle działały, musisz najpierw wywołać metodę setChanged(). Jeżeli zdecydujesz się na korzystanie z opisanego mechanizmu, warto również wspomnieć o metodach takich, jak clearChanged() (ustawia flagę changed na wartość „fałsz" (false)) czy hasChange() (zwraca aktualny stan flagi changed).

Kolejna wersja kodu Stacji Meteo — zastosowanie wbudowanych mechanizmów Javy do obsługi wzorca Obserwator

Po pierwsze, dopasujemy kod klasy DanePogodowe tak, aby można było skorzystać z klasy java.util.Observable

1. Upewnij się, że importujemy właściwą klasę i właściwy interfejs.

2. Teraz dla klasy Observable tworzymy nową klasę podrzędną.

3. Nie musimy już dłużej śledzić naszych obserwatorów ani zarządzać procesami ich rejestracji oraz usuwania z listy (tym zajmuje się już klasa nadrzędna), więc usunęliśmy kod przeznaczony uprzednio do rejestracji, usuwania i wysyłania powiadomień do obserwatorów.

```
import java.util.Observable;
import java.util.Observer;

public class DanePogodowe extends Observable {
    private float temperatura;
    private float wilgotność;
    private float ciśnienie;

    public DanePogodowe() { }

    public void odczytyZmiania() {
        setChanged();
        notifyObservers();*
    }

    public void ustawOdczyty(float temperatura, float wilgotność, float ciśnienie) {
        this.temperatura = temperatura;
        this.wilgotność = wilgotność;
        this.ciśnienie = ciśnienie;
        odczytyZmiana();
    }

    public float pobierzTemperaturę() {
        return temperatura;
    }

    public float pobierzWilgotność() {
        return wilgotność;
    }

    public float pobierzCiśnienie() {
        return ciśnienie;
    }
}
```

4. Nasz konstruktor już nie musi tworzyć osobnej struktury danych przeznaczonej do przechowywania danych obserwatorów.

* Zwróć uwagę, że nie wysyłamy już obiektu danych do metody notifyObservers(). Oznacza to, że właśnie korzystamy z metody POBIERANIA stanu obiektu obserwowanego.

5. Zanim wywołamy metodę notifyObservers(), musimy wywołać metodę setChanged(), która ustawi flagę wskazującą, że stan obiektu obserwowanego zmienił się.

6. Te metody nie są nowe, ale ponieważ chcemy skorzystać z metody „pobierania", pomyśleliśmy sobie, że przypomnimy Ci o ich istnieniu. Obserwatorzy będą ich używać do pobierania informacji o stanie obiektu klasy DanePogodowe.

A teraz przyszedł czas na modyfikację klasy WarunkiBieżąceWyświetl

1 Ponownie upewnij się, że importujemy właściwą klasę i właściwy interfejs.

2 Obecnie importujemy interfejs Observer z pakietu java.util

```
import java.util.Observable;
import java.util.Observer;

public class WarunkiBieżąceWyświetl implements Observer, WyświetlElement {
    Observable observable;
    private float temperatura;
    private float wilgotność;

    public WarunkiBieżąceWyświetl(Observable observable) {
        this.observable = observable;
        observable.addObserver(this);
    }

    public void update(Observable obs, Object arg) {
        if (obs instanceof DanePogodowe) {
            DanePogodowe danePogodowe = (DanePogodowe)obs;
            this.temperatura = danePogodowe.pobierzTemperaturę();
            this.wilgotność = danePogodowe. pobierzWilgotność();
            wyświetl();
        }
    }

        public void wyświetl() {
        System.out.println("Warunki bieżące " + temperatura
            + "stopni C oraz " + wilgotność + v% wilgotność");
    }
}
```

3 Konstruktor naszej klasy pobiera obecnie obiekt klasy Observable, a my wykorzystujemy to do wpisania obiektu opisującego warunki bieżące na listę obserwatorów.

4 Zmodyfikowaliśmy metodę update() tak, aby mogła pobierać jako argumenty zarówno obiekt klasy Observable, jak i opcjonalny obiekt danych.

5 W przypadku metody update() najpierw musimy przekonać się, że obiekt obserwowany jest typu DanePogodowe, a następnie możemy wykorzystać jego metody służące do pobierania danych do uzyskania wartości pomiarów temperatury otoczenia i wilgotności.
Na zakończenie wywołujemy metodę wyświetl().

Magnesiki z kodem

Jak widać, definicja klasy PrognozaWyświetl znajduje się w kompletnej rozsypce. Czy możesz poukładać poszczególne fragmenty kodu w odpowiedniej kolejności tak, aby całość znów mogła poprawnie działać? A przy okazji, niektóre z nawiasów klamrowych pospadały nam przy okazji na ziemię, a są zdecydowanie za małe, aby podejmować wysiłek ich podnoszenia, więc możesz swobodnie dopisać tyle nawiasów klamrowych, ile będzie potrzeba!

```
public PrognozaWyświetl(Observable
observable) {
```

```
wyświetl();
```

```
observable.addObserver(this);
```

```
if (observable instanceof DanePogodowe) {
```

```
public class PrognozaWyświetl implements
Observer, WyświetlElement {
```

```
public void wyświetl() {
// kod metody wyświetl()
```

```
ostatnieCiśnienie = bieżąceCiśnienie;
bieżąceCiśnienie = danePogodowe.pobierzCiśnienie();
```

```
private float bieżąceCiśnienie = 1010.1f;
private float ostatnieCiśnienie;
```

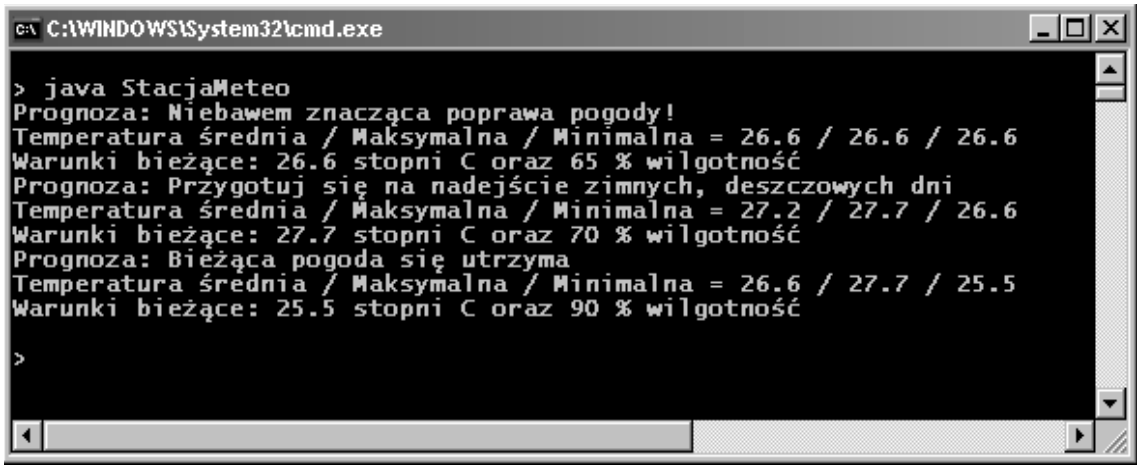
```
DanePogodowe danePogodowe =
(DanePogodowe) observable;
```

```
public void update(Observable observable,
                   Object arg) {
```

```
import java.util.Observable;
import java.util.Observer;
```

Uruchamianie nowego kodu

Aby się upewnić, że wszystko działa, jak należy, spróbujmy uruchomić nową wersję naszej aplikacji...



Hmmm, czy zauważyłeś tutaj jakąś różnicę? Nie? Spójrz zatem jeszcze raz...

Zobaczysz, że o ile wyniki obliczeń niczym się od siebie nie różnią, o tyle (i nieco tajemniczo) kolejność wyświetlania poszczególnych wierszy jest zupełnie inna. Dlaczego tak się stało? Pomyśl przez chwilę, zanim przeczytasz odpowiedź poniżej…

Nigdy nie polegaj na oszacowaniu kolejności wykonywania poszczególnych powiadomień

Klasa java.util.Observable posiada zaimplementowaną metodę notifyObservers(), w której poszczególni obserwatorzy są powiadamiani w *zupełnie innej* kolejności, niż to miało miejsce w naszej własnej implementacji. Kto zatem ma rację? Otóż nikt; po prostu, my wybraliśmy inny sposób implementacji.

Problem mógłby się jednak pojawić w sytuacji, kiedy kod programu *zależałby* w istotny sposób od określonej kolejności powiadomień. Dlaczego? Ano dlatego, że jeżeli musiałbyś zmienić implementację klasy Observable i (lub) interfejsu Observer, kolejność powiadomień mogłaby się również zmienić, a Twoja aplikacja najzwyczajniej w świecie zaczęłaby generować nieprawidłowe rezultaty. Oczywiście, taka implementacja jest jak najbardziej *daleka* od tego, co określamy mianem „luźnych powiązań”.

Ciemna strona klasy java.util.Observable

Tak, to bardzo dobre pytanie. Jak zauważyłeś, Observable jest *klasą*, a nie *interfejsem*. Co gorsza, klasa ta nawet nie posiada *zaimplementowanego* interfejsu. Niestety, tak się jednak złożyło, że implementacja klasy java.util.Observable jest obciążona szeregiem problemów, które znacznie ograniczają jej użyteczność i możliwości zastosowania. Nie mówimy tutaj, że jest ona całkowicie bezużyteczna, stwierdzamy jedynie, że posiada kilka poważnych dziur i wybojów, na które po prostu trzeba uważać.

Observable jest klasą

Dzięki przedstawionym wcześniej regułom projektowania już wiesz, że to nie jest dobre rozwiązanie, ale tak właściwie, jakie zagrożenia to ze sobą niesie?

Po pierwsze, ponieważ Observable jest *klasą*, to aby z niej skorzystać, musisz stworzyć jej *klasy podrzędne*. Oznacza to w praktyce, że nie możesz dodać zachowań klasy Observable do już istniejącej klasy, która dziedziczy po innej klasie nadrzędnej. Taka cecha zdecydowanie ogranicza możliwości wielokrotnego wykorzystywania tej klasy (i czy przypadkiem nie dlatego zdecydowaliśmy się na korzystanie z wzorców projektowych?).

Po drugie, ponieważ nie istnieje interfejs Observable, nie możesz nawet zbudować swojej własnej implementacji, która będzie dobrze współpracowała z wbudowanym API interfejsu Observer języka Java. Nie masz również możliwości wymiany pakietu java.util na inny (powiedzmy, na taki, który potrafi obsługiwać wielowątkowość).

Klasa Observable chroni swoje kluczowe metody

Jeżeli przyjrzysz się API wzorca Observer, przekonasz się, że metoda setChanged() jest chroniona. I co z tego? No cóż, oznacza to mniej więcej tyle, że nie możesz wywołać metody setChanged() dopóty, dopóki nie dokonasz dziedziczenia z superklasy Observable. Oznacza to również, że nie możesz nawet stworzyć obiektu klasy Observable i wyposażyć go w swoje własne obiekty, po prostu *jesteś skazany* na dziedziczenie. Struktura klasy Observable stanowi zatem pogwałcenie również drugiej reguły projektowania… *Przedkładaj kompozycję nad dziedziczenie.*

Co zatem robić?

Klasa Observable może, mimo wszystko, odpowiadać na Twoje potrzeby, jeżeli możesz dziedziczyć z klasy java.util.Observable. Z drugiej jednak strony, zawsze możesz zdecydować się na stworzenie swojej własnej implementacji, tak jak pokazaliśmy na początku niniejszego rozdziału. Niezależnie od tego, na jakie rozwiązanie się zdecydujesz, znasz już dobrze wzorzec Obserwator i dzięki temu masz dobre podstawy do pracy z dowolnym API, które go wykorzystuje.

Inne miejsca w JDK, w których możesz znaleźć wzorzec Obserwator

Jeżeli jesteś ciekawy, jak wygląda implementacja wzorca Obserwator w JavaBeans, zainteresuj się bliżej interfejsem PropertyChangeListener.

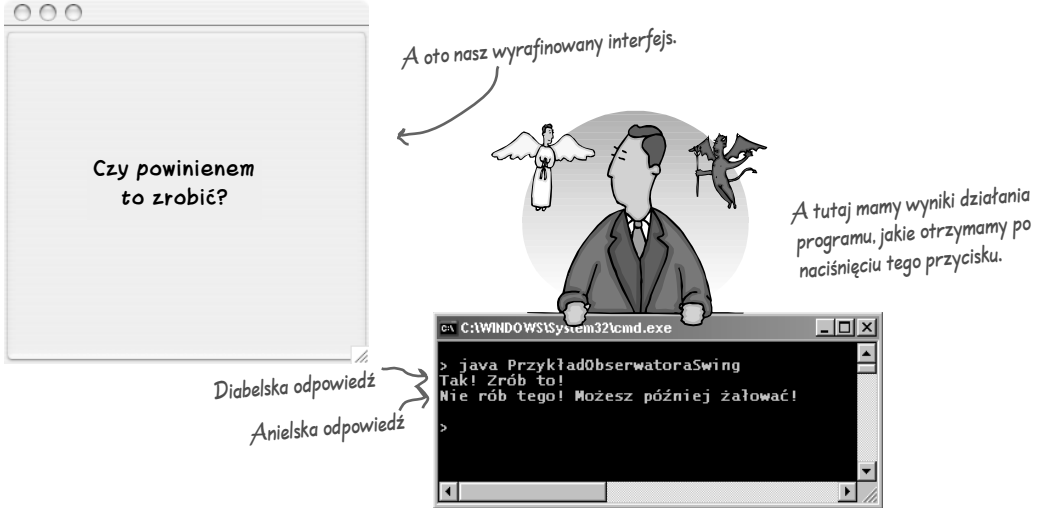
java.util.Observer oraz java.util.Observable nie są jedynymi miejscami w JDK (ang. Java Development Kit), w których możesz odnaleźć implementację wzorca Observer; zarówno komponenty JavaBean, jak i Swing mogą się pochwalić swoimi własnymi implementacjami tego wzorca. Na tym etapie wiedzy, na jakim obecnie się znajdujesz, możesz spokojnie pokusić się o samodzielną analizę wymienionych API; ale teraz, mimo wszystko, szybko zaprezentujemy przykład zastosowania biblioteki Swing, ot, tak po prostu, dla przyjemności.

Tytułem wprowadzenia...

Rzućmy okiem na jeden z prostych elementów Swing API, JButton. Jeżeli zajrzysz „pod maskę" klasy nadrzędnej w stosunku do JButton, czyli superklasy AbstractButton, zobaczysz, że posiada ona wiele metod pozwalających na dodawanie i usuwanie metod obiektów nasłuchujących (inaczej słuchaczy). Wspomniane metody pozwalają na dodawanie i usuwanie obserwatorów — lub, jak je nazywają w pakiecie Swing, obiektów nasłuchujących — z listy obiektów, które „nasłuchują" różnego rodzaju zdarzeń mogących nastąpić w obiekcie będącym częścią biblioteki Swing (obiekt nasłuchiwany). Przykładowo, ActionListener pozwala na „nasłuchiwanie" dowolnego typu zdarzeń, jakie mogą wystąpić w związku z przyciskiem (np. naciśnięcie przycisku). W Swing API znajdziesz wiele różnych rodzajów obiektów nasłuchujących.

Mała aplikacja, która odmieni nasze życie

No dobrze, nasza aplikacja jest całkiem prosta. Posiada ona jeden przycisk, na którym figuruje pytanie „Czy powinienem to zrobić?"; kiedy klikniesz ten przycisk, słuchacze (obserwatorzy) muszą odpowiedzieć na to pytanie w taki sposób, w jaki będą chcieli. Zaimplementujemy dwa takie obiekty nasłuchujące, nazywane AniołSłuchacz oraz DiabełSłuchacz. Poniżej przedstawiono zachowanie naszej aplikacji:



A oto i sam kod...

Nasza zmieniająca-życie-na-lepsze aplikacja nie wymaga zbyt dużej ilości kodu. Wszystko, co musimy zrobić, to stworzyć obiekt JButton, dołożyć go do obiektu JFrame, a następnie skonfigurować nasze obiekty nasłuchujące. W przypadku obiektów nasłuchujących użyjemy klas wewnętrznych (ang. *inner classes*), co w bibliotece Swing jest dosyć rozpowszechnioną techniką programowania. Jeżeli nie masz doświadczenia w programowaniu klas wewnętrznych lub nie znasz biblioteki Swing, powinieneś zajrzeć do rozdziału „Historia bardzo graficzna" książki *Java. Rusz głową! Wydanie II*.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;

public class PrzykładObserwatoraSwing {
    JFrame frame;

    public static void main(String[] args) {
        PrzykładObserwatoraSwing przykład  = new PrzykładObserwatoraSwing ();
        przykład.uruchom();
    }

    public void uruchom() {
        frame = new JFrame();
        JButton button = new JButton("Czy powinienem to zrobić?");
        button.addActionListener(new AniołSłuchacz());
        button.addActionListener(new DiabełSłuchacz());
        frame.getContentPane().add(BorderLayout.CENTER, button);

        // Parametry okienka
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(BorderLayout.CENTER, button);
        frame.setSize(300,300);
        frame.setVisible(true);
    }

    class AniołSłuchacz implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            System.out.println("Nie rób tego! Możesz później żałować!");
        }
    }

    class DiabełSłuchacz implements ActionListener {
        public void actionPerformed(ActionEvent event) {
            System.out.println("Tak! Zrób to!");
        }
    }
}
```

Prosta aplikacja wykorzystująca bibliotekę Swing. Tworzy ona na ekranie ramkę i umieszcza wewnątrz niej przycisk.

Powoduje, że zarówno anielski, jak i diabelski obiekt staje się obiektem nasłuchującym (obserwatorem) zdarzeń związanych z przyciskiem.

Tutaj umieszczone zostały klasy obserwatorów, zdefiniowane jako klasy wewnętrzne (ale wcale tak być nie musi).

Kiedy zmienia się stan obiektu obserwowanego (w naszym przypadku jest to oczywiście przycisk), zamiast metody update() wywoływana jest metoda actionPerformed().

Twoja skrzynka narzędziowa

Witamy na końcu rozdziału 2. Możesz teraz z czystym sumieniem dodać do swojej skrzynki narzędziowej kilka nowych elementów...

Oto Twoja najnowsza reguła projektowania. Pamiętaj, że struktury wykorzystujące luźne związki są o wiele bardziej elastyczne i łatwiej adaptują się do zmian.

Nowy wzorzec projektowy, który pozwala na powiadamianie zbioru obiektów o zmianach stanu, wykorzystujący luźne związki pomiędzy poszczególnymi obiektami. Nie omawialiśmy jeszcze ostatniego typu wzorca Obserwator — poczekajmy z tym jednak do czasu, aż przedstawimy wzorzec MVC!

CELNE SPOSTRZEŻENIA

- Wzorzec Obserwator definiuje pomiędzy obiektami relację jeden-do-wielu.
- Obiekty obserwowane (obiekty Podmiot) przesyłają powiadomienia i aktualizacje do obiektów obserwujących przy użyciu jednego, wspólnego interfejsu.
- Obiekty obserwujące są luźno powiązane z obiektem obserwowanym, co oznacza, że obiekt obserwowany wie o nich tylko tyle, że posiadają zaimplementowany interfejs Obserwator.
- Zgodnie z definicją wzorca informacje o zmianach stanu obiektu obserwowanego mogą być wysyłane lub pobierane (metoda „pobierania" jest uważana za bardziej politycznie poprawną).
- Nie można uzależniać poprawnego funkcjonowania aplikacji od określonej kolejności powiadamiania przesyłanego do obiektów obserwujących.
- Java posiada kilka różnych implementacji wzorca obserwator, włączając w to implementację klasy ogólnego przeznaczenia w pakiecie java.util.Observable.
- Należy zwracać baczną uwagę na ograniczenia związane z implementacją klasy java.util.Observable.
- Nie należy się obawiać tworzenia własnych implementacji klasy Observable.
- Biblioteka Swing bardzo mocno wykorzystuje implementację wzorca Obserwator; jest to cecha charakterystyczna niemal wszystkich pakietów GUI.
- Wzorzec Obserwator jest wykorzystywany w wielu innych miejscach, jak np. JavaBeans czy RMI.

Ćwiczenia

Reguły projektowania — wyzwanie

Napisz, w jaki sposób wzorzec Obserwator wykorzystuje poszczególne reguły projektowania.

Reguła projektowania

Dokonaj identyfikacji wszystkich zmiennych elementów Twojej aplikacji, a następnie oddziel je od elementów, które przez cały czas pozostają stałe.

Reguła projektowania

Skoncentruj się na tworzeniu interfejsów, a nie implementacji

Reguła projektowania

Przedkładaj kompozycję nad dziedziczenie.

Ten przypadek może sprawić nieco problemów, zatem wskazówka: zastanów się, w jaki sposób obiekty obserwujące współpracują z obiektem obserwowanym.

Rozwiązania

Zaostrz ołówek

Opierając się na pierwszej implementacji naszego systemu, określ, które z wymienionych zdarzeń są prawdziwe. (Zaznacz wszystkie poprawne odpowiedzi).

- ✓ A. Tworzymy poszczególne implementacje, a nie interfejsy.
- ✓ B. Dodanie nowego trybu wyświetlania będzie każdorazowo wymuszało modyfikację kodu programu.
- ✓ C. Nie mamy żadnych możliwości dodawania (lub usuwania) wybranych trybów wyświetlania podczas działania programu.
- D. Poszczególne wyświetlane elementy nie posiadają wspólnego interfejsu.
- ✓ E. Nie dokonaliśmy hermetyzacji tych elementów aplikacji, które się zmieniają.
- F. Naruszyliśmy hermetyzację klasy PogodaDane.

Reguły projektowania — wyzwanie

Reguła projektowania

Dokonaj identyfikacji wszystkich zmiennych elementów Twojej aplikacji, a następnie oddziel je od elementów, które przez cały czas pozostają stałe.

Elementami, które we wzorcu Obserwator podlegają zmianom, są: stan obiektu obserwowanego oraz ilość i rodzaj obiektów obserwujących. Zgodnie z założeniami wzorca możesz modyfikować obiekty, które są zależne od stanu obiektu obserwowanego bez konieczności modyfikacji samego obiektu obserwowanego. To jest to, co nazywamy planowaniem z wyprzedzeniem!

Reguła projektowania

Skoncentruj się na tworzeniu interfejsów, a nie implementacji

Zarówno obiekt obserwowany Podmiot, jak i obiekty obserwujące wykorzystują interfejsy. Obiekt obserwowany śledzi wszystkie obiekty, które mają zaimplementowany interfejs Obserwator, podczas gdy obiekty obserwujące do rejestracji, usuwania z listy zarejestrowanych obiektów oraz do otrzymywania powiadomień wykorzystują interfejs Podmiot. Jak już zauważyliśmy, takie rozwiązanie pozwala na utrzymanie „zgrabnej", luźno związanej struktury obiektów.

Reguła projektowania

Przedkładaj kompozycję nad dziedziczenie.

Wzorzec Obserwator wykorzystuje zasadę kompozycji do tworzenia dowolnej liczby obiektów obserwujących powiązanych z danym obiektem obserwowanym. Takie relacje nie są w żaden sposób tworzone poprzez mechanizm dziedziczenia — są za to tworzone całkowicie poprzez kompozycję i to w trakcie działania programu!

Rozwiązania

Magnesiki z kodem

```
import java.util.Observable;
import java.util.Observer;

public class PrognozaWyświetl implements
Observer, WyświetlElement {

    private float bieżąceCiśnienie = 1010.1f;
    private float ostatnieCiśnienie;

    public PrognozaWyświetl(Observable
    observable) {
        DanePogodowe danePogodowe =
        (DanePogodowe) observable;
        observable.addObserver(this);
    }
    public void update(Observable observable,
            Object arg) {
        if (observable instanceof PogodaDane) {
            ostatnieCiśnienie = bieżąceCiśnienie;
            bieżąceCiśnienie = pogodaDane.pobierzCiśnienie();
            wyświetl();
        }
    }
    public void wyświetl() {
    // kod metody wyświetl()
    }
}
```