

IDŹ DO

PRZYKŁADOWY ROZDZIAŁ



SPIS TREŚCI

KATALOG KSIĄŻEK

KATALOG ONLINE

ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK

DODAJ DO KOSZYKA

CENNIK I INFORMACJE

ZAMÓW INFORMACJE
O NOWOŚCIACH

ZAMÓW CENNIK

CZYTELNIA

FRAGMENTY KSIĄŻEK ONLINE

C#. Wzorce projektowe

Autor: Steven John Metsker

Tłumaczenie: Zbigniew Banach, Krzysztof Trawiński

ISBN: 83-7361-936-4

Tytuł oryginału: [Design Patterns in C#](#)

Format: B5, stron: 416



Wzorce projektowe są modelami gotowych rozwiązań problemów programistycznych, przed jakimi często stają twórcy oprogramowania. Dzięki nim nie musimy ponownie „wymyślać koła”. Dysponując wzorcami projektowymi, jesteśmy w stanie szybciej i efektywniej zbudować aplikację, ponieważ koncentrujemy się na samej implementacji algorytmu, a nie na jego opracowaniu. Najczęściej stosowane, klasyczne już, 23 wzorce projektowe opracowane przez twórców notacji UML implementowano już w wielu językach programowania. Jak sprawdzają się w połączeniu z jedną z najnowszych technologii, jaką jest .NET?

„C#. Wzorce projektowe” to kompletny przewodnik po wzorcach projektowych w C# i środowisku .NET. Przedstawia sposoby wykorzystania obiektowych cech języka C# i tworzenia eleganckiego kodu poprzez zastosowanie wzorców projektowych oraz korzystanie z ogromnych możliwości oferowanych przez biblioteki klas FCL dla środowiska Microsoft .NET. Wszystkie przedstawione w książce wzorce projektowe zostały zilustrowane przykładami kodu oraz diagramami UML, co ułatwia ich zrozumienie oraz zastosowanie w praktyce.

- Podstawowe wiadomości o wzorcach projektowych
- Interfejsy i klasy abstrakcyjne
- Wzorce interfejsów
- Wzorce odpowiedzialności
- Wzorce konstrukcyjne
- Wprowadzenie do operacji
- Wzorce operacji
- Wzorce rozszerzeń
- Wzorce rozszerzające

**Poznaj zastosowanie wzorców projektowych
i wykorzystaj je w swojej pracy**



Spis treści

Wprowadzenie	13
--------------------	----

1.

Wstęp	15
Dlaczego wzorce?	15
Dlaczego wzorce projektowe?	16
Dlaczego C#?	16
UML	17
Zadania	17
Układ książki	18
Witamy w firmie Oozinoz!	19
Podsumowanie	19

I

WZORCE INTERFEJSÓW	21
--------------------------	----

2.

Wprowadzenie do interfejsów	23
Interfejsy a klasy abstrakcyjne	23
Interfejsy a delegacje	24
Interfejsy a właściwości	28
Szczegóły interfejsów	28
Podsumowanie	29
Poza zwykłe interfejsy	29

3.

Adapter	31
Adaptacja kodu do potrzeb interfejsu	31
Adaptory klas i obiektów	34

Adaptacja danych w .NET	37
Podsumowanie	41

4.

Facade (Fasada)	43
Zwykła fasada	43
Refaktoryzacja do postaci fasady	45
Fasady, klasy narzędziowe i demonstracje	54
Podsumowanie	55

5.

Composite (Kompozyt)	57
Zwykły kompozyt	57
Rekurencja w kompozytach	58
Kompozyty, drzewa i cykle	59
Kompozyty z cyklami	64
Konsekwencje cykli	67
Podsumowanie	67

6.

Bridge (Pomost)	69
Zwykła abstrakcja	69
Od abstrakcji do pomostu	71
Sterowniki jako pomosty	73
Sterowniki baz danych	73
Podsumowanie	74

II

WZORCE ODPOWIEDZIALNOŚCI	75
--------------------------------	----

7.

Pojęcie odpowiedzialności	77
Zwykła odpowiedzialność	77
Kontrola odpowiedzialności poprzez kontrolę dostępu	79
Podsumowanie	81
Poza zwykłą odpowiedzialność	82

8.

Singleton	83
Zasada działania wzorca Singleton	83
Singletony a wątki	84
Rozpoznawanie singletonów	86
Podsumowanie	86

9.

Observer (Obserwator)	87
Obsługa wzorca Observer w C#	87
Sposób działania delegacji	88
GUI — klasyczny przykład obserwacji	91
Model-View-Controller	97
Warstwy	99
Podsumowanie	103

10.

Mediator	105
Klasyczny przykład: mediacja w GUI	105
Mediatorzy integralności relacji	109
Podsumowanie	115

11.

Proxy (Pośrednik)	117
Prosty pośrednik	117
Pośrednik danych	121
Zdalny pośrednik	126
Podsumowanie	130

12.

Chain of Responsibility (Łańcuch odpowiedzialności)	131
Zwykły łańcuch odpowiedzialności	131
Wprowadzanie łańcucha odpowiedzialności	132
Zaczepianie łańcucha	135
Łańcuch odpowiedzialności bez kompozytów	137
Podsumowanie	137

13.

Flyweight (Waga piórkowa)	139
Niezmiennosc	139
Wydzielenie niezmiennego fragmentu obiektu	140
Współużytkowanie obiektów	141
Podsumowanie	145

III

WZORCE KONSTRUKCYJNE	147
----------------------------	-----

14.

Pojęcie konstrukcji	149
Kilka zadań konstrukcyjnych	149
Podsumowanie	152
Poza zwykłą konstrukcję	152

15.

Builder (Budowniczy)	153
Zwykły budowniczy	153
Budowanie z ograniczeniami	155
Pobłażliwy budowniczy	157
Podsumowanie	158

16.

Factory Method (Metoda fabrykująca)	159
Klasyczny przykład: enumeratory	159
Rozpoznawanie wzorca Factory Method	161
Kontrola wyboru klasy	161
Metody fabryczne w hierarchiach równoległych	163
Podsumowanie	164

17.

Abstract Factory (Fabryka abstrakcji)	167
Klasyczny przykład: zestawy GUI	167
Fabryki abstrakcji a metody fabrykujące	172
Przestrzenie nazw a fabryki abstrakcji	175
Podsumowanie	175

18.

Prototype (Prototyp)	177
Prototypy jako fabryki	177
Klonowanie prototypów	178
Podsumowanie	181

19.

Memento	183
Klasyczny przykład: cofnięcie ostatniej czynności	183
Trwałe memento	190

Memento wielosesyjne	191
Podsumowanie	193

IV

WZORCE OPERACJI	195
-----------------------	-----

20.

Operacje — wprowadzenie	197
Operacje a metody	197
Sygnatury	198
Delegacje	199
Wyjątki	200
Algorytmy a polimorfizm	201
Podsumowanie	202
Poza operacje niestandardowe	203

21.

Template Method (Metoda szablonu)	205
Klasyczny przykład: sortowanie	205
Uzupełnianie algorytmu	208
Punkty zaczepienia w Template Method	210
Refaktoryzacja do wzorca Template Method	211
Podsumowanie	213

22.

State (Stan)	215
Modelowanie stanów	215
Refaktoryzacja do wzorca State	218
Stałe obiekty stanu	222
Podsumowanie	224

23.

Strategy (Strategia)	225
Modelowanie strategii	225
Refaktoryzacja do wzorca Strategy	227
Porównanie wzorców Strategy i State	232
Porównanie wzorców Strategy i Template Method	232
Podsumowanie	232

24.

Command (Polecenie)	235
Klasyczny przykład: polecenia menu	235
Zastosowanie wzorca Command w usługach	236

Punkty zaczepienia w Command	239
Command a inne wzorce	241
Podsumowanie	242

25.

Interpreter (Interpreter)	243
Przykład interpretera	243
Interpretery, języki i parsery	254
Podsumowanie	255

V

WZORCE ROZSZERZEŃ	257
-------------------------	-----

26.

Wprowadzenie do rozszerzeń	259
Zasady projektowania obiektowego	259
Zasada podstawienia Barbary Liskov	259
Prawo Demeter	261
Usuwanie z kodu „przykrych zapachów”	262
Poza rozszerzenia niestandardowe	263
Podsumowanie	264

27.

Decorator (Dekorator)	265
Klasyczny przykład: strumienie	265
Opakowania funkcji	273
Wykorzystanie dekoratora w graficznym interfejsie użytkownika	279
Decorator a inne wzorce	279
Podsumowanie	279

28.

Iterator (Iterator)	281
Zwykła iteracja	281
Iteracja z bezpiecznymi wątkami	281
Iteracja kompozytu	286
Podsumowanie	296

29.

Visitor (Wizytator)	297
Mechanizm działania wzorca	297
Standardowy wizytator	299
Cykle wizytatora	304

Kontrowersje związane z wizytatorem	308
Podsumowanie	309

DODATKI	311
---------------	-----

A

Dalsze kierunki	313
Wykorzystanie wiedzy zawartej w książce	313
Zrozumienie klasyki	313
Wplatanie wzorców we własny kod	314
Ciągła nauka	315

B

Rozwiązania	317
-------------------	-----

C

Źródła Oozinoz	383
Pobranie i wykorzystanie źródeł	383
Wykonywanie kodu Oozinoz	383
Pomoc w lokalizacji plików	384
Testowanie kodu za pomocą NUnit	385
Samodzielne szukanie plików	385
Podsumowanie	386

D

Rzut oka na UML	387
Klasy	387
Relacje pomiędzy klasami	389
Interfejsy	390
Delegacje i zdarzenia	391
Obiekty	392
Stany	393

E

Słowniczek	395
Bibliografia	403
Skorowidz	405

2

Wprowadzenie do interfejsów

Ujmując rzecz najbardziej abstrakcyjnie, interfejs klasy to zbiór metod i pól, jakie ta klasa udostępnia obiektom innych klas. Tak udostępniony interfejs najczęściej wiąże się ze zobowiązaniem, że dostępne metody wykonywać będą operacje sugerowane przez ich nazwy i określone przez komentarze w kodzie czy inną dokumentację klasy. *Implementację* klasy stanowi kod składający się na jej metody.

W języku C# pojęcie interfejsu zostało wyniesione do rangi samodzielnej konstrukcji. Interfejs obiektu (czyli zestaw wykonywanych operacji) jest jawnie oddzielony od implementacji (czyli konkretnego sposobu wykonania deklarowanej operacji). Interfejsy w C# pozwalają kilku klasom oferować te same operacje i dopuszczają możliwość implementowania przez klasę więcej niż jednego interfejsu.

Kilka wzorców projektowych korzysta z wbudowanych możliwości C# — można na przykład zastosować wzorec *Adapter*, używając interfejsu w celu zaadaptowania interfejsu klasy do potrzeb klienta. Zanim jednak przejdziemy do omówienia bardziej zaawansowanych możliwości, wypada pokrótce przyjrzeć się zasadom działania podstawowych mechanizmów języka C# — w przypadku tej części będą nas interesować przede wszystkim interfejsy.

Interfejsy a klasy abstrakcyjne

W oryginalnym wydaniu książki *Design Patterns* [Gamma et al. 1990] często pojawiają się wzmianki o klasach abstrakcyjnych, ale nie ma ani słowa o interfejsach. Wynika to z faktu, że wykorzystane do ilustracji wzorców języki C++ i Smalltalk po prostu nie mają takiej konstrukcji jak interfejs. Nie ma to jednak większego wpływu na użyteczność książki dla programistów C#, gdyż interfejsy C# są bardzo podobne do klas abstrakcyjnych.

Zadanie 2.1

Wypisz trzy różnice między klasami abstrakcyjnymi a interfejsami w C#.

Rozwiązanie na stronie 317.

Gdyby trzeba było obyć się bez interfejsów, można by radzić sobie, korzystając z klas abstrakcyjnych (co jest czynione w języku C++). Interfejsy odgrywają jednak kluczową rolę w tworzeniu aplikacji wielopoziomowych i z pewnością zasługują na status odrębnej konstrukcji językowej. Podobnie możliwe jest radzenie sobie bez delegacji i zastępowanie ich interfejsami (co ma miejsce w języku Java), ale delegacje pozwalają ujednoczyć pospolite zadanie, jakim jest zarejestrowanie metody do wywołania zwrotnego, i tym samym stanowią użyteczny dodatek do języka C#.

Interfejsy a delegacje

Delegacje są podobne do interfejsów, gdyż określają oczekiwania względem danego obiektu. Zasady działania delegacji bywają z początku nieco mylące, gdyż już samo pojęcie delegacji może się odnosić do różnych rzeczy. Przed przystąpieniem do porównania interfejsów z delegacjami przyjrzyjmy się samym delegacjom w języku C#.

Nowy typ delegowany jest w języku C# oznaczany słowem kluczowym `delegate`. Wskazuje on rodzaje metod, które mogą posłużyć do utworzenia jego instancji, ale *nie* określa nazwy żadnej konkretnej metody, lecz jedynie typy argumentów i typ wartości zwracanej przez metodę. Weźmy na przykład taką deklarację:

```
public delegate object BorrowReader(IDataReader reader);
```

Tworzy ona nowy typ delegowany o nazwie `BorrowReader` i określa, że jego instancję można utworzyć za pomocą dowolnej metody przyjmującej parametr typu `IDataReader` i zwracającej w wyniku obiekt typu `object`. Załóżmy, że mamy pewną klasę zawierającą następującą metodę:

```
private static object GetNames(IDataReader reader)
{
    // ...
}
```

Metoda `GetNames()` posiada wymagany przez delegację `BorrowReader` typ parametru i typ zwracany, więc można ją wykorzystać do utworzenia instancji tej delegacji:

```
BorrowReader b = new BorrowReader(GetNames);
```

Zmienna `b` przechowuje teraz instancję typu delegowanego `BorrowReader`. Instancję tę może wywołać dowolny kod mający do niej dostęp — spowoduje to wywołanie przez delegację zawartej w niej metody. Najważniejszą cechą utworzonego w ten sposób obiektu `b` jest to, że inna metoda może w dowolnym momencie skorzystać z obiektu i metody w nim zawartej. Można sobie na przykład wyobrazić klasę usług danych, która pobierze obiekt odczytujący informacje z bazy danych, wykorzysta go do utworzenia instancji delegacji `BorrowReader`, a następnie zwolni zasób tego obiektu. Takie właśnie usługi udostępnia przestrzeń nazw `DataLayer` z zestawu `Oozinoz`.

W kodzie `Oozinoz` delegacja `BorrowReader` wchodzi w skład przestrzeni nazw `DataLayer`:

```
namespace DataLayer
{
    // ...
}
```

```

    public delegate object BorrowReader(IDataReader reader);
    //...
}

```

Przeznaczeniem tej delegacji jest rozdzielenie odpowiedzialności między klasy wykorzystujące obiekty typu `IDataReader` (interfejs `IDataReader` należy do przestrzeni nazw `System.Data` środowiska .NET).

W przestrzeni nazw `DataLayer` znajduje się klasa `DataServices`, udostępniająca metodę `LendReader()` wypożyczającą obiekt odczytujący obiektowi żądającemu. Kod tej metody wygląda następująco:

```

public static object LendReader(
    string sql, BorrowReader borrower)
{
    using (OleDbConnection conn = CreateConnection())
    {
        conn.Open();
        OleDbCommand c = new OleDbCommand(sql, conn);
        OleDbDataReader r = c.ExecuteReader();
        return borrower(r);
    }
}

```

W powyższym kodzie wykorzystane jest połączenie z bazą danych, zwracane przez metodę `CreateConnection()`. Kod tej metody wygląda tak:

```

public static OleDbConnection CreateConnection()
{
    String dbName =
        FileFinder.GetFileName("db", "oozinoz.mdb");
    OleDbConnection c = new OleDbConnection();
    c.ConnectionString =
        "Provider=Microsoft.Jet.OLEDB.4.0;" +
        "Data Source=" dbName;
    return c;
}

```

Metoda `CreateConnection()` korzysta z pomocniczej klasy `FileFinder` z przestrzeni nazw `Oozinoz Utilities`, ułatwiającej dostęp do bazy danych MS Access znajdującej się w sąsiednim katalogu `db`. Działanie metody sprowadza się do utworzenia połączenia z bazą, skonfigurowania go za pomocą nazwy pliku bazy i zwrócenia obiektu połączenia. Korzystając z tak pobranego połączenia, metoda `LendReader()` tworzy, wypożycza i zamyka obiekt odczytujący informacje z bazy danych.

Metoda `LendReader()` przyjmuje dwa parametry: polecenie `select` oraz instancję delegacji `BorrowReader`. Działanie metody polega na utworzeniu połączenia z bazą, utworzeniu na jego podstawie obiektu odczytującego oraz utworzeniu instancji delegacji o nazwie `borrower`. Delegacja ta zawiera pewną metodę — może to być dowolna metoda pobierająca argument typu `IDataReader` i zwracająca obiekt. W chwili utworzenia przez metodę `LendReader()` delegacji `borrower` delegacja ta wywołuje zawartą w niej metodę i przekazuje obiekt odczytujący, z którego wywołana metoda może dowolnie korzystać. Po zakończeniu działania wywołanej metody sterowanie powraca do metody `LendReader()`.

Warto zwrócić uwagę, że wywołanie delegacji jest w metodzie `LendReader()` umieszczone w ramach klauzuli `using`, która po zakończeniu pracy (lub w razie zgłoszenia wyjątku) automatycznie wywoła dla obiektu połączenia metodę „sprzątającą” `Dispose()`. Z kolei połączenie „posprząta” po kodzie obiektu odczytującego tworzonoego w ramach polecenia `using`. Metoda `LendReader()` została zaprojektowana w taki sposób, że kod tworzący i zwalnający obiekt odczytujący dane z bazy może się znajdować zarówno przed kodem korzystającym z tego obiektu, jak i po nim.

Oto kod przykładowej klasy `ShowBorrowing`, ilustrującej sposób pożyczania obiektu odczytującego informacje z bazy danych:

```
using System;
using System.Data;
using DataLayer;
public class ShowBorrowing
{
    public static void Main()
    {
        string sel = "SELECT * FROM ROCKET";
        DataServices.LendReader(
            sel, new BorrowReader(GetNames));
    }
    private static object GetNames(IDataReader reader)
    {
        while (reader.Read())
        {
            Console.WriteLine(reader["Name"]);
        }
        return null;
    }
}
```

Uruchomienie tej klasy spowoduje wypisanie na ekranie nazw wszystkich rakiet w bazie danych Oozinoz:

```
Shooter
Orbit
Biggie
...
```

Dodatek C zawiera informacje na temat sposobu pobrania i uruchomienia przykładów Oozinoz.

Klasa `ShowBorrowing` zawiera metodę `GetNames()` o takich samych parametrach i typie wartości zwracanej, jak delegacja `BorrowReader`. Pozwala to metodzie `Main()` utworzyć instancję tej delegacji i przekazać ją jako jeden z parametrów metody `LendReader()`. Metoda `GetNames()` zwraca wartość `null`, choć tego rodzaju metoda pożyczająca częściej zwraca wyniki wykorzystania pożyczonego obiektu odczytującego.

Skorzystanie z delegacji pozwala wywołać metodę `GetNames()` w odpowiednim momencie, czyli po utworzeniu obiektu odczytującego, ale przed jego usunięciem. Ten sam efekt można osiągnąć za pomocą interfejsu definiującego wywołanie zwrotne metody. Załóżmy, że mamy nową metodę o nazwie `LendReader2()`, która przyjmuje interfejs, a nie delegację:

```
public static object LendReader2(  
    string sel, IBorrower borrower)  
{  
    using (OleDbConnection conn =  
        DataServices.CreateConnection())  
    {  
        conn.Open();  
        OleDbCommand c = new OleDbCommand(sel, conn);  
        OleDbDataReader r = c.ExecuteReader();  
        return borrower.BorrowReader(r);  
    }  
}
```

Zadanie 2.2

Implementacja interfejsu `IBorrower` musi obsługiwać metodę wywoływaną przez `LendReader2()` między utworzeniem a usunięciem obiektu odczytującego. Napisz kod implementacji `IBorrower.cs`.

Rozwiązanie na stronie 317.

Jeśli celem użycia delegacji jest jedynie zwrotne wywołanie pewnej metody, to równoważne rozwiązanie bazujące na interfejsach może być równie efektywne. Zalety delegacji widać najlepiej w sytuacjach, w których konieczne jest przechowywanie i wywoływanie kilku różnych metod. Typowym zastosowaniem jest wykorzystanie przez obiekt delegacji w celu zarejestrowania zainteresowania wielu klientów pewnym zdarzeniem (na przykład kliknięciem przycisku myszy). Cytując z podrozdziału 17.5 książki *C# Language Specification* [Wiltamuth]: „Zdarzenie jest składową umożliwiającą obiektowi lub klasie udostępnianie powiadomień”.

Zadanie 2.3

W języku `C#` interfejs może zawierać metody, właściwości i indeksery. Może też zawierać zdarzenia, ale delegacji już nie. Dlaczego?

Rozwiązanie na stronie 317.

Zrozumienie delegacji w `C#` nie jest proste, zwłaszcza że znaczenie tego pojęcia często jest niejako przeciążane — terminu *delegacja* można równie dobrze użyć w znaczeniu deklaracji delegacji, wynikowego typu delegowanego lub instancji tego typu. Często też mówimy o wywoływaniu delegacji, chociaż w rzeczywistości obiekty mogą wywoływać jedynie instancje delegacji, a nie same typy delegowane. Jeśli gubisz się nieco w tych niuansach terminologicznych, możesz się pocieszyć, że nie jesteś odosobniony w swym zakłopotaniu. Zrozumienie sposobu działania delegacji w `C#` jest jednak niezwykle przydatne, gdyż stanowią one kluczowy element funkcjonowania typowych aplikacji oraz współpracy klas.

Interfejsy a właściwości

Interfejsy C# mogą wymagać od typów implementujących dostarczenia indeksów lub właściwości. Potrzeba dostarczenia indeksa może wystąpić przy definiowaniu nowego typu kolekcji, natomiast znacznie częściej występuje konieczność określenia właściwości. Dotyczy to zwłaszcza sytuacji, w których typ implementujący musi posiadać określone metody do pobierania i (lub) ustawiania wartości atrybutu.

Poniższy interfejs nie korzysta z właściwości C#, ale mimo to wymaga od typów go implementujących zapewnienia dostępu do atrybutów obiektu — w tym przypadku jest to obiekt reprezentujący reklamę:

```
public interface IAdvertisement
{
    int GetID();
    string GetAdCopy();
    void SetAdCopy(string text);
}
```

Interfejs ten wchodzi w skład biblioteki klas Oozinoz o nazwie ShowProperties. Ogólnie można powiedzieć, że interfejs wymaga od typów implementujących dostarczenia dwóch właściwości, jednak ściśle rzecz biorąc, nie są to prawdziwe właściwości C#. Korzystanie z wbudowanych właściwości C# ma dwie podstawowe zalety: bardziej elegancką składnię i (co ważniejsze) dostępność właściwości interfejsu również dla nowych, nieznanych klientom za pośrednictwem refleksji. Na przykład obiekt klasy DataGrid będzie mógł wyświetlać właściwości obiektów kolekcji ArrayList pod warunkiem, że są one zdefiniowane jako prawdziwe właściwości C#.

Zadanie 2.4

Przepisz kod interfejsu IAdvertisement tak, by atrybuty ID i AdCopy były dla typów implementujących właściwościami C#.

Rozwiązanie na stronie 319.

Obecność właściwości w interfejsach jest pewnym niuansiem składniowym, ale ma wpływ na działanie typów implementujących. Jak to często bywa z językami programowania, znajomość szczegółów działania interfejsów C# jest równie istotna co zrozumienie ogólnych zasad ich stosowania.

Szczegóły interfejsów

Kluczową zaletą interfejsów w C# jest to, że ograniczają one zakres kontaktów między obiektami. W rzeczywistości ograniczenie to jest jednak wyzwoleniem — zawartość klasy implementującej interfejs może zostać całkowicie zmieniona, a kod klientów tej klasy pozostanie bez zmian. Sama idea interfejsów jest prosta, ale w praktyce często pojawiają się problemy powodujące nerwowe wertowanie książek.

Jeśli nie lubisz quizów, możesz od razu przejść do podsumowania. Jeśli jednak chcesz sprawdzić swoją znajomość szczegółów korzystania z interfejsów w C#, zapraszam do zmierzenia się z kolejnym zadaniem.

Zadanie 2.5

Poniższe zdania pochodzą ze specyfikacji języka C# [Wiltamuth]. W dwóch z nich zostały wprowadzone niewielkie zmiany, przez co stały się one błędne. Które stwierdzenia są fałszywe?

- (1) Interfejs definiuje kontrakt.
- (2) Interfejsy mogą zawierać metody, właściwości, delegacje i indeksery.
- (3) Deklaracja interfejsu musi deklarować jedną lub kilka składowych.
- (4) Wszystkie składowe interfejsu mają domyślnie dostęp publiczny.
- (5) Dołączenie specyfikatora do deklaracji składowej interfejsu powoduje błąd kompilacji.
- (6) Podobnie jak klasy nieabstrakcyjne, klasa abstrakcyjna musi dostarczyć implementację dla wszystkich składowych interfejsów wymienionych w liście klas bazowych tej klasy.

Rozwiązanie na stronie 319.

Podsumowanie

Potęga interfejsów polega na tym, że pozwalają one oddzielić zachowanie wymagane od opcjonalnego w kontaktach między klasami. Interfejsy są podobne do klas czysto abstrakcyjnych, gdyż definiują wymagania, nie implementując ich. Są również podobne do delegacji, choć te ostatnie określają jedynie typy parametrów i typ wartości zwracanej dla pojedynczej metody.

Delegacje są typami, interfejs nie może więc zawierać delegacji składowych. W interfejsach mogą się natomiast pojawiać zmienne typu delegowanego, deklarowane ze słowem kluczowym `event`.

Poza metodami i zdarzeniami interfejsy mogą też zawierać indeksery i właściwości. Składnia takich składowych jest bardzo ciekawa, a korzystanie z nich znacznie ułatwia klientowi określenie zachowania klas implementujących interfejs za pośrednictwem refleksji. Biegłe opanowanie zarówno ogólnych zasad, jak i szczegółów stosowania interfejsów C# jest ze wszech miar opłacalne, gdyż interfejsy stanowią podstawę wielu doskonałych projektów i kilku wzorców projektowych.

Poza zwykłe interfejsy

Odpowiednie stosowanie interfejsów C# może znacznie uprościć i usprawnić architekturę aplikacji, jednak niekiedy interfejs musi przybrać postać wykraczającą poza definicję i możliwości interfejsów wbudowanych.

Celem każdego wzorca projektowego jest rozwiązanie pewnego problemu w określonym kontekście. Wzorce interfejsów dotyczą kontekstów wymagających zdefiniowania lub przedefiniowania dostępu do klasy lub grupy klas. Jeśli na przykład mamy klasę wykonującą potrzebne nam operacje, ale o nazwach metod nieodpowiadających wymaganiom klienta, możemy zastosować wzorec *Adapter*.

Jeśli chcesz:	Zastosuj wzorzec:
<ul style="list-style-type: none">• Zadaptować istniejący interfejs klasy do postaci oczekiwanej przez klienta• Stworzyć prosty interfejs dla zestawu klas• Zdefiniować interfejs uwzględniający zarówno pojedyncze obiekty, jak i grupy obiektów• Oddzielić operacje abstrakcyjne od ich implementacji w celu wprowadzania w nich niezależnych zmian	<i>Adapter</i> <i>Facade</i> <i>Composite</i> <i>Bridge</i>