Język C#. Szkoła programowania

Autor: Klaus Michelsen
Tłumaczenie: Krystyna Zięba, Marzena Latawiec, Marek Bury, Piotr Lubicz
ISBN: 83-246-0358-1
Tytuł oryginału: C# Primer Plus
Format: B5, stron: 1128

Poznaj C# i rozpocznij podróż po świecie programowania

- Poznaj elementy języka i zasady programowania obiektowego.
- Wykorzystaj możliwości platformy .NET.
- Napisz i uruchom własne aplikacje.

C# to jeden z języków programowania wchodzących w skład platformy .NET. Według założeń producenta – firmy Microsoft – ma stanowić podstawowe narzędzie programistyczne dla tej platformy. C#, łączący w sobie najlepsze cechy języków Visual Basic, C++ i Java, jest łatwy do opanowania, a daje ogromne możliwości. Oparty na komponentach i obiektach doskonale nadaje się zarówno do tworzenia aplikacji dla komputerów osobistych, jak i dla urządzeń typu PocketPC. Twórcy aplikacji w C# mogą wybierać między doskonałym środowiskiem Visual Studio .NET a kilkoma narzędziami dostępnymi na licencji open-source.

Książka „Język C#. Szkoła programowania" to podręcznik, który wprowadzi Cię w arkana sztuki pisania programów w tym języku. Poznasz elementy języka C# i dowiesz się, na czym polega programowanie obiektowe. Nauczysz się korzystać z zaawansowanych możliwości oferowanych przez technologię obiektową, takich jak polimorfizm, interfejsy i struktury. Przeczytasz tu także o obsłudze wyjątków, tworzeniu dokumentacji w formacie XML w oparciu o komentarze w kodzie oraz o operacjach wejścia i wyjścia na plikach.

- Kompilowanie i uruchamianie programów w C#
- Typy i operatory
- Instrukcje warunkowe i pętle
- Tablice
- Klasy i obiekty
- Dziedziczenie
- Obsługa zdarzeń
- Rekurencja

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

SPIS TREŚCI

TWÓJ PIERWSZY PROGRAM OBIEKTOWY W C#

W tym rozdziale dowiesz się:

- Jakie niepodzielne elementy zawiera program źródłowy C#,
- Jakie konwencjonalne style nazywania klas, metod i zmiennych funkcjonują podczas programowania w C#,
- Na jakiej zasadzie działają operatory, argumenty i wyrażenia,
- Jak pisać i tworzyć instancje swoich własnych klas,
- Jak można wdrożyć teoretyczną dyskusję obiektową o klasach `Elevators` i `Person` z rozdziału 3., „C# — wycieczka z przewodnikiem, część I", aby utworzyć w pełni działający program C#,
- Jak wygląda prosty program obiektowy i jego ważne elementy,
- Jak inicjalizować zmienne instancji nowo utworzonych obiektów,
- W jaki sposób obiekty osiągają zdolność do zawierania zmiennych instancji, które zawierają inne obiekty,
- Jak programiści implementują zależności między klasami, aby umożliwić im współpracę i tworzyć programy obiektowe,
- Po co utworzono zunifikowany język modelowania (Unified Modeling Language — UML) i jak można go zastosować do ilustracji graficznej oraz modelowania zależności między klasami,
- W jaki sposób asocjacja, agregacja i kompozycja regulują zależności między klasami.

Wstęp

Dotychczas zapoznaliśmy się z dwoma programami źródłowymi przedstawionymi w listingach 3.1 w rozdziale 3. i 4.1 w rozdziale 4. Przedstawione tam konstrukcje języka C# w pewnym stopniu zależały od treści poszczególnych wierszy kodu i w rezultacie zajmowały się jednocześnie wieloma różnorodnymi, ale wzajemnie związanymi ze sobą aspektami. Aby nadrobić braki tej szybkiej wycieczki po C#, pierwsza część tego rozdziału będzie poświęcona przeglądowi podstawowych elementów C#.

W ostatniej części rozdziału zobaczymy pierwszy w tej książce program obiektowy C#. Jest on oparty na poprzednich teoretycznych dyskusjach obiektowych, a zwłaszcza na omówieniu symulacji windy na początku rozdziału 3. Pozwala m.in. zaobserwować, jak omówiony wcześniej związek między klasami `Elevator` i `Person` jest implementowany w C#.

Struktura leksykalna

Kiedy kompilator C# otrzymuje kawałek kodu źródłowego do kompilacji, staje przed pozornie zniechęcającym zadaniem rozszyfrowania długiej listy znaków (dokładniej, znaków Unicode, przedstawionych w dodatku E, „Zestaw znaków Unicode") i przekształcenia ich na odpowiedni MSIL o znaczeniu dokładnie tym samym, co oryginalny kod źródłowy. Aby znaleźć sens tej masy kodu źródłowego, musi rozpoznać niepodzielne elementy C# — niedające się rozbić części tworzące kod źródłowy C#. Przykładami takich niepodzielnych elementów są: nawias klamrowy (`{`), nawias (`(`) oraz słowa kluczowe, np. `class` i `if`. Zadanie, które wykonuje kompilator, związane z rozróżnieniem otwierających i zamykających nawiasów klamrowych, słów kluczowych, nawiasów itd., nazywa się *analizą leksykalną*. W zasadzie, kwestie leksykalne, którymi zajmuje się kompilator, odnoszą się do tego, jak znaki kodu źródłowego można przetłumaczyć na znaki zrozumiałe dla kompilatora.

Programy C# są zbiorem identyfikatorów, słów kluczowych, odstępów, komentarzy, literałów, operatorów i separatorów. Z większością tych elementów C# już spotkaliśmy się. W dalszym ciągu rozdziału dokonamy ich przeglądu, a także wprowadzimy jeszcze kilka innych aspektów.

Identyfikatory i style stosowania wielkich liter

Identyfikatory służą do nazywania klas, metod i zmiennych. Dowiedzieliśmy się już, jakich zasad należy przestrzegać, aby identyfikator był prawidłowy, oraz tego, że dobrze wybrane identyfikatory mogą poprawić klarowność kodu źródłowego i sprawić, że kod będzie samodokumentujący. Teraz wprowadzimy inny aspekt związany z identyfikatorami, a mianowicie styl stosowania wielkich liter.

Programiści często wybierają identyfikatory składające się z kilku słów, aby kod źródłowy był bardziej klarowny i samodokumentujący. Można by np. zastosować słowa „urodzenia rocznie". Jednakże kompilator reaguje na odstęp i każdy identyfikator podzielony na słowa za pomocą odstępu będzie błędnie interpretowany. Przykładowo zmiennej, która ma przedstawiać średnią prędkość na godzinę, nie można nazwać `srednia predkosc na godzine`[1]. Musimy wyrzucić odstępy, aby utworzyć jedną prawidłową nazwę, zachowując styl pozwalający czytelnikowi kodu źródłowego rozróżniać poszczególne słowa identyfikatora. Niektóre języki komputerowe akceptują konwencję `srednia_predkosc_na_godzine`. Natomiast w C# większość programistów stosuje uzgodnioną sekwencję dużych i małych liter w celu rozróżnienia poszczególnych słów identyfikatora.

W świecie C# istnieje wiele ważnych stylów używania wielkich liter:

- *Styl Pascal* — pierwsza litera każdego słowa w nazwie jest dużą literą, np. `SredniaPredkoscNaGodzine`,
- *Styl Camel* — tak jak Pascal, z tym wyjątkiem, że całe pierwsze słowo identyfikatora jest pisane małymi literami, np. `sredniaPredkoscNaGodzine`.

Styl *Pascal* jest zalecany do nazywania klas i metod, a styl *Camel* służy do nazywania zmiennych.

Wskazówka

Nie wszystkie języki komputerowe odróżniają małe i wielkie litery. W takich językach słowa `Srednia` i `srednia` są identyczne dla kompilatora. W celu uzyskania kompatybilności z tymi językami, należy unikać sytuacji, w których wielkość liter decyduje o rozróżnialności identyfikatorów typu `public`, które są dostępne dla innych języków.

Literały

Rozważmy dwa następujące wiersze kodu źródłowego:

```
int number;
number = 10;
```

`number` jest wyraźnie zmienną. W pierwszym wierszu deklarujemy, że `number` ma być typu `int`. W drugim wierszu przypisujemy `number` wartość `10`. Ale co to jest `10`? No właśnie, `10` nie może zmieniać swojej wartości i nazywa się *literałem*. Literały to nie tylko liczby. Mogą to być również znaki, takie jak `B`, `$` i `z`, a także tekst, np. `To`

[1] W wielu językach programowania pojawiają się kłopoty przy korzystaniu w identyfikatorach z polskich znaków diakrytycznych (ę, ś, ć itp.). Jeśli więc używamy polskich nazw, warto takich liter unikać — *przyp. tłum.*

`jest literał`. Literały mogą być przechowywane w dowolnej zmiennej, której typ jest kompatybilny z typem literału.

Komentarze i dokumentacja kodu źródłowego

Główną cechą komentarzy jest to, że są całkowicie ignorowane przez kompilator. Dotychczas poznaliśmy dwa sposoby tworzenia komentarzy; jednowierszowe oznaczamy za pomocą `//`, a wielowierszowe — przy użyciu `/* */`.

Istnieje jeszcze trzeci typ, który umożliwia zapisywanie dokumentacji jako części kodu źródłowego, co będzie pokazane w tym rozdziale, przy czym dodatkowo można umieścić tę dokumentację w osobnych plikach eXtensible Markup Language (XML). W tej chwili wystarczy docenić szczególnie przydatny wynik końcowy tej właściwości; trzeba tylko zajrzeć do dokumentacji bibliotek klas .NET Framework, która została stworzona przez utworzenie plików XML z komentarzy/dokumentacji znajdujących się w oryginalnym kodzie źródłowym.

Separatory

W języku C# separatory służą do oddzielania od siebie różnych elementów. Spotkaliśmy już wiele z nich. Przykładem może być powszechnie stosowany średnik `;`, który jest wymagany do zakończenia instrukcji. W tabeli 5.1 zebrano separatory, które już poznaliśmy.

Tabela 5.1. Ważne separatory w C#

Nazwa	Symbol	Cel
Nawiasy klamrowe	`{ }`	Służą do zamknięcia bloku kodu dla klas, metod oraz, co pokażemy później, instrukcji rozgałęzień i pętli.
Nawiasy	`( )`	Zawierają listy parametrów formalnych w nagłówkach metod oraz listy argumentów w instrukcjach wywoływania metod. W nawiasie muszą znajdować się również wyrażenia logiczne (boole'owskie) w instrukcji `if`, a także w instrukcjach rozgałęzień i pętli, co będzie pokazane później.
Średnik	`;`	Kończy instrukcję.
Przecinek	`,`	Oddziela parametry formalne w nawiasie nagłówka metody i oddziela argumenty w instrukcji wywołania metody.
Kropka	`.`	Służy do wyznaczania przestrzeni nazw zawartych wewnątrz innych przestrzeni i do określania klas wewnątrz tych przestrzeni, i metod (o ile są dostępne) wewnątrz klas i obiektów. Może być również stosowana do określania zmiennych instancji wewnątrz klas i obiektów (o ile są dostępne), ale tej praktyki należy unikać.

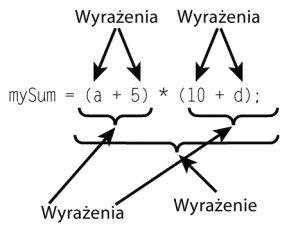
Operatory

Operatory są przedstawiane za pomocą symboli, takich jak +, =, == i *. Operatory działają na *argumenty*, które znajdują się obok operatora, np.:

```
sumTotal + 10
```

zawiera operator + otoczony dwoma argumentami, `sumTotal` i `10`. W tym kontekście operator + łączy dwa argumenty w celu uzyskania wyniku, a więc jest to *operator dwuargumentowy*. Niektóre operatory działają na tylko jeden argument; nazywa się je więc *operatorami jednoargumentowymi.*

Operatory wraz ze swoimi argumentami tworzą *wyrażenia*. Sam literał lub zmienna są również wyrażeniami, tak jak kombinacje literałów i zmiennych z operatorami. W rezultacie wyrażenia mogą być stosowane jako argumenty pod warunkiem, że są spełnione zasady, które mają zastosowanie do każdego operatora, co pokazano na następującym przykładzie:



`a`, 5 i 10, `d` to są wyrażenia, na które działa operator +. Natomiast

(a + 5) i (10 + d)

są również wyrażeniami, na które działa operator *. Na koniec,

(a + 5) · (10 + d)

można uważać za jedno wyrażenie. Operator przypisania = działa na to ostatnie wyrażenie i wyrażenie `mySum`. Wyrażenia są często zagnieżdżone w sobie nawzajem, tworząc hierarchię wyrażeń, tak jak w poprzednim przykładzie.

Operatory można podzielić na następujące kategorie: operatory przypisania, arytmetyczne, jednoargumentowe, równości, porównania, logiczne, warunkowe, przesunięcia i podstawowe.

W dalszych rozdziałach poświęcimy trochę więcej czasu na omówienie operatorów, ale na razie podsumujmy szybko te, które dotychczas napotkaliśmy.

- *Operator przypisania* (=) — powoduje, że argument po jego lewej stronie zmienia swoją wartość na wyrażenie znajdujące się po prawej, jak w wierszu

  ```
  29:         sumTotal = a + b;
  ```

  gdzie `a + b` można uważać za jeden argument.

- Dwuargumentowe *operatory arytmetyczne* (+ i *) — w następującym przykładzie

  ```
  a * b
  ```

  występuje mnożenie a i b bez zmiany ich wartości.
- *Operator konkatenacji* (+) — łączy dwa łańcuchy znakowe w jeden łańcuch.
- *Operator równości* (==) — porównuje dwa wyrażenia, aby sprawdzić, czy są równe, np.:

  ```
  leftExpression == rightExpression
  ```

  to prawda tylko wtedy, gdy oba wyrażenia będą równe; w przeciwnym razie będzie to fałsz.

Słowa kluczowe

W dodatku C podano 77 słów kluczowych C# (wszystkie). Dotychczas spotkaliśmy słowa kluczowe, takie jak `if`, `class`, `public`, `static`, `void`, `string`, `int` i `return`. Składnia (zasady językowe) operatorów i separatorów połączonych ze słowami kluczowymi tworzy definicję języka C#.

Pewne przemyślenia na temat symulacji windy

Program C# przedstawiony po poniższej analizie przypadku jest pierwszą próbą (prototypem) implementacji obiektowej symulacji windy. Analiza przypadku wprowadza kilka celów, problemów i rozwiązań prostej symulacji windy i usiłuje wprowadzić odpowiedni nastrój dla praktycznego przykładu C#. Jest to również przypomnienie dyskusji o hermetyzacji na początku rozdziału 3.

Pojęcia, cele i rozwiązania w programie symulacji windy — zbieranie cennych danych statystycznych do oceny systemu wind

Analiza przypadku bada strategię zbierania ważnych informacji z danej symulacji windy, aby odpowiedzieć na pytanie: „Czy system wind, który symulujemy, wykonuje prawidłowo zadanie przewożenia pasażerów między piętrami?”. Innymi słowy, gdybyśmy wzięli rzeczywisty system wind, wyciągnęli jego wszystkie stosowne cechy i przekształcili w symulację komputerową, to jak dobrze działałby taki system?

Statystyka

Liczbowy fakt lub dane, zwłaszcza obliczone z próbki, nazywa się *statystyką*.

Statystyka jest nauką, która zajmuje się zbieraniem, klasyfikacją, analizą i interpretacją liczbowych faktów lub danych, i która, przy użyciu matematycznych teorii prawdopodobieństwa, narzuca porządek i regularność agregatów mniej lub bardziej odmiennych elementów.

Termin statystyka stosuje się również w odniesieniu do samych liczbowych faktów lub danych.

Większość użytkowników, oceniając system wind, uważa za ważne dwie statystyki:

- czas, przez jaki przeciętna osoba musi czekać na piętrze po naciśnięciu przycisku na przyjazd windy,
- średni czas, jaki zajmuje podróż na jedno piętro.

Próba zebrania tych wartości z symulacji windy może wyglądać następująco.

Ponieważ obiekty `Person` (osoba) „podróżują" dzięki systemowi wind, dla nich jest oczywiste, że obiekty `Elevator` (winda) muszą świadczyć usługi. W rezultacie obiekty `Person` będą „wiedziały", jak dobrze działa system i będą w stanie zebrać niektóre z wymaganych, zasadniczych statystyk. Analogiczną strategią w świecie rzeczywistym byłoby przeprowadzenie wywiadów z użytkownikami systemu wind i zebranie pewnych ich doświadczeń z tym systemem.

Każdy obiekt `Person` programu symulacyjnego windy można zaimplementować w taki sposób, aby miał parę zmiennych instancji śledzących jego całkowity czas oczekiwania poza windą (jako odpowiedź na pierwszy wyróżnik) i średni czas jazdy na piętro (odnoszący się do drugiego wyróżnika). Wskazywałyby one, jak dobrze działa system wind i byłyby częścią statystyk zebranych dla każdej symulacji. Moglibyśmy nazwać te zmienne `totalWaitingTime` (całkowity czas oczekiwania) i `averageFloorTravelingTime` (średni czas jazdy na piętro).

Obliczenie `totalWaitingTime` wymagałoby metody umieszczonej wewnątrz obiektu `Person`, zawierającej polecenia uruchomienia stopera wbudowanego w komputer, za każdym razem gdy osoba „naciśnie przycisk" na danym obiekcie `Floor` (piętro), wzywając windę. Gdy tylko „przyjedzie" obiekt `Elevator`, stoper zostanie zatrzymany, a czas będzie dodany do bieżącej wartości `totalWaitingTime`.

I podobnie, `averageFloorTravelingTime` oblicza się inną metodą wewnątrz `Person`, uruchamiając stoper, gdy tylko obiekt `Person` „wejdzie" do obiektu `Elevator`. Kiedy `Person` dotrze do „miejsca przeznaczenia", stoper jest zatrzymywany, a czas jest dzielony przez liczbę „przebytych" pięter, aby uzyskać średni czas jazdy na piętro. Ten wynik jest przechowywany na liście razem z innymi wielkościami

averageFloorTravelingTime. Kiedy trzeba obliczyć końcową statystykę averageFloorTravelingTime, metoda wyliczy średnią z liczb przechowywanych na liście.

Wszystkie te obliczenia wykorzystujące uruchamianie i zatrzymywanie stoperów, sumowanie liczb, obliczanie średnich itd. nie są w żaden sposób interesujące dla innych obiektów w symulacji i nadmiernie komplikowałyby sprawy innym programistom, gdyby je ujawniono. W konsekwencji powinniśmy ukryć wszystkie zaangażowane tutaj metody przez zadeklarowanie ich jako private.

Również każdy obiekt Person musi być w stanie podać swój totalWaitingTime i averageFloorTravelingTime. Można to zrobić za pomocą dwóch metod public, nazwanych arbitralnie getTotalWaitingTime() i getAverageFloorTravelingTime(). Każdy inny obiekt wywołujący którąś z tych metod otrzyma odpowiednią statystykę.

Inny programista, który także pracuje nad tym projektem, pisze klasę do zbierania ważnych statystyk każdej symulacji. Nazwał tę klasę StatisticsReporter. Musi zapewnić, aby na końcu symulacji „przeprowadzono wywiad" z wszystkimi obiektami Person przez umożliwienie klasie StatisticsReporter zbierania ich statystyk totalWaitingTime i averageFloorTravelingTime. Teraz StatisticsReporter może to zrobić po prostu przez wywołanie metod getTotalWaitingTime() i getAverageFloorTravelingTime() każdego obiektu Person biorącego udział w danej symulacji.

Podsumowując:

- getTotalWaitingTime() i getAverageFloorTravelingTime() są częścią interfejsu, który ukrywa, czyli hermetyzuje wszystkie złożoności, nieistotne dla naszego programisty klasy StatisticsReporter.
- Podobnie, zmienne instancji obiektów Person powinny być ukryte za pomocą deklaracji private. Uniemożliwia to innym obiektom, ze StatisticsReporter włącznie, błędną zmianę tych wielkości. Innymi słowy, metody getTotalWaitingTime() i getAverageFloorTravelingTime() „przykrywają" zmienne instancji totalWaitingTime i averageFloorTravelingTime za pomocą hermetyzacji, jak opakowanie, umożliwiając klasie StatisticsReporter jedynie uzyskanie ich wartości, bez możliwości wprowadzania zmian.

Programowanie obiektowe — przykład praktyczny

Dotychczas nasza rozmowa o programowaniu obiektowym jest raczej teoretyczna. Omówiliśmy różnicę między klasami a obiektami oraz to, jak trzeba tworzyć instancje obiektów, zanim będą mogły rzeczywiście wykonywać jakiekolwiek czynności.

Widzieliśmy parę prostych klas w listingach 3.1 w rozdziale 3. i 4.1 w rozdziale 4., ale nie zawierały one żadnych rzeczywistych obiektów; były biernymi kontenerami, utworzonymi jedynie po to, aby zawierały metodę `Main()`. Żadnego z tych programów nie można właściwie uważać za obiektowy. Ponieważ mają one tylko metody zawierające sekwencje instrukcji wykonywanych jedna po drugiej, przypominają programy napisane w proceduralnym stylu programowania.

Przez zwykłe dodanie wielu metod i instrukcji do klasy `SimpleCalculator` z listingu 4.1 moglibyśmy, aczkolwiek wyjątkowo nieefektywnie, napisać prawidłową i skomplikowaną aplikację arkusza obliczeniowego, nie martwiąc się o pisanie innych klas, uczenie się teoretycznych zasad obiektowych lub nawet wykorzystanie jakichkolwiek dających się wymienić cech obiektowych C#. Struktura tego programu byłaby bliższa programowi napisanemu w języku proceduralnym, takim jak C lub Pascal, niż typowemu programowi obiektowemu C#.

I odwrotnie (oraz zadziwiająco), można by było również napisać program obiektowy w C lub Pascalu, ale byłoby to niewygodne, ponieważ te języki, w przeciwieństwie do C#, nie mają wbudowanej obsługi tego paradygmatu.

Aby nasze dotychczasowe dyskusje obiektowe uczynić bardziej praktycznymi i uniknąć ryzyka skonstruowania nieobiektowego, dużego programu arkusza obliczeniowego, przedstawię przykład ilustrujący parę ważnych cech C#, ściśle związanych z naszą teoretyczną dyskusją o klasach, obiektach i tworzeniu instancji.

Prezentacja SimpleElevatorSimulation.cs

Listing 5.1 zawiera kod źródłowy dla prostego programu symulacyjnego windy. Jego celem jest zilustrowanie, jak obiekt użytkownika wygląda w C# i jak jest tworzona instancja obiektu klasy użytkownika. Szczególnie pokazuje on, jak tworzony jest obiekt klasy `Elevator` oraz jak wywołuje on metodę obiektu `Person` o nazwie `NewFloorRequest()` (żądanie nowego piętra) i jak powoduje, że ta metoda zwraca numer żądanego „piętra", dzięki czemu `Elevator` może spełnić tę prośbę.

Na etapie projektowania kodu źródłowego w listingu 5.1 dokonano wielu abstrakcji, umożliwiających uproszczenie programu i skoncentrowanie się na zasadniczych, obiektowych częściach kodu źródłowego.

Poniżej podano krótki opis konfiguracji systemu windy, użytej dla tej symulacji, z podkreśleniem głównych różnic w stosunku do rzeczywistego systemu wind.

- Klasa `Building` (budynek) ma jeden obiekt klasy `Elevator` o nazwie `elevatorA`.
- Jeden obiekt `Person`, rezydujący w zmiennej `passenger` (pasażer), „wykorzystuje" `elevatorA` (windę A).
- Obiekt `Elevator` może „jechać" na dowolne „piętro" znajdujące się w zakresie określonym przez typ `int` (od –2147483648 do 2147483647). Jednak obiekt `Person` jest zaprogramowany na losowy wybór pięter od 1 do 30.

- Winda „pojedzie" natychmiast na „piętro przeznaczenia".
- „Ruchy" `elevatorA` będą wyświetlane na konsoli.
- `passenger` po „wejściu" do `elevatorA` pozostanie w `elevatorA` podczas całej symulacji i po prostu będzie wybierał nowe piętro, gdy jego poprzednie żądanie zostanie już spełnione.
- W tej symulacji zastosowano tylko klasy `Elevator`, `Person` i `Building`, natomiast pominięto klasy `Floor`, `StatisticsReporter` i inne klasy uważane za ważne dla pełnej symulacji.
- Na końcu każdej symulacji na konsoli zostanie wyświetlona całkowita liczba pięter przebytych przez `elevatorA`. Ta liczba może być bardzo ważną statystyką w poważnej symulacji windy.

Tak więc, mimo prostoty i wysokiej abstrakcji tej symulacji, możemy faktycznie wyciągnąć z niej jedną ważną statystykę i bylibyśmy w stanie (bez zbyt wielu dodatków) utworzyć małą, ale przydatną symulację umożliwiającą użytkownikowi uzyskanie cennego wglądu w rzeczywisty system windy.

Proszę poświęcić chwilę na zbadanie kodu źródłowego z listingu 5.1. Spróbujmy najpierw popatrzeć szerzej na kod. Proszę np. zwrócić uwagę na trzy definicje klas (`Elevator`, `Person` i `Building`), które tworzą cały program (oprócz wierszy od 1. do 4.). Następnie zwróćmy uwagę na metody zdefiniowane w każdej z tych trzech klas oraz zmienne instancji każdej z nich.

Uwaga!

Przypomnijmy sobie, że to, jak program jest wykonywany, nie zależy od kolejności, w jakiej metody klasy są zapisane w kodzie źródłowym. To samo dotyczy kolejności klas w programie. Można wybrać dowolną kolejność pasującą do naszego stylu. W listingu 5.1 wstawiłem klasę zawierającą metodę `Main()` jako ostatnią, a mimo to `Main()` jest pierwszą metodą, jaka będzie wykonana.

Typowy wynik listingu 5.1 jest pokazany pod listingiem. Ponieważ numery żądanych pięter są wybierane losowo, `"Odjeżdżam z piętra"` i `"Jadę na"` oraz `"Ilość przebytych pięter"` będą inne przy każdym uruchomieniu programu (z wyjątkiem pierwszego opuszczanego piętra, którym zawsze będzie piętro 1.).

LISTING 5.1. Kod źródłowy dla SimpleElevatorSimulation.cs

```
01:  // Prosta symulacja windy
02:
03: using System;
04:
05: class Elevator
06: {
07:     private int currentFloor = 1;
```

```
08:     private int requestedFloor = 0;
09:     private int totalFloorsTraveled = 0;
10:     private Person passenger;
11:
12:     public void LoadPassenger()
13:     {
14:         passenger = new Person();
15:     }
16:
17:     public void InitiateNewFloorRequest()
18:     {
19:         requestedFloor = passenger.NewFloorRequest();
20:         Console.WriteLine("Odjeżdżam z piętra: " + currentFloor
21:             + " Jadę na: " + requestedFloor);
22:         totalFloorsTraveled = totalFloorsTraveled +
23:             Math.Abs(currentFloor - requestedFloor);
24:         currentFloor = requestedFloor;
25:     }
26:
27:     public void ReportStatistic()
28:     {
29:         Console.WriteLine("Ilość przebytych pięter: " +
            ➥totalFloorsTraveled);
30:     }
31: }
32:
33: class Person
34: {
35:     private System.Random randomNumberGenerator;
36:
37:     public Person()
38:     {
39:         randomNumberGenerator = new System.Random();
40:     }
41:
42:     public int NewFloorRequest()
43:     {
44:          // Zwróć losowo wygenerowaną liczbę
45:         return randomNumberGenerator.Next(1,30);
46:     }
47: }
48:
49: class Building
50: {
51:     private static Elevator elevatorA;
52:
53:     public static void Main()
54:     {
55:         elevatorA = new Elevator();
56:         elevatorA.LoadPassenger();
```

```
57:         elevatorA.InitiateNewFloorRequest();
58:         elevatorA.InitiateNewFloorRequest();
59:         elevatorA.InitiateNewFloorRequest();
60:         elevatorA.InitiateNewFloorRequest();
61:         elevatorA.InitiateNewFloorRequest();
62:         elevatorA.ReportStatistic();
63:     }
64: }

Odjeżdżam z piętra: 1 Jadę na: 2
Odjeżdżam z piętra: 2 Jadę na: 24
Odjeżdżam z piętra: 24 Jadę na: 15
Odjeżdżam z piętra: 15 Jadę na: 10
Odjeżdżam z piętra: 10 Jadę na: 21
Ilość przebytych pięter: 48
```

Ogólna struktura programu

Zanim będziemy kontynuować bardziej szczegółową analizę programu, popatrzmy na rysunek 5.1. Łączy on ilustrację użytą na rysunku 3.1 w rozdziale 3. z konkretnym programem C# z listingu 5.1.

Klasy `Elevator` i `Person` z listingu 5.1 definiują wersje naszych znajomych już odpowiedników, wyłowionych ze świata rzeczywistego. Są one pokazane graficznie obok swoich odpowiedników C# na rysunku 5.1. Każda część klas `Elevator` i `Person` zapisana w kodzie źródłowym C# (wskazywana nawiasami klamrowymi) została połączona strzałką ze swoim odpowiednikiem graficznym. Proszę zwrócić uwagę, jak metody `public` dwóch klas (interfejs) hermetyzują prywatne, ukryte zmienne instancji. W tym przypadku nie były potrzebne żadne metody prywatne.

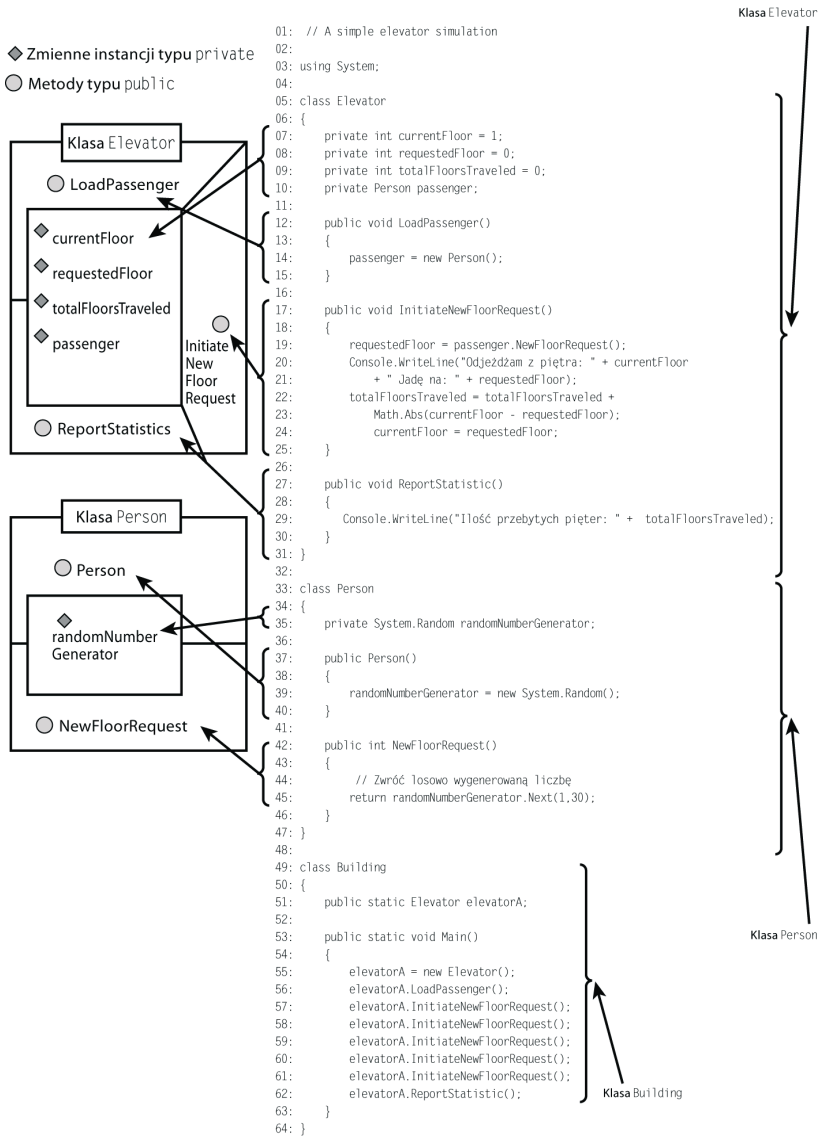
Klasa `Building` ma jedną windę (`Elevator`), która jest reprezentowana przez zmienną instancji `elevatorA`, zadeklarowaną w wierszu 51. Ma również metodę `Main()`, gdzie zaczyna się aplikacja. Obiekt tej klasy nigdy nie jest tworzony w programie. Jest ona wykorzystywana przez środowisko uruchomieniowe .NET w celu uzyskania dostępu do `Main()` i uruchomienia programu.

Tak jak w poprzednich listingach, poniżej przedstawiłem krótkie objaśnienie wierszy w kodzie źródłowym. Wiele wierszy pominąłem, ponieważ były już omawiane w poprzednim przykładzie.

LISTING 5.2. Krótka analiza listingu 5.1

```
05: Początek definicji klasy o nazwie Elevator

07: Deklaracja zmiennej instancji typu int o nazwie currentFloor;
    ➥ustawienie jej poziomu dostępu na private oraz jej początkowej
    ➥wartości na 1.
```



RYSUNEK 5.1. Połączenie rysunku 3.1 z rzeczywistym programem C#

```
01:  // A simple elevator simulation
02:
03: using System;
04:
05: class Elevator
06: {
07:     private int currentFloor = 1;
08:     private int requestedFloor = 0;
09:     private int totalFloorsTraveled = 0;
10:     private Person passenger;
11:
12:     public void LoadPassenger()
13:     {
14:         passenger = new Person();
15:     }
16:
17:     public void InitiateNewFloorRequest()
18:     {
19:         requestedFloor = passenger.NewFloorRequest();
20:         Console.WriteLine("Odjeżdżam z piętra: " + currentFloor
21:             + " Jadę na: " + requestedFloor);
22:         totalFloorsTraveled = totalFloorsTraveled +
23:             Math.Abs(currentFloor - requestedFloor);
24:             currentFloor = requestedFloor;
25:     }
26:
27:     public void ReportStatistic()
28:     {
29:        Console.WriteLine("Ilość przebytych pięter: " +  totalFloorsTraveled);
30:     }
31: }
32:
33: class Person
34: {
35:     private System.Random randomNumberGenerator;
36:
37:     public Person()
38:     {
39:         randomNumberGenerator = new System.Random();
40:     }
41:
42:     public int NewFloorRequest()
43:     {
44:          // Zwróć losowo wygenerowaną liczbę
45:         return randomNumberGenerator.Next(1,30);
46:     }
47: }
48:
49: class Building
50: {
51:     public static Elevator elevatorA;
52:
53:     public static void Main()
54:     {
55:         elevatorA = new Elevator();
56:         elevatorA.LoadPassenger();
57:         elevatorA.InitiateNewFloorRequest();
58:         elevatorA.InitiateNewFloorRequest();
59:         elevatorA.InitiateNewFloorRequest();
60:         elevatorA.InitiateNewFloorRequest();
61:         elevatorA.InitiateNewFloorRequest();
62:         elevatorA.ReportStatistic();
63:     }
64: }
```

10: Deklaracja zmiennej passenger, która może zawierać
➥obiekt klasy Person. O klasie Person mówi się, że odgrywa rolę
➥pasażera w swoim związku z klasą Elevator.

12: Początek definicji metody o nazwie LoadPassenger(). Zadeklarowanie
➥jej jako public to część interfejsu klasy Elevator.

```
14: Tworzenie nowego obiektu klasy Person przy użyciu
    ➥słowa kluczowego new. Przypisanie tego obiektu do zmiennej
    ➥passenger.

17: Początek definicji metody o nazwie InitiateNewFloorRequest().
    ➥Zadeklarowanie jej jako public to część interfejsu klasy Elevator.

19: Wywołanie metody NewFloorRequest() obiektu passenger; przypisanie
    ➥liczby zwracanej przez tę metodę do zmiennej requestedFloor.

20-21: Drukowanie informacji o "ruchu" operatora na konsoli poleceń.

22-23: Obliczanie liczby pięter przebytych prze windę w czasie jednej,
       ➥konkretnej jazdy (wiersz 23.). Dodanie tego wyniku do całkowitej
       ➥liczby pięter już przebytych przez windę.
24: Zezwolenie na "jazdę" windy na żądane piętro przez przypisanie
    ➥wartości requestedFloor do currentFloor.

29: Drukowanie zmiennej totalFloorsTraveled, za każdym razem gdy
    ➥wywoływana jest metoda ReportStatistics().

33: Początek definicji klasy o nazwie Person.

35: Deklaracja randomNumberGenerator jako zmiennej, która może
    ➥zawierać obiekt klasy System.Random.

37: Początek definicji specjalnej metody, która będzie wywoływana
    ➥automatycznie, za każdym razem kiedy będzie tworzony nowy obiekt
    ➥klasy Person.

39: Tworzenie nowego obiektu klasy System.Random przy użyciu
    ➥słowa kluczowego C#: new; przypisanie tego obiektu do zmiennej
    ➥randomNumberGenerator.

42: Początek definicji metody o nazwie NewFloorRequest(). public
    ➥deklaruje ją jako część interfejsu klasy Person. int określa, że
    ➥zwraca ona wartość typu int.

45: Person decyduje, na które piętro chce jechać,
    ➥zwracając losowo wybraną liczbę z zakresu od 1 do 30.

51: Klasa Building deklaruje zmienną instancji typu Elevator,
    ➥o nazwie elevatorA. O klasie Building mówi się, że ma związek
    ➥posiadania (lub agregacji) z klasą Elevator.

53: Początek definicji metody Main(), gdzie zacznie się program.

55: Tworzenie obiektu klasy Elevator za pomocą słowa kluczowego new;
    ➥przypisanie tego obiektu do zmiennej elevatorA.
```

```
56: Wywołanie metody LoadPassenger() obiektu elevatorA.

57-61: Wywołanie metody InitiateNewFloorRequest()
       ➥obiektu elevatorA, 5 razy.

62: Wywołanie metody ReportStatistic() obiektu elevatorA.
```

Głębsza analiza SimpleElevatorSimulation.cs

Poniżej omówiono ważne kwestie związane z listingiem 5.1.

Definiowanie klasy, która ma być skonkretyzowana w naszym własnym programie

Wiemy już, jak definiuje się klasę. Wciąż zwracam uwagę na wiersz 5., ponieważ pierwszy raz definiujemy swoją własną klasę, która jest konkretyzowana, i dlatego potem stosowana do tworzenia obiektu w naszym programie.

```
05: class Elevator
```

Inicjalizacja zmiennych

Nowa cecha przedstawiona w wierszu 7. jest kombinacją instrukcji deklaracji i instrukcji przypisania. Nazywamy to *inicjalizacją*. `private int currentFloor;` jest prostą deklaracją, a przez dodanie na końcu `= 1`, skutecznie przypisujemy 1 do `currentFloor` podczas tworzenia lub zaraz po utworzenie obiektu, do którego należy zmienna instancji.

```
07:     private int currentFloor = 1;
```

Zmienna często wymaga wartości początkowej, zanim będzie można ją wykorzystać w obliczeniach, w których bierze udział. `currentFloor` w wierszu 7. jest dobrym przykładem. Chcemy, aby winda ruszyła z piętra numer 1, więc inicjalizujemy ją tą wartością.

Kiedy zmiennej przypisuje się wartość przed jej udziałem w obliczeniach, to mówi się, że zmienna jest *inicjalizowana*. Istnieją dwie ważne metody inicjalizacji zmiennej instancji. Można:

- przypisać zmiennej wartość początkową w jej deklaracji (jak w wierszu 7.) lub
- wykorzystać cechę C#, zwaną konstruktorem. Kod źródłowy zawarty w *konstruktorze* będzie wykonywany wtedy, gdy jest tworzony obiekt klasy, bo to idealny moment dla wszelkich inicjalizacji. Z konstruktorami zapoznamy się wkrótce.

Zmiennym instancji, które nie są wyraźnie inicjalizowane w kodzie źródłowym, kompilator C# automatycznie przypisze wartość domyślną. Gdybyśmy np. nie przypisali wartości 1 do `currentFloor` w wierszu 7., C# nadałby domyślną wartość 0.

Warto wyraźnie inicjalizować swoje zmienne

Nie należy zostawiać kompilatorowi C# inicjalizacji swoich zmiennych. Wyraźne inicjalizacje sprawiają, że kod jest bardziej klarowny, a poza tym lepiej nie polegać na inicjalizacjach kompilatora i ich wartościach domyślnych, które mogą się zmienić w przyszłości i wprowadzić błędy do kodu źródłowego.

Wskazówka

Można zadeklarować składową klasy bez wyraźnego podawania jej dostępności dzięki opuszczeniu słowa kluczowego `public` lub `private`. Wtedy dostępność jest domyślnie ustawiana na `private`.

Tym niemniej, lepiej używać słowa kluczowego `private` w celu zadeklarowania wszystkich prywatnych składowych klasy, aby poprawić jej przejrzystość.

Deklarowanie zmiennej reprezentującej obiekt danej klasy

W wierszu 10. stwierdzamy, że zmienna instancji `passenger` może zawierać obiekt utworzony z klasy `Person`.

```
10:     private Person passenger;
```

Jeszcze nie wstawiliśmy tutaj szczególnego obiektu `Person`; aby to zrobić, potrzebowalibyśmy instrukcji przypisania, którą zobaczymy trochę później w tym rozdziale. Dotąd jedynie stwierdzamy, że obiekt `Elevator` może „transportować" jeden obiekt `passenger`, który musi być klasy `Person`. Przykładowo w `Elevator` nie mogą być przechowywane obiekty `Dog`, `Airplane` czy `Submarine`, gdybyśmy kiedyś zdefiniowali takie klasy w naszym programie.

Teraz przeskoczmy na chwilę do wierszy od 33. do 47. Przez zdefiniowanie w tych wierszach klasy `Person` skutecznie utworzyliśmy nowy typ użytkownika. A więc, oprócz możliwości deklarowania, że zmienna będzie typu `int` lub typu `string`, możemy również zadeklarować, że będzie ona typu `Person`, co właśnie robimy w wierszu 10.

Uwaga!

`int` i `string` są to wbudowane, wstępnie zdefiniowane typy. Klasy, które piszemy i definiujemy w swoim kodzie źródłowym, są typami użytkownika (ang. *custom-made types*).

Proszę zwrócić uwagę, że wiersz 10. jest ściśle związany z wierszami 14., 19. i wierszami od 33. do 47. Wiersz 14. przypisuje nowy obiekt klasy `Person` do zmiennej `passenger`; wiersz 19. wykorzystuje pewną funkcję zmiennej `passenger`, a więc

obiektu `Person` przez wywołanie jednej z jego metod, a wiersze od 33. do 47 definiują klasę `Person`.

Wskazówka

Kolejność deklaracji i definicji składowych klasy jest dowolna. Jednak warto podzielić składowe klasy na sekcje zawierające składowe o podobnych modyfikatorach dostępu w celu poprawienia przejrzystości.

Poniżej przedstawiono przykład powszechnie stosowanego stylu:

```
class NazwaKlasy
{
     // deklaracje prywatnych zmiennych instancji
     // definicje metod prywatnych
     // definicje metod publicznych
}
```

Zmienne instancji reprezentujące stan obiektu Elevator

Wiersze od 7. do 10. zawierają listę zmiennych instancji, które w tej symulacji uważałem za istotne dla opisu obiektu `Elevator`.

```
07:     private int currentFloor = 1;
08:     private int requestedFloor = 0;
09:     private int totalFloorsTraveled = 0;
10:     private Person passenger;
```

Stan obiektu `Elevator` jest opisany przez zmienne instancji zadeklarowane w następujących wierszach:

- *Wiersz 7.* — zmienna `currentFloor` śledzi piętro, na którym znajduje się obiekt `Elevator`.
- *Wiersz 8.* — nowe żądanie piętra będzie przechowywane w zmiennej `requestedFloor`; obiekt `Elevator` będzie usiłował jak najszybciej spełnić to żądanie (wiersz 24.), co zależy od szybkości procesora naszego komputera.
- *Wiersz 9.* — kiedy tworzony jest obiekt `Elevator`, można uważać go za „fabrycznie nowy". On jeszcze nigdy nie jeździł do góry ani na dół, więc `totalFloorsTraveled` musi na początku zawierać wartość 0. Uzyskuje się to przez inicjalizację wartością 0.

  Liczba przebytych pięter jest dodawana do `totalFloorsTraveled` (wiersze od 22. do 23.) tuż przed zakończeniem jazdy (wiersz 24.).
- *Wiersz 10* — pasażer (pasażerowie) windy podejmuje ostatecznie decyzję o numerach pięter odwiedzanych przez windę. Obiekt `Person` rezydujący w `passenger` wybiera piętra, które musi odwiedzić nasz obiekt `Elevator`. Żądanie jest uzyskiwane od `passenger` w wierszu 19. i przypisywane zmiennej `requestedFloor`.

Abstrakcja i wybór zmiennych instancji

Przypomnijmy sobie dyskusję o abstrakcji na początku rozdziału 3. Celem związanym z abstrakcją jest identyfikacja zasadniczych cech klasy obiektów, które są istotne dla naszego programu komputerowego.

Podczas procesu podejmowania decyzji, które zmienne instancji włączyć do definicji klasy oraz deklarowania ich w kodzie źródłowym, programista stosuje pojęcie abstrakcji w bardzo praktyczny sposób.

Mógłbym np. spróbować włączyć do klasy `Elevator` zmienną instancji typu `string`, taką jak kolor, deklarując, co następuje:

```
private string color; ←—— Bezużyteczna do naszych celów
```

Ale jaki użytek mogę z tego zrobić? Mógłbym, być może, przypisać do niej `string` „czerwony" i napisać metodę, która po każdym wywołaniu drukowałaby na konsoli poleceń:

```
Mój kolor to: czerwony
```

Ale byłoby to całkowicie nieistotne dla celu, który próbujemy uzyskać w naszej małej, prostej symulacji, więc jest to nieekonomiczne i niepotrzebnie komplikuje klasę `Elevator`.

Inny programista mógłby dołączyć zmienną instancji klasy `Elevator`, która zlicza liczbę jazd wykonywanych przez windę; mógłby nazwać ją `totalTrips` i zadeklarować następująco:

```
private int totalTrips; ←—— Potencjalnie przydatna
```

Klasę `Elevator` można by potem tak zaprojektować, żeby metoda dodawała 1 do `totalTrips`, za każdym razem gdy żądanie zostało spełnione. Ta zmienna instancji umożliwiałaby śledzenie innej, być może ważnej statystyki, a więc potencjalnie może być przydatna.

Jak widać, przy podejmowaniu decyzji, które zmienne instancji należy dołączyć, można stosować wiele różnych sposobów przedstawiania obiektów świata rzeczywistego. Wybór zmiennych instancji zależy od programisty oraz od tego, co chce on zrobić z każdym obiektem.

Umożliwienie obiektowi klasy Elevator przyjęcia nowego pasażera

Obiekt klasy `Elevator` musi być w stanie przyjąć nowy obiekt klasy `Person`. Uzyskuje się to za pomocą metody `LoadPassenger()` rezydującej wewnątrz obiektu `Elevator` (patrz wiersz 12.). Metoda `LoadPassenger()` musi być zadeklarowana jako `public`, aby był do niej dostęp spoza obiektu. Wywołanie `LoadPassenger()` odbywa się w wierszu 56. listingu 5.1.

```
12:     public void LoadPassenger()
```

Opisywanie klas i obiektów

Rozważmy następującą instrukcję deklaracji:

```
private Person passenger;
```

Można zastosować wiele różnych, mniej lub bardziej prawidłowych sposobów opisania, o czym mówi to zdanie, a zwłaszcza tego, jak identyfikuje obiekt. Według mnie, dobry, chociaż może trochę nieporęczny, jest następujący opis:

„`passenger` jest zmienną zadeklarowaną tak, aby zawierała obiekt klasy `Person`".

Ponieważ ten opis jest trochę przydługi, często stosuję następujące wyrażenie:

„`passenger` jest zmienną zadeklarowaną tak, aby zawierała obiekt `Person`".

W rozdziale 6., „Typy, część I — typy proste", dowiemy się, dlaczego najbardziej prawidłowym, ale również najdłuższym jest następujący opis:

„`passenger` jest zmienną zadeklarowaną tak, aby zawierała referencję do obiektu klasy `Person`".

Tworzenie nowego obiektu za pomocą słowa kluczowego new

Ręka w rękę z wierszem 10. idzie wiersz 14.

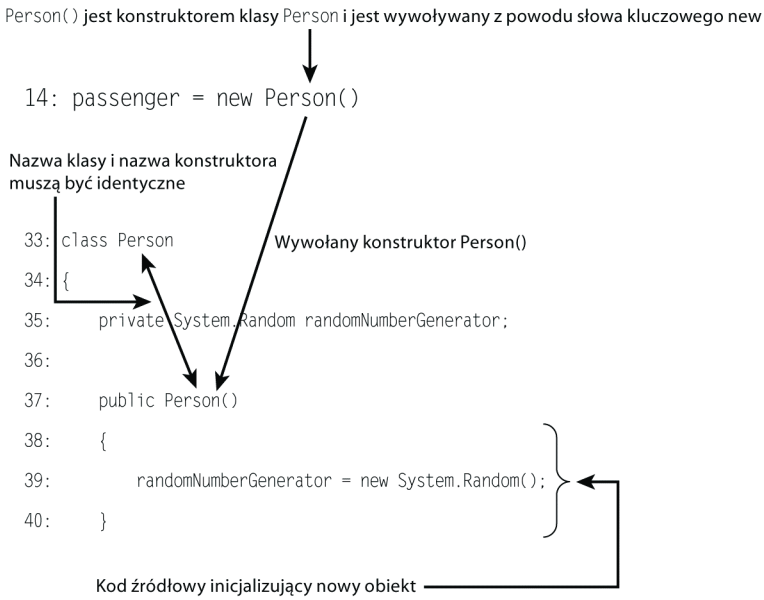
```
14:         passenger = new Person();
```

Występujące w tym wierszu słowo `new` (nowy) jest słowem kluczowym w C#, stosowanym w celu tworzenia nowego obiektu. Wytworzy ono nową instancję (obiekt) klasy `Person`. Ten nowy obiekt klasy `Person` jest następnie przypisany do zmiennej `passenger`. Teraz `passenger` zawiera obiekt `Person`, który można wywołać i zastosować do wykonania czynności.

Uwaga!

Kiedy tworzy się instancję obiektu, często potrzebna jest inicjalizacja zmiennych instancji tego obiektu. *Konstruktor* jest to metoda specjalnego rodzaju, która wykonuje zadanie inicjalizacji.

Metoda określana jako konstruktor musi mieć taką samą nazwę jak jej klasa. Tak więc konstruktor dla klasy `Person` nazywa się `Person()` (patrz wiersz 37.). Ilekroć za pomocą słowa kluczowego `new` tworzony jest nowy obiekt `Person`, automatycznie wywoływany jest konstruktor `Person()` w celu wykonania niezbędnych inicjalizacji. To wyjaśnia obecność w wierszu 14. nawiasów, które są wymagane zawsze, kiedy wywołuje się dowolną metodę i dowolny konstruktor. (Zobacz rysunek na następnej stronie).



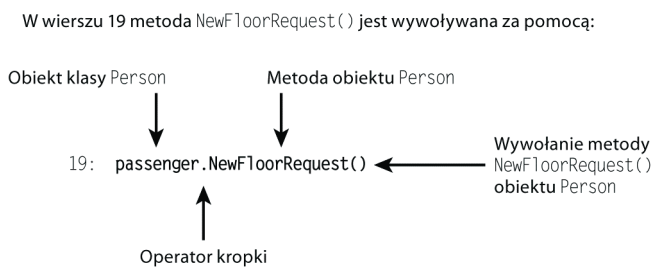
Umożliwienie odbierania i spełniania żądań pasażerów przez obiekt Elevator

Przez wywołanie metody `InitiateNewFloorRequest` obiekt `Elevator` jest instruowany, aby:

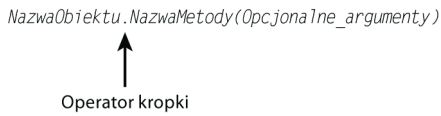
- odebrał nowe żądanie od `passenger` (wiersz 19.),
- wydrukował piętro, z którego odjeżdża i piętro przeznaczenia (wiersze 20. i 21.),
- zaktualizował statystykę `totalFloorsTraveled` (wiersze 22. i 23.),
- spełnił żądanie obiektu klasy `Person` w zmiennej `passenger` (wiersz 24.).

```
17:     public void InitiateNewFloorRequest()
18:     {
19:         requestedFloor = passenger.NewFloorRequest();
20:         Console.WriteLine("Odjeżdżam z piętra: " + currentFloor
21:             + " Jadę na: " + requestedFloor);
22:         totalFloorsTraveled = totalFloorsTraveled +
23:             Math.Abs(currentFloor - requestedFloor);
24:         currentFloor = requestedFloor;
25:     }
```

Zacznijmy od wiersza 19.:



Tutaj stosujemy następującą, wprowadzoną wcześniej składnię:

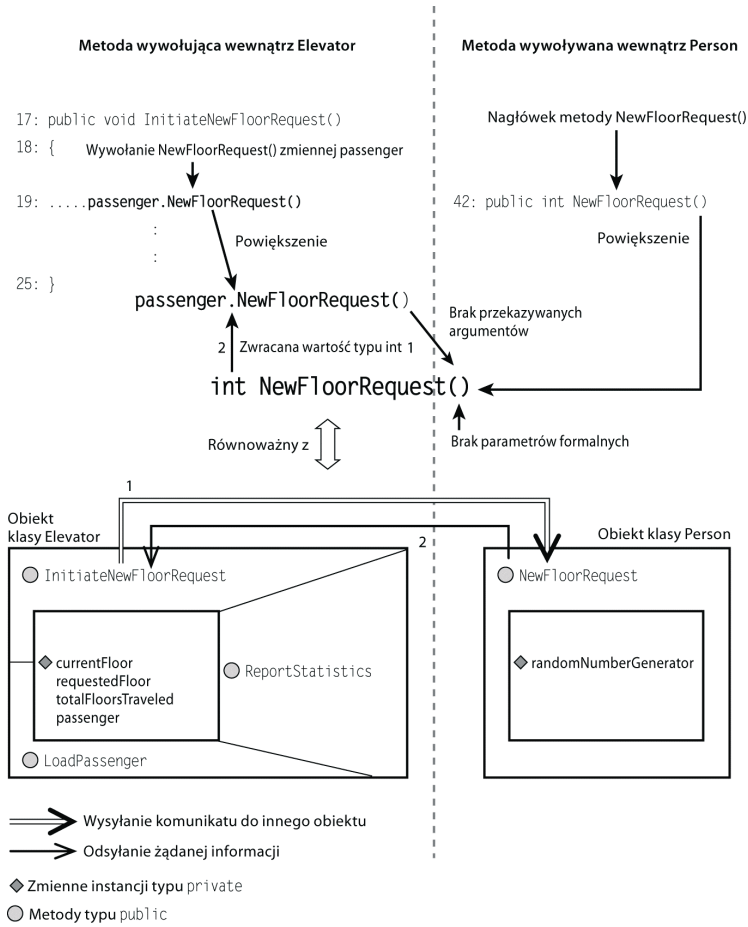


gdzie użyty jest operator kropki (`.`) w celu odwołania się do metody rezydującej wewnątrz obiektu. Z operatora tego korzystaliśmy już wiele razy, np. występował przy wywoływaniu metody `WriteLine()` z `System.Console` za pomocą `System.Console.WriteLine("Pa, pa!")`. Tym razem jednak, zamiast wywoływać już napisaną metodę z .NET Framework, wywołujemy własną metodę użytkownika z klasy `Person`.

Uwaga!

W kodzie źródłowym można zauważyć stałą definicję klasy. Natomiast obiekt jest dynamiczny i ożywia się podczas wykonywania programu.

Zamiast wywoływania metody rezydującej w tym samym obiekcie, wywołujemy metodę rezydującą w innym obiekcie. Tutaj o obiekcie `Elevator` mówi się, że wysyła komunikat do obiektu `Person`. Wywołanie metody rezydującej w innym obiekcie jest podobne do wywoływania metody lokalnej. Na rysunku 5.2 połączyłem rysunek 3.1 z mechanizmem ogólnym, pokazanym na rysunku 4.1. Górna połowa rysunku ilustruje wywołanie `NewFloorRequest()` przy zastosowaniu takiego samego stylu graficznego, co na rysunku 4.1. Krok 1. symbolizuje wywołanie `NewFloorRequest()`. Ponieważ dla `NewFloorRequest()` nie określono parametrów formalnych, więc do metody nie są przekazywane żadne argumenty. Ten krok jest równoważny z krokiem 1. z dolnej połowy rysunku. Po zakończeniu `NewFloorRequest()` metoda zwraca wartość typu `int`, co symbolizuje strzałka 2. Graficzny równoważnik kroku 2. jest pokazany w dolnej połowie tego rysunku. Jak widzieliśmy poprzednio (wiersz 18. listingu 4.1 w rozdziale 4.), po zakończeniu kroku 2. można zastąpić `passenger.NewFloorRequest()` wartością typu `int`. Na koniec tę wartość można przypisać do zmiennej `requestedFloor` za pomocą operatora przypisania `=`.



RYSUNEK 5.2. Wywoływanie metody innego obiektu

Po odebraniu przez windę nowego żądania od obiektu klasy `Person` w zmiennej `passenger` w wierszu 19. obecne położenie obiektu `Elevator` znajduje się w zmiennej `currentFloor`, a inne piętro, do którego musi on dotrzeć, znajduje się w `requestedFloor`. Jeżeli `Elevator` jest „fabrycznie nowy" i właśnie został utworzony, `currentFloor` będzie mieć wartość 1. Jeżeli `Elevator` odbył już jedną lub kilka „jazd", `currentFloor` będzie równy miejscu przeznaczenia ostatniej „jazdy". Wiersz 20. wydrukuje wartość `currentFloor` przy użyciu znanej metody `WriteLine()`. W wierszu 21. jest następnie drukowane żądanie `passenger`.

Proszę zwrócić uwagę, że wiersze 22. i 23. przedstawiają tylko jedną instrukcję. Dowodzi tego jeden średnik znajdujący się na końcu wiersza 23. Instrukcja została rozłożona na dwa wiersze z powodu braku miejsca. Wiersz 23. został wcięty w sposób wskazujący na powiązanie z wierszem 22.; to wskazuje czytelnikowi, że zajmujemy się tylko jedną instrukcją.

Wartość bezwzględna

Wartością bezwzględną liczby dodatniej jest sama liczba.

Wartością bezwzględną liczby ujemnej jest liczba bez znaku minus.

Przykłady:

Wartością bezwzględną (–12) jest 12.

Wartością bezwzględną 12 jest 12.

W matematyce wartość bezwzględna jest podawana za pomocą dwóch pionowych kresek otaczających literał lub zmienną, np.:

```
|-12| = 12
```

Zastosowanie Math.Abs()

Metoda `Abs()` klasy `Math` rezydującej w .NET Framework zwraca wartość bezwzględną wysłanego do niej argumentu. W rezultacie wywołanie metody `Math.Abs(99)` zwraca 99, podczas gdy `Math.Abs(-34)` zwraca 34.

Obliczając odległość przebytą przez windę, jesteśmy zainteresowani dodatnią liczbą przebytych pięter, bez względu na to, czy winda jedzie do góry, czy na dół. Jeżeli liczbę przebytych pięter obliczymy za pomocą następującego wyrażenia:

```
currentFloor – requestedFloor
```

to zawsze wtedy, gdy `requestedFloor` będzie większe od `currentFloor`, otrzymamy w wyniku ujemną liczbę przebytych pięter. Ta ujemna liczba jest następnie dodawana do `totalFloorsTraveled`; więc w tym przypadku liczba przebytych pięter nie będzie dodana, lecz odjęta od `totalFloorsTraveled`. Tego problemu można uniknąć przez dodawanie wartości bezwzględnej z (`currentFloor – requestedFloor`), jak to zostało zrobione w wierszu 23.:

```
Math.Abs(currentFloor - requestedFloor);
```

Instrukcję tę można rozbić na następujące podinstrukcje:

1. `Math.Abs(currentFloor - requestedFloor)` — obliczyć liczbę pięter przebytych przy następnej jeździe windy,
2. `totalFloorsTraveled + Math.Abs(currentFloor - requestedFloor)` — dodać liczbę pięter przebytych przy następnej jeździe do bieżącej wartości `totalFloorsTraveled`,
3. Przypisać wynik z pkt. 2. do `totalFloorsTraveled`.

Występowanie `totalFloorsTraveled` po obu stronach operatora przypisania może na początku wydawać się trochę dziwne. Jednak jest to bardzo przydatna operacja dodawania liczby pięter przebytych przy ostatniej jeździe windy do początkowej wartości `totalFloorsTraveled`. Tu `totalFloorsTraveled` jest

licznikiem odległości windy, cały czas śledzącym całkowitą ilość przebytych pięter. Ten typ instrukcji będzie omówiony bardziej szczegółowo w rozdziale 6.

Niespodziewanie prosta instrukcja przypisania w wierszu 24. reprezentuje „jazdę windy". Spełnia żądanie przekazane od `passenger`. Przez przypisanie wartości zmiennej `requestedFloor` do `currentFloor` możemy powiedzieć, że winda „przesunęła się" z wartości `currentFloor`, którą zawierała przed tym przypisaniem, do `requestedFloor`.

Klasa Person — szablon dla pasażera windy

Wiersz 33. zaczyna definicję drugiej klasy użytkownika, która będzie stosowana jako szablon w celu utworzenia obiektu.

```
33: class Person
```

Jak widać, obiekt `Person` jest tworzony i przechowywany w obiekcie `Elevator`, z którego jest wywoływana metoda `NewFloorRequest()`, przekazująca obiektowi `Elevator` informację, gdzie chce jechać jej pasażer (zmienna `passenger` przechowująca obiekt klasy `Person`).

W wierszu 35. obiektowi klasy `Person` umożliwiono żądanie piętra przez wyposażenie go w obiekt zawierający metodę, która generuje losowe liczby. Obiekt jest przechowywany w zmiennej `randomNumberGenerator`. Klasa, z której tworzony jest obiekt, znajduje się w bibliotece klas .NET i nazywa się `System.Random`. W wierszu 35. deklarujemy, że `randomNumberGenerator` ma zawierać obiekt typu `System.Random`.

```
35:     private System.Random randomNumberGenerator;
```

Ponieważ `randomNumberGenerator` jest zmienną instancji klasy `Person`, jest zadeklarowana jako `private`. Proszę zwrócić uwagę, że po tej deklaracji `randomNumberGenerator` jest nadal pusta. Żaden obiekt nie został jeszcze przypisany do tej zmiennej. Musimy przypisać obiekt do `randomNumberGenerator`, zanim ta zmienna będzie mogła generować liczby losowe. Wiersze od 37. do 40. spełniają to ważne zadanie, dzięki temu, że zawierają konstruktor dla klasy `Person`.

Pojęcie konstruktora zostało już omówione i nie będę dalej drążył tego tematu. W tej chwili nie jest ważne, aby w pełni zrozumieć mechanizm konstruktora pod warunkiem, że zdajemy sobie sprawę, iż zasadniczą częścią jest wiersz 39., gdzie nowy obiekt klasy `System.Random` jest przypisany do `randomNumberGenerator` każdego nowo utworzonego obiektu klasy `Person`.

```
37:     public Person()
38:     {
39:         randomNumberGenerator = new System.Random();
40:     }
```

`NewFloorRequest()` zdefiniowana w wierszach od 42. do 46. będzie po wywołaniu generowała i zwracała liczbę losową, która wskazuje nowe piętro żądane przez `passenger`. Wiersz 45. znajduje liczbę losową z zakresu od 1 do 30 (podanego w nawiasie `.Next(1,30)`). Słowo kluczowe `return` wysyła tę liczbę losową z powrotem do wywołującego, którym w tym przypadku jest metoda `InitiateNewFloorRequest()` obiektu `Elevator`. Nie będziemy tutaj omawiać bliższych szczegółów klasy `System.Random`. Zainteresowanych bliższym zbadaniem tej klasy odsyłam do dokumentacji .NET Framework.

```
42:     public int NewFloorRequest()
43:     {
44:          // Zwróć losowo wygenerowaną liczbę
45:         return randomNumberGenerator.Next(1,30);
46:     }
```

Uwaga!

Programista, który implementuje klasę `Elevator`, wykorzystuje metodę `NewFloorRequest()` obiektu `Person`. Jednak nie musi znać definicji metody, aby ją zastosować. Najprawdopodobniej nie będzie go interesować, w jaki sposób obiekt `Person` decyduje o zwracanych wartościach i z powodzeniem może nie zdawać sobie sprawy z generatorów liczb losowych i tym podobnych. Nagłówek metody i krótki opis intencji metody w zupełności mu wystarczy. Na informacje, w jaki sposób jest realizowana intencja, nie ma tutaj miejsca.

Nagłówek metody tworzy umowę z jej użytkownikiem. Nadaje metodzie nazwę i decyduje o tym, ile argumentów należy wysłać za pomocą wywołania metody. Następnie metoda obiecuje, że albo nie zwróci żadnej wartości (jeżeli jest określona jako `void`), albo zwróci jedną wartość określonego typu.

Także pod warunkiem, że nie „majstrujemy" przy nagłówku metody (nie zmieniamy nazwy metody, liczby ani typu jej parametrów formalnych lub typu zwracania), możemy tworzyć wszystkie rodzaje zawiłych procesów pozwalających obiektowi `Person` decydować o następnym piętrze przeznaczenia.

Metoda Main() — prowadzenie symulacji

W wierszach od 53. do 63. znajduje się nasza stara znajoma, nieunikniona metoda `Main()`.

```
53:     public static void Main()
54:     {
55:         elevatorA = new Elevator();
56:         elevatorA.LoadPassenger();
57:         elevatorA.InitiateNewFloorRequest();
58:         elevatorA.InitiateNewFloorRequest();
59:         elevatorA.InitiateNewFloorRequest();
60:         elevatorA.InitiateNewFloorRequest();
```

```
61:          elevatorA.InitiateNewFloorRequest();
62:          elevatorA.ReportStatistic();
63:      }
```

Nawet wtedy, gdy `Main()` znajduje się, jak w tym przypadku, na końcu programu, to zawsze zawiera instrukcje, które mają być wykonane jako pierwsze po uruchomieniu programu. Na `Main()` można spojrzeć jak na „punkt kontrolny" całego programu. Aby kierować ogólnym przebiegiem wykonania, wykorzystuje ona ogólne funkcje innych klas w programie.

W skrócie, metoda `Main()` tworzy początkowo obiekt `Elevator` i przypisuje go do `elevatorA` (wiersz 55.); każe mu wpuścić pasażera (wiersz 56.); prosi `elevatorA`, aby „zażądał następnego piętra" i wykonał to pięć razy (wiersze od 57. do 61.); a na koniec prosi obiekt `Elevator`, aby podał statystykę, którą zebrał w czasie tych pięciu jazd (wiersz 62.).

Proszę zwrócić uwagę, że metoda `Main()` stosunkowo łatwo prowadzi tę symulację. Wszystkie trudne, szczegółowe prace są wykonywane w klasach `Elevator` i `Person`.

Związki klas i UML

Trzy klasy zdefiniowane przez użytkownika wraz z klasą `System.Random` w listingu 5.1 współpracują ze sobą, aby zapewnić funkcjonalność naszego prostego programu symulacji windy. Klasa `Building` zawiera obiekt `Elevator` i wywołuje jego metody, obiekt `Elevator` zatrudnia obiekt `Person` do kierowania swoimi ruchami, a obiekt `Person` wykorzystuje obiekt `System.Random` w celu decydowania o następnym piętrze. W dobrze skonstruowanych programach obiektowych klasy współpracują w podobny sposób — każda za pomocą swoich własnych unikatowych funkcji przyczynia się do działania całego programu.

Jeżeli dwie klasy mają współpracować, musi istnieć związek (współdziałanie) między nimi. Między dwiema klasami może istnieć kilka zależności, a decyzja, jakie konkretne związki powinny być zaimplementowane w danym programie, należy do projektanta programu. Ta faza projektowania zwykle ma miejsce podczas identyfikowania klas programu (etap 2b procesu projektowania opisanego w rozdziale 2., „Twój pierwszy program C#"). W dalszej części rozdziału omówiono kilka powszechnie stosowanych związków.

`Building-Elevator` i `Elevator-Person` to dwa rodzaje związków, które wykorzystamy jako przykłady.

Związek Building-Elevator

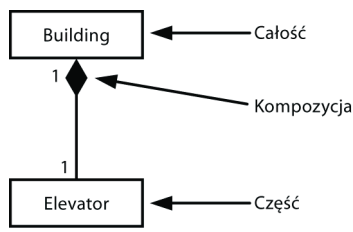
Typowy budynek składa się z wielu różnych części, takich jak piętra, ściany, sufity, dach i czasami windy. W naszym przypadku można powiedzieć, że `Building`, który symulujemy, posiada `Elevator` jako jedną ze swoich części. Czasami nazywa się

to zależnością całość / część. Zaimplementowaliśmy ten związek w listingu 5.1 przez zadeklarowanie zmiennej instancji typu `Elevator` wewnątrz klasy `Building` (wiersz 51.):

```
51:     private static Elevator elevatorA;
```

Dzięki temu `Building` może zawierać obiekt `Elevator` i wywoływać jego metody `public`. Klasa może mieć wiele różnych zmiennych instancji zawierających wiele obiektów różnych klas. Moglibyśmy np. również wyposażyć `Building` w zmienne instancji reprezentujące liczbę obiektów `Floor`. Ta idea konstruowania klasy (`Building`) za pomocą innych klas (`Elevator` lub `Floor` albo obu i wielu innych) nazywa się ogólnie *agregacją*, a towarzyszące jej zależności to *związki agregacyjne*.

Jeżeli związek agregacyjny (jak w przypadku związku `Building-Elevator`) odzwierciedla sytuację, gdzie jedna klasa jest integralną częścią innej, możemy nazwać tę agregację *kompozycją*. (Zaraz zobaczymy, dlaczego związek `Elevator-Person` jest agregacją, ale nie jest kompozycją.) Związek kompozycyjny można zilustrować za pomocą schematu klasy zunifikowanego języka modelowania (Unified Modeling Language — UML), pokazanego na rysunku 5.3. Dwie prostokątne ramki symbolizują klasy, a łącząca je linia z czarnym rombem (wskazującym na całą klasę) ilustruje związek kompozycyjny między klasami. Obie klasy są oznaczone liczbą 1, aby wskazać, że jeden `Building` (budynek) ma jedną `Elevator` (windę).



RYSUNEK 5.3. Diagram UML symbolizujący kompozycję

Zunifikowany język modelowania (UML) — Lingua Franca modelowania obiektowego

Pseudokod jest przydatną pomocą w wyrażaniu algorytmów, które są implementowane w pojedynczych metodach, ponieważ czyta się je od góry do dołu, tak jak środowisko uruchomieniowe wykonuje program i dlatego, że abstrahuje on od sztywnego rygoru języka komputerowego (średniki, nawiasy itd.). Jednak klasy mogą składać się z wielu metod, a większe programy składają się z wielu klas. Pseudokod nie jest odpowiednim narzędziem do ilustrowania modeli powiązań klas programu, ponieważ klasy zrywają z sekwencyjnym, proceduralnym sposobem myślenia (każda klasa może potencjalnie mieć związek z inną klasą zdefiniowaną w programie) i dlatego, że format pseudokodu jest zbyt szczegółowy, aby zapewnić przegląd dużego programu obiektowego.

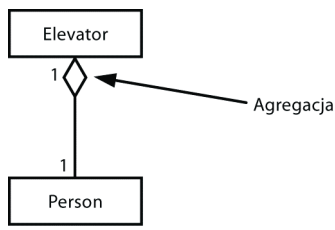
Aby skutecznie zaprezentować związki klas i całą architekturę programu obiektowego, potrzebujemy języka, który umożliwia abstrahowanie od wewnętrznych szczegółów metod, a zamiast tego zapewnia środki do wyrażania związków klas i pojęć obiektowych na odpowiednim poziomie szczegółowości. W tym celu większość programistów obiektowych, bez względu na swój język programowania, stosuje obecnie język przedstawiania graficznego o nazwie *zunifikowany język modelowania* (Unified Modeling Language — UML). UML to język o wielu cechach i potrzebna jest cała książka, aby go wystarczająco zaprezentować; tu przedstawiono tylko mały podzestaw UML.

Szczegółowe informacje o UML można uzyskać w niedochodowej organizacji Object Management Group (OMG) (*www.omg.org*) na stronie *www.omg.org/uml*. Napisano wiele dobrych książek o UML, m.in. *The Unified Modeling Language User Guide*, którą napisali twórcy UML, Grady Booch, James Rumbaugh i Ivar Jacobson.

Związek Elevator-Person

Przycisk jest integralną częścią windy, ale pasażer już nie. (Winda jest czynna bez pasażera, ale nie bez przycisków). Więc gdybyśmy nawet w naszej implementacji (z powodów abstrakcyjnych) uczynili pasażera permanentną częścią `Elevator` (obiekt `Person` pozostaje wewnątrz `Elevator` przez całą symulację), nie jest to związek kompozycyjny, a jedynie agregacja. Związek ten ilustruje UML na rysunku 5.4. Proszę zwrócić uwagę, że biały romb, w przeciwieństwie do czarnego rombu na rysunku 5.3, symbolizuje agregację.

RYSUNEK 5.4. Diagram UML symbolizujący agregację

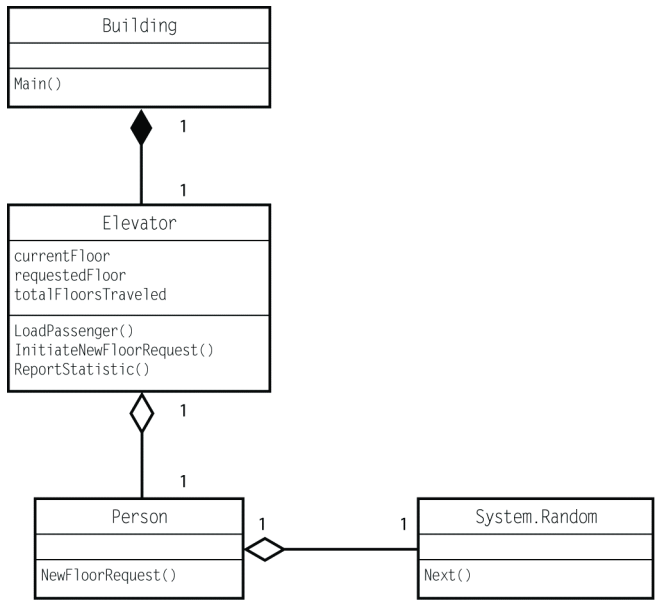


Cały schemat klasy UML dla programu z listingu 5.1 został pokazany na rysunku 5.5. UML umożliwia, tak jak to pokazano, podzielenie prostokąta reprezentującego klasę na trzy przedziały — górny przedział zawiera nazwę klasy, w środkowym mieszczą się zmienne instancji (lub atrybuty), a w dolnym — metody (zachowanie) należące do klasy.

Asocjacje

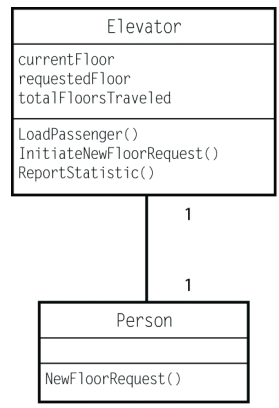
Stałe związki między klasami, takie jak związki agregacyjne i kompozycyjne, omówione w poprzednich rozdziałach, są ogólnie nazywane *związkami strukturalnymi* lub, bardziej oficjalnie, *asocjacjami*. Istnieją jednak inne typy asocjacji, które nie są agregacjami (a więc nie są również kompozycjami). Aby dać przykład takiej asocjacji, rozważmy następujący scenariusz. Chcemy uczynić naszą symulację windy bardziej realistyczną, zatem zmieniamy pokazaną wcześniej początkową listę abstrakcji,

RYSUNEK 5.5. Diagram UML klas z listingu 5.1



umożliwiając wielu obiektom `Person` wchodzenie, podróżowanie i wychodzenie z obiektu `Elevator`, zamiast tylko jednego obiektu `Person` pozostającego na stałe w windzie. Teraz już o żadnym obiekcie `Person` nie można powiedzieć, że jest stałą częścią obiektu `Elevator`. Możemy po prostu powiedzieć, że klasa `Elevator` jest skojarzona z klasą `Person`. Asocjacja jest pokazana za pomocą zwykłej linii, tak jak na rysunku 5.6.

RYSUNEK 5.6. Asocjacja klas Elevator/Person



Inne przykłady asocjacji, które nie są agregacjami:

- `Pracownik` pracuje w `Firmie`,
- `KlientBanku` współpracuje z `KasjeremBanku`.

Uwaga!

Asocjacje, które precyzyjnie łączą dwie klasy, nazywają się asocjacjami binarnymi; jest to najpopularniejszy rodzaj asocjacji.

Podsumowanie

Ten rozdział składa się z dwóch głównych części. Pierwsza dotyczy leksykalnej struktury programu źródłowego C#. W drugiej części zamieszczono przykład programu obiektowego, który jest bezpośrednio związany z dyskusją z rozdziału 3. o abstrakcji i hermetyzacji.

Poniżej przedstawiono najważniejsze zagadnienia omówione w tym rozdziale.

Na program źródłowy C# można popatrzeć jak na zbiór identyfikatorów, słów kluczowych, odstępów, komentarzy, literałów, operatorów i separatorów.

C# jest językiem rozróżniającym małe i wielkie litery. Aby kod był klarowniejszy dla innych czytelników, ważne jest, aby zachować pewien styl stosowania małych i wielkich liter. Preferowanymi stylami, które są używane w różnych konstrukcjach C#, są style Pascal (`ToJestStylPascal`) i Camel (`toJestStylCamel`).

Literał ma wartość, która jest zapisana w kodzie źródłowym (co widzisz, to otrzymasz). Przykładem literału może być wartość 10 lub wartość „To jest pies".

Separatory, takie jak średniki (`;`), przecinki (`,`) i kropki (`.`),w C# oddzielają od siebie różne elementy.

Operatory działają na argumenty. Argumenty łączą się z operatorami, tworząc wyrażenia.

Zmienne instancji muszą być inicjalizowane przy tworzeniu obiektu. Odbywa się to albo automatycznie, przez środowisko uruchomieniowe, albo przez inicjalizację podczas deklaracji, albo za pomocą konstruktora klasy.

Obiekt może zawierać referencję do innego obiektu w zmiennej instancji. Taki stały związek nazywa się asocjacją.

Obiekt tworzy się przy użyciu słowa kluczowego `new`.

W programie obiektowym klasy współpracują w celu zapewnienia funkcjonalności programu.

Dwie klasy mogą współpracować poprzez pozostawanie w związku.

Częstymi związkami asocjacyjnymi są agregacje i kompozycje.

Zunifikowany język modelowania (ang. *Unified Modeling Language* — UML) jest dotąd najpopularniejszym językiem modelowania graficznego, stosowanym do wyrażania i ilustrowania projektów programów obiektowych.

Pytania kontrolne

1. Co to jest analiza leksykalna?
2. Jakie są niepodzielne części programu C#?
3. Co to są style Pascal i Camel? Dla jakich części programu C# należy je stosować?
4. Jaka jest główna różnica między zmiennymi a literałami?
5. Co to są operatory i argumenty? Jaki jest między nimi związek?
6. Czy 50 jest wyrażeniem? A `(50 + x)`? Czy jest typu `public`?
7. Podaj przykłady typowych słów kluczowych w C#.
8. Dlaczego pseudokod nie nadaje się do wyrażania ogólnej konstrukcji programu obiektowego?
9. Jakiego rodzaju związek istnieje między obiektem `KlientBanku` a obiektem `KasjerBanku`? Jak to się wyraża w UML?
10. Jakiego rodzaju związek istnieje między obiektem `Serce` a obiektem `LudzkieCiało`? Jak to się wyraża w UML?
11. Jakiego rodzaju związek istnieje między obiektem `Zarowka` a obiektem `Lampa`? Jak to się wyraża w UML?
12. Jak implementuje się związek asocjacyjny w C#?
13. Jak można inicjalizować zmienne instancji przy tworzeniu obiektu?
14. Opisz zmienną `passenger`, która jest zadeklarowana w następującym wierszu:

```
private Person passenger;
```

Ćwiczenia z programowania

Spraw poprzez zmianę kodu źródłowego, aby program z listingu 5.1 realizował następujące funkcje.

1. Wydrukuj `Zaczęła się symulacja` na konsoli poleceń zaraz po uruchomieniu programu.
2. Wydrukuj `Symulacja skończyła się` jako ostatnią rzecz, tuż przed zakończeniem programu.
3. Zamiast wybierać piętra z zakresu od 1 do 30, niech klasa `Person` wybiera piętra od 0 do 50.

4. Przy pierwszej jeździe winda rusza z piętra numer 0 zamiast z piętra numer 1.
5. Obiekt `Elevator` wykonuje 10 jazd zamiast obecnych 5.
6. Obecnie `Elevator` zlicza wszystkie przebyte piętra za pomocą zmiennej `totalFloorsTraveled`. Zadeklaruj zmienną instancji w klasie `Elevator`, która będzie śledzić, ile jazd wykonuje ta winda. Tę zmienną należy nazwać `totalTripsTraveled`. `Elevator` powinien aktualizować tę zmienną, dodając jeden do `totalTripsTraveled` po każdej jeździe. Uaktualnij metodę `ReportStatistics` klasy `Elevator`, aby drukowała nie tylko `totalFloorsTraveled`, ale także `totalTripsTraveled`, z wyjaśnieniem, po co jest to drukowane.
7. Dodaj zmienną instancji do klasy `Elevator`, aby mogła zawierać nazwę windy. Można ją nazwać `myName`. Jakiego typu powinna być ta zmienna instancji? Czy należy ją zadeklarować jako `private` czy `public`? Napisz konstruktor, przy użyciu którego można zadawać wartość tej zmiennej przy tworzeniu obiektu `Elevator` za pomocą `new` i przypisać go do zmiennej. (Uwaga! Konstruktor jest metodą i musi mieć taką samą nazwę jak jego klasa. Ten konstruktor w swoim nagłówku musi mieć parametr formalny typu `string`). Ustaw wywołanie konstruktora, kiedy używa się słowa kluczowego `new`, przez wstawienie nazwy windy jako argumentu; między nawiasami zamiast `new Elevator()` wpisz

   ```
   new Elevator("WindaA").
   ```

   Za każdym razem gdy `Elevator` zakończy jazdę, powinien drukować swoją nazwę razem z opuszczanym piętrem i piętrem docelowym. Innymi słowy, zamiast drukować

   ```
   Odjeżdżam z piętra: 2 Jadę na: 24
   ```

   powinien drukować:

   ```
   WindaA: Odjeżdżam z piętra: 2 Jadę na: 24
   ```

   gdzie `WindaA` jest jego nazwą rezydującą wewnątrz `myName`.