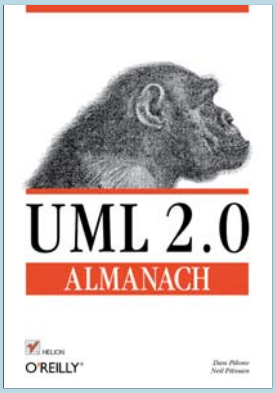
UML 2.0. Almanach



Autor: Dan Pilone, Neil Pitman
Tłumaczenie: Łukasz Piwko
ISBN: 978-83-246-0822-5
Tytuł oryginału: UML 2.0 in a Nutshell
Format: B5, stron: 248

Wyczerpujący przewodnik po języku UML 2.0

- Specyfikacja języka UML 2.0
- Modelowanie statyczne i dynamiczne
- Rozszerzanie i zastosowania UML-a

Ujednolicony język modelowania (UML) początkowo służył do opisu elementów oprogramowania, jednak z powodu swej elegancji i przejrzystości zyskuje na popularności w zakresie modelowania zagadnień z innych dziedzin. W związku z tym coraz więcej osób ma szansę zetknąć się z diagramami w języku UML. Jeśli sięgnąłeś po tę książkę, prawdopodobnie czeka to także Ciebie. Chciałbyś wiedzieć, co oznaczają różne zakończenia linii na diagramach klas albo zrozumieć skomplikowany diagram interakcji? Zajrzyj do środka.

„UML 2.0. Almanach" to kompletny podręcznik dla użytkowników tego języka. Dzięki tej książce poznasz podstawy modelowania w UML-u. Nauczysz się tworzyć i rozumieć diagramy statyczne, na przykład klas, pakietów czy struktur złożonych, a także diagramy zachowania, takie jak przypadków użycia, aktywności czy interakcji. Dowiesz się, jak wszechstronne zastosowania ma ten język oraz w jaki sposób można go rozszerzać do wykonywania specyficznych zadań. Znajdziesz tu także krótkie wprowadzenie do języka Object Constraint Language (OCL) oraz architektury sterowanej modelem (MDA).

- Podstawy modelowania w UML-u
- Diagramy statyczne i diagramy zachowania
- Dobór odpowiedniego rodzaju diagramu
- Znaczenie symboli, notacji i linii
- Rozszerzanie UML-a za pomocą etykiet, stereotypów i profili
- Architektura sterowana modelem
- Język Object Constraint Language (OCL)
- Praktyczne wskazówki z zakresu modelowania

Poznaj tajniki modelowania w języku UML 2.0

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

Spis treści

Wstęp **9**

1. Podstawy UML-a **15**
Zaczynamy 15
Historia 15
Podstawy UML-a 16
Specyfikacje UML-a 17
Używanie UML-a 18
Modelowanie 19
Praktyczne zasady UML-a 23

2. Diagramy klas **25**
Klasy 25
Atrybuty 26
Operacje 33
Metody 38
Klasy abstrakcyjne 38
Powiązania 39
Interfejsy 44
Szablony 46
Różne wersje diagramów klas 48

3. Diagramy pakietów **53**
Reprezentacja 53
Widoczność 54
Import pakietów i dostęp do nich 55
Łączenie pakietów 56
Różne wersje diagramów pakietów 57

4. Struktury złożone 65
Struktury złożone 65
Kolaboracje 73
Przypadki kolaboracji 75

5. Diagramy komponentów 77
Komponenty 77
Widoki komponentów 78

6. Diagramy wdrożenia 87
Artefakty 87
Węzły 89
Wdrażanie 93
Nietypowe diagramy wdrożenia 96

7. Diagramy przypadków użycia 99
Przypadki użycia 99
Aktorzy 100
Zaawansowane modelowanie przypadków użycia 103
Zasięg przypadków użycia 108

8. Diagramy stanów 111
Maszyny stanowe zachowań 111
Stany 113
Rozszerzanie maszyny stanów 123
Protokołowe maszyny stanów 123
Pseudostany 125
Przetwarzanie zdarzeń 126
Nietypowe diagramy stanów 127

9. Diagramy aktywności 129
Aktywności i akcje 129
Znaczniki sterowania 136
Aktywności 137
Zaawansowane modelowanie aktywności 144

10. Diagramy interakcji 155
Co to są interakcje 155
Uczestnicy interakcji 156
Komunikaty 158
Wykonywanie interakcji 163
Stany niezmienne 164

Zdarzenia 166
Ślady 166
Fragmenty wyodrębnione 167
Wystąpienia interakcji 178
Dekompozycja 179
Kontynuacje 182
Przepływy czasowe sekwencji 183
Alternatywne notacje interakcji 184

11. Metki, stereotypy i profile UML 193
Modelowanie i UML w kontekście 194
Stereotypy 196
Metki 198
Ograniczenia 199
Profile UML 199
Narzędzia a profile 201

12. Tworzenie efektywnych diagramów 203
Diagramy tapetowe 203
Zbyt duży zakres 208
Jeden diagram — jedna abstrakcja 209
Poza UML-em 211

A MDA — Model-Driven Architecture 215
Co to jest MDA 215
Modele MDA 216
Decyzje projektowe 219
Łączenie modeli w jedną całość 221
Transformacja modeli 222
Języki formalnego opisu MDA 223

B Object Constraint Language 225
Podstawy OCL-a 225
Składnia OCL-a 226
Zaawansowane modelowanie OCL-a 229

Skorowidz 233

ROZDZIAŁ 2.

Diagramy klas

Diagramy klas należą do najbardziej podstawowych typów diagramów UML-a. Stosuje się je do przedstawiania statycznych powiązań w programach — mówiąc inaczej: jak wszystko łączy się w jedną całość.

Pisząc program, cały czas trzeba podejmować decyzje: jakie klasy przechowują odniesienia do innych klas, która klasa „posiada" inną klasę itd. Diagramy klas pozwalają na przedstawienie fizycznej struktury systemu.

Klasy

Klasa reprezentuje grupę obiektów o wspólnym stanie i zachowaniu. Klasę można traktować jako plan obiektu w systemie zorientowanym obiektowo. W UML-u klasa jest rodzajem *klasyfikatora*. Na przykład Volkswagen, Toyota i Ford to marki samochodów, a zatem można je zaprezentować za pomocą klasy o nazwie `Samochód`. Każdy konkretny samochód jest *egzemplarzem* tej klasy, czyli *obiektem*. Klasa może reprezentować konkretne namacalne obiekty, takie jak na przykład faktura. Może być abstrakcyjna, np. dokument lub pojazd (w odróżnieniu np. od faktury i motocykla o pojemności powyżej 1000 cm^3). Może także reprezentować obiekty niematerialne, takie jak np. strategia inwestycji wysokiego ryzyka.

Klasa ma postać prostokąta podzielonego na *komórki*. Komórka to wydzielony obszar prostokąta, do którego można wpisywać pewne informacje. Pierwsza komórka zawiera nazwę klasy, druga atrybuty (patrz „Atrybuty"), a trzecia operacje (patrz „Operacje"). W celu zwiększenia czytelności diagramu można ukryć wybrane komórki. Czytając diagram, nie można domyślać się treści brakujących komórek — to, że są ukryte, wcale nie oznacza, że są puste. Do klas można wstawiać komórki z dodatkowymi informacjami, takimi jak wyjątki lub zdarzenia, ale działania te wykraczają już poza typową notację.

UML proponuje następujące zasady dotyczące nazw klas:

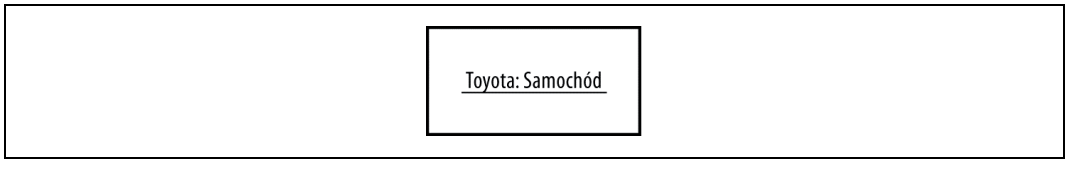
- powinny być pisane z wielkiej litery,
- powinny być wyśrodkowane w górnej komórce,
- powinny być pisane tłustym drukiem,
- nazwy klas *abstrakcyjnych* powinny być pisane kursywą (patrz „Klasy abstrakcyjne").

Na rysunku 2.1 przedstawiono prostą klasę.

Rysunek 2.1. Prosta klasa

Obiekty

Obiekt jest *egzemplarzem* klasy. Na przykład klasa o nazwie `Samochód` może mieć kilka egzemplarzy: jeden czerwony dwudrzwiowy samochód, jeden niebieski czterodrzwiowy samochód i jeden zielony samochód typu hatchback. Każdy egzemplarz klasy `Samochód` jest obiektem i może mieć własną nazwę, aczkolwiek w diagramach obiektów często używa się obiektów anonimowych, czyli bez nazw. Zazwyczaj po nazwie obiektu stawia się dwukropek, a po nim nazwę typu obiektu (tzn. klasy). Aby pokazać, że dany obiekt jest egzemplarzem klasy, należy podkreślić jego nazwę i typ. Na rysunku 2.2 pokazano egzemplarz klasy `Samochód` o nazwie `Toyota`. Warto zwrócić uwagę, że na tym rysunku puste komórki zostały ukryte.



Rysunek 2.2. Egzemplarz klasy Samochód

Atrybuty

Szczegóły klasy (kolor samochodu, liczba boków figury itd.) przedstawiane są jako *atrybuty*. Atrybuty mogą być prostymi typami podstawowymi (liczby całkowite, liczby zmiennopozycyjne itd.) lub powiązaniami do innych, bardziej skomplikowanych obiektów (patrz „Powiązania").

Atrybut można przedstawić przy użyciu jednej z dwóch dostępnych notacji: wewnętrznej (ang. *inlined*) lub powiązań z innymi klasami. Ponadto dostępna jest notacja, za pomocą której można pokazywać liczność, unikalność oraz uporządkowanie. W tej części rozdziału opisujemy oba rodzaje notacji, a następnie przedstawiamy szczegóły specyfikacji atrybutów.

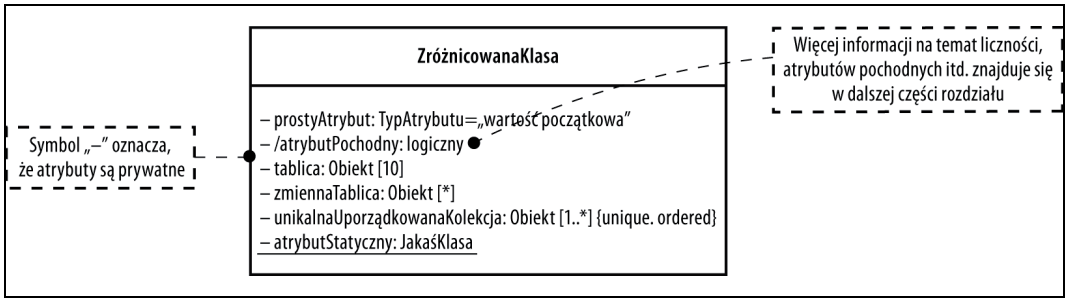
Atrybuty wpisane

Atrybuty klasy można wymienić bezpośrednio w prostokącie. Nazywają się wtedy *atrybutami wpisanymi* (ang. *inlined attributes*). Pomiędzy atrybutami wpisanymi a przedstawianymi za pomocą powiązań nie ma żadnych różnic semantycznych — ich wybór zależy tylko od tego, jak dużo szczegółów chcemy zaprezentować (lub, w przypadku typów fundamentalnych takich jak liczby całkowite, ile szczegółów *zdołamy* przekazać).

W celu zaprezentowania atrybutu wewnątrz klasy należy umieścić go w jej drugiej komórce. Atrybuty wpisane w UML-u nazywają się *notacją atrybutów*. Korzystają one z następującej notacji:

```
widoczność / nazwa : typ liczność = domyślna
 {łańcuchy właściwości i ograniczenia}

widoczność ::= {+|-|#|~}

liczność ::= [górna..dolna]
```

Na rysunku 2.3 zademonstrowano różne aspekty notacji atrybutów na przykładzie listy kilku atrybutów.



Rysunek 2.3. Przykładowe atrybuty

Elementy składni są następujące:

widoczność
: Informuje o widoczności atrybutu. Do wyboru są następujące symbole: +, -, # i ~, oznaczają one odpowiednio: `public`, `private`, `protected` i `package` (patrz „Widoczność" w rozdziale 3.).

/
: Oznacza atrybut *pochodny*. Atrybut pochodny to taki, którego wartość można obliczyć na podstawie innych atrybutów klasy. Patrz „Atrybuty pochodne".

nazwa
: Rzeczownik lub krótkie wyrażenie stanowiące nazwę atrybutu. Z reguły pierwsza litera pierwszego członu jest mała, a pierwsze litery wszystkich następnych członów są wielkie.

typ
: Typ atrybutu w postaci innego klasyfikatora. Zazwyczaj klasa, interfejs lub typ wbudowany, np. `int`.

liczność
: Określa liczbę egzemplarzy typu atrybutu, do których odnosi się atrybut. Może nie zostać podany (wtedy oznacza 1), może być liczbą całkowitą lub zakresem wartości rozdzielonych symbolem . . podanym w nawiasach kwadratowych. Symbol * służy do oznaczania braku górnego limitu. Gwiazdka użyta samodzielnie oznacza zero lub więcej. Patrz „Liczność".

domyślna
: Domyślna wartość atrybutu.

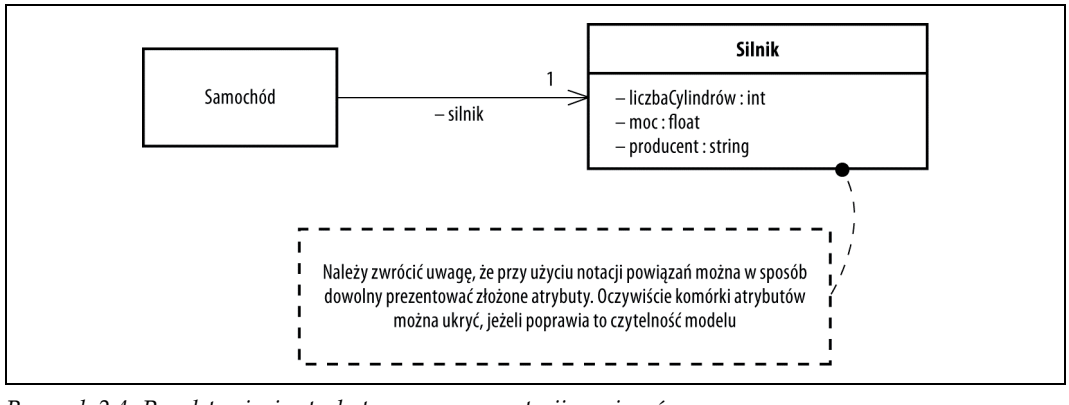
łańcuchy właściwości
: Zbiór właściwości lub znaczników, które można dołączyć do atrybutów. Są one zazwyczaj zależne od kontekstu i oznaczają takie cechy jak porządek lub unikalność. Występują pomiędzy nawiasami klamrowymi `{}`, a znakiem rozdzielającym jest przecinek. Patrz „Właściwości atrybutów".

ograniczenia
: Jedno lub więcej ograniczeń zdefiniowanych dla atrybutu. Mogą używać języka naturalnego lub jakiejś gramatyki formalnej, np. OCL. Patrz „Ograniczenia".

Atrybuty w postaci powiązań

Atrybuty można również prezentować za pomocą notacji powiązań. Jej zastosowanie wiąże się z powstawaniem większych diagramów, ale zapewnia większą szczegółowość w przypadku złożonych typów atrybutów. Notacja powiązań przekazuje również dokładne informacje na temat tego, jak atrybut zawarty jest wewnątrz klasy (więcej informacji na temat typów powiązań znajduje się w podrozdziale „Powiązania"). Modelując na przykład `Samochód`, można za pomocą *powiązań* o wiele jaśniej pokazać, że zawiera on `Silnik`, niż tylko wpisując ten silnik na listę atrybutów w prostokącie samochodu. Niemniej jednak prezentowanie nazwy samochodu za pomocą powiązań jest już prawdopodobnie lekką przesadą, gdyż jest ona tylko łańcuchem znaków.

W celu reprezentacji atrybutu za pomocą powiązań można użyć jednego z powiązań asocjacyjnych pomiędzy klasą zawierającą atrybut a reprezentującą go, co pokazano na rysunku 2.4. Widać, że powiązanie pomiędzy samochodem a jego silnikiem ma liczność 1 — samochód ma jeden silnik.

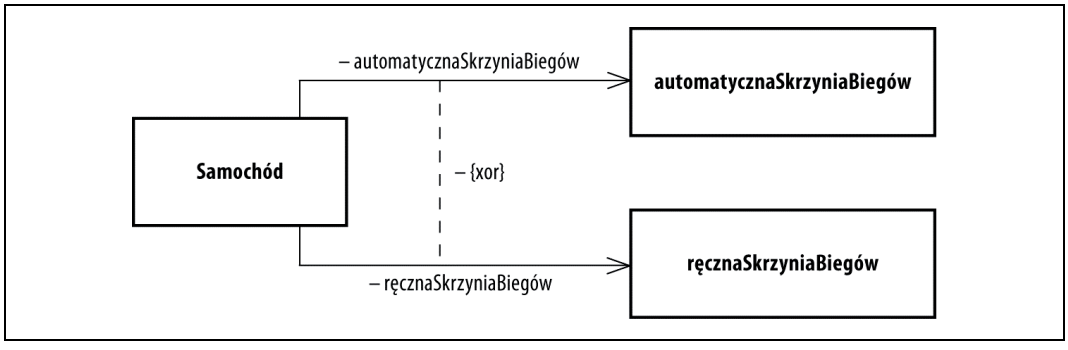


Rysunek 2.4. Przedstawienie atrybutu za pomocą notacji powiązań

Tak, tak — mój redaktor mówi, że niektóre samochody, jak na przykład Toyota Prius, mają dwa silniki.

Notacja powiązań posługuje się taką samą składnią jak notacja wewnętrzna, aczkolwiek układ jest nieco inny. Widoczność atrybutu i jego nazwa umieszczone są w pobliżu linii powiązania. Dla liczności nie należy stosować nawiasów kwadratowych — należy ją podawać w pobliżu klasyfikatora atrybutu.

Podobnie jak liczność, dla atrybutów można zdefiniować ograniczenia (patrz „Ograniczenia"). W notacji powiązań ograniczenia wpisuje się w pobliżu klasyfikatora atrybutu wzdłuż linii powiązania. Za pomocą notacji powiązań w UML-u można również wyrażać ograniczenia pomiędzy atrybutami, jak pokazano na rysunku 2.5.

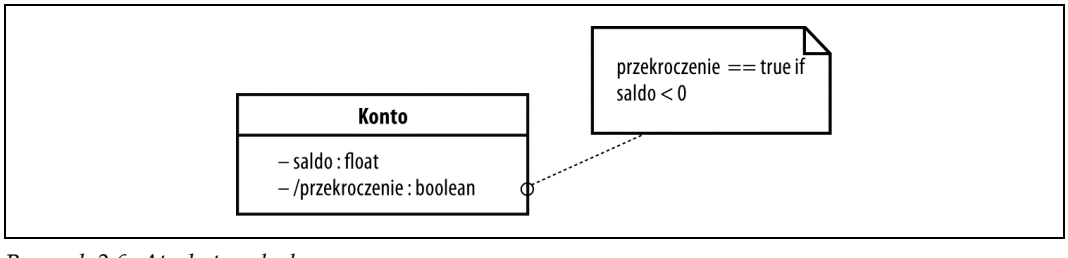


Rysunek 2.5. Notacja powiązań przy użyciu ograniczeń

Na rysunku 2.5 standardowe ograniczenie UML-a `xor` pokazuje, że w danym czasie może być ustawiona tylko `automatycznaSkrzyniaBiegów` albo `ręcznaSkrzyniaBiegów` (alternatywa wykluczająca — exclusive `or`). W przypadku zastosowania notacji wewnętrznej konieczne byłoby umieszczenie tego ograniczenia na notce.

Atrybuty pochodne

Notacja pochodna, którą oznacza symbol ukośnika (`/`), może być używana jako wskazówka dla implementatora, że dany atrybut nie jest konieczny. Przypuśćmy na przykład, że modelujemy rachunek bankowy za pomocą prostej klasy o nazwie `Rachunek`. Klasa ta przechowuje saldo bieżące jako liczbę zmiennopozycyjną o nazwie `saldo`. Aby śledzić informację, czy stan konta nie został przekroczony, dodajemy wartość logiczną (`boolean`) o nazwie `przekroczenie`. O tym, czy stan konta nie został przekroczony, informuje dodatnie saldo, a nie dodana przed chwilą wartość logiczna. Można o tym poinformować programistę, pokazując, że `przekroczenie` jest atrybutem pochodnym, którego stan opiera się na saldzie. Na rysunku 2.6 pokazano sposób prezentacji atrybutów `saldo` i `przekroczenie` z ukazaniem powiązania za pomocą notki.

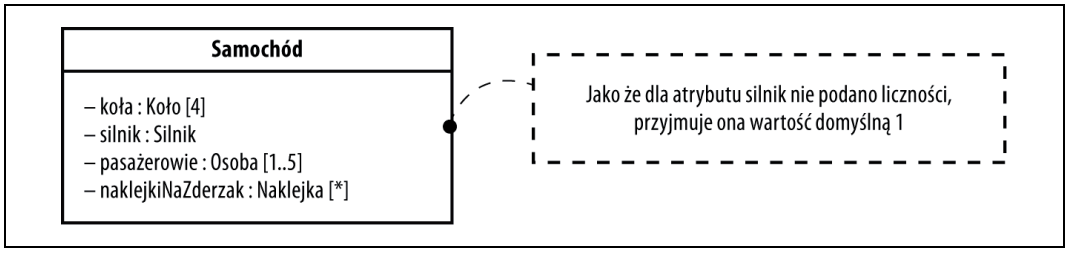


Rysunek 2.6. Atrybut pochodny

W specyfikacji UML-a zaznaczono, że atrybuty pochodne są tylko do odczytu (`readOnly`), co oznacza, że użytkownik nie może zmieniać ich wartości. Jeżeli jednak użytkownik ma pozwolenie na modyfikację ich wartości, to klasa powinna odpowiednio zaktualizować źródło informacji pochodnej.

Liczność atrybutu

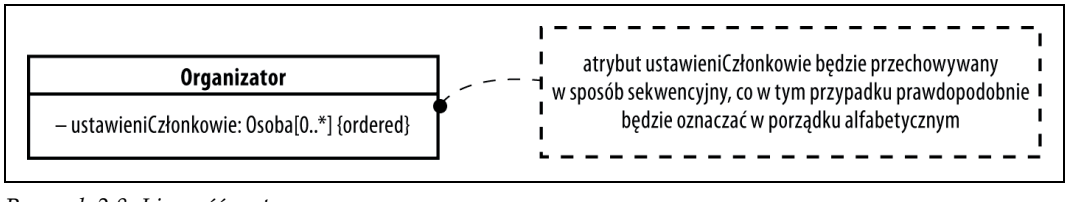
Liczność atrybutu określa liczbę egzemplarzy typu tego atrybutu utworzonych w procesie instancjonowania klasy. Na przykład nasz klasowy `Samochód` będzie najprawdopodobniej miał cztery koła, a więc liczność atrybutu `koła` będzie wynosiła 4. Jeżeli liczność nie została określona, to domyślnie wynosi 1. Liczność może być pojedynczą liczbą całkowitą, listą liczb całkowitych oddzielanych przecinkami lub zakresem wartości. W przypadku zakresu wartości nieskończoność dodatnia jest reprezentowana za pomocą symbolu `*`. Jeżeli dolna granica nie zostanie podana, symbol * oznacza zero lub więcej. Wartość liczności podawana jest pomiędzy nawiasami kwadratowymi w postaci pojedynczej liczby całkowitej lub za pomocą dwóch liczb całkowitych rozdzielonych dwiema kropkami (. .). Na rysunku 2.7 przedstawiono różne sposoby reprezentacji liczności atrybutu.



Rysunek 2.7. Przykłady liczności

Porządek

Atrybut, którego liczność przekracza 1, może być *sortowany* (ang. *ordered*). W takim przypadku elementy muszą być przechowywane w odpowiedniej kolejności. Można na przykład zdecydować się na przechowywanie listy nazwisk w kolejności alfabetycznej, nadając jej właściwość `ordered`. Co dokładnie oznacza przechowywanie elementów w odpowiedniej kolejności, zależy od rodzaju atrybutu. Domyślnie atrybuty nie są sortowane. Aby oznaczyć atrybut jako sortowany, należy po nim dodać właściwość `ordered`, jak widać na rysunku 2.8.

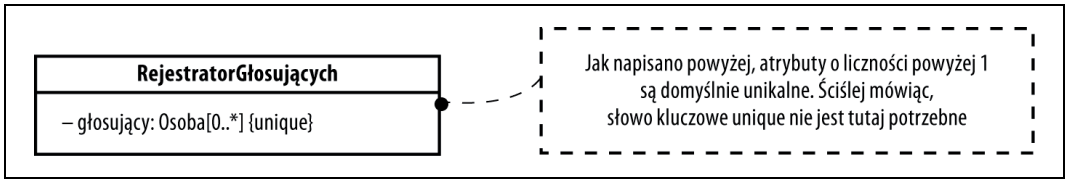


Rysunek 2.8. Liczność sortowana

Unikalność

Poza sortowaniem atrybuty o liczności powyżej 1 mogą być również *unikalne* (ang. *unique*). Jeżeli atrybut ma być unikalny, to wszystkie jego elementy również muszą takie być. Domyślnie wszystkie atrybuty o liczności powyżej 1 *są unikalne*. Oznacza to, że żaden z elementów atrybutu nie może się powtarzać. Na przykład jeżeli klasa przechowywałaby listę osób biorących udział w głosowaniu i każdy głosujący miałby tylko jeden głos, to każdy element takiej listy byłby unikalny. Aby uczynić atrybut unikalnym, należy umieścić po nim

w nawiasach klamrowych słowo kluczowe `unique`, jak pokazano na rysunku 2.9. Aby zezwolić na przechowywanie powtarzających się obiektów przez atrybut, należy użyć słowa kluczowego `not unique`.



Rysunek 2.9. Liczność unikalna

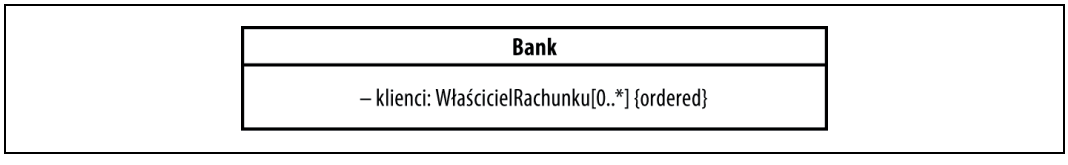
Typy kolekcji

Specyfikacja UML-a określa zestaw odwzorowań różnych właściwości sortowania i unikalności do typów kolekcji UML-a. W tabeli 2.1 przedstawiono odwzorowania właściwości atrybutów do typów kolekcji UML-a. Należy pamiętać, że typy kolekcji zaprezentowane w tabeli 2.1 są odwzorowaniami UML-a i nie mogą być przekładane bezpośrednio na klasy języka docelowego.

Tabela 2.1. Typy kolekcji atrybutów

Sortowanie	Unikalność	Typ skojarzonej kolekcji
False	False	Bag
True	True	OrderedSet
False	True	Set
True	False	Sequence

Na przykład aby pokazać, że klienci banku powinni zostać zaprezentowani za pomocą kolekcji `OrderedSet`, model atrybutu `klienci` mógłby wyglądać jak na rysunku 2.10.



Rysunek 2.10. Przykładowy atrybut przechowywany w kolekcji OrderedSet

Właściwości atrybutów

Poza właściwościami związanymi z licznością atrybut może mieć jeszcze kilka innych właściwości pozwalających przekazać czytelnikowi diagramu dodatkowe informacje. Najczęściej używane właściwości zdefiniowane w UML-u to:

`readOnly`
: Oznacza, że nie można modyfikować początkowo nadanej wartości atrybutu. W języku, w którym tworzony jest program, właściwość ta najczęściej znajduje odwzorowanie w postaci stałej. UML nie narzuca, kiedy należy podać wartość początkową. Aczkolwiek jeżeli dla atrybutu zostanie podana wartość domyślna, to jest ona traktowana jako początkowa i nie może być zmieniana.

`union`
: Oznacza, że typ atrybutu jest unią wartości dozwolonych dla tego atrybutu. Właściwość ta jest często używana w połączeniu z właściwością `derived` w celu zaznaczenia, że atrybut jest unią pochodną od innego zbioru atrybutów.

`subsets <nazwa-atrybutu>`
: Oznacza, że ten typ atrybutu jest podzbiorem wszystkich prawidłowych wartości tego atrybutu. Właściwość ta nie jest zbyt często używana, ale jeżeli już zostanie użyta, to zazwyczaj jest związana z podklasami typu atrybutu.

`redefines <nazwa-atrybutu>`
: Oznacza, że ten atrybut stanowi alias danego atrybutu. Właściwość ta nie jest często używana, ale może mieć zastosowanie do pokazania, że jakaś podklasa ma atrybut będący aliasem atrybutu klasy nadrzędnej.

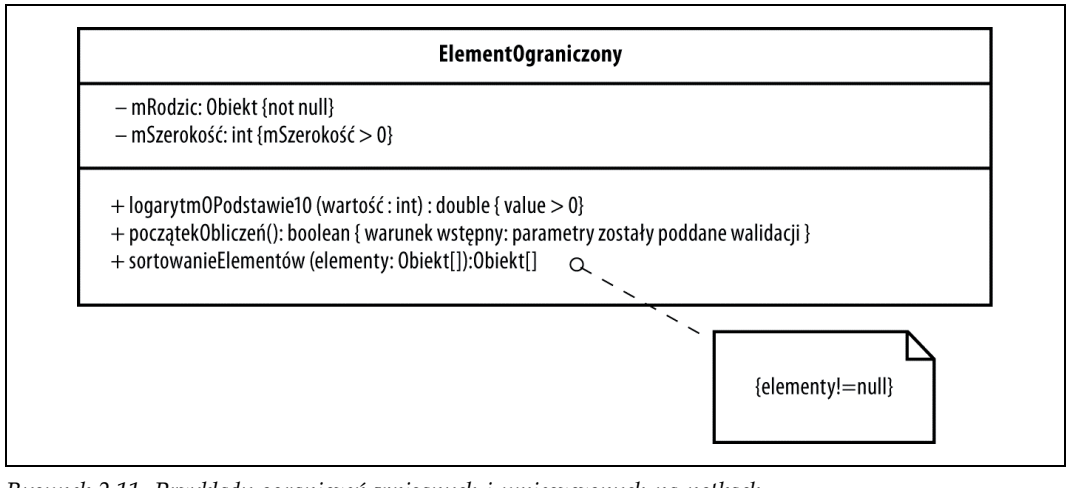
`composite`
: Oznacza, że ten atrybut jest częścią związku całość-część z klasyfikatorem. Więcej informacji o kompozycjach znajduje się w podrozdziale „Powiązania”.

Ograniczenia

Ograniczenia stanowią pewne ściślejsze zasady dotyczące elementów. Mogą być pisane w języku naturalnym lub przy użyciu gramatyk formalnych, np. OCL. Muszą mieć jednak wartość logiczną. Zazwyczaj ograniczenia umieszcza się w nawiasach klamrowych po elemencie, do którego się odnoszą, ale można je także umieszczać w notkach dołączanych do niego za pomocą linii przerywanej.

Nazwę ograniczenia podaje się po dwukropku (:) przed wyrażeniem logicznym. Jest to częsty sposób identyfikacji ograniczeń operacji (patrz „Ograniczenia operacji”).

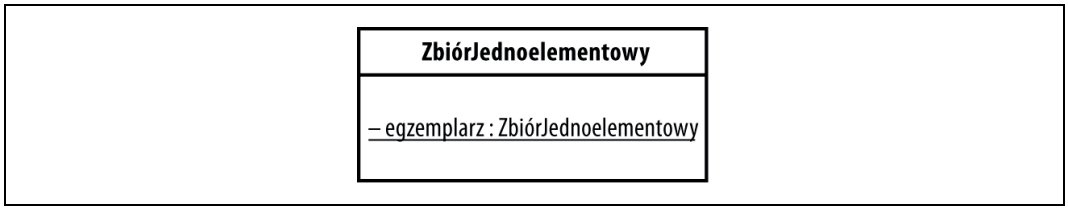
Na rysunku 2.11 przedstawiono kilka ograniczeń dla atrybutów i operacji.



Rysunek 2.11. Przykłady ograniczeń wpisanych i umieszczonych na notkach

Atrybuty statyczne

Atrybuty statyczne są atrybutami klas, a nie ich egzemplarzy. Można na przykład zainicjalizować stałe wartości dla klasy, a następnie udostępnić je wszystkim jej egzemplarzom. Atrybuty statyczne reprezentowane są za pomocą podkreślenia ich specyfikacji zarówno w notacji wewnętrznej, jak i w prezentacjach opartych na powiązaniach. Pokazano to na rysunku 2.12.



Rysunek 2.12. Atrybut statyczny

Operacje

Operacje to właściwości klas, które określają, w jaki sposób wywołać określone zachowanie. Na przykład klasa może zawierać operację rysującą na ekranie prostokąt lub sprawdzającą liczbę elementów wybranych z listy. W UML-u istnieje wyraźne rozróżnienie pomiędzy określeniem, jak wywołać zachowanie (operacja), a rzeczywistą implementacją tego zachowania (metoda). W celu uzyskania większej ilości informacji patrz podrozdział „Metody".

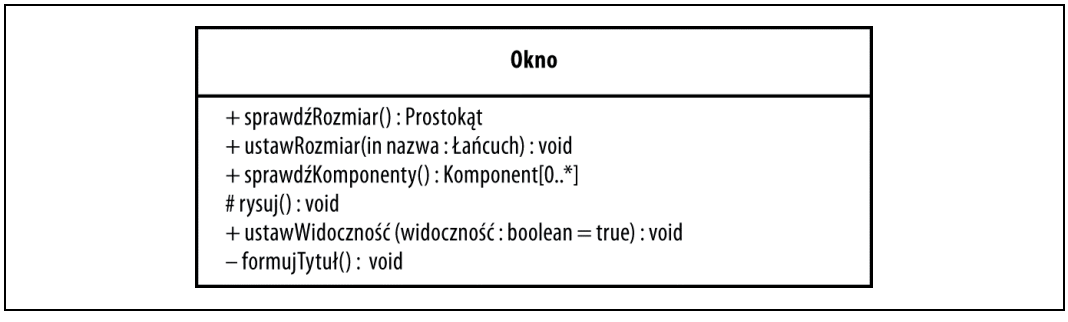
Operacje umieszcza się w oddzielnych komórkach przy użyciu następującej składni:

```
widoczność nazwa ( parametry ) : typ-zwracany {właściwości}
```

gdzie parametry zapisuje się następująco:

```
kierunek nazwa_parametru : typ [ liczność ]
  = wartość_domyślna { właściwości }
```

Na rysunku 2.13 przedstawiono kilka przykładowych operacji klasy.



Rysunek 2.13. Przykładowe operacje klasy

Elementy składni są następujące:

widoczność
: Określa widoczność operacji. Do wyboru są następujące symbole: `+`, `-`, # i ~, oznaczają one odpowiednio: `public`, `private`, `protected` i `package` (patrz „Widoczność" w rozdziale 3.).

nazwa
: Krótkie wyrażenie stanowiące nazwę operacji. Operacje są zazwyczaj wyrażeniami czasownikowymi reprezentującymi czynności, które klasyfikator powinien wykonać w imieniu wywołującego. W specyfikacji UML-a zaleca się, aby pierwsza litera pierwszego członu była mała, a pierwsze litery wszystkich następnych członów wielkie. Przykład pokazano na rysunku 2.13.

typ-zwracany
: Określa rodzaj informacji zwracanej przez operację (jeżeli operacja coś zwraca). Jeżeli operacja nie zwraca żadnych danych (w niektórych językach programowania zwana jest w takich przypadkach **podprogramem**), to typem zwracanym powinien być `void`. Jeżeli jednak operacja zwraca jakąś wartość (w niektórych językach programowania zwana jest wtedy **funkcją**), to należy pokazać typ tej wartości, np. inny klasyfikator, typ podstawowy lub kolekcja. Specyfikacja UML-a stanowi, że podawanie typu zwracanego jest opcjonalne. Jeżeli typ zwracany nie zostanie podany, nie można go odgadnąć, a nawet nie wiadomo, czy istnieje.

właściwości
: Określa ograniczenia i właściwości operacji. Pole to jest opcjonalne. W przypadku nieużywania właściwości nie wpisuje się nawiasów klamrowych. Więcej informacji na ten temat można znaleźć w podrozdziale „Ograniczenia operacji".

Elementy składni parametru są następujące:

kierunek
: Opcjonalny element składni, który informuje, w jaki sposób parametr jest używany przez operację. Dostępne wartości to: `in`, `inout`, `out` oraz `return`. `in` oznacza, że parametr jest przekazywany do operacji przez wywołującego; `inout`, że parametr jest przekazywany przez wywołującego, a następnie prawdopodobnie modyfikowany przez operację i wysyłany z powrotem; `out`, że parametr nie jest ustawiany przez wywołującego, ale przez operację i jest przekazywany z powrotem na zewnątrz; `return` oznacza natomiast, że wartość ustawiona przez wywołującego jest przekazywana z powrotem na zewnątrz jako wartość zwracana.

nazwa_parametru
: Jest rzeczownikiem lub wyrażeniem rzeczownikowym stanowiącym nazwę parametru. Zazwyczaj nazwa parametru rozpoczyna się z małej litery, a następne składające się na nią wyrazy zaczynają się od wielkich liter.

typ
: Typ parametru. Zazwyczaj jest to inna klasa, interfejs, kolekcja lub typ podstawowy.

liczność
: Określa liczbę obecnych egzemplarzy typu parametru. Może nie zostać podana (wtedy oznacza 1), może być liczbą całkowitą, zakresem wartości rozdzielonych przecinkami lub zakresem wartości podanym w nawiasach kwadratowych i rozdzielonych symbolem `..`. Symbol `*` służy do oznaczania górnego limitu. Jeżeli użyty jest samodzielnie, oznacza zero lub więcej. Patrz „Liczność".

wartość_domyślna
: Określa wartość domyślną tego parametru. Wartość domyślna jest opcjonalna. Jeżeli nie zostanie podana, to nie należy wstawiać znaku równości. Należy zwrócić uwagę, że UML nie określa, czy parametr z wartością domyślną może zostać pominięty podczas wywoływania operacji (np. implementacja wartości domyślnych parametrów w C++). Rzeczywista składnia wywoływania operacji jest zależna od konkretnego języka programowania.

właściwości
: Określa wszelkie właściwości odnoszące się do parametru. Element ten podawany jest w nawiasach klamrowych. Właściwości definiuje się zazwyczaj w kontekście określonego modelu, ale jest kilka wyjątków: `ordered`, `readOnly` oraz `unique`. W celu zdobycia większej ilości informacji zajrzyj do podrozdziałów „Liczność" i „Właściwości atrybutów". Właściwości są opcjonalne. Jeżeli nie ma żadnych, to nie należy wstawiać nawiasów klamrowych.

Ograniczenia operacji

Z operacją może być skojarzonych kilka ograniczeń, które pomagają zdefiniować sposób jej interakcji z resztą systemu. Razem wzięte ograniczenia operacji stanowią kontrakt, który musi być przestrzegany w implementacji operacji.

Ograniczenia operacji wykorzystują zwykłą notację i umieszczane są albo bezpośrednio po sygnaturze operacji, albo w notce dołączonej za pomocą linii przerywanej. Więcej informacji na ten temat można znaleźć w podrozdziale „Ograniczenia".

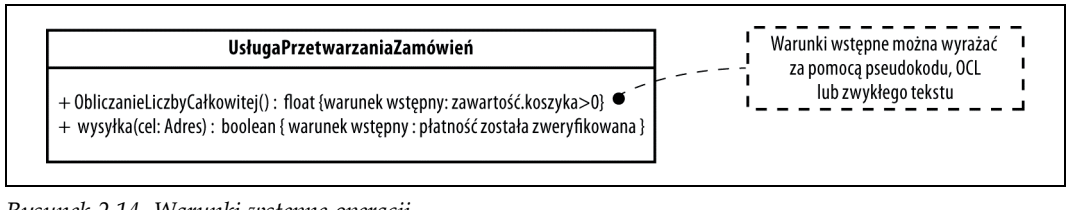
Warunki wstępne

Warunki wstępne (ang. *preconditions*) precyzują to, w jakim stanie powinien znajdować się system przed wywołaniem operacji. Z praktycznego punktu widzenia nie da się wyrazić stanu całego systemu. W związku z tym warunki wstępne zazwyczaj określają prawidłowe wartości parametrów, stan klasy zawierającej operację lub kilka atrybutów kluczowych dla systemu.

W specyfikacji jasno napisano, że operacja nie musi sprawdzać warunków wstępnych w jej ciele przed wykonaniem. Teoretycznie operacja nie zostanie nawet wywołana, jeżeli warunki wstępne nie zostaną spełnione. W praktyce niewiele języków programowania umożliwia taką ochronę. Jeżeli ktoś poświęcił czas na zdefiniowanie warunków wstępnych, to najczęściej w interesie programisty jest sprawdzenie, czy podczas implementacji operacji są one prawidłowe.

Warunki wstępne dają deweloperowi jedną z niewielu okazji do chronienia własnego stanowiska i dokładnego wyrażenia, jak powinno wszystko wyglądać po wywołaniu implementacji. Należy je stosować.

Na rysunku 2.14 przedstawiono kilka przykładowych warunków wstępnych.

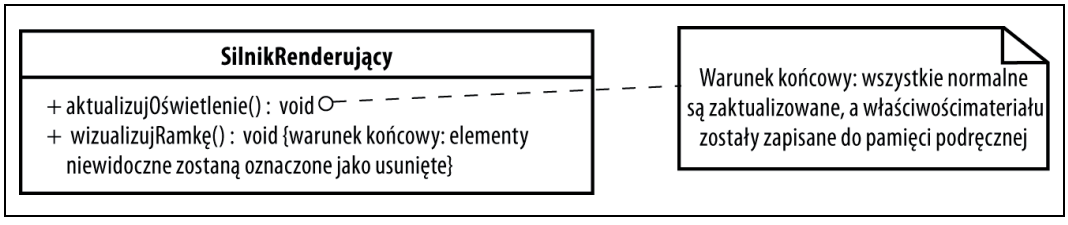


Rysunek 2.14. Warunki wstępne operacji

Warunki końcowe

Warunki końcowe określają gwarancje na temat stanu systemu po wykonaniu operacji. Podobnie jak warunki wstępne, warunki końcowe zazwyczaj wyrażają stan jednego lub większej liczby kluczowych atrybutów systemu albo pewną gwarancję na temat stanu klasy zawierającej operację.

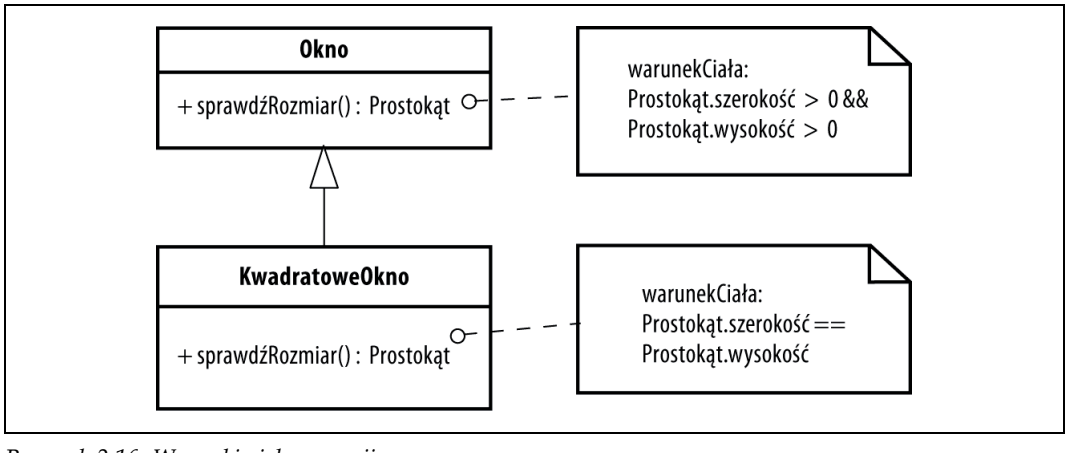
Rysunek 2.15 przedstawia przykładowe warunki końcowe dołączone do operacji.



Rysunek 2.15. Warunki końcowe dołączone do operacji

Warunki ciała operacji

Operacja może mieć `warunekCiała` kładący ograniczenia na typ zwracany. Jest on oddzielony od warunków końcowych, ponieważ może zostać zastąpiony metodami podklas klasy nadrzędnej (więcej informacji na temat podklas znajduje się w podrozdziale „Generalizacje"). Na przykład klasa o nazwie `Okno` może definiować warunek ciała operacji dla metody o nazwie `sprawdźRozmiar()`, który będzie pilnował, aby długość i szerokość okna nie miały wartości zerowych. Podklasa o nazwie `KwadratoweOkno` może mieć własny warunek ciała operacji określający, że szerokość musi być równa długości. `warunekCiała` jest podobny do warunku wejściowego i końcowego w tym, że można go wyrazić zarówno w języku naturalnym, jak i w OCL-u (patrz „Ograniczenia"). Rysunek 2.16 przedstawia przykładowy `warunekCiała` operacji.

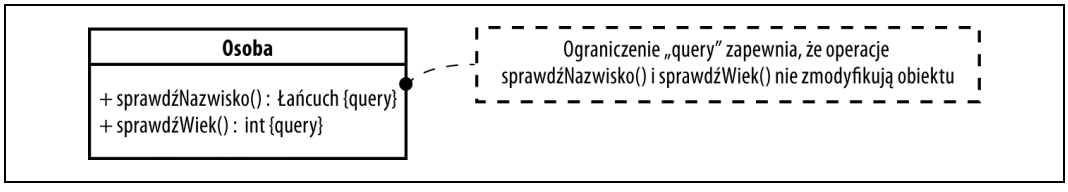


Rysunek 2.16. Warunki ciała operacji

Operacje odpytujące

Operację można zadeklarować jako *odpytującą*, jeżeli jej implementacja nie powoduje żadnych modyfikacji klasy nadrzędnej. W praktyce twórcy modeli często używają właściwości odpytywania do oznaczania metod, które nie modyfikują żadnego znaczącego atrybutu obiektu. Mogą na przykład istnieć wewnętrzne atrybuty pamięci podręcznej, które podlegają uaktualnieniom w wyniku wykonania jakiegoś zapytania. Ważne jest, że stan systemu, z zewnętrznej perspektywy, nie ulega zmianie w wyniku wykonania metody zapytania. Wywołanie tej metody nie może mieć żadnych skutków ubocznych.

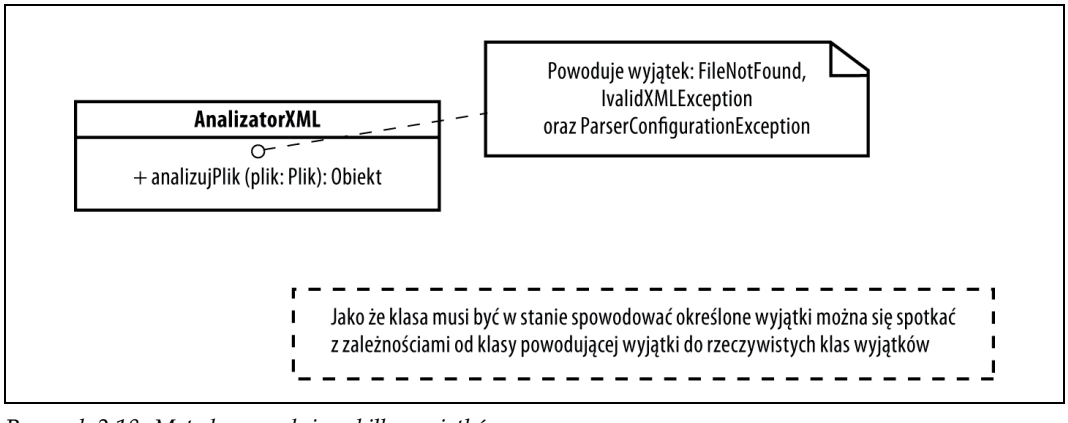
Metodę zapytania sygnalizuje się poprzez umieszczenie ograniczenia zapytania po sygnaturze operacji. Na przykład operacja o nazwie `sprawdźWiek()` zwracająca tylko liczbę całkowitą, a niezmieniająca żadnych wewnętrznych wartości zawierającej ją klasy może być uznana za operację odpytującą. W języku C++ operacje tego typu znajdują odwzorowanie w postaci metod typu `const`. Na rysunku 2.17 przedstawiono kilka metod odpytujących klasy.



Rysunek 2.17. Przykładowe operacje odpytujące

Wyjątki

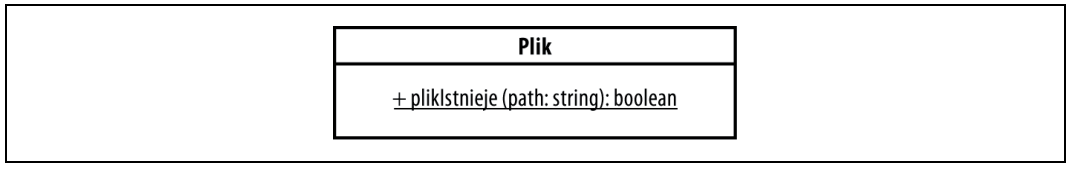
Mimo że wyjątki powodowane przez operacje z technicznego punktu widzenia nie są tym samym co ograniczenia, można je definiować przy użyciu takiej samej notacji. Przeważnie wyjątki są innymi klasami (często zaklasyfikowane są za pomocą słowa kluczowego `«exception»`, ale to tylko ze względu na konwencję) powodowanymi przez operację w przypadku wystąpienia błędu. Listę spowodowanych wyjątków można umieścić w notce dołączonej do operacji za pomocą linii przerywanej. Na rysunku 2.18 pokazano przykładową operację powodującą kilka wyjątków.



Rysunek 2.18. Metoda powodująca kilka wyjątków

Operacje statyczne

Operacje z reguły określają zachowanie *egzemplarza* klasy. Chociaż UML pozwala także na określanie zachowania *samych klas* za pomocą operacji. Operacje takie noszą nazwę *operacji statycznych* i wywoływane są bezpośrednio dla klas, a nie dla ich egzemplarzy. Operacje statyczne są często używane jako operacje użytkowe, które nie muszą wykorzystywać atrybutów klasy nadrzędnej. W UML-u nie ma formalnego opisu notacji dla tych operacji, aczkolwiek na ogół są one reprezentowane przy użyciu tych samych konwencji co atrybuty statyczne. Aby oznaczyć operację jako statyczną, należy ją podkreślić. Rysunek 2.19 przedstawia przykładową operację statyczną.



Rysunek 2.19. Klasa z operacją statyczną

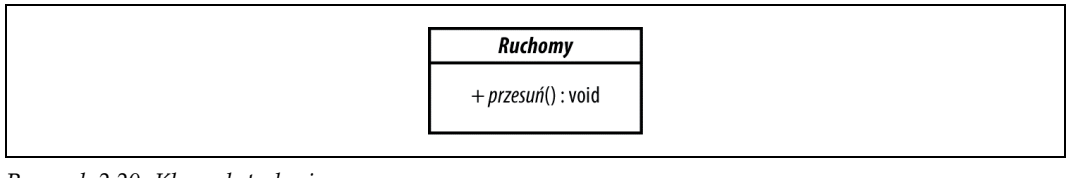
Metody

Metoda jest implementacją operacji. Z reguły każda klasa dostarcza implementacji swoich operacji lub dziedziczy je od swojej klasy nadrzędnej (patrz „Generalizacje"). Jeżeli klasa nie dostarcza implementacji jakiejś operacji i brakuje jej też w klasie nadrzędnej, to operacja jest *abstrakcyjna* (patrz „Klasy abstrakcyjne"). Jako że metody są implementacjami operacji, nie mają własnej notacji. Ilustrują one operacje na klasie.

Klasy abstrakcyjne

Klasa abstrakcyjna to z reguły klasa, która dostarcza sygnatury operacji, ale nie jej implementację. Można także utworzyć klasę abstrakcyjną nieposiadającą żadnych operacji. Klasy abstrakcyjne są przydatne do identyfikacji wspólnej funkcjonalności obiektów kilku typów. Można na przykład utworzyć klasę abstrakcyjną o nazwie `Ruchomy`. `Ruchomy` obiekt to taki, który ma jakieś bieżące położenie i może być przesuwany za pomocą operacji o nazwie `przesuń()`. Klasa ta może mieć kilka specjalizacji, np.: `Samochód`, `KonikPolny` i `Osoba`, z których każda ma własną implementację operacji `przesuń()`. Jako że klasa bazowa `Ruchomy` nie ma implementacji operacji `przesuń()`, jest klasą abstrakcyjną.

Klasy abstrakcyjne oznacza się, pisząc ich nazwy kursywą (tak samo jak operacje abstrakcyjne). Na rysunku 2.20 przedstawiono przykład klasy abstrakcyjnej `Ruchomy`.



Rysunek 2.20. Klasa abstrakcyjna

Klasa abstrakcyjna nie może tworzyć obiektów. Aby było to możliwe, najpierw należy utworzyć jej podklasę implementującą operację i dopiero od tej podklasy utworzyć obiekty. Więcej na temat podklas można znaleźć w podrozdziale „Powiązania".

Powiązania

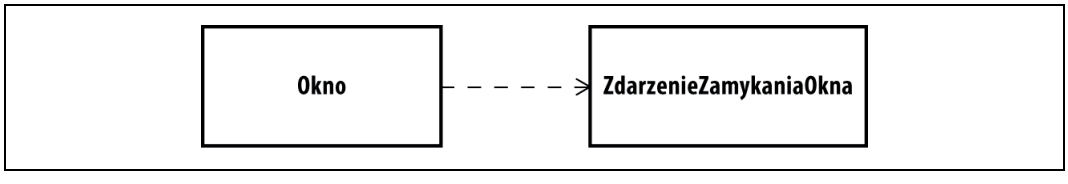
Same klasy nie dostarczyłyby zbyt wielu informacji na temat projektu systemu. W UML-u dostępnych jest kilka sposobów reprezentacji powiązań pomiędzy klasami. Każde powiązanie UML reprezentuje inny typ połączenia pomiędzy klasami i ma subtelne właściwości, które nie zostały w pełni ujęte w specyfikacji UML-a. Tworząc modele w świecie rzeczywistym, należy upewnić się, że informacje przekazywane za pomocą powiązań są zrozumiałe dla odbiorcy. Jest to zarówno ostrzeżenie dla tworzących modele, jak i nasze wyjaśnienie, że to, co piszemy poniżej, stanowi naszą własną interpretację specyfikacji UML-a. Na przykład dyskusja na temat tego, kiedy używać agregacji, a kiedy kompozycji, cały czas się toczy. Aby pomóc wybrać najlepsze powiązanie, podajemy krótką frazę dla każdego typu, która usprawnia dokonywanie rozróżnienia. Najważniejsze jest utrzymanie spójności modelu.

Zależności

Najsłabszym rodzajem powiązań między klasami są *zależności*. Zależność pomiędzy klasami oznacza, że jedna klasa używa drugiej klasy lub ma dotyczące niej informacje. Związek taki jest zazwyczaj krótkotrwały. Oznacza to, że klasa zależna wchodzi w krótką interakcję z klasą docelową, ale z reguły nie utrzymuje z nią związku przez realną ilość czasu.

Zależności zazwyczaj czytamy jako „…używa…". Na przykład jeżeli mielibyśmy klasę `Okno` wysyłającą klasę o nazwie `ZamykanieOkna` w momencie, gdy ma ono zostać zamknięte, powiedzielibyśmy: `Okno` używa `ZamykanieOkna`.

Zależności pomiędzy klasami oznacza się linią przerywaną zakończoną strzałką wskazującą od klasy zależnej do używanej. Na rysunku 2.21 pokazano zależność pomiędzy klasą o nazwie `Okno` a tą o nazwie `ZamykanieOkna`.



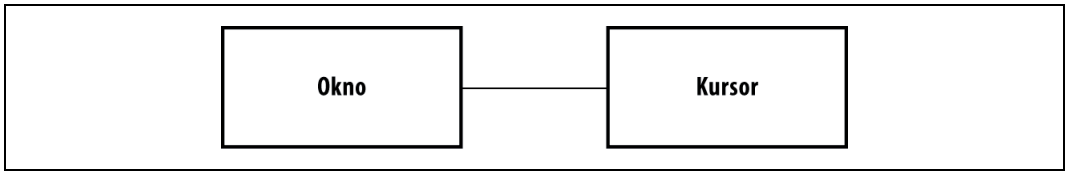
Rysunek 2.21. Zależność klasy Okno od klasy ZamykanieOkna

Przyglądając się powyższemu rysunkowi, można wyciągnąć wniosek, że klasa `Okno` nie pozostaje w dłuższym związku z klasą `ZamykanieOkna`. Wykorzystuje ją tylko wtedy, gdy jest jej potrzebna, a następnie o niej „zapomina".

Asocjacje

Asocjacje są silniejszym typem powiązań niż zależności i zazwyczaj oznaczają dłuższy okres trwania związku pomiędzy dwiema klasami. Czasy życia obiektów związanych asocjacją nie są ze sobą powiązane (co oznacza, że jeden z nich może zostać zniszczony bez potrzeby niszczenia drugiego).

Asocjacje zazwyczaj czyta się następująco: „…ma…”. Na przykład gdybyśmy mieli klasę o nazwie `Okno`, która miałaby odniesienie do bieżącego kursora myszy, to moglibyśmy powiedzieć `Okno` ma `Kursor`. Należy zauważyć różnicę pomiędzy wyrażeniami „…ma…” a „…posiada…” (patrz „Agregacje”). W tym przypadku klasa `Okno` nie jest właścicielem klasy `Kursor`, ponieważ jest ona współdzielona przez wszystkie aplikacje w systemie. Ma jednak do niej odniesienie, dzięki czemu może ją ukrywać, modyfikować, kształtować itd. Asocjacje oznacza się jednolitą linią umieszczoną pomiędzy klasami. Na rysunku 2.22 pokazano asocjację pomiędzy klasami `Okno` i `Kursor`.

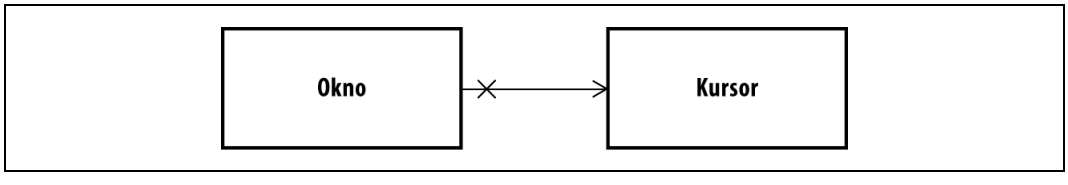


Rysunek 2.22. Asocjacja ilustrująca wyrażenie Okno ma Kursor

Komunikacyjność

Asocjacje mają specjalną notację służącą do obrazowania możliwości nawigacji. Jeżeli jedna klasa może komunikować się z drugą, to oznacza się to strzałką wskazującą w stronę klasy, z którą można się skomunikować. Jeżeli komunikacja może zachodzić w obie strony, powszechnie przyjęto, że nie rysuje się żadnych strzałek. Specyfikacja UML-a mówi jednak, że jeżeli wszystkie strzałki zostaną pominięte, to nie będzie się dało odróżnić asocjacji komunikacyjnych od niekomunikacyjnych. Jednak asocjacje niekomunikacyjne są niezmiernie rzadko stosowane, a więc sytuacja taka jest bardzo mało prawdopodobna.

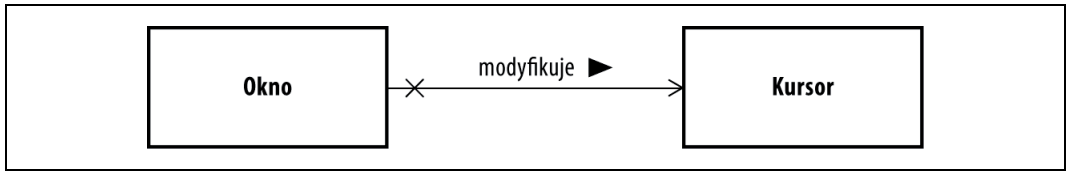
Można zabronić jednej klasie komunikowania się z inną. W tym celu należy umieścić symbol `X` na linii asocjacyjnej przy końcu należącym do klasy, z którą nie można się komunikować. Na rysunku 2.23 pokazano asocjację pomiędzy klasą o nazwie `Okno` a tą o nazwie `Kursor`. Ze względu na fakt, że egzemplarze klasy `Kursor` nie mogą komunikować się z egzemplarzami klasy `Okno`, została zastosowana strzałka z jednej strony i symbol `X` z drugiej.



Rysunek 2.23. Asocjacja pomiędzy klasami Okno i Kursor zabraniająca nawigacji z klasy Okno do Kursor

Nazywanie asocjacji

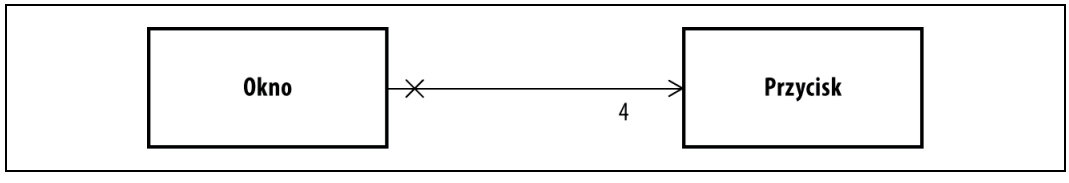
Asocjacje można ozdobić za pomocą kilku symboli dodających pewne informacje do modelu. Najprostszym symbolem jest jednolity grot strzałki wskazujący kierunek, w którym odczytujący powinien podążać, czytając asocjację. Do grota często dodawane są krótkie wyrażenia dostarczające pewnych informacji kontekstowych dotyczących asocjacji. Wyrażenia te rzadko znajdują odwzorowanie w kodzie źródłowym. Służą one wyłącznie do celów modelowania. Na rysunku 2.24 przedstawiono jednolity grot strzałki zastosowany w asocjacji `Okno`-`Kursor`.



Rysunek 2.24. Wskazuje kierunek czytania asocjacji pomiędzy klasami Okno i Kursor

Liczność

Jako że asocjacje zazwyczaj reprezentują trwałe związki, często są używane do wskazywania atrybutów klas. Jak pamiętamy z podrozdziału „Atrybuty powiązań", można pokazać, ile egzemplarzy określonej klasy jest zaangażowanych w dany związek. W przypadku niepodania żadnej wartości, zakłada się, że jest to 1. Aby zastosować inną wartość, wystarczy w pobliżu posiadanej klasy umieścić specyfikację liczności (informacje na temat dozwolonych typów liczności znajdują się w podrozdziale „Liczność atrybutu"). Należy zwrócić uwagę, że w przypadku określania liczności w asocjacjach wartości nie umieszcza się w nawiasach kwadratowych. Rysunek 2.25 prezentuje asocjację z określoną licznością.



Rysunek 2.25. Prosta asocjacja pokazująca 4 przyciski w oknie

Właściwości związane z licznością mogą być stosowane również do asocjacji. Nazwy i definicje dozwolonych właściwości można znaleźć w podrozdziale „Właściwości atrybutów".

Agregacje

Agregacja jest silniejszą wersją asocjacji. Jednak w przeciwieństwie do asocjacji z reguły zakłada posiadanie i może implikować związek pomiędzy czasami życia obiektów. Agregacje odczytuje się następująco: „…posiada…". Na przykład jeżeli klasa o nazwie `Okno` przechowuje dane dotyczące swojego położenia i rozmiaru w klasie `Prostokąt`, to można powiedzieć: `Okno` posiada `Prostokąt`. Prostokąt ten może być współdzielony przez inne klasy, ale `Okno` pozostaje z nim w bardzo ścisłym związku. Różnica w porównaniu z podstawową asocjacją jest bardzo subtelna — agregacja ma silniejszą konotację. Nie jest to jednakże najsilniejszy rodzaj powiązania międzyklasowego. Jeżeli związek polega na tym, że jedna klasa jest częścią drugiej, to nazywa się go *kompozycją*.

Agregację oznacza się za pomocą figury w kształcie diamentu, umieszczonej przy klasie posiadającej i linii ciągłej wskazującej klasę posiadaną. Na rysunku 2.26 przedstawiono agregację pomiędzy klasą o nazwie `Okno` a klasą o nazwie `Prostokąt`.

Jako że jest to związek asocjacyjny, na linii agregacyjnej można umieszczać informacje dotyczące komunikacyjności i liczności. Patrz podrozdział „Asocjacje".

Rysunek 2.26. Okno posiada Prostokąt

Kompozycje

Kompozycje reprezentują bardzo silny rodzaj powiązań pomiędzy klasami — jedna klasa zawiera drugą. Kompozycje mają zastosowanie w wyrażaniu związków typu *całość-część*. „Część" może być zaangażowana tylko w jeden związek tego typu w danym czasie. Czasy życia instancji biorących udział w takim związku są prawie zawsze ze sobą powiązane. Jeżeli większa z nich, zawierająca mniejszą, zostanie zniszczona, to prawie zawsze pociąga to za sobą zniszczenie tej mniejszej. UML daje możliwość przypisania części do innego właściciela przed zniszczeniem, co pozwala jej nadal istnieć, ale jest to raczej sytuacja wyjątkowa, a nie reguła.

Kompozycje z reguły czyta się następująco: „…jest częścią…". Oznacza to, że odczytywanie kompozycji należy zaczynać od części do całości. Na przykład jeżeli dojdziemy do wniosku, że okno w naszym systemie musi mieć pasek tytułu, to może go reprezentować klasa o nazwie `PasekTytułu`, która jest częścią klasy o nazwie `Okno`.

Kompozycje oznacza się, stawiając wypełnioną figurę diamentu obok klasy zawierającej oraz za pomocą linii ciągłej wskazującej klasę, którą ta klasa posiada. Na rysunku 2.27 przedstawiono związek kompozycyjny pomiędzy klasą o nazwie `Okno` a tą o nazwie `PasekTytułu`.

Rysunek 2.27. PasekTytułu jest częścią okna

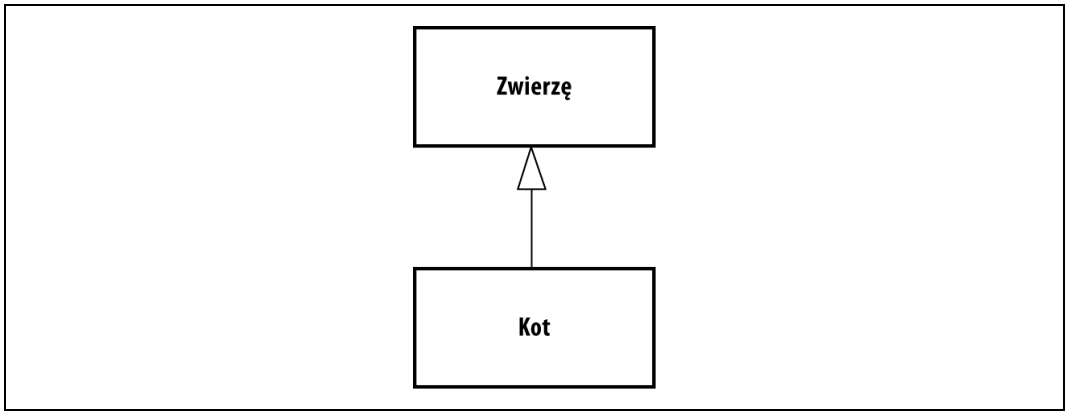
Jako że jest to związek asocjacyjny, na linii kompozycyjnej można umieszczać informacje dotyczące komunikacyjności i liczności. Patrz podrozdział „Asocjacje".

Generalizacje

Generalizacja implikuje, że cel związku to bardziej ogólna (lub mniej konkretna) wersja klasy źródłowej lub interfejsu. Generalizacje są często używane do wyciągania wspólnych cech różnych klasyfikatorów. Na przykład jeżeli mielibyśmy klasę o nazwie `Kot` i klasę o nazwie `Pies`, to moglibyśmy utworzyć ich generalizację o nazwie `Zwierzę`. Pełna analiza problemu, jak i kiedy stosować generalizacje (w szczególności w porównaniu z realizacją interfejsu), mogłaby być tematem książki o zorientowanej obiektowo analizie i projektowaniu obiektowym i nie będziemy się tym zajmować.

Generalizacje zazwyczaj odczytuje się następująco: „…to…". Czytanie należy zacząć od klasy o węższym znaczeniu. Wracając do przykładu z psem i kotem, moglibyśmy powiedzieć: `Kot` to `Zwierzę`.

Generalizacje oznacza się za pomocą linii ciągłej zakończonej pustą strzałką wskazującą w kierunku klasy bardziej ogólnej. Na rysunku 2.28 przedstawiono powiązanie `Kot`-`Zwierzę`.

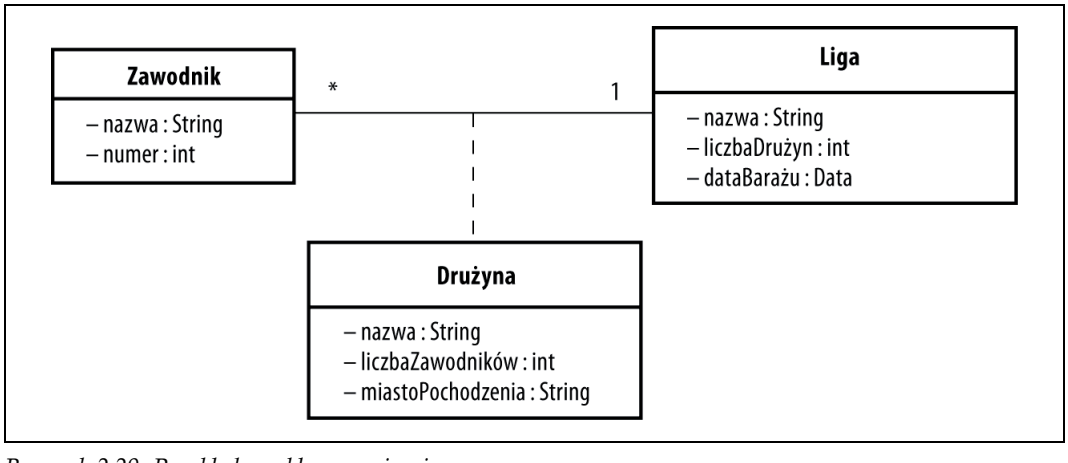


Rysunek 2.28. Kot jest bardziej konkretną wersją klasy Zwierzę

W przeciwieństwie do asocjacji generalizacje z reguły nie mają nazw ani żadnego rodzaju liczności. UML pozwala na wielodziedziczenie, co oznacza, że klasa może mieć więcej generalizacji niż jedną, z których każda reprezentuje pewien aspekt klasy potomnej. Jednakże niektóre nowoczesne języki programowania (np. Java i C#) nie obsługują wielodziedziczenia, oferując w zamian interfejsy i ich realizacje.

Klasy asocjacyjne

Często związek pomiędzy dwiema klasami nie jest prostą strukturą. Na przykład zawodnik piłkarski może być powiązany z ligą poprzez fakt bycia członkiem drużyny. Jeżeli związki pomiędzy dwoma elementami są bardzo złożone, to można je zaprezentować za pomocą tzw. *klas asocjacyjnych*. Klasa asocjacyjna ma nazwę i atrybuty jak normalna klasa. Prezentuje się ją w taki sam sposób jak inne klasy, łącząc ją z reprezentowaną przez nią asocjacją za pomocą linii przerywanej. Rysunek 2.29 przedstawia powiązania zawodnika piłkarskiego z ligą.

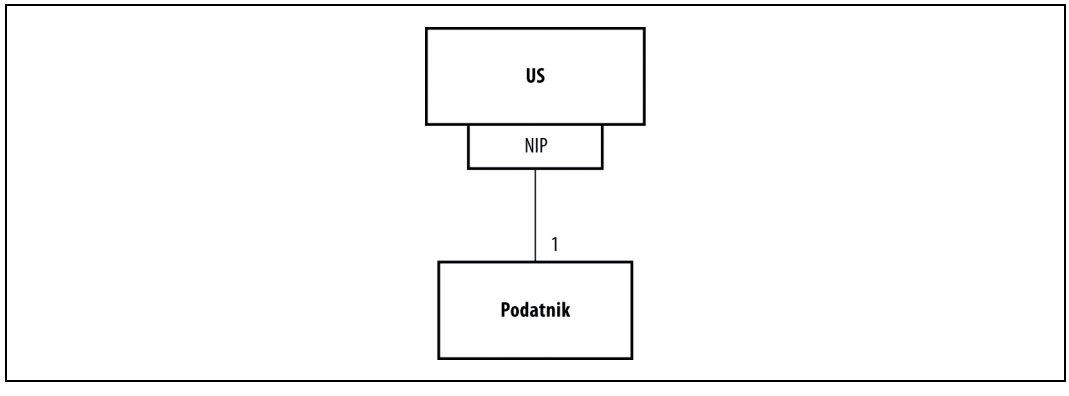


Rysunek 2.29. Przykładowa klasa asocjacyjna

W kodzie źródłowym powiązania z klasą asocjacyjną zazwyczaj dają trzy klasy: po jednej dla każdej strony asocjacji oraz jedną dla samej asocjacji. Pomiędzy przeciwnymi stronami asocjacji może być bezpośrednie połączenie, ale nie musi. Implementacja może wymagać, aby w celu dotarcia do klasy znajdującej się po przeciwnej stronie trzeba było przejść przez klasę asocjacyjną. Mówiąc inaczej, `Zawodnik` może nie mieć bezpośredniego połączenia z ligą, ale z drużyną, która z kolei ma połączenie z ligą. Konstrukcja powiązań zależna jest od konkretnej implementacji. Nie zmienia się tylko podstawowe pojęcie klasy asocjacyjnej.

Kwalifikatory asocjacyjne

Powiązania pomiędzy elementami często opatrzone są różnymi słowami kluczowymi lub innymi wartościami. Na przykład stały klient banku może być rozpoznawany po numerze konta, a płatnik podatku po numerze ubezpieczenia społecznego. Do przedstawiania takich informacji w UML-u służą *kwalifikatory* asocjacji. Kwalifikator z reguły jest atrybutem elementu docelowego, aczkolwiek nie jest to wymóg. Ma on postać niewielkiego prostokąta umieszczonego pomiędzy asocjacją a elementem źródłowym. Nazwę kwalifikatora (zazwyczaj jest to nazwa atrybutu) wpisuje się w reprezentującym go prostokącie. Rysunek 2.30 przedstawia związek pomiędzy urzędem skarbowym a podatnikiem, z kwalifikatorem w postaci numeru identyfikacji podatkowej podatnika.



Rysunek 2.30. Kwalifikator asocjacyjny

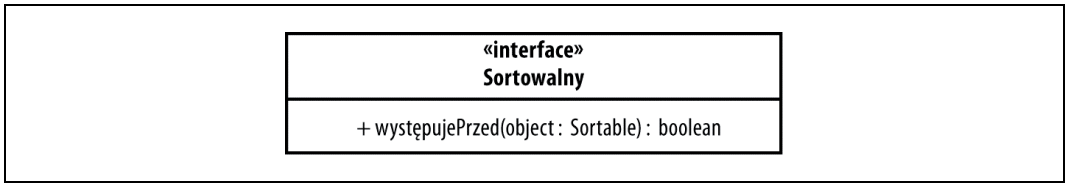
Warto zauważyć, że liczność pomiędzy kwalifikatorem asocjacji a podatnikiem wynosi 1. Oczywiście urząd skarbowy ma powiązania z wieloma podatnikami, ale użycie kwalifikatora wskazuje na fakt, że `NIP` w sposób jednoznaczny identyfikuje podatnika w `US`.

Interfejsy

Interfejs jest rodzajem klasyfikatora, który zawiera deklaracje właściwości i metod, ale nie implementuje ich. Interfejsy można stosować do grupowania wspólnych elementów klasyfikatorów w celu dostarczenia kontraktu, którego warunków musi dotrzymać klasyfikator implementujący te interfejsy. Można na przykład utworzyć interfejs o nazwie `Sortowalny` zawierający jedną operację o nazwie `występujePrzed(...)`. Każda klasa realizująca interfejs `Sortowalny` musi implementować operację `występujePrzed(...)`.

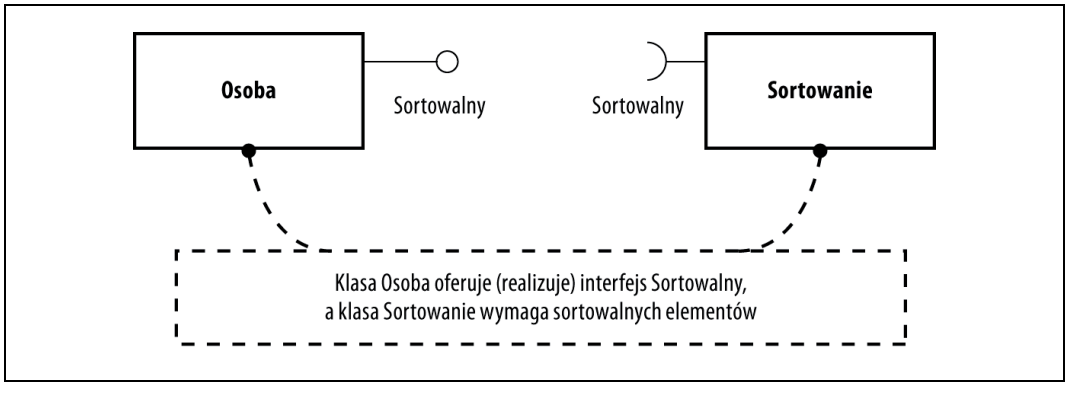
Niektóre nowoczesne języki programowania, np. C++, nie obsługują interfejsów. Interfejsy UML-a z reguły są reprezentowane w postaci klas czysto abstrakcyjnych. Inne języki, takie jak Java, obsługują interfejsy, ale nie pozwalają na definiowanie ich właściwości. W związku z tym nasuwa się wniosek, że zawsze w trakcie tworzenia modelu systemu należy pamiętać o sposobie jego przyszłej implementacji.

Interfejs można przedstawić na jeden z dwóch sposobów. Wybór jednego z nich powinien być oparty na tym, co próbujemy pokazać. Pierwszy sposób wykorzystuje standardową notację klasyfikatorów UML-a przy użyciu stereotypu «`interface`». Rysunek 2.31 przedstawia interfejs `Sortowalny`.



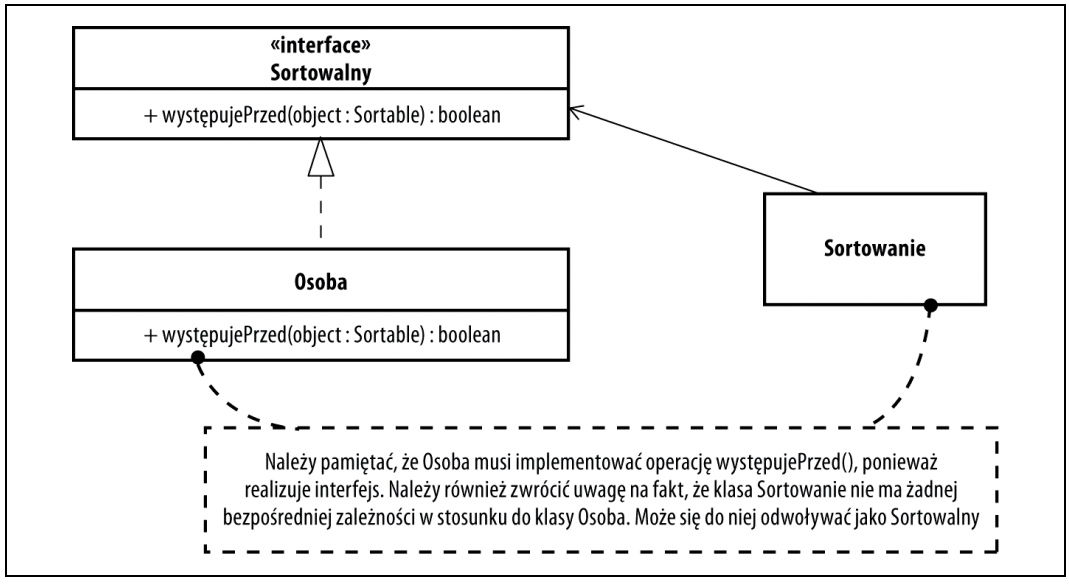
Rysunek 2.31. Interfejs Sortowalny

Drugi sposób to notacja typu *wtyczka-gniazdo*. Reprezentacja ta pokazuje mniej szczegółów dotyczących interfejsu, ale lepiej nadaje się do ukazywania powiązań z klasami. Interfejs reprezentowany jest przez symbol kółka, pod którym znajduje się jego nazwa. Klasy zależne od tego interfejsu dołączone są do odpowiadającego mu gniazda. Na rysunku 2.32 przedstawiono interfejs `Sortowalny` za pomocą notacji wtyczka-gniazdo.



Rysunek 2.32. Przykłady dostarczania i żądania interfejsów

Ze względu na fakt, że interfejs określa kontrakt tylko dla pewnego zestawu właściwości, nie można utworzyć jego egzemplarza bezpośrednio. W zamian mówi się, że klasa *realizuje* interfejs, jeżeli implementuje operacje i właściwości. Realizację sygnalizuje się poprzez zastosowanie przerywanej linii, która bierze początek w klasyfikatorze realizującym i prowadzi do interfejsu oraz jest zakończona pustą strzałką. Klasy zależne od interfejsu prezentowane są przy użyciu linii przerywanej z pustą strzałką (zależność). Rysunek 2.33 przedstawia klasę realizującą interfejs `Sortowalny` oraz zależną od niego klasę.



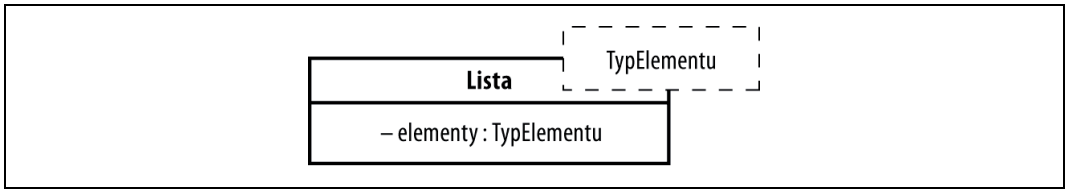
Rysunek 2.33. Klasa Osoba realizuje interfejs Sortowalny, a klasa Sortowanie jest od niego zależna

Implementacja operacji jest bardzo prosta — należy dostarczyć implementacji na klasyfikatorze realizującym o takiej samej sygnaturze co operacja wykonywana na interfejsie. Z reguły z operacją, która musi być honorowana przez jakąkolwiek implementację, związane są pewne ograniczenia semantyczne. Realizacja właściwości jest bardziej subtelna. Właściwość interfejsu stanowi, że dowolna realizująca go klasa musi w jakiś sposób przechowywać dane określone przez tę właściwość. Właściwość interfejsu niekoniecznie oznacza, że dla klasyfikatora realizującego będzie jakaś właściwość skojarzona. Klasyfikator musi jednak mieć możliwość przechowywania danych reprezentowanych przez tę właściwość i umożliwiać manipulowanie nią.

Szablony

Podobnie jak interfejsy pozwalają na określenie obiektów, z jakimi dana klasa może wchodzić w interakcje, UML umożliwia tworzenie abstrakcji dla typu klas, z którymi może komunikować się dana klasa. Można na przykład napisać klasę o nazwie `Lista` przechowującą obiekty dowolnego typu (w C++ prawdopodobnie byłby to typ `void*`, a w Java i C# `Object`). Jednak pomimo że chcemy, aby nasza klasa była w stanie obsłużyć obiekty dowolnego typu, chcemy również, aby wszystkie obiekty danej listy reprezentowały jeden typ. Tego rodzaju abstrakcje w UML-u można tworzyć za pomocą *szablonów*.

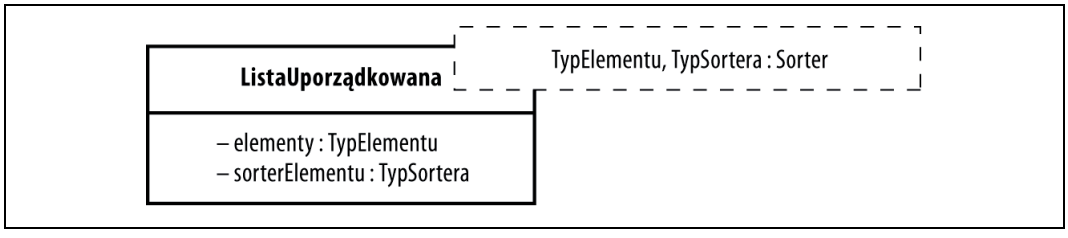
Aby zaznaczyć, że klasa jest szablonem (lub jest sparametryzowana), należy w jej górnym prawym rogu narysować prostokąt, którego boki są linią przerywaną. Dla każdego elementu, z którego chcemy zrobić szablon, należy podać nazwę stanowiącą znak zastępczy dla właściwego typu. Nazwę tego znaku należy wpisać w narysowanym prostokącie. Na rysunku 2.34 przedstawiono klasę `Lista` mogącą obsłużyć dowolny typ.



Rysunek 2.34. Klasa szablonowa Lista

W powyższym przykładzie dla jasności użyto nazwy `TypElementu` dla typu szablonowego. W praktyce często pisze się w skrócie tylko `T`.

W obrębie klasy można mieć wiele typów szablonowych. Ich nazwy należy wtedy rozdzielić przecinkami. W razie potrzeby nałożenia ograniczeń co do typów, które użytkownik może podmienić, należy zastosować dwukropek, a po nim wstawić nazwę typu. Rysunek 2.35 przedstawia bardziej skomplikowaną wersję klasy `Lista`, która wymaga sortera razem z typem obiektu, który ma być przechowywany na liście.



Rysunek 2.35. Klasa szablonowa z ograniczeniami dotyczącymi typów

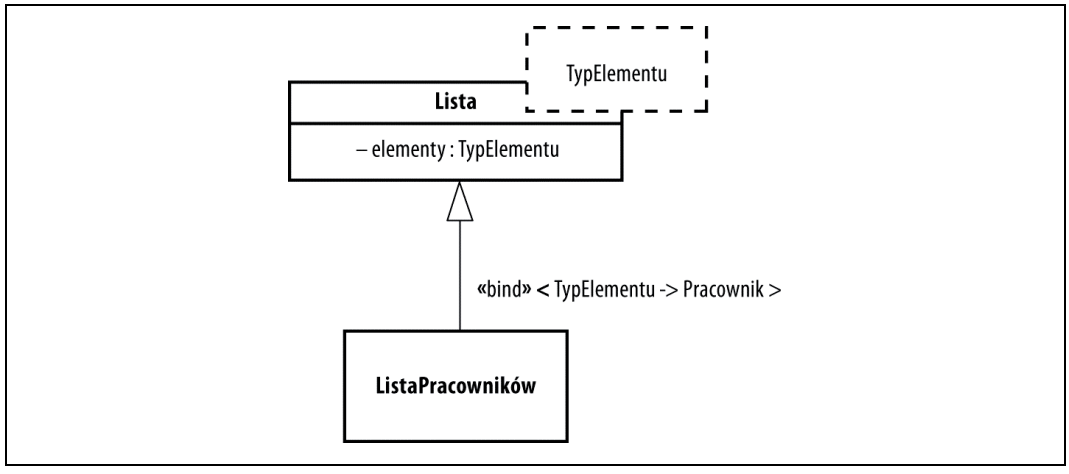
Definiowanie ograniczeń dla typów, które mogą zostać użyte, jest z funkcjonalnego punktu widzenia podobne do definiowania interfejsu dla składowej będącej szablonem. Wyjątkiem jest to, że użytkownik może mieć możliwość dalszego ograniczania egzemplarza naszej klasy poprzez zdefiniowanie podklasy naszego typu.

Tworząc egzemplarz klasy `Lista`, użytkownik musi podać rzeczywisty typ, który ma być zastosowany w miejsce `TypElementu`. Nazywa się to *wiązaniem* typu z szablonem. Wiązanie oznacza się za pomocą słowa kluczowego «`bind`», po którym następuje określenie typu przy użyciu następującej składni:

```
< TypSzablonowy -> RzeczywistyTyp >
```

Składnię wiązania można stosować wszędzie tam, gdzie odnosimy się do klasy szablonowej w celu zaznaczenia, że chcemy użyć wiązanej wersji tej klasy. Nazywa się to *wiązaniem jawnym*. Na przykład rysunek 2.36 przedstawia podklasę klasy `Lista` o nazwie `ListaPracowników`, która wiąże `TypElementu` klasy `List` z klasą o nazwie `Pracownik`.

Słowo kluczowe `bind` informuje również, jakie typy powinny zostać użyte z egzemplarzem szablonu. Nazywa się to *wiązaniem niejawnym* i zostało przedstawione na rysunku 2.37.



Rysunek 2.36. Jawne wiązanie szablonu

Rysunek 2.37. Niejawne wiązanie szablonu

Różne wersje diagramów klas

Ze względu na fakt, że diagramy klas świetnie nadają się do modelowania struktur, znajdują zastosowanie w obrazowaniu dobrze zdefiniowanych, hierarchicznych informacji. Z pewnym powodzeniem używa się ich do obrazowania schematów XML i bazodanowych. Aby wszystko było jasne: osoby zajmujące się schematami XML i bazodanowymi na co dzień mają pewne wątpliwości co do używania diagramów klas do obrazowania tego typu informacji. Konkretnie chodzi im o to, że diagramy te są zbyt ogólne. Każda domena ma własną notację, która może lepiej się nadawać do przedstawienia skomplikowanych powiązań lub informacji właściwych danej domenie. UML ma jednak tę zaletę, że jest wspólnym językiem rozumianym także przez osoby spoza branży XML-a i baz danych.

Schematy XML

Strukturę dokumentu XML można przedstawić za pomocą schematu XML. Schematy XML są dla dokumentów XML tym, czym klasy dla obiektów. Dokumenty XML są egzemplarzami schematów XML. W związku z tym nietrudno sobie zdać sprawę, że diagramy klas można stosować do modelowania schematów XML. Schematy te opisywane są za pomocą języka XSDL (XML Structure Definition Language). XSDL jest językiem tekstowym (w przeciwieństwie do graficznego UML-a), a jego zastosowania mogą być wszechstronne. Odwzorowanie XSDL-a w UML-u może ułatwić odczyt schematu.

Podstawowymi elementami XSDL są elementy XML połączone w sekwencje, wybory i struktury złożone. Do każdego elementu mogą być dołączone dodatkowe informacje w postaci atrybutów (co jest dosyć wygodne). Tworzący modele z reguły reprezentują elementy XML jako klasy UML-a, a atrybuty XSDL jako atrybuty UML-a. Każdy element jest połączony z następnym za pomocą strzałek kompozycji. Wartości dotyczące liczności podawane dla tych powiązań określają, ile razy jeden element pojawia się w innym. Listing 2.1 przedstawia przykładowy dokument XML opisujący element wyposażenia.

Listing 2.1. Przykładowy dokument XML

```
<?xml version="1.0" encoding="UTF-8"?>
<equipmentlist xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="equipment.xsd">
    <equipment equipmentid="H-1">
        <shortname>
            <langstring lang="en">Hammer</langstring>
        </shortname>
        <technicalcontact>
            <contact>
                <name>Ron</name>
                <telephone>555-1212</telephone>
            </contact>
        </technicalcontact>
        <trainingcontact>
            <contact>
                <name>Joe</name>
                <email>joe@home.com</email>
            </contact>
        </trainingcontact>
    </equipment>
</equipmentlist>
```

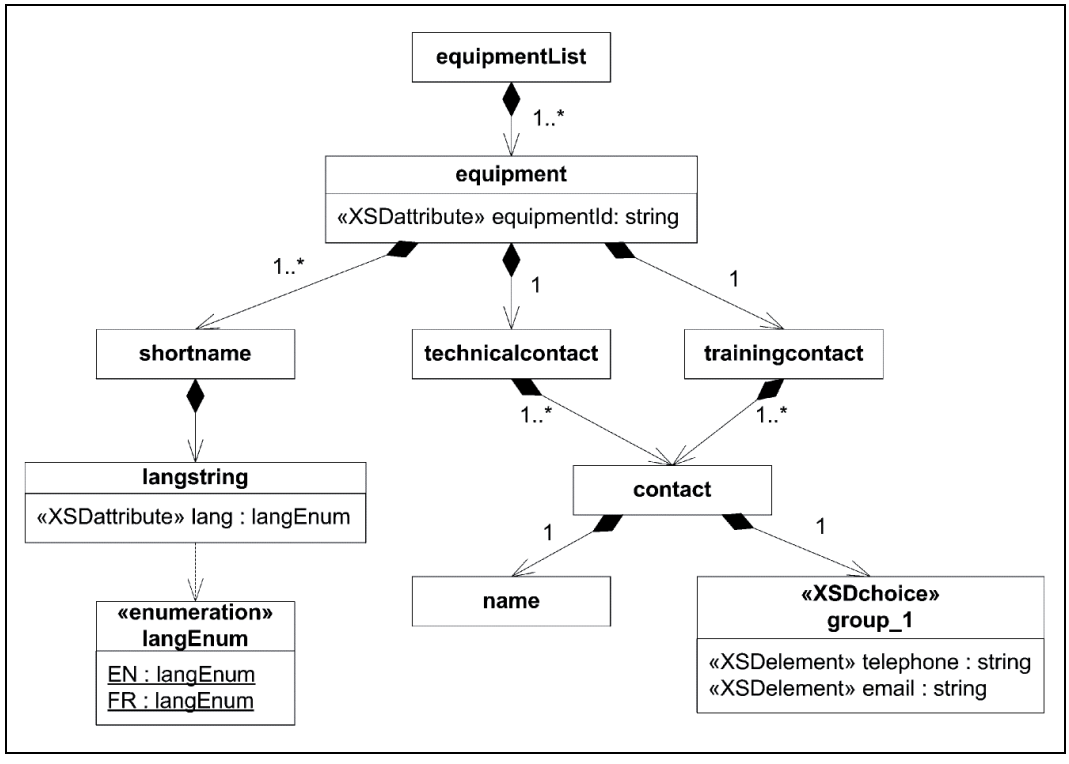
Listing 2.2 przedstawia kod XSDL opisujący powyższy dokument XML.

Listing 2.2. Schemat XML opisujący dokument XML z listingu 2.1

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified">

   <!-- ~Class: contact ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <xs:element name="contact" type="contact"/>
   <xs:complexType name="contact">
      <xs:sequence>
        <xs:element name="name" type="xs:string"/>
        <xs:choice>
          <xs:element name="telephone" type="xs:string"/>
          <xs:element name="email" type="xs:string"/>
        </xs:choice>
      </xs:sequence>
   </xs:complexType>

   <!-- ~Class: equipment ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <xs:element name="equipment" type="equipment"/>
   <xs:complexType name="equipment">
      <xs:sequence>
         <xs:element ref="shortname"/>
         <xs:element name="technicalcontact" type="technicalcontact"/>
         <xs:element name="trainingcontact" type="trainingcontact"/>
      </xs:sequence>
```

```
   <xs:attribute name="equipmentid" type="xs:string" use="required"/>
</xs:complexType>

<!-- ~Class: equipmentlist ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
<xs:element name="equipmentlist" type="equipmentlist"/>
<xs:complexType name="equipmentlist">
   <xs:sequence>
      <xs:element ref="equipment" minOccurs="1" maxOccurs="unbounded"/>
   </xs:sequence>
</xs:complexType>

<!-- ~Class: <<XSDtopLevelAttribute>> lang~~~~~~~~~~~~ -->
<xs:attribute name="lang" type="xs:language"/>

<!-- ~Class: langstring ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
<xs:element name="langstring" type="langstring"/>
<xs:complexType name="langstring">
   <xs:simpleContent>
      <xs:extension base="xs:string">
         <xs:attribute name="lang" use="required">
            <xs:simpleType>
               <xs:restriction base="xs:NMTOKEN">
                  <xs:enumeration value="en"/>
                  <xs:enumeration value="fr"/>
               </xs:restriction>
            </xs:simpleType>
         </xs:attribute>
      </xs:extension>
   </xs:simpleContent>
</xs:complexType>

<!-- ~Class: shortname ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
<xs:element name="shortname" type="shortname"/>
<xs:complexType name="shortname">
   <xs:sequence>
      <xs:element ref="langstring" minOccurs="1" maxOccurs="unbounded"/>
   </xs:sequence>
</xs:complexType>

<!-- ~Class: technicalcontact ~~~~~~~~~~~~~~~~~~~~~~~~~ -->
<xs:element name="technicalcontact" type="technicalcontact"/>
<xs:complexType name="technicalcontact">
   <xs:sequence>
      <xs:element ref="contact" minOccurs="1" maxOccurs="unbounded"/>
   </xs:sequence>
</xs:complexType>

<!-- ~Class: trainingcontact ~~~~~~~~~~~~~~~~~~~~~~~~~ -->
<xs:element name="trainingcontact" type="trainingcontact"/>
<xs:complexType name="trainingcontact">
   <xs:sequence>
      <xs:element ref="contact" minOccurs="1" maxOccurs="unbounded"/>
   </xs:sequence>
</xs:complexType>
</xs:schema>
```

Rysunek 2.38 przedstawia reprezentację UML-a powyższego schematu. Elementy reprezentowane są jako klasy, z jednym wyjątkiem: definicja `kontakt` zawiera opcję `wybór`. Rysunek 2.38 obrazuje to za pomocą innej klasy zaklasyfikowanej jako `XSDWybór`. Opcje zostały zaprezentowane jako atrybuty klasy.



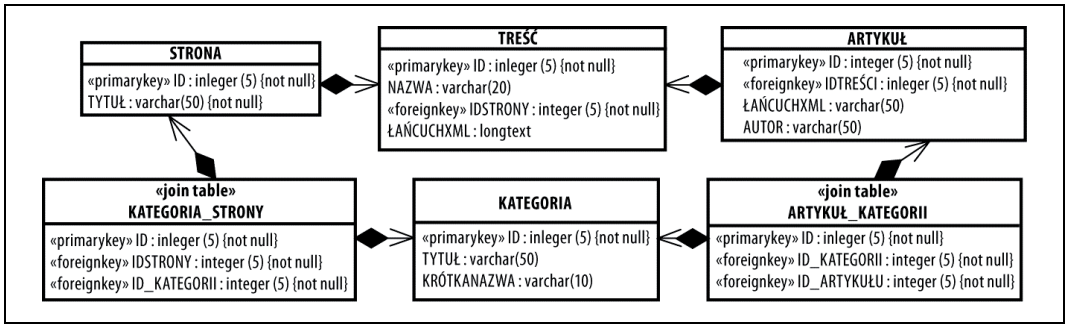
Rysunek 2.38. Reprezentacja w postaci diagramu klas z listingu 2.2

A więc mimo że graficzna reprezentacja informacji za pomocą UML-a upraszcza schemat, szczegóły takie jak sekwencje i rodzaj informacji mogą zostać utracone, jeżeli nie zostaną zastosowane dodatkowe mechanizmy rozszerzające UML-a, takie jak ograniczenia lub stereotypy.

Schematy baz danych

Schemat bazy danych można bardzo udanie przedstawić w UML-u, odwzorowując tabele w postaci klas, a wiersze w postaci atrybutów. Informacje dodatkowe, takie jak klucze podstawowe, klucze obce i ograniczenia, można przedstawić za pomocą ograniczeń lub stereotypów UML-a. Relacje pomiędzy tabelami można wykazać za pomocą asocjacji pomiędzy klasami (zazwyczaj przy użyciu kompozycji, ale to tylko konwencja). Rysunek 2.39 przedstawia schemat bazy danych zaprezentowany za pomocą diagramów UML-a.

Podobnie jak w przypadku schematów XML niewielu administratorów baz danych modeluje bazy przy użyciu diagramów UML-a. Diagramy klas UML-a są przydatne do przekazywania informacji schematów we wspólnym języku, ale brak im ekspresji (i personalizacji) bardziej standardowej notacji bazodanowej — Entity Relation Diagrams (ERD).



Rysunek 2.39. Przykładowy schemat bazy danych przedstawiony za pomocą diagramu klas