4. Język UML

Język modelowania pojęciowego (*Unified Modeling Language*) jest językiem formalnym. Jeśli coś w nim zapiszę, to każdy odczyta dokładnie to, co napisałem (nawet jeśli będzie miał nieco złej woli). Innymi słowy, jest to próba uwolnienia się od problemów związanych z niejednoznacznością języków naturalnych, w których tę samą wypowiedź można różnie zrozumieć. W konsekwencji każdy element UML znaczy coś konkretnego i tylko to – nic więcej. I każdy element może zostać wykorzystany tylko w określonych sytuacjach.

UML stworzyli: Grady Booch, Ivar Jacobson i James Rumbaugh, którzy byli pracownikami Rational Software Corporation. UML (według Rational) jest językiem przeznaczonym do specyfikowania, konsultowania, wizualizacji i dokumentowania elementów systemów. Jest też ugruntowanym językiem w środowisku inżynierii oprogramowania (Śmiałek M., 2005).

UML jest podzielony na diagramy, każdy z diagramów składa się z elementów, które służą do przedstawienia różnych sytuacji. Na przykład do przedstawienia dynamiki systemu, procesu służą diagramy czynności; do przedstawienia funkcjonalności jakiegoś systemu służy diagram przypadków użycia. W tej książce UML został ograniczony tylko do jednego diagramu: **diagramu klas** (lub inaczej: **diagramu struktury statycznej**). Ten diagram jest wykorzystywany m.in. do opisu informacji przestrzennej. Dyrektywa INSPIRE (Dyrektywa INSPIRE, 2007) i różne dokumenty z nią związane wykorzystują „oryginalny" UML nieco zmodyfikowany, by lepiej spełniał swoje zadania. Jest on zdefiniowany w specyfikacji technicznej ISO 19103 (ISO/TS 19103, 2005).

W książce tej będę opisywał UML „dostosowany" do INSPIRE, czyli ten zdefiniowany w ISO/TS 19103.

W niektórych opracowaniach są wyróżniane perspektywy UML: pojęciowa, specyfikacyjna, implementacyjna (Fowler M. i in., 2002). Dwie ostatnie dotyczą głównie modeli oprogramowania. My ograniczymy się do perspektywy pojęciowej, a więc konstruowane modele będą dotyczyły pojęć danej dziedziny (Fowler M. i in., 2002). W naszym przypadku będą to modele (schematy aplikacyjne), które mają być niezależne od późniejszego sposobu implementacji (rodzaju bazy danych, aplikacji, systemu operacyjnego, sprzętu komputerowego itp.). Modelujemy tylko interesujący nas fragment rzeczywistości.

Jak każdy język, UML ma swój alfabet dostosowany do potrzeb, które język ma spełniać – służyć do zapisu modelu. Zaczniemy więc od alfabetu UML. „Wypowiedziami" w UML są schematy aplikacyjne. Składają się one z różnych elementów: pakietów, klas, atrybutów, relacji, ograniczeń, liczności, stereotypów, typów itp. O każdym z tych elementów poniżej trochę opowiem. Wszystkie je zaliczam do alfabetu.

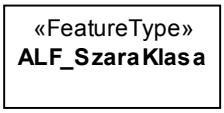
4.1. Alfabet

Klasy

Klasa jest abstrakcyjnym przedstawieniem obiektu istniejącego lub nie; obiektu, który z jakichś względów jest dla nas ważny i chcemy go umieścić w modelu. Z drugiej strony taki obiekt umieszczony w modelu jest reprezentantem wszystkich obiektów tego typu, które są tak samo opisywane (czyli dla których zbieramy informacje o takich samych właściwościach/cechach). Reasumując, można stwierdzić, że:

obiekt => klasa => wszystkie obiekty tego samego rodzaju i tak samo opisywane.

Klasa jest przedstawiana na modelu w postaci prostokąta, który może być podzielony na kilka części. Na rys. 4.1 klasa jest przedstawiona w sposób ogólny. Wynika z niego, że SzaraKlasa jest klasą, że należy do modelu o skrócie ALF, ma stereotyp «*FeatureType*» i albo nie ma atrybutów, albo je ma, ale ich nie przedstawiono. Ten ogólny sposób przedstawienia klasy stosuje się w przypadku, gdy klas jest bardzo dużo i po prostu nie ma miejsca na dokładniejszy opis albo gdy specjalnie zależy nam tylko na pokazaniu, jakie klasy będą występować w modelu i jakie zależności (relacje) będą je łączyć.



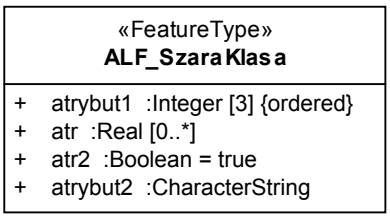
Rys. 4.1. Klasa – sposób 1

Stereotyp (a w zasadzie jego nazwa) jest umieszczany w podwójnych nawiasach (niektórzy nazywają je kątowymi) nad nazwą klasy. Stereotypy rozszerzają notację UML, tzn. dzięki ich wykorzystaniu możemy rozszerzyć podawane informacje dotyczące klasy. W tym przypadku sygnalizują, jaką rolę dana klasa odgrywa w modelu. Stereotyp «*FeatureType*» jest używany głównie do klas opisujących obiekty przestrzenne, «*DataType*» do klas definiujących typy danych, a «*CodeList*» i «*Enumeration*» do list wyliczeniowych. W dalszej części powiem jeszcze o stereotypach «*Union*» i «*Voidable*».

Już wiemy, że nazwą klasy na rys. 4.1 jest „SzaraKlasa". Przedrostek „ALF" informuje, do jakiego modelu dana klasa należy. Jest to istotne, zwłaszcza gdy będziemy w konstruowanym schemacie aplikacyjnym wykorzystywać klasy z różnych modeli. Taki sposób zapisu nazw klas (z przedrostkami) został przyjęty m.in. w INSPIRE. „ALF" będzie u nas oznaczał alfabet.

Atrybuty

Bardziej szczegółowy sposób definiowania klasy w schemacie jest przedstawiony na rys. 4.2. Gdy decydujemy, że jakiś obiekt ma się znaleźć w bazie danych, to zależy nam na umieszczeniu jakichś danych o tym obiekcie. Zależy nam, by opisać pewne jego cechy, które nas interesują. W UML cechy czy właściwości obiektu opisuje się z wykorzystaniem atrybutów klasy (która reprezentuje obiekt w modelu). Na rys. 4.2 są zdefiniowane cztery atrybuty.



Rys. 4.2. Klasa – sposób 2

Atrybut jest opisem właściwości obiektu. Jego definicja powinna się składać z określenia widoczności, po niej musi się pojawić nazwa atrybutu, po dwukropku powinien zostać podany typ wartości, jakie może przyjąć, liczność (czyli ile wartości ma lub może przyjąć). To są elementy konieczne. Opcjonalnie w skład definicji mogą jeszcze wchodzić różne informacje, np. czy wartości mają być uszeregowane (*{ordered}*), czy ustalamy także wartość domyślną. Autorzy UML podają ogólną definicję atrybutu (Booch G. i in., 2001):

[widoczność] nazwa [liczebność] [:typ] [=wartość początkowa][określenie właściwości]

Poniżej zostaną krótko omówione elementy składające się na definicję atrybutu. Można założyć, że elementy w nawiasach są umieszczane w definicji atrybutu w modelu w zależności od ogólności opisu. Nazwa musi być podana zawsze z wyjątkiem przypadków, gdy opis jest tak ogólny, że nie ma wymienianych atrybutów. Liczebność jest też często nazywana licznością i oznacza dokładnie to samo.

Przed nazwą każdego z atrybutów znajduje się plus („+"), który określa widoczność atrybutu. Plus oznacza, że atrybut jest publiczny (*public*), czyli jest widoczny (zob. dodatek A, gdzie są podane inne typy widoczności). Tylko takimi atrybutami będziemy się zajmować. Kwestie dotyczące atrybutów prywatnych, chronionych na naszym etapie poznawania UML-a nie są niezbędne, a mogą być wręcz szkodliwe!

- Atrybut pierwszy – nazwą jest „atrybut1", typem *Integer*, czyli liczby całkowite. Liczność – trzy wartości tego atrybutu mają zostać zapisane i mają być one uporządkowane (jeśli „atrybut1" będzie oznaczał współrzędne jednego punktu z pomiaru GPS, to ich kolejność ma zostać ustalona i zawsze pozostawać taka sama, np. X, Y, Z).
- Atrybut drugi – nazwą jest „atr", typem *Real*, czyli liczby rzeczywiste. Liczność może być różna i zmieniać się od 0 do wartości niezdefiniowanej – *.
- Atrybut trzeci – nazwą jest „atr2", typem *Boolean*, czyli typ logiczny (zob. w dodatku A). Wartością domyślną jest „true". Liczność wynosi jeden i taka jest standardowo przyjmowana, dlatego jedynek nie umieszcza się w opisie atrybutów. Liczność wynosząca 1 oznacza, że atrybut jest „**wymagalny**", czyli

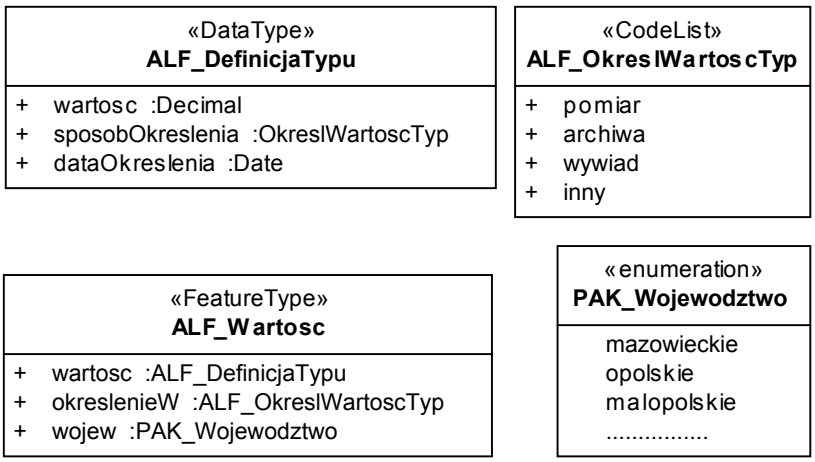
w późniejszej bazie danych „atr2" (i „atrybut2") muszą zostać wypełnione wartością, w przeciwnym razie nie powinien zostać zapisany w bazie cały obiekt (z powodu braku jednego atrybutu!).

- Atrybut czwarty – nazwą jest „atrybut2", typem *CharacterString*, czyli ciąg znaków (tekst).

W powyższych przykładach zostały wykorzystane tzw. typy proste. Występują też (bo muszą) typy złożone (zob. dodatek A): implementacyjne, zbiorowe, rekordowe itp. Wzmianki o niektórych z nich będą w dalszej części książki. Typ atrybutu może zostać także zdefiniowany w postaci klasy o odpowiedniej strukturze i stereotypie.

Liczność jest parametrem wchodzącym w skład definicji atrybutu i relacji. Podajemy ją w nawiasach kwadratowych dla atrybutów lub bez nawiasów dla relacji. W przypadku relacji należy podać dwie liczności – dla każdego jej zakończenia, bo relacja łączy dwie klasy (czasami na modelu można tak manipulować znakiem graficznym danej relacji, że wygląda to tak, jakby za pomocą jednej relacji połączono kilka różnych klas; bardzo często tak się robi w przypadku, gdy od jednej klasy dziedziczy kilka klas, ale to tylko tak wygląda na rysunku). Liczności równej 1 najczęściej na modelu się nie umieszcza – przyjmuje się, że jest to wartość standardowa. Atrybuty o liczność 1 są nazywane „wymagalnymi", czyli muszą zostać wypełnione wartością pod rygorem nieutworzenia obiektu w bazie danych. Podobnie jest w przypadku relacji.

Dla atrybutów liczność większa od 1 oznacza, że atrybut taki jest wielowartościowy – na rys. 4.2 „atrybut1" ma zawierać (można w nim zapisać) trzy wartości. Dla relacji liczność oznacza liczbę instancji klasy (czyli obiektów z bazy danych), które biorą jednocześnie udział w relacji. Często liczność jest podawana jako przedział, np. 1..*, 0..1, 3..15, gdy liczba wartości zapisanych w atrybucie lub liczba obiektów w relacji może być różna, ale w ściśle określonych granicach. Niektóre aplikacje umieszczają na modelu liczność po typie atrybutu, a nie przed nim.



Rys. 4.3. Klasa – sposób 2 cd.

Na rys. 4.3 przedstawiono klasę „ALF_Wartosc" z typami atrybutów, które są zdefiniowane w postaci klas. Klasa, która została utworzona po to, by zdefiniować nowy typ danych (na rys. klasa „ALF_DefinicjaTypu"), otrzymuje stereotyp «*DataType*». Klasa taka (poza nielicznymi wyjątkami) nie wchodzi w żadne relacje z innymi klasami.

Zdarza się, że atrybut może przyjąć wartości ze ściśle zdefiniowanej listy. Taki typ też jest definiowany jako klasa, której atrybutami są wartości, jakie może przyjąć atrybut. Klasa definiująca listę jest nazywana **klasą wyliczeniową** i otrzymuje stereotyp «*CodeList*» (klasa „ALF_OkreslWartoscTyp") lub «*Enumeration*» (klasa „PAK_Wojewodztwo"). «*CodeList*» występuje, gdy użytkownik pracujący z bazą danych może samodzielnie listę rozszerzyć, «*Enumeration*» otrzymują klasy, dla których lista zdefiniowana w schemacie aplikacyjnym jest ostateczna i potem – podczas działania aplikacji – nikt jej zmienić nie może. Należy wyraźnie zaznaczyć, że chociaż stereotyp «*CodeList*» jest często wykorzystywany w schematach będących częściami (zwykle załącznikami) rozporządzeń, to w tych przypadkach możliwość dodawania nowych elementów jest nie do zrealizowania, ponieważ zmiana czegokolwiek w opublikowanym rozporządzeniu wymaga jego nowelizacji.

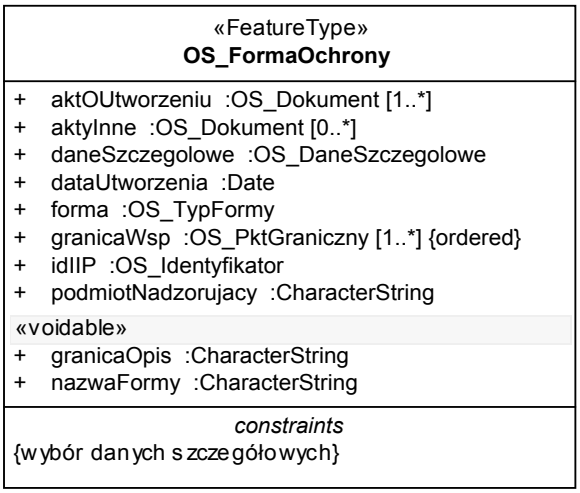
Przedrostek „PAK" w klasie „PAK_Wojewodztwo" oznacza, że klasa ta została zdefiniowana w modelu „PAK" – Podział Administracyjny Kraju. Z kolei typ atrybutu „dataOkreslenia" – „*Date*" jest zdefiniowany w specyfikacji ISO 19103 (ISO/TS 19103, 2005).

Nazwy klas, stereotypów, atrybutów, typów danych itp. też są elementami alfabetu. **Nazwa** jest ciągiem znaków (tak samo jak w językach naturalnych). Przy podawaniu nazw należy stosować kilka prostych zasad. Rozsądne jest oczekiwanie, by nazwa niosła ze sobą informację, co dana klasa, atrybut itp. reprezentują, jaką rolę odgrywają. Nazwy atrybutów z rysunku 4.2 nie są najlepsze, ale formalnie (poza tym, że są mało zrozumiałe) nic im zarzucić nie można. Nazwy powinny być rzeczownikami w liczbie pojedynczej. Gdy chcemy, by nazwa

składała się z kilku wyrazów, można je oddzielić myślnikiem, podkreśleniem lub następny wyraz zacząć pisać wielką literą, jak np. atrybut „okreslenieW" z rys. 4.3 czy typ tego atrybutu „OkreslWartoscTyp". Nie wolno w nazwach używać spacji! Nazwy klas poprzedzamy przedrostkiem (skrót nazwy modelu) i piszemy wielką literą. Nazwę atrybutu piszemy małą literą (widać to na dotychczasowych rysunkach). Nazwę stereotypów piszemy wielką lub małą literą. By uniknąć problemów ze stronami kodowymi w przypadku stosowania polskich znaków (co się zdarzało), najlepiej z nich zrezygnować i pisać np. „Slask" zamiast „Śląsk".

Ograniczenia

Na pewne elementy modelu można nakładać ograniczenia. Bardzo częste jest uzależnianie wartości czy zakresu wartości, jakie może przyjąć atrybut, od wartości innego atrybutu. Na rys. 4.4 jest pokazana klasa ze zdefiniowanym ograniczeniem. Dotyczy ono wyboru wartości atrybutu „daneSzczegolowe". Klasa „OS_FormaOchrony" została zdefiniowana w schemacie aplikacyjnym wykonanym dla Generalnej Dyrekcji Ochrony Środowiska i w sposób ogólny opisuje chronione formy.



Rys. 4.4. Klasa ze zdefiniowanym ograniczeniem

Opis bardziej szczegółowy znajduje się w atrybucie „daneSzczegolowe". Inaczej jest opisywany park krajobrazowy, park narodowy, użytek ekologiczny czy pomnik przyrody. Jaką formę będzie w bazie danych reprezentować klasa „OS_FormaOchrony", decyduje wartość atrybutu „forma". Gdy „forma" przyjmie wartość „obszarNatura2000", to dane szczegółowe muszą opisywać obszary Natura 2000. O ograniczeniach jest wzmianka w dodatku B. Ograniczenia mogą też dotyczyć liczności, np. atrybut ma liczność [0..1], ale w pewnym przypadku (po spełnieniu jakiegoś warunku) ma być atrybutem wymagalnym, czyli mieć liczność 1. Ograniczenia dotyczą działania aplikacji, są pewnym dodatkiem, więc dłużej nie będę się tutaj nad nimi rozwodził.

Stereotyp «Voidable»

Na rys. 4.4 pojawił się nowy stereotyp. «**Voidable**» jest rodzajem stereotypu, który nadajemy atrybutom lub relacjom. Ma on specjalne znaczenie i jest rozszerzeniem profilu UML zdefiniowanym w specyfikacji ISO/TS 19103. Atrybuty mające liczność 1 są tzw. atrybutami wymagalnymi. Ale może się tak zdarzyć, że nie będziemy mogli zdobyć informacji o wartości takiego atrybutu. Z drugiej strony chcemy móc zapisać taki obiekt w bazie danych. W takim przypadku trzeba zastosować specjalny atrybut, w którym zamiast wartości atrybutu zostanie zapisana informacja o przyczynach braku wartości. Oznaczeniem takiej sytuacji jest stereotyp «*Voidable*». Tym

Tabela 4.1. Wartości specjalnego atrybutu nilReason

Wartość	Definicja	Wartość w ISO 19136
Nie stosuje się	Nie ma zastosowania (wartości) w danym kontekście	*inapplicable*
Brak danych	Prawidłowa wartość atrybutu nie jest obecnie znana, ale właściwa wartość może nie istnieć	*missing*
Tymczasowy brak danych	Wartość atrybutu będzie dostępna w późniejszym terminie	*template*
Nieznany	Prawidłowa wartość atrybutu nie jest znana, ale właściwa wartość prawdopodobnie istnieje	*unknown*
Zastrzeżony	Wartość atrybutu jest zastrzeżona	*withheld*

specjalnym atrybutem jest *gml:nilReason*, który jest zdefiniowany w normie ISO 19136 (ISO 19136, 2007). Wartości tego atrybutu są przedstawione w tabeli 4.1.

Może się wydawać dziwne, jak dla atrybutu typu np. *Integer* zamiast wartości całkowitej można wpisać np. „*missing*". By móc zaimplementować taką sytuację, w normie ISO 19136 (ISO 19136, 2007) zostały zdefiniowane typy:

- *gml:booleanOrNilReason*,
- *gml:doubleOrNilReason*,
- *gml:integerOrNilReason*,
- *gml:NameOrNilReason*,
- *gml:stringOrNilReason*,

które umożliwiają wpisanie albo wartości zgodnej z danym typem lub wartości atrybutu specjalnego.

Operacje

Kolejna część prostokąta stanowiącego klasę może charakteryzować operacje. Operacje są usługami (inaczej procesami), które dana klasa (a w zasadzie jej implementacje w bazie danych, czyli konkretne obiekty) ma wykonać na polecenie innej klasy (rys. 4.5). Definicja operacji wymaga oczywiście podania jej nazwy (zaczynając od małej litery i dalej zgodnie z określonymi wcześniej regułami podawania nazw) i typu wyniku. Opcjonalnie można w nawiasach podać typy parametrów. Zapis jest w zasadzie identyczny jak w językach programowania, np. *Delphi*, podaje się definicje procedur. Diagram klas jest opisem, służy do zamodelowania struktury statycznej, więc operacje w modelach często nie występują. Tutaj podaję ten bardzo krótki opis dla porządku, że takie elementy też mogą wchodzić do definicji klasy.

Rys. 4.5. Klasa ze zdefiniowaną operacją

Do modelu można dodawać **notatki**, których głównym zadaniem jest wyjaśnienie dokładniej czegoś, co według autora modelu może wymagać dodatkowego komentarza (rys. 4.6). Mają wygląd prostokąta z zagiętym jednym rogiem. Notatki mogą być połączone z jakąś klasą (klasami), jak to jest na rys. 4.6, co ma pokazać, do czego odnosi się komentarz, lub mogą zostać umieszczone oddzielnie.

Relacje

Model, w którym zostały zdefiniowane tylko typy obiektów, bez podania związków zachodzących między nimi, nie jest modelem bazy danych. Jest raczej zbiorem definicji pojedynczych obiektów. Bazę danych tworzą obiekty połączone zachodzącymi między nimi relacjami oraz system zarządzania tą bazą. Na rys. 4.6 są przedstawione cztery najczęściej stosowane relacje. Pierwszą z nich, idąc od góry, jest relacja **dziedziczenia**, która oznacza, że klasa „ALF_DzieckoCzesc" przejmuje wszystkie atrybuty, relacje, ograniczenia itp. od klasy „ALF_RodzicCalosc", która jest wskazywana grotem strzałki. Przejmuje tzn., że oprócz „swoich" dwóch atrybutów ma także trzy atrybuty klasy „ALF_RodzicCalosc". Gdyby klasa „ALF_RodzicCalosc" była połączona relacją z jakąś inną klasą (np. ALF_X), to dzięki relacji dziedziczenia klasa „ALF_DzieckoCzesc" też byłaby połączona tą samą relacją z ALF_X. To właśnie przejmowanie cech innej klasy było powodem nazwania tej relacji „dziedziczeniem" w analogii do ludzi, którzy dziedziczą (przejmują) cechy rodziców. Istotna różnica między ludźmi a UML jest taka, że w UML klasa odgrywająca rolę „dziecka" przejmuje wszystkie cechy klasy będącej „rodzicem". Ludzkie dzieci zwykle dziedziczą część cech (często nie te, które rodzice uważają, że powinny).

Drugą relacją jest **agregacja silna** lub – jak jest często nazywana – **kompozycja**. Agregacja jest relacją całość – część i całością jest klasa znajdująca się po stronie zaczernionego rombu. Po stronie klasy będącej częścią „ALF_DzieckoCzesc" jest podana liczność [1..*], która oznacza, że w skład całości musi wchodzić przynajmniej jedna część, a może ich być wiele.

Trzecią relacją jest **agregacja słaba** lub po prostu **agregacja**. Całością, jak w przypadku kompozycji, jest klasa po stronie rombu. Można też podać liczność. Dla relacji, podobnie jak dla atrybutów, jeśli liczność nie jest podana, to oznacza, że wynosi ona 1.