V Konferencja PLOUG
Zakopane
Październik 1999

Język UML

Kazimierz Subieta
Instytut Podstaw Informatyki PAN, Warszawa
Polsko-Japońska Wyższa Szkoła Technik Komputerowych, Warszawa

Streszczenie

UML (Unified Modeling Language), zunifikowany język do modelowania, jest następcą i syntezą notacji występujących w obiektowych metodykach analizy i projektowania systemów informatycznych, które pojawiły się w końcu lat 80-tych i na początku lat 90-tych. Jest on oparty o pojęcia obiektowości, takie jak obiekty, klasy, atrybuty, związki, agregacje, dziedziczenie, metody i inne. UML powstał w wyniku połączonych wysiłków trzech znanych metodologów: Grady Boocha, Ivara Jacobsona i Jamesa Rumbaugh'a. Są oni autorami popularnych metodyk: OODA (Booch), Objectory (Jacobson) i OMT (Rumbaugh). UML jest zestawem pojęć oraz notacji graficznych (diagramów), które pozwalają wszechstronnie odwzorować modelowaną dziedzinę problemu, założenia projektowanego systemu informatycznego, oraz większość istotnych aspektów jego konstrukcji. UML jest obecnie wspomagany przez wiele narzędzi CASE. Został on także zaakceptowany jako przemysłowy standard przez ciało standardyzacyjne OMG rozwijające standard CORBA. Artykuł wprowadza w motywacje i cele UML oraz objaśnia na przykładach podstawowe pojęcia, notacje i zastosowania UML.

1 Wstęp

Wzrost popularności obiektowości w informatyce spowodował prawdziwą eksplozję metodyk i notacji określanych jako „obiektowe", wśród nich: BON, Catalysis, DOOS (Wirfs/Brock), EROOS, Express, Fusion, Goldberg/Rubin, MainstreamObjects, Martin/Odell, MOSES, Objectory (Jacobson), OMT (Rumbaugh), OOA/OOD (Coad/Yourdon), OOAD (Booch), OSA, Sintropy, OOSA (Shlaer/Mellor) i inne [Booc94, CoYo91a, CoYo91b, Cole+94, GoRu95, Jaco92, MaOd91, MaOd94, MaOd96, Mart93, Meye97, Rumb+91, Your+95]. Mimo różnic zarówno w podejściu jak i w przeznaczeniu, metodyki te mają ze sobą wiele wspólnego. Dość powszechne stało się odczucie, że różnice pomiędzy notacjami wprowadzanymi w tych metodykach są drugorzędne.

UML ma (w założeniu) dokonać unifikacji notacji używanych w istniejących metodykach. W odróżnieniu od metodyk obiektowych, które oprócz notacji ustalają także postępowanie w każdej fazie projektu, UML jest tylko zestawem pojęć i notacji. Może on być użyty do dowolnej metodyki. Zdaniem autorów, UML swoim zakresem przykrywa większość elementów i faz procesów analizy i projektowania.

UML jest dziełem trzech znanych metodologów: Grady Booch'a, Ivara Jacobsona i James'a Rumbaugh'a (określanych w literaturze jako „*three amigos*", trzej przyjaciele), w ramach firmy Rational Software Corporation. Są oni autorami popularnych metodyk: OMT [Rumb+91] (głównym autorem był J.Rumbaugh), OODA [Booc94] (autorem był G.Booch) oraz Objectory [Jaco92] (głównym autorem był I.Jacobson). Celem ich wysiłków jest uzyskanie popularności wybranej przez nich notacji i przez to opanowanie znacznej części rynku narzędzi CASE, szkoleń, analiz, ekspertyz, projektów, konsultacji, itd. Szanse rynkowe UML zostały wzmocnione poprzez zaakceptowanie go jako przemysłowego standardu przez ciało standardyzacyjne OMG (*Object Management Group*) opracowujące standard CORBA [OMG95].

Pierwsze wersje (UML 0.8-0.91) datują się na lata 1995-1996. W styczniu 1997 powstała wersja UML 1.0 [UML97], która została przesłana do OMG. Wersja UML 1.1, powstała w końcu 1997, została zatwierdzona jako składnik standardu OMG. Wersja UML 1.3 ukazała się w kwietniu 1999. Obecnie mówi się o wersji UML 1.4. Ponadto autorzy UML pracują nad zunifikowaną metodyką analizy i projektowania [Kruc98, JBR99]. UML jest przedmiotem ogromnej ilości książek, artykułów, stron WWW i innych materiałów, patrz np. [BJR98, Cant98, DoHu98, DSWi98, FSJ97, Laem97, Mull99, OdFo98, Oest99, RoSc99, RJB99, Subi98, UML97, WaKl99]. Słowniki [FiEy95, Subi99] wyjaśniają znaczenia terminów i pojęć obiektowości, które leżą u podstaw UML.

Podstawowym celem UML jest modelowanie systemów (nie tylko oprogramowania) z użyciem pojęć obiektowych. UML jest notacją pośrednią, pomostem pomiędzy ludzkim rozumieniem struktury i działania programów, a kodem programów. Taka notacja jest niezbędna do specyfikacji, konstrukcji, wizualizacji i dokumentacji wytworów związanych z systemami intensywnie wykorzystującymi oprogramowanie. Diagramy UML ustanawiają bezpośrednie powiązanie elementów modelu pojęciowego z wykonywalnymi programami. Jednocześnie umożliwiają one objęcie zagadnień związanych ze skalą problemu, towarzyszących złożonym systemom o krytycznej misji.

Ideałem byłoby utworzenie języka do modelowania użytecznego zarówno dla ludzi jak i maszyn, czyli czegoś w rodzaju super-języka programowania, który z jednej strony byłby całkowicie adekwatny w stosunku do ludzkiej percepcji, zaś z drugiej strony, mógłby być użyty do automatycznej generacji programów. W związku z tym (zdaniem autorów UML) prace nad konstrukcją UML nie są istotnie różne od zaprojektowania nowego języka programowania. Można jednak mieć sceptyczny stosunek do tego rodzaju zapowiedzi. UML jest przede wszystkim językiem odwołującym się do ludzkiej percepcji i wyobraźni. Uczynienie z niego języka algorytmicznego wymagałoby m.in. precyzji w specyfikacji struktur danych oraz środków manipulacji tymi strukturami danych, co nieuchronnie prowadziłoby do znacznego zmniejszenia czytelności diagramów, czyli zmniejszenia potencjału UML jako środka modelowania

pojęciowego. Uwzględnienie w jednym języku zarówno wymagań dotyczących naturalności, poglądowości i wysokiego poziomu abstrakcji niezbędnego dla ludzi zajmujących się projektem, jak i wymagań dotyczących algorytmicznej precyzji niezbędnej dla komputera, wydaje się nieosiągalne.

Zgodnie z deklaracjami autorów, UML przykrywa wszystko to, co może być zrobione przy pomocy istniejących metodyk. Jak można się domyśleć, to twierdzenie ma podłoże marketingowe i nie jest ani do udowodnienia, ani do obalenia, gdyż bazuje na subiektywnych odczuciach. (Autorzy innych metodyk są często innego zdania.) Wysiłek autorów UML jest skoncentrowany na stworzeniu wspólnego meta-modelu (unifikacji semantyki) i wspólnej notacji (odbioru tej semantyki przez ludzi). Promowany jest iteracyjny i przyrostowy proces rozwoju oprogramowania, który jest napędzany przez przypadki użycia (*use cases*) i skoncentrowany na koncepcyjnej architekturze projektowanego systemu. Zdaniem autorów, UML przykrywa także projektowanie systemów współbieżnych i rozproszonych.

2 Diagramy definiowane w UML

Ze względu na mnogość aspektów projektowanych systemów żadna pojedyncza perspektywa spojrzenia na projektowany system nie jest wystarczająca. Tę sytuację można porównać do projektu złożonego mechanizmu. Pojedynczy rzut tego mechanizmu na jedną płaszczyznę nie jest wystarczający do zrozumienia jego budowy; potrzebne jest wiele rzutów i przekrojów. Stąd metodyki projektowania wprowadzają wiele środków graficznych (diagramów) służących do „rzutowania" analizowanego lub projektowanego systemu na pewien szczególny jego aspekt.

W notacji UML można zapisać następujące diagramy:

- **Diagramy przypadków użycia** (*use case*). Są one podstawą metodyki Objectory. Ich głównym celem jest odwzorowanie funkcji projektowanego systemu w taki sposób, w jaki będą je widzieć jego użytkownicy. W metodykach opartych na UML przypadkom użycia przypisuje się szczególne znaczenie jako środka napędzającego cały proces rozwoju systemu.
- **Diagramy klas** (*class diagrams*). Są one odmianą dość klasycznych diagramów encja-związek (*entity-relationship*). Zostały praktycznie bez większych zmian przejęte z OMT. Wprowadzone są rozszerzenia poprawiające czytelność diagramów i przystosowujące je do konkretnej dziedziny zastosowań (np. stereotypy i odpowiadające im ikony). Odmianą diagramów klas sa diagramy pakietów (*package diagrams*).
- **Diagramy odwzorowujące dynamiczne własności systemu** (*behavior*), w tym:
  - **Diagramy sekwencji** (szczególny przypadek diagramów interakcji): pokazanie kolejności komunikatów przesyłanych pomiędzy obiektami dla pewnej sytuacji, np. przypadku użycia.
  - **Diagramy współpracy** (*collaboration*) (szczególny przypadek diagramów interakcji): podobne do diagramów sekwencji, ale z jednoczesnym odwzorowaniem statycznej struktury obiektów.
  - **Diagramy stanów**: odwzorowanie istotnych stanów (w których może znaleźć się proces przetwarzania) oraz przejść pomiędzy tymi stanami.
  - **Diagramy aktywności**: diagramy przepływu sterowania (*flowcharts*) uzupełnione o proste środki odwzorowania równoległych procesów.
- **Diagramy implementacyjne**, w tym:
  - **Diagramy komponentów**
  - **Diagramy wdrożeniowe** (*deployment*)

Ze względu na ograniczoną objętość tego artykułu omówione zostaną tylko niektóre z wymienionych diagramów.

1.1 Przypadki użycia

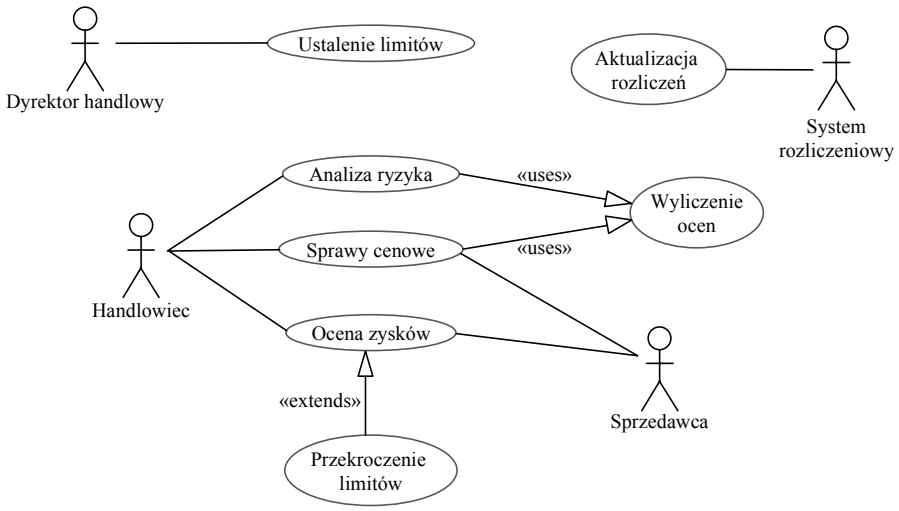
Przypadki użycia (*use cases*) [Jaco92, RoSc99, SWJ98] były w sposób intuicyjny stosowane w tradycyjnym projektowaniu systemów informatycznych na długo przed pojawieniem się metodyk obiektowych. Zasługą Jacobsona jest zarówno wyodrębnienie przypadków użycia jako metody projektowania, jak i stworzenie metodyki i notacji graficznej dla tego paradygmatu. Jak często bywa z powszechnie stosowanymi, lecz nie nazwanymi rzeczami, kariera przypadków użycia w literaturze z zakresu projektowania SI zaczęła się od wprowadzenia dla nich specjalnej nazwy, przypisaniu tej nazwie określonego znaczenia i rozpropagowanie tego pojęcia jako specjalnego paradygmatu projektowania.

Przypadek użycia jest to pewna nazwana lub dobrze określona interakcja pomiędzy użytkownikiem a systemem komputerowym. Przypadek użycia odwzorowuje pewną funkcję systemu w taki sposób, w jaki będą ją widzieć jego przyszli użytkownicy. Jakkolwiek w tym stwierdzeniu można podejrzewać banał, rzecz tak oczywistą, że nie warto o niej mówić, okazuje się, że dla dużych systemów o wielu złożonych i wzajemnie powiązanych funkcjach tego rodzaju podejście ma ogromny sens. Pozwala ono zapomnieć o strukturze/architekturze systemu i jego detalach technicznych i skoncentrować się na zewnętrznych funkcjach systemu. Podejście do projektowania od strony przypadków użycia jest uważane za znacznie bardziej zdrowe od podejść „technokratycznych", ponieważ sprzyja ono punktowi widzenia, w którym centralnym ośrodkiem zainteresowania staje się człowiek - przyszły użytkownik systemu - a nie budowa mechanizmu systemu. Jak pokazały doświadczenia wielu projektów, jedną z głównych przyczyn ich niepowodzeń było zbytnie skoncentrowanie się projektantów na detalach technicznych, z pominięciem lub brakiem dostatecznej uwagi dla interakcji pomiędzy użytkownikami a systemem komputerowym.

Podejście przypadków użycia ma przede wszystkim na względzie określenie wymagań na projektowany system. Celem tej metody jest:

- Głębsze zrozumienie użycia systemu będącego przedmiotem procesu projektowania.
- Zwiększenie stopnia świadomości analityków i projektantów co do celów tego systemu.
- Umożliwienie interakcji zespołu projektowego z przyszłymi użytkownikami systemu.
- Weryfikacja poprawności i kompletności projektu.
- Ustalenie wszystkich strukturalnych i funkcjonalnych własności systemu.
- Ustalenie składowych systemu i związanego z nimi planu konstrukcji systemu.
- Dostarczenie podstawy do sporządzenia planu testów systemu.

Diagramy przypadków użycia są bardzo proste. Rys.1 przedstawia diagram przypadków użycia dla pewnej (hipotetycznej) firmy zajmującej się pośrednictwem w sprzedaży.



Rys.1. Przykładowy diagram przypadków użycia

Model przypadków użycia dostarcza bardzo abstrakcyjnego poglądu na system z pozycji aktorów, którzy go używają. Nie włącza jakichkolwiek szczegółów, co pozwala wnioskować o systemie na odpowiednio ogólnym, abstrakcyjnym poziomie. Diagram przypadków użycia zawiera znaki graficzne oznaczające aktorów (ludziki) oraz przypadki użycia (owale z wpisanym tekstem). Te oznaczenia połączone są liniami odwzorowującymi powiązania poszczególnych aktorów z poszczególnymi przypadkami użycia.

Podstawowym zastosowaniem takich diagramów jest dialog z użytkownikami zmierzający do sformułowania wymagań na system. Stąd wynika, że diagramy i opis przypadków użycia muszą być bardzo naturalne, proste i nie mogą wprowadzać elementów komputerowej egzotyki i żargonu. W późniejszej fazie przypadki użycia mogą być wyspecyfikowane bardziej precyzyjnie przy pomocy notacji lub diagramów obiektowych. Następną fazą w postępowaniu opartym na przypadkach użycia jest rozpisanie ich przy pomocy notacji wprowadzanych w innych diagramach UML. Należy podkreślić, że tworzenie przypadków użycia jest niezdeterminowane i subiektywne. Na ogół różni analitycy tworzą różne modele.

Metoda przypadków użycia wymaga od analityka określenia wszystkich aktorów związanych w jakiś sposób z projektowanym systemem. Każdy aktor używa lub będzie używać systemu na kilka specyficznych sposobów (przypadków użycia). Zazwyczaj aktorem jest osoba, ale może nim być także pewna organizacja (np. biuro prawne) lub inny system komputerowy. Należy zwrócić uwagę, że metoda modeluje aktorów, a nie konkretne osoby. Jedna osoba może pełnić rolę wielu aktorów; np. być jednocześnie sprzedawcą i klientem. I odwrotnie, jeden aktor może odpowiadać wielu osobom, np. strażnik budynku. Aktor jest pierwotną przyczyną napędzającą przypadki użycia. Jest on sprawcą zdarzeń powodujących uruchomienie przypadku użycia, jak też odbiorcą informacji wyprodukowanych przez przypadki użycia. Aktor reprezentuje rolę, którą może grać w systemie jakiś jego użytkownik, np. kierownik, urzędnik, klient. Można tworzyć aktorów abstrakcyjnych, na podobieństwo klas abstrakcyjnych. Np. aktor „urzędnik" jest nadklasą dla aktorów „kasjer" i „dyrektor"; powiązania z konkretnymi przypadkami użycia mogą być ustalone zgodnie z tą hierarchią klas aktorów.

Przypadek użycia reprezentuje sekwencję operacji lub transakcji wykonywanych przez system w trakcie interakcji z użytkownikiem; np. potwierdzenie pisma, złożenie zamówienia. Przypadki użycia reprezentują przepływ zdarzeń w systemie i są uruchamiane (inicjowane) przez aktorów. Przypadek użycia jest zwykle charakteryzowany przez krótką nazwę. Opis przypadku użycia może być jednak dowolnie rozbudowany, w szczególności może zawierać następujące fragmenty:

- Jak i kiedy przypadek użycia zaczyna się i kończy?
- Opis interakcji przypadku użycia z aktorami, włączając w to *kiedy* interakcja ma miejsce i *co* jest przesyłane.
- Kiedy i do czego przypadek użycia potrzebuje danych zapamiętanych w systemie, lub jak i kiedy zapamiętuje dane w systemie?
- Wyjątki przy przepływie zdarzeń.
- Jak i kiedy używane są pojęcia dziedziny problemowej?

UML wprowadza także bardzo proste powiązania pomiędzy przypadkami użycia, oznaczane strzałkami dodatkowymi napisami «extends» (rozszerza) i «uses» (używa), Rys.1. Przypadek użycia podłączony przez strzałkę «extends» oznacza, że niekiedy (nie zawsze) dany przypadek użycia jest rozszerzony przez inny przypadek użycia. Przypadek użycia podłączony przez strzałkę «uses» oznacza wspólny fragment w wielu przypadkach użycia, który warto jest wyodrębnić ze względu na jego podobieństwo koncepcyjne oraz ze względu na późniejszą możliwość uniknięcia wielokrotnej implementacji tego fragmentu. Taki fragment jest niekiedy określany jako blok ponownego użycia (*reuse*).

Typowa dokumentacja przypadków użycia powinna zawierać następujące elementy:

- Krótki opis przypadku użycia.
- Przepływ zdarzeń opisany nieformalnie.
- Związki pomiędzy przypadkami użycia.

- Uczestniczące obiekty.
- Specjalne wymagania (np. czas odpowiedzi, wydajność).
- Obrazy interfejsu użytkownika.
- Ogólny pogląd na przypadki użycia (powiązania w postaci diagramów).
- Diagramy interakcji dla każdego aktora.

1.2 Diagramy klas

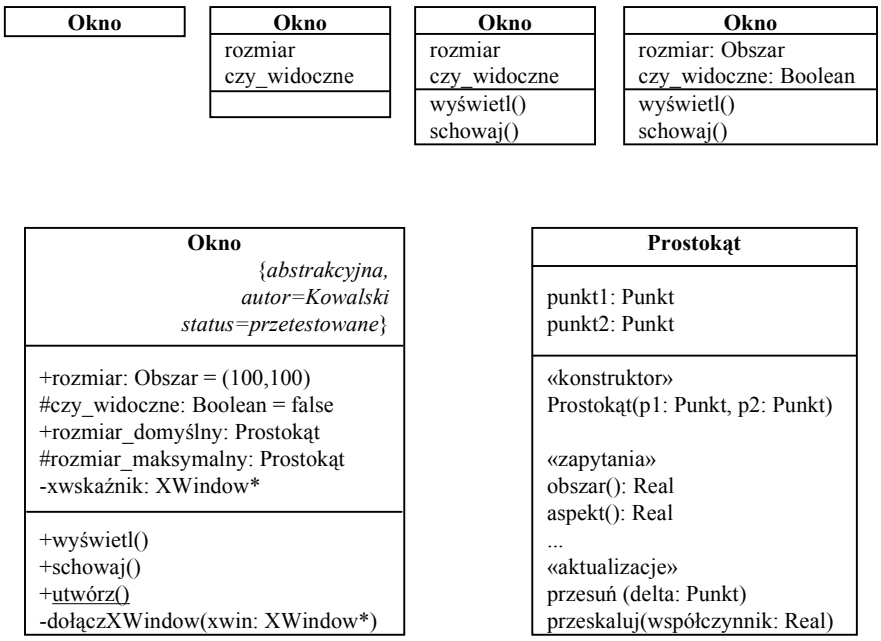
Diagram klas (zwany poprzednio także *diagramem asocjacji klas* lub *modelem obiektowym*) jest pojęciem centralnym we wszystkich znanych metodykach obiektowych. Z reguły diagram klas jest zmodyfikowanym diagramem encja-związek (z nieco innymi oznaczeniami) rozbudowanym o nowe elementy. W porównaniu do diagramów encja-związek (objaśnionych np. w [BCN92]) diagramy klas wprowadzają istotną nowość, mianowicie metody przypisane do specyfikowanych klas. Oprócz tego zasadniczego elementu pojawiają się w diagramach klas różnorodne oznaczenia o charakterze pomocniczym. Diagram klas pokazuje klasy w postaci pewnych oznaczeń graficzno-językowych powiązanych w sieć zależnościami należącymi do trzech kategorii:

- **Dziedziczenie** (*inheritance*), czyli ustalenie związku generalizacji/specjalizacji pomiędzy klasami.
- **Asocjacja** (*association*), czyli dowolny związek pomiędzy obiektami dziedziny przedmiotowej, który ma znaczenie dla modelowania.
- **Agregacja** (*aggregation*), czyli szczególny przypadek asocjacji, odwzorowujący stosunek całość-część pomiędzy obiektami z modelowanej dziedziny przedmiotowej.

Diagram klas w identycznej wersji jest stosowany zarówno do zapisu wyników analizy jak i do specyfikowania założeń projektowych. Diagramy klas są podstawą dowolnej analizy i dowolnego projektu, oraz sednem metody określanej jako „obiektowa”. Żaden projekt obiektowy nie może się obejść bez diagramu klas.

Istnieją trzy podstawowe zastosowania diagramów klas:

- **Zapis modelu pojęciowego**. Diagramy reprezentują pojęcia w dziedzinie zastosowań, które aktualnie podlegają analizie. Model pojęciowy nie musi być związany z jakimkolwiek oprogramowaniem; jest wyłącznie sformalizowaną wizją wyobrażeń powstających w trakcie twórczych procesów myślowych związanych z danym problemem.
- **Sformalizowana specyfikacja danych i metod**. Jest ona bardziej związana z oprogramowaniem, ale dotyczy jego zewnętrznego opisu (interfejsów) bez wchodzenia w szczegóły implementacyjne. Często podkreślaną zaletą obiektowości jest hermetyzacja, polegająca na oddzieleniu specyfikacji od implementacji. Dla wielu celów zależy nam wyłącznie na określeniu cech zewnętrznych pewnych bytów programistycznych (obiektów, metod, itd.), bez wchodzenia w szczegóły. Tego rodzaju rozróżnienie pomiędzy specyfikacją i implementacją jest charakterystyczne dla nowo powstających języków i standardów (CORBA, Java, ODMG).
- **Implementacja**. W tym zastosowaniu diagram klas może służyć bezpośrednio jako graficzny środek pokazujący szczegóły implementacji klas, np. w C++.



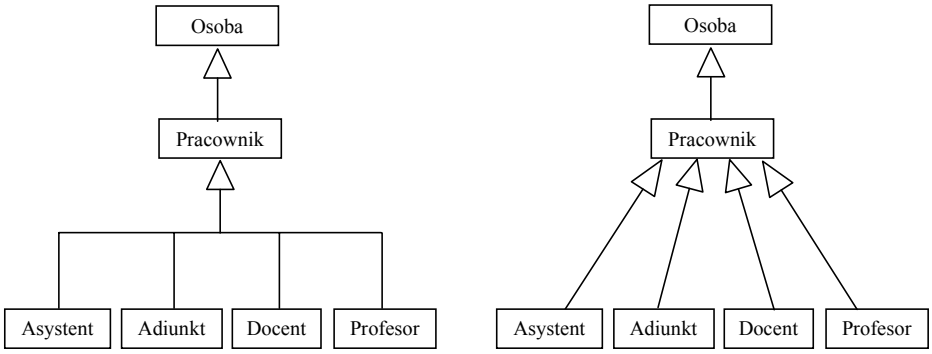
Rys.2. Przykłady specyfikacji klas

Oznaczenia klas w UML są prawie identyczne jak w OMT. Ilustruje je Rys.2. W wersji abstrahującej od szczegółów klasa jest zapisana w postaci prostokąta z wpisaną w środku nazwą klasy. W wersjach bardziej szczegółowych klasa jest reprezentowana przez prostokąt z trzema polami, gdzie pierwsze pole zawiera nazwę klasy, drugie pole zawiera specyfikację atrybutów, które posiadają obiekty tej klasy, zaś trzecie pole zawiera specyfikację metod. O ile dana klasa nie definiuje atrybutów (odp. metod) wówczas drugie (odp. trzecie) pole jest puste. Trzecie pole może być pominięte.

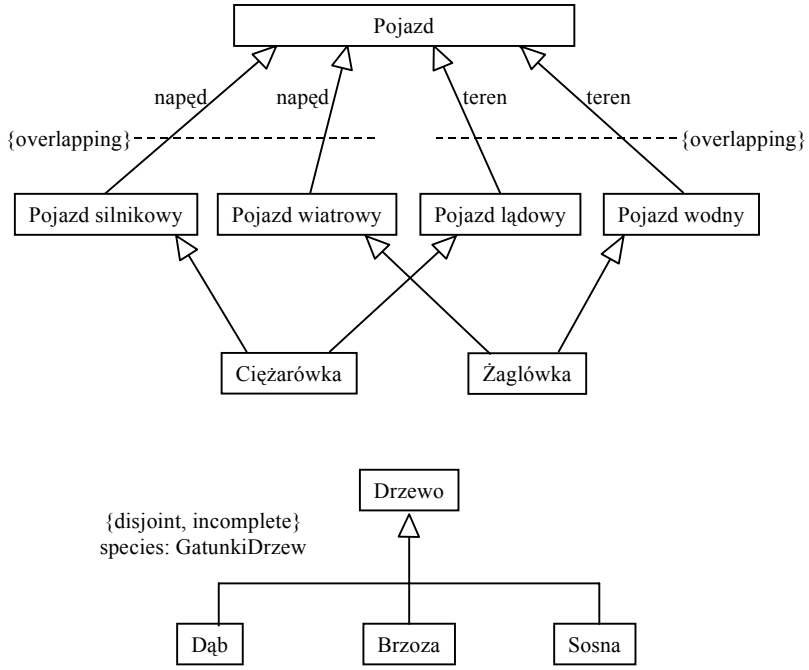
Jak widać z Rys.2, klasy mogą być wyspecyfikowane zarówno w sposób zgrubny, abstrahujący od szczegółów, jak i w sposób bardzo zbliżony do specyfikacji w języku programowania. Preferowany jest C++; +, #, - oznaczają odpowiednio *public, protected, private*, podkreślenie oznacza atrybut lub metodę klasową (*static*). Oznaczenia specyficzne dla C++ nie są jednak składową UML; nie ma przeszkód, aby UML zastosować do dowolnego języka programowania. UML przewiduje także specjalną notację dla interfejsów, tj. tych własności klas, które są uwzględnione w ich specyfikacji i mogą być używane przez programistów.

Dziedziczenie

Oznaczenia dla dziedziczenia zachowują składnię OMT (z minimalną zmianą). Strzałka (z białym trójkątnym grotem) prowadzi od pod-klasy do jej bezpośredniej nadklasy, Rys.3. Zakłada się, że obiekt pod-klasy automatycznie dziedziczy wszystkie atrybuty, metody, asocjacje i agregacje z wszystkich jej nadklas. W stosunku do OMT wprowadzono nieco bardziej precyzyjne oznaczenia dla przypadków, kiedy zakresy znaczeniowe klas nie są rozłączne. Mianowicie, użytkownik może explicite zadeklarować aspekt, według którego specjalizuje się dany obiekt (*napęd* lub *teren* na Rys.4), oraz określić fakt niepustego przecięcia zakresów znaczeniowych - zbiorów obiektów (*overlapping*). Klasy *Ciężarówka* i *Żaglówka* zostały zdefiniowane z użyciem wielokrotnego dziedziczenia. Na Rys.4 zilustrowano również możliwość określania faktu, że podklasy są rozłączne (*disjoint*) i nie przykrywają całego zakresu znaczeniowego ich nadklasy (*incomplete*).



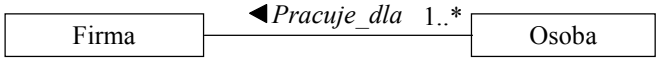
Rys.3. Równoważne oznaczenia dla związku generalizacji



Rys.4. Dodatkowe elementy specyfikacji dziedziczenia

Asocjacje

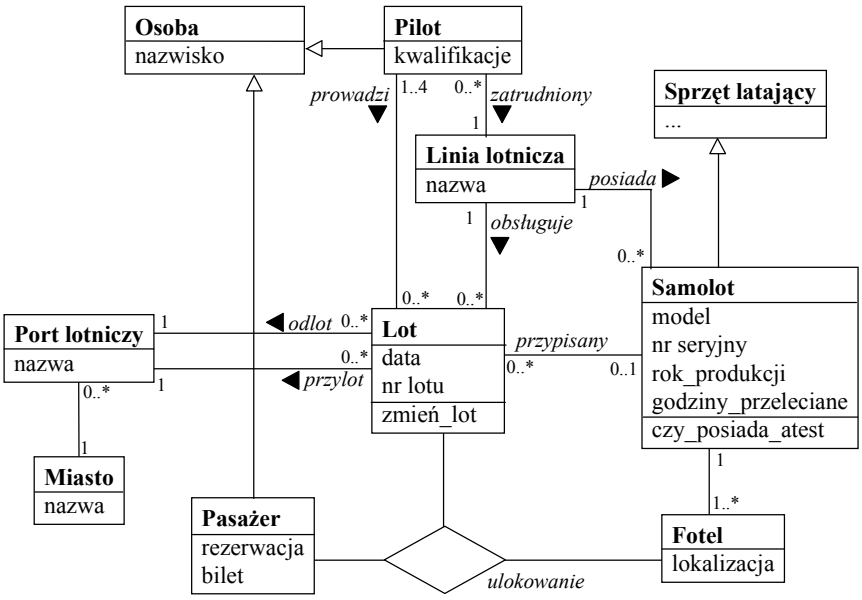
Oznaczenia klas w UML mogą być połączone liniami oznaczającymi asocjacje, czyli powiązania pomiędzy obiektami tych klas. Rys. 5 pokazuje specyfikację asocjacji *Pracuje_dla* pomiędzy obiektami klasy *Osoba* i obiektami klasy *Firma*. Czarny trójkącik określa kierunek wyznaczony przez nazwę powiązania (w danym przypadku określa on, że osoba pracuje dla firmy, a nie firma pracuje dla osoby). Asocjacje mają nazwy, takie jak *Pracuje_dla*, które wyznaczają znaczenie tej asocjacji w modelu pojęciowym. Jeżeli to znaczenie jest oczywiste, wówczas nazwę asocjacji można pominąć.



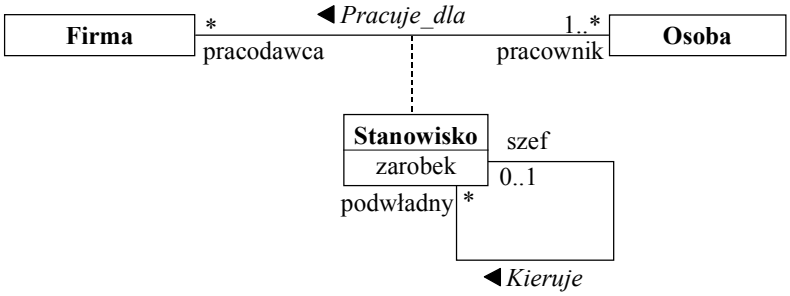
Rys.5. Oznaczenie dla asocjacji

Asocjacje mogą być wyposażone w oznaczenia liczności. Jak zwykle, liczność oznacza, ile obiektów innej klasy może być powiązane z jednym obiektem danej klasy; zwykle określa się to poprzez parę liczb (znaków) oznaczającą minimalną i maksymalną liczbę takich obiektów. Zapis liczności w UML jest naturalny i nie wprowadza specjalnych symboli graficznych. Oznaczenie 0..* oznacza liczność od zera do dowolnie wielkiej liczby; może być skrócone do pojedynczej

gwiazdki. Analogicznie, 1..* oznacza liczność od jeden do dowolnie wielkiej liczby, zaś 0..1 oznacza liczność zero lub jeden. Generalnie, oznaczenia liczności mogą być zapisem dowolnego podzbioru liczb całkowitych nieujemnych, gdzie podwójna kropka oznacza „od...do...", zaś gwiazdka oznacza dowolną liczbę. Rys.6 przedstawia prosty diagram klas zapisany w UML, przedstawiający założenia systemu dla potrzeb transportu lotniczego.



Rys.6. Przykładowy diagram klas w UML



Rys.7. Asocjacje z licznościami, rolami i atrybutami

Asocjacje mogą być także wyposażone w dodatkowe nazwy ról (przy odpowiednich końcach), np. *pracodawca* i *pracownik* na Rys.7. Asocjacje mogą być wyposażone w atrybuty. W tym celu przewidziano linię przerywaną łączącą daną asocjację z klasą, określaną jako „klasa asocjacji". Wewnątrz klasy asocjacji można zdefiniować atrybuty, operacje i inne cechy asocjacji. Podobnie jak w OMT, klasa asocjacji może być uważana za samodzielną klasę, w szczególności podlegać związkom dziedziczenia i asocjacji. Asocjacje mogą być również n-arne; oznaczenie (pusty w środku romb) jest identyczne jak w OMT.

Atrybuty i operacje

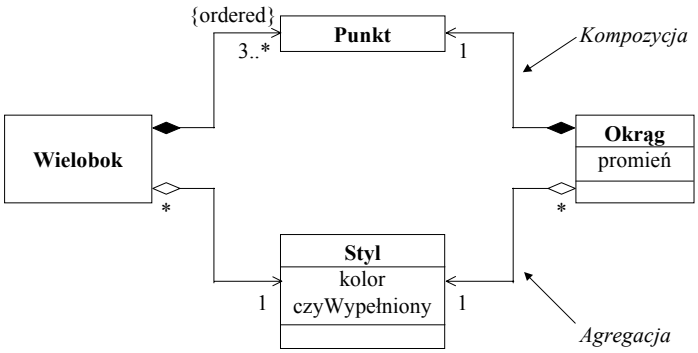
Atrybuty oraz operacje są specyfikowane w UML na zasadach zbliżonych do konwencji OMT. Możliwe jest dowolnie precyzyjne wyspecyfikowanie atrybutów i metod, włącznie z podaniem typów atrybutów, nazw i typów parametrów operacji, typu wyniku zwracanego przez operację oraz reguł hermetyzacji (widoczności). Klasy mogą zawierać oznaczenia atrybutów i operacji, które nie odnoszą się do obiektów tych klas, lecz do ich ekstensji. Przykładem jest operacja *średniZarobek* dla klasy *Pracownik*. Tego rodzaju własnościom odpowiada deklaracja *static* w definicji klasy w C++. W UML nazwy tego rodzaju atrybutów i operacji są podkreślane.

Atrybuty (również asocjacje) mogą być pochodne, tj. redundantne, wyliczone z innych atrybutów lub własności. Własności pochodne zaznacza się znakiem / na początku ich nazwy (tak jak w OMT).

Agregacje i kompozycje

Agregacja jest szczególnym przypadkiem asocjacji wyrażającym zależność część-całość. Np. silnik jest częścią samochodu, czyli obiekt-samochód jest agregatem obiektów będących jego częściami. Niestety, nie istnieje powszechnie akceptowana definicja agregacji, zaś wątpliwości co do jej znaczenia są zasadnicze. Dodatkowy mętlik wynika z wykorzystywania pojęcia agregacji do rozwiązania pewnych technicznych problemów związanych z ograniczeniami modelu obiektowego. Np. popularne wyjaśnienie technik obejścia braku wielo-dziedziczenia podaje, że można je „obejść przez agregację". Oznacza to np., że jeżeli klasa emerytowanych pracowników dziedziczy zarówno od klasy Emeryt jak i od klasy Pracownik, wówczas obiekt emerytowanego pracownika zawiera jako swoją „część" inny obiekt grupujący informację o cechach osoby jako emeryta. Mówi się, że obiekty pracownika i emeryta pozostają w związku agregacji; emeryt „jest częścią" pracownika (sic).

Autorzy UML podejmują próbę uporządkowania agregacji w taki sposób, aby zmniejszyć nieco zamieszanie dookoła tego pojęcia. Pozostawiając klasyczne pojęcia agregacji znane m.in. z OMT, wprowadzili oni mocniejszą formę agregacji, nazywając ją *kompozycją*. Związek kompozycji oznacza, że dana część może należeć tylko do jednej całości. Co więcej, część nie może istnieć bez całości, pojawia się i jest usuwana razem z całością. Usunięcie całości powoduje automatyczne usunięcie wszystkich jej części związanych z nią związkiem kompozycji. Klasycznym przykładem związku kompozycji jest *zamówienie* i *pozycja zamówienia*: pozycja zamówienia nie występuje oddzielnie (poza zamówieniem), nie podlega przenoszeniu od jednego zamówienia do innego zamówienia i znika w momencie kasowania zamówienia.



Rys.8. Agregacja i kompozycja

Rys.8 ilustruje zastosowanie agregacji i kompozycji. Każde wystąpienie obiektu *Punkt* należy albo do obiektu *Wielobok* albo do obiektu *Okrąg*; nie może należeć do dwóch obiektów naraz. Wystąpienie obiektu *Styl* może być dzielone przez wiele obiektów *Wielobok* i *Okrąg*. Usunięcie obiektu *Wielobok* powoduje kaskadowe usunięcie wszystkich związanych z nim obiektów *Punkt*, natomiast nie powoduje usunięcia związanego z nim obiektu *Styl*. Zwrócimy uwagę, że ograniczenie mówiące, iż obiekt *Punkt* może należeć do dokładnie jednego obiektu *Wielobok* lub *Okrąg* nie może być wyrażone w inny sposób.

Stereotypy

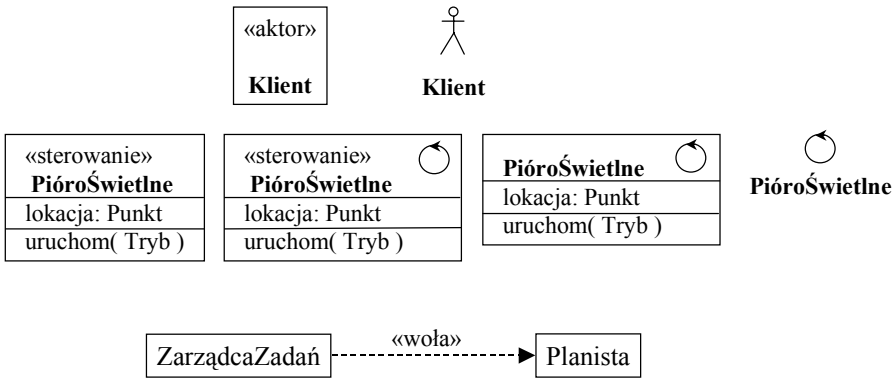
Idea stereotypów polega na ustaleniu pewnej meta-klasyfikacji obiektów (i innych bytów) i wprowadzeniu oznaczeń graficznych klas zgodnych z tą meta-klasyfikacją. Pomysł ten miał pewne zaszłości we wcześniejszych metodykach. Np. w metodyce Objectory (I. Jacobson'a) wprowadzono trzy rodzaje obiektów: obiekty interfejsu (*interface objects*), obiekty sterujące

(*control objects*) i obiekty rzeczywiste (*entity objects*). Te rodzaje obiektów miały swoje odbicie w graficznych ikonach oznaczających klasy. Pomysł ten został w UML uogólniony. Stereotypy mają tam specjalne oznaczenie (ciągi znaków wewnątrz nawiasów « »; np. «control object»). Mogą one występować w różnych kontekstach oraz mogą być zastąpione przez specjalne ikony. Ikony te nie są określone; mogą one być dowolnie wybrane przez użytkowników UML. Stereotypy są pomyślane jako środek zapewniający rozszerzalność UML; rozszerzenia te polegają na zdefiniowaniu nowych stereotypów. Wewnątrz diagramów klas mogą występować stereotypy klas, asocjacji i generalizacji, itd. Stereotypy są pewnymi wspólnymi nazwanymi własnościami tych bytów, dzięki czemu ich definicja może ulec uproszczeniu lub uszczegółowieniu.

Przykłady stereotypów sugerowane przez twórców UML są następujące:

- Stereotypy klas i obiektów: zdarzenie, wyjątek, interfejs, metaklasa, udogodnienie.
- Stereotypy typu obiektów: obiekty rzeczywiste, obiekty sterujące, obiekty interfejsu.
- Stereotypy zadań: proces, wątek.
- Stereotypy węzłów.
- Stereotypy pakietów.

Rys.9 pokazuje różne oznaczenia i zastosowania stereotypów.



Rys.9. Stereotypy

1.3 Diagramy interakcji

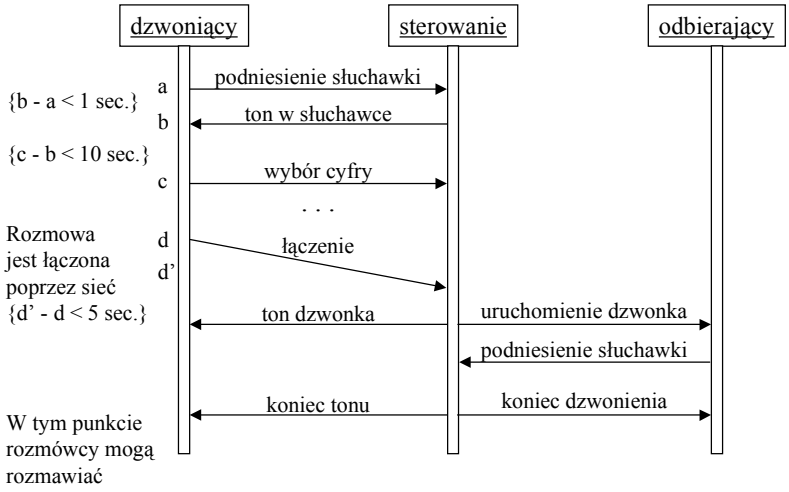
Jedną z trudniejszych rzeczy przy budowie obiektowego programu jest zrozumienie zależności w przepływie sterowania. Metody realizujące to sterowanie są rozproszone w wielu klasach, co powoduje, że ich wzajemna zależność i interakcja są często trudne do wyobrażenia. Diagramy interakcji służą do opisu zależności przy przesyłaniu komunikatów dla pewnej grupy obiektów. UML wprowadza dwa rodzaje takich diagramów: diagramy sekwencji i diagramy współpracy.

Diagramy sekwencji

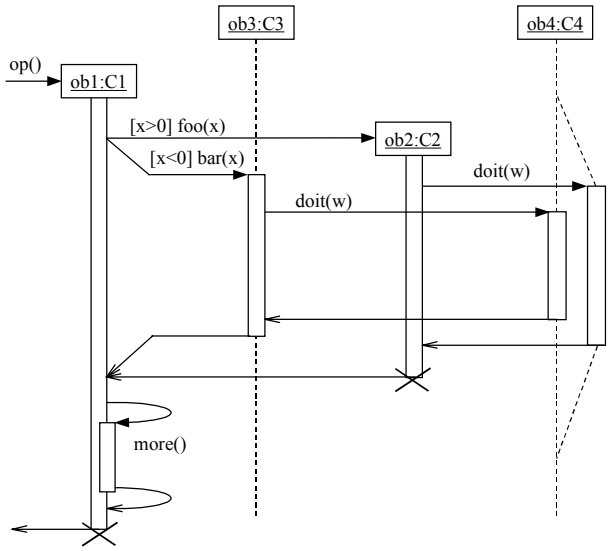
Diagramy sekwencji posiadają dwa wymiary: wymiar pionowy reprezentuje czas, zaś wymiar poziomy reprezentuje różne obiekty (kolejność obiektów nie ma znaczenia). Orientację wymiarów można zmienić: obiekty mogą być zaprezentowane w wymiarze pionowym, zaś czas w wymiarze poziomym. Generalnie, istotna jest kolejność pewnych zdarzeń, natomiast nie jest istotna rzeczywista miara czasu. Niekiedy (dla systemów uwarunkowanych czasowo) czas może być przedstawiony w pewnej mierzalnej skali.

Obiekty są zaznaczane w postaci prostokątów z wpisaną wewnątrz nazwą obiektu (klasy). Od każdego obiektu prowadzi linia reprezentująca „linię życia” obiektu, Rys.10. Na tych liniach zaznacza się momenty wysłania komunikatów przez dany obiekt do innego obiektu w postaci

strzałek prowadzących od jednej linii do innej linii. Diagram taki może także pokazywać pewne akcje zachodzące równolegle.



Rys.10. Prosty diagram sekwencji



Rys.11. Diagram sekwencji z warunkami, rekurencją, tworzeniem i kasowaniem

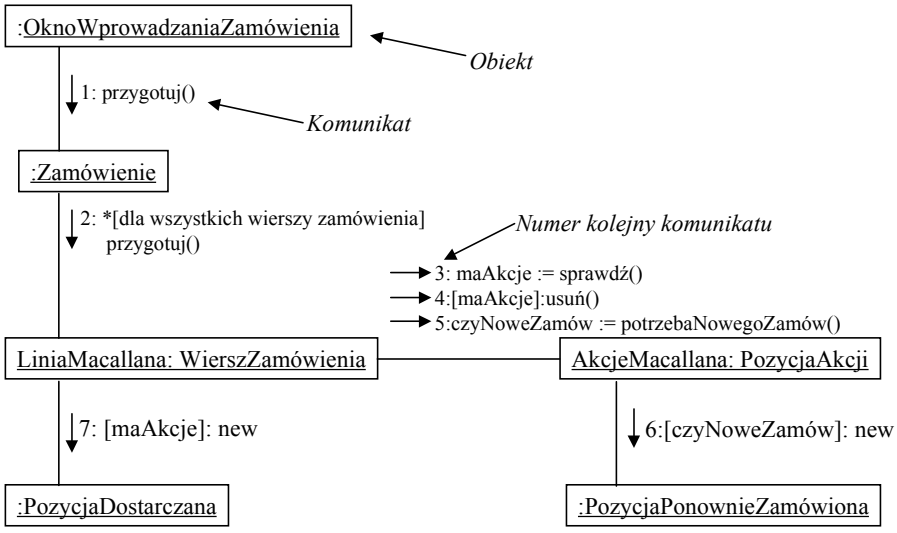
Obiekt może być aktywny lub nie. Jeżeli nie jest aktywny, wówczas jego linia życia jest przedstawiona w postaci linii przerywanej; jeżeli jest aktywny, to linia życia jest wąskim, długim prostokątem, Rys.11. Linia życia danego obiektu może zostać „rozszczepiona" na dwie lub więcej linii dla zobrazowania alternatywnego zachowania się obiektu w zależności od pewnych warunków. Na Rys.11 pokazaliśmy także tworzenie obiektów (strzałka dochodząca do oznaczenia obiektu), kasowanie obiektu (X na końcu linii życia), alternatywne zachowanie się obiektów w zależności od pewnych warunków, rekurencję (operacja *more*()). Istnieją dalsze propozycje rozszerzeń tego rodzaju diagramów, m.in. o oznaczenia iteracji.

Diagramy współpracy

W swojej idei diagramy współpracy są podobne do diagramów sekwencji. Ich istotą jest przedstawienie przepływu komunikatów pomiędzy obiektami. Współpraca pomiędzy obiektami włącza dwa aspekty: statyczną strukturę uczestniczących obiektów, włączając związki, atrybuty i operacje (jest to nazywane „kontekstem współpracy"), oraz sekwencję komunikatów wymienianych pomiędzy obiektami dla realizacji konkretnego zadania. Diagram współpracy może

być rozrysowany dla pewnych typów obiektów, dla pewnych operacji lub dla pewnych przypadków użycia.

W odróżnieniu od diagramów sekwencji wymiar czasu nie jest tu bezpośrednio odwzorowany. Natomiast odwzorowane są powiązania pomiędzy obiektami (prezentujące pewną część powiązań z diagramu klas). Jak poprzednio, obiekty są przedstawione w postaci prostokątów z wpisaną w środku nazwą. Diagram współpracy pokazuje komunikaty przesyłane pomiędzy poszczególnymi obiektami, które służą do realizacji określonego celu. Przepływ komunikatów jest zaznaczany strzałką, obok której znajduje się nazwa komunikatu oraz jego parametry. Notacja ta jednocześnie odwzorowuje dwa aspekty: strukturę obiektów uczestniczących w realizacji pewnego celu oraz zachowanie się obiektów podczas wykonywania programu. Takie diagramy mogą być obudowane dużą liczbą różnorodnych dodatkowych opcji (warunków, iteracji, stereotypów, tworzenia i usuwania obiektów, numerów sekwencji akcji, itd.).



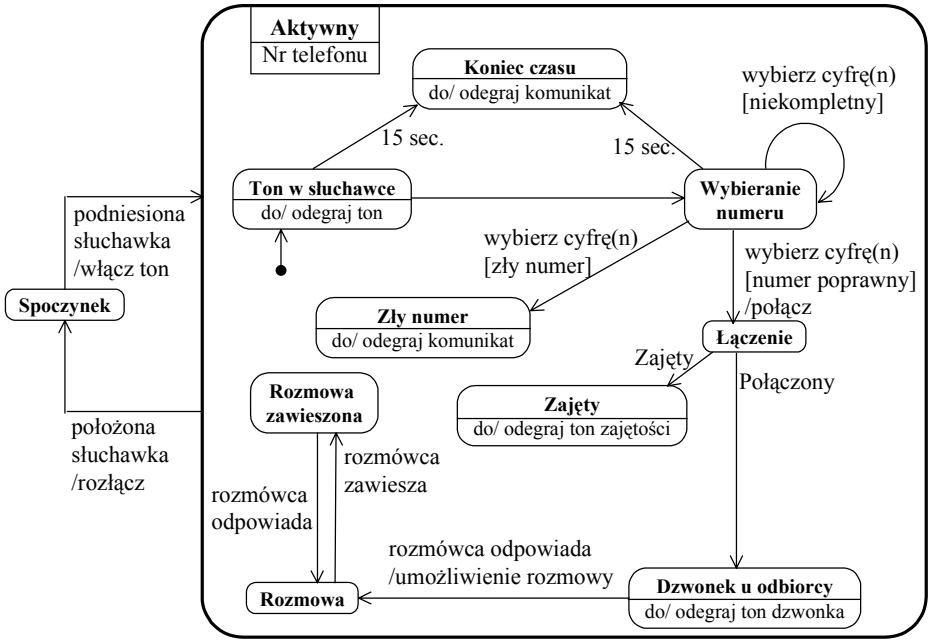
Rys.12. Przykład diagramu współpracy

Przedstawiony na Rys.12 diagram współpracy włącza obiekty narysowane jako prostokąty. Nazwy obiektów są zapisane w konwencji *nazwa obiektu : nazwa klasy*, z pominięciem nazwy obiektu, o ile jest w danym przypadku nieistotna. Linie pomiędzy oznaczeniami obiektów reprezentują związki (asocjacje) pomiędzy obiektami, strzałki reprezentują komunikaty przesyłane pomiędzy obiektami. Komunikaty są mogą być poprzedzone warunkami (w nawiasach kwadratowych). Numery przed komunikatami mogą być użyte dla zaznaczenia ich kolejności, co jest traktowane jako sposób odwzorowania czasu. Gwiazdka oznacza (bliżej nieokreśloną) iterację.

1.4 Diagramy stanów

Diagram stanów w swojej idei nawiązuje do automatu skończonego. Opisuje on stany pewnego procesu, które są istotne z punktu widzenia modelu pojęciowego tego procesu, oraz przejścia pomiędzy stanami. W swojej pierwotnej idei diagram stanów miał odwzorowywać stany obiektów pewnej klasy podczas ich cyklu życiowego oraz przejścia (*transitions*) pomiędzy tymi stanami powodowane przez zdarzenia lub komunikaty. Jak się wydaje (sądząc z przykładu zamieszczonego w dokumentacji UML) ta pierwotna idea uległa jednak na tyle silnej modyfikacji, że w istocie diagramy stanów *nie są* tymi diagramami stanów, o których jest mowa w teorii automatów. Są to dość klasycznie diagramy przepływu sterowania (*flowcharts*), z szeregiem drugorzędnych opcji syntaktycznych i semantycznych.

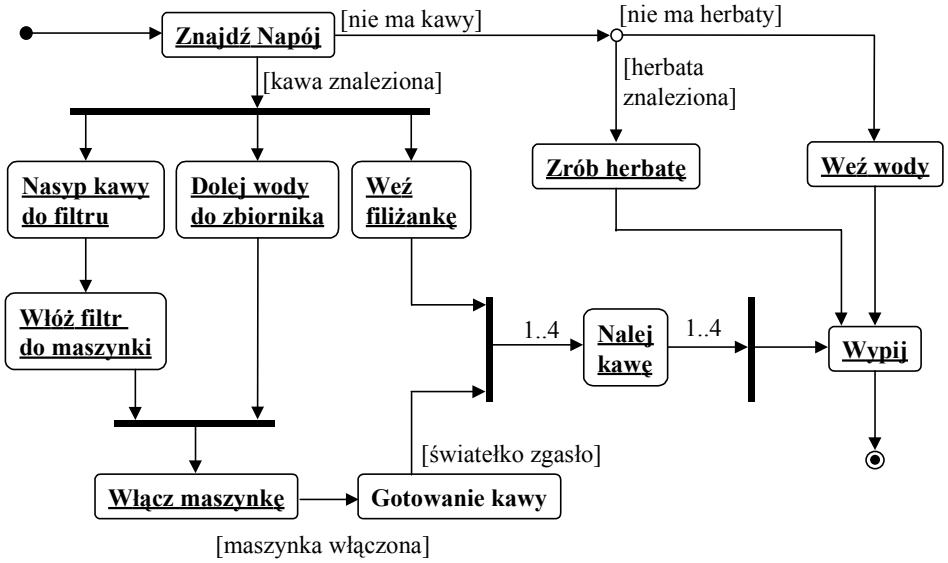
Na Rys.13 przedstawiony został przykładowy diagram stanów (z oczywistych względów uproszczony). Stany, w istocie reprezentujące stany sterowania procesu, zostały przedstawione w postaci prostokątów z zaokrąglonymi rogami. Przejścia pomiędzy stanami zostały zaznaczone w postaci strzałek. Tego rodzaju diagramy mogą być zagnieżdżane, tj. dowolny „stan” reprezentowany na diagramie może być uszczegółowiony w postaci odrębnego diagramu.



Rys.13. Diagram stanów

1.5 Diagramy aktywności

Diagramy aktywności w swej zasadniczej idei są dokładnie tym samym, co diagramy przepływu sterowania (*flowcharts, control flow diagrams*). Jedyną różnicą koncepcyjną jest to, że pojawiają się na nich elementy synchronizacji równoległych procesów (w dość uproszczonej formie, która dla pewnych celów może być niewystarczająca) Zdaniem autorów UML, diagramy aktywności są szczególnym przypadkiem diagramów stanów. Wydaje się jednak, że wprowadzenie diagramów stanów i diagramów aktywności w UML jest redundantne: w gruncie rzeczy różnice sprowadzają się do składni i niezbyt klarownych ustaleń, jakie informacje mogą być prezentowane na każdym z tych typów diagramów. Można sądzić, że obecność tych dwóch środków w ramach jednego języka jest powodowana nie merytoryczną potrzebą, lecz pewnymi zaszłościami historycznymi.



Rys.14. Diagram aktywności

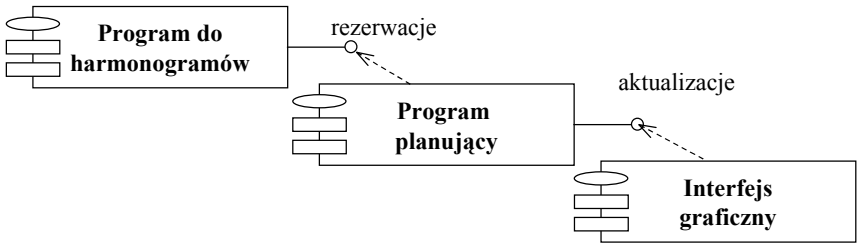
Rys.14 przedstawia diagram aktywności. Grube linie oznaczają miejsce „rozejścia się" i synchronizacji równoległych procesów. Pozostałe elementy składni, semantyki i pragmatyki diagramów aktywności wydają się dość oczywiste. Tak samo jak w przypadku diagramów stanów, zastosowanie diagramów aktywności sprowadza się do przypadków, kiedy opisywany proces lub przypadek użycia jest dostatecznie złożony, ale możliwy do ogarnięcia na diagramie graficznym o niezbyt dużych rozmiarach.

1.6 Diagramy implementacyjne

Diagramy implementacyjne pokazują niektóre aspekty implementacji SI, włączając w to strukturę kodu źródłowego oraz strukturę kodu czasu wykonania (*run-time*). W UML wprowadza się dwa typy takich diagramów: diagramy komponentów pokazujące strukturę samego kodu oraz diagramy wdrożeniowe pokazujące strukturę systemu czasu wykonania.

Diagramy komponentów

Diagramy komponentów pokazują zależności pomiędzy komponentami oprogramowania, włączając komponenty kodu źródłowego, kodu binarnego oraz kodu wykonywalnego. Poszczególne komponenty mogą istnieć w różnym czasie: niektóre z nich w czasie kompilacji, niektóre w czasie konsolidacji (*linking*) zaś niektóre w czasie wykonania. Diagram komponentów jest przedstawiany jako graf, gdzie węzłami są komponenty, zaś strzałki (z przerywaną linią) prowadzą do klienta pewnej informacji do jej dostawcy. Rodzaj zależności jest zależny od typu języka programowania. Diagram może także pokazywać interfejsy poszczególnych komponentów. Strzałki oznaczające zależności mogą prowadzić od komponentu do interfejsu, Rys.15.

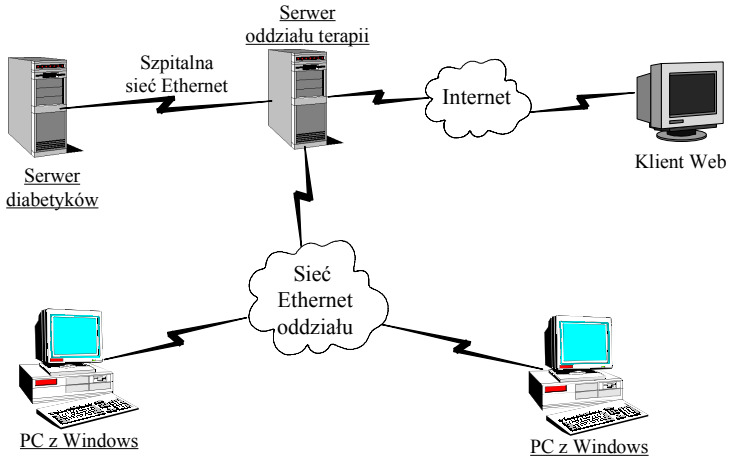


Rys.15. Diagram komponentów

Diagramy wdrożeniowe

Diagram wdrożeniowe pokazują konfigurację elementów czasu wykonania: komponentów sprzętowych, komponentów oprogramowania, procesów oraz związanych z nimi obiektów. Komponenty, które nie istnieją w trakcie czasu wykonania nie pojawiają się na tych diagramach; powinny one być pokazane na diagramach komponentów.

Diagram wdrożeniowe jest grafem, gdzie węzłami są elementy czasu wykonania połączone przez linie odwzorowujące ich połączenia komunikacyjne. UML określa pewną standardowa postać tego rodzaju diagramów, ale poprzez odpowiednie stereotypy (oznaczenia graficzne) można uczynić tego rodzaju diagram bardziej czytelnym. W szczególności, całkowicie legalna jest postać przedstawiona na Rys.16.



Rys.16. Diagram wdrożeniowy

1.7 Zależności pomiędzy diagramami UML

Autorzy UML nie zakładają, że w każdym projekcie będą użyte wszystkie diagramy wprowadzone w UML. W zależności od rozmiaru projektu oraz jego specyfiki niektóre z prezentowanych wyżej diagramów mogą być pominięte. Wydaje się, ze sednem UML są dwa typy diagramów: diagramy przypadków użycia oraz diagramy klas. Pozostałe diagramy mają znaczenie pomocnicze i mogą być użyte przy tworzeniu dokumentacji projektu w zależności od potrzeby.

Zależności pomiędzy różnymi diagramami w UML polegają na wykorzystaniu tych samych nazw klas i operacji w różnych diagramach. Zgodność użytych nazw i ich intuicyjna semantyka są podstawą do badania wzajemnej spójności diagramów, ich kompletności oraz minimalności.

3 Podsumowanie

UML w krótkim czasie stał się bardzo popularny. Należy oczekiwać, że przez wiele lat będzie dominował w obszarze analizy i projektowania, zwłaszcza, że stał się składową standardu OMG (CORBA). W odróżnieniu od innych tego rodzaju propozycji, UML nie ma ambicji być metodyką projektowania. Jest to raczej zestaw pojęć, oznaczeń i reguł syntaktycznych, który może być użyty w dowolnej metodyce. Notacja ta opiera się o podstawowe pojęcia obiektowości. UML wprowadza pojęcia i diagramy, które w założeniu mają przykryć większość aspektów modelowanych systemów. Jednakże kwestia semantyki tej notacji pozostaje mglista, co jest prawdopodobnie nieuchronnym skutkiem sprzeczności pomiędzy naturą procesów twórczych, wysokim poziomem abstrakcji i precyzyjną formalizacją. Równie poważnym problemem jest pragmatyka użycia tej notacji (tj. reguły dopasowania notacji do konkretnej sytuacji występującej w procesie projektowym). Niektóre rozwiązania zastosowane w UML (w szczególności, opis jego semantyki, tzw. metamodel) są przedmiotem krytyki, co jest prawdopodobnie nieuchronne, zważywszy na wagę tematu dla biznesu i konkurencję, która nie jest zainteresowana powodzeniem UML. Jest on, jak każda tego rodzaju notacja, pewnym kompromisem pomiędzy prostotą, uniwersalnością, precyzją i ograniczeniami wyobraźni i percepcji różnych kategorii uczestników procesu projektowego. Wydaje się, ze dla większości celów jest to kompromis bardzo rozsądny.

4 Prace cytowane

[BCN92] C. Batini, S. Ceri, S.B. Navathe. **Conceptual Database Design: An Entity-Relationship Approach.** The Benjamin/Cummings Publishing Company, Inc., 1992

[Booc94] G. Booch. **Object-Oriented Analysis and Design with Applications**. Redwood City, CA, Benjamin/Cummings 1994

[BJR98] G. Booch, I. Jacobson, J. Rumbaugh. **The Unified Modeling Language User Guide** (The Addison-Wesley Object Technology Series), Addison-Wesley, 1998

[Cant98] M. Cantor. **Object-Oriented Project Management With UML**. John Wiley & Sons, 1998

[Cole+94] D. Coleman, et al. **Object-Oriented Development. The Fusion Method.** Prentice Hall, 1994

[CoYo91a] P. Coad, E. Yourdon. **Object-Oriented Analysis**. Prentice Hall, 1991.

[CoYo91b] P. Coad, E. Yourdon. **Object-Oriented Design**. Prentice Hall, 1991.

[DoHu98] P. Dorsey, J.R. Hudicka. **Oracle 8 Design Using UML Object Modeling**. Osborne McGraw-Hill, 1998

[DSWi98] D.F. D'Souza, A.C. Wills. **Objects, Components, and Frameworks With UML : The Catalysis Approach** (Addison-Wesley Object Technology Series), Addison-Wesley, 1998

[FiEy95] D.G. Firesmith, E.M. Eykhold**. Dictionary of Object Technology - The Definitive Desk Reference**. SIGS Books, New York, 1995.

[FSJ97] M. Fowler, K. Scott, I. Jacobson. **UML Distilled : Applying the Standard Object Modeling Language** (Addison-Wesley Object Technology Series), Addison-Wesley, 1997

[GoRu95] A. Goldberg, K.S. Rubin. **Succeeding with Objects. Decision Frameworks for Project Management.** Addison-Wesley, 1995.

[Jaco92] I. Jacobson. **Object-Oriented Software Engineering - A Use Case Driven Approach**. Addison-Wesley, 1992

[JBR99] I. Jacobson, G. Booch, J. Rumbaugh. **Unified Software Development Process** (Addison-Wesley Object Technology Series) Addison-Wesley, 1999

[Kruc98] P. Kruchten. **The Rational Unified Process**. Addison-Wesley, 1998

[Larm97] C. Larman. **Applying UML and Patterns : An Introduction to Object-Oriented Analysis and Design**. Prentice Hall 1997

[MaOd91] J. Martin, J.J. Odel. **Object-Oriented Analysis and Design.** Englewood Cliffs, NJ, Prentice Hall 1991

[MaOd94] J. Martin, J.J. Odell. **Object-Oriented Methods: A Foundation**. Prentice Hall, 1994.

[MaOd96] J. Martin, J.J. Odell. **Object-Oriented Methods: Pragmatic Considerations**. Prentice Hall, 1996.

[Mart93] J. Martin. **Principles of Object-Oriented Analysis and Design**. PTR Prentice Hall, Englewood Cliffs, New Jersey, 1993

[Meye97] B. Meyer. **Object-Oriented Software Construction**. Prentice Hall, 1997.

[Mull99] R.J. Muller. **Database Design for Smarties: Using UML for Data Modeling**. Morgan Kaufmann Publishers 1999

[OMG95] Object Management Group. **CORBA: The Common Object Request: Architecture and Specification**, July 1995, Release 2.0.

[OdFo98] J.J. Odell, M.Fowler. **Advanced Object-Oriented Analysis and Design Using UML** (Sigs Reference Library , No 12) SIGS Books/Cambridge Press, 1998

[Oest99] B. Oestereich. **Developing Software with UML** (The Addison-Wesley Object Technology Series), Addison-Wesley, 1999

[RoSc99] D.Rosenberg, K. Scott. **Use Case Driven Object Modeling with UML: A Practical Approach** (Object Technology Series) Addison-Wesley, 1999

[Rumb+91] J. Rumbaugh, et al. **Object-Oriented Modeling and Design.** Prentice Hall, Englewood Cliffs, New Jersey 07632, 1991

[RJB98] J. Rumbaugh, I. Jacobson, G. Booch. **The Unified Modeling Language Reference Manual** (Addison-Wesley Object Technology Series), Addison-Wesley, 1998

[SWJ98] G. Schneider, J.P. Winters, I. Jacobson. **Applying Use Cases : A Practical Guide** (Addison-Wesley Object Technology Series), Addison-Wesley, 1998

[Subi98] K. Subieta. **Obiektowość w projektowaniu i bazach danych**. Akademicka Oficyna Wydawnicza PLJ, Warszawa 1998

[Subi99] K. Subieta. **Słownik terminów z zakresu obiektowości**. Akademicka Oficyna Wydawnicza PLJ, Warszawa 1999

[UML97] **Unified Modeling Language**, version 1.0. Rational Software Corporation, 1997, http://www.rational.com

[WaKl99] J.B. Warmer, A.G. Kleppe. **The Object Constraint Language : Precise Modeling With UML** (Addison-Wesley Object Technology Series), Addison-Wesley, 1999

[Your+95] E. Yourdon, et al. **Mainstream Objects. An Analysis and Design Approach for Business**. Yourdon Press, Prentice Hall, 1995