ZESZYTY NAUKOWE POLITECHNIKI ŚLĄSKIEJ 1996

Seria: INFORMATYKA z. 31 Nr kol. 1348

Dariusz AUGUSTYN
Katarzyna STĄPOR

OBIEKTOWO ZORIENTOWANY SYSTEM ZARZĄDZANIA BAZĄ DANYCH O_2

Streszczenie. Artykuł prezentuje cechy obiektowo zorientowanego języka oraz obiektowego systemu zarządzania bazą danych w implementacji produktu O_2. W szczególności, przedstawiono w nim charakterystyczne w implementacji O_2 mechanizmy takie, jak prezentacja obiektu, sposób deponowania obiektów w bazie danych, system wyszukiwania oparty na obiektowym języku zapytań.

OBJECT-ORIENTED DATABASE MANAGEMENT SYSTEM O_2

Summary. The article presents an object-oriented language and an object-oriented database management system implemented in product O_2. In particular in this article mechanisms like an object presentation, storing objects in database, data searching using an object query language which are characteristic in O_2 system implementation are explained.

O_2 - DAS OBJEKTORIENTIERTE DATENBANKVERWALTUNGSSYSTEM

Zusammenfassung. Im Artikel wurden Merkmale einer objektorientierten Sprache und eines objektorientierten Datenbankverwaltungssystems am Beispiel des O_2 - Systems dargestellt. Insbesondere hat man folgende, für das O_2 - System charakteristische Mechanismen besprochen: die Darstellung eines Objektes, das Speichern von Objekten in der Datenbank und das Aussuchen, welches auf einer objektorientierten Abfragesprache basiert.

1. Wprowadzenie

Najbardziej rozpowszechnionym modelem danych jest model relacyjny, stosowany na ogół we współczesnych, komercyjnych systemach zarządzania bazami danych (ang. DBMS - Database Management Systems). Oprócz szeregu zalet i powszechnej akceptacji model ten posiada pewne ograniczenia. Narzuca on każdej relacji co najmniej pierwszą postać normalną (1NF), co oznacza konieczność odwzorowania przestrzeni obiektów dziedziny przedmiotowej na zbiór płaskich relacji. W takim podejściu zostaje najczęściej pominięta część semantyki, dotycząca struktury obiektu. W celu zrekonstruowania informacji dotyczących złożonego obiektu konieczne jest dokonanie szeregu operacji złączeń relacji względem klucza obcego. Dla wielu nowych zastosowań obejmujących między innymi systemy informacji geograficznej GIS, systemy komputerowego wspomagania projektowania i zarządzania, systemy wspomagania inżynierii programowania CASE pojawiła się potrzeba umożliwienia innej, lepszej reprezentacji bardziej złożonych struktur danych. Dla takich zastosowań relacje w 1NF są nieodpowiednie, gdyż nie umożliwiają reprezentowania w sposób naturalny obiektów świata rzeczywistego. Nowe zastosowania wymagają takich "relacji", których elementy krotek mogą nie muszą wartościami atomowymi, czyli mogą stanowić krotki, zbiory lub nawet relacje (tzw. *zagnieżdżone relacje*). Relacje takie nazwane są relacjami NF^2 (ang. Non First Normal Form relations). Modele przedrelacyjne oraz model relacyjny nie umożliwiają ponadto wyrażania w sposób jawny wszystkich związków między danymi. Model obiektowy jest wynikiem prowadzonych badań nad tzw. *modelami postrelacyjnymi*, czyli modelami o bogatszej semantyce niż model relacyjny.

W dziedzinie modelowania danych można dokonać następującego podziału:

- *modele oparte na wartości* (ang. value-based) np. model relacyjny,
- *modele oparte na obiekcie* (ang. object-based).

Podstawowa różnica między wymienionymi modelami polega na różnym sposobie reprezentacji danych:

- za pomocą zbioru wartości atrybutów (*modele oparte na wartości*),
- za pomocą tzw. stanu skojarzonego z obiektem (z jego identyfikatorem), tzn. stanu, który może ulegać zmianie, zaś sam obiekt (charakteryzowany przez identyfikator) istnieje od momentu konstrukcji do destrukcji (*modele oparte na obiekcie*).

W istniejących obecnie, eksperymentalnych *obiektowo zorientowanych systemach baz danych* (ang. object-oriented database systems, w skrócie OODBS) obserwuje się tendencje do zachowania zgodności z nowo powstałym, rozwijanym standardem Object Database Standard ODMG-93 (ang. ODMG - Object Database Management Group).

2. Podstawowe cechy systemów obiektowo zorientowanych

Pojęcie obiektowo zorientowany (ang. object-oriented, w skrócie o-o) znajduje zastosowanie w analizie i projektowaniu systemów informatycznych, w językach programowania, systemach baz danych i baz wiedzy. U podstaw podejścia obiektowo zorientowanego w wymienionych wyżej dyscyplinach leży pojęcie *obiektu* stanowiącego model (abstrakcyjną reprezentację) zarówno stanu (struktury albo danych), jak i zachowania wyróżnionego fragmentu rzeczywistości. *Zachowanie obiektu* jest określone przez zbiór akcji oddziałujących na obiekt, co reprezentowane jest w systemie informatycznym przez *metody*, stan zaś poprzez zbiór wartości wyróżnionych cech obiektu (w skrócie wartość obiektu).

Artykuł [4] stanowi jedną z wielu prób usystematyzowania wspólnych cech istniejących obecnie OODBS. W artykule tym do ogólnych, obowiązkowych cech, czyli takich, które system powinien wykazywać, aby zasługiwać na miano obiektowo zorientowanego, zaliczono:

– *Identyfikowalność obiektu* - reprezentacja *tożsamości* obiektu
Tożsamość obiektu polega na nadawaniu każdemu z obiektów unikalnego identyfikatora, który istnieje i jest zarezerwowany dla obiektu przez cały czas jego istnienia w bazie. Konsekwencją tego jest fakt, że istnienie obiektu i jego rozróżnialność są niezależne od wartości jego atrybutów (stanu).
– *Możliwość definiowania wartości strukturalnych*
Oprócz możliwości definiowania typów atomowych (np. integer, real, string itp.) istnieje również możliwość tworzenia wartości bardziej złożonych, posiadających pewną strukturę wewnętrzną tzw. wartości strukturalnych.
– *Hermetyczność*
Polega na przedstawieniu obiektu jako połączenia danych (wartości atrybutów) oraz zachowania - *metod* stanowiących zbiór operacji mogących zmieniać stan obiektu. Metody umożliwiają manipulowanie obiektami bez znajomości struktury tych obiektów (oraz kodu odpowiednich metod).
– *Reprezentacja typów i klas*
– *Możliwość reprezentacji hierarchii typów*, tzn. różnorodnych związków między elementami klas
– *Możliwość przeciążania definicji typów*
– *Kompletność obliczeniowa języka manipulacji danymi*, czyli możliwość wyrażania dowolnych funkcji matematycznych, logicznych, relacyjnych
– *Rozszerzalność o nowe typy danych*, definiowane od podstaw lub na drodze specjalizacji typów istniejących
– *Trwałość obiektów* (ang. persistency), umożliwiająca przechowywanie między kolejnymi sesjami użytkownika wszystkich lub wyróżnionych obiektów bazy danych
– *Zarządzanie pamięcią zewnętrzną*
Programista aplikacji powinien móc manipulować obiektami w swoim programie, tak jakby znajdowały się one w nieograniczonej pamięci wirtualnej.
– *Integracja języka programowania, w którym są pisane aplikacje z językiem manipulowania danymi w bazie danych*
W klasycznych systemach obsługi baz danych występuje na ogół niedopasowanie tych dwóch języków.
– *Odtwarzanie*
– *Współbieżność wykonywania transakcji*
– *Możliwość formułowania zapytań ad hoc* (implementacja języka zapytań).

3. Charakterystyka modelu danych w systemie O_2

System O_2 łączy w sobie obiektowo zorientowany język programowania oraz system zarządzania bazą danych.

3.1. Charakterystyka obiektowo zorientowanego języka w systemie O_2

Podstawowym językiem opisu i manipulowania danymi jest O_2C (język zbliżony do C z rozszerzeniami, wynikającymi z obiektowego charakteru systemu O_2).

W modelu danych systemu O_2 charakterystykę obiektu definiują:

- wewnętrzny, niewidoczny dla programisty identyfikator, którego unikalność zapewniona jest przez system,
- wartości (dane),
- zachowanie (zbiór metod).

Definicja struktury danych określa *typy* wartości danej. Wartości mogą być typu:

- atomowego (boolean, char, integer, real, string, bits)
- strukturalnego (ang. complex)

  Wartości tego typu są tworzone za pomocą następujących konstruktorów: krotki (tuple), zbioru (set), unikalnego zbioru (unique set), listy (list).
- klasowego

  Przykładowymi klasami (importowanymi z dostarczanej biblioteki O_2kit) są Date, Bitmap. Wszędzie gdzie semantyka języka zezwala na wykorzystanie nazwy typu, możliwe jest też wystąpienie nazwy klasy.

Konstruktor **tuple** umożliwia grupowanie danych różnych typów. Postać deklaracji typu jest następująca:

tuple (*a1: t1, ..., an: tn*) gdzie

a1,...,an - nazwy atrybutów,

t1,...,tn - nazwy typów (dopuszczalne typy atomowe, strukturalne, nazwy klas).

Przykładowo:

```
tuple (nazwa:  tuple(   nazwisko: string,
                        imię: string,
                        tytuł: string),
       wiek:   integer,
       płeć:   char,
       adres:  Adres)
```

Konstruktor **list** umożliwia tworzenie uporządkowanych kolekcji wartości lub obiektów. Postać deklaracji typu jest następująca:

list (*t*) gdzie:

t - nazwa typu.

Przykładowo: `list (integer)`.

list() oznacza pustą listę.

Konstruktor **set** umożliwia tworzenie nieuporządkowanej kolekcji wartości lub obiektów. Deklaracja typu jest następująca: **set** (*t*).

Przykładowo: set (real)

W kolekcjach stanowiących wartości typu utworzonego przez konstruktor **unique set** nie jest możliwe powtarzanie się tych samych elementów.

Każdy z typów posiada zespół dozwolonych operatorów z nim związanych. Poniżej podano wybrane, dozwolone operacje dla typu **set**:

- przypisanie,

  (Np. x = set (1, 2, 3))
- rozmiar,

  (count (x) zwraca liczbę elementów w zbiorze x)
- przynależność,

  ((p in x) posiada wartość true, jeśli p jest elementem zbioru x)
- operator dodawania do zbioru,

  (Np. x += set(p))
- operator iteracji

  (Np. for (p in x where *warunek*) {*instrukcje*})

gdzie x, p zmienne zadeklarowane następująco:

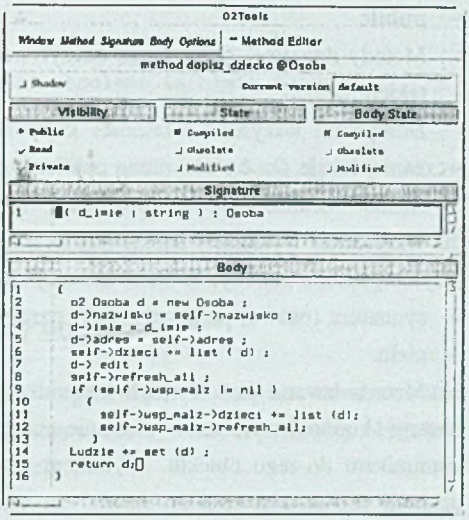
- o2 set (integer) x,
- o2 integer p.

(Deklaracja zmiennych wszystkich typów w języku O_2C poprzedzona jest słowem o2.)

Klasa stanowi abstrakcję reprezentującą zbiór obiektów o tej samej charakterystyce, na którą składają się dwie części:

- typ wartości charakteryzujący strukturę danych wszystkich wystąpień klasy (obiektów),
- zbiór metod umożliwiających wszelki dostęp do danych obiektów.

Obiekt jest charakteryzowany poprzez swoją klasę.



Rys. 1. Okno edytora metod podczas edycji metody *dopisz_dziecko*
Fig. 1. The window of methods editor invoked for *dopisz_dziecko* method

Przykład deklaracji klas:

```
class Adres
      type tuple (ulica: string,
                  miasto: string,
                  kod_poczt: string,
                  numer_domu: string)


class Osoba
     type tuple  (nazwa: tuple (nazwisko: string,
                                imię : string),
                  adres: Adres,
          public  zdjecie: Bitmap,
                  wiek: integer,
                  wsp_malz: Osoba,
                  dzieci: list (Osoba))
public method pokaz,
       method czy_jest_starszy
```

Atrybuty klasy (dane) oraz metody nazywane są własnościami klasy. Wyróżnia się trzy typy widzialności własności:

- **private**

  Tylko metody danej klasy oraz jej specjalizacji (podklas) mogą dokonywać modyfikacji takich atrybutów lub wykorzystywać taką metodę,
- **read**

  Metody dowolnej klasy mogą odczytywać wartości takich atrybutów, zaś modyfikacji tylko metody danej klasy lub podklas,

– **public**

Metody dowolnej klasy mogą odczytywać i modyfikować takie atrybuty oraz używać takich metod.

Domyślnie wszystkie własności klasy są prywatne. Przykładowo, w zdefiniowanej wcześniej klasie *Osoba* uczyniono publicznymi jedynie atrybut *zdjecie* oraz metodę *pokaz*.

Każda metoda stanowi procedurę związaną z typem i jest charakterystyczna dla wszystkich wystąpień tego typu. Wszelki dostęp do danych powinien odbywać się jedynie poprzez metody. W skład definicji dowolnej metody w języku O_2C wchodzą:

– sygnatura (nazwa, parametry, typy parametrów, typ zwracanej wartości),
– ciało.

Metoda zwana jest również komunikatem (dokładniej, będąca kodem realizującym obsługę komunikatu), zaś wywołanie metody danego obiektu nazywa się przesłaniem komunikatu do tego obiektu. Wynikiem przesłania komunikatu jest wykonanie pewnych operacji (zgodnie z kodem metody) na wartościach obiektu. W języku O_2C przesłanie komunikatu odbywa się w następujący sposób:

odbiorca_komunikatu -> *nazwa_metody* [(*argumenty*)]

gdzie *nazwa_metody* musi być metodą obiektu reprezentowanego odbiorcą. Możliwe jest również wywołanie metody nadklasy klasy danego obiektu:

odbiorca_komunikatu -> *nazwa_klasy@nazwa_metody* [(*argumenty*)].

Przykład metody (sygnatury oraz ciała) dla zdefiniowanej wcześniej klasy *Osoba* podano poniżej:

– sygnatura:

```
method public czy_jest_starszy (os1: Osoba): boolean in class Osoba
```

– ciało:

```
method body czy_jest_starszy (osl: Osoba): boolean in class Osoba
{ if (self->age > osl->age)
      return true;
  else
      return false;
}
```

gdzie *self* określa aktualny obiekt jako odbiorcę komunikatu.

3.2. Definiowanie związków między obiektami

Pomiędzy klasami można definiować różnego typu związki, stanowiące odwzorowanie odpowiednich logicznych powiązań, jakie występują w rzeczywistości pomiędzy obiektami tych klas. Służą temu różne mechanizmy abstrakcji.

Specjalizacja jako mechanizm abstrakcji jest to proces definiowania zbioru bardziej szczegółowych *podklas* S_i danej klasy obiektów C, zwanej *nadklasą*. Pomiędzy klasą nadrzędną C, stanowiącą nadklasę dla pewnej klasy S, zachodzi *związek nadklasa/podklasa* (ang. is_a, jest_jednym_z). Jeśli klasa posiada więcej niż jedną nadklasę, to nazywana jest *klasą dzieloną*.

Generalizacja jest procesem odwrotnym, w którym abstrahuje się od różnic pomiędzy różnymi klasami obiektów i biorąc pod uwagę tylko ich cechy wspólne uogólnia się różne klasy w jedną, zwaną nadklasą. Klasy uogólniane zwane są podklasami. Proces specjalizacji/generalizacji może być stosowany sukcesywnie definiując kolejne podklasy danej klasy, która sama stanowi podklasę. Jeśli w całej strukturze każda klasa bierze udział dokładnie w jednym związku is_a, to uzyskujemy tzw. *hierarchie*, w przeciwnym przypadku wobec istnienia klas dzielonych uzyskujemy sieć. Z procesem specjalizacji/generalizacji ściśle związany jest mechanizm *dziedziczenia*. Jest to mechanizm współdzielenia atrybutów oraz metod na podstawie istniejącego powiązania is_a, gdzie podklasy specjalizacji dziedziczą wszystkie własności nadklasy.

Powiązanie jest to abstrakcja umożliwiająca wyrażenie związku pomiędzy obiektami z kilku klas. Jego szczególnym przypadkiem jest agregacja, gdzie klasy składowe stanowią komponenty agregatu. Typ związku, jaki występuje między agregatem a jego komponentami, nazywany jest: *stanowi_część* (ang. is_a_part_of).

Przykład dziedziczenia:

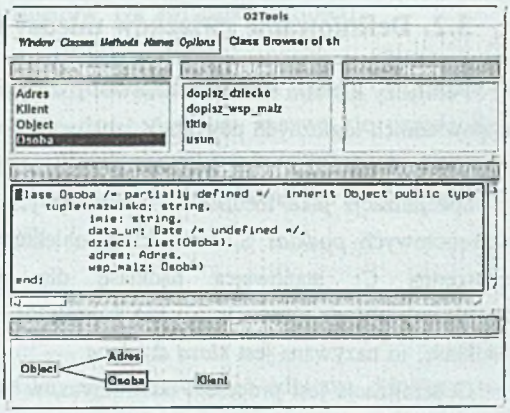
```
class Klient inherit Osoba
type tuple  (nazwa: string,
             stan_konta: integer)
method zmiana_stanu_konta
```

Klasa *Klient* dziedziczy całą charakterystykę (atrybuty i metody) zdefiniowanej wcześniej klasy *Osoba*. Oprócz tego posiada dwa dodatkowe atrybuty oraz jedną metodę.

Drugi typ związku pomiędzy klasami - powiązanie (agregacja), zilustrowany jest w przykładzie poprzez związek między klasą *Osoba* oraz *Adres*.

Możliwa jest redefinicja - czyli zmiana w podklasie metody dziedziczonej z nadklasy. Poniższy przykład dotyczy ponownego zdefiniowania metody *pokaz* w klasie *Klient*, stanowiącej podklasę klasy *Osoba*:

```
class Klient inherit Osoba
type tuple  (nazwa: string,
             stan_konta: real)
method zmiana_stanu_konta,pokaz
```

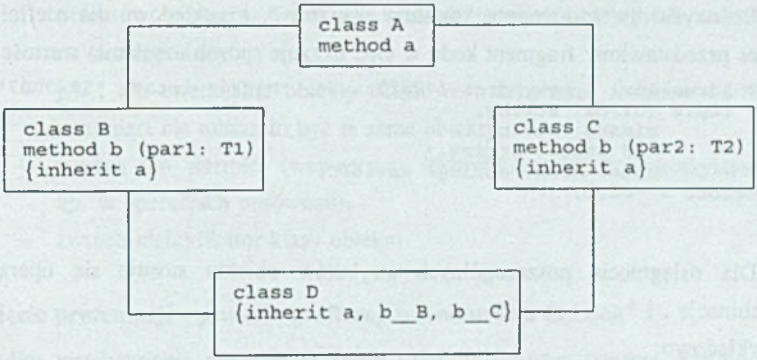


Rys. 2. Definicja klasy *Osoba* w oknie "Class Browser"
Fig. 2. Definition of class *Osoba* in "Class Browser" window

Wszystkie podklasy klasy *Klient* odziedziczą już oczywiście tę "nową" postać metody *pokaz*. Sygnatura metody redefiniującej musi być jednak kompatybilna z redefiniowaną (zgodność listy parametrów, typu wyniku). W trakcie wykonania na podstawie przynależności obiektu do danej klasy następuje zastosowanie odpowiedniej metody. Własność ta nazywa się późnym wiązaniem (ang. late binding). Możliwe jest również omawiane wcześniej jawne wywołanie metody nadklasy danej klasy, np. `obiekt_klient->Osoba@pokaz`, nazywane statycznym wiązaniem (ang. static binding).

3.3. Dziedziczenie wielokrotne

Każda klasa dziedziczy wszystkie własności swoich nadklas. W przypadku gdy w kilku nadklasach danej klasy pewna własność (atrybut, metoda) ma tę samą nazwę, powstaje konflikt. W systemie O_2 możliwe są dwa rozwiązania. Jeśli własności te pochodzą z różnych źródeł (klas), jak np. metoda *b* na rysunku 3, to system automatycznie nadaje im różne nazwy (*b_B* oraz *b_C*, co umożliwia nowej klasie - *D* stanowiącej podklasę klas *B* i *C* odziedziczenie wszystkich tych własności. Nazwy te mogą być oczywiście zmieniane później przez programistę. Jeśli metoda pochodzi z jednego źródła (np. metoda *a* z klasy *A* na rys. 3), tylko dziedziczona jest wzdłuż różnych ścieżek (z *A* poprzez *B* do *D* oraz z *A* poprzez *C* do *D*), to klasa *D* dziedziczy tylko jedną kopię metody (dziedziczenie wielokrotne). Jeśli jednak w klasach nadrzędnych (*B* i *C*) metoda *a* zostałaby przedefiniowana, to w klasie *D* metoda *a* musi zostać zdefiniowana lokalnie, uwzględniając zasady kompatybilności. Jeśli z pewnych względów jest to niemożliwe, oznacza to, że nie może być utworzona wspólna podklasa *D* dla danych nadklas *B* i *C*.



Rys. 3. Ilustracja wielokrotnego dziedziczenia z mechanizmami zmiany nazwy w przypadku tzw. dziedziczenia powtórzonego
Fig. 3. Multiple inheritance and changing names mechanism in case of repeated inheritence

3.4. Wybrane operacje na obiektach

Tworzenie nowych obiektów odbywa się poprzez wywołanie instrukcji **new**. Przykładowo wykonanie komendy: `adr1 = new Adres` powoduje utworzenie nowego obiektu klasy *Adres* i zainicjowanie wartości nowo utworzonego obiektu standardowymi wartościami (w ramach domyślnego wywoływania metody **init**). Użytkownik może oczywiście dokonać redefinicji metody **init**. Przykładowa definicja ciała metody **init** dla klasy *Adres*:

```
method body init (miasto: string, ulica: string) in class Adres
{ self->miasto= miasto;
  self->ulica = ulica;
}
```

Przykład pokazujący przekazywanie argumentów dla metody **init** przy tworzeniu obiektu klasy *Adres*:

```
adr1 = new Adres ("Katowice", "Kolejowa").
```

init jako jedyna metoda nie może być dziedziczona.

Do uzyskania wartości obiektu służy operator *. Przykładowo dla zdefiniowanej klasy *Adres* przedstawiony fragment kodu w O_2C ukazuje sposób uzyskania wartości obiektu:

```
{ o2 Adres adr1 = new Adres ("Warszawa", "Mickiewicza", "59-0007", "23/6");
  o2 tuple (ulica: string,
            miasto: string,
            kod_poczt: string,
            numer_domu: string) wartość;
  wartość = *adr1;
}
```

Dla osiągnięcia poszczególnych atrybutów obiektu stosuje się operator -> lub kombinację . i *.

Przykładowo:

```
{  o2 Adres adr1, adr2;
      adr1 = new Adres;
      adr2 = new Adres;
      adr1->miasto = "Katowice";
      (*adr1).ulica = "Asnyka";
      adr2->ulica = adr1->ulica;
}
```

3.5. Wybrane cechy implementacji modelu obiektowego systemu O_2

3.5.1. Podstawowe metody obiektu

Każda klasa w systemie O_2 stanowi podklasę klasy *Object*, posiadającej zestaw podstawowych metod, które w związku z tym są dziedziczone przez wszystkie klasy:

display – wyświetla wartość obiektu-odbiorcy komunikatu na ekranie (sposób prezentacji zależy od środowiska znakowego lub graficznego),

copy – zwraca kopię danego obiektu (inny obiekt o tej samej wartości; jeśli obiekt zawierał inne obiekty, to utworzona kopia zawiera wskazania na te obiekty),

deep_copy – j.w., ale jeśli obiekt zawierał inne obiekty, to utworzona kopia zawiera kopie składowych obiektów, tzn. utworzenie kopii danego obiektu pociąga za sobą utworzenie kopii obiektów składowych,

equal – zwraca *true*, jeśli odbiorca komunikatu i obiekt stanowiący argument wywołania metody posiadają te same wartości. Odbiorca i obiekt, będący argumentem metody w przypadku zawierania składowych obiektów, muszą posiadać jako wartości te same obiekty lub wskazania na te same obiekty,

deep_equal – j.w., ale ewentualne obiekty składowe muszą mieć te same wartości, natomiast nie muszą to być te same obiekty,

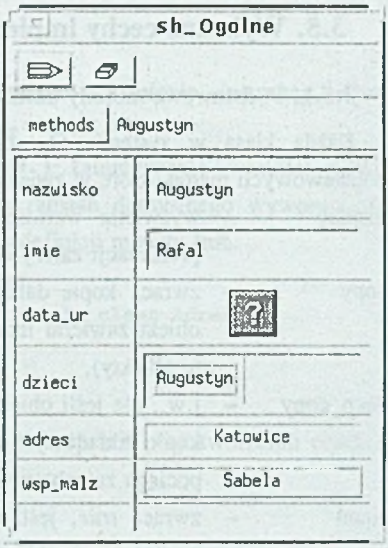
type_of – zwraca typ wartości (wewnętrzny identyfikator typu, wykorzystywany np. w operacjach porównań),

class_of – zwraca identyfikator klasy obiektu.

3.5.2. Pojęcie prezentacji - praca w graficznym środowisku O_2Look

W przypadku uruchomienia systemu w środowisku graficznym (system XWindows) każda wartość oraz obiekt niezależnie od swojego typu mogą zostać wyświetlone w okienku zwanym prezentacją. Prezentacja (np. utworzona poprzez wywołanie metody *edit)* obejmuje wyświetlenie wartości atrybutów obiektu wraz z menu, będącym listą metod publicznych obiektu (rys. 4.). Użytkownik ma możliwość wywołania tych metod (poprzez wybór z listy menu). Jeśli w prezentacji występuje inny obiekt jako komponent danego obiektu, to posiada on swoją własną prezentację. W ramach prezentacji możliwy jest zapis dokonanych zmian (ikona ołówka) lub zaniechanie (ikona gumki).

Przy tworzeniu programu dla środowiska graficznego z wykorzystaniem biblioteki O_2Look, wykorzystywane jest pojęcie prezentacji (zmienna typu *Lk_presentation*) w sensie kontekstu wyświetlania. Prezentacja w ten sposób rozumiana służy do określenia sposobu obsługi ekranu poprzez: ustawienia maski dotyczącej sposobu obsługi poszczególnych wartości obiektu, deklaracji postaci menu dostępnych metod obiektu, którego prezentacja dotyczy itp.



Rys.4. Okno prezentacji dla obiektu z klasy *Osoba*
Fig. 4. Presentation window for an object from class *Osoba*

W związku z dziedziczeniem metod z nadklasy *Object*, domyślnie dla każdej klasy w graficznym środowisku O_2Look dostępne są następujące metody:

edit – wyświetla wartości obiektu (podobnie jak display); umożliwia edycję wartości i aktywację wybranych metod; w zależności od wyboru użytkownika "Pencil"-Zatwierdź (aktualizacja stanu obiektu), "Eraser"-Odwołaj zwraca wartości odpowiadające LK_SAVE, LK_ERASE,

present – buduje nową prezentację obiektu; nie wyświetla wartości; zwraca identyfikator prezentacji, który może być wykorzystywany później do:
- określania sposobu obsługi poszczególnych wartości obiektu,
- określania parametrów okna wyświetlającego,
- wykonania samego wyświetlenia (funkcje *lk_map*, *lk_grab*),
- aktualizacji obiektu na podstawie nowych wartości wyświetlanych na ekranie...,

consult – nadaje nową wartość obiektowi zgodnie z wartościami związanymi z prezentacją, której identyfikator stanowi parametr metody,

refresh_all – wykonuje "odświeżenie" wszystkich prezentacji (obrazów na ekranie), związanych z danym obiektem,

erase_all – usuwa wszystkie prezentacje obiektu,

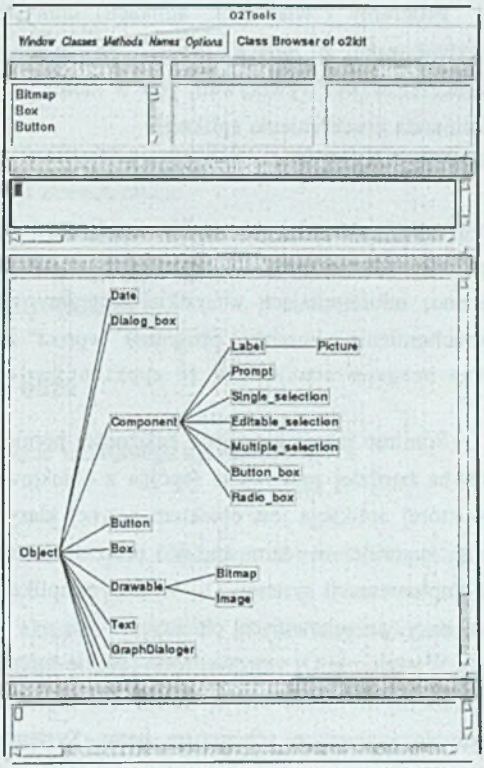
rubout – ustawia na wartość *nil* wszystkie prezentacje wartości obiektu,

all_presentation – zwraca listę prezentacji (identyfikatorów) danego obiektu w kontekście aktualnego stanu wykonania aplikacji,

disable_method,
enable_method – zezwala/zabrania wykonania metody, której nazwa stanowi parametr wywołania. Nazwy nieaktywnych metod są wyciemnione na liście menu dostępnych (publicznych) metod.

W systemie O_2Look wykorzystuje się metody wewnętrzne, których nie można jawnie wywołać w ramach aplikacji czy własnych metod "użytkownika", ale można dokonać ich redefinicji. Są to metody: *save, erase, title, bitmap, menu* (nazwy metod komentują zdarzenia, dla których są one aktywowane przez system).

Dostarczana przez producenta biblioteka O_2Kit stanowi zespół użytecznych klas (rys. 5). Mogą one być widoczne w schemacie każdej bazy po wykonaniu operacji importu.



Rys. 5. Hierarchia klas z biblioteki O_2Kit
Fig. 5. Class hierarchy from library O_2Kit

3.5.3. Aplikacje, programy, funkcje

W implementacji systemu O_2 aplikacja składa się ze zbioru programów. Przykładowo:

```
application obsluga_instytutu
  program  init
           public statystyka,
           public zmiany_masowe,
           exit
end
```

Aplikacje nie są związane z klasami/obiektami.

Programy ("własności" aplikacji) domyślnie są prywatne, tzn. mogą być jedynie wywoływane z innych programów danej aplikacji. Program *init/exit* (jeśli został zadeklarowany) wywoływany jest w momencie uruchomienia/zakończenia aplikacji. Jeżeli komenda uruchomienia aplikacji:

```
run application obsluga_instytutu
```

wykonana została w trybie graficznym w środowisku O_2Look (i nie został przedefiniowany program *dashboard*), wyświetlone zostanie okno prezentacji, zawierające menu, udostępniające wszystkie programy typu public. Istnieje oczywiście możliwość uruchomienia wybranego programu "wprost" z "wnętrza" aplikacji, np.:

```
run program statystyka in application obsluga_instytutu.
```

Pomimo jasnej koncepcji zależności pomiędzy aplikacją i programem w systemie O_2, chyba bardziej przejrzysta (spójna z obiektowego "punktu widzenia") byłaby koncepcja, w której aplikacja jest obiektem pewnej klasy, zaś programy są metodami tej klasy (tak jak sugeruje to sam sposób prezentacji aplikacji). Zastosowana koncepcja wynika z implementacji systemu O_2, w którym aplikacja i programy należą do schematu, nie zaś do bazy, przechowującej obiekty.

W systemie O_2 wprowadzono pojęcie funkcji, w znaczeniu kodu, który nie dotyczy ani konkretnych metod ani aplikacji. Funkcje definiowane przez użytkownika związane są jedynie z pewnym schematem bazy. System O_2 udostępnia podstawowy zbiór funkcji "standardowych" (niezależnych od schematu), np.: *display* (wyświetlenie: wartości, atrybutów obiektu ...), *input*, *o2query* (zanurzanie zapytań O_2SQL w kodzie O_2C), *o2_graphics* (przełączanie trybu: znakowy/graficzny), *o2_in_transaction* (test trybu: transaction mode albo read-only transaction mode), *type_of*.

3.5.4. Wielowersyjność

Metody, programy i funkcje mogą występować w wielu wersjach (mogą posiadać tzw. nazwane wersje). (Jeżeli przy tworzeniu metody, funkcji czy programu nazwa metody nie jest podana, domyślnie przyjmowana jest nazwa "default"). Domyślnie uruchamiana jest wersja ostatnio skompilowana (najbardziej aktualna). Oczywiście, istnieje możliwość wywołania metody, programu, funkcji poprzez jawne podanie nazwy wersji. W języku O_2C zaimplementowano instrukcje do manipulacji wersjami, np:

```
display method body nazwa_metody version nazwa_wersji
edit program body nazwa_programu version nazwa_wersji
display versions method nazwa_metody
display versions program nazwa_programu
```

Mechanizm wielowersyjności tworzy środowisko ułatwiające adaptację systemu do dynamicznie zmieniających się specyfikacji zewnętrznych.

4. System zarządzania bazą danych

4.1. System, wolumin, schemat, baza

W systemie O_2 wprowadzone zostało pojęcie **nazwanego systemu**.

4.1.1. Fizyczna struktura systemu

System stanowi kolekcję woluminów. Wolumin jest fizycznym zbiorem w systemie plików. Typy woluminów:

- *catalog* "systemowy" (przechowujący aktualne położenie i rozmiary wszystkich woluminów, przechowujący listę schematów (rozdział 4.2.2) i baz należących do systemu),
- *log* (przechowujący obrazy transakcji, wykorzystywany przez mechanizm odtwarzania),
- *shadow* (przechowujący dane tymczasowe, wykorzystywany między innymi przy "dużych" transakcjach),
- *user* (co najmniej jeden, zawierający dane tworzone przez użytkownika, inicjalnie ładowany jedynie schematem z biblioteki O_2Kit).

Parametry każdego systemu:

- nazwa systemu,
- nazwa komputera, unix-owego host-a (na którym ma być uruchomiany jedyny proces serwera, obsługującego opisywany system),
- położenie woluminów,
- rozmiar woluminów (jednokrotna prealokacja obszaru dla woluminów *log*, *shadow*, możliwe dynamicznie zwiększanie rozmiaru dla woluminów *user*),
- domyślny tryb pracy serwera (wielodostęp, transakcyjne przetwarzanie),
- lista unix-owych użytkowników z uprawnieniami O_2 *super-user* (użytkownik bez ograniczonych uprawnień w danym, nazwanym systemie)

zawarte są w konfiguracjnym pliku o nazwie *system*.

Przykładowa postać komendy wydanej z poziomu shell-a, uruchamiającej proces serwera dla wybranego systemu:

```
o2server -system <nazwa_systemu>
```

oraz komendy rozpoczynającej sesję z wybranym systemem, uruchamiającej proces klienta:

```
o2 -system <nazwa_systemu>
```

Oczywiście, procesy klientów i serwera nie muszą być uruchamiane na tych samych komputerach sieci.

4.1.2. Logiczna struktura systemu

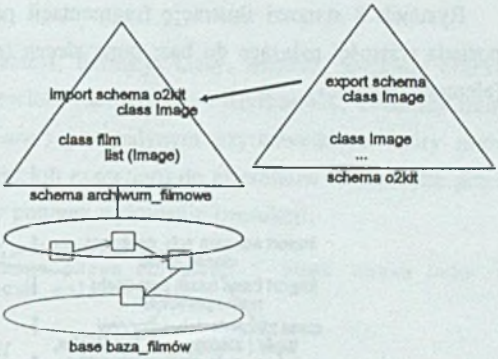
Z punktu widzenia logicznej struktury każdy system składa się ze schematów i baz.

W O_2 "struktury baz danych", nazywane **schematami** (ang. schema), to definicje struktur i zachowań ewentualnych elementów bazy danych, tzn. definicje typów, klas z metodami, nazw (rozdział 4.3), aplikacji, programów, uprawnień.

Bazą danych nazywana jest "instancja" schematu, zawierająca obiekty i wartości, które posiadają cechy trwałości. Schemat może być związany z wieloma bazami, ale nie odwrotnie.

Poszczególne elementy schematu, np. definicje klas, mogą być eksportowane do innych schematów (rys. 6). Oczywiście, importowane klasy mogą być redefiniowane poprzez dalszą specjalizację w danym schemacie.

Oprócz importowania/eksportowania elementów schematu możliwe jest współdzielenie danych poprzez import/eksport wybranych nazw (nazwanych zbiorów, obiektów, wartości itp.). W celu zapewnienia możliwości manipulowania danymi związanymi z importowaną nazwą konieczny jest również import odpowiednich struktur ze schematu.



Rys. 6. Import klasy *Image* z biblioteki O_2Kit
Fig. 6. Import of class *Image* from library O_2Kit

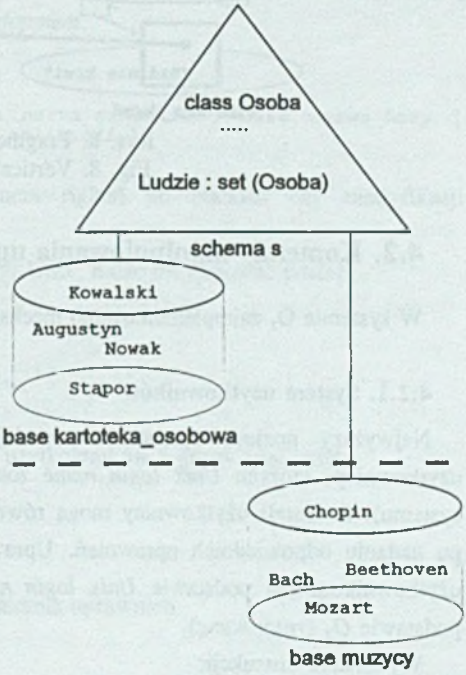
W następującym przykładzie:

```
schemat_A: name Z: set(K)
baza_A:    (obiekty ze zbioru Z)
schemat_B: import schema_A class K
baza_B:    import base A name Z
```

ewentualna aplikacja ze schematu *schemat_B*, uruchomiona na rzecz bazy *baza_B*, może operować (dostępność metod klasy *K*, pochodzącej ze schematu *schemat_A*) na obiektach za zbioru *Z*, tzn. danych z bazy *baza_A*.

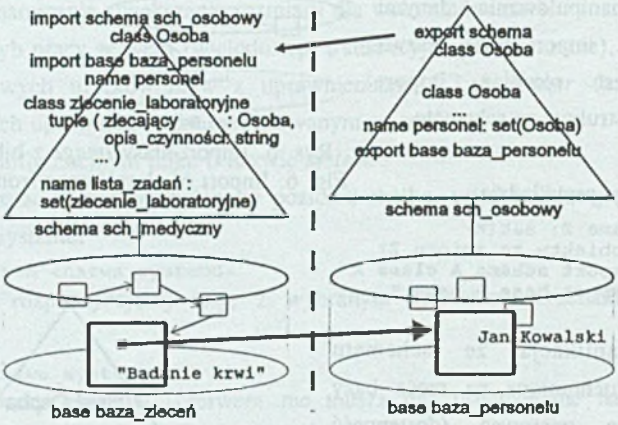
W O_2 zapewniona jest możliwość fragmentacji pionowej i poziomej (w ramach danego systemu).

Rysunek 7 stanowi ilustrację fragmentacji poziomej, w ramach której danemu schematowi może odpowiadać wiele baz. W przykładzie z rysunku 8 możliwe jest manipulowanie obiektami należącymi do jednego zbioru nazwanego (zdefiniowanego w jednym schemacie), obiektami, których składowe należą do jednej lub drugiej bazy (opartej na tym samym schemacie).



Rys. 7. Fragmentacja pozioma
Fig. 7. Horizontal fragmentation

Rysunek 8 stanowi ilustrację fragmentacji pionowej - każdy obiekt ze zbioru zleceń posiada wartości należące do baz: *baza_zlecen* (element *opis_czynności*) i *baza_personelu* (element *zlecajacy*).



Rys. 8. Fragmentacja pionowa
Fig. 8. Vertical fragmentation

4.2. Komendy manipulowania uprawnieniami

W systemie O_2 zaimplementowano mechanizm sterowania prawami dostępu.

4.2.1. System użytkowników

Najwyższy poziom uprawnień posiadają użytkownicy typu O_2 super-user (tzn. użytkownicy, których *Unix login name* został podany na liście w definicji nazwanego systemu). Pozostali użytkownicy mogą również uzyskać dostęp do bazy danych dopiero po nadaniu odpowiednich uprawnień. Uprawnienia mogą być nadawane poszczególnym użytkownikom (na podstawie *Unix login name*) lub całym grupom użytkowników (na podstawie *O_2 group name*).

Wykonanie instrukcji:

```
group nazwa_grupy nazwa_użytk1 nazwa_użytk2 ...
```

tworzy grupę użytkowników.

4.2.2. Definicja właściciela elementu

Każdy wolumin, schemat, baza, aplikacja, definicja klasy, metody, nazwany obiekt, nazwana wartość posiada swojego właściciela (domyślnie – użytkownik, tworzący dany element). Właściciel (oprócz O_2 *super-user*) jest jedynym użytkownikiem, który może zmieniać prawa dostępu (typu *development* lub *execution*) do elementów tworzonych przez siebie. Właściciel może zostać zmieniony poprzez wykonanie instrukcji:

```
modify owner { nazwa_grupy | nazwa_użytkownika }
in { volume nazwa_woluminu | schema nazwa_schematu | base nazwa_bazy |
class nazwa_klasy | application nazwa_aplikacji }
```

4.2.3. Uprawnienia typu *development*

Prawa tego rodzaju dotyczą elementów systemu mogących znajdować się w trybie tworzenia/modyfikacji, tzn. dotyczą elementów schematu (oraz baz z nim związanych). Domyślnie (bezpośrednio po utworzeniu) nie ma żadnych ograniczeń dla użytkowników, należących do tej samej grupy co właściciel. Pozostali użytkownicy (z wyjątkiem O_2 *super-user*) nie mają żadnych praw typu *development*.

Wykonanie instrukcji:

```
create | modify rights prawa
in { volume nazwa_woluminu | schema nazwa_schematu | base nazwa_bazy |
class nazwa_klasy | application nazwa_aplikacji }
```

umożliwia ustalenie praw (ang. development rights) do odczytu czy modyfikacji schematu, bazy, klasy czy aplikacji.

Fraza *prawa*, zgodnie z konwencją systemu Unix, może przyjmować postać:

{a|g|o|u} {+|–} {r|w}

gdzie:

- a – dotyczy wszystkich użytkowników,
- g – dotyczy grupy właściciela,
- o – dotyczy wszystkich z wyjątkiem użytkowników z grupy właściciela,
- u – dotyczy tylko właściciela,
- r – prawo odczytu,
- w – prawo zapisu,
- +|– – określa operację nadawania|odbierania uprawnień.

4.2.4. Uprawnienia typu *execution*

Wykonanie instrukcji:

```
create | modify rights nazwa_grupy ...
in { method nazwa_metody | program nazwa_programu }
```

pozwala ustalić prawa (ang. execution rights) do wykonywania danej **metody** czy **programu** w trybie interaktywnym. Przykładem pracy w trybie interaktywnym może być wywołanie programu/metody poprzez wybór nazwy z listy menu, wyświetlanego w oknie prezentacji aplikacji/obiektu. Domyślnie (bezpośrednio po stworzeniu programu, metody ...) żadne ograniczenia tego typu uprawnień dla żadnego z użytkowników nie są nadawane. Ewentualne uprawnienia mogą być nadawane jedynie na poziomie grup (poprzez O_2 *group name*).

Prawa typu *execution* związane z pracą w trybie interaktywnym mogą być również nadawane dla nazwanych obiektów i wartości poprzez wykonanie instrukcji:

```
create | modify rights {read | write | public} nazwa_grupy ...
in name nazwana_wartości_lub_obiekt.
```

Przykłady pracy w trybie interaktywnym, w którym uwzględniane byłyby prawa typu execution dla nazwanych obiektów:

- kierowanie zapytań do nazwanego zbiory obiektów w interaktywnej sesji O_2SQL,
- wywołanie bezpośrednio z poziomu programu *O2* funkcji *display*, wyświetlającej zawartość zbioru nazwanego.

4.3. Trwałość obiektów

W systemie O_2 obiekty mogą być trwałe (ang. **persistent**) albo tymczasowe.

Domyślnie wszystkie, nowo tworzone obiekty są tymczasowe i widoczne tylko w aplikacji (ich istnienie zależy od stanu wykonania aplikacji).

Wszystkie **nazwy** (np. nazwane obiekty, nazwane wartości, nazwane zbiory itp.) są trwałe oraz wszystkie komponenty trwałych obiektów są trwałe. Obiekt pozostaje trwały, jeżeli transakcja, w której został dołączony do nazwy, została wypełniona.

Definicje klas obiektów są trwałe.

Przykład nazwanej wartości:

```
name pi: real;
pi = 3.1415;
```

W poniższym przykładzie zapewnienie trwałości obiektu realizowane jest poprzez dołączenie go do nazwanego zbioru.

```
name Ludzie : set (Osoba);

method body dopisz_dziecko (imie: string) in class Osoba {
  o2 Osoba dziecko = new Osoba;
  self->dzieci = dziecko;
  dziecko->imie = imie;
  dziecko->nazwisko = self->nazwisko;
  dziecko->edit;
  transaction;
  Ludzie += set (dziecko);
  commit;
}
```

Usunięcie obiektu z bazy danych wymaga jego usunięcia z wszystkich źródeł, gdzie jest on zdeponowany w sposób trwały (odcięcie od wszystkich nazw). Przykładowo instrukcja `Ludzie -= set (osoba_x)` powoduje faktyczne usunięcie obiektu *osoba_x*, jeżeli należał on jedynie do nazwanego zbioru *Ludzie*.

4.4. Przetwarzanie transakcyjne

W systemie O_2 zaimplementowany jest mechanizm przetwarzania transakcyjnego, konieczny dla zachowania integralności bazy danych. W uproszczeniu działanie mechanizmu transakcyjnego polega na automatycznym wycofaniu modyfikacji w bazie danych przeprowadzonych przez niewypełnoną transakcję. Transakcją można nazwać wyodrębniony przez użytkownika fragmentu programu (wydzielona część akcji), modyfikujący bazę danych, który powinien być wykonany w całości. W przypadku przerwania wykonania w wyniku awarii systemu, akcji użytkownika, kolizji dostępu itp. rezultaty jego częściowego wykonania powinny być usunięte z bazy danych automatycznie (mechanizm odtwarzania - ang. rollback).

System może pracować w trybie:

- transakcyjnym/bez transakcji,
- z kontrolą wielodostępu/bez kontroli wielodostępu (tryb *single user*).

Wymienione tryby pracy, ustalane są na poziomie *O_2 server*-a i dotyczą wszystkich, ewentualnych sesji O_2 z tym serwerem.

W trybie transakcyjnym przewidziano możliwość pracy w dwóch podtrybach: *transaction mode* (inaczej *read-write transaction mode*) i *read-only transaction mode*.

W trybie *read-only transaction* nie można tworzyć, modyfikować czy usuwać trwałych obiektów. Natomiast oczywiście można tworzyć i modyfikować obiekty tymczasowe.

Podczas odczytu trwałych obiektów żadna blokada nie jest zakładana. Żadna współbieżna transakcja typu *read-write* poprzez blokadę w trybie wyłącznym nie może spowodować wstrzymania operacji odczytu dla zadań w trybie *read-only transaction*. Tryb *read-only transaction* zapewnia odczyt spójnych danych, tzn. danych potwierdzonych. W sytuacji żądania dostępu do danych objętych aktualnie transakcją typu *read-write* serwer dostarczy ostatnią wersję danych, sprzed rozpoczęcia transakcji.

Domyślnie, aplikacja utworzona z wykorzystaniem C, C++, O_2C startuje w trybie *read-only transaction*.

Zarządzanie otwieraniem i zamykaniem transakcji z poziomu aplikacji odbywa się poprzez wywołania instrukcji:

- **transaction** - start transakcji,
- **validate** - potwierdzenie modyfikacji, zdjęcie blokad nałożonych na obiekty modyfikowane (obiekty "pozostają" w pamięci, odpowiednie strony bazy danych nie są blokowane),
- **abort** - przerwanie transakcji, unieważnienie modyfikacji zrealizowanych w ramach bieżącej transakcji,
- **commit** - koniec bieżącej transakcji, potwierdzenie modyfikacji i zwolnienie pamięci dla zmiennych przechowujących kopie zmodyfikowanych obiektów trwałych.

W przypadku wykorzystania C, C++ należy zapewnić zwolnienie pamięci po wykonaniu **commit**.

Domyślnie, w ramach sesji O_2, O_2Tools system startuje w trybie transakcyjnym. Możliwe jest użycie dodatkowych instrukcji:

- **end transaction** - *validate* i przejście do trybu *read-only transaction*,
- **quit** - *commit* i zakończenie sesji O_2.

Różnica pomiędzy *validate* i *commit* polega na usunięciu z pamięci kopii obiektów w przypadku *commit*, albo pozostawieniu tych kopii w przypadku **validate**. Po **validate** obiekty trwałe modyfikowane w transakcji są dostępne dla innych użytkowników, a kopie dostępnych obiektów trwałych mogą się stać nieaktualne (w bieżącej aplikacji) ze względu na modyfikacje w innych transakcjach.

4.5. Język zapytań O_2SQL

W systemie O_2 zaimplementowano język zapytań O_2SQL, składniowo zbliżony do standardowego języka zapytań SQL (Structured Query Language), znanego z relacyjnych systemów baz danych.

Przykładowe najprostsze pytanie wyrażone przy pomocy języka O_2SQL może polegać tylko na podaniu nazwy zbioru (trwały, nazwany zbiór obiektów):

```
Ludzie
```

Przykład typowego, prostego pytania wyrażonego w języku O_2SQL, pozwalającego na wybranie wszystkich osób (obiektów) ze zbioru *Ludzie*, o nazwisku "Sabela":

```
select x
   from x in Ludzie
     where x->nazwisko = "Sabela"
```

Przykład pytania wyrażonego w języku O_2SQL, pozwalającego na wybranie wszystkich imion dzieci tych osób ze zbioru *Ludzie*, których nazwisko zaczyna się od liter "Kowal":

```
select dziecko->imie
   from dziecko in rodzic->dzieci and rodzic in Ludzie
     where rodzic->nazwisko like "Kowal%"
```

O_2SQL operuje zarówno na wartościach atomowych, jak i strukturalnych oraz obiektach. W celu przyspieszenia wykonania operacji wyszukiwania wyrażonej poprzez język O_2SQL serwer bazy danych wykorzystuje indeksy. Tworzenie indeksu może być zrealizowane poprzez wykonanie instrukcji:

create index { *nazwany_zbiów* | *nazwana_lista* } **on** *atrybut_kluczowy*.

O_2SQL może być wykorzystywany w ramach:

- sesji O_2 w trybie tekstowym, po wykonaniu komendy Query,
- sesji O_2Tools, po przejściu do trybu O_2Shell,
- fragmentu programu wyrażonego w języku O_2C, poprzez odpowiednie wywołanie funkcji o2query.

Przykład realizacji wywołania komendy języka O_2SQL z poziomu O_2C:

```
program body szukaj ( wzorzec : string ) in application Obsluga;
{
  o2 set (Ludzie) kto;
  o2query ( kto , "SELECT czlowiek FROM czlowiek in Ludzie
                   WHERE czlowiek->nazwisko like $1", wzorzec );
  display (kto);
}
```

Powyższy program pozwala na wyświetlenie osób należących do zbioru *Ludzie* o nazwisku zgodnym z parametrem *wzorzec*.

5. Opis modułów pakietu O_2

Aplikacja współpracująca z obiektową bazą danych może być utworzona w oparciu o języki: C, C++, O_2C (z wykorzystaniem biblioteki "okienkowej" O_2Look), gdzie O_2C to (podstawowy w systemie O_2) zorientowany obiektowo, zbliżony składniowo do C, język należący do klasy 4GL.

W skład systemu O_2 wchodzą między innymi następujące produkty:

- O_2 – program przeznaczony do projektowania, tworzenia, uruchamiania aplikacji i obsługi obiektowo zorientowanej bazy danych O_2; podstawowy program otwarcia sesji z wybranym systemem; program domyślnie startujący w trybie znakowym z możliwością przełączenia się w tryb O_2Tools,
- O_2Tools – podstawowe, kompletne, graficzne środowisko przeznaczone do projektowania, tworzenia, uruchamiania aplikacji oraz obsługi obiektowo zorientowanej bazy danych O_2,
- O_2Server – program serwera bazy danych (O_2Engine); serwer stron baz danych,
- O_2Monitor – program przeznaczony do monitorowania pracy serwera bazy danych (operacje typu: wyświetlenie listy aktywnych użytkowników, opisu otwartych transakcji itp.),
- O_2Kit – biblioteka użytecznych klas (użyteczny schemat, używając terminologii O_2),
- O_2API – interfejs programisty, biblioteka funkcji dostępu do obiektowej bazy danych z poziomu C, C++,
- O_2Look – biblioteka (interfejs programisty), wykorzystywana przy tworzeniu graficznego interfejsu użytkownika,
- O_2Graph – środowisko i interfejs programisty, wykorzystywany przy tworzeniu prezentacji danych w formie grafów.

6. Podsumowanie

Obiektowo zorientowane języki programowania na obecnym etapie rozwoju systemów informatycznych zdobyły sobie uznanie. Są powszechnie stosowane nawet w "systemach relacyjnych", tzn. są używane do tworzenia *front-end*-ów w systemach klient-serwer, z serwerem bazy danych, pracującym w oparciu o relacyjny model danych. Obiektowe systemy zarządzania bazami danych zyskują dopiero popularność. Na szczególną uwagę w dziedzinie propagowania i standaryzacji obiektowych systemów zarządzania bazami danych (dokument ODMG-93) zasługują prace grupy ODMG (ang. Object Database Management Group).

O_2 stanowi nowoczesny i rozwijający się system zarządzania obiektową bazą danych. System ten wchodzi już w fazę zastosowań komercyjnych.

Obiektowa semantyka rzeczywistości sprawia, że należy sądzić, iż przyszłość baz danych należy do sytemów obiektowych takich jak O_2. Dalsze kierunki rozwoju systemu dotyczą optymalizacji zapytań czy integracji z usługami internetu (np. dostęp do obiektowej bazy danych za pomocą serwisu WWW), ale przede wszystkim integracji z innymi sytemami obiektowymi.

LITERATURA

[1] O_2Technology: O_2User Manual.

[2] O_2Technology: O_2Tools User Manual.

[3] Abiteboul S., Fisher P.C., Schek H.J. (eds): Nested relations and complex objects in databases, Lecture Notes in Computer Science 361, Springer-Verlag 1989.

[4] Atkinson M. et. al.: The Object-Oriented Database System Manifesto, w [8].

[5] Beeri C.: Formal Models for Object-Oriented Databases, w [8].

[6] Hull R., King R.: Semantic database modeling: survey, applications and research issues, ACM Comput. Surv., 19, 3 (1987).

[7] Kim W., Nicolas J.M., Nishio S. (eds): Deductive and Object-Oriented Databases, Proceedings of the First International Conference on Deductive and Object-Oriented Databases, (Kyoto, 1989), Elsevier Science Publishers B.V., Amsterdam 1990.

[8] Peckham J., Maryanski F.: Semantic Data Models, ACM Comput. Surv., 20, 3 (1988).

[9] Sernadas A. Sernadas C., Ehrich H.D.: Object-oriented specification of databases: an algebraic approach, w: Proc. 13th Internat. Conf. on Very Large Databases, Stocker P.M, Kent W., eds., (Brighton 1987).

Recenzent: Dr hab. inż. Stanisław Wołek

Wpłynęło do Redakcji 22 czerwca 1996 r.

Abstract

The article presents an object-oriented language and an object-oriented database management system implemented in product O_2. Principles of object-oriented database management are described. In particular in this article mechanisms like an object presentation, storing objects in database (making an object persistent), transaction processing, data searching using an object query language, rights system management which are characteristic in O_2 system implementation are explained.