O modelach danych i innych rzeczach, które utrudniają życie informatykom

Krzysztof Goczyła
Politechnika Gdańska
e–mail: kris@pg.gda.pl

Abstrakt. W artykule przedstawiono różne podejścia do modelowania systemów realizowanych w oparciu o bazy danych. Zaprezentowano klasyfikację aplikacji bazodanowych z punktu widzenia stosowanego modelu danych. Skoncentrowano się na nowej klasie aplikacji, jaką są aplikacje uniwersalne, oraz na nowej klasie systemów DBMS, jaką są systemy obiektowo-relacyjne i systemy uniwersalne. Sformułowano zasadnicze wymagania stawiane tym systemom oraz dokonano porównania dostępnych na rynku systemów tego typu. Prezentowane modele danych porównano na przykładzie hipotetycznego projektu informatycznego. Na tym przykładzie pokazano, na jakie aspekty projektu informatycznego ma wpływ przyjęty model danych.

1. Wprowadzenie

Model danych to zestaw pojęć używanych do opisu świata rzeczywistego. Jest to swego rodzaju metajęzyk, w którym analityk systemu formułuje swoją wizję systemu informatycznego. Zastosowany model danych w istotny sposób wpływa na przebieg projektu informatycznego oraz na jakość efektu, jakim jest system informatyczny oparty na bazie danych. Stąd też niezmiernie ważne jest, jaki model danych zastosowano w projekcie informatycznym i w jaki sposób zostanie od przełożony na język implementacyjny.

Aktualnie, we współczesnych systemach baz danych stosowane są różne modele danych. Z grubsza biorąc, odpowiadają one kolejnym edycjom norm językowych, czyli edycjom języka SQL w wypadku systemów relacyjnych baz danych oraz edycjom języka OQL w wypadku systemów obiektowych. Niewątpliwie dominującą pozycję na rynku zajmują systemy oparte na modelu relacyjnym, czy też – ściślej mówiąc – na różnych wariantach tego modelu. Od czasu sformułowania pierwszych norm dotyczących systemów relacyjnych [1] model relacyjny ulega jednak stopniowej ewolucji, zmierzającej do rozszerzenia sztywnych zasad relacyjnych o możliwości zaczerpnięte z paradygmatu obiektowego. W tym artykule zajmiemy się konsekwencjami tej tendencji z punktu widzenia projektantów systemów informatycznych opartych na bazach danych.

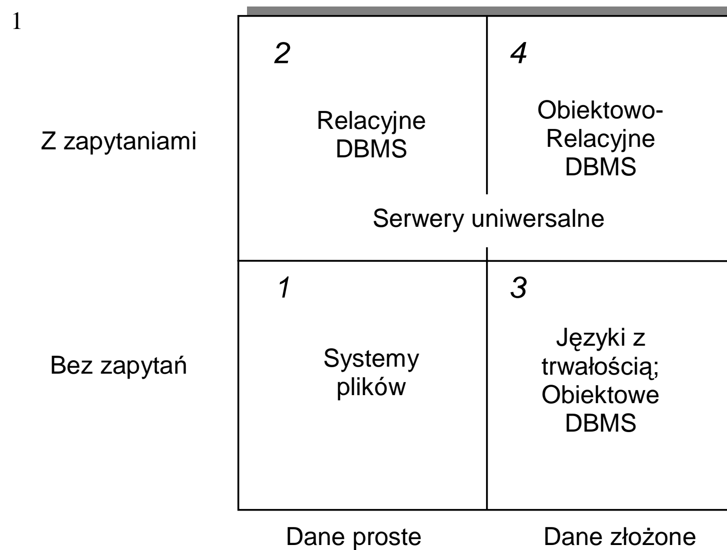
2. Klasyfikacja aplikacji bazodanowych

Bardzo istotnym elementem każdego projektu informatycznego, nie tylko takiego, który jest oparty na systemie baz danych, jest określenie wymagań projektowanej aplikacji w stosunku do środowiska sprzętowego i programowego, w szczególności – w stosunku do oprogramowania wspierającego wykonywanie danej aplikacji. Takim oprogramowaniem jest na przykład podstawowy system operacyjny, system obsługi grafiki, system obsługi plików, czy wreszcie system zarządzania bazą danych (DBMS, *Database Management System*). W zależności od rodzaju aplikacji, wymagania te mogą się znacznie różnić. Poniżej zaprezentujemy klasyfikację aplikacji bazodanowych z punktu widzenia ich wymagań w stosunku do systemów zarządzania bazami danych, wprowadzoną w [9] w formie tzw. „macierzy DBMS” (rys. 1).

Klasyfikacja ta ma charakter dwukryterialny. Pierwszym kryterium jest złożoność danych, na jakich operuje aplikacja. Wyróżnia się tu dwa rodzaje danych: dane proste i dane złożone. Dane proste to dane odpowiadające klasycznemu modelowi relacyjnemu, a więc spełniające wymóg pierwszej postaci normalnej (1NF). Są to takie dane, które nie składają się z innych danych (nie mają wewnętrznej struktury lub mają prostą strukturę o charakterze liniowym, bez zagnieżdżania).

Dane złożone natomiast to takie dane, w skład których wchodzą inne dane, tworząc w ten sposób dowolnie złożone, zagnieżdżone struktury danych (rekordy, tablice, zbiory, listy, drzewa itd.). Danym złożonym poświęcimy sporo uwagi w dalszej części tego artykułu.

Drugim kryterium klasyfikacji aplikacji w macierzy DBMS jest stosowanie w aplikacjach zapytań o charakterze deklaratywnym (a więc formułowanych w języku typu SQL lub OQL). Jest to kryterium ortogonalne w stosunku do kryterium złożoności danych: aplikacja może wymagać realizacji zapytań deklaratywnych niezależnie od tego, na jakich danych operuje.



Rys. 1. Macierz DBMS

Omówimy teraz pokrótce cztery grupy aplikacji odpowiadające czterem kwadratom macierzy DBMS z rys. 1.

2.1. Grupa 1: Dane proste bez zapytań

Ten rodzaj aplikacji w zasadzie nie wymaga stosowania systemu zarządzania bazą danych. Przykładem może być tu prosty procesor tekstu. Aplikacja taka otwiera wskazany plik dyskowy, umożliwia jego edycję (czyli aktualizowanie) oraz zapisuje zmodyfikowany plik z powrotem na dysku. Te operacje są realizowane przez system obsługi plików wchodzący w skład każdego systemu operacyjnego. Dane, na których operuje taka aplikacja, są danymi prostymi (ciągami znaków o dowolnej długości), w stosunku do których nie są formułowane żadne zapytania deklaratywne, wymagające zastosowania zaawansowanego mechanizmu wyszukiwawczego. Zauważmy, że bardziej wyrafinowane procesory dokumentów, wchodzące w skład pakietów do pracy grupowej (*groupware*) takich jak np. Lotus Notes, mogą się już nie kwalifikować do tej grupy, gdyż do zapisu dokumentów stosują złożone struktury danych.

Warto dodać, że systemy plików stanowią podstawę zarządzania danymi na poziomie fizycznym współczesnych DBMS (to jest na poziomie pamięci zewnętrznych), i z tego punktu widzenia są niezbędne dla wykonywania wszystkich aplikacji bazodanowych. Specjalizowane systemy plików stanowią zasadniczy składnik zarządców pamięci (*storage manager*) systemów DBMS. Oto niektóre systemy plików opracowane na potrzeby komercyjnych systemów DBMS: ISAM, CSAM, VSAM, WiSS.

2.2. Grupa 2: Dane proste z zapytaniami

Tę grupę stanowią aplikacje bazodanowe oparte na klasycznym modelu relacyjnym. Przez „klasyczny" model relacyjny rozumiemy tu model spełniający wymóg pierwszej postaci normalnej

(1NF), a więc model odpowiadający pierwszej normie języka SQL (SQL-89 [2]) i jej rozszerzeniu znanym jako SQL-92 [7]. W aplikacjach z tej grupy występują wyłącznie proste typy danych (dane znakowe, numeryczne, dane typu data/czas, duże dane binarne). Rozmiar takich prostych danych może być jednak bardzo duży i może sięgać wielu terabajtów na jedną bazę danych. Stąd też aplikacje z tej grupy wymagają silnych i sprawnych mechanizmów operowania na obszernych danych. Mechanizmów takich dostarczają systemy zarządzania relacyjnymi bazami danych (RDBMS), wzbogacone o cały wachlarz narzędzi wspomagających tworzenie aplikacji. Komercyjne systemy RDBMS stanowią dziś najbardziej rozwinięte technologicznie oprogramowanie do zarządzania dużymi wolumenami danych.

Rozwój technologii sprzętowej oraz nowe zastosowania baz danych stawiają przed producentami tych systemów coraz to nowe wyzwania, wykraczające poza typowe wymagania efektywnego i niezawodnego zarządzania danymi. Sprostanie tym wymaganiom to kwestia przetrwania na silnie konkurencyjnym rynku. A oto niektóre z tych nowych wymagań:

- Zdolność do pracy na różnych platformach, w tym wieloprocesorowych.
- Możliwość wymiany danych z RDBMS innych dostawców.
- Obsługa bardzo dużych wolumenów danych.
- Obsługa replikacji danych.
- Równoległa realizacja zapytań.
- Możliwość łagodnego przełączania się na serwer zapasowy w razie awarii.
- Optymalizacja realizacji zapytań typu OLAP dla przetwarzania danych hurtowych.

Aktualnie najwięksi dostawcy systemów typu RDBMS to Oracle, Informix, Sybase, IBM, Progress, Microsoft, Computer Associates. Wielkość rynku na systemy typu RDBMS ocenia się na ok. 10 mld USD rocznie i spodziewany jest dalszy stały wzrost tej sumy (o ok. 20% rocznie).

2.3. Grupa 3: Dane złożone bez zapytań

Grupę tę stanowią aplikacje, które wymagają znacznie bogatszego zestawu typów danych niż oferowany przez klasyczny model relacyjny. U źródeł poszukiwania nowych modeli danych, zapoczątkowanego w latach 80., legły nowe zastosowania komputerów, w szczególności w dziedzinach związanych z projektowaniem i przetwarzaniem danych o nieregularnej, często nie dającej się z góry określić strukturze [8]. Te dziedziny to m.in.: inżynieria oprogramowania (CASE, *Computer Aided Software Engineering,*), wspomaganie projektowania (CAD, *Computer Aided Design*), obsługa produkcji (CAM, *Computer Aided Manufacturing*), wspomaganie podejmowania decyzji (DSS, *Decision Support Systems*), wspomaganie prac wydawniczych (CAP, *Computer Aided Publishing*), automatyzacja prac biurowych, kartografia, obróbka danych multimedialnych, specjalizowane systemy informacyjne, sztuczna inteligencja i bazy wiedzy. W tego typu zastosowaniach klasyczny relacyjny model danych i relacyjne bazy danych wykazują wiele ograniczeń i niedogodności. Wśród nich najważniejsze to:

- zbyt prosty model danych, nieodpowiedni do modelowania struktur złożonych (np. zagnieżdżonych);
- ograniczony zestaw dostępnych typów danych i brak możliwości definiowania nowych;
- brak użytecznych koncepcji semantycznych, takich jak np. związki typu całość–część (agregacja) czy uogólnienie–uszczegółowienie (dziedziczenie);
- niedopasowanie pomiędzy językiem operowania danymi a językiem programowania aplikacji (*impedance mismatch*);
- model transakcji nieadekwatny dla długich transakcji, charakterystycznych na przykład dla procesów projektowania;

- brak pojęć reprezentujących czas i wersje;
- mała elastyczność w możliwościach modyfikowania schematu bazy danych.

Odpowiedzią na tego typu wyzwania stało się rozpowszechnienie paradygmatu obiektowego w projektowaniu i implementacji systemów informatycznych. Szczególnie rozwinięto języki i narzędzia programowania obiektowego (C++, Smalltalk, Java, VisualBasic i inne). Wraz z rozwojem takich języków pojawiły się systemy baz danych, w których model danych odpowiada modelowi danych zastosowanemu w tych językach, a ściślej – odpowiada modelowi definiowanemu przez Object Data Management Group [3], w tym – językowi OQL, stanowiącemu obiektowy odpowiednik SQL. W ten sposób powstały systemy zarządzania obiektowymi bazami danych (OODBMS, *Object-Oriented DBMS*), które do obiektowych języków programowania wprowadzają elementy trwałości, przetwarzania transakcyjnego i obsługi zapytań deklaratywnych.

Najistotniejszą z punktu widzenia pisania aplikacji cechą odróżniającą systemy obiektowe od innych systemów DBMS jest fakt, że takie same instrukcje języka programowania odnoszą się zarówno do obiektów trwałych (przechowywanych w bazie danych), jak i obiektów nietrwałych (programowych). Nie występuje więc efekt *impedance mismatch*, przejawiający się w potrzebie programowania aplikacji w dwóch językach: w zwykłym, proceduralnym języku programowania (3GL lub 4GL) i w języku dostępu do danych, czyli w języku SQL lub jego wersji „zanurzonej" (*embedded*) w zwykłym języku programowania. Efekt *impedance mismatch* powraca jednak, gdy z poziomu aplikacji obiektowej trzeba wystosować zapytanie do bazy danych, przechowującej obiekty trwałe. Wymaga to sformułowania zapytania w języku OQL i jawnego przekazania wyników do zmiennych aplikacji.

Mimo przewidywań formułowanych w pierwszej połowie lat 90., systemy OODBMS stanowią wciąż pewną niszę rynkową, o niezbyt szerokim zakresie zastosowań. Wynika to z tego, że nie są one technologicznie tak rozwinięte (i tak sprawdzone w praktyce) jak systemy RDBMS. Nie są w stanie sprostać współczesnym wymaganiom stawianym systemom baz danych, w szczególności dotyczącym skalowalności, niezawodności i efektywności przetwarzania zapytań odnoszących się do bardzo dużych wolumenów danych. Jest to też powód, dla którego w macierzy DBMS systemy tego typu zaliczane są do kategorii „bez zapytań". Brak jest również użytecznych narzędzi wspomagających tworzenie aplikacji dla tego typu systemów. Wielkość rynku systemów OODBMS szacuje się obecnie na zaledwie 1% rynku wszystkich systemów DBMS. Czołowi dostawcy OODBMS to Object Design, Objectivity, Versant, Computer Associates, Ardent.

2.4. Dane złożone z zapytaniami

Do tej grupy należą aplikacje, w których występują złożone struktury danych i które ponadto wymagają efektywnego operowania dużymi wolumenami informacji. W tej klasie mieszczą się więc te aplikacje wymienione w punkcie 2.3 (dane złożone bez zapytań), których potrzeby wykraczają poza definiowanie złożonych, nierelacyjnych struktur danych i wykonywanie na nich prostych operacji obliczeniowych. Wyobraźmy sobie system informacji przestrzennej typu GIS (*Geographical Information System*), w którym przechowywane są informacje o obiektach geograficznych położonych na dużym obszarze. Informacje te mają różny charakter i format: są to na przykład dane tekstowe, zawierające słowne opisy charakterystyk obiektów, i dane graficzne, mające postać fotografii. W takim systemie istotne jest wyszukiwanie obiektów według ich cech graficznych, takich jak kolor, charakter obrzeża, tekstura itp. (jest to wyszukiwanie typu QBIC, *Query By Image Content*). Potrzebna jest też wiedza na temat nakładania się na siebie poszczególnych obiektów, zawierania się obiektów w innych, określanie stopnia podobieństwa jednych obiektów do drugich itd. Wymagane jest więc nie tylko zastosowanie złożonych typów danych, ale także możliwość operowania na nich odpowiednimi funkcjami lub operatorami wbudowanymi w DBMS lub definiowanymi przez użytkownika. Ponadto potrzebny jest mechanizm wyszukiwawczy, który potrafi efektywnie wykonywać zapytania, w których występują typy i funkcje definiowane przez użytkownika.

Dla lepszego zobrazowania tych kwestii rozpatrzmy prosty przykład bazy danych dla takiego systemu GIS, przechowującej slajdy miejsc położonych w pewnym obszarze geograficznym:

```
create table Slajdy (ident    integer,
                     data     datetime,
                     opis     document,
                     obraz    photo);

create table Miejsca (nazwa         varchar(64),
                      położenie     point);
```

Pole opis tablicy Slajdy zawiera opis tekstowy z nazwą miejsca, z którego pochodzi dany slajd. Opis ten jest niestandardowego, strukturalnego typu o nazwie document. Sam slajd przechowywany jest w polu obraz tej tablicy o typie niestandardowym photo, przeznaczonym do wygodnej (odpowiednio skompresowanej) reprezentacji danych fotograficznych. Tablica Miejsca przechowuje nazwy (w polu nazwa standardowego typu znakowego varchar) i współrzędne geograficzne (w polu położenie niestandardowego typu point) miejsc, z których pochodzą slajdy. Taka baza danych może być bardzo użyteczna pod warunkiem, że mamy możliwość definiowania własnych funkcji operujących na niestandardowych typach oraz używania tych funkcji w zapytaniach. Załóżmy na przykład, że interesują nas wszystkie slajdy, na których zarejestrowano zachód słońca nad morzem w miejscach odległych od mola w Sopocie o nie więcej niż 10 km. Odpowiednie zapytanie może wyglądać na przykład tak:

```
select S.ident
    from Slajdy S, Miejsca M1, Miejsca M2
    where  sunset (S.obraz) and
           contains (S.opis, M1.nazwa) and
           distance (M1.polozenie, M2.polozenie) < 10 and
           M2.nazwa = 'Molo w Sopocie';
```

W tym zapytaniu użyto trzech funkcji zdefiniowanych przez użytkownika. Funkcja sunset potrafi określić, czy dane typu photo przedstawiają zachód słońca (czyni to na przykład na podstawie charakterystycznej kolorów i kształtów na fotografii). Funkcja contains sprawdza, czy opis typu document zawiera wskazany łańcuch znaków. Funkcja distance, określona na dwóch parametrach typu point, oblicza odległość między dwoma punktami geograficznymi, w kilometrach.

Przykład ten ilustruje dwie ważne cechy systemów spełniających wymogi aplikacji z grupy 4: definiowanie przez użytkownika własnych typów danych (w tym typów złożonych) oraz przypisywanie do tych typów funkcji. Takich możliwości nie mają tradycyjne relacyjne DBMS, oparte na języku SQL-92. Są one dopiero wprowadzone do języka SQL-99 (zwanego też SQL-3, [6]). Aplikacje typu „dane złożone z zapytaniami” wymagają zatem systemów DBMS nowej generacji, zwanych systemami obiektowo-relacyjnymi (ORDBMS). W dalszej części artykułu sprecyzujemy wymagania stawiane systemom obiektowo-relacyjnym. Krótko mówiąc, w takich systemach powinien być zaimplementowany dialekt języka SQL-3 w części dotyczącej typów złożonych i funkcji, optymalizacja złożonych zapytań w języku SQL-3 oraz silne narzędzia do prezentacji danych o złożonej strukturze (np. multimedialnych).

W macierzy DBMS warto wyróżnić jeszcze jedną grupę aplikacji, zwanych *aplikacjami uniwersalnymi*. Są to aplikacje wymagające zarówno typowych cech tradycyjnych, dojrzałych systemów relacyjnych (skalowalność, wydajność, niezawodność), jak i nowych cech systemów obiektowo-relacyjnych (elastyczność, bogactwo typów danych). Aplikacje te mieszczą się w grupie 2 i grupie 4 (można by więc je umieścić w grupie „dane proste i złożone z zapytaniami”). Analogicznie, serwery baz danych zdolne do obsługi tego typu aplikacji nazywane są serwerami uniwersalnymi (*Universal Server*). Pojawienie się na rynku serwerów uniwersalnych wynikło z przyjętej przez producentów strategii wzbogacania ich tradycyjnych systemów RDBMS o elementy obiektowo-relacyjne. Strategią tą jest albo stopniowe, ewolucyjne rozszerzanie funkcjonalności RDBMS (taką strategię przyjęły firmy Oracle i Computer Associates), albo połączenie serwera

relacyjnego z serwerem obiektowo-relacyjnym lub obiektowym (Informix, IBM i ponownie Computer Associates).

3. Systemy obiektowo-relacyjne

W tej części opiszemy podstawowe wymagania stawiane obiektowo-relacyjnym systemom zarządzania bazami danych z punktu widzenia modeli danych. Innego rodzaju wymagania to wymagania stawiane współczesnym komercyjnym systemom baz danych, wymienione w punkcie 2.2. Zakładamy, że pełnowartościowy system obiektowo-relacyjny musi spełniać wymagania z punktu 2.2. w sposób nie gorszy niż tradycyjne systemy RDBMS.

3.1. Rozszerzalność typów bazowych

Język SQL-92 przewiduje następujące typy danych dla kolumn tablic relacyjnych:

- liczby całkowite,
- liczby rzeczywiste stało– i zmiennopozycyjne,
- łańcuchy znakowe stałej i zmiennej długości,
- dane typu data, czas i przedział czasu.

Do operowania na danych tego typu przewidziano szereg standardowych operatorów i funkcji, jawnych (jak funkcje agregujące) i niejawnych (jak funkcje konwersji typów). Dla szeregu zastosowań te typy danych (nazwijmy je *typami bazowymi*) oraz operujące na nich funkcje nie są wystarczające, co sprawia, że pewne operacje trzeba wykonywać poza instrukcjami języka SQL, a potrzebne typy danych trzeba symulować na poziomie aplikacji. W efekcie programowanie aplikacji jest trudniejsze, a sama aplikacja wykonuje się mniej sprawnie. Nie można na przykład w pełni wykorzystać zalet konfiguracji klient-serwer, gdyż znaczna część operacji musi być wykonywana w procesie klienckim, co pociąga za sobą kosztowne przełączanie kontekstów i przesyłanie danych.

Systemy obiektowo-relacyjne oferują możliwość rozszerzania zestawu typów bazowych o typy definiowane przez użytkownika (UDT, *User-Defined Types*) i funkcje definiowane przez użytkownika (UDF, *User-Defined Functions*) W najprostszym rozwiązaniu, takie typy wraz z potrzebnymi do manipulowania nimi funkcjami i operatorami mogą być definiowane dla potrzeb jednej bazy danych lub aplikacji, z poziomu języka SQL, za pomocą instrukcji create type, create function, create operator itp. Rozpatrzmy poniższy przykład:

```
create type mojtyp_t
    (internallength = 8);

create cast (varchar to mojtyp_t)
    as mojTypInput;

create cast (mojtyp_t to varchar)
    as mojTypOutput;

create function mojaFunkcja (mojtyp_t Arg)
    returning float
    as external name 'fun'
    language C;

create table Mojatablica
    (nazwa        varchar (32),
     dane         mojtyp);
```

Zdefiniowano tu nowy typ bazowy o nazwie mojtyp, który zajmuje 8 bajtów pamięci. Wartości tego typu są wprowadzane i wyprowadzane za pomocą zdefiniowanych przez użytkownika funkcji konwersji (rzutowań): wejściowej mojTypInput i wyjściowej mojTypOutput. Ponadto zdefiniowano

funkcję mojaFunkcja, która jest funkcją zewnętrzną (a więc funkcją napisaną w języku innym niż SQL, w tym wypadku C). Następnie zdefiniowano tablicę bazy danych, w której użyto tego nowego typu. W stosunku do takiej tablicy można wykonać na przykład następujące instrukcje SQL:

```
update Mojatablica
    set dane = '12A307'
    where name = 'Ala';

select nazwa, dane
    from Mojatablica
    where mojaFunkcja(dane) < 0.7;
```

W pierwszej instrukcji zostanie w sposób niejawny zastosowane rzutowanie wejściowe mojTypInput. W drugiej instrukcji zostanie zastosowane (również niejawnie) rzutowanie wyjściowe mojTypOutput oraz funkcja mojaFunkcja. Wszystkie operacje zostaną zrealizowane przez serwer bez konieczności angażowania programu aplikacji.

Niektóre systemy ORDBMS mają też możliwość rozszerzania zakresu typów za pomocą modułów, które można dołączać do serwerów, rozszerzając w ten sposób ich funkcjonalność. Moduły takie zawierają nie tylko definicje typów, potrzebnych rzutowań, funkcji i operatorów, ale także metody dostępu odpowiednie dla tych niestandardowych typów danych (np. inne metody indeksowania). Informix dla oznaczenia takich modułów stosuje termin *BataBlade modules*, Oracle nazywa je *cartridges*, a IBM – *extenders*. Jako przykład takiego rozszerzającego modułu może służyć moduł DataBlade o nazwie 2–D Spatial w systemie Informix Dynamic Server with Universal Data Option (IDS–UDO), przeznaczony do obsługi dwuwymiarowych obiektów geometrycznych. Moduł ten definiuje nowy bazowy typ danych point oraz szereg funkcji użytecznych w przetwarzaniu figur na płaszczyźnie (m.in. funkcje distance, contained, overlaps, rectangle, circle itp.). Ponadto moduł 2–D Spatial zawiera indeksową metodę dostępu opartą na R-drzewie, które nieporównanie lepiej niż klasyczne B-drzewo nadaje się do indeksowania danych dwuwymiarowych.

3.2. Typy złożone

Obiekty złożone to obiekty, które składają się z innych obiektów typów bazowych, typów złożonych lub typów zdefiniowanych przez użytkownika. Jednym z najistotniejszych wymagań stawianym systemom obiektowo-relacyjnym jest obsługa przynajmniej następujących typów obiektów złożonych:

- typy wierszowe,
- typy kolekcyjne,
- typy referencyjne.

Typy te pozwalają wyjść poza sztywne ramy pierwszej postaci normalnej oraz wprowadzić identyfikowanie obiektów przechowywanych w bazie danych nie poprzez wartości atrybutów (klucze podstawowe), ale poprzez jednoznaczne identyfikatory. Jak nietrudno spostrzec, koncepcje te pochodzą z paradygmatu obiektowego.

Typ *wierszowy* składa się z pól danych dowolnego typu, zgromadzonych w jeden rekord; np.:

```
create type samochód_t
    (marka       varchar (24),
     rok_prod    integer,
     nr_rej      varchar(12));
```

Tak zdefiniowany typ można zastosować w definicji tablicy w następujący sposób:

```
create table Samochody of type samochód_t;
```

Tablica Samochody jest przykładem tablicy typowanej. Oczywiście, w jednej bazie danych może istnieć wiele różnych tablic typu samochód_t. Daje to efekt w postaci prostszego, bardziej

przejrzystego schematu bazy danych. Już z samej definicji tablic wynika bowiem, że przechowywane są w nich instancje (czyli wartości) tego samego typu. Tablice typowane pozwalają też na zastosowanie mechanizmu *dziedziczenia tablic*, opisanego w następnym punkcie.

Typ wierszowy można zastosować jako typ kolumny tablicy (typowanej lub nietypowanej). Załóżmy, że w bazie danych chcemy przechowywać dane o kierowcach i użytkowanych przez nich pojazdach samochodowych. Posłuży nam do tego nietypowana tablica Kierowcy1, w której zastosujemy wcześniej zdefiniowany typ samochód_t:

```
create table Kierowcy1
    (kierowca    varchar (36),
     pojazd      samochód_t);
```

Typ *kolekcyjny* umożliwia zgrupowanie wartości jednego typu w jeden zestaw o nieokreślonej z góry (teoretycznie nieograniczonej) liczności. W języku SQL-99 zdefiniowano 3 rodzaje konstruktorów typów kolekcyjnych: konstruktor set, odpowiadający zbiorowi bez powtórzeń, konstruktor multiset, odpowiadający zbiorowi z powtórzeniami, oraz konstruktor list, odpowiadający liście (uporządkowanemu zbiorowi z powtórzeniami). Zakłada się przy tym możliwość dowolnego (ortogonalnego) zagnieżdżania konstruktorów kolekcji (w celu tworzenia kolekcji złożonych z kolekcji dowolnego typu) oraz innych konstruktorów typów złożonych (np. w celu tworzenia kolekcji złożonych z wierszy). Daje to analitykowi i programiście aplikacji bazodanowych potężne narzędzie modelowania, pozwalające w sposób naturalny tworzyć stosowne schematy bazy danych dla nawet najbardziej złożonych struktur danych. Z uwagi na trudności implementacyjne, komercyjne ORDBMS nakładają jednak szereg ograniczeń na stosowanie konstruktorów kolekcyjnych (patrz tabela 1 i towarzyszące jej przypisy).

Jako przykład zastosowania typu kolekcyjnego rozpatrzmy rozwiniętą (i zarazem bardziej realistyczną) wersję bazy danych o kierowcach, w której jeden kierowca może użytkować wiele samochodów. Naturalnym rozwiązaniem jest tu zastosowanie typu kolekcyjnego set, składającego się z elementów typu wierszowego samochód_t:

```
create table Kierowcy2
    (kierowca    varchar (36),
     pojazdy     set(samochód_t));
```

Uważny analityk dostrzeże natychmiast pewną niedogodność tego schematu w sytuacji, gdy jeden pojazd może być użytkowany przez wielu kierowców. Wówczas dane o jednym pojeździe (a więc instancja typu wierszowego samochód_t) mogą wystąpić w tablicy Kierowcy2 wielokrotnie. Spowoduje to nie tylko marnotrawstwo pamięci (co może nie stanowić poważnego problemu), ale przede wszystkim problemy przy aktualizacji. Jeśli na przykład dany pojazd otrzyma nowy numer rejestracyjny, trzeba będzie dokonać stosownej modyfikacji w wielu miejscach. W takich sytuacjach użyteczny jest kolejny złożony typ danych — typ *referencyjny*, o konstruktorze ref.

Wartości typu referencyjnego można traktować jako odsyłacze do obiektów (czyli wierszy tablic) przechowywanych w bazie danych. Odsyłacze te mają użyteczne cechy: nie są modyfikowalne i nie są powtarzalne (przynajmniej w ramach jednej tablicy). Odpowiadają zatem pojęciu identyfikatora obiektu (OID, *Object Identifier*) w systemach obiektowych. W systemach relacyjnych pełnią rolę niejawnego klucza podstawowego każdej relacji, stąd też są nazywane *identyfikatorami wierszy*. Pojęcie referencji w systemach obiektowo-relacyjnych jest jednak znacznie szersze niż prostego identyfikatora wiersza. Typ referencyjny, jako normalny typ złożony, może być stosowany razem z innymi typami złożonymi: możemy więc stosować referencję do kolekcji, kolekcję referencji itd. Ponadto można stosować operację odwrotną do referencji, która zamienia odsyłacz na obiekt wskazywany.

Dla przykładu rozpatrzmy kolejną wersję bazy danych o kierowcach, w której dane o samochodach przechowywane są w oddzielnej tablicy typowanej Samochody. W tablicy przechowującej dane o kierowcach zastosowano kolekcję identyfikatorów, z których każdy odnosi

się do jednego samochodu. Ten schemat nie wykazuje już niedogodności związanych z aktualizacją danych o samochodzie.

```
create table Kierowcy3
    (kierowca    varchar (36),
     pojazdy     set(ref(samochód_t) references Samochody));
```

Zauważmy, że z konstruktorem ref konieczne jest użycie frazy references, gdyż może istnieć wiele tablic typu samochód_t.

Operatorem odwrotnym do ref jest deref (operator dereferencji). Umożliwia on zamianę identyfikatora obiektu na sam obiekt. Działanie obu operatorów ilustrują poniższe przykłady:

```
select ref (Samochody)
    from Samochody
    where nr_rej = 'GD 03473';

select deref (pojazdy).nr_rej
    from Kierowcy3
    where kierowca = 'Stefan Nowak';
```

Pierwsze zapytanie zwraca identyfikator samochodu o wskazanym numerze rejestracyjnym. Identyfikator ten może dalej służyć do wydobywania dowolnych informacji o tym samochodzie. Drugie zapytanie zwraca kolekcję numerów rejestracyjnych tych samochodów, które użytkuje wskazany kierowca. Zastosowano tu operator dereferencji do zamiany kolekcji referencji na kolekcję wierszy, a następnie operator kropki („.”)do wydobycia określonego pola wiersza. Zauważmy, że wydobycie tej informacji w instrukcji języka SQL-92 wymaga zastosowania jawnego złączenia dwóch tablic (jednej, zawierającej dane o kierowcach, i drugiej, zawierającej dane o samochodach) według równości kluczy obcych i kluczy podstawowych.

3.3. Dziedziczenie

Dziedziczenie, będące immanentną zasadą paradygmatu obiektowego, także w bazach danych [8, 10], polega na przejmowaniu cech (*atrybutów* i *metod*) jednych typów danych przez inne typy. Typ dziedziczący nazywamy podtypem, a typ, z którego dziedziczone są cechy, nazywamy nadtypem. Z reguły podtyp dziedziczy z nadtypu wszystkie jego cechy, dodając cechy nowe, nieobecne w nadtypie. Stąd też dziedziczenie jest w istocie związkiem typu uogólnienie-uszczegółowienie, zachodzącym pomiędzy typami danych. Stosowanie dziedziczenia upraszcza koncepcyjnie model systemu, pozwala na wielokrotne wykorzystywanie w różnych systemach raz zdefiniowanych elementów (*komponentów*), polepsza modyfikowalność i elastyczność rozwiązań projektowych.

W systemach obiektowo-relacyjnych baz danych dziedziczenie stosuje się do definiowania hierarchii typów danych i tablic typowanych. Zilustrujemy to następującym przykładem. Załóżmy, że wśród samochodów użytkowanych przez kierowców można wyróżnić samochody osobowe i samochody ciężarowe. Każdy samochód ma atrybuty występujące w poprzedniej definicji typu samochód_t, a ponadto atrybuty specyficzne dla rodzaju samochodu (na przykład liczba osób dla samochodu osobowego oraz ładowność i maksymalny nacisk na jedną oś dla samochodu ciężarowego). Możemy więc zdefiniować następującą hierarchię typów wierszowych:

```
create type samochód_t
    (marka       varchar (24),
     rok_prod    integer,
     nr_rej      varchar(12));

create type osobowy_t
    (ile_osób    integer)
under samochód_t;

create type ciężarowy_t
```

```
    (ładowność   integer,
     nacisk      integer)
under samochód_t;
```

Tak zdefiniowana hierarchia typów danych umożliwia zdefiniowanie odpowiadającej jej hierarchii tablic. Rolę *korzenia* w tej hierarchii pełni tablica Samochody.

```
create table Samochody of type samochód_t;

create table Osobowe of type osobowy_t
under Samochody;

create table Ciężarowe of type ciężarowy_t
under Samochody;
```

Zauważmy, że w razie potrzeby można łatwo rozbudować hierarchię typów i tablic, wprowadzając nowe podtypy istniejących typów i definiując odpowiednie dla nich tablice typowane.

Dla hierarchii tablic powinna istnieć możliwość określania zasięgu zapytania, tzn. określania tablic, których dotyczy dane zapytanie. Załóżmy na przykład, że interesują nas marki wszystkich pojazdów samochodowych, niezależnie od ich rodzaju. Ponieważ domyślnym zasięgiem zapytania jest tablica występująca we frazie from i wszystkie jej podtablice w jej hierarchii tablic, odpowiednie zapytanie wygląda następująco:

```
select distinct marka
    from Samochody;
```

Jeśli jednak interesują nas tylko takie samochody, które nie są ani osobowe, ani ciężarowe (np. motocykle) wówczas musimy ograniczyć zasięg tego zapytania wyłącznie do wierszy tablicy Samochody).

```
select distinct marka
    from only Samochody;
```

Pełnowartościowy system ORDBMS powinien dawać możliwość stosowania wielodziedziczenia, a więc sytuacji, w której jeden typ dziedziczy bezpośrednio z kilku nadtypów. W naszym przykładzie może zajść potrzeba przechowywania informacji o samochodach osobowo-ciężarowych, które mają wszystkie atrybuty zarówno samochodów osobowych, jak i ciężarowych. Ponadto mogą one mieć atrybuty specyficzne dla tej klasy pojazdów (np. rodzaj konstrukcji dla ładunków: skrzynia, nadbudowa, kontener itd.). Wcześniejszą hierarchię typów danych i tablic można w tym celu rozszerzyć w następujący sposób:

```
create type osob_cięż_t
    (rodzaj      varchar (18))
under osobowy_t, ciężarowy_t;

create table Osob_ciężarowe of type osob_cięż_t
under osobowe, ciężarowe;
```

Istnieje kilka sposobów implementacji hierarchii tablic [5]. Jeden sposób polega na przeznaczeniu oddzielnej tablicy fizycznej na każdą tablicę z hierarchii, przy czym w podtablicy przechowywane są jedynie te atrybuty, które nie występują w nadtablicy. Odpowiadające sobie wiersze z poszczególnych tablic identyfikowane są poprzez takie same identyfikatory wierszy: identyfikator będący kluczem podstawowym podtablicy jest jednocześnie kluczem obcym odpowiadającym kluczowi podstawowemu nadtablicy. Przy takim rozwiązaniu zapytania o zasięgu ograniczonym tylko do nadtablicy (tj. zapytania dotyczące atrybutów wspólnych dla wszystkich typów w hierarchii) wykonywane są najefektywniej, jednak wydobywanie pełnych informacji o instancjach podtypów wymaga realizacji złączeń. Tej wady nie ma rozwiązanie, w którym każdej tablicy z hierarchii odpowiada jedna tablica fizyczna, zawierająca wszystkie atrybuty jej typu danych. Jednak selekcja atrybutów wspólnych dla wszystkich typów w hierarchii wymaga przeglądnięcia wielu tablic i dokonania unii wyników. Możliwe jest jeszcze inne rozwiązanie: jedna

tablica fizyczna na wszystkie tablice z całej hierarchii. Takie rozwiązanie jest najefektywniejsze czasowo (nie są potrzebne operacje złączenia ani unii), jednak pociąga za sobą marnotrawstwo pamięci (wiersze mają różną liczbę atrybutów znaczących) i konieczność przechowywania w takiej tablicy dodatkowego atrybutu: typu danych w wierszu.

Zasady dziedziczenia odnoszą się nie tylko do danych, ale także do funkcji. Funkcja zdefiniowana na nadtypie działa także na wszystkich podtypach tego typu, chyba że dla danego podtypu zdefiniowano inną funkcję, która przesłania funkcję zdefiniowaną dla nadtypu. Funkcja przesłaniająca musi mieć przy tym sygnaturę (nazwę, liczbę parametrów i typ wyniku) zgodną z sygnaturą funkcji przesłanianej. Zilustrujmy to na naszej przykładowej hierarchii pojazdów samochodowych. Załóżmy, że dla typu samochód_t zdefiniowano funkcję określającą stawkę ubezpieczenia w zależności od marki i roku produkcji:

```
create function stawka (samochód_t Arg)
    returning float
    as ... ;
```

Wówczas zapytanie

```
select stawka(S)
    from Samochody S;
```

zwróci wielkości stawek ubezpieczeniowych dla wszystkich pojazdów przechowywanych w hierarchii tablic o korzeniu Samochody (czyli dla wszystkich wystąpień typu samochód_t i jego podtypów). Jeśli jednak zdefiniujemy jeszcze jedną funkcję stawka, określoną na typie osobowy_t, która inaczej oblicza stawkę ubezpieczeniową dla samochodów osobowych, wówczas powyższe zapytanie zwróci wartości wynikające z zastosowania odpowiedniej funkcji stawka do odpowiedniego typu danych wybieranych z tablic hierarchii o korzeniu Samochody. Taka własność systemu wykonawczego aplikacji, zwana *późnym wiązaniem metod*, jest typowa dla języków programowania obiektowego. Jest także wymogiem stawianym obiektowo-relacyjnym DBMS obsługującym dziedziczenie.

3.4. Reguły

Od obiektowo-relacyjnego systemu DBMS wymagamy silnego, uniwersalnego systemu reguł. Reguły są bardzo pożyteczną własnością systemów DBMS, gdyż znacznie upraszczają utrzymywanie integralności danych, wykonywanie czynności kontrolnych „w tle" itp., delegując je na poziom schematu bazy danych, a nie kodu aplikacji. Tradycyjne systemy RDBMS zwykle obsługują reguły poprzez mechanizm wyzwalaczy (*triggers*), jednak systemy ORDBMS wymagają silniejszego i bardziej elastycznego mechanizmu.

Ogólna postać reguły jest następująca:

```
create rule <nazwa_reguły>
   on <zdarzenie>
   where <warunek>
   do <czynność>
```

A zatem, w sytuacji zajścia określonego zdarzenia, przy spełnieniu podanego warunku, DBMS wykonuje wskazane czynności. Czynności te mogą być wykonywane zarówno przed, jak i po zajściu wskazanego w regule zdarzenia. Poniżej precyzujemy podstawowe wymagania stawiane mechanizmom reguł w systemach ORDBMS.

1. Zdarzenia i czynności mogą być zarówno aktualizacjami, jak i odczytami.

   We frazie on i we frazie do reguły mogą wystąpić dowolne ze słów kluczowych SQL: select, insert, update, delete, a więc dowolne rodzaje instrukcji operujących na bazie danych. Instrukcja występująca we frazie on nie ogranicza przy tym w żaden sposób instrukcji, jaka może wystąpić we frazie do, i na odwrót.

2. Mechanizm reguł musi być zintegrowany z cechami obiektowo-relacyjnymi.

W regule mogą wystąpić dowolne instrukcje języka SQL-99, w tym instrukcje zawierające typy i funkcje zdefiniowane przez użytkownika oraz typy złożone. Ponadto reguły mogą być dziedziczone: reguły zdefiniowane dla nadtablicy są automatycznie dziedziczone przez podtablice w hierarchii dziedziczenia (nie ma potrzeby ponownego definiowania reguł).

3. Mechanizm reguł musi być zintegrowany z mechanizmem obsługi transakcji.

   Jeśli zdarzenie inicjujące wykonanie czynności reguły wykonywane jest w ramach transakcji, również reguła wykonywana jest w ramach (zwykle tej samej) transakcji. Jeśli transakcja jest wycofywana, również czynność wykonana w ramach reguły jest wycofywana. Czasem takie postępowanie może być niepożądane. Przykładowo, jeśli w ramach reguły rejestrowane są wszystkie próby dostępu do pewnych danych, to próba, która została wycofana, nie zostanie zarejestrowana. Potrzebna jest więc możliwość jawnego specyfikowania, czy reguła ma być wykonywana w ramach tej samej czy oddzielnej transakcji.

   Innym aspektem związanym z obsługą reguł w transakcji jest konieczność odkładania wykonywania czynności reguły do momentu zakończenia transakcji. Wiadomo, że pomiędzy instrukcjami składającymi się na transakcję stan bazy danych może być nieintegralny. Może to powodować niepotrzebne inicjowanie wykonania reguły nadzorującej integralność. Potrzebna jest więc możliwość jawnego specyfikowania, czy reguła ma być inicjowana w trakcie transakcji, czy też jej ewentualne wykonanie ma być odkładane do momentu zatwierdzenia transakcji.

4. Mechanizm reguł nie powinien pozwalać na zapętlanie się inicjowania reguł.

   Łatwo wyobrazić sobie łańcuch reguł: czynność jednej reguły jest zdarzeniem dla innej reguły, której czynność jest zdarzeniem dla kolejnej itd. Łańcuch taki może powodować efekt nie kończącej się kaskady inicjowania reguł. Zaawansowany mechanizm obsługi reguł powinien nie pozwalać na tego typu zapętlenie.

Szczegółową dyskusję wymagań dotyczących reguł, wraz z licznymi przykładami, można znaleźć w [9].

3.5. Przegląd systemów ORDBMS

Aktualnie nie jest dostępny na rynku komercyjny system ORDBMS, który spełniałby w sposób satysfakcjonujący wszystkie wymagania opisane w punktach 3.1. – 3.4. Szereg systemów, deklarowanych przez producentów jako „obiektowo-relacyjne" lub „uniwersalne", spełnia niektóre z tych wymagań. W tabeli 1 zestawiono cechy ważniejszych z tych systemów. Anonsowane są też ich nowe wersje, które — zgodnie z zapowiedziami producentów — coraz pełniej realizować będą funkcje pełnowartościowych systemów obiektowo-relacyjnych.

Tabela 1. Cechy komercyjnych obiektowo-relacyjnych DBMS

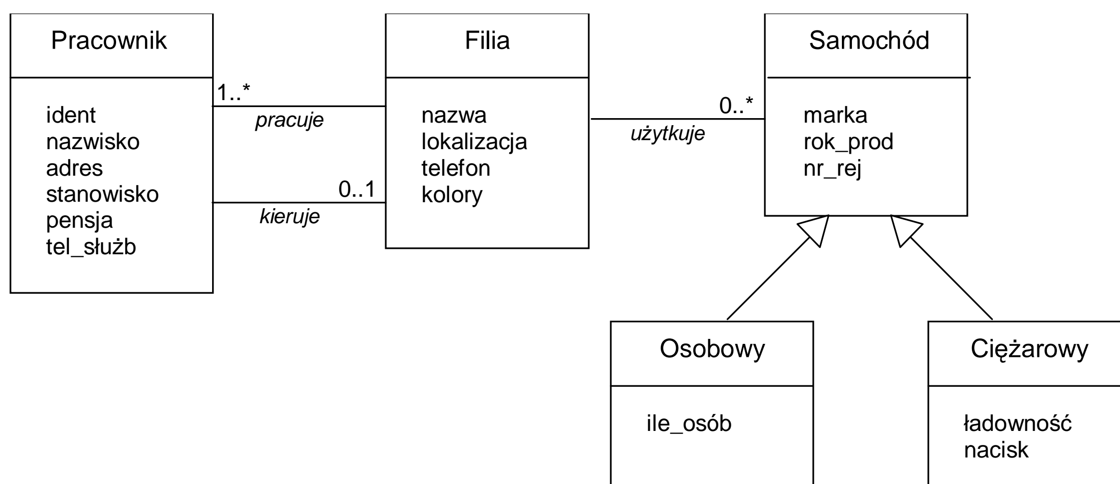
System	Producent	Rozszerzalność typów bazowych	Typy złożone	Dziedziczenie	Reguły
CA-Ingres	Computer Assoc.	Tak	Nie	Nie	Tak[4]
DB2 6000	IBM	Tak	Nie	Nie	Tak[4]
IDS–UDO	Informix	Tak	Tak[1]	Tak[3]	Tak[4]
Oracle V8.0	Oracle	Nie	Tak[2]	Nie	Tak[4]

[1]) brak typów referencyjnych
[2]) bez możliwości zagnieżdżania
[3]) bez wielodziedziczenia
[4]) realizacja częściowa

4. Przykład: Baza danych o filiach firmy

W tej części zilustrujemy podstawowe różnice pomiędzy podejściem relacyjnym a obiektowo-relacyjnym na przykładzie prostej bazy danych przeznaczonej dla dużej firmy składającej się z wielu filii. Każda filia ma swoją nazwę i lokalizację. Każda filia ma do swojej wyłącznej dyspozycji samochody osobowe i ciężarowe. W każdej filii pracują pracownicy firmy, przy czym każdy pracownik pracuje tylko w jednej filii. Jeden z pracowników filii jest jej kierownikiem. Każda filia ma swój charakterystyczny zestaw kolorów, wyróżniający go spośród innych filii.

Mimo tak ogólnych i nieprecyzyjnych założeń, można przystąpić do utworzenia pierwszego przybliżenia modelu systemu. Ponieważ nasz system planujemy zrealizować w technologii obiektowo-relacyjnej, celowe jest stworzenie modelu obiektowego, który następnie w sposób naturalny będzie można przełożyć na schemat obiektowo-relacyjnej bazy danych. Model obiektowy, w postaci diagramu klas według notacji UML [4], przedstawiono na rys. 2. W następnym etapie modelowania należałoby wzbogacić definicje klas ten o potrzebne metody, które w schemacie bazy danych zostałyby odzwierciedlone jako funkcje zdefiniowane przez użytkownika.



Rys. 2. Diagram klas dla bazy danych o filiach firmy

Przekształcając ten model na schemat obiektowo-relacyjnej bazy danych należy rozważyć kilka kwestii (szczegółową dyskusję można znaleźć w [5]). Jedną z nich jest kierunkowość związków. W modelu relacyjnym mamy w tym względzie niewielkie możliwości manewru: jedynie dla związków typu 1:1 (jak związek *kieruje*) mamy możliwość ustalenia położenia klucza obcego: albo w relacji przechowującej dane o filiach, albo w relacji przechowującej dane o pracownikach, albo w obu. W wypadku związków typu 1:*n* lub *n*:*m* zasady normalizacji wyznaczają nam jednoznacznie sposób (a tym samym i kierunkowość) implementacji związków. W modelu obiektowo-relacyjnym, z uwagi na możliwość zastosowania konstruktorów typów złożonych. możemy przyjąć dowolną kierunkowość związków, stosownie do wymogów aplikacji. W naszym przykładzie przyjmijmy, że centralną klasą jest klasa Filia i że chcemy efektywnie wydobywać informacje o zbiorze pracowników danej filii i zbiorze użytkowanych przez daną filię samochodów. Ponadto chcemy móc szybko dowiedzieć się, w jakiej filii zatrudniony jest dany pracownik, natomiast drugorzędne znaczenie ma informacja, przez jaką filię użytkowany jest dany samochód.

Drugą kwestią zasadniczo odróżniającą w naszym przykładzie schemat obiektowo-relacyjny od czysto relacyjnego jest możliwość bezpośredniego zagnieżdżenia w jednym obiekcie wielu innych obiektów. Rozważmy atrybut kolory klasy Filia. Z definicji problemu wynika, że atrybut ten ma charakter wielowartościowy. W schemacie relacyjnym należałoby utworzyć oddzielną relację Kolor, w której kluczem podstawowym byłoby złożenie nazwy koloru z kluczem obcym odpowiadającym

kluczowi podstawowemu relacji Filia, gdyż kolory w zestawach charakterystycznych dla filii mogą się powtarzać.

Można jeszcze zauważyć, że atrybut tel_służb klasy Pracownik i atrybut telefon klasy Filia są w istocie atrybutami złożonymi o tej samej strukturze (numer kierunkowy, numer lokalny, numer wewnętrzny), warto więc dla tych atrybutów zdefiniować odpowiedni typ wierszowy. Dla tego typu wierszowego nie ma oczywiście potrzeby definiowania oddzielnej tablicy.

Pełny schemat obiektowo-relacyjnej bazy danych odpowiadający modelowi z rys. 2, przy przyjętych założeniach implementacyjnych, przedstawiono poniżej. Zwróćmy uwagę na pewną konwencję, zastosowaną w tym przekształceniu. Nazwy klas na diagramie klas są rzeczownikami w liczbie pojedynczej, co jest odzwierciedleniem zasady, że w modelowaniu obiektowym świat rozważamy jako składający się z autonomicznych i hermetycznych obiektów o własnej tożsamości. Natomiast w schemacie bazy danych relacja jest pojemnikiem dla obiektów jednej klasy (jest *obiektem kontenerowym*). W naszym przykładzie relacja Pracownicy jest pojemnikiem dla obiektów klasy Pracownik, relacja Filie jest pojemnikiem dla obiektów klasy Filia itd.

```
create type telefon_t
    (kierunek    char(6),
     numer       char(8),
     wewn        char(4));

create type samochód_t
    (marka       varchar(24),
     rok_prod    integer,
     nr_rej      varchar(12));

create type osobowy_t
    (ile_osób    integer)
under samochód_t;

create type ciężarowy_t
    (ładowność   integer,
     nacisk      integer)
under samochód_t;

create type pracownik_t
    (ident       varchar(8),
     nazwisko    varchar(36),
     adres       varchar(48),
     stanowisko  varchar(18),
     pensja      numeric,
     tel_służb   telefon_t,
     pracuje     ref (filia_t) references Filie);

create type filia_t
    (nazwa       varchar (36),
     telefon     telefon_t,
     lokalizacja varchar(24),
     kolory      set (varchar (18)),
     kierownik   ref (pracownik_t) references Pracownicy,
     zatrudnieni set (ref (pracownik_t) references Pracownicy),
     pojazdy     set (ref (samochód_t) references Samochody));

create table Samochody of type samochód_t;

create table Osobowe of type osobowy_t
under Samochody;

create table Ciężarowe of type ciężarowy_t
under Samochody;
```

```
create table Pracownicy of type pracownik_t

create table Filie of type filia_t
```

Jako ćwiczenie pozostawiamy Czytelnikowi skonstruowanie odpowiedniego schematu relacyjnej bazy danych, zgodnego z postacią 1NF i wykorzystującego klucze podstawowe i obce.

5. Podsumowanie

Typowe aplikacje biznesowe stają się coraz bardziej złożone. Wzbogacane są o moduły analityczne, wspierające procesy podejmowania decyzji. Dane przetwarzane w takich modułach mają charakter złożony. Również ekspansja Internetu walnie przyczynia się do gwałtownego wzrostu potrzeb w zakresie przetwarzania i prezentacji danych multimedialnych, danych o złożonej strukturze, a także danych niekompletnych. Współczesne aplikacje stają się nieuchronnie aplikacjami o charakterze uniwersalnym, wymagającymi zarówno dużej elastyczności i ekspresji w modelowaniu, jak i wydajności przetwarzania. W szczególności należy oczekiwać rozwoju i upowszechnienia technologii zapytań według zawartości (*Query By Content*), w których poszukuje się obiektów nie na podstawie konkretnych wartości ich atrybutów, ale na podstawie ogólnie i często nieprecyzyjnie wyrażonych charakterystyk, takich jak wygląd, podobieństwo do innych obiektów czy cechy o charakterze subiektywnym.

Tradycyjne systemy i aplikacje relacyjne nie są w stanie sprostać takim potrzebom. Uzasadnione więc jest wyrażone w [9] stwierdzenie, że popyt na systemy typu *Universal Server* i na systemy ORDBMS będzie stale i szybko rósł tak, że niebawem (za ok. 5 lat) rynek systemów uniwersalnych i obiektowo-relacyjnych stanie się większy niż rynek tradycyjnych systemów relacyjnych. Można więc śmiało powiedzieć, że stoimy w obliczu wielkiej fali systemów baz danych nowej generacji.

Na zakończenie odnieśmy się do tytułowego „utrudniania życia informatykom". Czy nowe modele danych wprowadzane w systemach obiektowo-relacyjnych ułatwią czy utrudnią pracę informatyka analityka, którego zadaniem jest przede wszystkim skonstruowanie modelu jak najwierniej odzwierciedlającego rzeczywistość, a jednocześnie adekwatnego do stosowanych narzędzi implementacyjnych? Przecież do tej pory jakoś udawało się zmieścić wszystko w sztywne, ale mocne ramy tablic relacyjnej bazy danych... Pamiętajmy, że model obiektowo-relacyjny, promowany przez normę SQL-99, mieści w sobie klasyczny model relacyjny. A więc w aplikacjach, w których nie ma potrzeby operowania na złożonych strukturach danych czy też używania niestandardowych funkcji w zapytaniach, nadal będzie możliwe stosowanie podejścia czysto relacyjnego, z jego wewnętrzną prostotą i matematyczną logiką. Uważny i otwarty analityk dostrzeże jednak, że nawet „typowe" projekty relacyjne (np. takie jak baza danych wydziałowych z przykładu z punktu 4.) dają się lepiej wyrazić w terminach modelu obiektowo-relacyjnego. Lepiej – to znaczy prościej, elastyczniej, bardziej ekspresywnie. Takie są bowiem korzyści płynące z zastosowania w technologii baz danych pojęć wywodzących się z paradygmatu obiektowego.

Bibliografia

1. American National Standards Institute: Database Language SQL. Document ANSI X3.135-1986, 1986.
2. American National Standards Institute/International Organization for Standardization. ISO/ANSI Database Language SQL2 (working draft). Document ANSI X3H2-89-151/ISO DBL CAN-3a, 1989.
3. Cattell R.G.G, Barry D.K. (eds): The Object Data Standard: ODMG 3.0. Morgan Kaufmann Pub. Inc., 2000, ISBN 1-55860-647-5
4. Eriksson H.-E., Penker M.: UML Toolkit. J.Wiley & Sons, Inc., 1997.
5. Goczyła K.: Bazy danych w systemach obiektowych. W: Inżynieria oprogramowania w projekcie informatycznym (red. J. Górski), wyd. 2 rozszerzone. MIKOM, 2000, ISBN 83-7279-028-0
6. Gulutzan P., Pelzer T.: SQL-99 Complete, Really. R&D Books, 1999, ISBN 0-87930-568-1
7. International Organization for Standardization: Database Language SQL. Document ISO/IEC 9075,

1992.

8. Kim W.: Wprowadzenie do obiektowych baz danych. WNT, 1996, ISBN 83-204-2026-1
9. Stonebraker M., Brown P.: Object-Relational DBMSs: Tracking the Next Great Wave. Morgan Kaufmann Pub. Inc., 1999, ISBN 1-55860-452-5
10. Subieta, K.: Obiektowość w projektowaniu i bazach danych. Akademicka Oficyna Wydawnicza PLJ, 1998, ISBN 83-7101-401-5