

**Prace Naukowe
Instytutu Informatyki Stosowanej
Politechniki Wrocławskiej**

6

BAZY DANYCH

Redakcja naukowa

Zygmunt Mazur

**Oficina Wydawnicza
Politechniki Wrocławskiej**



Prace Naukowe
Instytutu Informatyki Stosowanej
Politechniki Wrocławskiej

6 BAZY DANYCH

Redakcja naukowa

Zygmunt Mazur



Oficina Wydawnicza Politechniki Wrocławskiej
Wrocław 2005



Politechnika Wrocławska

Instytut Informatyki Stosowanej

Recenzent

Aleksander ZGRZYWA

Opracowanie redakcyjne materiałów

Hanna MAZUR

Projekt okładki

Zygmunt MAZUR

© Copyright by Oficyna Wydawnicza Politechniki Wrocławskiej, Wrocław 2005

OFICyna WYDAWNICZA POLITECHNIKI WROCLAWSKIEJ

Wybrzeże Wyspiańskiego 27, 50-370 Wrocław

ISBN 83-7085-847-3

Drukarnia Oficyny Wydawniczej Politechniki Wrocławskiej. Zam. nr 915/2005.

SPIS TREŚCI

<i>Wstęp</i>	5
A. Iwanejko., R. Pawlicki, <i>Realizacja funkcjonalności języka ODL w systemach zarządzania obiektowymi bazami</i>	7
Z. Mazur, P. Michalec, <i>Zastosowanie opisu temporalno-probabilistycznego do harmonogramowania transakcji</i>	27
M. Bednarski, A. Klosov, <i>Metody inżynierii wstecznej w systemach obiektowych baz danych</i>	35
Z. Szpunar, A. Wróblewski, <i>Rzutowanie modelu obiektowego do relacyjnego modelu danych</i>	51
J. Gruber, H. Mazur, Z. Mazur, M. Standzoń, <i>Modelowanie i generowanie bazodanowych aplikacji WWW w narzędziu PowerDesigner</i>	67
M. Burda, D. Król, <i>Koncepcja modelu semantycznej bazy danych wykorzystującej skojarzenia</i>	109

CONTENTS

<i>Introduction</i>	5
A. Iwanejko, R. Pawlicki, <i>Realization of ODL Functionality in Object Database Management Systems</i>	7
Z. Mazur, P. Michalec, <i>Applying of Temporal-Probabilistic Description of Transaction to Transactions' Scheduling</i>	27
M. Bednarski, A. Klosow, <i>Reverse Engineering Methods in the Object Database Systems</i>	35
Z. Szpunar, A. Wróblewski, <i>Object-To-Relational Model Mapping</i>	51
J. Gruber, H. Mazur, Z. Mazur, M. Standzoń, <i>PowerDesigner Tool for Modelling and Building Database Web Applications</i>	67
M. Burda, D. Król, <i>Semantic Database Model Based on Associations</i>	109

*Konstruktor wie, iż osiągnął doskonałość
nie wtedy, gdy nic już nie może dodać,
lecz wtedy, gdy nic już nie da się ująć*
• Antoine de Saint-Exupery

WSTĘP

Pierwszy numer Zeszytu Naukowego serii „*Bazy danych*” – *Prace Naukowe Wydziałowego Zakładu Informatyki Politechniki Wrocławskiej* pod redakcją Zygmunta Mazura został wydany w 2000 r. i od tego czasu, systematycznie co roku, ukazywały się kolejne numery. W dniu 01 grudnia 2004 r. dwa zakłady: Wydziałowy Zakład Informatyki oraz Wydziałowy Zakład Systemów Informacyjnych, utworzyły Instytut Informatyki Stosowanej. Dlatego też obecny numer Zeszytów jest wydawany pod szyldem Instytutu Informatyki Stosowanej na Wydziale Informatyki i Zarządzania Politechniki Wrocławskiej.

Tematyka bieżącego Zeszytu Naukowego związana jest z:

- realizacją funkcjonalności języka ODL w systemach zarządzania obiektowymi bazami danych,
- zastosowaniem opisu temporalno-probabilistycznego do harmonogramowania transakcji,
- opisem metod inżynierii wstecznej w systemach obiektowych baz danych,
- opisem metody rzutowania modelu obiektowego do relacyjnego modelu danych,
- modelowaniem i generowaniem bazodanowych aplikacji WWW w technologii *PowerDesigner*,
- problematyką efektywności przetwarzania transakcji w rozproszonych bazach danych.

Pierwszy artykuł Zeszytu 6. dotyczy funkcjonalności języka ODL w systemach zarządzania obiektowymi bazami danych. Jest próbą wyjaśnienia pojęcia obiektowej bazy danych, przedstawia wybrane aspekty języka ODL (skrót od ang. *Object Definition Language*) oraz porównuje język ODL z językami definicji danych używanymi w wybranych systemach zarządzania obiektowymi bazami danych.

W artykule omówiono też wady i zalety języka ODL oraz zaproponowano zmiany w standardzie, które mogą wpłynąć na zminimalizowanie jego wad.

Autorzy drugiego artykułu przedstawiają algorytm przydzielania priorytetów transakcjom w podejściu temporalno-probabilistycznym.

W pracy *Metody inżynierii wstecznej w systemach obiektowych baz danych* przedstawiono propozycję prostego narzędzia typu CASE, które odtwarza w postaci graficznej obiektowy model danych w oparciu o pliki źródłowe zapisane w języku C++. Diagramy klas reprezentujące schemat bazy danych wyrażone zostały w notacji UML (ang. *Unified Modeling Language*).

W ostatnich latach programowanie obiektowe zyskało ogromną popularność. W aplikacjach obiektowych bardzo często wykorzystuje się dane z relacyjnych baz danych, zbudowanych na modelu relacyjnych. Prowadzi to do sytuacji, w których z poziomu języka obiektowego, programista musi posługiwać się pojęciami modelu relacyjnego danych takimi jak relacja, krotka, klucz, a równocześnie wykorzystywać pojęcia języka obiektowego takie jak klasa, obiekt, dziedziczenie, referencja. W artykule czwartym autorzy opisują jedno z rozwiązań, które umożliwia programiście operowanie na relacyjnej bazie danych za pomocą pojęć modelu obiektowego, a także przedstawiają korzyści i wady z zastosowania takiego rozwiązania.

Celem kolejnego (piątego) artykułu jest przedstawienie możliwości projektowania, modelowania i generowania baz danych za pomocą pakietów *Sybase PowerDesigner* i *Sybase PowerBuilder* na przykładzie konkretnego zadania projektowego. W pracy omówiono składowe *ProcessAnalyst*, *DataArchitect* i *AppModeler*, które wspomagają projektowanie i generowanie trzywarstwowych aplikacji internetowych.

Jednym z problemów komputerowego przetwarzania danych jest fakt, że komputer nie rozumie treści, którą przetwarza. Dlatego nie wystarczy odpowiednio przechowywać dane w bazie danych ale trzeba je rozumieć. Do tego wymagana jest odpowiednia wiedza. Aby poradzić sobie z tym problemem, autorzy ostatniego artykułu wykorzystują strukturę pojęciową umożliwiającą reprezentację wiedzy niesformalizowanej czyli sieci semantyczne.

Autorzy składają podziękowanie Recenzentowi za uwagi zamieszczone w recenzji, które zostały wykorzystane w trakcie przygotowywania końcowych wersji artykułów.

Zygmunt Mazur

Agnieszka IWANEJKO*
Rafał PAWLICKI**

REALIZACJA FUNKCJONALNOŚCI JĘZYKA ODL W SYSTEMACH ZARZĄDZANIA OBIEKTOWYMI BAZAMI DANYCH

W artykule przedstawiono pojęcie obiektowej bazy danych i wybrane aspekty *Object Definition Language* oraz porównano ODL z językami definicji danych używanymi w wybranych systemach zarządzania obiektowymi bazami danych. Omówiono również wady i zalety ODL oraz zaproponowano zmiany w standardzie mające na celu zminimalizowanie jego wad.

1. WPROWADZENIE

Rozpowszechnienie technologii obiektowej oraz rosnąca krytyka modelu relacyjnego spowodowała powstanie nowej generacji baz danych znanych jako obiektowe bazy danych (OBD – *Object Database*). OBD przechowują obiekty będące instancjami klasy. Klasy umożliwiają enkapsulację danych o dowolnej strukturze wraz ze skojarzonymi metodami [VAHE].

Obiektowe bazy danych powstały 20 lat temu jako rozwinięcie programowania zorientowanego obiektowo, a dostępne na rynku są od około 12 lat. Wprowadzenie obiektowości do baz danych uwalnia programistę od wielu ograniczeń nałożonych przez istniejące technologie, a zwłaszcza technologię relacyjną (dane i związki nie są przedstawiane w postaci relacji, łatwiejsze jest zamodelowanie rzeczywistości). Podejście obiektowe umożliwia nie tylko przechowywanie danych, ale również udostępnia metody dające możliwość manipulowania tymi danymi.

OBD dziedziczą zasadnicze cechy technologii obiektowej (istnienie złożonych obiektów, tożsamość obiektów, enkapsulacja danych i metod, dziedziczenie, metody polimorficzne, rozszerzalność o nowe typy danych) i baz danych (trwałość danych, oddzielenie logicznego i fizycznego poziomu danych, zarządzanie wielodostępem, odtwarzanie spójnego stanu danych po awariach, zapytania ad hoc, zarządzanie transakcjami itd.) [APSIN].

* Politechnika Wrocławska, Wydział Informatyki i Zarządzania, Instytut Informatyki Stosowanej, 50-370 Wrocław, Wyb. Wyspiańskiego 27 (doktorantka), agnieszka.iwanejko@pwr.wroc.pl

** Politechnika Wrocławska, Wydział Informatyki i Zarządzania, Instytut Informatyki Stosowanej, 50-370 Wrocław, Wyb. Wyspiańskiego 27 (doktorant), rafal.pawlicki@pwr.wroc.pl

System baz danych służy do przechowywania informacji i udostępnianie jej na każde życzenie użytkownika. Składa się on z czterech zasadniczych części: danych, sprzętu, programów zarządzających danymi oraz użytkowników [DATE 2000]. Obiektowy system baz danych jest to system przechowywania danych zapisanych w postaci obiektów.

2. JĘZYK ODL

Język definiowania obiektów (ang. *Object Definition Language* – *ODL*) stanowi próbę standaryzacji obiektowych języków do specyfikowania struktury baz danych w terminach obiektowych, w której starano się połączyć elementy programowania wysokiego poziomu SQL z paradygmatami programowania obiektowego takich języków programowania jak C++, C#, Java czy Smalltalk [LAUSEN 2000].

2.1. SYNTAKTYKA I SEMANTYKA

Definiując obiekty w języku ODL konieczne jest opisanie trzech rodzajów własności:

- (1) atrybutów,
- (2) związków,
- (3) metod.

Atrybut definiowany jest jako właściwość, której typ powstaje z typów pierwotnych.

Związek definiowany jest jako odniesienie do obiektu pewnej klasy lub jako kolekcja (zbiór) takich odniesień.

Metody to funkcje lub procedury operujące na obiektach danej klasy.

2.1.1. DEKLARACJA INTERFEJSU

Deklaracja interfejsu w języku ODL składa się z czterech podstawowych części:

- (1) Słowa **interface**,
- (2) Nazwy interfejsu,
- (3) List właściwości,
- (4) Deklaracji metod.

2.1.2. ATRYBUTY W ODL

Atrybuty w języku ODL opisują cechy obiektu dające się wyrazić wartościami typów podstawowych i strukturalnych. Przykładem takich atrybutów dla klasy *faktura*

może być *data_wystawienia* (typ czasowy), *nazwa_odbiorky* (typ tekstowy), czy *wartość_do_zapłaty* (typ zmiennoprzecinkowy).

Możliwe jest także definiowanie typów wyliczeniowych. Typ wyliczeniowy definiuje się używając słowa **enum**, nazwy typu wyliczeniowego oraz ujętego w nawiasy klamrowe zbioru dopuszczalnych wartości. Przykładem typu wyliczeniowego może być typ *sposób_płatności*, definiujący sposób płatności za fakturę:

```
enum sposób_płatności {gotówka, przelew, karta_kredytowa}
```

Język definiowania obiektów umożliwia też tworzenie struktur (podobnie jak w języku C++). Strukturę definiuje się używając słowa kluczowego **struct** oraz nazwy struktury. Następnie w nawiasach okrągłych umieszcza się nazwy pól poprzedzone nazwami typów. Definiując *adres_odbiorky* naturalnym wydaje się użycie struktury:

```
struct adres (string ulica, integer nr_domu, string kod_pocztowy, string miasto)
```

Atrybuty w języku ODL mogą przyjmować wartości typów strukturalnych. Typ strukturalny tworzy się z typów bazowych (typ bazowy to typ prosty lub interfejs) poprzez zastosowanie następujących konstruktorów typów:

- (1) Zbiór – definiowany jest jako typ, którego wartości są dowolnymi skończonymi zbiorami wartości danego typu. W zbiorze nie ma powtórzeń. Przykładem konstruktora zbioru może być zbiór studentów zapisanych na dany przedmiot:

```
Set<Student> zapisani_na_przedmiot
```

- (2) Multizbiór – definiowany jest jako typ, którego wartości są dowolnymi skończonymi zbiorami wartości danego typu. W multizbiorze możliwe są powtórzenia. Konstruktor multizbioru ma następującą postać:

```
Bag<typ_elementu_multizbioru> nazwa_multizbioru
```

- (3) Lista – to typ, którego wartości są skończonymi listami wartości danego typu. W liście możliwe są powtórzenia. Istotna jest także kolejność elementów listy. Szczególnym przykładem listy jest typ **String**, który jest skróconym zapisem typu:

```
List<char>
```

- (4) Tablica – definiowana jest przez podanie jej wielkości (integer) oraz typu elementów tablicy. Na przykład napis **Array**<char,6> *kod_pocztowy* oznacza 6 elementową tablicę znaków umożliwiającą przechowywanie kodu pocztowego.

2.1.3. ZWIĄZKI W ODL

Aby możliwe było zamodelowanie rzeczywistości w ODL nie wystarczy opisanie obiektu tylko za pomocą wartości atrybutów. Konieczne jest także przedstawienie

powiązań danego obiektu z innymi obiektami (tej samej bądź innej klasy). Aby zdefiniować związek pomiędzy obiektami należy użyć słowa **relationship**. Przykład związku, który przedstawia zależność jeden do wielu pomiędzy klasą *kurs* i klasą *student* modelującą fakt, że w kursie może uczestniczyć wielu studentów przedstawiono poniżej:

```
interface kurs{
    ....
    relationship Set<student> uczestnicy
}
```

Powyższy przykład pokazuje powiązanie pomiędzy jednym obiektem klasy *kurs* i wieloma obiektami klasy *student*. Możliwa jest też sytuacja, w której obiekt danej klasy jest powiązany tylko z jednym obiektem innej bądź tej samej klasy. Przykładem takiej sytuacji może być powiązanie *maz–zona*. Mąż ma tylko jedną żonę a żona ma tylko jednego męża. Jeśli chcemy podkreślić fakt, że w przypadku, gdy dana osoba jest czyjąś żoną, to jednocześnie ma męża to można to zrobić używając związków odwrotnych. W celu określenia, że związek *jestZona* w klasie *kobieta* jest odwrotny do związku *jestMezem* w klasie *mezczyzna*, konieczne jest określenie w deklaracji klasy związku odwrotnego. Poniżej przedstawiono składnię deklarowania związków odwrotnych:

```
interface <nazwa interfejsu>{
    ...
    relationship < nazwa klasy powiązanej > <nazwa relacji>
        inverse <nazwa klasy powiązanej>:: <nazwa relacji odwrotnej>
}
```

Przykład pokazuje deklarację związku odwrotnego w klasie *mąż*:

```
interface mezczyzna{
    ...
    relationship kobieta jestMezem
        inverse kobieta::jestZona
}
```

W języku ODL przedstawienie liczności relacji możliwe jest w ograniczonym zakresie. Można zdefiniować związki jeden do jeden (powyższy przykład relacji *maz–zona*), relację jeden do wielu (*kurs–student*) oraz wiele do wielu:

```
interface student{
    ...
    relationship Set<kurs> uczestniczyW
        inverse kurs:: maKursanta
}
```

```

interface kurs{
    ...
    relationship Set<student> maKursanta
    inverse student::uczestniczyW
}

```

Możliwe jest także zamodelowanie sytuacji, że spodziewana relacja nie istnieje (np. mężczyzna jest kawalerem i nie ma relacji *jestMezem*). W tej sytuacji stosuje się wartość pustą *nil* (podobnie jak *NULL* w C++ czy w Javie).

2.1.4. PODKLASY

Jedną z najważniejszych zalet obiektowych baz danych jest możliwość dziedziczenia cech klasy bazowej. Wygodnie jest określić cechy wspólne dla szerszej klasy obiektów i wykorzystywać te informacje w podklasach. Język ODL zawiera mechanizmy służące definiowaniu podklas. Możliwe jest zatem tworzenie hierarchii klas, możliwe jest także wielodziedziczenie. Podklasa dziedziczy wszystkie właściwości swojej nadklasy. Każdy związek podklasy dziedziczy wszystkie atrybuty, związki i metody swojej nadklasy. Poniższy przykład pokazuje w jaki sposób zapisuje się w języku ODL dziedziczenie:

```

interface <nazwa klasy> : <nazwa klasy bazowej>{[,<nazwa klasy bazowej>]...}

```

Ciekawą cechą jest możliwość dziedziczenia po wielu klasach bazowych. Istnieje tu jednak pewne zagrożenie – konflikt nazw w klasach bazowych. Występuje on wtedy, gdy dwie klasy bazowe mają w swej definicji taką samą nazwę atrybutu lub relacji. Syntaktyka języka ODL nie rozstrzyga w jaki sposób konflikt ten ma być rozstrzygany.

2.1.5. MODELOWANIE WIĘZÓW INTEGRALNOŚCI

Pewnych ważnych aspektów świata rzeczywistego nie da się przedstawić za pomocą przedstawionych powyżej mechanizmów. Aspekty te nazywane są więzami integralności i nie da się ich oddać w strukturze typów i ograniczeń wynikających z definicji klas, atrybutów i związków. Podstawowe rodzaje więzów to:

- (1) Klucze – są one zbiorami atrybutów jednoznacznie identyfikującymi obiekt wewnątrz zbioru encji. Nie mogą wystąpić dwa obiekty należące do tej samej klasy mające takie same klucze;
- (2) Więzy jednoznaczności nakładają wymaganie, by wartości danego atrybutu były unikalne dla całej klasy obiektów (np. dla klasy *Osoba* wymagane jest by atrybut PESEL był niepowtarzalny);
- (3) Więzy integralności referencyjnej nakładają wymaganie, by obiekt, na który wskazuje relacja rzeczywiście istniał w bazie danych;

- (4) Więzy domenowe – nakładają dodatkowe ograniczenia dotyczące wartości atrybutu, (np. rok urodzenia ma być nie tylko liczbą całkowitą, ale i liczbą z zakresu 1900–2004);
- (5) Więzy zasadnicze – są warunkami, których przestrzeganie jest bezwzględnie wymagane w bazie danych (np. student może być zapisany na maksymalnie 10 kursów).

W języku ODL do definiowania kluczy używane jest słowo kluczowe *key* lub *keys*. Następnie w nawiasach podawana jest lista argumentów stanowiąca klucz. Deklaracja klucza musi nastąpić zaraz po deklaracji **interface**. Kluczy może być wiele. Sama deklaracja także jest ujęta w nawiasy okrągłe.

Poniżej przedstawiono definicję dwóch kluczy prostych dla interfejsu

IPracownik. :

interface IPracownik (key nr_pracownika, nr_legitymacji_ubezp){...}

Aby zadeklarować klucz złożony należy użyć następującej składni:

interface IPracownik (key (nr_pracownika, nr_legitymacji_ubezp)){...}

Ograniczenie, by wartość atrybutu nie przyjmowała wartości *nil*, można uzyskać poprzez umieszczenie atrybutu w definicji klucza. Pozostałe atrybuty z definicji mogą przyjmować wartości *nil*.

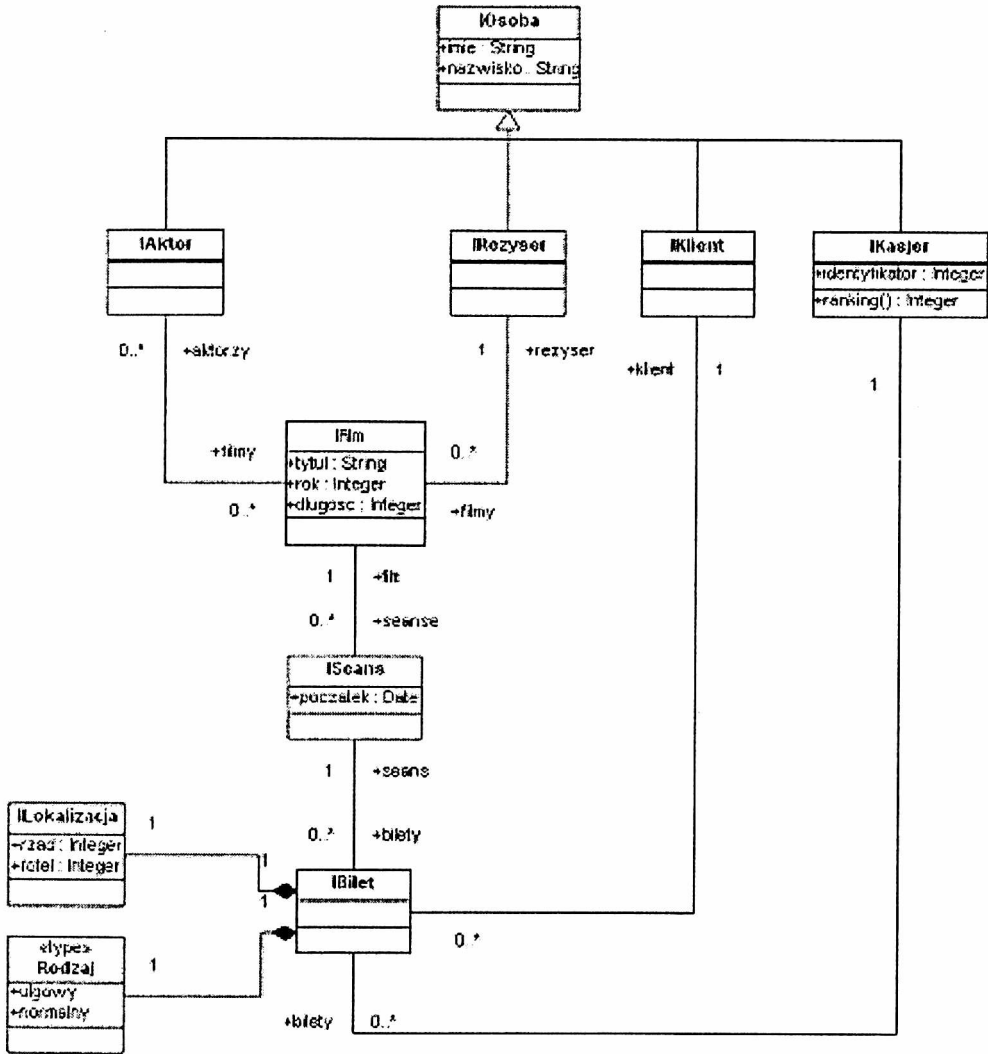
Ograniczenia dotyczące więzów integralności referencyjnej pozostawiono implementacjom.

2.2. METODY W ODL

W języku ODL możliwe jest deklarowanie metod. ODL w zamierzeniach nie ma być pełnym językiem programowania. Jest to język do definiowania obiektów. Za pomocą tego języka można zdefiniować klasy (interfejsy), włącznie z ich atrybutami oraz metodami operującymi na tych danych. ODL tworzy jedynie sygnatury operacji, ale nie mówi nic na temat implementacji tych operacji.

2.3. PRZYKŁAD DEFINICJI BAZY W ODL

W rozdziale przedstawimy przykład, który będzie wykorzystany w dalszej części artykułu do porównania składni definiowania danych w ODL oraz w wybranych SZOBD.

Rys. 1. Diagram klas w UML dla bazy danych **Kino**

Rozważmy opis wycinka rzeczywistości dotyczący kina.

W kinie wyświetlane są filmy. Film posiada następujące cechy: tytuł, rok produkcji, reżysera, aktorów i określoną długość w minutach. Klient może kupić u kasjera bilet (ulgowy lub normalny) na seans filmowy. Na podstawie ilości sprzedanych biletów można stworzyć ranking kasjerów. Bilet ma cenę. Seans ma określoną godzinę i datę rozpoczęcia i dotyczy konkretnego filmu. Kino ma tylko jedną salę kinową.

Na rysunku 1. zamieszczono diagram klas UML przedstawiający schemat bazy danych **Kino**.

Poniżej przedstawiona jest definicja bazy danych **Kino** w ODL.

```

interface Ifilm{
    attribute string tytul;
    attribute integer rok;
    attribute integer dlugosc;
    relationship Set<Iaktor> aktorzy;
    relationship Irezyser rezyser;
    relationship Set<Iseans> seanse
        inverse Iseans::film;
};

interface Iseans{
    attribute timestamp poczatek;
    relationship Ifilm film
        inverse Ifilm::seanse;
    relationship Set<Ibilet> bilety
        inverse Ibilet::seans;
};

interface Ibilet{
    attribute enum rodzaj{ulgowy, normalny} rodzajBiletu;
    attribute struct lokalizacja{integer rzad, integer fotel} miejsce;
    relationship Iseans seans
        inverse Iseans::bilety;
    relationship Iklient klient;};

interface Iosoba{
    attribute string imie;
    attribute string nazwisko;
};

interface Iklient: Iosoba{
};

interface Ikasjer: Iosoba{
    attribute integer identyfikator;
    relationship Set<Ibilet> bilety;
    ranking(out integer);
};

interface Iaktor: Iosoba{
};

```

```
interface Irezyser: Iosoba{
};
```

Powyższy przykład pokazuje sposób definiowania interfejsów, ich dziedziczenia oraz przykłady definicji relacji asocjacji (jeden do jednego, jeden do wielu, wiele do wielu, jedno- i dwukierunkowe) oraz sygnatur metod.

3. SYSTEMY ZARZĄDZANIA OBIEKTOWYMI BAZAMI DANYCH

Istnieje wiele systemów zarządzania obiektowymi bazami danych – SZOBD (np. ObjectStore, FastObject, Versant, Objectivity/DB, GemStone, Jasmine, O2).

W rozdziale przedstawiono porównanie ODL z FastObject (rozdz. 3.1) oraz z ObjectStore (rozdz. 3.2). Te dwa systemy zarządzania obiektowymi bazami danych zostały wybrane z uwagi na dobrą dokumentację tych SZOBD.

3.1. FASTOBJECT

Bardzo istotną cechą w SZOBD FastObject jest trwałość obiektów. Poprzez obiekty trwale nazywane są obiekty, których stan jest pamiętany po zakończeniu programu. Kiedy program jest uruchamiany ponownie te trwałe obiekty mogą być odzyskane, a dane w nich zawarte będą ponownie dostępne. Interfejs C++ dla FastObject używa Menadżera obiektowej bazy danych, który umożliwia przechowywanie i dostęp do obiektów trwałych.

Podczas tworzenia aplikacji programista powinien ustalić, które klasy są trwałe, a które nie. Aby uczynić klasy języka C++ trwałymi, konieczne jest dodanie do nich kilku elementów językowych do deklaracji klasy. Te wzbogacone deklaracje klas są zazwyczaj przechowywane w plikach z rozszerzeniem .hcd. Ponieważ nowe elementy językowe nie są rozumiane przez kompilator C++, deklaracja klasy musi być przełożona przez preprocesor FastObject dla C++. Interfejs FastObject dla C++ dostarcza także klasy i metody możliwe do wykorzystania w klasach definiowanych przez programistę. Zawierają one między innymi kolekcje, obsługę transakcji i operacje na bazie danych.

Baza danych FastObject składa się z dwóch części: właściwej bazy danych i słownika. Wszystkie trwałe obiekty tworzone i używane przez aplikację są przechowywane w bazie danych. Słownik, nazywany niekiedy schematem klas, jest miejscem, w którym są przechowywane wszystkie informacje dotyczące struktury klas. Słownik i baza są automatycznie tworzone, gdy wywołuje się preprocesor C++ FastObject'a.

Interfejs C++ FastObject używa specjalnego programu (ptxx) do odczytania i przetworzenia deklaracji klas trwałych. Uruchamiając preprocesor pozyskuje się

konieczne informacje z rozszerzonych o specyficzne składniki leksykalne FastObject deklaracji klas C++ i używa się tych informacji do zarejestrowania klas w słowniku. Preprocesor tworzy także nową pustą bazę danych, jeśli ta baza nie istniała wcześniej. Dodatkowo preprocesor tworzy wszystkie potrzebne pliki C++, które zawierają przetworzone deklaracje klas oraz kod służący do przechowywania i pozyskiwania trwałych obiektów.

FastObject wspiera różne środowiska programowania. Obiekty i literały są przechowywane w formacie niezależnym od języka i systemu operacyjnego. Manager typów FastObject definiuje podstawowe typy, które mogą być przechowywane w bazie danych. Następnie typy te są wiązane z poszczególnymi językami obiektowymi (np. C++ lub Java).

W dalszej części zajmiemy się powiązaniem modelu FastObject z językiem C++. Definiując nową klasę trwałą (interfejs w terminologii ODMG) używa się rozszerzonej syntaktyki C++. Podstawowym rozszerzeniem jest dodanie słowa kluczowego **persistent** do deklaracji klasy. Słowo to powoduje, że tworzona klasa dziedziczy po klasach zawierających mechanizmy do przechowywania obiektów trwałych [FASTOBJECT].

3.1.1. TYPY DANYCH

FastObject używa wszystkich podstawowych typów C++. Dodaje jednak kilka własnych. Istotną cechą FastObject jest istnienie typów do przechowywania daty (**class PtDate** i **class d_Date**), czasu (**class PtTime** i **class d_Time**), znacznika czasu (**class PtDateTime** i **class d_Timestamp**) oraz interwału czasu (**d_Interval**). Istnieje także typ logiczny, referencyjny oraz typ służący do identyfikacji obiektów. Do przechowywania dużych obiektów binarnych, takich jak pliki, służy typ **PtBlob**.

Typy wyliczeniowe tworzy się używając składni języka C++:

```
enum Plec
{
    Kobieta          = 1,
    Meczyczna       = 2
};
```

Możliwe jest definiowanie własnych struktur:

```
class Adres
{
    PtString  ulica_nr;
    PtString  miasto;
    PtString  kodPocztowy;
};
```

Pełną listę typów w FastObject przedstawiono w tabeli 1.

Tabela 1. Typy danych w FastObject

Typ FastObject	Klasa (typ) C++	Model
PtCHAR_T	char (1 byte)	C++
PtSHORT_T	short (2 bytes)	C++
PtLONG_T	long (4 bytes)	C++
PtDOUBLE_T	double (8 bytes)	C++
PtFLOAT_T	float (4 bytes)	C++
PtWORD_T	unsigned short (2 bytes)	C++
PtDWORD_T	unsigned long (4 bytes)	C++
PtINT_T	int (4 bytes)	C++
PtUNSIGNED_T	unsigned int (4 bytes)	C++
PtENUM_T	enum (4 bytes)	C++
PtBYTE_T	unsigned char (1 byte)	C++
PtSTRING_T	class PtString	FastObjects
PtDATE_T	class PtDate	FastObjects
PtTIME_T	class PtTime	FastObjects
PtBLOB_T	class PtBlob	FastObjects
PtDATETIME_T	class PtDateTime	FastObjects
D_DATE_T	class d_Date	ODMG
D_TIME_T	class d_Time	ODMG
D_TIMESTAMP_T	class d_Timestamp	ODMG
D_INTERVAL_T	class d_Interval	ODMG
PtINT64_T	class PtInt64	FastObjects
PtBOOLEAN	class PtBoolean	FastObjects
PtREF_T	class PtRef	FastObjects
PtOBJID_T	class PtObjId	FastObjects
PtCLID_T	class PtClassId	FastObjects

W definicji klasy niedopuszczalne jest używanie wskaźników do obiektów nie będących obiektami trwałymi. Obiekt trwały nie może także zawierać pól bitowych, unii. Obiekt trwały nie może zostać stworzony przy wykorzystaniu szablonów.

Przykład:

persistent class Osoba

```
{
    PtString    imie;
    PtString    nazwisko;
    int         wiek;
    PtDate      data_urodzenia;
    PtBlob      zdjecie;
    Adres       adres;
    Osoba*      malzonek;
```

```

        lset<Osoba*>      dzieci;
};

```

3.1.2. DZIEDZICZENIE

Dziedziczenie w FastObject używa mechanizmów C++. Przy definicji klasy podaje się nazwy klas z których dziedziczy oraz sposób dziedziczenia (prywatny, publiczny i chroniony). Możliwe jest wielodziedziczenie. Zapewnione jest dziedziczenie używające mechanizmów późnego wiązania.

W rozdziale 3.1.6. przedstawiono przykład dziedziczenia, w którym to klasy Klient i Kasjer dziedziczą po klasie IOsoba.

3.1.3. ZWIĄZKI

FastObject realizuje relację jeden do jeden na trzy różne sposoby:

- (1) poprzez wskaźniki do obiektów trwałych;
- (2) używając słowa kluczowego *ondemand* – obiekty tak zadeklarowane ładowane są do pamięci operacyjnej dopiero, gdy nastąpi taka potrzeba. Przyspiesza to przetwarzanie i sprawia, że jest potrzebne mniej pamięci operacyjnej;
- (3) poprzez obiekty osadzone – nie mogą one istnieć samodzielnie bez klasy, w której są zdefiniowane.

Relacja jeden do wielu realizowana jest przy użyciu zbiorów. System FastObject rozróżnia małe (**cset**<klasa użytkownika*>) i duże zbiory (**lset**<klasa użytkownika*>). Implementacja obu typów zbiorów jest identyczna. Dla zbiorów dużych używany jest synonim **hset** dla wstecznej kompatybilności. Zbiory małe to zbiory, których wielkość ograniczona jest do 32 KB. Przykład zastosowania relacji przedstawiono w rozdziale 3.1.6.

3.1.4. WIĘZY INTEGRALNOŚCI

Do zachowania więzów integralności w FastObject używa się słowa kluczowego **depend**. Jeżeli obiekt bazowy przestaje istnieć, to niszczone są wszystkie obiekty zależne.

3.1.5. METODY

ODL umożliwia tworzenie jedynie sygnatury operacji, ale nie umożliwia implementacji tych operacji. W FastObject metody deklarowane są i definiowane podobnie jak w języku C++.

3.1.6. PRZYKŁAD DEFINICJI BAZY W FASTOBJECT

Przykład przedstawia definicję klas w FastObject. Klasy te opisują wycinek rzeczywistości dotyczący kina (rysunek 1).

```
persistent class IFilm{
    PtString tytul;
    int rok;
    int dlugosc;
    lset<IAktor*> aktorzy;
    IRezyser* rezyser;
    lset<ISeans*> seanse;
};
```

```
persistent class ISeans{
    d_Timestamp poczatek;
    IFilm* film;
    lset<IBilet*> bilety;
};
```

```
enum Rodzaj{
    ulgowy = 1,
    normalny = 2
};
```

```
persistent class lokalizacja{
    int rzad;
    int fotel;
};
```

```
persistent class IBilet{
    Rodzaj rodzajBiletu;
    lokalizacja miejsce;
    ISeans* seans;
    IKlient* klient;
};
```

```
persistent class IOsoba{
    PtString imie;
    PtString nazwisko;
};
```

```
persistent class IKlient: public IOsoba{
};
```

```
persistent class IKasjer: public IOsoba{
    int identyfikator;
    lset<IBilet*> bilety;
    int ranking();
};
```

```
persistent class IAktor: public IOsoba{
};
```

```
persistent class IRezyser: public IOsoba{
};
```

3.2. OBJECTSTORE

ObjectStore oparty jest na języku programowania C++. Model danych, stanowiący podstawę ObjectStore jest w zasadzie modelem obiektów C++. Obiekty każdego typu można podzielić na trwałe oraz ulotne (niektóre obiekty tego samego typu mogą być trwałe, a inne ulotne). Obiekty trwałe zapamiętane mogą zostać przeniesione do pamięci i od tej pory traktowane podobnie jak obiekty ulotne [OBJECTSTORE].

3.2.1. TYPY

W ObjectStore typy podstawowe odpowiadają typom podstawowym w C++. Są to:

(1) typy atomowe:

- całkowite: short, int, long, signed short, signed int, signed long, unsigned short, unsigned int, unsigned long,
- zmiennoprzecinkowe: float, double, long double, long float,
- znakowe: char, signed char, unsigned char,
- tekstowy: brak, ale najczęściej jest reprezentowany jako tablica znaków,
- logiczny: brak,
- czasowy: brak,
- przedział czasowy: brak.

(2) typ wyliczeniowy (**enum**);

(3) kolekcje w ObjectStore są rozszerzeniem języka C++. Wyróżniamy następujące kolekcje:

- zbiór: os_set, os_Set;
- wielozbiór: os_bag, os_Bag;
- lista: os_list, os_List, os_nlist, os_nList;
- tablica: os_array, os_Array;
- słownik: os_Dictionary.

Podczas definicji kolekcji, których nazwy pisane są z małych liter (np. `os_set`) nie podaje się typu obiektów, jakie będą przechowywane w kolekcji. Natomiast dla pozostałych kolekcji (np. `os_Set`) podaje się typ.

Struktury są reprezentowane przez klasy.

3.2.2. DZIEDZICZENIE

Dziedziczenie w ObjectStore jest realizowane podobnie jak w C++. Przy definicji klasy–potomka podaje się klasy–rodziców, z których dziedziczy potomek. Dziedziczyć można w sposób publiczny, chroniony oraz prywatny. Możliwe jest dziedziczenie używające mechanizmów późnego wiązania jak również wielodziedziczenie.

3.2.3. ZWIĄZKI

Deklaracje związków (relacji asocjacji), podobnie jak relacji dziedziczenia, są realizowane tak jak w języku programowania C++.

Mogą wystąpić relacje jedno lub dwukierunkowe. W przypadku relacji dwukierunkowych w obu klasach biorących udział w asocjacji dodaje się dodatkowy atrybut. W relacjach jednokierunkowych, dodatkowy atrybut dodaje się tylko w jednej z klas.

W przypadku relacji jeden do jeden oraz wiele do jeden w klasie dodaje się atrybut – wskaźnik na obiekt klasy po drugiej stronie asocjacji. W przypadku relacji wiele do wiele oraz wiele do jeden, dodaje się atrybut typu jednej z kolekcji (zbiór, wielozbiór, lista lub tablica).

3.2.4. WIĘZY INTEGRALNOŚCI

W ObjectStore brak jest mechanizmu kontrolującego więzy integralności.

3.2.5. METODY

ODL tworzy jedynie sygnatury metod, ale nie mówi nic na temat ich implementacji. W ObjectStore metody deklaruje i definiuje się podobnie jak w C++. W deklaracji metod podaje się typ zwracanego wyniku, nazwę metody oraz listę parametrów.

3.2.6. PRZYKŁAD DEFINICJI BAZY W OBJECTSTORE

Przykład dotyczy bazy danych przedstawionej na rysunku 1 zbudowanej dla wycinka rzeczywistości przedstawionego w rozdziale 2.3. Poniżej przedstawiona jest definicja klas w ObjectStore:

```

class IFilm{
    char *tytul;
    int rok;
    int dlugosc;
    os_Set<IAktor*> &aktorzy;
    IRezyser rezyser;
    os_Set<ISeans*> &seanse;
};

class ITimestamp{

public:
    ITimestamp(int godzina, int minuta, int sekunda, int dzien, int miesiac, int
rok);
private:
    int godzina;
    int minuta;
    int sekunda;
    int dzien;
    int miesiac;
    int rok;
};

class ISeans{
    ITimestamp poczatek;
    IFilm film
    os_Set<IBilet*> &bilety;
};

class IBilet{
    enum rodzaj{ulgowy, normalny} rodzajBiletu;
    class ILokalizacja{int rzad, int fotel} miejsce;
    ISeans seans
    IKlient klient;
};

class IOsoba{
    char *imie;

```

```

        char *nazwisko;
    };

    class IKlient: public IOsoba{
    };

    class IKasjer: public IOsoba{
        int identyfikator;
        os_Set<IBilet*> &bilety;
        int ranking();
    };

    class IAktor: public IOsoba{
    };

    class IRezyser: public IOsoba{
    };

    ITimestamp(int godzina, int minuta, int sekunda, int dzien, int miesiac, int rok){
        // tutaj powinno być dokonane sprawdzenie poprawności
        // atrybutów.
    }
    int IKasjer::ranking(){
        // definicja metody
    }

```

Deklaracja relacji asocjacji nie mówi, czy relacja jest jedno- czy dwukierunkowa. W przypadku relacji dwukierunkowych, nie wiadomo jak nazywa się atrybut po drugiej stronie asocjacji.

W przykładzie podano jedynie kolekcje typu `os_Set`, jednak mogą występować dowolne inne.

W języku programowania C++ nie istnieją typy czasowe, dlatego należy je stworzyć. Dodatkowe ograniczenia nakładane na atrybuty (np. dla typu czasowego) powinny być sprawdzane np. w konstruktorach.

4. ZALETY I WADY ODL

Analiza języka ODL oraz jego implementacji wykazała istnienie poniżej wymienionych zalet i wad.

Zalety:

- (1) ODL posiada wszystkie cechy technologii obiektowej (istnienie złożonych interfejsów, enkapsulacja atrybutów i metod, relacja dziedziczenia) oraz baz danych (trwałość danych, oddzielenie logicznego i fizycznego poziomu danych),
- (2) Istnieje możliwość definicji relacji dwukierunkowych,
- (3) Zaimplementowano niektóre ograniczenia (np. klucze).

Wady:

- (1) Trudność w implementacji relacji dwukierunkowych w istniejących językach programowania obiektowego,
- (2) Wiele mechanizmów pozostawiono implementacjom (np. ograniczenia dotyczące więzów integralności referencyjnej, definicje metod),
- (3) Składnia ODL jest niepodobna do najczęściej używanych języków obiektowych baz danych,
- (4) Brak możliwości definiowania wyzwalaczy.

W kolejnym rozdziale przedstawiono propozycje zmian w ODL mających na celu zminimalizowanie opisanych w artykule wad ODL.

5. PROPOZYCJA ZMIAN W ODL

Języki C++, Java oraz C# są obecnie najbardziej popularne wśród języków programowania obiektowego. Pożądaną cechą standardu jest zatem podobieństwo do tych języków.

Wadą ODL jest brak definicji metod. Rozszerzenie standardu mogłoby polegać na umożliwieniu definiowania metod używając podstawowych konstrukcji znanych z języków programowania:

- (1) Instrukcje warunkowe:

```

if (<warunek>)
    <instrukcje>
else
    <instrukcje>,

switch(<zmienna>)
    case <wartość>: <instrukcje>
    ...
    default: <instrukcje>
  
```

- (2) Pętle:

```

for (<start>;<stop>;<krok>)
    <instrukcje>,
  
```

```

while (<warunek>)
    <instrukcje>,

do
    <instrukcje>
while (<warunek>),

for each <obiekt> in <kolekcja>
    <instrukcje>

```

- (3) instrukcja przypisania (=);
- (4) operatory porównania (=, <, <=, >, >=, !=);
- (5) Operatory arytmetyczne (+, -, *, /, %) i logiczne (and, or, not, xor);
- (6) Deklarowanie zmiennych lokalnych;
- (7) Funkcje wejścia/wyjścia (zapis/odczyt do/ze strumienia);
- (8) Możliwość wywoływania metod (własnych oraz innych obiektów);
- (9) Zgłaszanie i obsługa wyjątków.

Standard SQL3 definiuje specjalne procedury wyzwalane nazywane z ang. *triggerami*. Rozszerzają one możliwość definiowania warunków spójności danych. Wywoływane są automatycznie w wyniku wykonania instrukcji wstawiania, usuwania bądź modyfikacji danych. Mogą one także specyfikować operację wykonującą się zamiast operacji głównej.

W ODL nie istnieje mechanizm wyzwalaczy. Standard SQL3 definiuje wyzwalacze jako zbiór działań wywoływany poprzez zdarzenie (dodania, usunięcia oraz modyfikacji krotki). Pożądaną cechą ODL byłoby wprowadzenie wyzwalaczy na następujące zdarzenia:

- (1) Dodawanie obiektów do bazy;
- (2) Usuwanie obiektów z bazy;
- (3) Zmiana atrybutów obiektów.

Realizacja wyzwalaczy mogłaby wyglądać następująco: każda klasa mogłaby zawierać dodatkowo metody (np. onCreate, onDelete, onUpdate), które byłyby wywoływane po zajściu odpowiedniego zdarzenia.

ODL nie daje możliwości nadawania wartości domyślnych atrybutom oraz parametrom wywołania metod. Przydatną własnością byłoby umożliwienie inicjalizowania tych wartości.

Niekiedy konieczne jest przedstawienie zależności pomiędzy obiektami (np. każdy film musi być wyświetlany co najmniej dwa razy i co najwyżej pięć razy). ODL nie umożliwia przedstawienia takich więzów integralności. Kontrola więzów integralności może być zapewniona dzięki zastosowaniu mechanizmu wyzwalaczy.

6. PODSUMOWANIE

W artykule przedstawiono implementację Object Definition Language (ODL) w wybranych systemach zarządzania obiektowymi bazami danych. Analiza samego języka oraz jego implementacji wykazała potrzebę wprowadzenia zmian w standardzie.

Autorzy artykułu pokazali możliwą drogę rozszerzenia standardu.

LITERATURA

- | | |
|---------------|---|
| [APSIN] | http://www.icie.com.pl/APSIN/m_iv.doc |
| [DATE 2000] | Date C.J., <i>Wprowadzenie do systemów baz danych</i> . Warszawa, WNT, 2000 |
| [FASTOBJECT] | http://www.fastobjects.com |
| [LAUSEN 2000] | Lausen G., Vossen G., <i>Obiektowe bazy danych</i> . Warszawa, WNT, 2000 |
| [OBJECTSTORE] | http://www.objectstore.com/ |
| [ULLMAN 2001] | Ullman J., Widom J., <i>Podstawowy wykład z systemów baz danych</i> . Warszawa, WNT, 2001 |
| [VAHE] | http://galaxy.uci.agh.edu.pl/~vahe/obdfaq/faq-obd.htm |
| [ZOOKO] | http://www.eros-os.org/pipermail/e-lang/2001-February/004594.html |

REALIZATION OF ODL FUNCTIONALITY IN OBJECT DATABASE MANAGEMENT SYSTEMS

The article describes chosen aspects of Object Definition Language. The article compares ODL with data definition languages which are used in ObjectStore and FastObject. Advantages and disadvantages of ODL are presented. Authors show possible development ways of the standard.

Zygmunt MAZUR*

Piotr MICHAŁEC*

ZASTOSOWANIE OPISU TEMPORALNO- PROBABILISTYCZNEGO DO HARMONOGRAMOWANIA TRANSAKCJI

Systemy czasu rzeczywistego znajdują coraz szersze zastosowanie w praktycznych rozwiązaniach. Wynika to z faktu, że istnieje potrzeba tworzenia rozproszonych aplikacji, które muszą spełniać określone wymagania nałożone na czas wykonywania zadań. W pracy zaprezentowano definicję opisu temporalno-probabilistycznego transakcji oraz przedstawiono algorytm przydzielania priorytetów działający w oparciu o ten opis. Zaproponowano zastosowanie opisu temporalno-probabilistycznego transakcji do opisu otwartych systemów czasu rzeczywistego ogólnego zastosowania.

1. WPROWADZENIE

Systemy czasu rzeczywistego znajdują coraz szersze zastosowanie praktyczne, obecnie łączą one dziedziny dotychczas traktowane oddzielnie. System bazy danych czasu rzeczywistego jest integracją systemu zarządzania bazą danych i systemu czasu rzeczywistego. Dziedziczy on wiele własności zarówno z systemów czasu rzeczywistego jak i z systemów baz danych. Wzrost zainteresowania takimi systemami wynika z faktu, że coraz częściej istnieje potrzeba tworzenia aplikacji zorientowanych na przetwarzanie ogromnej ilości danych, które muszą spełniać określone wymagania nałożone na czas wykonywania zadań, gdzie wartości danych zależą od czasu i mają ograniczony czas ważności.

Zastosowania systemów czasu rzeczywistego obejmują między innymi systemy kontroli ruchu lotniczego, zintegrowane systemy produkcyjne, systemy wojskowe, zarządzania sieciami komputerowymi i telekomunikacyjnymi, jak również aplikacje bankowe, giełdowe czy multimedialne.

Ze względu na rozproszoną naturę wymienionych systemów ich podstawowym zadaniem jest przetwarzanie transakcji rozproszonych. Ponadto w systemach czasu rzeczywistego przetwarzanie transakcji związane jest z nałożonymi na nie ograniczeniami czasowymi. Ograniczenia czasowe mogą dotyczyć zarówno

* Politechnika Wrocławska, Wydział Informatyki i Zarządzania, Instytut Informatyki Stosowanej,
50-370 Wrocław, Wyb. Wyspiańskiego 27, {Zygmunt.Mazur, Piotr.Michalec}@pwr.wroc.pl

przetwarzanych transakcji jak i terminu ważności danych. System musi przetwarzać transakcje i zwracać wyniki przed wystąpieniem punktu krytycznego (ang. *deadline*).

Głównym celem systemu czasu rzeczywistego jest, zatem zwiększenie liczby transakcji, które wykonałyby się przed wystąpieniem ich punktów krytycznych.

Celem niniejszego artykułu jest przedstawienie możliwości zastosowania opisu temporalno-probabilistycznego transakcji, w algorytmach przydzielania priorytetów, do zwiększenia liczby przetwarzanych transakcji (w systemach baz danych czasu rzeczywistego) zakończonych powodzeniem.

2. WŁASNOŚCI TRANSAKCJI W SYSTEMACH CZASU RZECZYWISTEGO

Transakcja czasu rzeczywistego jest transakcją zdefiniowaną tak, jak w konwencjonalnych systemach, ale dodatkowo charakteryzowana jest przez trzy następujące własności [Prichard00].

Pierwsza własność określa typ transakcji ze względu na sposób przetwarzania danych:

- Transakcja pomiarowa (ang. *sensor transaction*) jest transakcją, która zapisuje dane do bazy danych, np. okresowo pobiera stan środowiska i otrzymane dane zapisuje do bazy danych.
- Transakcja modyfikująca (ang. *update transaction*) zarówno odczytuje jak i zapisuje dane do bazy danych. Zapis i odczyt może być okresowy lub stały.
- Transakcja tylko do odczytu (ang. *read-only*). Transakcja odczytuje dane z bazy danych. Może ona być, lecz nie musi, transakcją okresową.

Dru ga własność definiuje ograniczenie czasowe transakcji. Ograniczenie to może wynikać z wymagań spójności temporalnej zapisywanych danych, tzn. okresowość może wynikać z bezwzględnego czasu ważności zapisywanych danych lub z wymagań nałożonych na czas reakcji systemu.

Trzecia własność definiuje typ transakcji ze względu na konsekwencje wynikające z niespełnienia nałożonych na transakcje punktów krytycznych. Ze względu na konsekwencje transakcje możemy podzielić na [Ulusoy98]:

- silnie uwarunkowane czasowo (ang. *hard deadline transaction*). Poprawność operacji transakcji zależy od czasu zwrócenia rezultatu. W systemie musi istnieć taki algorytm harmonogramowania, który zagwarantuje, że wszystkie transakcje wykonają się przed wystąpieniem ich punktów krytycznych. W przeciwnym przypadku może mieć to poważne konsekwencje.
- średnio uwarunkowane czasowo (ang. *firm deadline transaction*) nie jest już tak ściśle powiązana z punktem krytycznym. Fakt, że transakcja nie wykona się w określonym czasie nie jest już tak katastrofalny. Jeśli jednak przekroczy ona swój punkt krytyczny – zostaje przerwana. Do systemu nie jest przekazywana wtedy żadna wartość.

- słabo uwarunkowane czasowo (ang. *soft deadline transaction*). Tego typu transakcje harmonogramowane są w oparciu o swoje punkty krytyczne. Wykonanie się transakcji przed wystąpieniem punktu krytycznego dalej jest głównym zadaniem. Jednak w przypadku tego typu transakcji nie ma żadnej gwarancji, że tak się stanie. Transakcja wykonuje się do czasu, aż zwróci pożądaný rezultat, bez względu na to czy przekroczyła już punkt krytyczny, czy też nie.

Transakcje w systemach czasu rzeczywistego związane są z występowaniem następujących atrybutów [Harista01]:

- czas przybycia transakcji do systemu A (ang. *arrival time*),
- czas (punkt) krytyczny D (ang. *deadline*),
- czas przetwarzania E (ang. *execution time*),
- maksymalne opóźnienie S (ang. *slack time*),
- funkcja wartości V (ang. *value function*),
- wartość krytyczna C (ang. *criticalness*),
- priorytet P (ang. *priority*).

Czas przybycia A i punkt krytyczny D określają ograniczenia czasowe związane z przetwarzaniem transakcji. Czas przetwarzania E stanowi przybliżoną wartość i zwykle nie jest znany przed wykonaniem transakcji. Atrybut S informuje o maksymalnym dopuszczalnym opóźnieniu rozpoczęcia wykonywania transakcji. Funkcja wartości V określa ważność zakończenia transakcji w danej chwili, a wartość tej funkcji dla punktu krytycznego nosi nazwę wartości krytycznej C . Priorytet P nadawany transakcji w celu optymalizacji efektywności przetwarzania transakcji.

Ograniczenia czasowe nałożone na transakcje pochodzą z dwóch źródeł. Pierwszym są wymagania dotyczące temporalnej spójności danych. Dane przechowywane w systemach czasu rzeczywistego określane są jako dane temporalne, których wartość ważna jest tylko przez pewien określony przedział czasu. Aktualny stan środowiska może być pobrany tylko w okresie czasu wyznaczonym przez ten przedział. Spójność temporalna jest osiągnięta tylko wtedy, gdy dane są dostępne w przedziale czasu, kiedy ich wartości są ważne. Można wyróżnić dwa rodzaje temporalnej spójności danych: wewnętrzną spójność temporalną wynikającą z powiązania wartości danych i zewnętrzną spójność temporalną, czyli spójność wynikającą ze specyfiki aplikacji.

Drugim źródłem ograniczeń czasowych są wymagania systemu dotyczące czasu reakcji na pewne zdarzenie, czyli rezultat działania systemu ma być zwrócony w określonym przedziale czasu (przed punktem krytycznym) [Porkka97].

3. HARMONOGRAMOWANIE TRANSAKCJI

Algorytmy harmonogramowania transakcji działają zwykle na zasadzie przydzielania transakcjom priorytetów i stosowania metod rozwiązujących konflikty w dostępie do zasobów systemowych. System może używać różnych strategii

przydzielania priorytetów dla różnych typów zasobów. Jeśli wystąpi jakiś konflikt w dostępie do zasobów systemowych, przydzielony priorytet jest wykorzystany do rozwiązania tego konfliktu. Jednak standardowe algorytmy harmonogramowania transakcji są nieodpowiednie dla systemów czasu rzeczywistego. Wynika to z faktu, że w systemach czasu rzeczywistego transakcje muszą być szeregowane z uwzględnieniem ich wartości krytycznej i ograniczeń nałożonych przez punkt krytyczny.

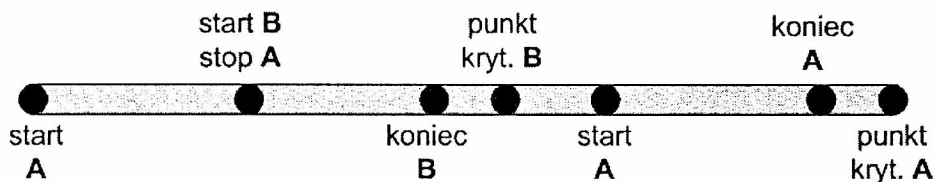
3.1. ALGORYTMY PRZYDZIELANIA PRIORYTETÓW

Podstawowym celem harmonogramowania w środowisku aplikacji czasu rzeczywistego jest wykonanie się wszystkich transakcji przed wystąpieniem ich punktów krytycznych. Ponieważ dostęp do zasobów jest wyłączny, w jednej chwili z zasobu może korzystać tylko jedna transakcja. Wykonanie pojedynczej transakcji zachowuje poprawny stan bazy danych czasu rzeczywistego, jednak wykonanie wielu transakcji w jednej chwili wymaga koordynacji. W tym celu wprowadza się procedury szeregujące, które wykorzystują mechanizm priorytetów. Wartość priorytetu ma tak wpływać na współbieżne wykonywanie transakcji, aby optymalizować parametry efektywnościowe systemu. Przedstawione w poprzednim rozdziale atrybuty transakcji mogą wpływać na wartość priorytetu w następujący sposób:

- wartość krytyczna – im wyższa wartość krytyczna transakcji, tym wyższy jest jej priorytet,
- punkt krytyczny – im wcześniejszy punkt krytyczny, tym wyższy jest priorytet transakcji,
- czas przybycia transakcji do systemu – transakcja, która wcześniej została zgłoszona do systemu powinna mieć wyższy priorytet, niż transakcja późniejsza,
- opóźnienie – im krótszy czas maksymalnego opóźnienia transakcji, tym wyższy jest jej priorytet,
- ilość obliczeń wykonanych do tej pory – transakcja, której przetwarzanie pochłonęło już dużą ilość obliczeń może mieć przydzielony wyższy priorytet. Systemy baz danych wymagają nie tylko zwolnienia zasobów, ale również operacji wycofania transakcji (ang. *rollback a transaction*). Łatwiej i z mniejszą stratą czasową jest przeprowadzenie transakcji do końca, jeżeli została przetworzona prawie w całości.

Konflikty dostępu są rozwiązywane na korzyść transakcji o wyższym priorytecie, oznacza to, że transakcja o niższym priorytecie jest restartowana, jeżeli w systemie pojawi się transakcja o wyższym priorytecie z zapotrzebowaniem na dostęp do tych samych danych (rysunek 1). Temu mechanizmowi towarzyszą również niekorzystne zjawiska. Możliwa jest sytuacja, kiedy transakcja o wyższym priorytecie restartuje transakcję o niższym priorytecie, a następnie sama jest restartowana przez kolejną transakcję, która ma jeszcze wyższy priorytet. Inna, również niekorzystna, sytuacja ma miejsce, kiedy transakcja o wyższym priorytecie restartuje transakcję o niższym

priorytecie, a następnie przekracza swój punkt krytyczny, po czym restartowana transakcja również przekracza swój punkt krytyczny (z powodu przymusowego restartu).



A - transakcja o niższym priorytecie

B - transakcja o wyższym priorytecie

Rys. 1. Wykonywanie transakcji w systemie z mechanizmem priorytetów

3.2. ALGORYTM PRZYDZIELANIA PRIORYTETÓW Z WYKORZYSTANIEM OPISU TEMPORALNO- PROBABILISTYCZNEGO TRANSAKCJI

Straty czasu powodowane przez sytuacje opisane powyżej można ograniczyć wprowadzając opis temporalno-probabilistyczny transakcji. Przez opis temporalno-probabilistyczny transakcji należy rozumieć prawdopodobieństwo p wykonania transakcji zgodnie z odpowiednią funkcją rozkładu σ w określonym przedziale czasu t i oznaczony jako trójka (t, p, σ) , w której:

t - oznacza przedział czasu,

p - jest prawdopodobieństwem wykonania transakcji,

σ - jest funkcją rozkładu prawdopodobieństwa.

Ilustracja graficzna opisu temporalno-probabilistycznego transakcji jest przedstawiona na rysunku 2.

Algorytm przydzielania transakcji z wykorzystaniem opisu temporalno-probabilistycznego transakcji określany jest wzorem:

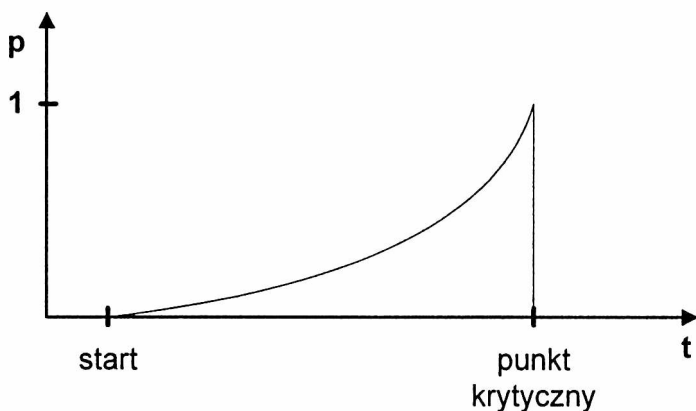
$$P = \max(p_i(t_i)),$$

gdzie: $p_i(t_i)$ – prawdopodobieństwo wykonania transakcji w czasie t_i , gdzie $t_i \leq D$ (D – czas krytyczny transakcji).

System zarządzający transakcjami może nie dopuszczać do wykonywania transakcji, które mają małe prawdopodobieństwo zakończenia powodzeniem przed punktem krytycznym dla tej transakcji (w przedziale czasu od momentu wystąpienia zapotrzebowania na transakcję do jej punktu krytycznego).

Ograniczenie strat czasu i zwiększenie liczby transakcji zakończonych powodzeniem jest uzyskiwane poprzez eliminację transakcji, które mają małe (lub żadne) szanse, aby zakończyć się powodzeniem. Transakcje o niższych priorytetach

nie będą restartowane przez transakcje, które nie mają szans zakończyć się przed swoim punktem krytycznym, a zatem czas przewidziany na ich wykonanie wzrośnie. Możliwa jest adaptacja tego systemu tak, aby dopuszczać transakcje o małych prawdopodobieństwie na zakończenie, ale z niższym priorytetem. Wówczas transakcje te zachowują swoje szanse na wykonanie się (jeżeli nie będzie założonej blokady na dane, do których chcą uzyskać dostęp), ale nie blokują innych transakcji z większymi prawdopodobieństwami.



Rys. 2. Przykładowy rozkład prawdopodobieństwa dla wykonania transakcji

Zastosowanie opisu temporalno-probabilistycznego transakcji i klasyfikowanie według kryterium prawdopodobieństwa zakończenia powodzeniem może być traktowane jako uzyskanie dodatkowej informacji dla znanych, zaawansowanych algorytmów zarządzania współbieżnym wykonywaniem transakcji w systemach czasu rzeczywistego.

4. ZASTOSOWANIE OPISU TEMPORALNO- PROBABILISTYCZNEGO TRANSAKCJI DO OTWARTYCH SYSTEMÓW CZASU RZECZYWISTEGO

Rozproszona architektura systemów czasu rzeczywistego ogólnego zastosowania, zwana architekturą otwartą, ma bardzo wiele zalet. Jej zastosowanie pozwoliłoby na dynamiczne łączenie niezależnych aplikacji czasu rzeczywistego występujących na jednej lub wielu maszynach, również komunikujących się ze sobą poprzez Internet. Klient systemu mógłby uruchamiać wiele aplikacji ze swoimi wymaganiami czasu wykonania.

Obecnie występuje kilka utrudnień związanych z otwartą architekturą wspierającą przetwarzanie w systemach czasu rzeczywistego. Architektura taka musi radzić sobie z nieznaną charakterystyką systemu:

- hardware'ową, taką jak prędkość procesora, pamięci cache, RAM, magistral i urządzeń I/O,
- oraz software'ową: połączenia między aplikacjami, ich dostęp do zasobów i wymagania czasowe

Te niewiadome wpływają na niemożność utworzenia dokładnego schematu harmonogramowania. Dlatego systemy rozproszone potrzebują bardziej elastycznych podejść niż te typowe, używane dzisiaj do budowania niezmiennych systemów czasu rzeczywistego. Wszystkie te wymagania, dla rozproszonych systemów otwartych, stanowią znaczące wyzwanie naukowe.

Zastosowanie opisu temporalno-probabilistycznego charakterystyki działania systemów czasu rzeczywistego jest analogiczne jak dla opisu temporalno-probabilistycznego pojedynczej transakcji przedstawionego w rozdziale 3.2.

Ponieważ klient systemu czasu rzeczywistego ogólnego zastosowania z reguły nie zna jego charakterystyki, należy wprowadzić opis systemu przez rozkład prawdopodobieństwa w przedziale czasu potrzebnym do zakończenia operacji wykonywanej przez system. Wówczas klient komunikując się z systemami czasu rzeczywistego otrzymuje najpierw ich charakterystyki w postaci opisu temporalno-probabilistycznego (podobne jak przedstawiono na rysunku 2). Na podstawie tej informacji może zdecydować, z których usług systemów może skorzystać, tak, aby przeprowadzić wszystkie potrzebne operacje ze spełnieniem założonych ograniczeń czasowych.

System czasu rzeczywistego ogólnego zastosowania może również udostępniać wykonanie wielu różnych, niezależnych operacji i dla każdej z nich udostępniać właściwe charakterystyki temporalno-probabilistyczne.

Przedstawione wykorzystanie opisu temporalno-probabilistycznego pojedynczego systemu czasu rzeczywistego pozwala na dynamiczne łączenie niezależnych systemów komunikujących się ze sobą. Działanie takie zapewnia prawidłowe funkcjonowanie całego systemu złożonego z wielu aplikacji.

5. PODSUMOWANIE

W pracy przedstawiono własności systemów baz danych czasu rzeczywistego, podano własności transakcji w systemach czasu rzeczywistego, oraz wskazano źródła ograniczeń czasowych. Przedstawiono, również, krótką charakterystykę algorytmów przydzielania priorytetów.

W dalszej części pracy zaprezentowano definicje opisu temporalno-probabilistycznego transakcji oraz algorytmu przydzielania priorytetów wykorzystującego zalety tego opisu. Następnie przedstawiono propozycję wykorzystania opisu

temporalno-probabilistycznego transakcji do opisu otwartych systemów czasu rzeczywistego ogólnego zastosowania.

LITERATURA

- [Dekhtyar01] Dekhtyar A., Ross R., Subrahmanian V., *Probabilistic Temporal Databases, part I: Algebra*. ACM Transactions on Database Systems vol. 26, str. 41–95, New York, 2001.
- [Dubois89] Dubois D., Prade H., *Processing fuzzy temporal knowledge*. IEEE Transactions on Systems, Man, and Cybernetics, str. 729–744, New York, 1989.
- [Dutta89] Dutta S., *Generalized events in temporal databases*. Proceedings of the 5th International Conference on Data Engineering, str. 118–126, Los Angeles, 1989.
- [Haritsa01] J.R. Haritsa, *Performance Analysis of Real-Time Database Systems*, USA, 2001.
- [Krzyza98] Krzyżagórski P., *Systemy baz danych czasu rzeczywistego*, [w:] *Systemy czasu rzeczywistego*, Oficyna Wydawnicza Politechniki Wrocławskiej, Wrocław, 1997.
- [Macyna00] Macyna W., Mazur Z., *Temporalno – obiektowe modele baz danych*. Prace naukowe Wydziałowego Zakładu Informatyki Politechniki Wrocławskiej – Bazy Danych 1, strony 19–36, Oficyna Wydawnicza Politechniki Wrocławskiej, Wrocław, 2000.
- [O’Neil99] O’Neil P. E., Ramamritham K., Pu C., *A two-phase Approach to Predictably Scheduling Real-Time Transactions*, Prentice-Hall Publisher, New Jersey, 1999.
- [Porkka97] Porkka P., *Overview of Real-Time Databases*, USA 1997.
- [Prichard00] Prichard J.J., Fortier P.J., Fay W., *Flexible Real-Time SQL Transaction*, IEEE Real-Time Systems Symposium, Puerto Rico, 2000.
- [Ulusoy98] Ulusoy O., Belford G.G., *Real-Time Transaction Scheduling in Database Systems*, Elsevier Science, Oxford, 1998.

APPLYING OF TEMPORAL-PROBABILISTIC DESCRIPTION OF TRANSACTION TO TRANSACTIONS’ SCHEDULING

Real time systems are used wider and wider in practical solutions. It results from fact, that there is need of creating distributed applications, which have to deal with specific requirements applied to task’s execution time. In this paper, there is presented definition of temporal - probabilistic description of transaction and priority assigning algorithm, which works basing of that description. There is proposition of applying temporal-probabilistic description of transaction to description of open general- purpose real time systems.

Aleksander KLOSOV*
Marcin BEDNARSKI

METODY INŻYNIERII WSTECZNEJ W SYSTEMACH OBIEKTOWYCH BAZ DANYCH

Mechanizmy inżynierii wstecznej dotyczące odwzorowania kodu źródłowego na postać modeli pojęciowych zajmują ważne miejsce wśród współczesnych problemów baz danych. Ważnym problemem jest odwzorowanie obiektowych struktur danych w kodzie źródłowym na postać graficzną, która ukazuje wyższy poziom abstrakcji. W części praktycznej publikacji opisane zostaną metody generowania diagramów klas w języku UML na podstawie analizy semantycznej schematu obiektowej bazy danych w kodzie źródłowym języka C++. W pracy przedstawiono przykładową aplikację CASE z mechanizmem inżynierii wstecznej wykonanej w ramach pracy magisterskiej na Wydziale Elektroniki Politechniki Wrocławskiej [BEDN04].

1. WPROWADZENIE

Obecnie w tworzeniu oprogramowania bardzo popularne jest programowanie zorientowane obiektowo. Od współczesnych aplikacji użytkownicy oczekują coraz więcej, zatem są one coraz bardziej złożone. Tworzenie dużych, złożonych aplikacji wymaga pracy sztabu ludzi: projektantów, programistów, czy menedżerów zarządzających projektem. Sprawne tworzenie aplikacji, czy to projektu, czy też implementacji, pociąga za sobą potrzebę użycia odpowiednich narzędzi, które wspierają projektantów w ich pracy.

Narzędzia pomocne w procesie tworzenia oprogramowania można ogólnie podzielić na dwie grupy: programistyczne i wspierające pracę projektantów. Do pierwszej grupy zalicza się wszelkiego rodzaju narzędzia typu RAD (ang. *Rapid Application Development*) takie jak np. Borland C++ Builder, czy J-Builder, środowiska programistyczne (np. Microsoft Visual Studio), czy nawet zwykłe edytory tekstu. Do grupy drugiej należą wszelkiego rodzaju narzędzia CASE (ang. *Computer Aided Software Engineering*), wspomagające tworzenie projektów, schematów baz danych, specyfikacji lub dokumentacji.

* Politechnika Wrocławska, Wydział Elektroniki, Instytut Cybernetyki Technicznej, 50-370 Wrocław, Wyb. Wyspiańskiego 27, Aleksander.Klosow@pwr.wroc.pl

Tworzenie projektu oprogramowania zorientowanego obiektowo wymaga odpowiedniego języka, który w przejrzysty sposób pozwoli opisać strukturę i działanie aplikacji. Obecnie bardzo popularnym językiem używanym do tworzenia projektów i dokumentacji programów zorientowanych obiektowo jest język UML (ang. *Unified Modeling Language*) [BOOC02, FOWL99].

Sposób pracy, w którym najpierw tworzy się projekt, a potem go implementuje, jest typowy dla inżynierii do przodu (ang. *forward engineering*). Jednak w trakcie kolejnych iteracji tworzenia programu istnieje potrzeba powrotu do fazy projektu z fazy implementacji. Takie podejście charakterystyczne jest dla inżynierii wstecznej (ang. *reverse engineering*). Typowym zadaniem inżynierii wstecznej w tworzeniu oprogramowania jest odtworzenie struktury klas na podstawie kodu źródłowego.

Inżynieria wsteczna odgrywa istotną rolę również przy projektowaniu lub utrzymaniu systemów z bazami danych [PREM94]. Ważną rolę w mechanizmach odtwarzania modelu danych z kodu źródłowego odgrywa sposób prezentacji informacji na wyższym poziomie abstrakcji [KLOS03]. Schemat obiektowej bazy danych reprezentuje sobą diagram klas [NEIS02]. Celem danej pracy jest pokazanie możliwości prezentacji w notacji UML schematu obiektowej bazy danych w oparciu o kod źródłowy w języku C++ z wykorzystaniem mechanizmów inżynierii wstecznej. Zakłada się, że kod źródłowy został napisany poprawnie i przedstawiony w postaci plików nagłówkowych. Przyjmuje się również, że komentarze w kodzie nie zawierają żadnych sugestii na potrzeby inżynierii wstecznej. Celem inżynierii wstecznej jest przedstawienie schematu bazy danych na wyższym poziomie abstrakcji, w postaci graficznej, ułatwiając tym samym proces analizy i ponowne projektowanie modelu danych.

W pracy przedstawiono również analizę pojęcia tarcia w inżynierii wstecznej oraz charakterystykę istniejących narzędzi CASE z inżynierią wsteczną. W rozdziale 5. zaprezentowano własny system CASE, który generuje oraz pozwala zarządzać diagramami klas w UML w oparciu o pliki języka C++. Zaimplementowany system może być wykorzystany w trakcie projektowania obiektowych baz danych lub w procesie edukacyjnym na potrzeby zajęć laboratoryjnych z baz danych.

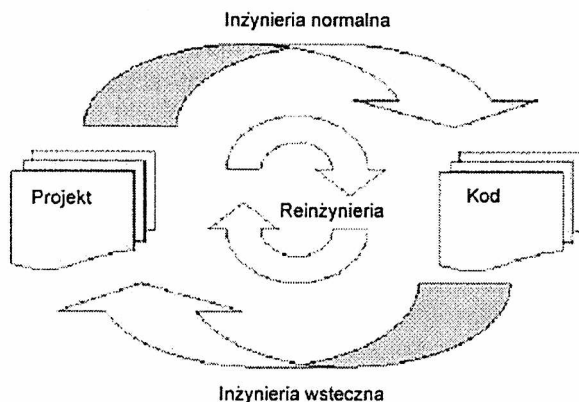
2. INŻYNIERIA WSTECZNA

Iteracyjny sposób pisania programów zamienia tworzenie oprogramowania w proces ciągłej cyklicznej inżynierii. Elementami inżynierii cyklicznej oprogramowania są: reinżynieria, inżynieria do przodu oraz inżynieria wsteczna. Chikofsky [CHIK90] sformułował następujące definicje:

Reinżynieria (ang. *reengineering*) jest to badanie i przebudowa systemu, w celu eliminacji błędów, poprawienia wydajności i zmiany produktu w taki sposób, by spełniał dodatkowe, bądź nowe wymagania.

Inżynieria do przodu (ang. *forward engineering*) jest to tradycyjny sposób przemieszczania się z wysokiego poziomu abstrakcji i logicznie niezależnego projektu w kierunku fizycznej implementacji systemu

Inżynieria wsteczna (ang. *inverse engineering*) – jest to proces analizy systemu, który poprzez identyfikację jego komponentów i związków między nimi tworzy inną formę jego reprezentacji, często na wyższym poziomie abstrakcji



Rys. 1. Cykl życia oprogramowania w reinżynierii

W tradycyjnym podejściu inżynierii do przodu, zanim twórca oprogramowania będzie w stanie modyfikować system, musi poświęcić trochę sił na przestudiowanie kodu źródłowego by zrozumieć strukturę i architekturę systemu. Zadanie inżynierii wstecznej jest również nazywane przywracaniem projektu. Jeśli chce się wspierać inżynierów w ich pracy, trzeba najpierw poznać wszechstronnie listę ich celów. Lista celów sporządzona przez Chikofsky'ego [CHIK90], znajduje się poniżej.

- **Poradzenie sobie ze złożonością.** Objęcie umysłem dużej ilości kodu źródłowego jest trudnym zadaniem. Jednym ze sposobów radzenia sobie ze złożonością jest redukcja objętości poprzez ukrycie niektórych szczegółów i wypunktowanie innych. Innym sposobem jest stworzenie reprezentacji o różnym znaczeniu.
- **Wygenerowanie wielorakich reprezentacji.** W zależności od punktu widzenia istnieje potrzeba tworzenia różnych widoków i reprezentacji systemu.
- **Przywrócenie utraconej informacji.** Zalicza się do tego ponowne tworzenie dokumentacji, jeśli jest ona nie kompletna, lub jeśli nie ma jej wcale. Często zdarza się, że istniejąca dokumentacja jest oparta o pierwotny projekt, i nie jest adekwatna do obecnej postaci systemu. Czasami podczas tego procesu firmy odkrywają zapomniane szczegóły, które zostały pominięte przy kodowaniu.
- **Wykrywanie efektów ubocznych.** Bez przejrzystego obrazu całościowej architektury systemu i bez wsparcia narzędzi do jego analizy, często bardzo trudno jest zidentyfikować zależności między komponentami systemu. W szczególności kiedy modyfikuje się oprogramowanie, w którym wiele

technologii musi razem współpracować. Ciężko jest wykryć ich wpływ na cały system.

- **Synteza wyższych poziomów abstrakcji.** Wyższe poziomy abstrakcji mogą ewoluować w procesie inżynierii wstecznej, poprzez wprowadzenie nowych pomysłów i wiedzy użytkowników. Jako przykład można podać agregację artefaktów do kontenerów poprzez inspekcję „wirtualnych przestrzeni nazw” – ukrytych semantyk biorących się ze schematu nazw.
- **Ułatwianie ponownego użycia.** Twórcy oprogramowania unikają ponownego użycia czegoś, czego nie rozumieją. Wyższe poziomy abstrakcji pomagają zredukować wysiłek konieczny do zrozumienia zasad działania komponentów.

Głównym powodem, dla którego sformułowano powyższe cele, jest potrzeba tworzenia ciągłych modyfikacji tworzonego systemu. To zaś pociąga za sobą konieczność użycia inżynierii wstecznej na początku każdej fazy cyklu reinżynierii. Aby zachować dobry kształt systemu często jego przeróbkom towarzyszą próby restrukturyzacji. Restrukturyzacja jest obecnie znana, szczególnie w kontekście technologii obiektowych, jako *refabrykacja*.

Refabrykacja jest wewnętrzną modyfikacją systemu na korzyść projektu, bez zmian jego zewnętrznego zachowania – Martin Fowler [FOWL99].

Ulepszanie projektu prowadzi w konsekwencji do powstania systemów, które są prostsze do zrozumienia, kodowania i rozbudowy (rozszerzania). Czasami dobrym powodem by zastosować refabrykację mogą być aspekty wydajnościowe lub względy bezpieczeństwa. Dobrze jest przemyśleć, czy zastosowanie *refabrykacji*, uczyni dokonywanie przeróbek prostszym i czy projekt systemu także nie wymaga przebudowy, po dokonaniu zmian.

Podsumowując powyższe można wskazać listę powodów, dla których istnieje inżynieria wsteczna, oraz dlaczego jej znaczenie i użycie z każdym rokiem zwiększa się [HEML01]:

- Osobista edukacja;
- Zrozumienie ograniczeń i defektów w oprogramowaniu;
- Zrozumienie i opracowanie defektów urządzeń;
- Umożliwienie współpracy produktu z innym produktem;
- Umożliwienie wymiany danych produktu z innym produktem;
- Nauczenie się zasad, którymi kierowała się konkurencja przy projektowaniu;
- Wykrycie, czy inna firma ukradła i wykorzystała czyjś kod źródłowy;
- Wykrycie, czy produkt jest zdolny wykonać to co głosi jego reklama.

3. TARCIE W INŻYNIERII WSTECZNEJ

Przystępując do budowy systemu z inżynierią wsteczną należy uświadomić sobie te czynniki, które mogą prowadzić do pogorszenia efektywności docelowego systemu. Jednym z takich pojęć jest tarcie w inżynierii wstecznej.

Sądzi się, że warunkiem koniecznym pomyślnego procesu inżynierii wstecznej jest posiadanie przejrzystego planu działań. Na samym początku należy pomyśleć czego właściwie chce się dowiedzieć. Kiedy już zdefiniuje się konkretne cele, potrzeba dobrych narzędzi wspierających sprawną eksplorację systemów. Aby określić główne czynniki mające wpływ na wydajność systemu, należy najpierw zidentyfikować typowe źródła braku wydajności. To prowadzi do pojęcia *tarcia* w inżynierii wstecznej. W dalszej części zostanie wyjaśnione co ten termin oznacza i dlaczego tarcie powinno się redukować.

W procesie inżynierii wstecznej potrzebne jest wsparcie narzędzia, które poradzi sobie z dużą ilością złożonej informacji. Typowe narzędzia dla inżynierii wstecznej pomagają zredukować złożoność i zagłębiają się w części struktury systemu, poprzez tworzenie jego reprezentacji w innej formie, często na wyższym poziomie abstrakcji. Tak czy inaczej zorientowanie się w tej raczej statycznej reprezentacji jest rzeczą trudną. W czasie eksploracji dużych systemów łatwo można się zagubić i stracić dużo czasu, nie mogąc dostać się w określone miejsce. Nazywa się to stratą wydajności spowodowaną przez narzędzie, czyli *tarcie* [SCHW02].

Tarcie w inżynierii wstecznej jest to dodatkowy, spowodowany przez narzędzie, sztuczny wysiłek, który jest zbyteczny, ale konieczny by umożliwić określoną czynność, w celu osiągnięcia określonego celu.

Poniżej opisano sześć głównych przyczyn *tarcia*.

Niekompletność modelu i cech. Aby można było znaleźć informację, musi być spełnione najważniejsze kryterium: kompletność modelu i cech. Jest to warunek konieczny do poruszania się po prawdziwych referencjach i zależnościach pomiędzy częściami składowymi systemu. Nie można dostać się w jakieś miejsce, jeśli ono nie istnieje. Ważne jest również, aby cała informacja była dostępna z poziomu jednego narzędzia. Przełączanie się pomiędzy kilkoma różnymi narzędziami w celu znalezienia brakujących porcji informacji powoduje wielkie *tarcie*.

Pośredniość. Czasami wiadomo dokładnie dokąd chce się iść, ale żeby się tam dostać, potrzeba wielu dodatkowych czynności. Brak możliwości bezpośredniego dostępu do istotnej informacji jest kosztowny, nużący i demotywujący, w szczególności kiedy jakieś zadanie należy wykonać wiele razy. Wsparcie dla szybkiego dotarcia od punktu A do punktu B dostarcza skrótów i zawiera eliminację przeszkód.

Przesycenie. Trudno jest poradzić sobie z dużą ilością jednostek informacji występujących równolegle. Sposobem na zredukowanie złożoności (i szansy na zagubienie się) jest redukcja tej informacji, która jest prezentowana w tym samym czasie, na przykład przez zastosowanie filtrów.

Odwroćenie uwagi. Łatwo zagubić się w systemie, kiedy coś rozprasza uwagę. Może to być jakiś fragment systemu, o którym ma się błędne pojęcie co do jego funkcji lub znaczenia dla bieżącego zadania. Dobrym przykładem może być przeładowane menu z wieloma rzadko używanymi funkcjami, które rozprasza użytkownika. Dobrze zaprojektowane narzędzie redukuje ilość kroków potrzebnych do wykonania jakiejś operacji do kilku. Często dla podobnych operacji kroki są te same. Po prostu dynamiczne menu czyni złożone akcje łatwiejszymi. Automatyczne i

natychmiastowe wyświetlanie wszystkich istotnych informacji na temat określonej jednostki (np. zaznaczonej klasy) powoduje, że dodatkowe czynności w celu jej uzyskania są zbędne. Dodatkowo przydatne jest zaimplementowanie mechanizmu historii operacji, żeby łatwo można było odwołać wprowadzone zmiany.

Degradacja wiedzy. By wspierać ponowne użycie system musi pamiętać czego już się dowiedział użytkownik, a użytkownik musi mieć możliwość dostępu do tej wiedzy w przyszłości. Potrzebny jest pewien rodzaj pamięci by zredukować możliwość utraty odkrytej wiedzy. W najnowocześniejszych narzędziach do inżynierii wstecznej, zakładki i temu podobne mechanizmy są rzadkością. Większość sesji inżynierii wstecznej, nie zostaje przechowana w żadnej pamięci trwałej. Jedyne co pozostaje to statyczne raporty i rysunki. Użytkownicy wyrzucają wszystko po każdym jednokrotnym użyciu. Użytkownicy muszą zaczynać od zera, kiedy chcą dowiedzieć się tych samych rzeczy w późniejszym czasie. Natychmiastowe wyrzucanie spostrzeżeń jest nieefektywne. Podsumowując, użytkownicy powinni być w stanie wygenerować szczegółowe raporty na temat ich sesji. Raporty te muszą być przyswajalne i rozszerzalne. Rezultaty ich pracy powinny być zapisywane w postaci, która łatwo da się zaimportować w celu dalszego użycia, także przez inne narzędzia. Jest to tym bardziej ważne, jeśli użytkownicy pracują w grupach, gdzie niezbędne jest współdzielenie i wymiana informacji, wyników prac i sesji między kolegami.

Brak klasyfikacji. Bez dobrej koncepcji klasyfikacji, ciężko rozróżnić różne rodzaje jednostek informacji. Naturalne jest, że nie każda jednostka jest tak samo ważna, Pomocne może okazać się sklasyfikowanie ich według ich ważności, aspektu, czy kontekstu. Zazwyczaj nie wiadomo, które części systemu już zostały zbadane, a które jeszcze nie tknięte. Sprawozdanie z odkrytych części systemu może również zostać wyrażone poprzez klasyfikację.

4. NARZĘDZIA KOMPUTEROWE Z INŻYNIERIĄ WSTECZNĄ

Jako, że inżynieria wsteczna jest powszechnie stosowana przez twórców aplikacji komputerowych, powstało wiele narzędzi, które umożliwiają za jej pomocą odtwarzanie wyższych poziomów abstrakcji w oparciu o kod źródłowy. Wspomniany wcześniej *Rational Rose* również posiada mechanizmy pozwalające na podstawie kodu źródłowego wygenerować diagram klas w notacji UML. Podobny rezultat można uzyskać łącząc Microsoft Visual Studio C++ z programem *Visio*.

Jednak sam diagram klas może nie wystarczać przy analizie systemu. Istnieją narzędzia specjalne zaprojektowane do przeprowadzania procesu inżynierii wstecznej. Kilka z nich scharakteryzujemy pokrótce.

- *Imagix 4D* [www1] firmy *Imagix Corporation*. Oprócz struktury klas odtwarza również strukturę plików, w szczególności plików nagłówkowych. Dodatkowo zawiera metryki całego projektu.

- *Understand* [www2] firmy *STI*. *Understand for C++* umożliwia odtworzenie struktury dziedziczenia klas, nawigację kodu źródłowego, oraz jego edycję (zawiera edytor tekstu z kolorowaniem składni języka C++). Podobnie jak *Imagix* zawiera informację o plikach nagłówkowych oraz metryki.
- *Object Marker* [www3] firmy *MarkV* jest programem funkcjonalnie podobnym do *Rational Rose* jest. Za jego pomocą można projektować aplikacje w jednej z kilku najbardziej popularnych notacji (m.in. UML). Umożliwia generowanie kodu i przeprowadzanie procesu inżynierii wstecznej, dzięki czemu odtwarza strukturę klas.
- *WinTranslator*[www4] firmy *Excel software* to narzędzie inżynierii wstecznej oferujące m.in.: odtwarzanie struktury wywołań funkcji dla języków Pascal, C, Basic i Fortran, generowanie diagramów klas dla języków C++, JAVA i ADA, generowanie modelu danych ze skryptu SQL.
- *SoDa*[www5] firmy *Rational Corporation* automatycznie tworzy dokumentację z projektów wykonanych w narzędziu *Rational Rose*. Potrafi również tworzyć dokumentację przeprowadzając proces inżynierii wstecznej kodu źródłowego języka C++.
- *SNiFF+* [www6] firmy *Wind River* to potężne narzędzie do inżynierii wstecznej dla dużych projektów (od stu tysięcy do pięciu milionów linii kodu źródłowego). Używane przez duże korporacje, m.in.: Alcatel, BMW, Boeing, Bosch, Daimler Chrysler, Ericsson, Motorola, Nokia, i Siemens.

Należy zaznaczyć, iż w pewnych zastosowaniach są potrzebne bardzo proste i efektywne narzędzia, które pozwalają na podstawie istniejącego kodu źródłowego z opisem schematu bazy danych odtworzyć diagram klas UML tegoż schematu lub jego fragmentu. Wymienione wyżej narzędzia spełniają to zadanie, lecz często są niedostępne z powodów finansowych lub technicznych. Poza tym, samo zagadnienie zbudowania systemu, który automatycznie generuje graficzną reprezentację schematu klas stanowi ciekawe wyzwanie naukowe i praktyczne. Przy tworzeniu baz danych w oparciu o model obiektowy taki system z inżynierią wsteczną może znacznie ułatwić i przyspieszyć proces projektowanie bazy, szczególnie w zespole wieloosobowym.

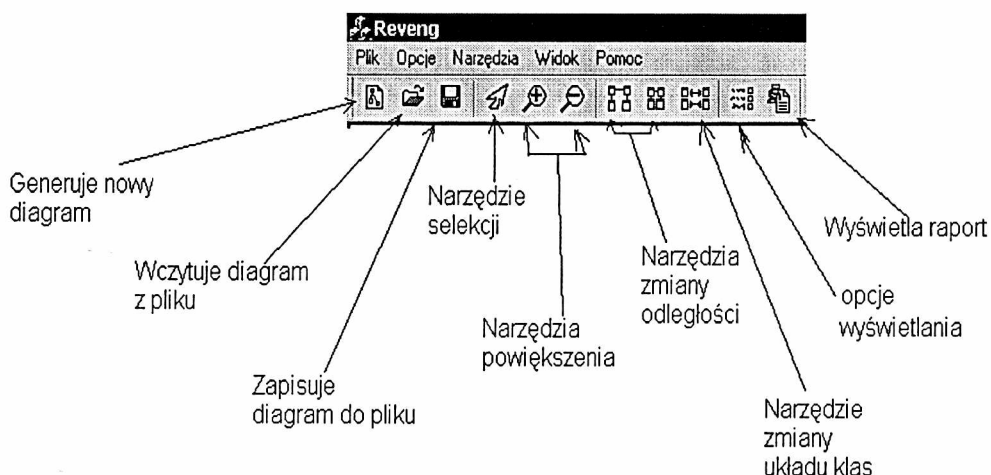
W rozdziale 5 przedstawiono własny system typu CASE realizujący wyżej opisane zadanie.

5. SYSTEM REVENG JAKO PRZYKŁAD CASE Z INŻYNIERIĄ WSTECZNĄ

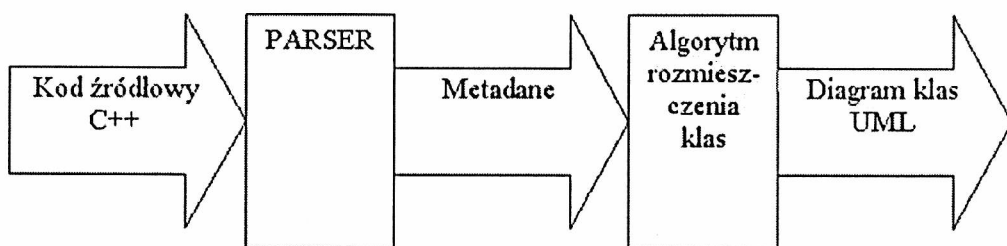
Do stworzenia aplikacji wykorzystano język C++ [GREB99] i środowisko programistyczne firmy Microsoft – Visual Studio 6.0 [HOLZ99]. Przy testowaniu

działania programu wspierano się programem *Rational Rose* 2001, w którym tworzone diagramy klas i generowano dla nich kod źródłowy w języku C++. Następnie na podstawie otrzymanego kodu generowano diagramy klas za pomocą własnej aplikacji i sprawdzano ich zgodność z oryginałem oraz poprawność notacji.

Wszystkie funkcje programu są dostępne z poziomu paska narzędzi (rysunek 2), a blokową architekturę aplikacji przedstawia rysunek 3.



Rys. 2. Pasek narzędzi i menu w systemie REVENG



Rys. 3. Schemat blokowy architektury programu

Parser

Parser jest główną częścią programu. Przeprowadza on analizę leksykalną kodu języka C++. W trakcie analizy ma za zadanie rozpoznać odpowiednie znaczniki języka i na ich podstawie utworzyć obiektową formę reprezentacji klas. Dodatkową funkcją parsera jest generowanie komunikatów o znalezionych znacznikach. Na podstawie tych komunikatów zostanie później utworzony raport o stanie sesji.

W parserze zaimplementowano również rozpoznawanie błędów składniowych języka. Jeśli parser natrafi na błąd składni, na przykład brak średnika na końcu linii deklaracji, to generuje komunikat o błędzie. Komunikaty o błędach również wchodzą w skład raportu.

Działanie parsera rozpoczyna się od usunięcia komentarzy z bufora, w którym zgromadzony jest kod źródłowy języka C++, pobrany wcześniej z plików.

Tabela 1. Usuwanie komentarzy z bufora

Bufor przed usunięciem komentarzy	Bufor po usunięciu komentarzy
<pre>class Klasa { //komentarz linowy private: int atrybut; /*Komentarz blokowy*/ void metoda; //i jeszcze jeden komentarz };</pre>	<pre>class Klasa { private: int atrybut; void metoda; };</pre>

Następnie po usunięciu komentarzy rozpoczyna się właściwe przetwarzanie kodu. W pierwszej kolejności wyszukiwana jest nazwa klasy, oraz określany jest ewentualny związek generalizacji. Po tej operacji parser przetwarza deklaracje metod i atrybutów dla odpowiedniej klasy.

Po znalezieniu klasy i ewentualnej nadklasy następuje szukanie atrybutów i metod.

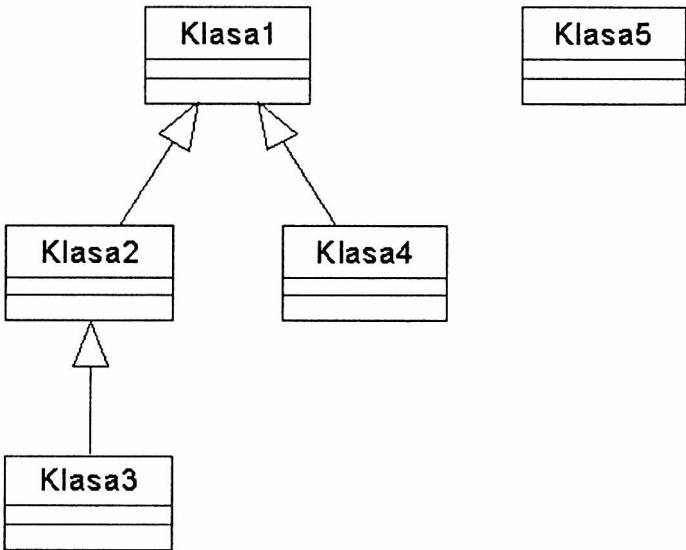
Algorytm rozmieszczenia klas

Jedną z najtrudniejszych w implementacji części omawianej aplikacji był algorytm rozmieszczenia klas. Z punktu widzenia efektywności algorytmów, problem rozmieszczenia elementów jest problemem NP trudnym. Oznacza to, że nie istnieje algorytm rozwiązujący problem w „rozsądnym” (akceptowalnym) czasie. Rozwiązaniem optymalnym byłoby takie ułożenie klas, aby klasy nie zachodziły na siebie, aby widoczne było drzewo dziedziczenia i aby linie asocjacji nie przecinały się wzajemnie i nie przebiegały przez klasy. Niestety, ze względu na złożoność problemu nawet profesjonalne narzędzia inżynierii wstecznej nie spełniają tych założeń. Wybierając sposób ułożenia klas na ekranie autorzy starali się przede wszystkim odtworzyć drzewo dziedziczenia, tzn. aby nadklasa była umieszczona wyżej od swoich podklas.

Dla struktury przedstawionej na rysunku 4., opisywany algorytm wypełni tablicę rozmieszczenia klas w sposób przedstawiony w tabeli 2.

Jak łatwo zauważyć, wiersze w tabeli zawierają duplikaty. Jest to spowodowane tym, że hierarchia dziedziczenia, była rozpatrywana dla każdej klasy znajdującej się

na liście klas. Jako, że np. klasa *Klasa1* jest nadklasą dla klasy *Klasa2*, ale i dla klasy *Klasa4* i pośrednio dla klasy *Klasa3*, klasa *Klasa1* występuje w tablicy aż cztery razy.



Rys. 4. Drzewo dziedziczenia do przykładu omawiającego algorytm rozmieszczenia klas

Tabela 2. Rozmieszczenie klas w tablicy po określeniu hierarchii dziedziczenia

	0	1	2	3	4
0	<i>Klasa1</i>	<i>Klasa1</i>	<i>Klasa1</i>	<i>Klasa5</i>	<i>Klasa1</i>
1	<i>Klasa2</i>	<i>Klasa2</i>			<i>Klasa4</i>
2		<i>Klasa3</i>			

Kolejnym krokiem algorytmu rozmieszczenia klas jest więc usunięcie duplikatów z poszczególnych wierszy. Dla każdego elementu z każdego wiersza jest sprawdzane, czy elementy w danym wierszu w pozostałych kolumnach, nie są takie same. Jeśli są, są usuwane. Po usuwaniu duplikatów z wierszy tablica rozmieszczenia klas ma postać przedstawioną w tabeli 3.

Tabela 3. Rozmieszczenie klas w tablicy po usunięciu duplikatów

	0	1
0	<i>Klasa1</i>	<i>Klasa5</i>
1	<i>Klasa2</i>	<i>Klasa4</i>
2	<i>Klasa3</i>	

Tak wypełniona tablica jest następnie rzutowana na ekran.

Obiektowa baza danych

W trakcie opisanego wcześniej przetwarzania kodu, tworzona jest obiektowa baza danych, zawierająca metadane opisujące klasy języka C++. W bazie zgromadzone są pełne informacje na temat klas, takie jak: nazwy klas, nazwy i typy atrybutów, nazwy, typy i parametry metod, widoczność atrybutów i metod. Oprócz informacji o klasach, przechowywane są informacje o zależnościach między nimi. Rozróżnia się dwa typy zależności: generalizację i asocjację. Na podstawie przetworzonego kodu nie da się natomiast odtworzyć informacji o agregacji i kompozycji, od zwykłej asocjacji.

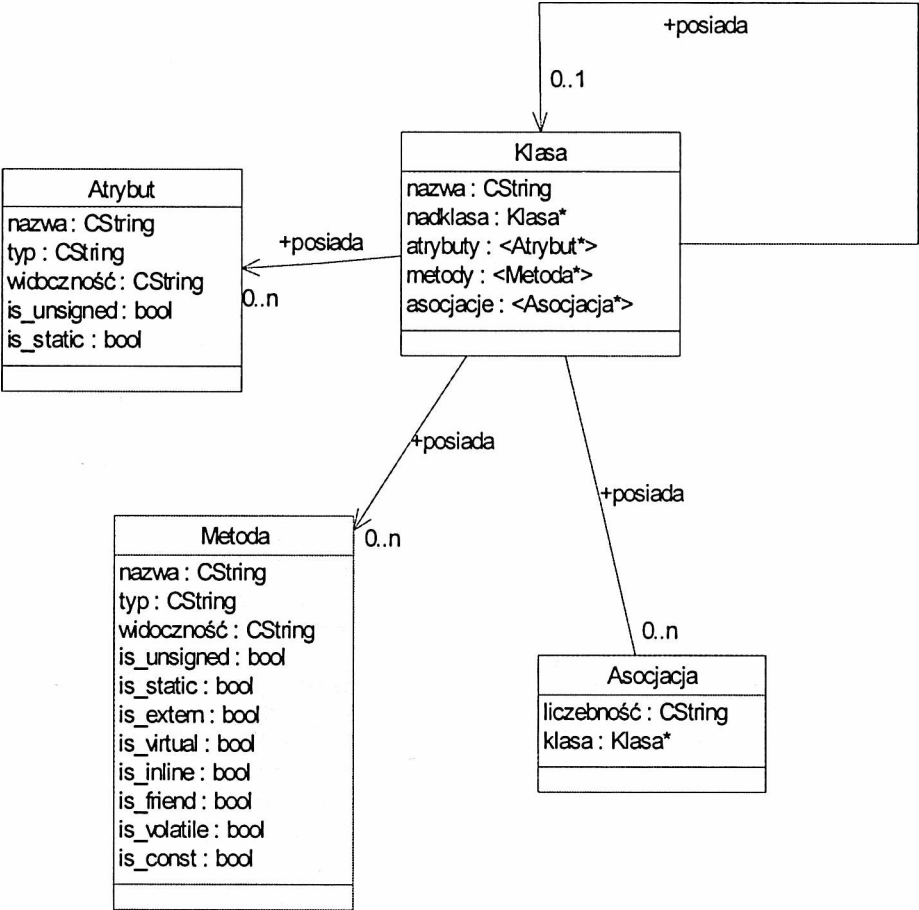
Strukturę obiektowej bazy danych przedstawia rysunek 5.

Bazę danych zaimplementowano w systemie ObjectStore z wykorzystaniem środowiska Visual C++.

Funkcje systemu

Do głównych funkcji systemu REVENG należą [BEDN04]:

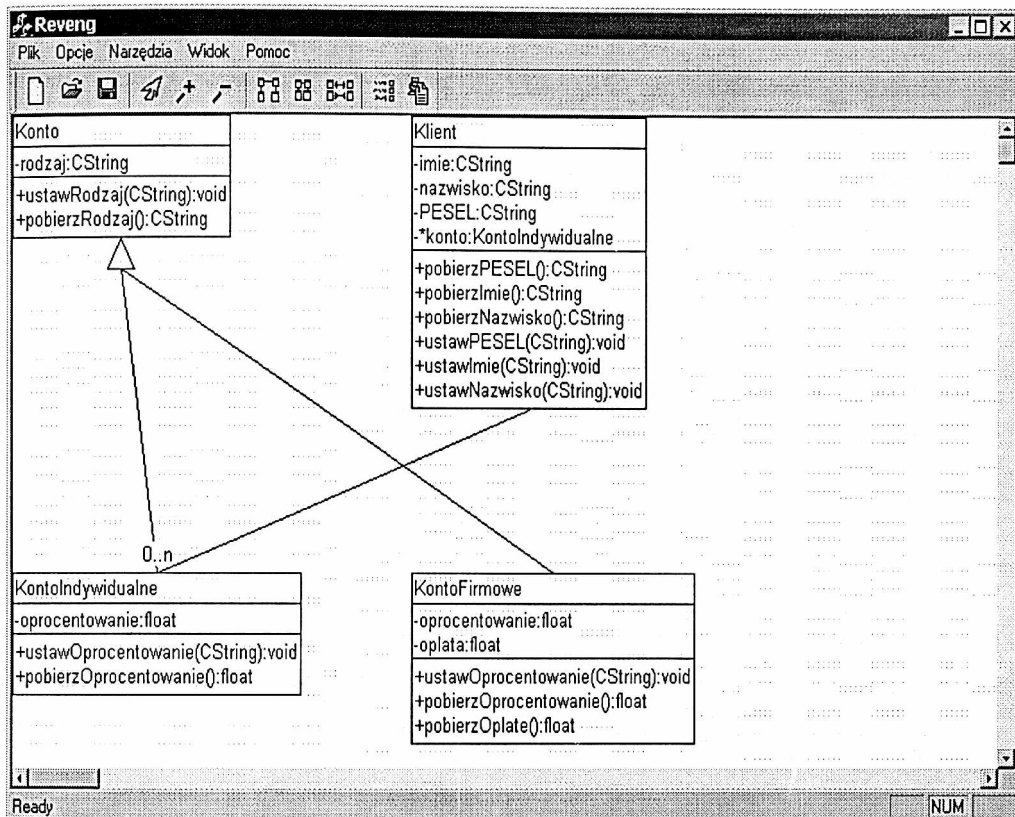
- generowanie nowego diagramu;
- wczytanie oraz zapis istniejącego diagramu;
- narzędzie selekcji;
- narzędzie powiększenia;
- narzędzie zmiany odległości;
- narzędzie zmiany układu klas;
- ustawienie opcji wyświetlania;
- generowanie raportu.



Rys. 5. Struktura obiektowej bazy danych

Tworzenie nowego diagramu

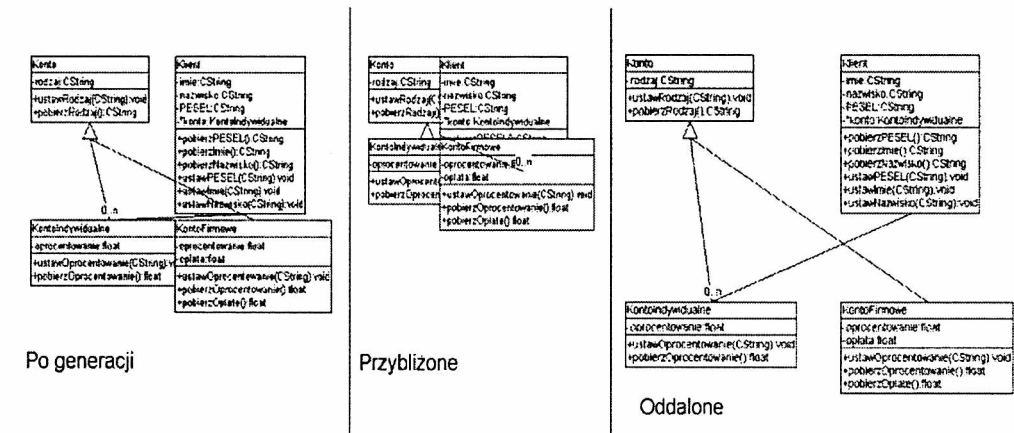
Obecnie projekty programistyczne są często zorganizowane w taki sposób, że deklaracja każdej klasy znajduje się w oddzielnym pliku nagłówkowym. Taką strukturę ma też kod wygenerowany przy użyciu narzędzi do tworzenia diagramów języka UML (np. *Rational Rose*), dlatego w aplikacji umożliwiono przetwarzanie wielu plików na raz.



Rys. 6. Przykład wygenerowanego diagramu klas

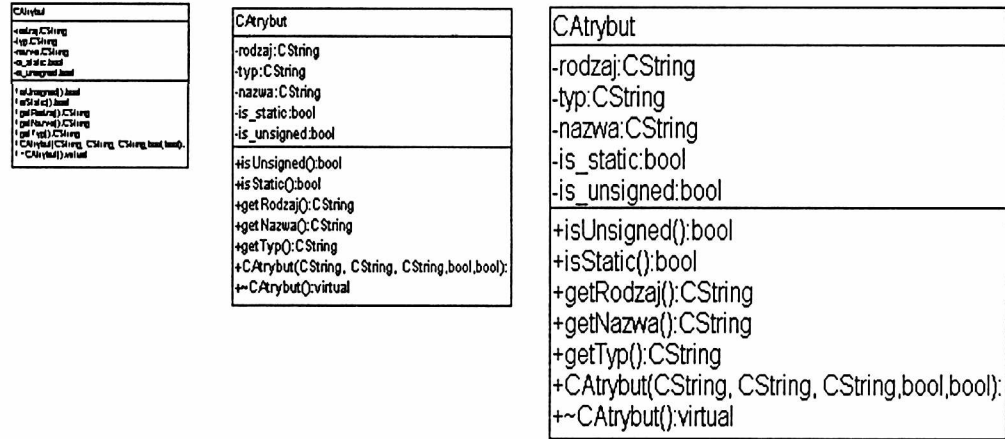
Tworzenie nowego diagramu rozpoczyna się od wskazania przez użytkownika plików źródłowych, bądź nagłówkowych, z których kod języka C++ zostanie wczytany do bufora. Program może przetwarzać zarówno pliki nagłówkowe jak i źródłowe. Kiedy użytkownik naciśnie przycisk na pasku narzędzi, lub wybierze z menu *Plik* opcję *Nowy diagram* otwierane jest okno dialogowe wyboru plików. Po zaznaczeniu plików i naciśnięciu przycisku *Otwórz* pliki zostaną przepisane do bufora i rozpoczyna się analiza kodu źródłowego, oraz rozmieszczanie klas. Po zakończeniu tych czynności, jeśli program nie znalazł błędów składniowych języka, diagram klas zostaje wyświetlony na ekranie.

Po wygenerowaniu diagramu, klasy są rozmieszczane według opisanego wcześniej algorytmu. Początkowo, każdej klasie przydzielona zostaje część ekranu o rozmiarze 200x200 pikseli. Jeśli klasy zawierają dużo atrybutów i metod lub posiadają długie nazwy, to mogą na siebie zachodzić. Aby wyeliminować ten problem, użytkownik ma możliwość zwiększenia lub zmniejszenia odstępów między klasami, przy pomocy odpowiednich przycisków na pasku narzędzi.



Rys. 7. Zmiana odstępu między klasami

Program *Reveng* daje użytkownikowi możliwość powiększania lub pomniejszania klas widocznych na diagramie. Dzięki temu, można dobrać odpowiednią wielkość klas, by uczynić diagram bardziej czytelnym lub dopasować go do wielkości ekranu. Powiększanie lub pomniejszanie jest dostępne z poziomu paska narzędzi lub menu.



Rys. 8. Powiększenie i pomniejszenie klasy *CAtrybut*

Dodatkowo system pozwala generować raporty o przebiegu procesu inżynierii wstecznej oraz konfigurować opcję wyświetlania szczegółowości diagramów klas.

6. PODSUMOWANIE

Popularność inżynierii wstecznej w tworzeniu oprogramowania zaowocowało pojawieniem się odpowiednich narzędzi umożliwiających odtwarzanie różnych reprezentacji projektu w oparciu o kod źródłowy. Zagadnienie zbudowania takiego narzędzia przedstawia sobą niemniej ciekawy i złożony problem, a niżeli same mechanizmy inżynierii wstecznej. W pracy przedstawiono propozycję prostego narzędzia typu CASE, które odtwarza w postaci graficznej obiektowy model danych w oparciu o pliki źródłowe języka C++. Diagramy klas reprezentujące schemat bazy danych wyrażone zostały w popularnej notacji UML.

W trakcie tworzenia systemu generującego diagramy klas zidentyfikowano szereg problemów, związanych z czytelnym odwzorowaniem graficznej postaci schematu. Rozwiązując zidentyfikowane problemy utworzono narzędzia wspomagające proces inżynierii wstecznej, między innymi do skalowania oraz alokacji klas będących częścią diagramu. Do przechowywania informacji na potrzeby inżynierii wstecznej zaprojektowano obiektową bazę danych. Przechowywanie danych trwałych w postaci obiektowej ułatwiło mechanizm generowania diagramów klas. System wyróżnia się dużą elastycznością w formowaniu postaci graficznej modelu conceptualnego. Użytkownik posiada szerokie możliwości dopasowywania wyglądu modelu do swoich potrzeb.

Ważną właściwością systemu jest możliwość konfiguracji opcji szczegółowości prezentacji diagramów klas, a także możliwość skalowania obiektów graficznych.

Do najciekawszych problemów, rozwiązanych w trakcie budowy systemu można zaliczyć prezentację drzewa generalizacji. Ciekawym problemem rozwojowym jest prezentacja związku dziedziczenia wielokrotnego.

LITERATURA

- [BEDN04] Marcin Bednarski, *Metody generowanie diagramów klas w notacji UML w oparciu o kod źródłowy języka C++*, Praca magisterska, Politechnika Wrocławska, 2004.
- [BOOC02] Grady Booch, James Rumbaugh, Ivar Jacobson, *UML Przewodnik Użytkownika*, WNT, 2002.
- [CHIK90] Chikofski E.J., Cross J.H., *Reverse Engineering and design recovery: A taxonomy*, IEEE Software, 7, 13–17, 1990.
- [CLAY1] R. Clayton, S. Rugaber, L. Wills, *On the Knowledge Required to Understand a Program*, [www1]
- [FOWL97] M. Fowler. *UML Distilled*. Addison Wesley, 1997.
- [FOWL99] M. Fowler, K. Scott, *UML Distilled Second Edition A Brief Guide to the Standard Object Modeling Language*, Addison Wesley, 1999.
- [GREB99] Jerzy Grębosz, *Symfonia C++*, Oficyna Kallimach Kraków, 1999.
- [HEML01] J. Hemlight *Reverse engineering in computer applications*, 2001.

- [HOLZ99] Steven Holzner *Visual C++*, Helion, 1999.
- [KLOS03] Aleksander Klosow, *Problemy pozyskiwania wiedzy w procesie inżynierii wstecznej z wykorzystaniem języka UML*, XI Konferencja Pozyskiwanie wiedzy i jej zarządzanie, Turawa, 2003.
- [NEIS02] Kevin McNeish, *UML class diagrams*, CoDe Magazine, 01/02-2002.
- [PREM94] W.J. Premerlani, M.R. Blaha, *An Approach for Reverse Engineering of Relational Database*, Communications of the ACM, vol. 37, nr. 5, may 1994
- [RUGA01] Spencer Rugaber, Terry Shikano, R.E. Kurt Stirewalt, *Adequate Reverse Engineering*, 2001.
- [RUGA1] S. Rugaber, *Program Comprehension for Reverse Engineering*, Institute of Technology, Atlanta, [www1]
- [RUGA2] S. Rugaber, R. Clayton, *The Representation Problem in Reverse Engineering*, Institute of Technology, Atlanta, [www1]
- [SCHW02] Daniel Schweizer. *Navigation in Object-Oriented Reverse Engineering*, University of Bern, Institute of Computer Science and Applied Mathematics, 2002.
- [www1] www.imagix.com - strona producenta programu *Imagix 4D*
- [www2] www.scitools.com/ucpp.html - strona firmy *STI*, producenta programu *Understand for C++*
- [www3] www.markv.com/markv.com/objectmaker.htm - strona firmy *MarkV* producenta programu *Object Marker*
- [www4] www.excelsoftware.com - strona firmy *Excel Software* producenta programu *WinTranslator*
- [www5] www.rational.com - strona producenta *Rational Rose* i *SoDa*
- [www6] www.windriver.com/products/development_tools/ide/sniff_plus/ - strona firmy *Wind River* producenta programu *SniFF+*
- [www7] <http://www.cc.gatech.edu/reverse/>

REVERSE ENGINEERING METHODS IN THE OBJECT DATABASE SYSTEMS

One of the most important problem in reverse engineering is presentation of the physical data model on the higher level of abstraction. Especially graphical representation of the conceptual object model is important. The issue can be useful in object database system during design phase of engineering process. In the paper some mechanisms of generating UML class diagrams from C++ source are presented. The CASE system REVENG with reverse engineering implemented on Wrocław University of Technology is described.

Zbigniew SZPUNAR*
Artur WRÓBLEWSKI**

RZUTOWANIE MODELU OBIEKTOWEGO DO RELACYJNEGO MODELU DANYCH

Jednym z najbardziej popularnych sposobów programowania jest obecnie programowanie obiektowe. Aplikacje obiektowe bardzo często odwołują się do danych przedstawionych w modelu relacyjnym i przechowywanych w relacyjnych bazach danych. Programista, z poziomu języka obiektowego, najczęściej musi posługiwać się w takich przypadkach pojęciami modelu relacyjnego danych (relacja, krotka, klucz główny, klucz obcy) równocześnie z pojęciami języka obiektowego (klasa, obiekt, referencja). W celu umożliwienia programiście aplikacji obiektowej łatwego operowania danymi, zawartymi w relacyjnej bazie danych, za pomocą pojęć i konstrukcji języka obiektowego zostało stworzonych wiele bibliotek dla różnych języków obiektowych, które nie tylko umożliwiają konwersję modelu obiektowego na relacyjny, ale także znaczną automatyzację procesu powstawania oprogramowania przy minimalnym, kontrolowanym wzroście wymagań sprzętowych systemu, pod kontrolą którego działa aplikacja obiektowa.

W artykule zostanie opisane jedno z rozwiązań, które umożliwia programiście operowanie na relacyjnej bazie danych za pomocą pojęć modelu obiektowego, a także zostaną przedstawione korzyści oraz wady z zastosowania takiego rozwiązania.

1. WPROWADZENIE

Wiele współczesnych aplikacji jest tworzonych w sposób obiektowy. Klasy, obiekty, dziedziczenie, hermetyzacja i polimorfizm, a także rozwój odpowiednich metod modelowania takich jak *Unified Modeling Language* (UML) oraz języków programowania, by wymienić spośród licznych C++, Java, Python, Ocaml (*Objective Caml*), czynią model obiektowy atrakcyjnym dla wielu zastosowań, m.in.: do tworzenia aplikacji biznesowych, portali internetowych, a nawet systemów operacyjnych.

Zadaniem dużej części aplikacji obiektowych (czyli aplikacji wykonanych w obiektowych środowiskach programistycznych) jest dostarczanie odpowiednich informacji użytkownikowi. Autorzy oprogramowania, w celu zapewnienia

* Politechnika Wroclawska, Wydział Informatyki i Zarządzania, Instytut Informatyki Stosowanej,
50-370 Wroclaw, Wybrzeże Wyspiańskiego 27, Zbigniew.Szpunar@pwr.wroc.pl

** Student Wydziału Informatyki i Zarządzania Politechniki Wrocławskiej

odpowiedniej jego funkcjonalności oraz wydajności, często decydują się na wykorzystanie systemów zarządzania relacyjną bazą danych (ang. RDBMS). Model relacyjny, stosowany w tych systemach, jest obecnie najlepiej rozwiniętą i najszerzej implementowaną teorią w dziedzinie baz danych [UW01, Dat00].

Dostęp do różnych systemów baz danych może być realizowany za pomocą odpowiedniego interfejsu, np.: *Open Database Connectivity* (ODBC), *Java Database Connectivity* (JDBC), *Python Database API* (DB-API) [SIG] czy *Perl Database Interface* (DBI), co umożliwia, przy odpowiedniej kompatybilności używanych zapytań SQL, stosowanie różnych RDBMS przez jedną aplikację.

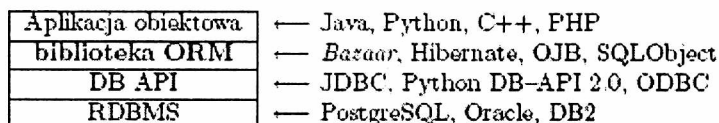
Podsumowując, programiści często tworzą aplikacje obiektowe operujące na informacjach znajdujących się w relacyjnej bazie danych, do której dostęp jest możliwy za pomocą odpowiedniego API. Tego typu aplikacje muszą dokonywać konwersji danych relacyjnych na dane obiektowe (i na odwrót).

W celu zautomatyzowania tego procesu powstało wiele bibliotek, które dostarczają autorom aplikacji szereg mechanizmów umożliwiających ich tworzenie za pomocą pojęć modelu obiektowego jednocześnie korzystając z relacyjnej bazy danych (tab. 1).

Tabela 1. Zestawienie pojęć modelu obiektowego i modelu relacyjnego

Model obiektowy	Model relacyjny
klasa	relacja
obiekt	krotka
referencja	klucz, klucz obcy
atrybut klasy	atrybut relacji

W literaturze anglojęzycznej tego typu biblioteki są klasyfikowane jako *Object-Relational Mappers* (ORM). Umiejscowienie biblioteki ORM pośredniczącej pomiędzy aplikacją i systemem bazy danych obrazuje rysunek 1.



Rys. 1. Biblioteka ORM pomiędzy aplikacją i systemem bazy danych

Potencjalnie biblioteki ORM pozwalają na automatyzację procesu tworzenia aplikacji obiektowej. Po zdefiniowaniu klas programista nie powinien tracić czasu na:

- implementację metod do pobierania i zmiany wartości atrybutów (tj. metod *set/get*, np.: *getArticle*, *setArticle*, *getName*, *setName*),
- tworzenie procedur dostępu do danych oraz ich aktualizacji w bazie danych,
- zarządzanie obiektami w pamięci operacyjnej,
- buforowanie danych.

2. PRZEGLĄD ROZWIĄZAŃ

W tabeli 2. została przedstawiona lista bibliotek ORM, które umożliwiają operowanie na relacyjnych bazach danych w sposób obiektowy.

Tabela 2. Lista bibliotek ORM

Nazwa	Język	Licencja	Adres
Hibernate	Java	LGPL	http://www.hibernate.org/
OJB	Java	ASF	http://db.apache.org/ojb/
Modeling Framework	Python	GPL	http://modeling.sourceforge.net/
SQLObject	Python	LGPL	http://sqlobject.org/
Bazaar	Python	LGPL	http://www.nongnu.org/bazaar/

Kod źródłowy przedstawionych bibliotek jest udostępniany na licencjach zgodnych z wytycznymi *Open Source Initiative*

<http://www.opensource.org/licenses/index.html>.

Rozwiązania komercyjne zostały pominięte w tym zestawieniu. Firma Sun opracowała standard *Java Data Object* (JDO), który definiuje API implementacji dostępu do danych w sposób obiektowy dla języka Java

<http://java.sun.com/products/jdo/>.

Specyfikacje JDO można przestudiować pod adresem

<http://jcp.org/aboutJava/communityprocess/final/jsr012/index.html>.

Standard jest implementowany przez niektóre firmy komercyjne. Informacje na temat komercyjnych i darmowych implementacji JDO, artykuły, aktualności oraz porady są dostępne na stronie

<http://jdocentral.com/>.

Biblioteki napisane w języku Python zostały wybrane na podstawie informacji znajdujących się na stronie PythonInfo Wiki

<http://www.python.org/cgi-bin/moinmoin/HigherLevelDatabaseProgramming>.

W zestawieniu znalazły się także dwa projekty napisane dla języka Java. Projekt Hibernate od roku 2003 jest wspierany przez JBoss Group

<http://www.jboss.org/>,

natomiast OJB jest projektem powstającym w ramach *Apache Software Foundation*

<http://www.apache.org/>.

Obydwie organizacje są obecnie jednymi z najbardziej twórczych organizacji w świecie *Open Source* dostarczając wielu zaawansowanych technologii.

Tabela 3 zawiera uogólnione zestawienie możliwości opisanych poniżej projektów (T/N oznacza Tak/Nie).

Tabela 3. Zestawienie możliwości różnych projektów konwertujących model obiektowy do modelu relacyjnego danych

	Bazaar	Hibernate	OJB	MF	SQLObject
Klucze wielokolumnowe	N	T	N	T	T
Asocjacje 1-1	T	T	T	T	T
Asocjacje 1-n	T	T	T	T	T
Asocjacje m-n	T	T	T	T	T
Asocjacje dwukierunkowe	T	T	T	N	N
Dziedziczenie	T	T	T	T	N
Konfigurowalność	T	T	T	T	N
Buforowanie danych	T	T	T	N	N
Kaskadowa modyfikacja	N	T	N	T	N
Wyszukiwanie obiektów	T	T	T	T	T
Przetwarzanie wsadowe	N	T	N	N	T
Wsparcie dla wątków	N	T	N	N	N

2.1. HIBERNATE

Projekt Hibernate jest wspierany przez JBoss Group, a także jest finalistą edycji konkursu JavaWorld Editors' Choice Awards z 2003 r. Oprócz własnego API dostępu do relacyjnej bazy danych, implementuje ODMG API

<http://www.odmg.org/>

oraz standard JDO. Biblioteka Hibernate może być używana z serwerami aplikacji implementującymi standard Java 2 Enterprise Edition (J2EE).

Mechanizmy biblioteki umożliwiają konwersję klas aplikacji na relacje bazy danych na kilka sposobów, m. in.:

- za pomocą pliku XML,
- z dokumentacji w formacie JavaDoc, wykorzystując mechanizmy projektu Xdoclet (<http://xdoclet.sourceforge.net>),
- poprzez generowanie klas aplikacji na podstawie schematu bazy danych.

Klucz główny rzutowanej relacji powinien być liczbą całkowitą, jednakże można także definiować klucze złożone. Projekt wspiera dziedziczenie oraz wszystkie rodzaje asocjacji, tj. jeden do jeden, jeden do wielu oraz wiele do wielu, także związki dwukierunkowe. Biblioteka wspiera także kaskadową aktualizację relacyjnych danych asocjacji.

Biblioteka Hibernate umożliwia wydajne buforowanie obiektów aplikacji. Hibernate Dual-Layer Cache Architecture (HDLCA) zapewnia dostęp do obiektów aplikacjom wielowątkowym i wspiera różne strategie buforowania danych.

Autorzy projektu opracowali elastyczny, obiektowy mechanizm wyszukiwania obiektów aplikacji. Jest możliwe także wyszukiwanie za pomocą zapytań SQL.

Pełna lista możliwości tej biblioteki znajduje się na stronie

<http://www.hibernate.org/4.html>.

2.2. OJB – OBJECT RELATIONAL BRIDGE

Projekt OJB – Object Relational Bridge jest projektem rozwijanym w ramach organizacji Apache Software Foundation. Podobnie jak Hibernate, projekt OJB dostarcza własne metody dostępu do danych, a także implementuje standardy JDO oraz ODMG API. Biblioteka może być używana razem z serwerem aplikacji J2EE.

OJB umożliwia rzutowanie modelu obiektowego na podstawie modelu relacyjnego za pomocą pliku konfiguracyjnego w formacie XML. Klucz główny powinien być liczbą całkowitą. Rzutowanie wspiera dziedziczenie oraz umożliwia definiowanie wszystkich rodzajów związków pomiędzy klasami aplikacji.

Projekt umożliwia różnego rodzaju strategie buforowania obiektów aplikacji, które są zintegrowane z systemem zarządzania pamięci języka Java (*garbage collector*). Mechanizm synchronizacji udostępnianych danych umożliwia uruchamianie kilku instancji OJB korzystających z jednej bazy danych.

Z możliwościami tej biblioteki można zapoznać się pod adresem
<http://db.apache.org/ojb/features.html>.

2.3. MODELING FRAMEWORK (MF)

Projekt Modeling Framework jest inspirowany platformą Enterprise Object Framework stworzoną przez firmę NeXT. Biblioteka może być używana z serwerem aplikacji Zope, który jest jednym z najbardziej zaawansowanych serwerów aplikacji napisanym w języku Python (<http://www.zope.org/>).

Modeling Framework umożliwia rzutowanie modelu obiektowego za pomocą języka Python. Oprócz tego klasy mogą być zdefiniowane za pomocą pliku XML. Rzutowanie umożliwia definiowanie dowolnych kluczy głównych relacji, jednokierunkowych asocjacji 1–1, 1–n i m–n oraz wspiera dziedziczenie.

Biblioteka nie umożliwia buforowania obiektów.

Dokumentacja użytkownika projektu

<http://modeling.sourceforge.net/UserGuide/UserGuide.html>

zawiera przegląd możliwości biblioteki.

2.4. SQLAlchemy

SQLAlchemy jest dość młodym projektem, który umożliwia rzutowanie modelu obiektowego do modelu relacyjnego danych. Rzutowanie modelu obiektowego do modelu relacyjnego danych odbywa się za pomocą metaklas. Rzutowanie pozwala na definiowanie kluczy złożonych, jednokierunkowych asocjacji jeden do jeden, jeden do wielu oraz wiele do wielu.

Projekt nie umożliwia zaawansowanego buforowania danych.

Autorzy projektu stworzyli abstrakcyjny mechanizm służący do wyszukiwania obiektów w bazie danych, z którym można się zaznajomić pod adresem
<http://sqlobject.org/docs/SQLBuilder.html>.

Możliwości projektu prezentuje dokumentacja użytkownika dostępna na stronie
<http://sqlobject.org/docs/SQLObject.html>.

2.5. BAZAAR

Biblioteka *Bazaar* powstała w ramach pracy inżynierskiej „Rzutowanie modelu obiektowego do modelu relacyjnego danych”, wykonanej przez Artura Wróblewskiego w 2004 r. w Wydziałowym Zakładzie Informatyki na Wydziale Informatyki i Zarządzania Politechniki Wrocławskiej.

Definiowanie modelu obiektowego, który jest rzutowany na model relacyjny, wykorzystuje mechanizmy języka Python i umożliwia konwersje asocjacji 1–1, 1–n oraz m–n (jedno– i dwukierunkowych), a także dziedziczenia.

Oprócz standardowych operacji pobierania i modyfikowania danych, biblioteka *Bazaar* umożliwia wyszukiwanie obiektów w bazie danych w sposób obiektowy oraz za pomocą zapytań SQL (zintegrowane z obiektowymi mechanizmami biblioteki), a także pozwala na określenie różnych strategii buforowania danych zintegrowanego z mechanizmami zarządzania pamięcią języka Python (*garbage collector*).

Z możliwościami biblioteki można zapoznać się pod adresem
<http://www.nongnu.org/bazaar/>.

Omówienie bibliotek ORM w dalszej części opracowania zostało dokonane w odniesieniu do biblioteki Bazaar.

3. ZASTOSOWANIE BIBLIOTEKI ORM

Programista korzystający z biblioteki ORM musi określić, które klasy aplikacji, ich atrybuty oraz asocjacje pomiędzy nimi będą przechowywane w relacyjnej bazie danych. Proces ten nazywamy konwersją lub rzutowaniem modelu obiektowego do modelu relacyjnego. Konwersja odbywa się zgodnie z zasadami opisanymi w pracy [Amba].

Biblioteki ORM umożliwiają rzutowanie modelu obiektowego do relacyjnego na różne sposoby, np. za pomocą plików XML lub za pomocą odpowiednich oznaczeń w dokumentacji programistycznej aplikacji.

W przypadku biblioteki *Bazaar* programista definiuje model obiektowy wykorzystując cechy języka Python takie jak metaklasy i deskryptory [Het].

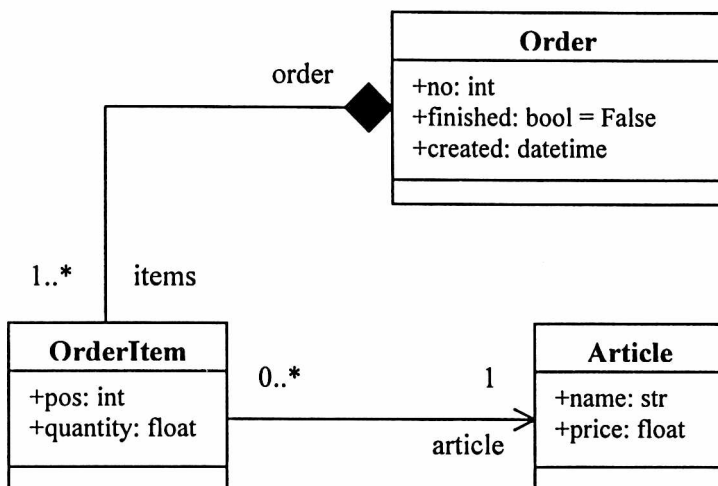


Diagram 1. Diagram przykładowej aplikacji System zamówień

```

1 # zdefiniowanie klasy aplikacji (sposób standardowy)
2 class Article:
3     __metaclass__ = bazaar.conf.Persistence
4     relation      = 'article'
5
6     # ...
7     # definicje metod
8     # ...
9
10 # dodanie atrybutów klasy
11 Article.addColumn('name')
12 Article.addColumn('price')
13
14 # zdefiniowanie klasy poprzez utworzenie instancji metaklas
15 Order = bazaar.conf.Persistence('Order', relation = 'order')
16 OrderItem = bazaar.conf.Persistence('OrderItem', \
17     relation = 'order_item')
18
19 OrderItem.addColumn('pos')
20 OrderItem.addColumn('quantity')
21
22 # zdefiniowanie jednokierunkowej asocjacji 1-n
23 # pomiędzy klasami pozycji zamówienia i artykułem
24 OrderItem.addColumn('article', 'article_fkey'.Article)
25
26 Order.addColumn('no')
27 Order.addColumn('finished')
28 Order.addColumn('created')
29
30 # zdefiniowanie dwukierunkowej asocjacji 1-n
31 # pomiędzy klasami zamówienia i pozycji zamówienia
32 OrderItem.addColumn('order', 'order_fkey', Order, vattr = 'items')
33 Order.addColumn('items', vcls = OrderItem, \
34     vcol = 'order_fkey', vattr = 'order')
  
```

Listing 1. Definicja klas przykładowej aplikacji System zamówień

Przykład definiowania klas aplikacji, przedstawionych na diagramie 1 (prosty system kontroli realizacji zamówień, którego zadaniem jest informowanie o zrealizowanych oraz aktualnie realizowanych zamówieniach na produkowane artykuły), znajduje się na listingu 1. Schemat relacyjnej bazy danych, utworzony dla klas aplikacji, został przedstawiony na listingu 2.

Po zdefiniowaniu klas aplikacji oraz określeniu, w jaki sposób dane obiektów tych klas mają być zapisywane i odczytywane z bazy danych, programista może przystąpić do implementacji logiki aplikacji.

```

1 create sequence order_seq;
2 create table "order" (
3     __key__ integer,
4     no integer not null unique,
5     finished boolean not null,
6     created timestamp not null default current_timestamp,
7     primary key (__key__)
8 );
9
10 create sequence article_seq;
11 create table article (
12     __key__ integer,
13     name varchar(20) not null,
14     price numeric(10,2) not null,
15     unique (name),
16     primary key (__key__)
17 );
18
19 create sequence order_item_seq;
20 create table order_item (
21     __key__ integer,
22     order_fkey integer,
23     pos integer not null,
24     article_fkey integer not null,
25     quantity numeric(10,3) not null,
26     primary key (__key__),
27     unique (order_fkey, pos),
28     foreign key (order_fkey) references "order"(__key__),
29     foreign key (article_fkey) references article(__key__)
30 );

```

Listing 2. Schemat bazy danych przykładowej aplikacji **System zamówień**

Pobieranie i wyszukiwanie danych obiektów w bazie danych zostało przedstawione na listingu 3, natomiast na listingu 4 zostały zaprezentowane metody tworzenia oraz modyfikacji obiektów w bazie danych.

Programista, korzystający z biblioteki *Bazaar*, definiuje klasy aplikacji i operuje na danych obiektów aplikacji w bazie danych w pełni wykorzystując obiektywne możliwości języka programowania oraz uwzględniając charakterystykę języka Python, w szczególności:

- tworzy klasy z wykorzystaniem metaklas,
 - w sposób standardowy, tj. poprzez definicje klasy (Article),
 - poprzez utworzenie instancji metaklas (OrderItem, Order),

- realizuje w sposób dynamiczny, za pomocą deskryptorów, dostęp do asocjowanych obiektów,
- uwzględnia cechę dynamicznej typizacji języka (programista nie podaje typów atrybutów klas aplikacji),
- ma możliwość korzystania z różnorodnych, dostępnych typów, jakie oferuje język, np. metoda wyszukiwania obiektów zwraca iterator [YvR], dzięki czemu programista może umieścić znalezione obiekty w strukturze danych (np. lista, zbiór lub sekwencja), która jest najbardziej odpowiednia dla danej części aplikacji.

```

1  # utworzenie warstwy Bazaar
2  bzt = bazaar.core.Bazaar((Article, Order, OrderItem))
3
4  # ustanowienie połączenia z bazą danych
5  bzt.connectDB('dbname = orders port = 5433 user = john')
6
7  # wyświetlenie numerów wszystkich zamówień
8  for ord in bzt.getObjects(Order):
9      print order.no
10
11 # znalezienie zamówienia nr 10
12 orders = list(bzt.find(Order, {'no': 10}))
13 ord = orders[0]
14
15 # wyświetlenie nazw artykułów pozycji zamówienia nr 10
16 for oi in ord.items:
17     print oi.article.name

```

Listing 3. Pobieranie obiektów przykładowej aplikacji System zamówień

```

1  # utworzenie obiektu
2  apple = Article()
3  apple.name = 'apple'
4  apple.price = 2.33
5
6  # utworzenie i jednocześnie nadanie wartości
7  # niektórym atrybutom obiektu aplikacji
8  oi = OrderItem(quantity = 10, article = apple)
9  oi.pos = 1
10
11 ord = Order(no = 999, finished = False)
12
13 # dodanie obiektu do asocjacji
14 ord.items.append(oi)
15
16 # dodanie obiektów do bazy danych
17 bzt.add(apple)
18 bzt.add(oi)
19 bzt.add(ord)
20
21 # aktualizacja asocjacji
22 ord.items.update()
23
24 # zatwierdzenie transakcji
25 bzt.commit()

```

Listing 4. Modyfikacja obiektów przykładowej aplikacji System zamówień

Podczas tworzenia aplikacji programista posługuje się pojęciami modelu obiektowego, dzięki czemu kod źródłowy aplikacji jest bardziej czytelny, niż w przypadku posługiwania się dwiema kategoriami pojęć (modelu obiektowego i modelu relacyjnego), przez co jest łatwiejszy do zrozumienia oraz utrzymania.

Umożliwienie obiektowego operowania na relacyjnej bazie danych nie jest realizowane w pełni. Jednym z celów bibliotek ORM jest ułatwienie i uproszczenie korzystania z relacyjnej bazy danych, jednakże całkowite wyeliminowanie ze świadomości programisty modelu relacyjnego, z którego wad i zalet należy sobie zdawać sprawę na etapie projektowania aplikacji obiektowej oraz definiowania klas [Ambb], nie jest podejmowane przez autorów tych bibliotek, o czym świadczy dokumentacja poszczególnych projektów.

Biblioteki ORM umożliwiają programowanie obiektowe w kontekście modelu relacyjnego, przy akceptacji faktu istnienia pewnych elementów systemów relacyjnych danych. Dotyczy to w szczególności możliwości wyszukiwania danych w bazie danych. Język SQL oraz jego implementacje w poszczególnych relacyjnych systemach baz danych pozwalają na wykonywanie bardzo skomplikowanych zapytań. Co prawda poszczególne biblioteki ORM oferują różne obiektowo zorientowane sposoby wyszukiwania danych, ale nie są one obecnie w stanie wyeliminować w pełni języka SQL z kodu aplikacji obiektowej, z czego doskonale zdają sobie sprawę autorzy bibliotek ORM integrując możliwość wykonywania zapytań SQL z mechanizmami tych bibliotek.

Jeśli chcemy podczas programowania obiektowego zapomnieć o systemach relacyjnych baz danych, to powinniśmy z nich zrezygnować i skorzystać z innego, np.: obiektowego systemu baz danych.

4. WYDAJNOŚĆ

Wprowadzenie dodatkowej warstwy, w postaci biblioteki ORM, pomiędzy aplikacją obiektową a systemem baz danych, ma wpływ na wydajność aplikacji obiektowej. W tym podrozdziale opisano testy, których celem było zbadanie zużycia zasobów komputera, na którym została uruchomiona aplikacja korzystająca z biblioteki *Bazaar*.

Należy zauważyć, że wiele bibliotek ORM umożliwia buforowanie danych obiektów pobranych z bazy danych. Poniższy test nie bada zysku wydajności wynikającego z buforowania obiektów przez bibliotekę *Bazaar*.

Testy wydajnościowe zostały podzielone na dwie części:

1. Szybkość wykonania podstawowych operacji, tj. pobierania, dodawania, aktualizacji i usuwania obiektów z bazy danych.
2. Skalowalność mechanizmów warstwy *Bazaar*.

Testy zostały przeprowadzone w systemie operacyjnym PLD Linux. Zostało użyte następujące oprogramowanie:

- Python 2.3.2,
- PostgreSQL 7.4 – system baz danych,
- psycopg 1.1.10 – moduł zgodny ze specyfikacją DB-API 2.0,
- glibc 2.3.2,
- ext2 – system plików,
- kernel 2.4.22.

Testy były przeprowadzone na sprzęcie o następujących parametrach:

- procesor AMD Athlon 800MHz,
- pamięć 256MB RAM,
- dysk twardy IBM DDYS-T09170N.

4.1. WYDAJNOŚĆ PODSTAWOWYCH OPERACJI

Wydajność podstawowych operacji warstwy *Bazaar* została porównana z czasem wykonywania standardowego pobierania, dodawania, aktualizacji oraz usuwania obiektów.

Jako standardowe wykonanie poszczególnych operacji zostało przyjęte:

- dodawanie obiektów, obejmujące:
 - utworzenie pustej listy obiektów,
 - utworzenie obiektu,
 - dodanie krotki do bazy danych na podstawie danych pobranych z utworzonego obiektu,
 - zapamiętanie obiektu w liście obiektów,
- pobieranie obiektów, obejmujące:
 - utworzenie pustej listy obiektów,
 - pobranie danych z bazy danych,
 - dla każdej z krotek utworzenie obiektu i umieszczenie go w liście obiektów,
- aktualizacja obiektów, obejmujące aktualizację w bazie danych każdego obiektu znajdującego się w liście obiektów,
- usuwanie obiektów, obejmujące:
 - pobranie obiektu z listy obiektów,
 - usunięcie krotki z bazy danych na podstawie pobranego obiektu,
 - usunięcie listy obiektów z pamięci.

Na potrzeby standardowego wykonywania operacji została utworzona klasa *OrderItem* o takich samych atrybutach jak klasa *OrderItem* przykładowej aplikacji. Dane były zapisywane w relacji *order item* zidentyfikowanej na potrzeby aplikacji testowej.

Jeden test obejmował wykonanie określonej metody na liczbie stu tysięcy obiektów. Każdy test został powtórzony dziesięciokrotnie a wyniki zostały uśrednione. Wyniki testów przedstawiono w tabeli 4.

Tabela 4. Zmniejszenie wydajności warstwy *Bazaar*

Metoda	Standard	Bazaar	Wydajność
Dodanie	201.96s	260.64s	1.29
Pobranie	32.01s	33.96s	1.06
Aktualizacja	159.48s	204.32s	1.28
Usuwanie	94.68s	107.55s	1.14

4.2. SKALOWALNOŚĆ

Skalowalność aplikacji obiektowej jest czynnikiem informującym o tym, czego możemy się spodziewać, gdy ilość danych w bazie danych wzrasta. Mechanizmy warstwy *Bazaar* mają wpływ na skalowalność aplikacji obiektowej. Szybkość, z jaką określona ilość danych jest pobierana i modyfikowana w bazie danych, jest jednym z elementów, które determinują wydajność aplikacji.

Test opisany w tym podrozdziale ma na celu zbadanie skalowalności mechanizmów warstwy *Bazaar*. Test skalowalności był wykonywany na obiektach pozycji zamówienia klasy *OrderItem* aplikacji testowej.

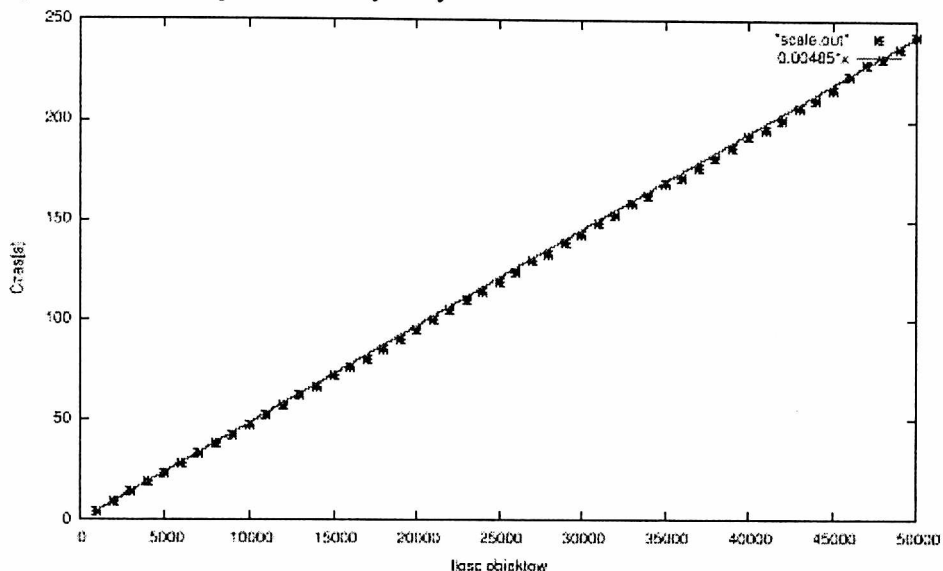
Scenariusz procedury testowej (listing 5), której zadaniem było objęcie wszystkich elementów biblioteki *Bazaar*, był następujący:

- utworzenie określonej liczby obiektów pozycji zamówienia klasy *OrderItem*,
- dodanie utworzonych obiektów pozycji zamówienia do bazy danych,
- przydzielenie obiektów pozycji zamówienia do zamówienia *ord* (operacja *ord.items.append*),
- aktualizacja asocjacji pomiędzy zamówieniem *ord*, a utworzonymi pozycjami zamówienia,
- natychmiastowe przeładowanie obiektów pozycji zamówienia,
- natychmiastowe przeładowanie relacyjnych danych asocjacji pomiędzy zamówieniem i pozycjami zamówienia.

```
1 # ord - obiekt zamówienia
2 # art - obiekt artykułu
3 items = ord.items
4 for i in range(amount):
5     oi = OrderItem()
6
7     oi.article = art
8     oi.quantity = 10
9     oi.pos = i
10
11     items.append(oi)
12     bsr.add(oi)
13
14 items.update()
15 bsr.reloadObjects(OrderItem, True)
16 Order.items.reloadData(True)
```

Listing 5. Procedura testująca skalowalność warstwy *Bazaar*

Wynik testu został przedstawiony na rysunku 2.



Rys. 2. Skalowalność warstwy *Bazaar*

Wykres przedstawia czas, w którym została wykonana, zdefiniowana wyżej, procedura testowa dla określonej liczby obiektów pozycji zamówienia.

Na podstawie wykresu możemy stwierdzić, że warstwa *Bazaar* jest skalowalna liniowo, tj. czas wykonania procedury testowej rośnie proporcjonalnie do liczby przetwarzanych obiektów. Jest to bardzo pożądana cecha, która oznacza, że przy zwiększeniu wydajności sprzętu wydajność aplikacji wzrośnie także proporcjonalnie.

5. WYKORZYSTANIE PAMIĘCI OPERACYJNEJ

W przypadku aplikacji działających na dużej liczbie danych kolejnym ważnym czynnikiem, oprócz szybkości działania, jest zużycie pamięci operacyjnej.

Możemy wyróżnić dwie kategorie elementów aplikacji obiektowej i warstwy *Bazaar*, które mają wpływ na ilość zajmowanej pamięci operacyjnej. Po pierwsze, jest zajmowany pewien stały rozmiar pamięci zależny od liczby:

- obiektów klas warstwy *Bazaar*,
- klas aplikacji,
- asocjacji między klasami aplikacji,

Po drugie, rozmiar aktualnie zajmowanej pamięci zmienia się wraz z działaniem aplikacji obiektowej i jest zależny od liczby:

- obiektów aplikacji,
- asocjacji pomiędzy obiektami aplikacji.

Procedura testująca (listing 6) zajmuje się badaniem drugiego przypadku i ma następujący scenariusz:

- utworzenie i dodanie do bazy danych obiektu zamówienia ord,
- utworzenie obiektów pozycji zamówienia klasy OrderItem,
- dodanie utworzonych obiektów pozycji zamówienia do bazy danych,
- przydzielenie obiektów pozycji zamówienia do zamówienia ord (operacja ord.items.append),
- powtórzenie opisanych wyżej kroków.

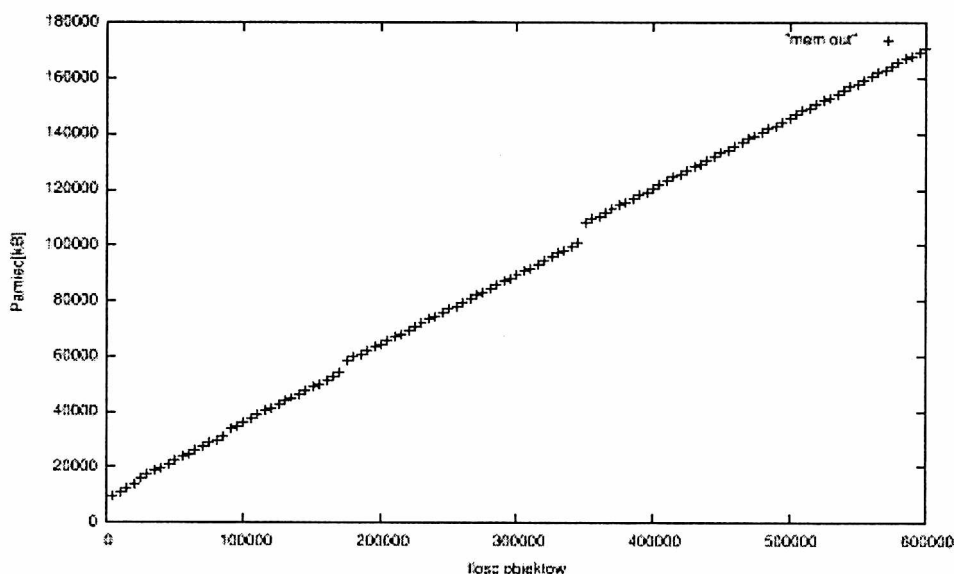
```

1  for j in range(0, amount, 5000):
2      ord = Order({'no': j, 'finished': False})
3      bzs.add(ord)
4      items = ord.items
5      for i in range(j, j + 5000):
6          oi = OrderItem()
7          oi.article = art
8          oi.quantity = 10
9          oi.pos = i
10         items.append(oi)
11         bzs.add(oi)

```

Listing 6. Procedura testująca wykorzystanie pamięci przez warstwę *Bazaar*

Na rysunku 3. przedstawiono wykres zużywanej pamięci w zależności od liczby przetwarzanych obiektów.



Rys. 3. Wykorzystanie pamięci operacyjnej przez warstwę *Bazaar*

Ilość danych, która jest generowana przez procedurę testową (aplikację obiektową) rośnie liniowo, a więc ilość zużywanej pamięci przez mechanizmy warstwy *Bazaar* także rośnie liniowo. Na tej podstawie możemy wywnioskować, że narzut wykorzystywanej pamięci dla pojedynczego obiektu jest stały w porównaniu do ilości pamięci zajmowanej przez krotkę w relacji bazy danych.

6. PODSUMOWANIE

Jedną z obietnic informatyki jest ułatwienie i automatyzacja wykonywanych codziennie czynności. Przewrotnie, dotyczy to także informatyków, których praca również może być prostsza i wymagająca mniej wysiłku, dzięki czemu wymieniona na początku tego akapitu obietnica może zostać szybciej i lepiej spełniona.

Jedną z czynności, którą można zautomatyzować, jest rzutowanie modelu obiektowego na relacyjny model danych. Przyjmując odpowiednie cele i założenia jest możliwe stworzenie dla programistów narzędzia:

- prostego w obsłudze,
- skracającego czas tworzenia aplikacji,
- skalowalnego.

Zastosowanie biblioteki ORM ma oczywiście wpływ na zwiększenie wykorzystania zasobów systemu, pod kontrolą którego została uruchomiona aplikacja obiektowa. Spadek wydajności operacji na bazie danych może zostać skompensowany buforowaniem danych, a większe zużycie pamięci – automatycznym zarządzaniem buforowanymi obiektami.

Twórcy aplikacji muszą sami zdecydować, czy wydajność oraz wykorzystanie pamięci są na akceptowalnym poziomie oraz, czy korzyści z zastosowania biblioteki ORM przewyższają ewentualne koszty związane z zużyciem zasobów sprzętowych.

LITERATURA

- [Amba] Ambler Scott W., *Mapping Objects to Relational Databases*.
<http://www.agiledata.org/essays/mappingObjects.html>
- [Ambb] Ambler Scott W., *The Object–Relational Impedance Mismatch*.
<http://www.agiledata.org/essays/impedanceMismatch.html>
- [Dat00] Date C.J., *Wprowadzenie do systemów baz danych*. WNT, Warszawa, 2000.
- [Het] Hettinger R., *How–To Guide for Descriptors*.
<http://users.rcn.com/python/download/Descriptor.htm>
- [SIG] Python Database SIG. *PEP 0249, Python Database API Specification v2.0*. <http://www.python.org/peps/pep-0249.html>
- [UW01] Ullman J.D., Widom J., *Podstawowy wykład z systemów baz danych*. WNT, Warszawa, 2001.

[YvR]

Yee Ka-Ping, van Rossum Guido, *PEP 0234, Iterators*.
<http://www.python.org/peps/pep-0234.html>

OBJECT-TO-RELATIONAL MODEL MAPPING

The article describes the features of a library developed in the Python object-oriented programming language at the Information Technology Department of the Wroclaw Technical University. The library allows developers of object-oriented applications to manipulate data stored in a relational database using object-oriented mechanisms. Additionally, the article contains results of performance tests and memory load tests, as well as a discussion on the advantages and disadvantages of the proposed solution.

Jacek GRUBER*
Hanna MAZUR*
Zygmunt MAZUR*
Marek STANDZON

MODELOWANIE I GENEROWANIE BAZODANOWYCH APLIKACJI WWW W NARZĘDZIU PowerDesigner

Celem pracy jest przedstawienie modelowania i generowania przykładowej bazodanowej aplikacji WWW w środowisku *Sybase PowerDesigner DataArchitect Suite 6.1* oraz w środowisku *Sybase PowerBuilder 7 Enterprise*.

1. WPROWADZENIE

W pracy [GrMaSt2004] autorzy przedstawili modelowanie i generowanie baz danych w środowisku *Sybase PowerDesigner DataArchitect Suite 6.1* oraz w środowisku *Sybase PowerBuilder 7 Enterprise*. Bazodanowa aplikacja jest to oprogramowanie, które umożliwia użytkownikowi pracę z wykorzystaniem zasobów bazy danych. Wśród systemów bazodanowych ważną grupę stanowią systemy biznesowe, opracowywane na potrzeby biznesu lub korporacji. Zależnie od wymagań i zastosowanych technologii mogą one posiadać różne architektury. Najpopularniejszą jest architektura klient-serwer stosowana w różnych odmianach. Architektura ta opiera się na dwóch typach oprogramowania: oprogramowaniu serwera, świadczącego usługi oraz oprogramowaniu klienta, korzystającego z usług tego serwera. Najczęściej oprogramowanie serwera i klienta jest osadzone na dwóch różnych komputerach połączonych siecią komputerową. Architektura klient-serwer może składać się z wielu warstw. W architekturze dwuwarstwowej po stronie serwera zgromadzone są dane,

* Politechnika Wrocławska, Wydział Informatyki i Zarządzania, Instytut Informatyki Stosowanej,
50-370 Wrocław, Wyb. Wyspiańskiego 27,
{Jacek.Grubler, Hanna.Mazur, Zygmunt.Mazur}@pwr.wroc.pl

które są udostępniane użytkownikowi, natomiast przetwarzanie tych danych jest po stronie klienta. W architekturze trzywarstwowej jest wyodrębniona pośrednia warstwa przetwarzania danych umieszczona na odrębnej maszynie, pomiędzy warstwą prezentacji danych a warstwą zarządzania danymi. Wielowarstwowość polega na rozbiciu warstwy przetwarzania danych na wiele części (ang. *partitioning*) znajdujących się najczęściej na różnych maszynach.

Celem pracy jest ukazanie przebiegu wykonania przykładu ćwiczeniowego obejmującego zaprojektowanie i wykonanie prototypu bazodanowego systemu informatycznego dla hipotetycznego przedsiębiorstwa „Telewizja kablowa”, którego model został uproszczony w stosunku do wymaganego modelu firmy istniejącej na rynku w rzeczywistości. Uproszczenia dotyczą przede wszystkim ilości modelowanych i implementowanych w prototypie danych i funkcjonalności. Jednak analitykę głównych funkcjonalności i danych systemu wykonano dla firmy rzeczywiście istniejącej i z zachowaniem należytej dokładności pozyskiwania wiedzy dziedzinowej i modelowania wycinka rzeczywistości.

Istnieje wiele narzędzi i środowisk CASE nowej generacji przeznaczonych do modelowania i projektowania sieciowych systemów bazodanowych i aplikacji wielowarstwowych. Jednym z nich jest *Sybase PowerDesigner*, dzięki któremu można projektować i wytwarzać aplikacje w architekturze dwu i trzywarstwowej. Niniejsza praca jest prezentacją możliwości tego narzędzia.

Na podstawie projektu aplikacji bazodanowej został wygenerowany automatycznie system bazodanowy z wykorzystaniem:

- systemu zarządzania bazą danych *Sybase Adaptive Server Anywhere 6.0.1*, (miejsce osadzenia baza danych),
- środowiska implementacyjnego *Sybase PowerBuilder 7.0*, do wygenerowania dwuwarstwowej aplikacji klient-serwer,
- środowiska implementacyjnego i Web serwera *Sybase PowerDynamo 3.0*, umożliwiającego wytwarzanie trzywarstwowych aplikacji klient-serwer z dostępem do WWW.

2. ZAŁOŻENIA DO PROJEKTOWANEGO SYSTEMU

Głównym i najważniejszym obszarem funkcjonowania telewizji kablowej, która jest przedmiotem naszych rozważań, jest dostarczanie sygnału stacji telewizyjnych i radiowych (nadawanych ze stacji naziemnych i drogą satelitarną) do mieszkań swoich klientów czyli abonentów. Abonenci mogą zamówić dodatkowe programy kodowane, co wpływa na cenę comiesięcznego abonamentu. Telewizja nadaje również własny program lokalny oraz umożliwia dostęp do telegazety zawierającej wszelkie informacje dla klientów.

Obsługą abonentów zajmuje się Biuro Obsługi Abonenta (BOA). Każdy klient przy zawieraniu umowy określa jeden lub wiele pakietów dostarczanych programów oraz zobowiązuje się do comiesięcznego opłacania abonamentu w odpowiedniej kwocie. Operator sieci natomiast zobowiązuje się do zamontowania gniazda telewizji w mieszkaniu abonenta, przesyłania sygnału obejmującego programy satelitarne, naziemne, polsko i obcojęzyczne zgodnie z wybranym pakietem. Zawierając umowę abonent nie staje się właścicielem urządzeń niezbędnych do odbioru programów (rozprowadzonych przez operatora sieci) i z chwilą wygaśnięcia umowy jest zobowiązany do ich zwrotu. W BOA rejestruje się wpłaty dokonywane przez abonenta (abonamentowe i za dokonywane usługi).

Operator sieci ma prawo zmienić wysokość opłat w przypadku wzrostu opłat licencyjnych, opłat KRRiTV, organizacji zbiorowego zarządzania prawami autorskimi i pokrewnymi (ZASP), ZAIKS), opłat retransmisyjnych, kosztów eksploatacji itd. O zmianach opłat informuje abonentów umieszczając odpowiednie informacje na kanale informacyjnym co najmniej 14 dni przed wprowadzeniem nowych opłat. Brak pisemnego oświadczenia odbiorcy o nieprzyjęciu nowych warunków umowy jest równoważne z akceptacją nowej wysokości opłaty. Rozwiązanie umowy następuje po opłaceniu ostatniego abonamentu. Opłata abonamentowa musi być wpłacona do 10. dnia każdego miesiąca rozliczeniowego. W przypadku nieuiszczenia wpłaty w należnym terminie, naliczane są odsetki karne. Jeśli zaległości w opłacie abonamentu sięgają dwóch miesięcy, abonent może zostać odłączony od sieci telewizyjnej. Ponowne podłączenie może nastąpić po uregulowaniu wszelkich należności.

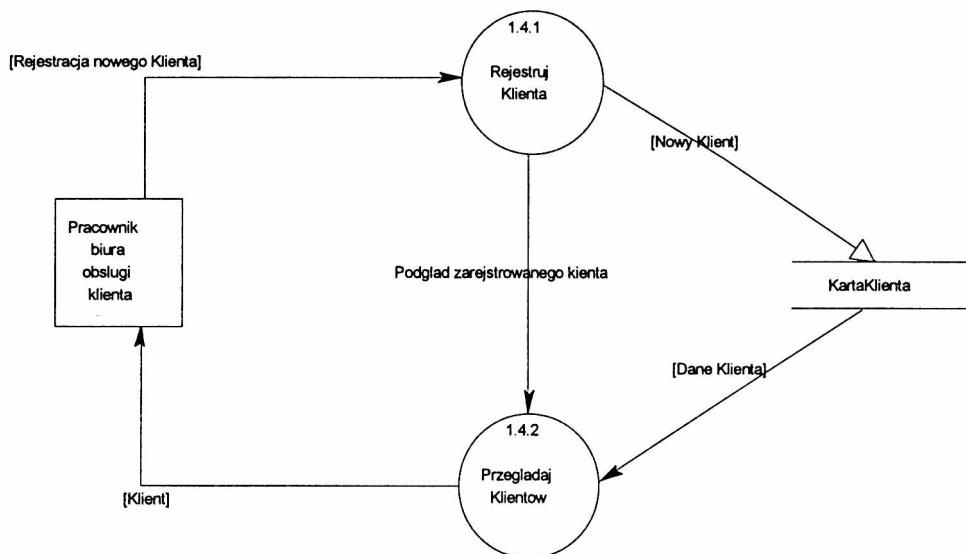
Abonent może złożyć czasową rezygnację z korzystania z usług telewizji kablowej. Ponowne podłączenie nastąpi po wniesieniu opłaty za aktywację po rezygnacji czasowej.

W przypadku kłopotów technicznych z odbiorem programów lub chęci złożenia zamówienia na oferowane usługi klient może zgłosić ten fakt w BOA, gdzie zostaje wystawione zlecenie na wykonanie usługi. Przyjęte zgłoszenie zostaje przekazane do Działu Serwisu telewizji. W dziale tym monterzy pełnią całodobowe dyżury. Po otrzymaniu zgłoszenia usługi do abonenta wysyłany jest monter. Wykonanie usługi odnotowywane jest w BOA. Usterka powinna być usunięta w ciągu 7 dni. W przypadku braku możliwości jej usunięcia, operator sieci powiadamia zgłaszającego o planowanym terminie jej usunięcia.

3. DIAGRAM PRZEPŁYWU DANYCH

Za pomocą środowiska projektowego *PowerDesigner* opracowano diagramy przepływu danych. Rysunek 1 przedstawia diagram dekompozycji procesu Klienci.

Process Model		
Project : PRO Telewizja Kablowa		
Model : Klienci		
Author : Jan Kowalski	Version 1.0	03-09-15



Rys. 1. Dekompozycja procesu Klienci

4. DEFINIOWANIE REGUŁ BIZNESOWYCH

Reguła biznesowa odwzorowuje semantykę modelowanego procesu. Semantyka narzucana jest przez uregulowania prawne, wymagania użytkowników albo regulacje wewnętrzne modelowanego przedsiębiorstwa lub organizacji. Początkowo reguły są prostymi obserwacjami z wycinka rzeczywistości, a w miarę rozwoju projektu przekształcane są one w coraz bardziej ściśle i szczegółowe wyrażenia dokumentujące i specyfikujące model procesowy. Reguły biznesowe i reguły danych uzupełniają graficzną postać modelu.

Tworząc reguły biznesowe (rysunek 2) formułuje się je odpowiadając na następujące pytania: (A) jakimi problemami biznesowymi zajmujemy się przy modelowaniu, (B) czy są jakieś procedury, które system musi przestrzegać, (C) czy jakaś specyfikacja wyznacza zakres projektu, (D) czy jakieś ograniczenia nie zawężają pewnego wyboru, (E) jak opisać każdą z procedur, specyfikacji i ograniczeń, (F) jak

sklasyfikować każdy z opisów – jako definicję, fakt, formułę czy wartościowanie. Gdy zbudujemy regułę, możemy ją zastosować do jakiegoś obiektu.

Business Rule Definition

Name: = Code: =

Type: Label:

Description:

Expression: ☐ Client ☒ Server

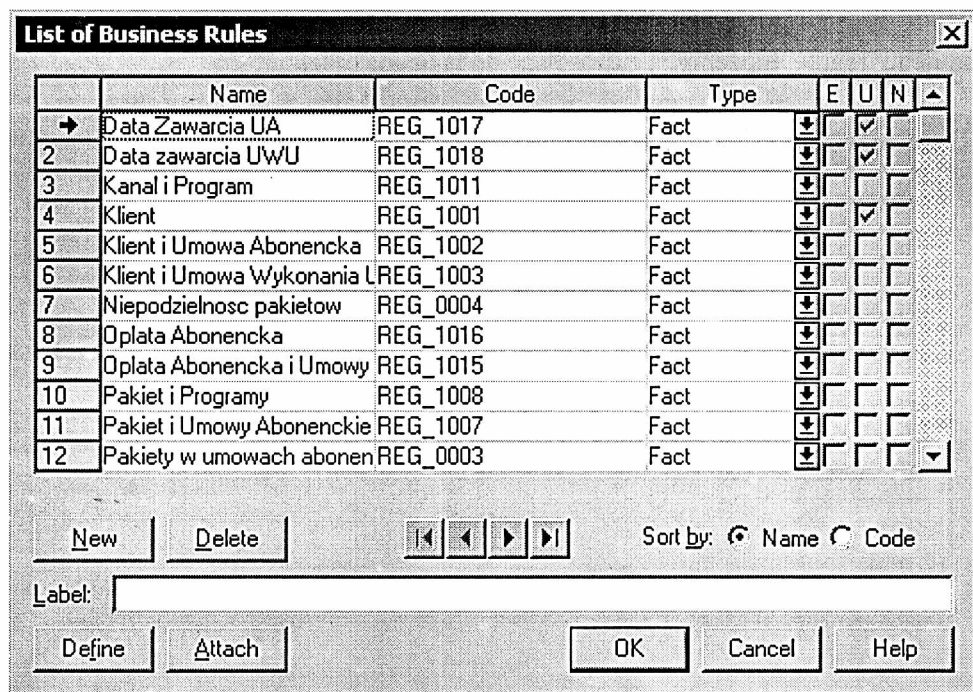
Rys. 2. Definiowanie reguł biznesowych

W modelu *ProcessAnalyst (PAM)* wyróżniamy cztery rodzaje reguł: definicja, fakt, formuła, wartościowanie (walidacja).

Reguły biznesowe początkowo mają najczęściej charakter opisowy. W miarę rozwijania modelu i analizowania problemu można je uzupełnić o znaczenie techniczne zastępując wyrażeniami. Rysunek 3 przedstawia przykładowy spis reguł biznesowych.

Każda reguła biznesowa zawiera dwa typy wyrażień:

- wyrażenia typu Klient,
- wyrażenia typu Serwer.



Rys. 3. Lista reguł biznesowych

Wyrażenie dotyczące klienta używane jest głównie w celach dokumentacyjnych. W bazie danych można wygenerować tylko wyrażenie dotyczące serwera. Tworzymy je jako wyrażenie kontroli (ang. *check parameters*) – jeżeli jest związane z tabelą, dziedziną, kolumną. Natomiast oba typy wyrażeń są stosowane w wyzwalaczach lub procedurach wbudowanych. Standardowe wyrażenia serwera można edytować w oknie dialogowym *Check Parameters*. Można tu określać domyślne wartości parametrów standardowych. Podczas generowania parametrów wartościowania tworzone są zmienne z wartościami standardowymi dla zmiennych z reguł walidacji. Trzy takie zmienne o ustalonych nazwach (spośród sześciu możliwych nazw) widoczne są na rysunku 4. w oknie *Expression*.

Składnia wyrażenia zależy od docelowej bazy danych. Wyrażenia kontroli opisują zakres danych (standardowe parametry) i reguły walidacji. Można je dołączyć do definicji atrybutów i definicji domen. Pozwala to na dostrojenie czynników analizy zanim zaimportujemy definicje atrybutów i domen do koncepcyjnego modelu danych CDM. Modelowanie danych w środowisku *PowerDesigner* jest omówione w pracy [GruMazSta2004].

Narzędzie *PowerDesigner ProcessAnalyst* umożliwia definiowanie danych na poziomie dziedzin (rysunek 5) i atrybutów (rysunek 6). Dziedziny ułatwiają identyfikację typów informacji w projekcie. Stosując dziedziny atrybutów można łatwiej standaryzować charakterystykę danych.

Check Parameters [X]

Data Item: statusUA (Status Umowy)
 Data type: A Length: 1 Precision:

Standard Parameters | Validation Rules

Values

Minimum:

Maximum:

Default: A

Parameters

Unit:

Format:

☐ Lowercase ☒ Uppercase

☐ Cannot modify

List of values

	Value	Label
→	A	Aktywna
2	Z	Zawieszona
3	R	Rozwiązana

Insert Delete [Down] [Up]

Expression: ☐ Client ☒ Server

%MINMAX% and
 %LISTVAL% and
 %RULES%

Rules Zoom Preview OK Cancel Help

Rys. 4. Definiowanie wyrażeń kontroli parametrów

List of Domains [X]

	Name	Code	Type	U	N
→	Walutowy	= PLN	MN8,2	☒	☐

New Delete [Left] [Right] Sort by: ☒ Name ☐ Code

Label:

Check Describe Annotate Used By OK Cancel Help

Rys. 5. Okno dialogowe służące do edycji dziedzin

List of Data Items [X]

Display Data items: ☐ Used ☐ Unused ☒ All Used by: ☒ Name ☐ Code

	Name	Code	Used By	Type	U	N	▲
10	Id Klienta	idKlienta	Nowy Klient;Dane Klient	NO6	☒	☐	
11	Id Usługi	idUslugi	Nowa umowa wykonanie	NO4	☒	☐	
12	Imie	imie	Nowy Klient;Rejestracja iVA20		☒	☐	
13	Kwota abonamentu	kwotaAbonamentu	Nowy pakiet;Informacje	MN8,2	☒	☐	
14	Kwota Opłaty Abonentki	kwotaOA	Dane OA do opłaty abon	MN8,2	☒	☐	
→ 15	Kwota wpłaty	kwotaPozycjiUWU	Nowa umowa wykonanie	MN8,2	☒	☐	
16	kwotaUWU	kwotaUWU	Informacje o opłacie za t	MN8,2	☒	☐	
17	liczbaAbonentow	liczbaAbonentow	Raport o pakietach	I	☒	☐	
18	liczbaZakupionychUslug	liczbaZakupionychUslug	Raport o uslugach	I	☒	☐	▼

New Delete |< < > >| Sort by: ☐ Name ☒ Code ☐ Used by

Label: Kwota wpłaty

Domain: PLN ▼ ...

Check Describe Annotate Used By OK Cancel Help

Rys. 6. Okno dialogowe umożliwiające edycję atrybutów

W *PAM* można skojarzyć z dziedziną następujące informacje: typ danych, długość, precyzję oraz reguły biznesowe.

Definiowanie danych

Elementarnym pojęciem w modelu danych i w słowniku danych jest atrybut. Zamiast tworzyć atrybuty bezpośrednio w obiekcie, zalecane jest tworzenie atrybutów w słowniku, a dopiero potem przypisywanie ich obiektom.

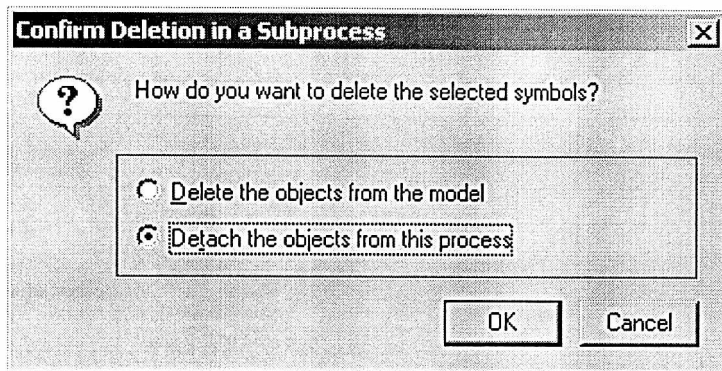
Definiowanie procesów

W modelu *PAM* występuje jeden wyróżniony model zwany modelem kontekstowym. Model kontekstowy to model *PAM* który posiada główny proces – rodzic poziomu 0. Każdy proces można zdekomponować, czyli podzielić na procesy niższego poziomu.

Obiekty globalne i lokalne

W modelu *PAM* wyróżniamy dwa typy obiektów: obiekty globalne i obiekty lokalne. Obiekty globalne to: składnice danych i terminatory (ang. *external entity*), a obiekty lokalne to: procesy, przepływy danych, przepływy przenoszone (ang. *migrated flow*), rozgałęźniki/koncentratory (ang. *split/merge*). Definicje obiektów są

przechowywane w słowniku danych. Obiekt to symbol graficzny i jego definicja w słowniku. Przy usuwaniu obiektu globalnego (rysunek 7) można zachować jego definicję słownikową.



Rys. 7. Dialog usuwania obiektu globalnego

Na każdym poziomie dekompozycji *PAM* używa się tych samych symboli globalnych. Obiekty lokalne należą do konkretnych poziomów przetwarzania danych, przepływów danych działających wewnątrz procesów (diagramów) i przepływów między procesami (w ramach diagramów i/lub pomiędzy nimi).

Macierz CRUD

Macierz *CRUD* (skr. ang. *Create, Read, Update, Delete*) jest tablicą, która pokazuje powiązania pomiędzy procesami i składnicami danych, albo pomiędzy procesami i atrybutami lub jednostkami danych. Gdy powiązanie istnieje, to pokazuje ono czy proces wykonuje operację tworzenia, czytania, aktualizacji lub usuwania ze składnicy – danych lub atrybutów albo jednostek danych.

W *ProcessAnalyst* istnieją dwa typy macierzy *CRUD*:

- (a) macierz *CRUD* proces/składnica danych (rysunek 8), w której powiązania istnieją pomiędzy procesami a składnicami danych; macierz pokazuje operacje wykonywane przez proces na składnicy danych,
- (b) macierz *CRUD* proces/attribut (jednostka danych) pokazująca operacje wykonywane przez proces na atrybucie lub jednostce danych (rysunek 9).

Macierz *CRUD* używana jest do przeglądania powiązań między procesami i używanymi przez nie danymi oraz wskazywania, do których danych z jakich składnic poszczególne procesy nie mają dostępu, oraz jakiego typu akcje są wykonywane na poszczególnych danych. Umożliwia to wykonywanie ekspresywnej weryfikacji modelu.

CRUD Matrix

	Administrowanie	Biuro Kontr...	Biuro Ob...	Generuj ...	Generuj ...	Kl
Kanal	CR					
KartaKlienta			CR			
OfertaPakietu	CRU		R	R		
OplataAbone	C	R	RU			
ProgramTK	CR					
UmowaAbone	R		CRU	R		
UmowaWykon	R	R	CRU		R	
Usługa	CR		R		R	

Display ☒ Name ☐ Code Sort by ☒ Name/code ☐ Number

Operations: ☒ Create ☒ Read ☐ Update ☐ Delete

Process: Administrowanie 3 ...

Data Store: Kanal ...

Rys. 8. Macierz CRUD proces/składnica

CRUD Matrix

	Administrowanie	Biuro Kontr...	Biuro Ob...	Generuj ...	Generuj ...	
Czas realizacji	CR				R	
Częstotliwość	CRU			R		
Data wpłaty			RU			
Data wpłaty z			RU			
Data wykonania		R	R			
Data zawarcia		R	CR			
Data zawarcia			CR			
Do Wpłaty za						

Display ☒ Name ☐ Code Sort by ☒ Name/code ☐ Number

Operations: ☒ Create ☒ Read ☐ Update ☐ Delete

Process: Administrowanie 3 ...

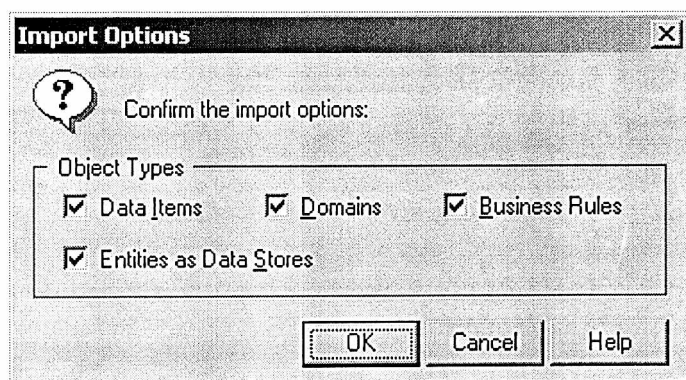
Data Item: Czas realizacji ...

Rys. 9. Macierz CRUD proces/atribut (jednostka danych)

Jeżeli używamy *PAM* do modelowania konceptualnego, to w wyniku otrzymuje się model konceptualny *CDM*. Z modelu *CDM* można następnie wygenerować model fizyczny (skr. ang. *PDM*). W modelu *CDM* z kolei składnice danych mogą stać się tabelami w bazie danych. Z typów operacji przedstawionych w macierzy *CRUD* wynika, jakiego typu dostępu potrzebują użytkownicy bazy danych do poszczególnych tabel.

Importowanie definicji danych z DataArchitect

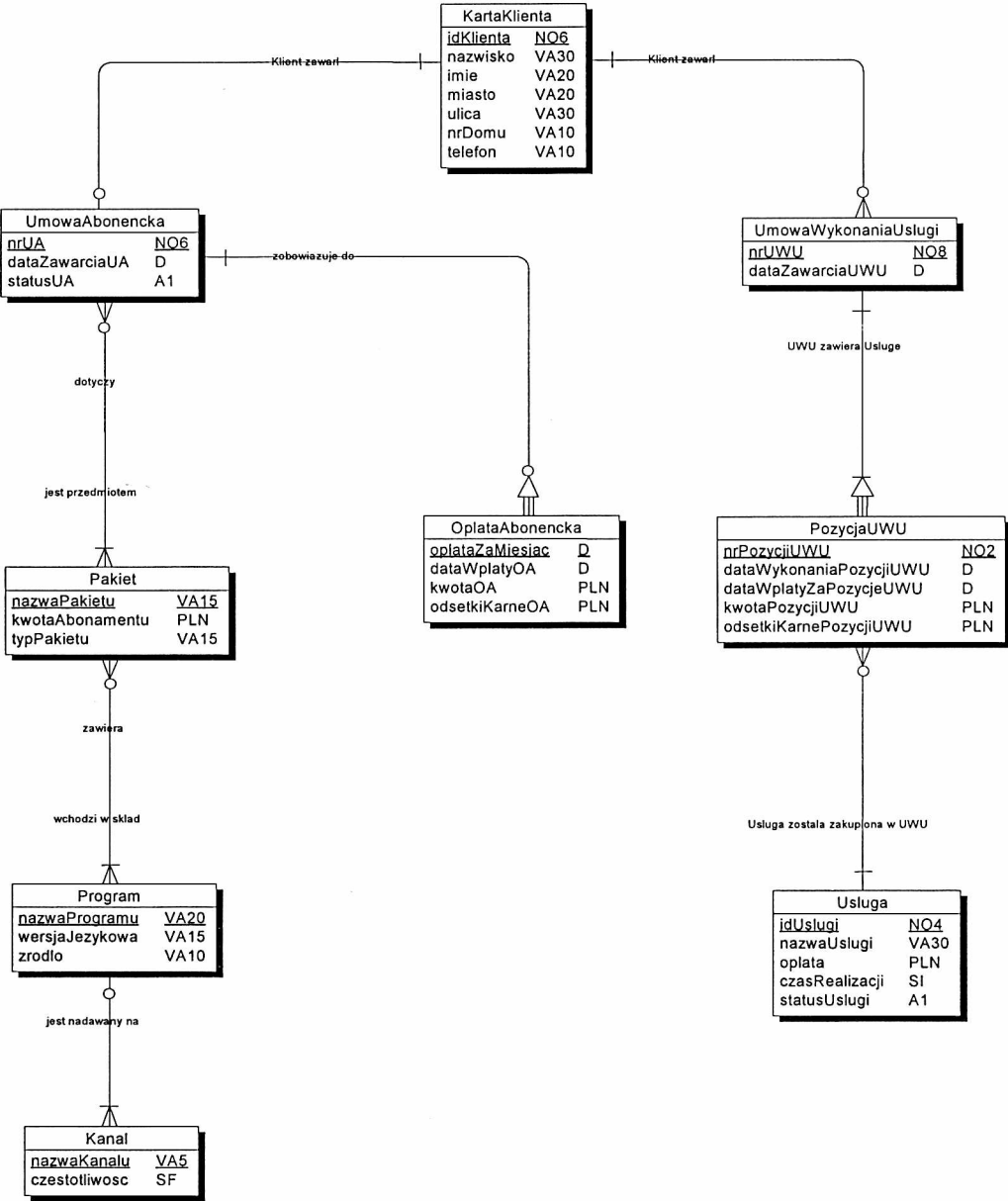
Narzędzie *ProcessAnalyst* daje możliwość importowania danych z modelu konceptualnego *CDM*. Ma to szczególne znaczenie dla utrzymania zgodności całego projektu. Importowaniu podlegają następujące elementy *CDM*: atrybuty, dziedziny, reguły biznesowe oraz encje, które w modelu *PAM* reprezentowane są jako składnice danych. Ważną funkcjonalnością jest tu możliwość wyboru danych do importu (rysunek 10). Funkcjonalność ta przydaje się podczas aktualizowania modelu *PAM*, na przykład, gdy chcemy uzyskać zgodność jedynie na poziomie atrybutów.



Rys. 10. Importowanie definicji danych z modelu *CDM*

5. MODEL KONCEPTUALNY

Przedstawimy teraz konceptualny model danych zbudowany dla systemu „Telewizja kablowa” z projektu ćwiczeniowego (rysunek 11). Reprezentacja projektowa modelu obejmuje definicje atrybutów i ich dziedziny, zbiorów encji, a także definicje typów, liczności i innych właściwości powiązań między zbiorami encji.



Rys. 11. Konceptualny model danych CDM dla systemu **Telewizja kablowa**

5.1. DEFINIOWANIE REGUŁ BIZNESOWYCH, DZIEDZIN I ATRYBUTÓW LUB JEDNOSTEK DANYCH

Reguły biznesowe w *CDM* mają taki sam charakter jak reguły biznesowe w modelu funkcjonalnym *PAM*. Gdy je tworzymy, odnosimy się do pojęć z opisu wycinka rzeczywistości. Również w *CDM* istnieją cztery typy reguł biznesowych. Są to: definicja, fakt, formuła, walidacja. Nie są one natomiast tłumaczone na kod wykonywalny. Kod taki jest tworzony wyłącznie z modelu *PAM*.

Analogicznie do postępowania przy tworzeniu modelu funkcjonalnego *PAM*, w modelowaniu *CDM* możemy definiować jednostki danych (ang. *data items*) i dziedziny. Definiując dziedziny można określić parametry wartościowania (ang. *check parameters*), zapewniające bardziej ściśle definiowanie dziedzin i ich specyfikację oraz bardziej ściśle definiowanie dozwolonych wartości jednostek danych i atrybutów. Zamiast tworzyć atrybuty bezpośrednio dołączane do encji, zaleca się ich tworzenie w pierwszej kolejności w słowniku danych, a dopiero później dołączanie do encji. Składnica danych z modelu *PAM* (ze składnika *ProcessAnalyst*) może być przekształcona w encję w modelu *CDM* (ze składnika *DataArchitect*). Może to nastąpić tylko wtedy gdy zaznaczono podczas definiowania składnicy danych opcję *is entity*.

5.2. DEFINIOWANIE ENCJI

W *DataArchitect* encję można zaprojektować bardzo ekspresywnie i precyzyjnie, dokładniej niż w jakimkolwiek innym natywnym lub uniwersalnym systemie projektowym. Służy do tego rozbudowane definicyjne okno dialogowe. Pojęcie encja modelu *CDM* jest tu rozumiane tak jak potocznie (i skrótowo) posługują się nim analitycy (i czasami projektanci) przy tworzeniu modelu logicznego bazy danych, tzn. pojęcie to jest tożsame z pojęciem kategorii danych w modelu logicznym bazy danych. Encja w *CDM* reprezentuje zatem obiekt zdefiniowany wewnątrz systemu informacyjnego, o którym chcemy gromadzić informacje. Wystąpienie encji to element należący do encji.

Atrybuty encji są elementarnymi jednostkami danych związanymi z encjami. W *CDM* istnieją cztery sposoby tworzenia atrybutów encji: (1) wykorzystanie jednostek danych jako atrybutów encji, (2) zduplikowanie jednostek danych jako atrybutów encji, (3) ponowne użycie jednostek danych jako atrybutów encji, (4) stworzenie atrybutów encji bezpośrednio na liście atrybutów. Zatem termin jednostka danych (ang. *data item*) oznacza zwykle to co atrybut, lecz nie zawsze. W modelu *CDM* jednostką danych nie musi być atrybut. Atrybut tworzymy tylko czasami bezpośrednio na liście atrybutów encji. Zwykle jako atrybutów używamy jednostek

danych przyłączając te jednostki do encji. Nie tworzymy zatem nowej jednostki danych a atrybut encji posiada ten sam kod co istniejąca już jednostka danych.

Z naciskiem podkreślamy, że w *CDM* każda encja powinna być jednoznacznie identyfikowana. Identyfikator encji składa się z jednego lub wielu atrybutów encji takich, że każda wartość identyfikatora odpowiada jednemu i tylko jednemu wystąpieniu encji. Jeżeli spośród atrybutów encji nie można wyznaczyć podzbioru jednoznacznie identyfikującego tę encję, wprowadza się sztuczny identyfikator. Identyfikator pełni zatem rolę surogatu klucza głównego, lecz na razie nim być nie może ponieważ model *CDM* może nie być znormalizowany oraz nie są w nim ujęte wymagania techniczne i implementacyjne, a także nie spełnia on wymagań rewersowych i nie daje możliwości obsługi przez sterowniki dostępu *ODBC* i *JDBC*.

Z encjami można również powiązać reguły biznesowe nakładane na te encje.

5.3. Definiowanie związków

Powiązanie jest szczególnym, nazwanym związkiem pomiędzy encjami. Związek binarny wyraża fakt, że dwie encje są ze sobą w jakiś sposób powiązane. W modelu *CDM* jest również możliwe określanie liczności (ang. *cardinality*) wskazującej maksymalną liczbę instancji encji (jedna lub wiele) pozostających w związku z inną encją – z opcjonalnością (obligatoryjnością) takich powiązań włącznie (rysunek 12).

Jeżeli z definiowanym związkiem wiązą się pewne informacje (posiada on atrybutową strukturę), to taki związek może zostać przekształcony na encję asocjacyjną powiązaną przez dwa nowe powiązania między encjami powiązanymi poprzednio związkiem strukturalnym. Dwa nowe powiązania są bez struktury. Encja pośrednicząca i dwa nowe powiązania bez struktury „zastępują” zatem jedno powiązanie posiadające strukturę. Następnie można dołączyć do nowej encji pośredniczącej atrybuty, których na przykład nie można było przedtem dołączyć do związku. Encja asocjacyjna przyjmuje nazwę i kod związku, z którego ją wyprowadzono.

W modelu konceptualnym składnika *DataArchitect* wyróżniamy również pojęcia takie jak **związek zależny** oraz **związek dominujący**. Dzięki nim uzyskujemy zaawansowane możliwości logicznego modelowania danych i projektowania encji. W związku zależnym jedna encja jest częściowo identyfikowana przez inną. Każda encja musi posiadać identyfikator, aczkolwiek w niektórych przypadkach atrybuty encji nie są wystarczające do zidentyfikowania instancji encji. W tym przypadku identyfikator encji zależnej zawiera identyfikator innej encji. W graficznej reprezentacji związek zależny można rozpoznać po trójkącie przy punkcie końcowym związku.

W powiązaniu jeden do jeden można zdefiniować jeden kierunek związku jako dominujący. Jeżeli zdefiniuje się dominujący kierunek, związek jeden do jeden wygeneruje tylko jedną referencję w modelu fizycznym *PDM*, który opisujemy niżej. Jeżeli natomiast nie zdefiniuje się dominującego kierunku, to związek jeden do jeden

wygeneruje dwie referencje, co może mieć niekorzystny wpływ na działanie przyszłej bazy danych i przyczyniać się do straty wydajności obliczeniowej systemu *DBMS*.

Relationship Properties

Definition | Description | Annotation

Karta Klienta — Umowa Abonencka

Name: UAKlienta = Code: UAKlienta =

Label: Umowa Abonencka Klienta

Cardinality

☒ One to One ☐ One to Many ☐ Many to One ☐ Many to Many

"Karta Klienta" to "Umowa Abonencka"

☐ Mandatory ☐ Dependent ☒ Dominant Min: 0 Max: 1

Role: Klient zawarł

"Umowa Abonencka" to "Karta Klienta"

☒ Mandatory ☐ Dependent ☐ Dominant Min: 1 Max: 1

Role:

Rules ☒ Generate

OK Cancel Apply Help

Rys. 12. Okno dialogowe Relationship Properties służące do definiowania związków

5.4. Definiowanie dziedziczenia encji

Dziedziczenie jest związkiem, który pozwala zdefiniować encję jako specjalny przypadek encji bardziej ogólnej. Encje biorące udział w dziedziczeniu posiadają dużo cech wspólnych, niemniej jednak są różne. Encję ogólną nazywamy supertypem (ang. *supertype*), encją nadrzędną lub encją rodzicem. Zawiera ona wszystkie wspólne

charakterystyki. Przypadki specjalne tej encji nazywamy podtypami (ang. *subtype*) lub encjami potomnymi. Zawierają one wszystkie charakterystyki szczególne.

W modelu CDM istnieje możliwość ustanowienia dziedziczenia jako dziedziczenia wspólnie rozłącznego. Wystąpienie tego dziedziczenia oznacza, że jednocześnie nie mogą istnieć instancje obu potomków. Taka transformacja służy jedynie celom dokumentacyjnym i nie wpływa na generowany model *PDM*.

Fizyczną implementację struktury dziedziczenia definiuje tryb generowania. Tryb generowania w modelu CDM wskazuje, które encje struktury dziedziczenia muszą odpowiadać tabelom w modelu *PDM*. Można tutaj wskazać, czy generowane będą tabele dla:

- encji rodzica (opcja *Generate Parent*),
- dla encji potomnych (opcja *Generate Children*).

Po wybraniu opcji *Generate Parent*, wygenerowana zostanie tabela odpowiadająca encji rodzica w modelu fizycznym w następujący sposób:

- rodzic dziedziczy atrybuty każdej encji potomnej,
- powiązania bezpośrednio oddziałujące na encję nadrzędną oddziałują na tabelę odpowiadającą encji nadrzędnej,
- jeśli mamy powiązanie jeden do wielu z encją potomną, wygenerowana tabela rodzica zawierać będzie referencje do tabeli powiązanej za pomocą relacji wiele do jednego.

Jeśli wybierze się generowanie tabel dla encji potomnych, to dodatkowo należy określić, czy mają być dziedziczone wszystkie atrybuty encji nadrzędnej (opcja *Inherit All*), czy tylko identyfikator encji nadrzędnej (opcja *Inherit Identifier*). Gdy generujemy tabele potomne, klucz główny w tej tabeli jest konkatencją identyfikatora encji podrzędnej i identyfikatora encji nadrzędnej.

W przypadku gdy nie wybierze się opcji *Generate Children* oraz opcji *Generate Parent*, *PowerDesigner* generuje tabele odpowiadające encjom potomnym.

6. GENEROWANIE MODELU FIZYCZNEGO

Model *PDM* generuje się dla poszczególnych systemów zarządzania bazami danych *DBMS*. Przed wygenerowaniem modelu *PDM* należy wybrać jeden z oferowanych docelowych systemów *DBMS* dla docelowej bazy danych. *DataArchitect* dokonuje translacji typów danych wyspecyfikowanych w modelu *CDM* do fizycznych typów danych dostępnych w docelowej bazie danych. Odpowiedniość pomiędzy typami konceptualnymi i fizycznymi definiuje się w specjalnym pliku (*DEF*).

Generując model *PDM*, *DataArchitect* transformuje obiekty konceptualne w obiekty fizyczne. Identyfikatory encji generują klucze główne i obce w modelu *PDM*. Encje odwzorowują się w tabele, atrybuty encji w kolumny tabel, identyfikatory w klucze główne, powiązania jeden do wielu w referencje, powiązania jeden do

jednego w złączenia tabel lub referencje. Powiązania wiele do wiele, które są dopuszczalne w modelowaniu conceptualnym są „w sposób oszczędny” zamieniane na referencje za pośrednictwem tworzonej dodatkowo tabeli pośredniczącej. Tabela zawiera klucz główny złożony z identyfikatorów obu encji generujących (dla związków binarnych).

6.1. MODEL FIZYCZNY

Model *PDM* reprezentuje strukturę fizyczną danych, która zostanie zaimplementowana w docelowym systemie *DBMS*. Generowany model opiera się na solidnej podstawie środowiska projektowego modelowania conceptualnego. Bierze się tu pod uwagę cechy i fizyczne ograniczenia poszczególnych systemów *DBMS*.

Za pomocą *PDM* można optymalizować charakterystyki bazy danych środkami dodefiniowywania tabel, kolumn, indeksów, integralności referencyjnej, perspektyw, wyzwalaczy i procedur pamiętanych. Model *PDM* specyfikuje fizyczną implementację bazy danych.

Model *PDM* spełnia następujące role:

- (a) reprezentuje fizyczną organizację danych w formie graficznej,
- (b) generuje skrypty tworzenia bazy danych i jej modyfikacji,
- (c) definiuje wyzwalacze i więzy integralności referencyjnej,
- (d) generuje atrybuty rozszerzone,
- (e) rewersuje istniejące bazy danych.

Model *PDM* reprezentuje interakcję obiektów takich typów jak tabele, kolumny, klucze, indeksy, referencje perspektywy. Definicje tabel zawierają definicje kolumn w bazie docelowej. Aby ustandaryzować charakterystyki typów kolumn w różnych tabelach stosuje się domeny. Domeny identyfikują typy informacji w projekcie. Definiują one zbiory wartości dopuszczalnych w kolumnach. W modelu *PDM* można wiązać następujące informacje o domenach: (1) typ danych, długość i precyzję, (2) kontrolę parametrów, (3) reguły biznesowe, (4) właściwości obligatoryjne, (5) atrybuty rozszerzone, (6) profile danych. W docelowych systemach *DBMS* dobrej jakości można specyfikować domeny abstrakcyjnych typów danych (skr. ang. *ADT*). Są to typy domen zdefiniowane przez użytkownika enkapsulujące szereg wartości danych i funkcji. Funkcje są zdefiniowane i operują na zbiorze wartości. Przykładem może być abstrakcyjny typ danych dla kalendarza Gregoriańskiego z definicjami następujących funkcji: (a) pisania i czytania numerów rzymskich, (b) konwersji danych z kalendarza Juliańskiego na Gregoriański, (c) konwersji danych z kalendarza Gregoriańskiego na Juliański.

Na poszczególne kolumny można nałożyć ograniczenia. Spowoduje to wygenerowanie więzu integralności w docelowej bazie danych. Ograniczenia można też nakładać na całe tabele co również spowoduje wygenerowanie więzu. Więz jest

nazwaną kontrolą, która wymusza odpowiedniość sprawdzanych parametrów. Można tutaj wyspecyfikować nazwę więzu oraz wyrażenia serwera i klienta.

Każdy klucz może generować indeks albo unikalny więz w docelowej bazie danych. W *PDM* można tworzyć trzy rodzaje kluczy: kandydujący, główny i obcy. Tworzenie indeksów dla kolumn klucza obcego i głównego odbywa się w *DataArchitect* automatycznie.

Dla modelu fizycznego *PDM* można utworzyć jego użytkowników. Użytkownik modelu *PDM* to nazwa identyfikująca osobę lub grupę będącą właścicielem obiektów z modelu - tabel lub perspektyw. System projektowy *Sybase PowerDesigner* (składnik *DataArchitect*) zapewnia, że po akcji wygenerowania modelu fizycznego, modyfikacje modelu *CDM* nie powodują utraty żadnego szczegółu fizycznej implementacji zdefiniowanej już w wygenerowanym uprzednio modelu – chyba, że model *PDM* zostanie zastąpiony inną wersją. Model *PDM* można denormalizować dodając do niego ręcznie dodatkowe tabele a do tabel tych dodawać atrybuty i tworzyć klucze główne.

6.2. DEFINIOWANIE REFERENCJI I INTEGRALNOŚCI REFERENCYJNEJ

W modelu *PDM* można utworzyć referencje między tabelami. Wówczas pewna kolumna może wskazywać inną tabelę. Rysunek 13 przedstawia okno dialogowe Referential integrity, w którym można zdefiniować własności wybranej referencji. Tworząc referencję dokonujemy (obliczanej) migracji klucza głównego z tabeli nadrzędnej (w rodzicu) na klucz obcy w tabeli potomnej – w sensie notacji łącznikowej Chena rodzic-potomek zależności referencyjnych encji. Jak wiemy integralność referencyjna dotyczy reguł zarządzania spójnością danych, w szczególności zależnościami między kluczami głównymi i obcymi w różnych tabelach.

Ustawiając opcję integralności referencyjnej w modelu *PDM* specyfikujemy reakcję na usunięcie lub modyfikację rekordu z pewnej tabeli. Możemy zdefiniować, czy operacja ta ma być zabroniona, czy też tabela potomna ma być uaktualniana kaskadowo. Integralność referencyjną można zdefiniować zmieniając właściwości poszczególnych referencji lub stosując wyzwalacz (ang. *trigger*). Dla docelowej bazy danych można zdefiniować integralność referencyjną za pomocą opcji generowania tej bazy. Dla wielu systemów *DBMS* nie ma możliwości definiowania integralności referencyjnej baz danych. Dlatego skrypt generujący bazę danych dla takiego *DBMS* produkowany przez *PowerDesigner* nie zawiera definicji wyzwalaczy i procedur kontrolujących integralność referencyjną tej bazy.

The dialog box 'Referential Integrity' displays a relationship between two tables: 'KartaKlienta' (Parent) and 'UmowaAbonencka' (Child). The Parent table has a 'Primary key' and the Child table has a 'Primary key' and a 'Foreign key'. The relationship is defined with the following settings:

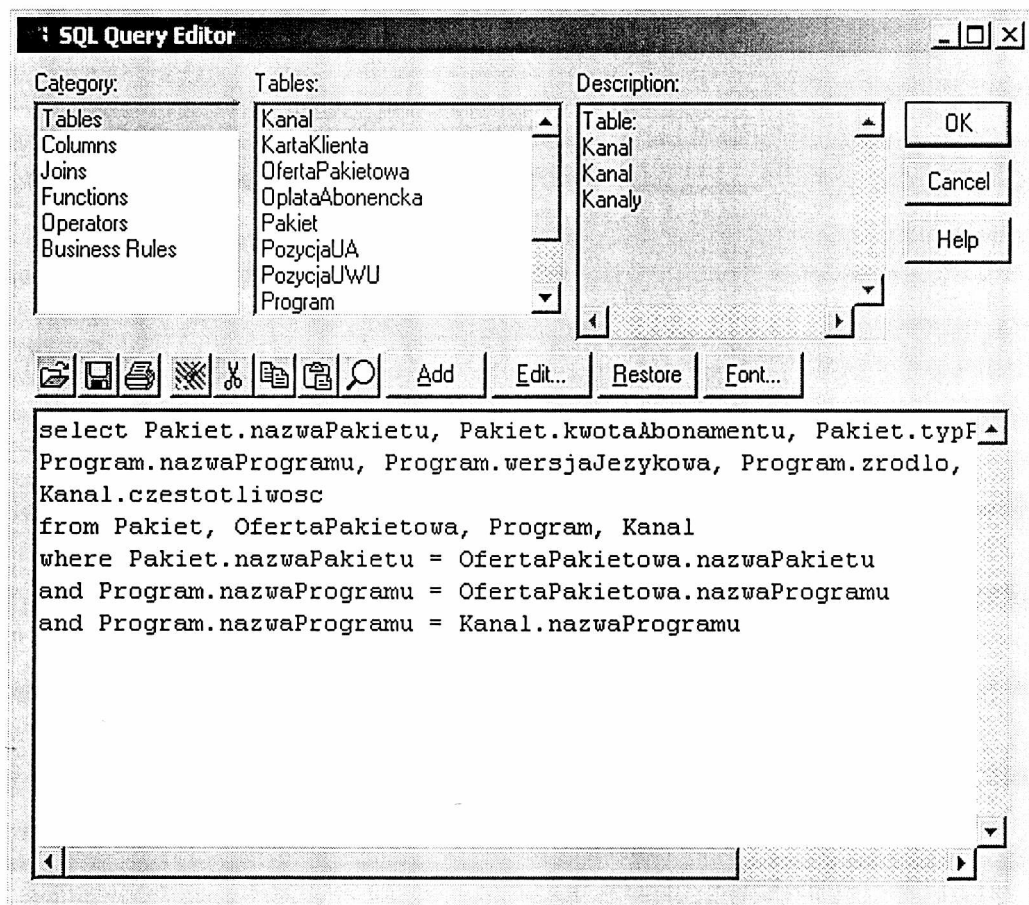
- Name:** Umowa Abonencka Klienta
- Code:** UAKlienta2
- Constraint name:** FK_UMOWAABO_UAKLIENTA_KAR
- User-defined:** ☐
- Cardinality:**
 - Number of children: Minimum: 0, Maximum: 1
- Update constraint:**
 - ☐ None
 - ☒ Restrict
 - ☐ Cascade
 - ☐ Set null
 - ☐ Set default
- Delete constraint:**
 - ☐ None
 - ☒ Restrict
 - ☐ Cascade
 - ☐ Set null
 - ☐ Set default
- Mandatory parent:** ☒
- Change parent allowed:** ☒
- Check on commit:** ☐

Buttons: OK, Cancel, Help

Rys. 13. Okno dialogowe Referential Integrity

6.3. TWORZENIE PERSPEKTYW

Perspektywy definiuje się formułując zapytania w języku *SQL* (rysunek 14). Istnieją dwa sposoby budowania perspektyw: (1) dla tabel wybranych w modelu *PDM*, (2) budowanie pustej perspektywy a następnie wybranie do niej tabel z listy tabel. Na podstawie wyboru tabel, kolumn i referencji *PowerDesigner* generuje zapytanie w języku *SQL* stosując dla kolumn powiązanych referencjami klauzule *WHERE*. W konsekwencji perspektywy zawierają kolumny, które użyto w zapytaniu *SQL*.



Rys. 14. Okno dialogowe SQL Query Editor, do definiowania zapytań *SQL* dla perspektyw

6.4. WŁAŚCICIELE OBIEKTÓW MODELU PDM

Użytkownik w *PDM* to nazwa identyfikująca projektanta lub grupę projektową, będącą właścicielem obiektów z modelu fizycznego. W modelu *PDM* właściciele mogą posiadać tabele lub perspektywy oraz czasami tabele reprezentujące powiązania w *CDM*. Domyślnie tworzona tabela nie ma właściciela, ale można go wybrać z listy i przypisać do danej tabeli. Każdy obiekt może mieć jednocześnie tylko jednego właściciela, który ma prawo do jego modyfikacji. Gdy do modelu ma dostęp wielu użytkowników, to dany użytkownik może wygenerować bazę danych zawierającą tylko te obiekty, które do niego należą.

6.5. WYZWALACZE I PROCEDURY WIĘZÓW INTEGRALNOŚCI REFERENCYJNEJ

Wyzwalacz jest specjalną postacią procedury pamiętanej. Wyzwalacze więzów integralności mogą być w modelu *PDM* generowane automatycznie w zależności od typu docelowej bazy danych. Użytkownik ma możliwość zdefiniowania dodatkowych wyzwalaczy.

PowerDesigner generuje wyzwalacze na podstawie wzorców szablonów. Każdy szablon wyzwalacza pobudza trzy rodzaje więzów integralności referencyjnej: wstawiania, modyfikacji oraz usuwania. Każdy docelowy *DBMS* posiada jeden wzorec wyzwalacza każdego typu.

System *Designer* operuje procedurami i funkcjami pamiętanymi w procesie budowania kodu dla wspomagających te procedury i funkcje systemów *DBMS*. Jednakże procedury i funkcje wbudowane nie zwracają wyników. Dlatego za pomocą funkcji i procedur pamiętanych nie można implementować więzów integralności.

Wyzwalacze i procedury bazy danych można tworzyć i modyfikować poprzez sterowniki *ODBC* albo bezpośrednio za pomocą skryptów.

6.6. TWORZENIE BAZY DANYCH

Bazę danych można wygenerować bezpośrednio z modelu *PDM* jako *pojemnik* bazy danych, albo można wygenerować tylko skrypt generujący bazę o określonym *schemacie* do uruchomienia w docelowym środowisku *DBMS*. Skrypty generuje się dla poszczególnych docelowych baz danych.

Dostępne parametry do generacji zależą od wyboru docelowej bazy danych. Domyślną docelową bazą danych jest baza wybrana przez projektanta przy otwarciu modelu *PDM*, ale można to zmienić po wygenerowaniu skryptu. Połączenie z bazą danych zapewnione jest poprzez interfejs *ODBC*, który umożliwia dostęp do danych bazy za pomocą języka *SQL* stanowiącego standard dostępu do danych.

Za pomocą narzędzia *PowerDesigner* można konfigurować przestrzenie tabel i pamięci. Przestrzeń tabel (ang. *tablespace*) i pamięć (ang. *storage*) wskazują na fizyczną lokację obiektów bazy danych (tabel i indeksów). Definicje przestrzeni i pamięci są specyficzne dla systemu *DBMS* i wybiera się je z list w oknach dialogowych. Dla różnych systemów *DBMS* stosuje różne określenia przestrzeni tabel i pamięci.

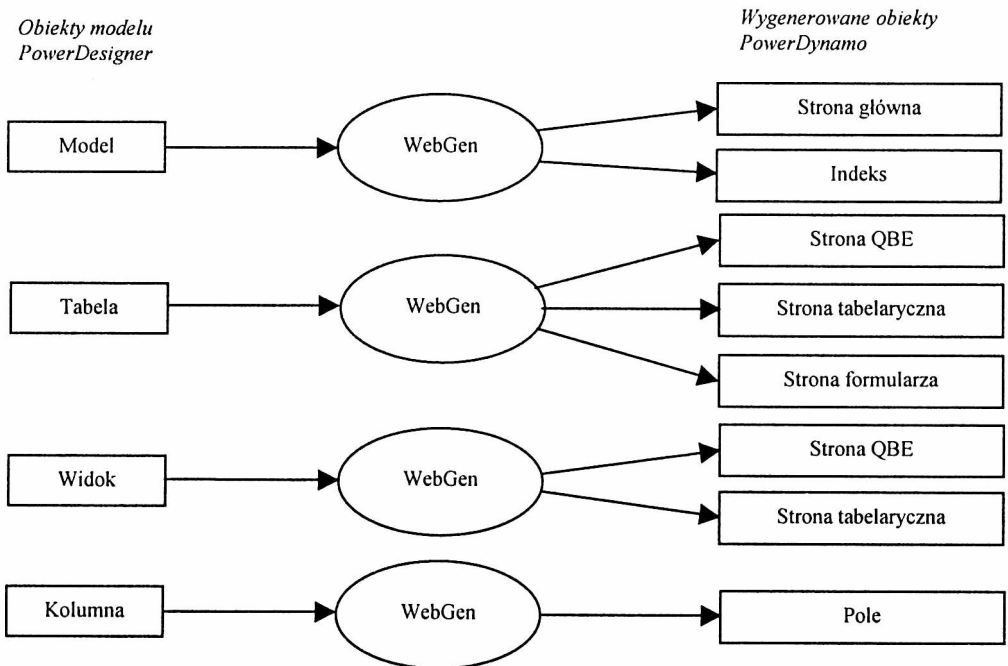
W różnych systemach bazodanowych stosuje się różne komendy do tworzenia przestrzeni tabel i pamięci. *PowerDesigner* pozwala utworzyć je dla bardzo wielu – w tym wiodących – systemów *DBMS*, na przykład wersja 6.0 tworzy bazy danych dla: *Allbase*, *DB2*, *Oracle 5*, *Oracle 6* lub *7*, *RDB 4* lub *5* lub *6*, *SQLBase*, *Sybase System 10* lub *System 11*, *SQL Server 7*, *SQL Anywhere Watcom 3* lub *4*.

6.7. GENEROWANIE DANYCH TESTOWYCH

Dane testowe są próbką danych, którą definiujemy dla jednej lub więcej tabel w modelu *PDM*. Można wygenerować skrypt testowy danych i uruchomić go w środowisku bazy danych, generując dane testowe. *PowerDesigner* automatycznie generuje wiersze danych testowych i umieszcza je w tabelach. Pozwala to oszacować rozmiar przestrzeni danych zajmowanej przez bazę oraz zweryfikować wydajność bazy zawierającej dużo danych lub udostępnianej wielu różnym aplikacjom lub użytkownikom. Mechanizm generowania danych zawiera dość zaawansowane możliwości funkcjonalne ale też pewne ograniczenia.

7. GENEROWANIE APLIKACJI

W skład pakietu *PowerDesigner* wchodzi generator aplikacji *AppModeler for the Web* (w skrócie *WebGen*) umożliwiający generowanie aplikacji typu klient – serwer [GruMazSta2004].



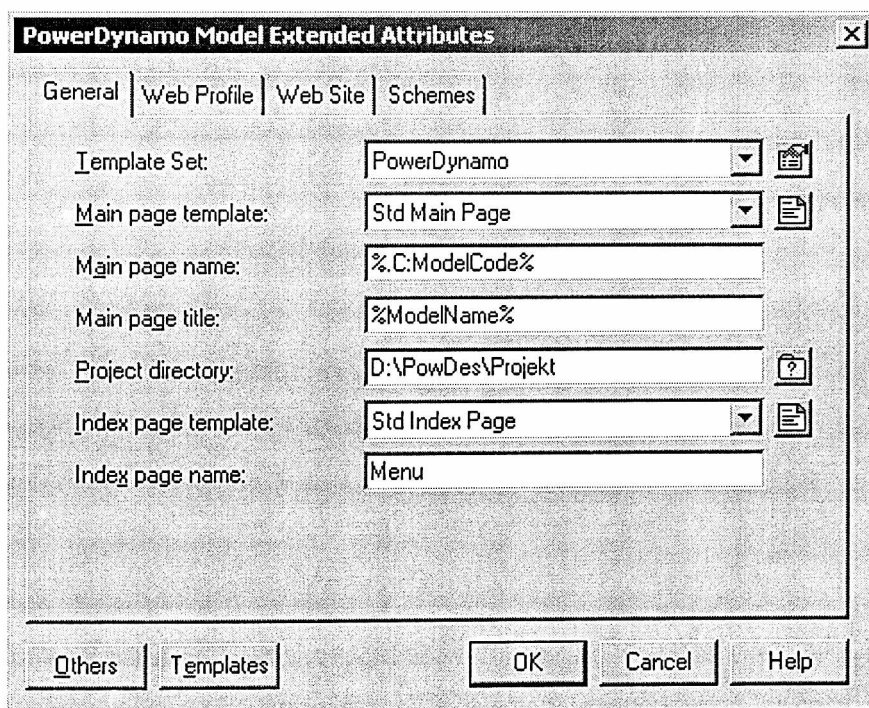
Rys. 15. Schemat transformacji obiektów podczas generowania aplikacji dla *PowerDynamo*

WebGen wykorzystuje dane z *PDM* do utworzenia obiektów z dostarczonych szablonów i wygenerowania aplikacji *WWW* (rysunek 15). Projekt witryny internetowej generowany jest bezpośrednio do wskazanego folderu *WWW* lub bazy danych skonfigurowanej dla *PowerDynamo*. Aplikacje, które generuje się dla *PowerDynamo* wymagają podania lokalizacji sieciowej rozpoznawanej przez serwer *PowerDynamo*.

7.1. TWORZENIE PROJEKTU WITRYNY

WebGen generuje projekt strony głównej służący do zarządzania aplikacją sieci Web. Strona główna zbudowana jest na ramkach i zawiera indeks stron wygenerowanych dla tabel i widoków.

WebGen generuje skrypty serwera poprzez tworzenie instancji szablonów z wartościami pochodzącymi z modelu *PDM*. Użytkownik w oknie dialogowym *PowerDynamo Model Extended Attributes* (rysunek 16) dołącza do modelu fizycznego szablon strony głównej i indeksu oraz definiuje pewne właściwości projektu. W oknie tym konfiguruje również ogólne właściwości projektu, profil sieci (ang. *Web profile*), lokalację witryny internetowej, zbiór szablonów schematu.



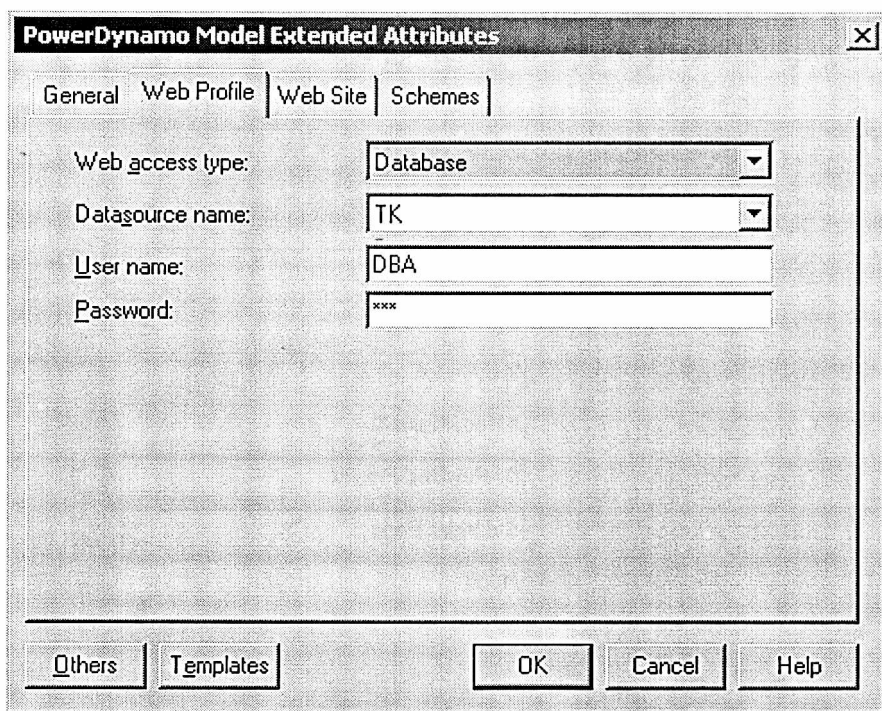
Rys. 16. Okno dialogowe *PowerDynamo Model Extended Attributes*, zakładka *General*

Użytkownik musi wybrać typ dostępu do swojej witryny. Zależnie od typu dostępu (*Web access type*) można wybrać następujące parametry:

1. ścieżkę dostępu – dla witryny umieszczonej w strukturze katalogów na dysku (opcja *Dynamic Files*),
2. nazwę źródła danych, użytkownika i hasło – dla witryny umieszczonej w bazie danych (opcja *Database*).

Niezależnie od tego, czy użyjemy opcji *Dynamic Files* czy *Database*, musimy przypisać do projektu witryny unikatowy adres *URL* (ang. *Uniform Resource Locator*).

Jeżeli generujemy projekt jako witrynę przechowywaną w postaci plików dynamicznych (opcja *Web Access Type* ustawiona na *Dynamic Files*), to w polu *Web Site* okna dialogowego *PowerDynamo Model Extended Attributes* należy wskazać prefiks adresu witryny po adresie serwera (rysunek 17). Musi to być ten sam prefiks, który został przypisany witrynie w narzędziu *PowerDynamo*. Prefiks ten konieczny jest do określenia pełnej ścieżki dostępu do takiej witryny.



Rys. 17. Okno dialogowe *PowerDynamo Model Extended Attributes*, zakładka *Web Profile*

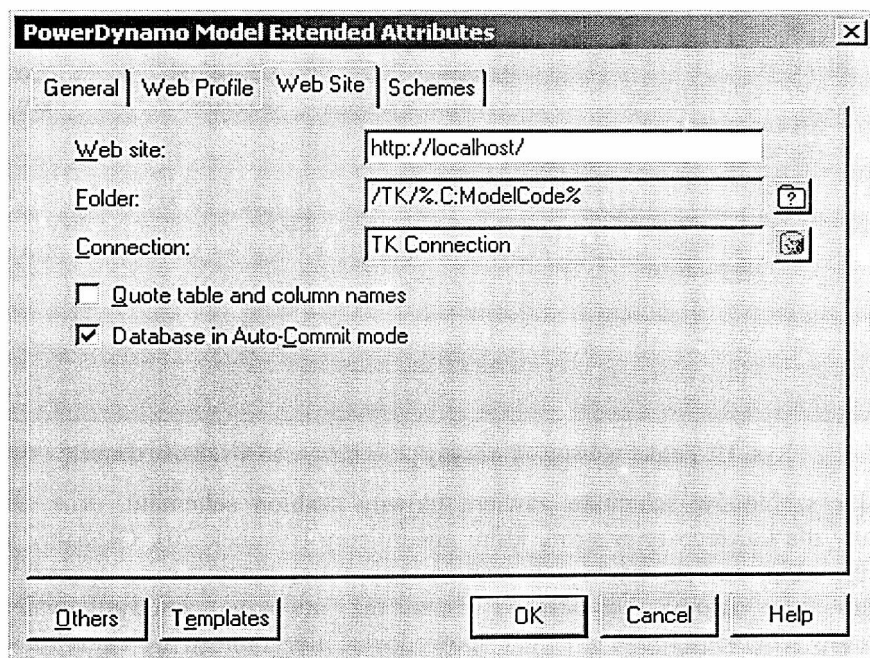
Dla witryny przechowywanej w bazie danych prefiks wskazuje główny folder z plikami w tej bazie. Jeżeli generuje się witrynę do bazy danych, to prefiks wpisany w polu *Folder* musi się rozpoczynać znakiem „/”. Prefiks ten jest automatycznie

pobierany, gdy *WebGen* łączy się z bazą danych w której przechowywana jest witryna. W serwerze *PowerDynamo* główny folder przechowywany w bazie danych posiada taką samą nazwę jak prefiks witryny.

Jeśli folder witryny *PowerDynamo* nie istnieje, to zostanie przez *WebGen* utworzony. Witryna musi też posiadać zdefiniowane w *PowerDynamo* połączenie ze źródłem danych.

Zależnie od typu witryny internetowej, *WebGen* generuje zapytania *SQL* z nazwami tabel i kolumn umieszczonymi w cudzysłowach albo nazwami bez cudzysłowów. Wyboru dokonuje się za pomocą przełącznika checkbox *Quote table and column names*. Opcja ta znajduje się w oknie dialogowym *PowerDynamo Model Extended Attributes*. Jeżeli użytkownik wybierze tę opcję, to *WebGen* w generowanych zapytaniach *SQL* wszystkie nazwy tabel i kolumn otoczy znakami cudzysłowów. Jeżeli jednak opcja ta nie zostanie wybrana, znaki cudzysłowów zostaną zastosowane do tych tabel i kolumn, które zawierają znaki specjalne (spacja, slash itd.).

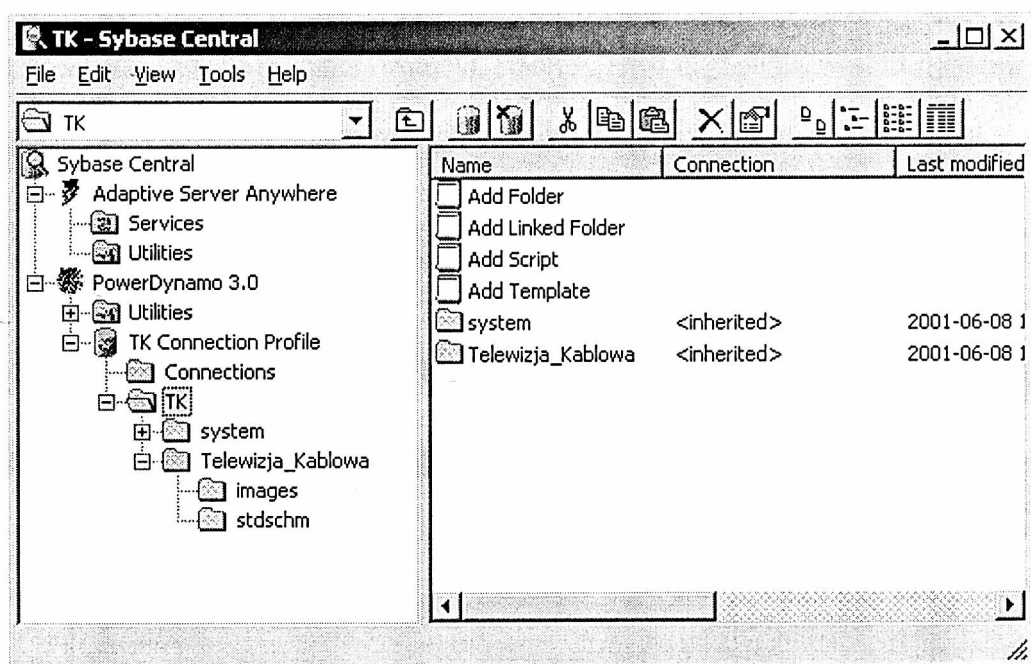
Jeżeli baza danych działa w trybie automatycznego potwierdzania, należy zaznaczyć opcję *Auto-Commit Mode* w zakładce *Web Site* okna dialogowego *PowerDynamo Model Extended Attributes* (rysunek 18).



Rys. 18. Okno dialogowe *PowerDynamo Model Extended Attributes* zakładka *Web Site*

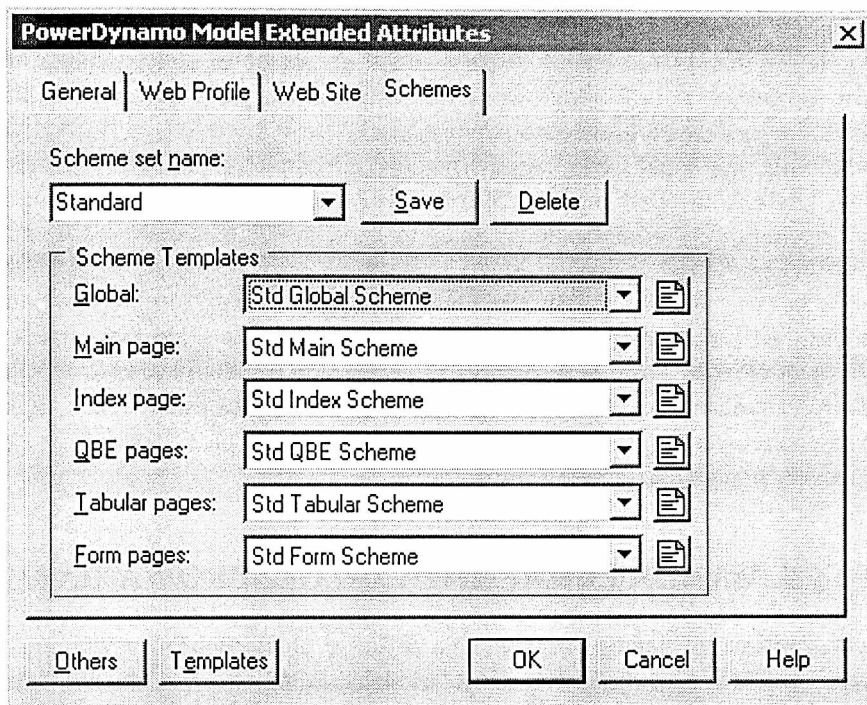
Folder witryny *PowerDynamo* można stworzyć albo wybrać w *WebGen*. Jeżeli wybierze się dynamiczny folder albo profil baz danych, które nie zawierają stron *PowerDynamo*, to *WebGen* zakomunikuje błąd. Domyślnym folderem witryny jest */Site/%.C:ModelCode%*. Site jest domyślną nazwą dla głównego folderu bazy danych. Jeżeli nie zmieni się wartości domyślnej, *WebGen* utworzy podkatalog katalogu głównego o nazwie modelu *PDM*.

Jeżeli generuje się witrynę w bazie danych, projekt zawsze posiada połączenie domyślne. Gdy wybieramy witrynę zapisaną jako pliki dynamiczne, połączenie domyślne nie będzie dostępne. W narzędziu *Sybase Central* (rysunek 19) można dodać połączenie do wielu źródeł danych, w tym ustawić profil połączenia dla *PowerDynamo*. Połączenie to musi zostać zdefiniowane zanim będzie dostępne w *WebGen*.



Rys. 19. Folder główny TK witryny przechowywanej w bazie danych

Zbiór szablonów schematu zawiera główny szablon schematu, oraz oddzielne szablony dla każdego typu stron, które generujemy (rysunek 20). Globalny schemat definiuje zbiór ramek, pasków narzędziowych i tła, które nie są specyficzne dla każdego typu strony. Użytkownik może też utworzyć zmodyfikowany zbiór szablonów schematu. Może to jednak spowodować niezgodność projektu i tła strony.



Rys. 20. Okno dialogowe PowerDynamo Model Extended Attributes, zakładka Schemes

7.2. DOSTRAJANIE APLIKACJI PRZED I PO WYGENEROWANIU

Przed pierwszym lub ponownym wygenerowaniem aplikacji można ją dostosować i przetestować z poziomu widoku drzewa okna *Generate PowerDynamo Web Site* generowania aplikacji dokonując:

- a) modyfikacji właściwości modelu,
- b) modyfikacji właściwości atrybutów rozszerzonych,
- c) wyboru kolumn do generowania,
- d) generowania odnośników hipertekstowych pomiędzy stronami opartymi na tabelach nadrzędnych i podrzędnych (ang. *parent* i *child tables*),
- e) otwarcia wygenerowanej aplikacji *WWW* w przeglądarce.

Możliwa jest modyfikacja właściwości wszystkich obiektów z widoku drzewa. Istnieje również możliwość modyfikowania z widoku drzewa właściwości aplikacji takich jak: nazwa, tytuł, szablon, biblioteka, profil bazy danych.

Po rozwinięciu wszystkich jednostek wyświetlanych w widoku drzewa, *WebGen* wylistuje kolumny do wygenerowania. W widoku drzewa można określić, dla których

kolumn tabeli chcemy wygenerować poszczególne strony. Wyboru dokonuje się za pomocą flagi generowania znajdującej się przed nazwą obiektu. Rozwijając drzewo obiektów można też wybrać generowane elementy na niższym poziomie, tzn. atrybuty kolumn znajdujące się na poszczególnych stronach.

Użytkownik ma możliwość wygenerowania odnośników pomiędzy stronami na podstawie związków tabel modelu. W tym celu musi zostać wygenerowana tabela rodzica i przynajmniej jedna tabela potomna.

Jeżeli wygeneruje się witrynę osadzoną w bazie danych, można ją przetestować z widoku drzewa. Należy tylko uruchomić witrynę z *WebGen*, oczywiście pod warunkiem, że uruchomiony jest serwer *PowerDynamo*. Program instalacyjny *AppModelera* określa, które przeglądarki są powiązane z plikami *HTML* w rejestrze. Dla generowanych witryn *WebGen* otwiera przeglądarkę domyślną. Jeżeli rejestr nie wskazuje preferowanej przeglądarki, to należy wyznaczyć przeglądarkę, za pomocą której chcemy przeglądać generowane witryny.

7.3. WYBÓR ELEMENTÓW DO GENEROWANIA

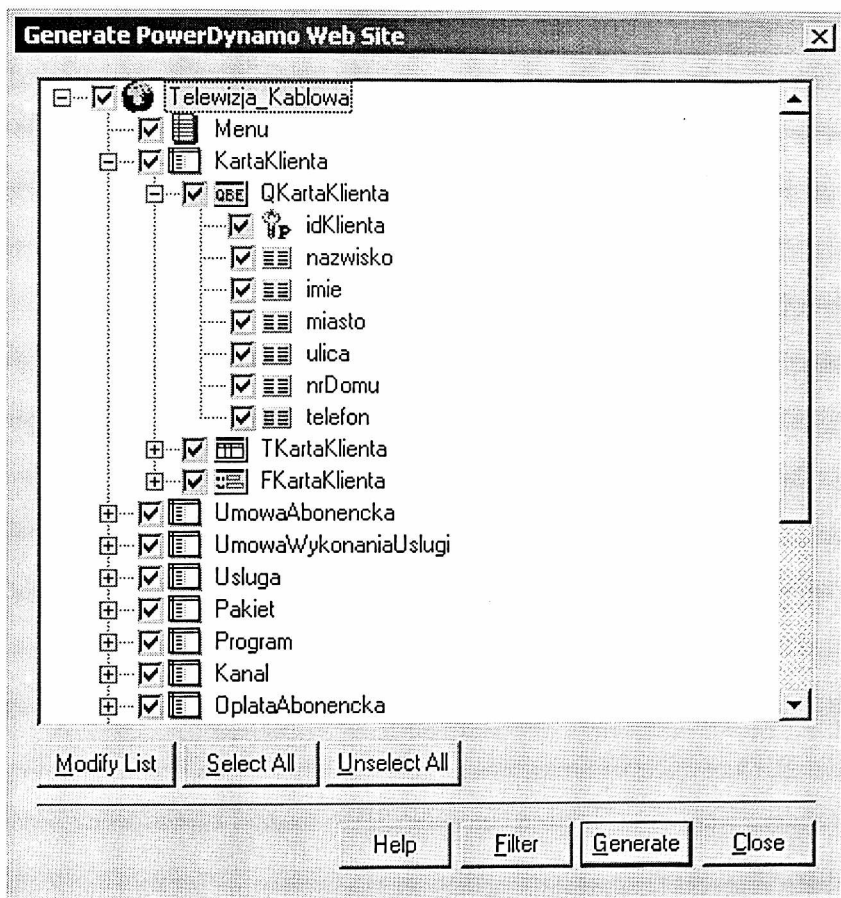
Użytkownik może generować aplikacje *WWW* z obiektów, które wybiera się na podstawie obiektów zdefiniowanych w modelu *PDM*. Widok drzewa w oknie *Generate PowerDynamo Web Site* (rysunek 21) pokazuje elementy, które mogą zostać wygenerowane. Drzewo zawiera takie elementy jak: obiekt strony głównej, obiekt strony indeksu, obiekty stron dla tabel i widoków, które wybiera się graficznie w modelu *PDM* lub obiekty stron dla tabel i widoków, które wybiera się za pomocą filtrów, pola obiektów oparte na kolumnach, związki pomiędzy tabelami rodzica i potomka.

Istnieją różne sposoby modyfikacji wstępnego wyboru obiektów w oknie *Generate PowerDynamo Web Site*. Aby zmodyfikować obiekt, możemy: ustawić flagę generowania obiektów w widoku drzewa, dodawać lub usuwać obiekty z widoku drzewa, zastosować filtr.

Ustawienie znaczników na obiektach w widoku drzewa wskazuje, że zostały one wybrane do generowania. Wyboru flag generowania wszystkich elementów najwyższego poziomu w widoku drzewa można dokonać przez naciśnięcie przycisku *Select All*. Użycie przycisku *Unselect All* usuwa wszystkie zaznaczenia. Jeśli nie generujemy pewnych jednostek najwyższego poziomu, nie możemy również zaznaczyć wyboru generowania jednostek od nich zależnych.

Filtry pomagają zapanować nad dużą ilością informacji. Model *PDM*, na przykład, może zawierać setki obiektów. Filtry służą do ograniczania liczby obiektów zawartych w widoku drzewa projektowego w oknie *Generate PowerDynamo Web Site*. Wybór ograniczony za pomocą filtra jest zapamiętywany przez *WebGen* niezależnie od tego, czy w bieżącej sesji wykonujemy generowanie aplikacji czy nie. Do wyboru mamy

różne typy filtrów: All objects (wybiera wszystkie obiekty z modelu *PDM*), Objects of submodel (wybiera obiekty nazwanego podmodelu), Modified objects (porównuje obiekty z bieżącego modelu *PDM* z obiektami z modelu zarchiwizowanego i wybiera obiekty zmienione), User-defined list (umożliwia wybór listy obiektów modelu *PDM*).

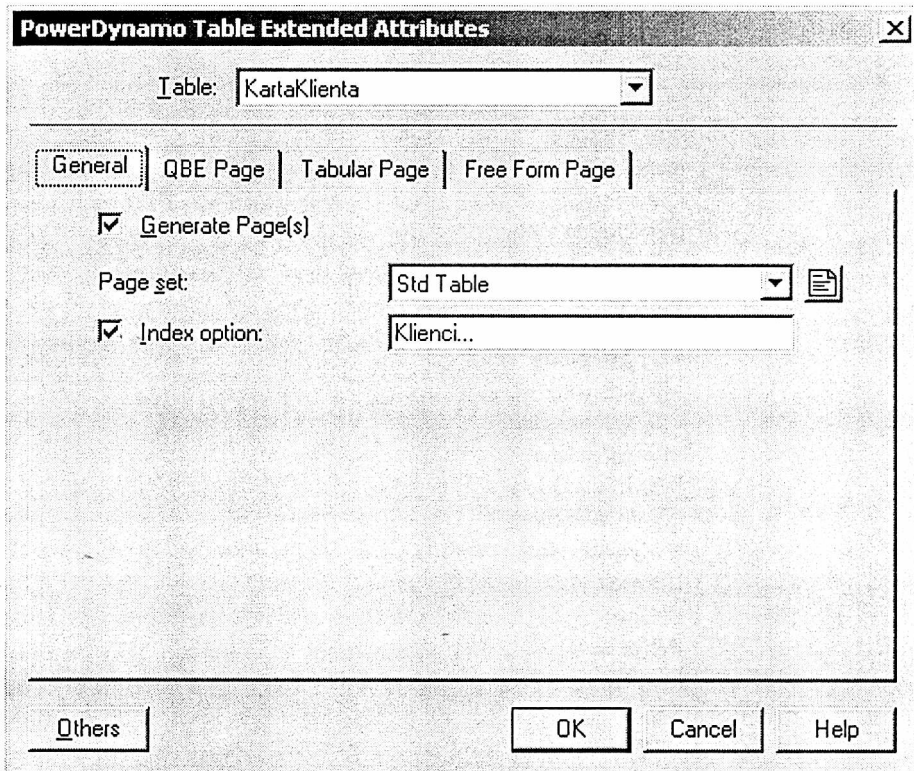


Rys. 21. Okno dialogowe Generate PowerDynamo Web Site

7.4. DEFINIOWANIE STRON I PÓŁ

Strony są głównym elementem interfejsu projektu witryny. Każda strona jest dokumentem *STM* zawierającym skrypt *HTML* wygenerowany przez *WebGen*. Dokumenty *STM* generuje się z poszczególnych tabel i widoków. Istnieje również możliwość generowania dokumentów *STM* jako plików.

Pola są elementami stron, które odpowiadają kolumnom bazy danych. Szablony pól są używane do definiowania kontrolki lub zbioru kontroltek, które potrafią wyświetlić dane z wierszy bazy danych. Przykładami kontroltek w aplikacjach są pola tekstowe oraz przyciski.



Rys. 22. Okno dialogowe PowerDynamo Table Extended Attributes, zakładka General

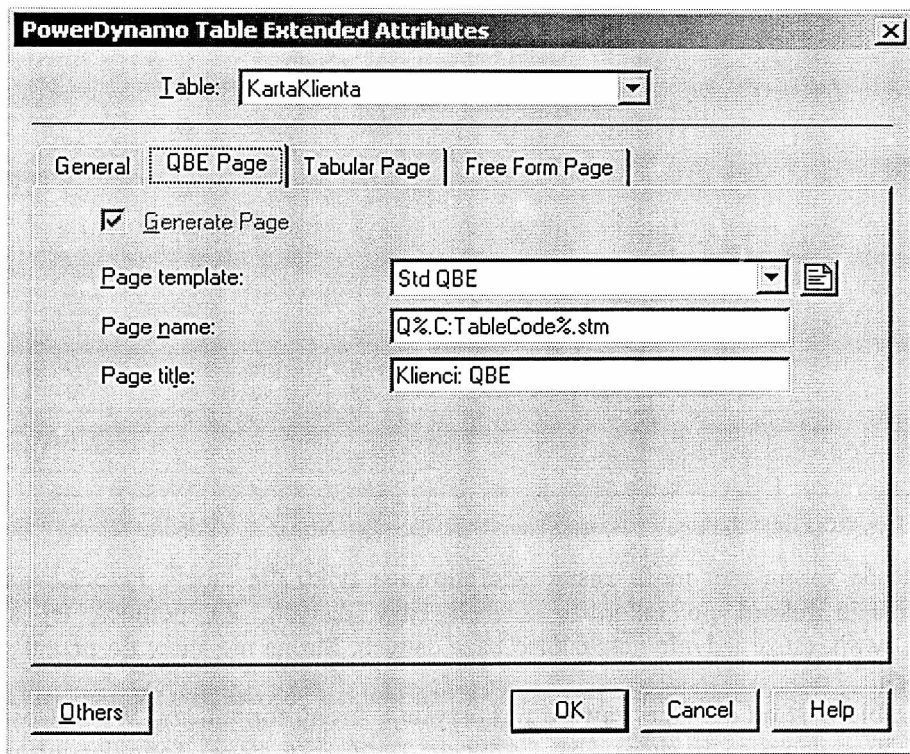
Dla każdej strony i widoku w modelu *PDM* można zdefiniować właściwości (rysunek 22):

- Generate page(s) - powoduje wygenerowanie strony HTML w dokumencie STM lub w pliku,
- Page set - nazwa szablonu, którego używa się do definiowania typów budowanych stron,
- Index option - powoduje wyświetlenie pozycji menu w ramce zawierającej indeks; można też określić nazwę jaka będzie wyświetlana, ale będzie ona wyświetlana tylko wtedy, gdy zaznaczona zostanie opcja Index option.

Dla każdej tabeli lub widoku, można używać standardowych właściwości strony, albo można zdefiniować specyficzne właściwości. Ponadto, dla każdej tabeli lub

widoku można wybrać szablon generujący z modelu fizycznego określony typ stron. Dla tabel *AppModeler* może generować strony typu *QBE*, strony tabelaryczne oraz strony-formularze. Dla widoków można generować strony typu *QBE* oraz strony tabelaryczne. Szablon stron determinuje ponadto kolejność wyświetlania stron generowanych z modelu *PDM*. Przypisuje on typ pierwszej strony, która otwiera się po wybraniu z menu odpowiedniego odnośnika.

Szablony stron dostarczone z *AppModelerem* definiują między innymi strony *QBE*. Po wybraniu aplikacji z menu oraz wygenerowaniu projektu jako pierwsze otwierają się strony *QBE*. Umożliwia to użytkownikowi wpisywanie wymaganych zapytań. Dla stron *QBE* definiuje się właściwości (rysunek 23): *Generate page* (zaznacza się dla wygenerowania strony; szablony stron określają, czy właściwość tę można modyfikować), *Page template* (nazwa szablonu *QBE*), *Page name* (nazwa pliku lub obiektu bazy danych, który zostanie wygenerowany obiekt), *Page title* (nagłówek, który pojawi się na górze strony).

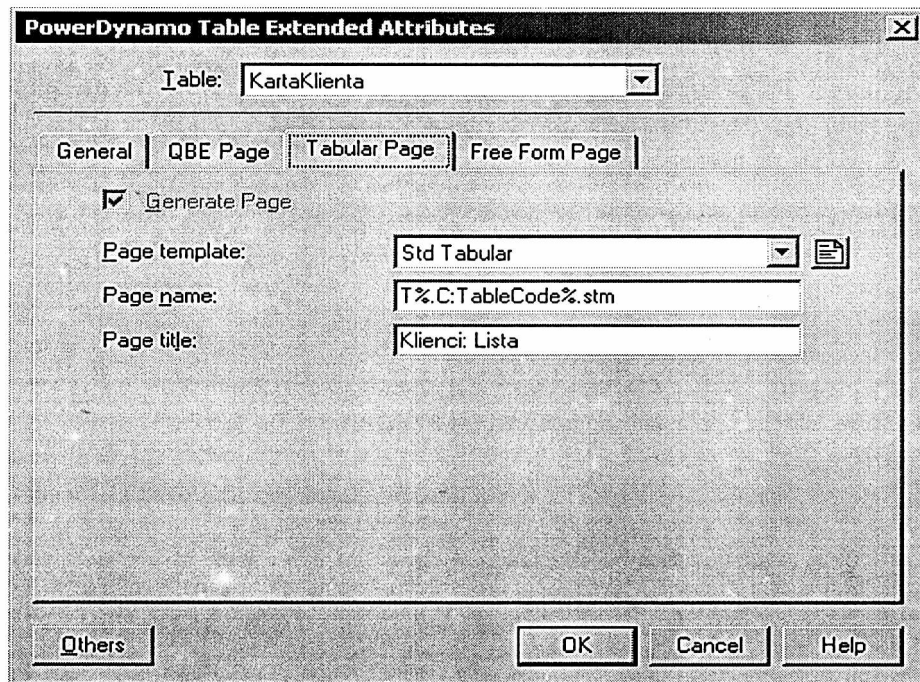


Rys. 23. Okno dialogowe PowerDynamo Table Extended Atributes, zakładka QBE Page

Strony tabelaryczne przedstawiają użytkownikowi informacje w formie tabeli. Jeśli użytkownik zatwierdzi przyciskiem Submit zapytanie *QBE*, serwer przetworzy zapytanie i zwróci dane w formie tabeli.

Strony tabelaryczne są zawsze generowane, zarówno dla szablonów stron tabel, jak i dla szablonów stron widoków. Jeżeli używa się tych szablonów, nie można wyłączyć opcji **Generate page**.

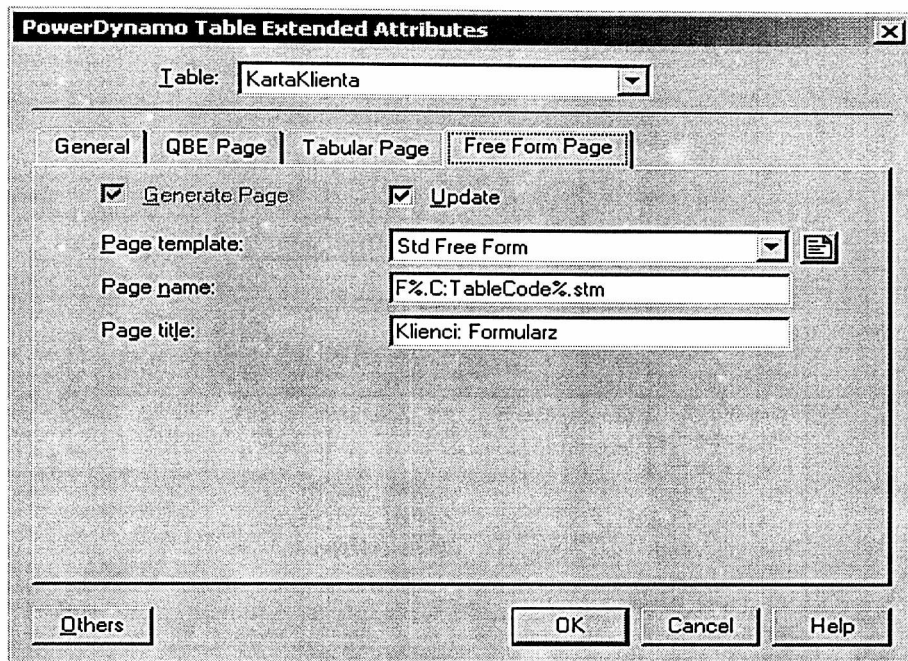
Strony tabelaryczne mogą mieć następujące właściwości (rysunek 24): **Generate page** (wskazuje, że strona tabelaryczna zostanie wygenerowana; szablon stron określa czy pole to może być modyfikowane), **Page template** (nazwa szablonu stron tabelarycznych), **Page name** (nazwa pliku lub bazy danych, w której zostanie wygenerowany obiekt), **Page title** (nagłówek, który pojawi się na początku strony).



Rys. 24. Okno dialogowe PowerDynamo Table Extended Attributes, zakładka Tabular Page

Strona formularza może zostać zdefiniowana tylko dla tabeli. Przedstawia ona informacje rekord po rekordzie z tabeli bazy danych. Za pomocą tej strony użytkownik może jedynie uaktualniać bazę danych. Strona nie służy do przeglądania danych.

Szablony stron dla tabel zawsze wykorzystują strony formularza. Właściwościami stron formularza są (rysunek 25): **Generate page** (czy strona formularza zostanie wygenerowana; szablony stron określają, czy właściwość tę można modyfikować), **Update** (zezwolenie wygenerowanej aplikacji na modyfikacje bazy danych), **Page template** (nazwa szablonu formularza), **Page name** (nazwa generowanego pliku lub obiektu bazy danych), **Page title** (nagłówek, który pojawia się na początku strony).



Rys. 25. Okno dialogowe PowerDynamo Table Extended Attributes, zakładka Free Form Page

7.5. STYLE I SZABLONY PÓŁ

Style i szablony pól opisują sposób formatowania danych na generowanych stronach (rysunek 26). Standardowym stylem pól jest pole edycji, natomiast TEXTBOX.SCT jest standardowym szablonem pól dla stylu pola edycji. Style pól opisują typy kontroltek, które generuje się w aplikacji. Jeżeli użytkownik zdefiniuje kontrolkę w polu szablonu, może jej używać pod warunkiem, że wspierana jest ona przez przeglądarkę internetową. Każdy szablon pola posiada listę atrybutów, których wartości można modyfikować.

Istnieje również możliwość dołączania stylów pól i szablonów pól do dziedzin. Są one automatycznie stosowane do wszystkich kolumn powiązanych z daną dziedziną. Ponadto mamy możliwość uaktualniania stylów i szablonów pól oraz innych atrybutów rozszerzonych kolumn powiązanych wcześniej z poszczególnymi dziedzinami.

PowerDynamo Column Extended Attributes

Table:

Column:

Bound Controls

Field style:

Field template: 

	Attribute	Value
→	DefaultValue	++
2	FRM_Label_Begin	*
3	FRM_Label_End	
4	FRM_Lbl_Align	%FRMLblAlign%
5	FRM_Lbl_BGColor	%FRMLblBGColor%
6	FRM_Lbl_VAlign	%FRMLblVAlign%
7	FRMSQLExpression	
8	QBE_Label_Begin	
9	QBE_Label_End	
10	QBE_Lbl_Align	%QBELblAlign%

Value: ...

Rys. 26. Okno dialogowe PowerDynamo Column Extended Attributes

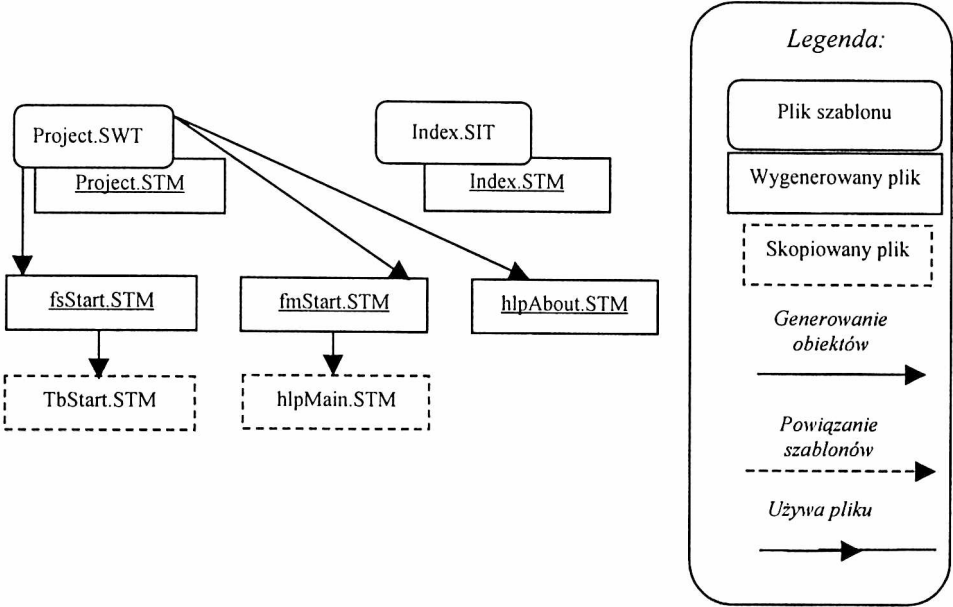
7.6. WYKORZYSTANIE SZABLONÓW

W narzędziu *AppModeler* używa się szablonów witryn, stron, obiektów modelu fizycznego i atrybutów rozszerzonych do wygenerowania witryny i stron. Pola szablonów definiują kontrolki, które zostaną wygenerowane na stronach *HTML*.

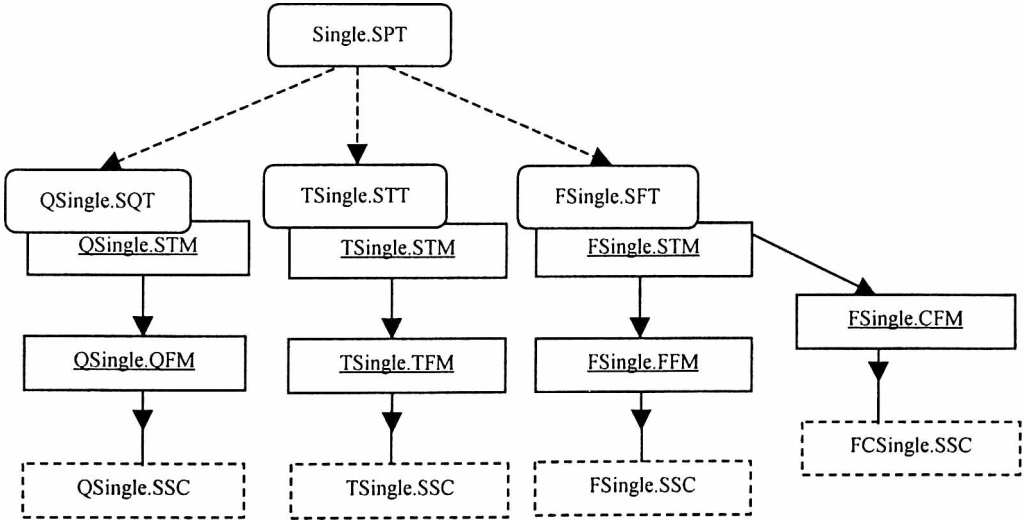
Zanim rozpocznie się generowanie witryny należy wybrać szablon projektu (plik o rozszerzeniu *.SWT*) oraz szablon indeksu (plik *.SIT*) (rysunek 27). Dla każdej tabeli należy również wybrać szablon strony *.SPT*. Szablon *SPT* musi posiadać trzy powiązane szablony. Odpowiadają one trzem typom generowanych stron – *QBE*, tabelarycznego i formularza. Ponadto musi być jeszcze dodany czwarty szablon dla tabel, które mogą być uaktualniane (rysunek 28).

Dla każdego widoku należy wybrać szablon *SPT* (rysunek 29). Posiada on dwa powiązane szablony, które odpowiadają stronom *QBE* i tabelarycznym. Strony

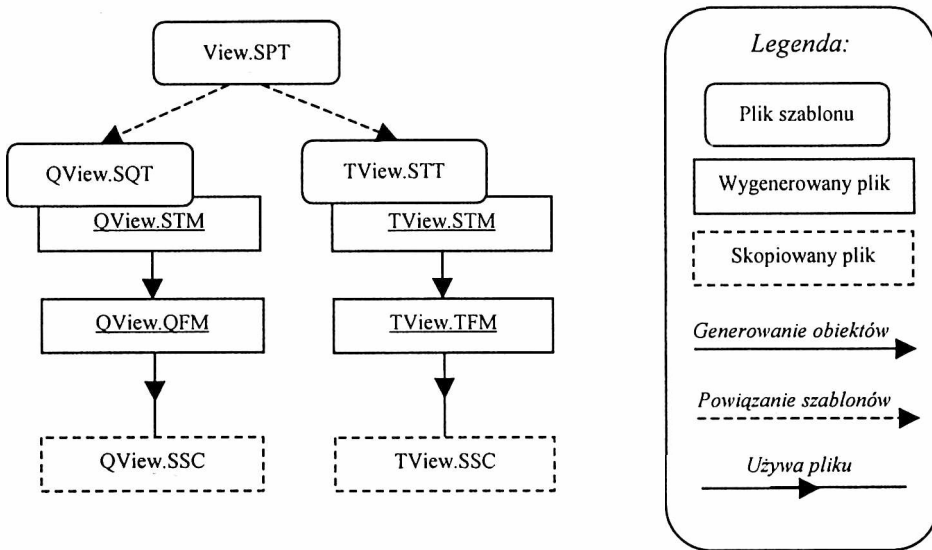
formularzy są niedostępne dla widoków i nie można uaktualniać bazy danych ze strony wygenerowanej dla widoku.



Rys. 27. Struktura szablonów na poziomie modelu (ang. *Model-level templates*)



Rys. 28. Szablony tabel



Rys. 29. Szablony widoków

Szablon strony głównej z rozszerzeniem .SWT generuje pliki STM dla strony głównej i umieszcza je w wybranym folderze lub w bazie danych. Szablon ten zapewnia powiązania z plikami zasobów, które *WebGen* kopiuje do wybranej lokalizacji. *WebGen* używa zmiennych %ProjectName% i %ProjectTitle% które odnoszą się do wartości wprowadzanych w polach Main Page Name i Main Page Title w oknie dialogowym PowerDynamo Model Extended Attributes.

Szablon strony indeksu o rozszerzeniu .SIT generuje strony z menu zawierającym odniesienia do stron, które wygenerowano dla tabel i widoków. Szablon INDEX.SIT zapewnia także powiązania z zasobami kopiowanymi przez *WebGen*, na przykład z bitmapą tła.

Szablon stron posiada rozszerzenie .SPT. Szablon ten zapewnia listę typów stron wygenerowanych dla tabel i widoków oraz definiuje typy stron które zostaną wygenerowane. Szablon stron SINGLE.SPT musi być używany w połączeniu z szablonami stron QSINGLE.SQT, TSINGLE.STT i FSINGLE.SFT. Szablon widoków VIEW.SPT jest używany w połączeniu z szablonami QVIEW.SQT i TVIEW.STT.

Szablony stron *QBE* są niezbędne do wygenerowania stron *QBE* dla tabel i widoków. Szablonem takim jest QSINGLE.SQT – szablon do wygenerowania stron *QBE* dla tabel oraz jego odpowiednik dla stron widoków QVIEW.SQT. Strona pomocy *QBE*, którą *WebGen* również umieszcza w generowanej witrynie, wyjaśnia użytkownikowi jak wykonywać standardowe operacje w zapytaniach oraz jak zadawać zapytania o wielu kryteriach wyszukiwania. Użytkownik może umieścić tam dodatkowe informacje edytując plik hlpQBE.STM w edytorze stron *HTML*.

Szablony stron tabelarycznych **TSINGLE.STT** i **TVIEW.STT** generują strony wyświetlające wiersze danych w postaci tabeli. Dane wyświetlane są na podstawie zapytania zadanego na odpowiedniej stronie QBE. Klucz główny w każdym wierszu danych jest odnośnikiem hipertekstowym. Otwiera on wybrany rekord w stronie formularza.

Szablon stron formularzy **FSINGLE.SFT** generuje strony, które potrafią wyświetlać wszystkie dane rekordu bazy danych. Jeżeli można uaktualniać bazę danych, to dostępna jest opcja wprowadzania rekordów za pomocą stron formularza.

Do zmiany tła wyświetlanego w projekcie witryny oraz do zmiany kształtu, koloru i innych własności charakteryzujących generowane obiekty, można używać schematu szablonów. Schematy szablonów dołączone do *AppModelera* opisane są w pliku **SCHEME.TXT** umieszczonym w podkatalogu **DYNTPL** (Dynamo Templates, gdy używane są długie nazwy).

Szablony pól definiują kontrolki wyświetlane na stronach *HTML*. Szablony pól zazwyczaj definiują jedną kontrolkę i jedną etykietę tekstową do identyfikacji danych wyświetlanych poprzez tą kontrolkę. Szablony pól mogą być dołączane do kolumn i dziedzin. Pewne typy pól w pełni nadają się do wyświetlania w nich danych określonych typów. Zalecane style pól dla poszczególnych typów danych zestawiono w tabeli 1.

Szablony opuszczanych pól kombinowanych (Dropdown combination box) **DBCOMBO.SCT** zastępują informacje znajdujące się w jednej kolumnie tabeli informacjami z kolumny innej tabeli. Naturalnie, tabele te muszą być powiązane. Informacje z powiązanej za pomocą klucza obcego tabeli są wyświetlane w rozwijalnym polu kombinowanym.

Tabela 1. Zalecane style pól dla wybranych typów danych

Typ danych*	Zalecany styl pola	Szablon
Boolean	Pole wyboru (Checkbox)	CHECKBOX.SCT lub CHKBOX2S.SCT
Counter	Etykieta (Label box)	LABELBOX.SCT
Text (ograniczony)	Pole tekstowe (Textbox)	TEXTBOX.SCT
Text (nieograniczony)	Wielowierszowe pole tekstowe (Multiline box)	MLTBOX.SCT

* Nazwa typu danych zależy od używanego systemu *DBMS*

Wyrażenia **Lookup** (Lookup expression) można przypisać do atrybutów, które zawierają wartości z różnych kolumn tabel powiązanych kluczem obcym. Wyrażenia muszą odpowiadać wyrażeniom *SQL* wspieranym przez *DBMS* na którym umieszczono źródło danych. Przykładowo, jeśli używa się *Sybase Adaptive Server Anywhere*, kody kolumn muszą być oddzielone znakiem „||”. Jeżeli natomiast używa się *MS Access*, to jako operatora konkatencji należy użyć znaku „+”.

Szablon pola RADIO.SCT definiuje kontrolki przycisków radiowych (Radio button). Szablon ten posiada atrybut RadioColumn, który wskazuje liczbę kolumn przycisków radiowych. Atrybut odnoszący się do liczby kolumn ma zastosowanie tylko w odniesieniu do stron formularza.

Tabela *QBE* generowana razem z szablonem RADIO.SCT wyświetla stałą pole listy zamiast przycisków radiowych. Zmiana atrybutu SetListSize powoduje zmianę rozmiaru pola listy wygenerowanej na stronie QBE. Użytkownik definiuje etykiety przycisków radiowych w oknie dialogowym Check Parameters w polu List of Values dla kolumn i dziedzin (rysunek 30).

Check Parameters

Column: statusUA (Status Umowy)
 Data type: char(1) Length: 1 Precision:
 Constraint name: CKC_STATUSUA_UMOWAABO ☐ User-defined

Standard Parameters | Validation Rules

Values
 Minimum:
 Maximum:
 Default: A

Parameters
 Unit:
 Format:
☐ Lowercase ☒ Uppercase
☐ Cannot modify

List of values

	Value	Label
→	A	Aktywna
2	Z	Zawieszona
3	R	Rozwiązana

Insert Delete

Expression: ☐ Client ☒ Server
 %MINMAX% and
 %LISTVAL% and
 %RULES%

Rules Extended Zoom Preview OK Cancel Help

Rys. 30. Okno dialogowe Check Parameters, zawierające listę wartości atrybutu statusUA

Szablon CHKLIST.SCT generuje standardowe kontrolki typu lista rozwijana (Listbox). Aby używać tego szablonu należy wpisać wartości i etykiety w oknie

dialogowym Check Parameters w polu List of Values. Etykiety, które się tam wpisuje, stają się wartościami widocznymi na liście.

Szablon typu checkbox generuje standardowe dwustanowe pola typu check na stronach formularzy. Użytkownik podaje wartości które są zapisywane w bazie danych poprzez zaznaczenie lub odznaczenie pola.

Pola generowane za pomocą szablonów checkbox pojawiają się na stronie *QBE* jako trzy przyciski radiowe. Pierwszy z nich ma etykietę Any. Etykiety pozostałych dwóch można zmodyfikować zmieniając standardowe wartości Onlabel i OffLabel.

Jeżeli używa się szablonu HTTPIMG.SCT, to do formularza można dodawać obrazki. Obrazek można obejrzeć na stronie tabelarycznej generowanej witryny jeżeli ustawi się wartość „ShowImageInTabular na True.

Użytkownik przypisuje szablon obrazka do kolumny z tekstowym typem danych. Tekst kolumny musi wskazywać adres obrazka. Adres w polu tekstowym musi posiadać składnię opisaną w tabeli 2.

Aby do witryny internetowej dodać dźwięk, należy użyć szablonu EMBEDDED.SCT. Szablon ten przypisuje się poprzez kolumnę o tekstowym typie danych. Kolumna ta musi wskazywać adres pliku, który zapisany jest w takiej samej składni jak ta pokazana w tabeli dla szablonu obrazków.

Za pomocą obiektów osadzonych można umieszczać na stronach formularzy wszystkie obiekty, które są wspierane przez przeglądarki stron *HTML*. Przykładowo, mogą to być kontrolki *ActiveX*.

Tabela 2. Przykładowe adresy dla różnych lokalizacji pliku obrazka

Lokalizacja obrazka	Składnia	Przykład
Internet	http://nazwa_serwera/ ścieżka/nazwa_pliku	http://localhost/pwrs/sybase.gif lub http://www.powersoft.com/gifs/ sybase.gif
Lokalny, kopiowany do serwera	/ścieżka/nazwa_pliku	/Site/Sybase.gif lub Sybase.gif
Plik lokalny	file://nazwa_maszyny /ścieżka/nazwa_pliku* lub file://napęd:/ścieżka /nazwa_pliku**	file://chicago/temporary/xyz.gif lub file://c:/temporary/xyz.gif

* Dostępność jest ograniczona do użytkowników mających dostęp do sieci lokalnej.

** Składnia jest przeznaczona do użytku na maszynach kompatybilnych z PC.

Ogólnie dostępność jest ograniczona do użytkowników posiadających dostęp do nazwanych napędów.

Opis szablonów i inne informacje o szablonach można uzyskać bezpośrednio w *WebGen*. Typ dostępnych informacji zależy od typu wybranego szablonu. Informacje o szablonach przedstawiono w tabeli 3.

Tabela 3. Informacje o szablonach

Typ	Dostępność w <i>WebGen</i>	Widoczne sekcje
Main Page	Model Extended Attributes	Description Generated Objects
Index	Model Extended Attributes	Description Menu Topic
Table Page Set	Table Extended Attributes	Description
View Page Set	View Extended Attribute	Description
QBE Page	Table Extended Attributes View Extended Attributes	Description Attributes GeneratedObjects
Tabular Page	Table Extended Attributes View Extended Attributes	Description Attributes GeneratedObjects
Free Form Page	Table Extended Attributes	Description Attributes GeneratedObjects
Field	Column Extended Attributes Domain Extended Attributes	Description Attributes
Computed Fields	View Extended	AttributesDescription

Informacje w szablonach możemy modyfikować za pomocą edytora tekstowego. Jeśli zmodyfikujemy informacje widoczne za pomocą *WebGen* w jakiejś sekcji szablonu, to uwidocznia się one w oknie dialogowym Show Template.

Za pomocą *WebGen* można wykreować własne szablony projektów i szablony stron. W tym celu mogą być wykorzystane istniejące już projekty witryn. Informacje o ścieżkach szablonów zostaną wpisane do rejestru bezpośrednio w *WebGen*.

Modyfikacji szablonów w *WebGen* dokonujemy za pomocą edytora tekstowego. Zmieniamy nazwy pól i identyfikatorów systemowych lub zmiennych definiowanych przez użytkownika.

8. PODSUMOWANIE

Celem niniejszego artykułu było przedstawienie możliwości projektowania, modelowania i generowania baz danych za pomocą pakietów *Sybase PowerDesigner* i *Sybase PowerBuilder* na przykładzie konkretnego zadania projektowego. Omówiono składowe *ProcessAnalyst*, *DataArchitect* i *AppModeler*, które wspomagają projektowanie i generowanie trzywarstwowych aplikacji internetowych. Autorzy planują dalsze badania nad narzędziami firmy *Sybase*.

LITERATURA

- [GruMazSta2004] Gruber J., Mazur Z., Standzoń M., *System projektowy Sybase PowerDesigner DataArchitect Suite 6.1 oraz środowisko Sybase PowerBuilder 7 Enterprise do modelowania i generowania baz danych*, Prace Naukowe Wydziałowego Zakładu Informatyki, nr 5, Bazy danych, Oficyna Wydawnicza Politechniki Wrocławskiej, Wrocław, 2004.

- [Powersoft] Pliki pomocy oraz dokumentacja firmowa w postaci elektronicznej dołączona do następujących pakietów oprogramowania firmy Powersoft: InfoMaker for Windows, PowerDesigner, PowerBuilder Enterprise, Power++ Enterprise.
- [Powersoft2] Pliki pomocników oraz dokumentacja firmowa w postaci elektronicznej dołączona do następujących pakietów oprogramowania firmy Powersoft (Sybase): PowerDesigner, PowerBuilder Enterprise, Software 2.0, 4 (2000) wraz z dołączonym pakietem Sybase Pack. Źródła internetowe i dokumentacja firmowa.

POWERDESIGNER TOOL FOR MODELLING AND BUILDING DATABASE WEB APPLICATIONS

The paper presents some of the typical problems found in software engineering, that can be solved by an approach based on middleware technologies, giving two examples of the technology.

*modele baz danych,
semantyczne bazy danych,
reprezentacja wiedzy*

Marek BURDA
Dariusz KRÓL*

KONCEPCJA MODELU SEMANTYCZNEJ BAZY DANYCH WYKORZYSTUJĄCEJ SKOJARZENIA

W pracy przedstawiono koncepcję semantycznej bazy danych. Zaproponowano model reprezentacji wiedzy oparty o definicyjne sieci semantyczne. Dla prezentowanego modelu opisano zasadę budowania konstrukcji pojęciowych zgodnych z modelem, zaproponowano ich formalną, graficzną notację. Następnie opisano sposoby implementacji najważniejszych jego elementów. Na koniec pokazano możliwości praktycznego zastosowania przedstawionej struktury wiedzy i zagadnienia istotne z punktu widzenia dalszego rozwoju.

1. WPROWADZENIE

Jednym z problemów komputerowego przetwarzania informacji jest to, że komputer nie rozumie treści, którą przetwarza. Nie wystarczy odpowiednio przechowywać informację w bazie danych. Potrzebujemy ją rozumieć, a do tego wymagana jest wiedza. Aby poradzić sobie z tym problemem, wykorzystamy strukturę pojęciową umożliwiającą reprezentację wiedzy niesformalizowanej. Takie właśnie zadanie spełniają sieci semantyczne.

Sieci semantyczne to z założenia reprezentacja treści [7, 9]. Oznacza to, że idealna sieć semantyczna ma dla danego stanu opisywanego fragmentu rzeczywistości tylko jedną postać, w odróżnieniu od języka naturalnego, którego cechuje mnogość form (np. wyrażenia równoważne to: „błękitne niebo”, „niebo w kolorze błękitu” i „niebo o kolorze błękitu”). Sprowadzenie wiedzy do postaci sieci semantycznej samo w sobie stanowi pewną formę normalizacji, jednak z punktu widzenia automatycznego wnioskowania jest to wciąż rozwiązanie dalekie od doskonałości. Silna normalizacja stwarza zagrożenie utraty jej uniwersalności. Pogodzenie zadań silnej normalizacji i zachowania uniwersalności w reprezentacji wiedzy jest głównym problemem postawionym przez autorów pracy.

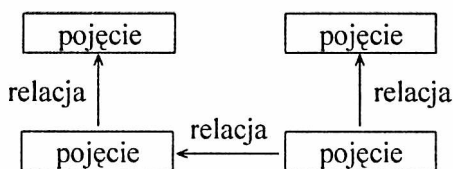
W pracy opisano sieci semantyczne (rozdział 2). Dalej umieszczony został opis zaprojektowanego systemu reprezentacji wiedzy ICKR (rozdział 3) oraz opis

* Politechnika Wrocławska, Wydział Informatyki i Zarządzania, Instytut Informatyki Stosowanej,
50-370 Wrocław, Wyb. Wyspiańskiego 27, Dariusz.Krol@pwr.wroc.pl

reprezentacji graficznej ICSN (rozdział 4). Następnie przedstawiono propozycję sposobu implementacji integracji proponowanego modelu z relacyjną bazą danych (rozdział 5). W rozdziale 6. znajduje się podsumowanie wyników pracy.

2. SIECI SEMANTYCZNE

Sowa [7] definiuje sieci semantyczne jako notację reprezentacji wiedzy połączonych węzłów i łuków. Hartley i Barnden [2] twierdzą zaś, że sieci semantyczne to sposób myślenia o wiedzy jako o sieci pojęć połączonych relacjami, a sam diagram to jedynie reprezentacja sieci semantycznej. Według Wikipedii [9] sieci semantyczne to forma reprezentacji wiedzy, będąca grafem skierowanym składającym się z węzłów reprezentujących pojęcia i krawędzi reprezentujących relacje znaczeniowe pomiędzy pojęciami (rysunek 1).



Rys. 1. Sieć semantyczna

W pracy definiujemy sieć semantyczną jako zbiór informacji przedstawiony w spójnej sieci pojęć wzajemnie połączonych skierowanymi relacjami o zdefiniowanych typach. W związku z tym tożsamość obiektu w sieci semantycznej określona jest na dwa sposoby: poprzez identyfikator oraz poprzez jego umiejscowienie w sieci pojęć. Znajomość tożsamości (znaczenia) obiektu pozwala na uzyskanie wszystkich danych jego dotyczących (kompletność) oraz uniknięcie niejednoznaczności podczas używania (precyzja).

Przykład 1. *Klucz jest bezużytecznym obiektem dopóki nie pozna się jego tożsamości. Gdy zidentyfikuje się klucz jako narzędzie otwierające drzwi, może posłużyć do otwarcia drzwi (i tylko drzwi). Można także zidentyfikować klucz jako otwieracz butelek, co pozwoli otwierać nim dodatkowo butelki.*

Sieci semantyczne znajdują zastosowanie w następujących dziedzinach:

- psychologia (objaśniają procesy zachodzące wewnątrz umysłu człowieka),
- filozofia (pozwalają dociec natury pojęć poprzez poznanie ich tożsamości),
- językoznawstwo (są pomocne przy badaniu znaczenia słów względem innych słów oraz przy analizie semantycznej wypowiedzi),
- bazy wiedzy (pozwalają nadać wiedzy strukturę, która później ułatwia wyszukiwanie informacji),
- systemy ekspertowe (nadana przez nie struktura pozwala uruchomić efektywne algorytmy wnioskujące),

- modelowanie i symulacje (dają możliwość budowy modelu wycinka rzeczywistości i uruchomienia symulacji procesów w nim zachodzących).

Celem niniejszej pracy jest zaproponowanie i zbadanie uniwersalnej reprezentacji wiedzy opartej o sieć semantyczną, w której położono nacisk na silną tożsamość obiektów oraz grupowanie obiektów bliskich znaczeniowo. Do minimum zostanie ograniczona liczba rodzajów relacji, co pozwoli rozproszone pojęcia pomiędzy węzłami i relacjami skupić w węzłach, wzmacniając tym samym ich tożsamość. Ograniczona liczba rodzajów relacji spowoduje większą jednorodność sieci. Precyzyjnie określone znaczenia i własności relacji umożliwią w przyszłości efektywne wnioskowanie.

3. REPREZENTACJA WIEDZY SKOJARZENIOWEJ – ICKR

Proponowana reprezentacja wiedzy została nazwana reprezentacją wiedzy IC (ICKR – IC Knowledge Representation). Nazwa pochodzi od dwóch podstawowych występujących w niej relacji: „jest” (Is) oraz „dotyczy” (Concerns). Jest to jednocześnie gra słów. IC jest w języku angielskim skrótem od „I see” czyli „rozumiem”.

ICKR to reprezentacja wiedzy, w której:

- (a) wszystkie pojęcia są wyrażone rzeczownikami,
- (b) pojęcia połączone są dwoma podstawowymi rodzajami relacji: „jest” oraz „dotyczy”,
- (c) zbiory relacji mogą być grupowane w sytuacje, odpowiada to istnieniu dodatkowej relacji „trzyma”.

Normalizacja uzyskana dzięki ograniczeniu liczby pojęć (a) i relacji (b) do niezbędnego minimum zmniejsza liczbę równorzędnych form dla pojedynczego znaczenia, a więc zbliża proponowaną reprezentację do ideału przedstawiającego jedynie znaczenie (bez możliwości wyboru formy). Sytuacje (c) służą tworzeniu reguł, które umożliwiają opisywanie równoważności form. Za pomocą ICKR można opisywać powiązania semantyczne (relacje „jest” i „dotyczy”), opisywać fakty (relacja „jest”) oraz opisywać reguły (sytuacje).

W ICKR pojęcia są wyrażone jedynie przez rzeczowniki. Oznacza to, że przekształcając zdanie z języka naturalnego, inne części mowy muszą zostać zastąpione przez odpowiednią konstrukcję pojęć będących rzeczownikami oraz dozwolonych relacji.

Przykład 2. Na potrzeby ICKR zdanie „Ściana jest biała” należy przekształcić do zdania „Biel jest kolorem ściany”.

Można powiedzieć, że jeśli A jest B, to A posiada cechy B albo A dziedziczy cechy B. Relacja „jest” jest poprowadzona albo od pojęcia bardziej szczegółowego do pojęcia bardziej ogólnego, albo pomiędzy pojęciami równoważnymi, co odpowiada to

sytuacji, w której A jest B i B jest A. W ICKR istnieje dodatkowa relacja na oznaczenie takiej sytuacji: „jest tożsamy z”.

Przykład 3. *Kanarek jest ptakiem. Numer telefonu jest numerem. Ziemniak jest tożsamy z kartoflem.*

Relacja „A dotyczy B” oznacza, że wszystko, czym jest A, definiowane jest w odniesieniu do B. Można także powiedzieć, że B jest zakresem A.

Przykład 4. *Numer telefonu jest numerem i dotyczy telefonu.*

W ICKR najprościej wyrazić fakty, które w języku naturalnym złożone są z rzeczowników oraz czasownika „jest”. Ich konstrukcja nie ulega wówczas zmianie.

Przykład 5. *Jan jest śpiewakiem. Piotr jest ojcem Marka.*

Aby pojęcie wyrażane przez czasownik mogło znaleźć się w ICKR, czasownik musi zostać przekształcony do rzeczownika, a następnie połączony z podmiotem pojęciem definiującym odpowiednią relację.

Przykład 6. *„Robert biegnie” zostanie zastąpione przez „Robert jest wykonawcą biegu”. „Ala ma kota” zostanie zastąpione przez „Ala jest posiadaczem kota”.*

Pojęcia opisujące czynności mogą być przedmiotem bardziej szczegółowego opisu. Aby wyrazić, że dana czynność miała miejsce w przeszłości lub będzie miała miejsce w przyszłości, należy użyć konstrukcji odpowiednio „jest elementem przeszłości” albo „jest elementem przyszłości”. W podobny sposób można wyrazić tryb czynności. Wyrażenie rozkazu wymaga użycia konstrukcji „jest przedmiotem rozkazu” lub „jest przedmiotem polecenia”. Dla odzwierciedlenia trybu przypuszczającego używana jest konstrukcja „jest przedmiotem przypuszczenia”.

Przykład 7. *„Ala miała kota” można wyrazić przez „Ala jest uczestnikiem posiadania kota. To posiadanie jest elementem przeszłości.”*

Przymiotnik, podobnie jak prawie wszystkie słowa nie będące rzeczownikami, należy zamienić na rzeczownikowy odpowiednik przed jego użyciem w ICKR. W ten sposób staje się on pojęciem opisującym (cechą), a rzeczownik oryginalnie znajdujący się w zdaniu – pojęciem opisywanym.

Przykład 8. *„Ten samochód jest zielony” zostanie zastąpione przez „Zielony jest kolorem tego samochodu”. „Piotr jest agresywny” zostanie zastąpione przez „Agresja jest cechą Piotra”.*

„Trzyma” to relacja między pojęciem będącym sytuacją, a każdą z relacji wchodzących w skład sytuacji. „A trzyma B” oznacza, że B jest relacją wchodzącą w skład sytuacji A. Relacja „trzyma” służy grupowaniu relacji w sytuacji. Sytuacje są pojęciami, w związku z czym można się do nich odwoływać jak do innych pojęć. Sytuacja połączona relacją z innym pojęciem tworzy regułę.

Przykład 9. *„Jeśli A jest synem B, to B jest rodzicem A” można wyrazić tak: „C trzyma: A jest synem B. D trzyma: B jest rodzicem A. C jest D”. „C jest D” można tu*

tłumaczyć jako „C oznacza D”. Nieprawda, że „C jest tożsame z D”, gdyż niekoniecznie „D jest C” (A może być córką B).

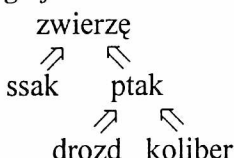
4. REPREZENTACJA GRAFICZNA – ICSN

Terminem ICKR oznaczamy sposób myślenia o wiedzy. Jego formalną, graficzną reprezentacją jest sieć semantyczna IC oznaczona ICSN (IC Semantic Network). ICSN to graf skierowany, w którym:

- nazwane pojęcie oznaczone jest napisem z nazwą pojęcia;
- nienazwane pojęcie oznaczone jest kropką;
- relacja „jest” oznaczona jest podwójną strzałką, $A \Rightarrow B$ oznacza, że A jest B;
- relacja „jest tożsame z” oznaczona jest podwójną obustronną strzałką, $A \Leftrightarrow B$ oznacza, że A jest tożsame z B;
- relacja „dotyczy” oznaczona jest pojedynczą strzałką, $A \rightarrow B$ oznacza, że A dotyczy B;
- grupowanie relacji w sytuacji odbywa się poprzez otoczenie grupowanych relacji zamkniętą figurą (najczęściej prostokątem), relacja wchodzi w obręb sytuacji, jeśli w całości zawiera się w figurze sytuacji;
- negację relacji otrzymuje się poprzez skreślenie symbolu relacji (umieszczenie krzyżyka $\times$ na symbolu relacji), usunięcie przynależności relacji do sytuacji polega na umieszczeniu małego krzyżyka $\times$ na jednym z boków figury sytuacji;
- przedmiot pytania oznaczony jest znakiem zapytania, znak zapytania może wystąpić w miejscu pojęcia (pytanie „Kto/co ...?”) lub obok sytuacji (pytanie „Czy ...?”).

W ICSN dąży się do tego, żeby nazwy pojęć prostych były pojedynczymi słowami (za wyjątkiem nazw własnych). Jeśli pojęcie składa się z kilku słów, należy utworzyć z niego pojęcie złożone. Pary nazwanych pojęć prostych łączy się zwykle relacją „jest” lub „jest tożsame z”. Służy to odzwierciedleniu odpowiednio hierarchii pojęć i ich synonimii.

Przykład 10. Zdania „Drozd oraz koliber są ptakami. Ptak i ssak są zwierzętami” są reprezentowane przez następujący graf:



Pojęć prostych nienazwanych używa się zazwyczaj, aby przedstawić obiekt danej klasy.

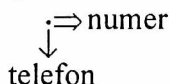
Przykład 11. Określenie „Pewien mężczyzna” jest symbolizowane przez kropkę.

mężczyzna

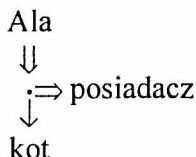


W ICSN dąży się do tego, aby przedstawiać pojęcia złożone za pomocą pojęć nienazwanych. Pojęcie nienazwane nie ma znaczenia w izolacji. Jego znaczenie definiują połączenia z innymi pojęciami.

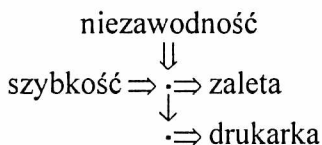
Przykład 12. Niech pojęcie „numeru telefonu” będzie pojęciem nienazwanym X . Aby wyrazić X w ICSN należy zdefiniować znaczenie X , a więc połączyć X z pojęciami „numer” i „telefon”. Potrzebne są dwie relacje „ X jest numerem” oraz „ X dotyczy telefonu”. W ICSN wygląda to tak (kropka w środku symbolizuje X):



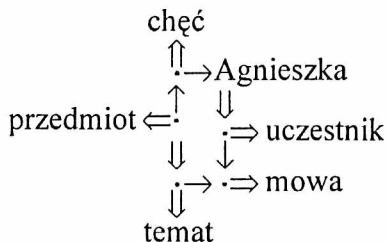
Przykład 13. Zdanie „Ala ma kota” w ICKR ma postać „Ala jest posiadaczem kota”. ICSN tego zdania wygląda tak:



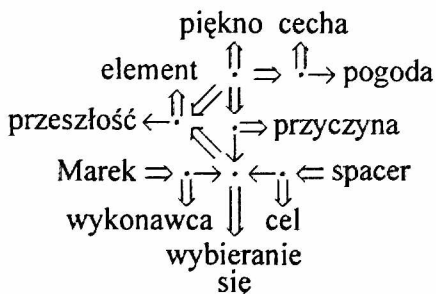
Przykład 14. Zdanie „Zaletami tej drukarki są szybkość i niezawodność” ma odwrócony szyk (szybkość jest zaletą, zaleta nie jest szybkością), więc bardziej odpowiednia do ICKR postać to „Szybkość i niezawodność są zaletami tej drukarki”. Usunięcie spójnika powoduje powstanie dwóch faktów ICKR „Szybkość jest zaletą tej drukarki. Niezawodność jest zaletą tej drukarki”. Całość w ICSN ma następującą postać:



Przykład 15. Zdanie „Agnieszka mówi, czego chce” ma w ICKR postać „Agnieszka jest uczestnikiem mowy. Przedmiot chęci Agnieszki jest tematem tej mowy”. W ICSN wygląda to następująco:

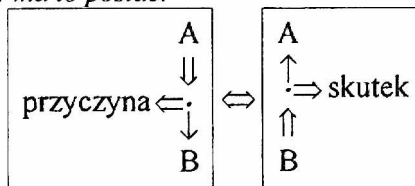


Przykład 16. „Była piękna pogoda, więc Marek wybrał się na spacer.” ma następującą postać w ICKR: „Piękno jest cechą pogody. Marek jest wykonawcą wybierania się. Spacer jest celem tego wybierania się. To piękno jest przyczyną tego wybierania się. To piękno jest elementem przeszłości. To wybieranie się jest elementem przeszłości”. Odpowiednik w ICSN wygląda następująco:

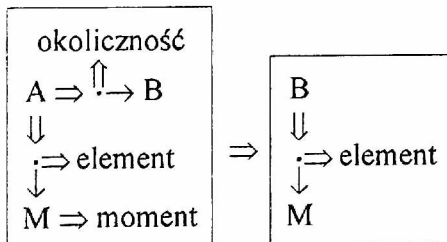


Zbiory relacji mogą być grupowane w sytuacje. Do sytuacji można odnosić się tak jak do innych pojęć, tj. łączyć relacjami „jest” i „dotyczy” z innymi pojęciami. Granice dwóch sytuacji mogą przecinać lub zawierać jedno w drugim. Podstawowym zadaniem sytuacji jest tworzenie reguł, tj. uwidacznianie zależności semantycznych pomiędzy różnymi konstrukcjami.

Przykład 17. „A jest przyczyną B wtedy i tylko wtedy, gdy B jest skutkiem A” wyrażone w ICKR to: „C trzyma: A jest przyczyną B. D trzyma: B jest skutkiem A. C jest tożsame z D”. W ICSN ma to postać:



Aby zdefiniować termin „okoliczność” (w ujęciu czasowym) można utworzyć następującą regułę ICKR: „C trzyma: (A jest okolicznością B. A jest elementem M. M jest momentem). D trzyma: B jest elementem M. C jest D”. W ICSN wygląda to następująco:



Jeśli w ICSN znajduje się choć jeden znak zapytania, cała sieć jest pytaniem. To pytanie może zostać porównane z inną siecią – adresatem pytania. Przedmiotem pytania może być zarówno prawdziwość danej sytuacji (odpowieź „prawda” lub

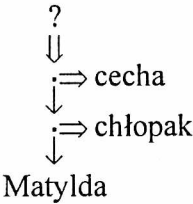
„fałsz”), jak i tożsamość pojęcia posiadającego określone właściwości. Jeśli przedmiotem pytania jest wiele obiektów wskazane jest nazwanie każdego ze znaków zapytania.

Przykład 18. Pytanie „Czy jaszczurka jest gadem?” w ICSN ma postać:

jaszczurka ⇒ gad

?

Przykład 19. Pytanie „Jaki jest chłopak Matyldy?” w ICSN wygląda następująco:



5. PROPOZYCJA IMPLEMENTACJI MODELU

Relacyjne bazy danych są najbardziej rozpowszechnionym, sprawdzonym oraz zoptymalizowanym sposobem przechowywania danych o ustalonej strukturze. Ponieważ struktura ICSN jest ustalona, naturalnym sposobem implementacji ich składowania jest relacyjna baza danych.

Tabela 1. Tabela Concepts

Nazwa pola	Typ	Klucz	Opis
ID	Numeryczny	Podstawowy	Identyfikator pojęcia
Name	Tekstowy	-	Nazwa pojęcia

Tabela 2. Tabela Relations

Nazwa pola	Typ	Klucz	Opis
ID	Numeryczny	Podstawowy	Identyfikator relacji
Type	Wyliczeniowy: „jest”, „dotyczy”, „trzyma”	-	Typ relacji
StartID	Numeryczny	Obcy (Concepts.ID)	ID dla początku relacji
EndID	Numeryczny	Obcy (Concepts.ID lub Relations.ID)	ID dla końca relacji

ICSN składa się z pojęć i relacji (związków), które mogą być zapamiętane w tabelach Concepts oraz Relations. Podczas zapisu ICSN do relacyjnej bazy danych każda relacja „jest tożsamy z” jest przekształcana na dwie relacje „jest”. Podczas

wnioskowania korzysta się jedynie z trójki relacji: „jest”, „dotyczy” oraz „trzyma”. Ponadto nie zapisuje się w bazie negacji relacji. Brak relacji w bazie oznacza negację relacji (założenie o zamkniętym świecie). Tabela 1 przechowuje odwzorowania pomiędzy identyfikatorami a nazwami pojęć. Tabela 2 przechowuje pary identyfikatorów pojęć dla relacji typu „jest” oraz „dotyczy” oraz pary identyfikatora pojęcia będącego sytuacją i identyfikatora relacji składowej dla relacji typu „trzyma”.

6. PODSUMOWANIE

W pracy opisano jedynie najważniejsze elementy modelu ICKR. Pełny opis modelu wymaga dalszych badań. Najważniejsze z nich to:

- **Weryfikacja eksperymentalna.** Dotychczasowa weryfikacja polegała na sprawdzaniu działania modelu w teoretycznych symulacjach przeprowadzanych na prostych przykładach. Odpowiedniej ilości użytecznych danych mogłaby dostarczyć praktyczna implementacja i przetestowanie modelu w rzeczywistym systemie.
- **Zbadanie efektywności i optymalizacja.** Potrzebne jest gruntowne sprawdzenie efektywności zaproponowanych algorytmów [1].
- **ICKR w obiektowej i dedukcyjnej bazie danych.** Uzasadnione jest zaproponowanie rozwiązań z wykorzystaniem baz danych zgodnych z innymi paradygmatami. Obiektowe bazy danych kładą duży nacisk na relację podtyp-nadtyp, ICKR opiera się na równoważnej relacji „jest”. Paradygmat dedukcyjny ma wspólny z ICKR sposób wnioskowania z wykorzystaniem implikacji, podstawiania zmiennych i wycyfowania.
- **Dostosowanie do języka polskiego.** Odpowiadające konstrukcje w języku polskim brzmią dość nienaturalnie. W związku z tym istnieje potrzeba zbudowania wersji dla języka polskiego. Nie jest to zadanie trywialne, gdyż w języku polskim nie istnieją tak użyteczne w przetwarzaniu języka naturalnego elementy jak „of” lub „s”.

Jednym z elementarnych założeń modelu było wzmocnienie tożsamości pojęć w modelu poprzez maksymalne ograniczenie liczby rodzajów podstawowych relacji pomiędzy nimi przy zachowaniu uniwersalności modelu. Mimo, iż liczba podstawowych relacji została ograniczona do dwóch, możliwe okazało się odwzorowanie w modelu szerokiego spektrum konstrukcji języka naturalnego, co autorzy uważają za znaczące osiągnięcie tej pracy.

Podsumowując, praca opisuje koncepcję reprezentacji wiedzy wykorzystującej skojarzenia opartej na relacyjnym modelu baz danych, która jednak wciąż wymaga dalszych badań, aby można było mówić o jej znaczącej praktycznej użyteczności.

LITERATURA

- [1] Burda M.. (2004), *Wprowadzanie, modyfikowanie i wyszukiwanie informacji niesformalizowanej wykorzystującej skojarzenia*. Praca magisterska Wydziałowego Zakładu Systemów Informacyjnych Politechniki Wrocławskiej, Wrocław, 2004.
- [2] Hartley R. T., Barnden J. A. (1997), *Semantic networks: visualisations of knowledge*. W: *Trends in Cognitive Sciences*, sierpień 1997, tom 1, nr 5, 169–175.
- [3] Hutchison Ch. (1996), *Reasoning with Semantic Nets*. W: Sharples, Hogg, Hutchinson, Torrance, Young: *Computers and Thought: A Practical Introduction to Artificial Intelligence*.
- [4] Klenk J., Binning G., Schmidt G. (2000), *Handling Complexity with Self-Organizing Fractal Semantic Networks*. W: *Emergence* (2000), 2(4), 151–162.
- [5] Król D., Kojas Ł. (2003), *Wyszukiwanie danych w obiektowej bazie danych*. Prace Naukowe Wydziałowego Zakładu Informatyki Politechniki Wrocławskiej – Bazy danych, Zeszyt 4, Wrocław, 2003, 19–40.
- [6] Reisig W. (1988), *Sieci Petriego*. WNT, Warszawa, 1988.
- [7] Sowa J. F. (2002), *Semantic networks* (<http://www.jfsowa.com/pubs/semnet.htm>).
- [8] Tadeusiewicz R. (1993), *Sieci neuronowe*. Akademicka Oficyna Wydawnicza, Warszawa, 1993.
- [9] http://en.wikipedia.org/wiki/Semantic_network, Wikipedia, *Semantic networks*, 2004.

SEMANTIC DATABASE MODEL BASED ON ASSOCIATIONS

Semantic database model based on associations is presented. For the model construction rules to formal graphical notations of ICSN are proposed. Two relation tables Concepts and Relations create the fundamentals for semantic database.

Wydawnictwa Politechniki Wrocławskiej
są do nabycia w następujących księgarniach:
„Politechnika”

Wybrzeże Wyspiańskiego 27, 50-370 Wrocław
budynek A-1 PWr., tel. (0-71) 320-25-34;
„Tech”

plac Grunwaldzki 13, 50-377 Wrocław
budynek D-1 PWr., tel. (0-71) 320-32-52
Prowadzimy sprzedaż wysyłkową

ISBN 83-7085-847-3