

Uniwersytet Mikołaja Kopernika  
Wydział Matematyki i Informatyki

Michał Zdzisław Burzański  
Nr albumu: 158 045

Praca magisterska  
na kierunku Informatyka

# YODA – eksperymentalna implementacja systemu zarządzania obiektową bazą danych

Praca wykonana pod kierunkiem  
Dr. Piotra Wiśniewskiego  
Wydział Matematyki i Informatyki

Toruń 2006



# Spis treści

1. Wstęp.....	4
1.1. Podstawowe pojęcia.....	4
1.2. Podstawowe koncepcje wykorzystane podczas tworzenia systemu YODA.....	5
2. Modele danych wraz ze składem danych.....	6
2.1. Model kartkowy.....	6
2.2. Model płaski.....	6
2.3. Model hierarchiczny.....	7
2.3.1. Struktury hierarchicznych baz danych.....	7
2.3.2. Własności schematu hierarchicznego.....	8
2.3.3. Wirtualne związki rodzic-potomek.....	9
2.3.4. Ograniczenia w modelu hierarchicznym.....	10
2.3.5. Skład danych w systemie IMS.....	11
2.3.6. Podsumowanie.....	11
2.4. Model sieciowy.....	12
2.4.1. Struktury sieciowego modelu danych.....	12
2.4.2. Typy kolekcji i ich podstawowe własności.....	12
2.4.3. Specjalne typy kolekcji.....	13
2.4.4. Skład instancji zbiorów.....	14
2.4.5. Użycie kolekcji do reprezentacji relacji 1:1 oraz M:N.....	15
2.4.6. Ograniczenia w sieciowym modelu danych.....	17
2.4.6.1. Opcje dodawania rekordów na kolekcjach.....	17
2.4.6.2. Opcje przechowywania rekordów na kolekcjach.....	17
2.4.6.3. Opcje porządkowania kolekcji.....	18
2.4.7. Sieciowy System Baz Danych - IDMS.....	18
2.4.7.1. Skład danych w systemie IDMS.....	18
2.4.8. Podsumowanie.....	19
2.5. Model Relacyjny.....	19
2.5.1. Reprezentacja danych w modelu relacyjnym.....	20
2.5.1.1. Dziedziny, krotki, atrybuty, relacje.....	20
2.5.1.2. Charakterystyka relacji.....	21
2.5.2. Ograniczenia modelu relacyjnego.....	22
2.5.2.1. Ograniczenia dziedziny atrybutów.....	22
2.5.2.2. Klucze.....	22
2.5.2.3. Ograniczenia związane z integralnością danych.....	22
2.5.2.4. Integralność referencji, integralność encji.....	23
2.5.3. 12 reguł Codd'a.....	23
2.5.4. Podsumowanie.....	24
2.6. Model relacyjno-obiektowy.....	26
2.6.1. Definicja danych.....	26
2.6.1.1. Abstrakcyjne typy danych (ang. ADT).....	26
2.6.1.2. Zagnieżdżone relacje.....	26
2.6.1.3. Mechanizmy przechowywania danych.....	27
2.6.1.4. Indeksowanie danych.....	28
2.6.2. Podsumowanie.....	28
2.7. Model obiektowy ODMG.....	29
2.7.1. Dlaczego zaistniała potrzeba stworzenia obiektowego modelu danych.....	29
2.7.2. Standard ODMG.....	29
2.7.2.1. Manifest systemów zarządzania obiektową bazą danych.....	30
2.7.2.2. Składowe standardu ODMG.....	30

2.7.2.3. Model obiektowy.....	31
2.8. Model składu obiektów prof. K. Subiety.....	33
2.8.1. Opis modeli składu obiektów.....	34
2.8.1.1. Model składu M0.....	34
2.8.1.2. Model składu M1.....	36
2.8.1.3. Model M2 - modelowanie dynamicznych ról.....	38
2.8.1.4. Model M3 – hermetyzacja i ukrywanie informacji.....	38
2.8.1.5. Schemat bazy danych dla modeli składu danych.....	39
2.8.2. Podsumowanie.....	40
3. Wykorzystane technologie.....	41
3.1. Java.....	41
3.2. XML - Extensible Markup Language.....	41
3.2.1. Modelowanie plików XML.....	42
3.3.2. DTD – Document Type Definition.....	43
3.3.3. Schemat XML.....	43
3.3.4. Przetwarzanie plików XML.....	43
3.3.5. Wersje XML.....	44
4. Architektura klient-serwer.....	45
4.1. Model klient serwer.....	45
4.2. Właściwości klienta i serwera.....	45
4.2.1. Typy serwerów.....	46
4.2.2. Podział klientów.....	46
4.3. Serwer iteracyjny a serwer współbieżny.....	46
4.4. Obsługa połączeń w modelu klient-serwer.....	47
4.4.1. Etapy nawiązywania komunikacji.....	47
4.5. Metody komunikacji.....	47
5. Baza danych YODA.....	49
5.1. Skład danych.....	49
5.1.1. Struktura klas obiektów w systemie YODA.....	51
5.1.2. Typy obiektów.....	52
5.2. Interakcja użytkownika z bazą.....	53
5.3. Konsola klienta systemu YODA.....	53
5.4. Budowa serwera.....	53
5.5. Uruchomienie systemu YODA.....	54
6. Zawartość dysku CD:.....	56
7. Bibliografia.....	57

## 1. Wstęp.

Przez ostatnie kilka lat nasiliła się tendencja w bazach danych do dążenia w kierunku obiektowości. Model relacyjny nie spełniał wszystkich potrzeb użytkownika. Również język manipulacji danymi był ujednolicony i opierał się w głównej mierze na SQL-u. Bazy relacyjne, choć podparte na silnych podstawach matematycznych, stawały się już powoli przestarzałe.

Zamysły na powstanie obiektowej bazy danych pojawiły się już we wczesnych latach 80-tych. Pojęcie “obiekto-zorientowane bazy danych” pojawiło się pierwszy raz około roku 1985. Zaczęły wtedy powstawać zaczątki obiektowych baz danych takie jak systemy: Encore-Ob/Server, EXODUS, IRIS, ORION, ODE oraz kilka innych. Niektóre z tych systemów przetrwały do dziś obudowane np.: w język Smalltalk (system GemStone).

Celem tej pracy było stworzenie i zaimplementowanie w pełni funkcjonalnego obiektowego składu danych. Część tekstowa podzielona jest na siedem rozdziałów (wraz z bibliografią). Pierwszym rozdziałem jest wstęp. W drugim rozdziale staram się przybliżyć w prosty i czytelny sposób ewolucje baz danych. Przedstawiam chronologicznie koncepcje jakie rodziły się w świecie baz danych. Począwszy od modelu kartkowego, poprzez modele hierarchiczne, relacyjne dochodzimy do modelu obiektowego. W rozdziale trzecim opisuje XML, czyli specyficzny język markowania danych. Opisany jest on w dość szczegółowy sposób, ponieważ kopia zapasowa bazy systemu YODA opiera się właśnie na XML-u. Z niego wczytywane są dane składu do pamięci, oraz po zakończeniu procesu serwera. Rozdział czwarty traktuje o architekturze klient-serwer. Przedstawiam w nim ogólne zarysy tego modelu. Na czym polega komunikacja, oraz jakie typy połączeń występują między klientem i serwerem. Rozdział piąty opisuje w pełni funkcjonalny skład danych zaimplementowany w systemie YODA. Kod źródłowy bazy danych zawarty jest na płycie CD, natomiast listing zawartości tego CD opisany jest w rozdziale szóstym.

### 1.1. Podstawowe pojęcia.

#### **Obiekt**

To podstawowe pojęcie wchodzące w skład pradygmatu obiektowości w analizie i projektowaniu oprogramowania oraz w oprogramowaniu.

**Jest to struktura zawierająca:**

- dane
- metody, czyli funkcje służące do wykonywania na tych danych określonych zadań.

Z reguły obiekty (a właściwie klasy, do których te obiekty należą) są konstruowane tak, aby dane przez nie przenoszone były dostępne wyłącznie przez odpowiednie metody, co zabezpiecza je przed niechcianymi modyfikacjami. Takie zamknięcie danych nazywa się enkapsulacją.

### **DBMS:**

**System Zarządzania Bazą Danych, SZBD** (ang. **Data Base Management System, DBMS**) nazywany też **serwerem baz danych** lub **systemem baz danych, SBD** to oprogramowanie bądź system informatyczny służący do zarządzania komputerowymi bazami danych. Systemy baz danych mogą być sieciowymi serwerami baz danych lub udostępniać bazę danych lokalnie lokalnie.

### **Encja:**

To model lub jednostka danych - reprezentacja wyobrażonego lub rzeczywistego obiektu stosowana przy modelowaniu danych podczas analizy informatycznej. Może posiadać atrybuty i operacje. Powinna mieć nazwę w liczbie pojedynczej.

## **1.2. Podstawowe koncepcje wykorzystane podczas tworzenia systemu YODA.**

System, jako że miał być wieloplatformowy i w pełni skalowalny, został zaimplementowany w języku Java w wersji 1.5\_06, przy pomocy środowiska programistycznego Eclipse. Jako, że Java pozwala na tworzenie aplikacji multiplatformowych, system nie ma ograniczeń na systemu operacyjnym. Jedynym warunkiem koniecznym do startu jest zainstalowana maszyna wirtualna Javy. System miał być łatwy w użyciu, a jego kod czytelny i przejrzysty.

W rezultacie moich prac nad składem i połączeniami klient serwer, oraz prac nad językiem PySBQL zaprojektowanym przez panią Martę Rogińską powstał system zarządzania bazą danych YODA.

## **2. Modele danych wraz ze składem danych.**

Model baz danych to pewien zbiór zasad, którymi należy się posługiwać podczas tworzenia bazy danych. Definiuje się w nim strukturę danych poprzez specyfikacje reprezentacji dozwolonych w modelu obiektów (encji) oraz ich związków. W modelu danych określa się operacje oraz reguły, zgodnie z którymi dane umieszczane są w strukturach. Opiszemy teraz poszczególne modele baz danych poczynając od najstarszego znanego modelu – kartkowego.

### **2.1. Model kartkowy.**

Nie jest on modelem danych w pełnym znaczeniu tego słowa, ale to najstarszy model przechowywania danych, jaki funkcjonuje do dziś. Jest oparty o informację zapisywaną na papierze. Model ten jest najpowszechniej stosowany w bibliotekach, które gromadzą w ten sposób informacje o książkach, czytelnikach i wypożyczeniach. Zastosowanie znajduje także w sądownictwie oraz urzędach. Ze względu na objętość papieru, większe bazy informacji mogą zajmować całe szafy.

### **2.2. Model płaski.**

Płaski model, czasem zwany kartotekowym, składa się ze zwykłej, dwuwymiarowej tablicy, w której elementy danego wiersza są powiązane ze sobą. Wartości danej kolumny są jednolitego typu. Dane w kartotekowych bazach danych można sortować, przeszukiwać, stosować w nich filtry ograniczające zakres wyświetlanych informacji. Każda tablica danych jest samodzielnym dokumentem i nie może współpracować z innymi tablicami, w przeciwieństwie do relacyjnej bazy danych. Przykładem takiego modelu może być arkusz kalkulacyjny.

## **2.3. Model hierarchiczny.**

W tym modelu dane zorganizowane są w struktury drzewiaste. Model hierarchiczny był szeroko stosowany we wczesnych systemach zarządzania bazami danych, np. w systemie IMS (Information Management System) stworzonym przez IBM. Model ten nadal używany jest przez komputery stacjonarne do reprezentacji struktury plików. Innym przykładem modelu hierarchicznego jest struktura dokumentu XML.

Hierarchiczny model danych został stworzony aby odwzorowywać zależności hierarchiczne istniejące w świecie rzeczywistym, głównie hierarchie różnych organizacji. Hierarchiczny porządek jest najstarszą formą organizacji danych. Ludzie od dawna stosowali go aby wytłumaczyć sobie zależności otaczającego ich świata. Przykładami mogą być klasyfikacja gatunków, drzewa genealogiczne, hierarchia wojskowa lub struktury organizacyjne. Model hierarchiczny reprezentuje tego typu informacje w sposób czytelny i naturalny, jednakże w przypadku innych struktur nie sprawdza się równie dobrze.

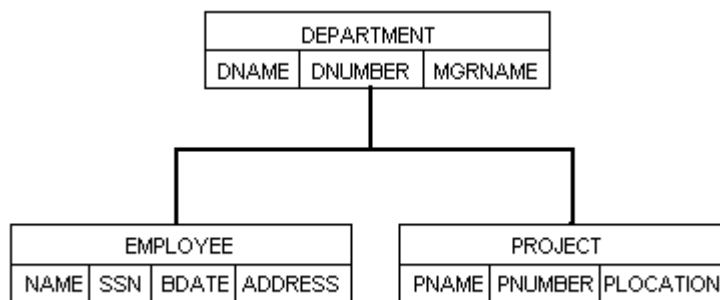
W odróżnieniu od modelu sieciowego i relacyjnego, nie istnieją żadne oficjalne dokumenty ani specyfikacje dotyczące modelu hierarchicznego.

### **2.3.1. Struktury hierarchicznych baz danych.**

Model hierarchiczny zakłada istnienie dwóch głównych struktur danych: rekordów oraz relacji rodzic-potomek (ang. Parent-child relationship PCR). Zbiór pól wartości, które zapewniają informacje o pewnej encji lub instancji relacji nazywany jest rekordem. Rekordy o takiej samej strukturze typologicznej systematyzowane są jako typy rekordów, z których każdy ma własną nazwę, a jego struktura zdefiniowana jest przez zbiór nazwanych pól lub jednostek danych

#### **Związki rodzic-potomek (PCR).**

Struktura ta umożliwia pojedynczą relację 1:N pomiędzy dwoma typami danych. Typ rekordu stojący po jednej stronie określany jest jako „rodzic”, zaś drugi typ rekordu – jako „potomek” relacji PCR. Typ PCR w schemacie hierarchicznym może być identyfikowany poprzez wymienienie pary (rodzic, potomek). Schemat bazy danych może składać się z dowolnej ilości schematów hierarchii, z których każdy zawiera pewne ilości typów rekordów oraz typów relacji PCR. Przykład schematu hierarchii przedstawiony jest na rysunku 2.1.

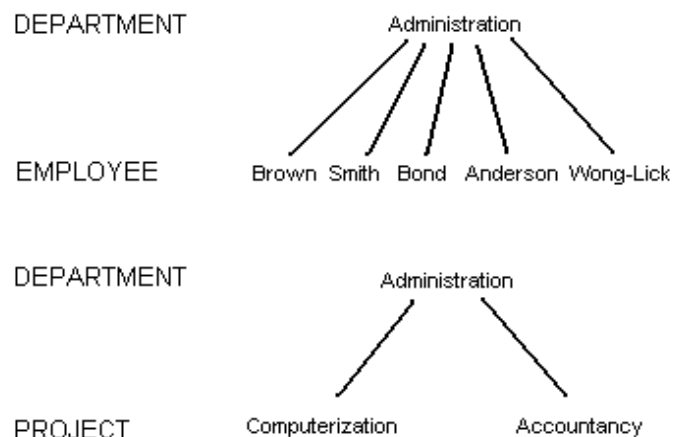


Rys. 2.1. Schemat hierarchii.

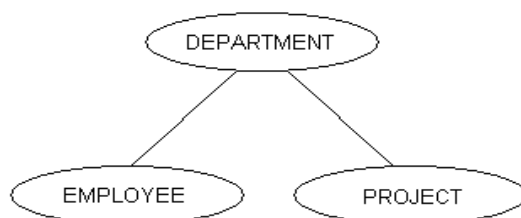
### 2.3.2. Własności schematu hierarchicznego

Schemat hierarchiczny typów rekordów PCR musi spełniać następujące własności:

- Jeden typ rekordu, zwany korzeniem schematu hierarchicznego, nie występuje w żadnym typie PCR jako potomek
- Każdy typ rekordu występuje jako potomek w dokładnie jednym typie PCR
- Typ rekordu może występować jako rodzic w dowolnej liczbie (od zera wzwyż) typów PCR
- Typ rekordu który nie występuje jako rodzic w żadnym typie PCR zwany jest liściem schematu hierarchicznego
- Jeżeli typ rekordu występuje jako rodzic w więcej niż jednym typie PCR wówczas jego potomkowie są uszeregowani. Porządek potomków odzwierciedlony jest w schemacie hierarchicznym poprzez uszeregowanie węzłów od lewej do prawej



Rys. 2.2. Instancje typów PCR



Rys. 2.3. Drzewiasta reprezentacja schematu hierarchicznego z rys. 2.1.

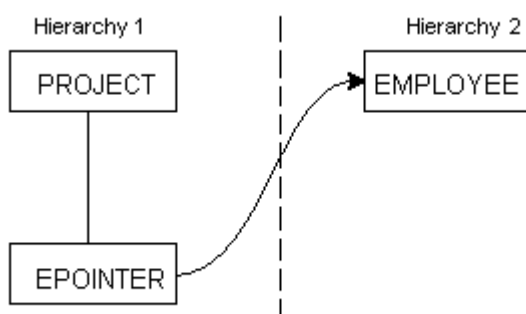
### 2.3.3. Wirtualne związki rodzic-potomek.

Podczas modelowania pewnych typów relacji uwidaczniane są niektóre problemy modelu hierarchicznego. Tak się dzieje dla poniższych przykładów:

- Relacje M:N
- Przypadki, w których dany rekord jest potomkiem w przynajmniej dwóch typach PCR
- Relacje wiążące więcej niż dwa rekordy członkowskie.

Duplikacja rekordów, nie dość że wpływa na zbędne obciążenie pamięci, powoduje problemy z utrzymaniem spójności danych wewnątrz kopii tego samego rekordu. Z pomocą przychodzi koncepcja wirtualnego typu rekordu wykorzystana w systemie IMS do obejścia wyżej wymienionych problemów.

Wirtualny (pointerowy) typ rekordu (ang. virtual child VC – wirtualny potomek) to typ rekordu, w którym każdy rekord zawiera wskaźnik do rekordu innego typu (ang. virtual parent VP – wirtualny rodzic). Wirtualne relacje pomiędzy typami VC i VP zwane są typami wirtualnych związków rodzic-potomek (ang. virtual parent-child relationships types VPCR)



Rys. 2.4. Reprezentacja relacji M:N przy użyciu typów VPCR  
(Wirtualnym rodzicem jest EMPLOYEE)

Employee jest zwany wirtualnym rodzicem Epointera, który z kolei zwany jest wirtualnym potomkiem Employee.

Koncepcyjnie typy PCR i VPCR są bardzo podobne. Główną różnicą jest sposób ich implementacji. Typy PCR są zwykle implementowane przy użyciu ciągów hierarchicznych, zaś typy VPCR – przy użyciu wskaźników (fizycznych – zawierających adres, bądź logicznych – zawierających identyfikator) do rekordu wirtualnego rodzica. Wpływa to na wydajność pewnych zapytań.

Ogólnie mówiąc jest wiele wygodnych metod projektowania bazy danych przy użyciu modelu hierarchicznego. W wielu wypadkach wydajność takiej bazy danych sprawia że dla pewnych zastosowań model hierarchiczny ma przewagę nad chociażby relacyjnym. Użyteczność schematu hierarchicznego opiera się na możliwościach implementacyjnych konkretnego systemu oraz na istnieniu ograniczeń konkretnego oprogramowania bazodanowego, np. wymogów co do typu pointerów, czy ograniczeniu liczby dostępnych poziomów.

#### 2.3.4. Ograniczenia w modelu hierarchicznym.

W hierarchicznym schemacie występuje pewna liczba wbudowanych dziedzicznych ograniczeń. Nie należy jednak mylić pojęcia ograniczenie z pojęciem wada. Ograniczenia te nakładają pewną kulturę postępowania z rekordami. Są nimi następujące reguły:

- Za wyjątkiem rekordów-korzeni, żaden rekord nie może istnieć w bazie bez rodzica. Ma to swoje następstwa:
  - Rekord potomny nie może zostać wstawiony do bazy bez połączenia go z rekordem rodzica
  - Rekord potomny może zostać usunięty niezależnie od swojego rodzica, jednakże usunięcie rodzica powoduje automatyczną kasację rekordów potomnych.
  - Powyższa reguła nie ma zastosowania do wirtualnych rodziców i potomków. W ich przypadku o ile istnieje wskaźnik do danego rekordu (czyniący go wirtualnym rodzicem) – nie może on zostać skasowany.
- Jeżeli dany rekord potomny ma dwóch lub więcej rodziców tego samego typu, powinien zostać skopiowany dla każdego rodzica z osobna.
- Rekord potomny aby posiadać dwóch lub więcej rodziców różnych typów, musi jednego wskazać jako rekord rodzica, a do pozostałych mieć wskaźniki (czyniąc ich wirtualnymi rodzicami).

Dodatkowo, każdy hierarchiczny DBMS posiada własne reguły integralnościowe. Dla przykładu w systemie IMS typ rekordu może być wirtualnym potomkiem w dokładnie jednym typie VPCR. Ograniczenia które nie są wbudowane w konkretny DBMS muszą być implementowane przez programistę programów do uaktualniania bazy. Dla przykładu, jeżeli rekord posiadający duplikaty ma zostać uaktualniony, wówczas program aktualizujący musi zapewnić jednolite uaktualnienie wszystkich kopii tego rekordu.

### 2.3.5. Skład danych w systemie IMS.

Omówię tu ogólnie kilka różnych typów organizacji plików, które są stosowane do fizycznego składu zawartości baz danych w systemie IMS. W tym systemie nazywane są one metodami dostępowymi. W porównaniu do innych hierarchicznych DBMS'ów IMS oferuje o wiele większy wachlarz owych metod.

- Każda fizyczna baza danych w IMSie jest przechowywana. Logiczne bazy to wirtualne hierarchiczne bazy danych które mogą być przeglądane jako fizyczne bazy jednakże w samym składzie nie posiadają osobnych danych. Logiczne bazy danych składają się z fizycznych baz wzbogaconych o struktury pointerowe.
- Każdy przechowywany segment zawiera pola danych plus prefiks nie widoczny dla użytkownika programu. Ów prefiks składa się z kodu typu segmentu, pointerów, flagi kasowania oraz innych informacji kontrolnych
- Niezależnie od metod dostępowych, przechowywana baza danych jest zawsze składowana w kolejności wystąpień drzew, zwanej fizycznymi rekordami bazy. Każda instancja drzewa (w skrócie zwana po prostu drzewem) zawiera uporządkowany ciąg segmentów. Jej właścicielem jest konkretny segment-korzeń.
- Różne metody dostępowe IMSu mają odrębne podejście do sposobu powiązania segmentów ze sobą wewnątrz fizycznej bazy danych. Różnią się również rodzajami struktur dostępowych umożliwiającymi zlokalizowanie fizycznego rekordu bazy, bądź pojedynczego segmentu wewnątrz owego rekordu.
- W oparciu o typ dostępu przypisany do danego rekordu fizycznej bazy IMS wprowadza dwie struktury: ciągi hierarchiczne oraz hierarchiczne wskaźniki.

### 2.3.6. Podsumowanie

W tym rozdziale omówiony został model hierarchiczny, który reprezentuje dane w postaci struktur drzewiastych. Używa on hierarchicznych relacji do reprezentacji zależności pomiędzy danymi. Omówione zostały ogólne założenia modelu, chociaż niektóre jego aspekty wzorowane były na implementacji systemu IMS firmy IBM.

Głównymi strukturami używanymi w modelu hierarchicznym są typy rekordów oraz typy relacji rodzic-potomek (PCR). Relacje są ściśle hierarchiczne, w związku z czym konkretny typ rekordu może uczestniczyć jako potomek w co najwyżej jednym typie PCR. To ograniczenie sprawia, że omawiany tu model ma trudności z odwzorowywaniem danych posiadających wielokrotne powiązania.

## **2.4. Model sieciowy.**

Model sieciowy (zdefiniowany w 1971 roku przez grupę CODASYL) przechowuje dane przy użyciu dwóch podstawowych struktur: rekordów i kolekcji. Rekordy zawierają pola, które mogą być zorganizowane w hierarchie, tak jak ma to miejsce w języku COBOL. Kolekcje definiują relacje 1:N pomiędzy rekordami: jednym rekordem „właścicielem” i wieloma „członkami”. Dany rekord może być jednocześnie właścicielem i członkiem dowolnej liczby kolekcji. Pomimo, że nie jest to istotną cechą modelu, sieciowe bazy danych zwykle implementują kolekcje przy użyciu wskaźników, które bezpośrednio adresują rekord przechowywany na dysku. Zapewnia to bardzo szybkie pozyskiwanie danych, kosztem wczytywania i reorganizacji bazy danych.

### **2.4.1. Struktury sieciowego modelu danych**

Jak już wcześniej wspominałem, w sieciowym modelu dane są przechowywane w rekordach. Rekordy możemy klasyfikować według ich typów. Każdy z typów rekordów posiada swoją nazwę, oraz dla każdego atrybutu w swoim obrębie, typ danych jaki on przechowuje. Atrybuty mogą być następujących typów:

- typy podstawowe – zawartość jest typu prostego np.: String, Integer, itp.,
- wektory – mogą one mieć wiele wartości w pojedynczym rekordzie,
- grupy powtórzeniowe – rekord zawiera powtarzające się struktury danych, mogą one być wielokrotnie zagnieżdżone.

Wszystkie wymienione powyżej struktury są przechowywane w rekordach, zatem można je nazwać faktycznymi danymi składowymi. Musimy jednak rozpatrzeć jeszcze jedną grupę danych, tzw. wirtualne dane składowe. Nie są one przechowywane w rekordach a służą nam do wydobywania pewnych wartości danych zawartych w tych rekordach.

Rozważmy zatem przykład bazy danych w której mamy pola: imię, nazwisko, nazwa\_egzaminu oraz data w których zawarte są imiona i nazwiska studentów oraz egzaminy i daty kiedy się owe odbędą. Możemy wtedy dla każdego studenta stworzyć wirtualną daną składową Ilość\_dni\_do\_egzaminu, która pozwoli nam składować (po napisaniu odpowiedniej procedury, bądź funkcji) obliczoną z pola data ilość dni do egzaminu.

### **2.4.2. Typy kolekcji i ich podstawowe własności.**

Każda definicja typu kolekcji zawiera trzy podstawowe informacje (własności):

- nazwę typu kolekcji,

- członka typu rekordu,
- właściciela typu rekordu,

oraz opisuje relację 1:N pomiędzy dwoma typami rekordów.

W samej bazie danych, może się zdarzyć wiele wystąpień kolekcji odpowiadających danemu typowi kolekcji. Dlatego każda instancja kolekcji składa się z:

- jednego typu właściciela ze wskazanego typu rekordów,
- pewnej liczby (zero lub więcej) powiązanych rekordów członkowskich o typie rekordu członkowskiego.

Rekord członkowski z pewnego typu rekordów nie może należeć do więcej niż jednego wystąpienia danego typu kolekcji. Wystąpienie kolekcji może być identyfikowane na podstawie rekordu właściciela, bądź też na podstawie jakiegokolwiek rekordu członkowskiego. Oczywiście wiadomym jest, że każda instancja kolekcji musi mieć właściciela, ale nie zawsze musi mieć rekord członkowski, ponieważ rekordów tych może być zero lub więcej.

Mówiąc o kolekcjach w sieciowym modelu danych musimy rozróżnić tą definicję od definicji zbioru stosowanej w matematyce. Ponieważ występują tu dwie podstawowe różnice:

- w matematyce nie mamy zróżnicowania na elementy zbioru, czyli żaden z elementów nie jest nadrzędnym (wyróżnionym) elementem jak to ma miejsce w opisie zbioru w sieciowym modelu danych
- w sieciowym modelu danych elementy członkowskie są uporządkowane, co w matematycznym podejściu do zbioru nie ma znaczenia. Uporządkowanie to daje nam możliwość odwołania się do poszczególnych elementów tego zbioru.

Dlatego bardziej poprawnym określeniem tego typu grup elementów jest pojęcie kolekcji. Z matematycznego punktu widzenia, oraz z dedukcji na podstawie powyższych różnic, kolekcja jest ciągiem.

### 2.4.3. Specjalne typy kolekcji.

Wyróżniamy dwa podstawowe typy kolekcji w sieciowym modelu danych CODASYL, oraz tzw typ rekurencyjny który nie jest jednak dozwolony w modelu CODASYL.

1. Kolekcje, których właścicielem jest system (system-owned sets) charakteryzują się tym, że nie posiadają typu rekordu właściciela. O systemie możemy myśleć jako o wirtualnym typie rekordu, który ma dokładnie jedną instancję.

Kolekcje te zapewniają punkty dostępowe do bazy danych, oraz służą do porządkowania danego typu rekordów.

2. Kolekcje, których członkowie są różnych typów (multimember sets) nie zostały zaimplementowane w obecnie dostępnych systemach zarządzania bazami danych, pomimo że model CODASYL zakłada ich istnienie.
3. Kolekcje rekurencyjne (recursive sets) charakteryzują się tym, że ten sam typ rekordu gra rolę zarówno właściciela jak i członka. Były one zabronione w sieciowym modelu CODASYL ponieważ istniały trudności z ich przetwarzaniem za pomocą języka manipulacji danymi (DML). Aby sobie poradzić z kolekcjami rekurencyjnymi zastosowano dodatkowy prymitywny łączący typ rekordu.

#### 2.4.4. Skład instancji zbiorów.

Instancje kolekcji są przedstawiane za pomocą pierścieni (bądź też za pomocą list cyklicznych) wiążących właściciela ze swoimi członkami.

Reprezentacja ta jest symetryczna w odniesieniu do wszystkich rekordów. Jednakże system zarządzania bazą danych (DBMS) zawiera specjalne pole określające czy dany rekord jest właścicielem czy członkiem kolekcji. Pole to jest widoczne tylko dla systemu zarządzania bazą danych, natomiast jest ukryte dla użytkownika. W odniesieniu do tego pola przydzielane jest pole pointerowe dla każdego typu kolekcji, który uczestniczy jako członek bądź właściciel. Jest ono oznaczane typem kolekcji do której się odnosi.

Mamy dwa rodzaje pól pointerowych:

1. pole typu FIRST – będące w rekordzie – właścicielu,
2. pole typu NEXT - zawarte w każdym rekordzie członkowskim.

Jeśli rekord należący do kolekcji rekordów członkowskich nie uczestniczy w żadnej instancji zbioru jego pole NEXT wypełniane jest specjalną wartością **nil**.

DBMS może mieć w różny sposób oprogramowane kolekcje, jednakowoż wybrana reprezentacja musi spełniać pewne warunki. Po pierwsze startując od właściciela jesteśmy w stanie znaleźć wszystkie rekordy członkowskie. Po drugie jeśli mamy danego właściciela jesteśmy w stanie wskazać jego pierwszy, i-ty, oraz ostatni rekord członkowski. Po trzecie posiadając jakikolwiek rekord członkowski jesteśmy w stanie przejść do poprzedniego, bądź kolejnego rekordu o ile taki istnieje (jeśli tak nie jest zgłaszamy tę okoliczność). Po czwarte na podstawie rekordu członkowskiego jesteśmy w stanie wskazać kto jest jego właścicielem.

Reprezentacja cykliczna za pomocą listy pozwala systemowi na robienie wszystkich operacji zgodnie ze skalą wydajności. Jest tak ponieważ schemat sieciowej bazy danych zakłada posiadanie wielu typów rekordów, oraz kolekcji typów uczestniczących jako właściciel, bądź członek w wielu zbiorach.

Występują także inne przedstawienia kolekcji, które w sposób bardziej efektywny wpływają na

przebieg operacji na tych zbiorach:

1. Podwójna lista cykliczna – pozwala ona na poruszanie się po kolekcji w obu kierunkach. Wprowadzone jest nowe dodatkowe pole – PRIOR.
2. Reprezentacja poprzez wskaźniki właściciela. Dla każdej kolekcji typów wprowadzone jest nowe pole OWNER (wskaźnik na właściciela) dla każdego rekordu członkowskiego.
3. Reprezentacja sąsiadujących rekordów. Polega ona na fizycznym umiejscowieniu członków obok siebie tuż za swoim właścicielem.
4. Tablice pointerów (wskaźników) – dla każdego elementu tablicy mamy skojarzony z nim element zbioru. Rozważmy zatem sytuację. Weźmy np.: i-ty element z tablicy pointerów. Naturalnym jest, że musi on wskazywać na i-ty element członkowski w zbiorze instancji. Tą konstrukcją (przedstawienie) stosuje się często w połączeniu z reprezentacją poprzez wskaźniki właściciela.
5. Reprezentacja indeksowana – dla każdego wystąpienia kolekcji przechowywana jest para indeksująca to wystąpienie. Składa się ona z pól: (wartość\_klucza, pointer). Często reprezentacja indeksowana przedstawiana jest za pomocą cyklicznej listy związanej

#### 2.4.5. Użycie kolekcji do reprezentacji relacji 1:1 oraz M:N.

Zazwyczaj typ kolekcji jest reprezentowany jako relacja 1:N na dwóch rekordach. Co oznacza, że rekord z kolekcji rekordów członkowskich może pojawić się dokładnie w jednej instancji zbioru. Jest to pewne ograniczenie nakładane na kolekcje, a przecież nie tylko takie relacje występują w życiu codziennym. Chcielibyśmy również mieć możliwość zastosowania relacji 1:1 oraz wiele do wielu (M:N).

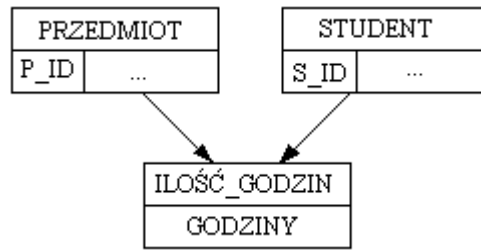
##### **Relacja 1:1.**

Aby móc reprezentować relację 1:1 potrzebne jest nam nałożenie pewnych reguł.

Każde wystąpienie kolekcji musi mieć pojedynczy rekord członkowski. Nad przestrzeganiem tego ograniczenia musi czuwać programista, by za każdym razem gdy dodawany jest nowy rekord nie nastąpiło naruszenie tej reguły.

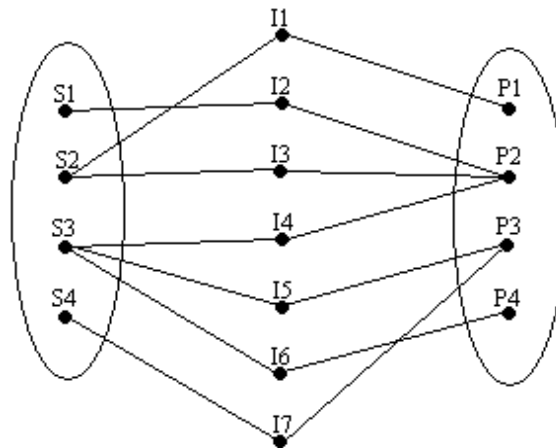
##### **Relacja M:N.**

Nie może być ona reprezentowana poprzez pojedynczy typ kolekcji. Do jej prawidłowej reprezentacji potrzebne są dwa typy kolekcji oraz dodatkowy typ rekordu. Aby lepiej zrozumieć wykorzystanie tych zbiorów i rekordu przeanalizujemy następujące rysunki:

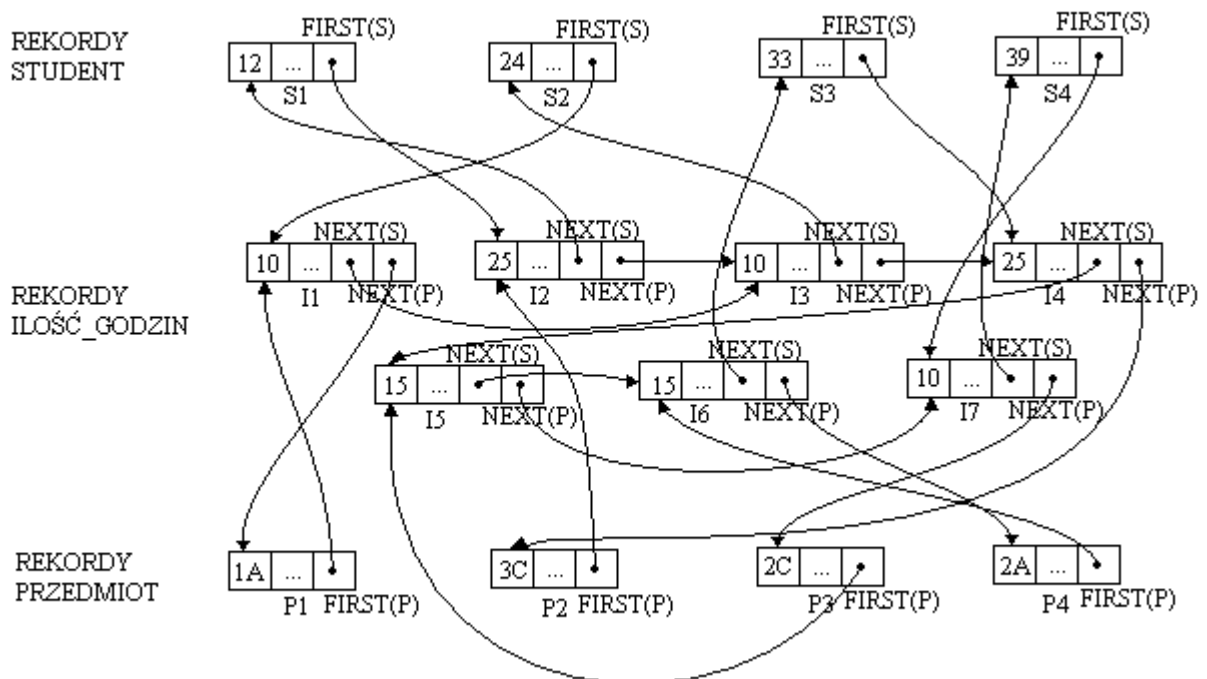


Rys. 2.4.5.1. Schemat relacji M:N

Aby pokazać powiązanie pomiędzy studentem a przedmiotem zastosujemy dodatkowy rekord, który określa ile godzin student poświęca na dany przedmiot (ile godzin w tygodniu jest przeznaczonych na przedmiot). Rekord ten oznaczyłem jako *ilość\_godzin*.



Rys. 2.4.5.2. Przykładowa instancja relacji z rys. 2.4.5.1.



Rys. 2.4.5.3. Szczegółowe przedstawienie powiązań pointerowych z rys. 2.4.5.2.

Zatem za każdym razem rekord ten musi być posiadany przez rekord *student*, oraz przez rekord *przedmiot*. Warto zauważyć, że rekord *ilość\_godzin* posiada dwa pola pointerowe typu NEXT. Jedno z tych pól wskazuje na kolejny rekord w instancji pokrewnej przedmiotowi, drugi natomiast do instancji pokrewnej studentowi. Słowo “pokrewnej” określa pewien sposób rozróżniania między instancjami przedmiotu i studenta.

#### 2.4.6. Ograniczenia w sieciowym modelu danych.

Do tej pory opisane zostały pewne ograniczenia związane ze strukturą modelu sieciowego. Jednak aby w pełni opisać cały model sieciowy musimy się też zastanowić nad pewnymi ograniczeniami zachowawczymi (behawioralnymi).

Związane są one z operacjami dodawania, usuwania, oraz modyfikacji rekordów.

Można je podzielić na dwie podstawowe kategorie:

- opcje dodawania
- opcje przechowywania

##### 2.4.6.1. Opcje dodawania rekordów na kolekcjach.

Opisują zachowanie modelu sieciowego kiedy dodajemy nowy rekord do bazy danych.

Dostępne są dwie opcje:

1. ręczna – nowy rekord nie jest połączony z żadną kolekcją. Programista może jednak połączyć dany rekord ręcznie z odpowiednią kolekcją.
2. Automatyczna – nowy rekord jest automatycznie łączony z odpowiednią kolekcją podczas jego dodawania.

##### 2.4.6.2. Opcje przechowywania rekordów na kolekcjach.

Określają one czy dany rekord może istnieć w bazie samodzielnie czy musi być ciągle związany z właścicielem jako element jakiejś instancji kolekcji. Także w tym przypadku mamy pewne opcje:

1. Stały (Fixed) – rekord nie może istnieć samodzielnie, co więcej posiada dokładnie jedną instancję do której się odnosi i nie może zostać przełączony do innej instancji.
2. Narzucony (Mandatory) – rekord nie może istnieć samodzielnie i musi być członkiem instancji pewnego typu kolekcji. Jednakowoż może być przełączony do innej instancji w obrębie typu kolekcji.
3. Opcjonalny (Optional) – rekord może istnieć jako samodzielny bez bycia członkiem w żadnej instancji kolekcji. Może być w każdym momencie podłączany i rozłączany od każdej

instancji kolekcji.

Można oczywiście stosować różne kombinacje opcji dodawania i przechowywania rekordów na kolekcjach np.: Automatic-Fixed, albo Manual-Optional. Jednakże nie wszystkie kombinacje są przydatne, a niektóre wręcz wymuszają pewne ograniczenia.

#### 2.4.6.3. Opcje porządkowania kolekcji.

Wyróżniamy różne metody porządkowania instancji kolekcji. Można je zestawić w następujący sposób:

1. Uporządkowane za pomocą pola porządkującego – system wykorzystuje odpowiednie wpisy do tego typu pól by w poprawny sposób wstawiać nowe rekordy w odpowiednie miejsca.
2. Domyślne ustawienie systemu – nowo dodawany rekord ma narzuconą pozycję w którą zostanie dodany.
3. Pierwszy bądź ostatni – nowo dodawany rekord stanie się pierwszym bądź ostatnim elementem instancji w obrębie typu kolekcji.
4. Następny bądź poprzedni – nowo dodawany rekord zostanie wstawiony po bądź przed obecnym rekordem instancji w obrębie typu kolekcji.

#### 2.4.7. Sieciowy System Baz Danych - IDMS.

W tym akapicie chciałbym przybliżyć system zarządzania bazą danych zwany IDMS (Integrated Database Management System), w szczególności skład danych w tym systemie. IDMS jest implementacją koncepcji DBTG CODASYL-u stworzoną przez firmę Cullinet Software. Praca nad stworzeniem tego systemu została zapoczątkowana już w latach 70-tych. Został stworzony aby współpracować z maszynami firmy IBM (serwerami IBM), oraz miał być dostępny na wszystkie systemy operacyjne, które były wtedy dostępne. Swą oficjalną nazwę otrzymał na początku lat 80-tych. Nadano mu wtedy nazwę (zmodyfikowano) IDMS/R, czyli IDMS/Relational (rok 1983). Nadanie nowej nazwy wynikało z tego, że system otrzymał obsługę relacji.

##### 2.4.7.1. Skład danych w systemie IDMS.

W IDMS baza danych jest podzielona na kilka obszarów. Obszary tworzą strony, natomiast te odnoszą się do rzeczywistych fizycznych bloków w plikach. Takie rozwiązanie tworzy ze stron podstawowy mechanizm wejścia / wyjścia.

Relacje pomiędzy obszarami są typu wiele do wielu (M:N). Jest to równoznaczne ze

stwierdzeniem, że każdy obszar może być zmapowany na wiele plików i odwrotnie.

Pliki są podzielone na stałej wielkości bloki zwane stronami. Każdy rekord na stronie posiada prefiks który zawiera linijkę postaci: numer linii (liczony od dołu strony), numer identyfikacyjny rekordu, i długość rekordu. Rekord zawiera ponadto pointery (przynajmniej jeden dla każdej kolekcji w której jest członkiem, bądź właścicielem)

Kolekcje implementowane są w dwojaki sposób. Bądź za pomocą list kolekcji, bądź za pomocą indeksowanych kolekcji.

Pointery typu NEXT są zawarte w tej implementacji zarówno w właścicielach jak i członkach kolekcji. Istnieje oczywiście możliwość stworzenia pointerów typu PRIOR, ale tylko na wyraźne życzenie twórcy bazy

W reprezentacji za pomocą listy indeksowanej, każda instancja kolekcji jest reprezentowana za pomocą rekordu właściciela, a indeks jako kolekcja indeksowanych rekordów. Nie jest konieczne by w tym przypadku indeksy były uporządkowane.

Kolekcje których właścicielem jest system są tworzone jako kolekcje indeksowane. Indeksy w tym przypadku są uporządkowane względem wartości pola porządkującego. Każdy rekord zawiera tę wartość oraz wartość klucza. Mamy tu do czynienia z pojedynczymi wystąpieniami każdej kolekcji, a pojedynczy indeks tu utworzony jest równoznaczny z indeksem porządkującym

W IDMS wprowadzono również dodatkowe rozszerzenia mające na celu uczynić go kompatybilnym z modelem relacyjnym. W rezultacie czego stał się on hybrydą modelu sieciowego i relacyjnego, w której dane można było traktować jako krotki w tabeli (wiersze w tabeli).

#### **2.4.8. Podsumowanie.**

W rozdziale tym przybliżyłem pokrótce model sieciowy. Popularny był on w latach 70-tych. W latach 80-tych został on wyparty przez model relacyjny. Do jego popularności przyczyniła się możliwość odwzorowania w nim wszystkich własności modelu hierarchicznego. Jako że był oparty o teorię grafów, bardzo długo opierał się modelowi relacyjnemu. Niestety przegrał, ponieważ ówczesne komputery nie były w stanie zapewnić mu odpowiedniej wydajności.

### **2.5. Model Relacyjny.**

**Model relacyjny** to model baz danych oparty na postulatach relacyjności. Twórcą teorii relacyjnych baz danych jest nieżyjący już Edgar F. Codd. Zaprezentował on swoją koncepcję modelu danych opisującą stosowane do dziś algorytmy zarządzania relacyjnymi bazami danych w roku 1970 w dokumencie pt.: "A Relational Model of Data for Large Shared Data Banks". Bazowała ona na prostej i zunifikowanej matematycznej teorii relacyjnej. Na koncepcji tej opartych zostało wiele

współczesnych baz danych.

### 2.5.1. Reprezentacja danych w modelu relacyjnym.

Dane w modelu relacyjnym są reprezentowane jako kolekcja krotek, w szczególności w znormalizowanych bazach danych wszystkie one są unikalne i nie gra roli ich kolejność, a dostęp do nich jest realizowany za pomocą algebry relacji (dostęp do danych definiujemy poprzez operatory relacyjne takie jak: rzutowanie, selekcja, złączenie, suma, różnica, produkt kartezjański). Ograniczenie redundancji danych dokonuje się w procesie przejścia do kolejnych postaci normalnych.

#### 2.5.1.1. Dziedziny, krotki, atrybuty, relacje.

W relacyjnym modelu danych wiersze nazywamy krotkami, natomiast nagłówki kolumn atrybutami, natomiast cała tabela nazywana jest relacją. Dlatego też dziedziną nazywamy opis typów danych znajdujących się w każdym wierszu danej tabeli. Alternatywnie dziedziną  $D$  nazywamy zbiór, który zawiera wartości atomowe. Przez wartości atomowe należy rozumieć wszystkie wartości niepodzielne. Każda dziedzina posiada swą nazwę, zbiór wartości oraz format zapisu danych. Rozważmy przykład.

Biorąc pod uwagę zwyczajny spis telefonów można na jego podstawie zbudować relacyjną bazę danych (dla potrzeb naszego przykładu tylko jedną tabelę z relacjami). Załóżmy zatem, że mamy w naszej tabeli następujące atrybuty: imie, nazwisko, numer\_telefonu, adres\_zamieszkania. Opisując dziedziny tych atrybutów:

- dziedzina imię może mieć nazwę imie, typ danych – łańcuch znaków, oraz format danych łańcuch 20 znakowy,
- dziedzina nazwisko może mieć nazwę nazwisko, typ danych – łańcuch znaków, oraz format danych łańcuch 35 znakowy,
- dziedzina numer\_telefonu może mieć nazwę telefon, typ danych – łańcuch znaków, oraz format danych łańcuch 10 znakowy, bądź też być określona bardziej precyzyjnie, jako typ danych – numeryczny, format 9 cyfrowy,
- dziedzina adres\_zamieszkania może mieć nazwę adres, typ danych – łańcuch znaków, oraz format danych łańcuch 152 znakowy.

Gdy wiemy już co to krotka, atrybut, dziedzina, oraz relacja możemy zdefiniować koncepcję relacyjnego schematu.

Schematem relacyjnym  $R$ , nazywamy relację  $n$ -argumentową postaci  $R(A_1, A_2, \dots, A_n)$ , gdzie  $R$  jest

nazwą relacji, natomiast  $A_1, A_2, \dots, A_n$  atrybutami. Każdy z argumentów  $A_i$  jest nazwą roli którą gra pewna dziedzina  $D$  w relacyjnym schemacie. Innymi słowy  $D$  jest dziedziną  $A_i$  (oznaczamy –  $\text{dom}(A_i)$ ). Ilość atrybutów w schemacie nazywamy stopniem schematu. Relacyjny schemat jest używany do opisu relacji (tabeli).

Rozważmy przykład:

Student(indeks,imie,nazwisko,rok\_studiow)

W takim schemacie relacyjnym jego nazwą jest Student, który posiada cztery atrybuty.

Dla tych atrybutów możemy wyspecyfikować ich dziedziny:  $\text{dom}(\text{indeks}) = \text{numery albumów}$ ,  $\text{dom}(\text{imie}) = \text{imiona}$ ,  $\text{dom}(\text{nazwisko}) = \text{nazwiska}$ ,  $\text{dom}(\text{rok\_studiow}) = \text{roczniki studiów (I,II,III,IV,V)}$ .

#### 2.5.1.2. Charakterystyka relacji.

Wspomniane przeze mnie definicje relacji implikują pewną charakterystykę relacji.

##### **Porządkowanie krotek.**

Relacja jest definiowana jako zbiór krotek. Z matematycznego punktu widzenia, elementy zbioru nie muszą być uporządkowane, dlatego też krotki w relacji nie mają konkretnego porządku. Jednakowoż w plikach elementy (rekordy) składowane są w pewien określony sposób. Dla porównania zauważmy, że jeśli wyświetlamy krotki w postaci tabeli, wiersze mają pewien porządek. Porządkowanie krotek nie jest częścią definicji relacji, ponieważ relacja reprezentuje fakty na poziomie logicznym, bądź też abstrakcji. Istnieje wiele logicznych porządków na krotkach. Przyglądając się wspomnianemu już przykładowi:

Student(indeks,imie,nazwisko,rok\_studiow)

możemy, na jego podstawie, uporządkować krotki względem indeksu, albo nazwiska, bądź też innych atrybutów.

Jeśli relacja implementowana jest jako plik, porządkowanie może być realizowane na poszczególnych rekordach tego pliku.

##### **Porządkowanie wartości.**

Cofając się do definicji relacji,  $n$ -krotka jest uporządkowaną listą o  $n$  wartościach, dlatego porządkowanie wartości krotek jest ważne. Jednakże, z logicznego punktu widzenia, porządek atrybutów i ich wartości nie ma większego znaczenia, o ile odniesienie pomiędzy atrybutami i ich wartościami jest zachowane.

### **Wartości krotek.**

Każda wartość krotki jest wartością atomową, co oznacza, że jest wartością niepodzielną. Dlatego w schemacie relacyjnym nie są dozwolone atrybuty wielowartościowe, bądź też złożone. Wartości takie muszą być reprezentowane jako osobne relacje.

### **Interpretacje relacji.**

Schemat relacyjny może być interpretowany jako deklaracja, bądź też typ przypisania. Niektóre schematy mogą oczywiście reprezentować fakty o encjach, tak samo jak inne schematy mogą reprezentować fakty o związkach (relationship).

Alternatywną reprezentacją schematu relacyjnego są tzw. predykaty. W tym przypadku wartości poszczególnych krotek są interpretowane jako wartości dające zadość pewnemu zbiorowi predykatów.

## **2.5.2. Ograniczenia modelu relacyjnego.**

W rozdziale tym opisze pewne ograniczenia nakładane na schemat relacyjny. Należą do nich ograniczenia związane z dziedziną atrybutów, kluczami, integralnością danych, oraz integralnością referencji.

### **2.5.2.1. Ograniczenia dziedziny atrybutów.**

Specyfikuje ona wartości atrybutów należących do pewnej dziedziny atrybutów. Precyzyjnie mówiąc każda wartość atomowa  $A$  musi być z określonej dziedziny atrybutów  $\text{dom}(A)$ .

### **2.5.2.2. Klucze.**

Relacja jest definiowana jako zbiór (kolekcja) krotek, co oznacza że każdy element w tym zbiorze da się rozróżnić (na podstawie definicji zbioru). Zatem nie istnieją dwie identyczne krotki zawierające identyczne wartości atrybutów. Dlatego też do rozróżniania krotek używa się pewnych podzbiorów atrybutów. Nazywamy je kluczami. Klucze te mają następującą własność, żaden z nich nie jest równy innemu. Zbiór kluczy nazywany jest superkluczem i każda relacja ma przynajmniej jeden superklucz. Klucze są zatem stosowane do identyfikacji poszczególnych krotek w relacji.

### **2.5.2.3. Ograniczenia związane z integralnością danych.**

Musimy sobie uświadomić, że baza danych zawiera wiele relacji, wraz z krotkami które są ze sobą powiązane w różny sposób.

Wiemy, że relacyjnym schematem bazy danych  $S$  nazywamy zbiór schematów  $S = \{R_1, R_2, \dots, R_m\}$ , oraz zbiór ograniczeń integralności  $IC$ . Zatem instancją relacyjnej bazy danych  $DB$  ze zbioru schematów  $S$  nazwiemy zbiór instancji relacji  $DB = \{r_1, r_2, \dots, r_m\}$  w którym każde  $r_i$  jest instancją  $R$  taką, że daje zadość ograniczeniom integralności założonym w zbiorze  $IC$ .

Ograniczenia nałożone na integralność danych w bazie danych mają oddziaływać na każdą instancję takiego schematu.

#### 2.5.2.4. Integralność referencji, integralność encji.

Zasada integralność encji stanowi o tym, że żaden klucz nie może mieć wartości null, dlatego że jest on stosowany do jednoznacznego rozróżnienia krotek.

Zasada integralności referencji mówi nam, o zachowaniu spójności pomiędzy krotkami dwóch relacji. Co znaczy ni mniej ni więcej tyle, że pojedyncza krotka jeśli jest związana z drugą relacją, to musi odnosić się do rzeczywistej istniejącej krotki w tej relacji.

W bazach danych występuje wiele relacji posiadających wiele ograniczeń na referencje. Aby w dokładnie sprecyzować te ograniczenia musimy w pełni zrozumieć rolę jaką pełnią poszczególne zbiory atrybutów w danej relacji.

#### 2.5.3. 12 reguł Codd'a.

Dwanaście reguł Codd'a brzmi następująco:

1. **Reguła Informacyjna.** Wszystkie informacje w relacyjnej bazie danych jest reprezentowana wprost i tylko w jeden sposób, jako wartości w tabelach.
2. **Reguła Gwarantowanego Dostępu.** Każda i wszystkie dane w relacyjnej bazie danych są logicznie dostępne poprzez nazwę tabeli, wartość klucza pierwotnego i nazwę kolumny.
3. **Reguła Systematycznego Traktowania Wartości Pustych.** Wartości puste (różne od pustego łańcucha znaków, łańcucha spacji i różne od zera lub innej liczby) są całkowicie wpierane przez relacyjny system zarządzania bazą danych dla reprezentacji braku informacji w sposób systematyczny (konsekwentny) niezależnie od typu danej.
4. **Reguła Organizacji Dostępu w Modelu Relacyjnym.** Opis bazy danych jest przedstawiany na poziomie logicznym w ten sam sposób jak dane, tak że upoważniony użytkownik może zastosować ten sam język zapytań tak w celu poznania opisu bazy jak i danych.
5. **Reguła Pełności Danych Podjęzka.** System relacyjny może wspierać wiele języków, jednakże musi istnieć przynajmniej jeden, którego instrukcje tworzą wyrażenia dla dobrze zdefiniowanej

składni w postaci łańcuchów znaków i są zdolne do pełnego wspierania następujących elementów: definicji danych, definicji perspektyw, manipulacji danymi (interaktywnie lub przez program), więzów integralności danych i zakresów transakcji (begin, commit i rollback).

6. **Reguła Przeglądania Modyfikacji.** Wszystkie modyfikacje działające na perspektywach muszą wykonywalne przez System Zarządzania Bazą Danych.
7. **Reguła Wysokiego Poziomu Wstawiania, Aktualizacji i Usuwania.** System musi wspierać zespół jednoczesnych działań takich jak wstawianie, aktualizacja i usuwanie danych.
8. **Reguła Fizycznej Niezależności Danych.** Programy aplikacyjne lub akcje wykonywane na terminalu pozostają logicznie nienaruszone w przypadku zmian dokonywanych w reprezentacji pamięci fizycznej lub metod dostępu.
9. **Reguła Logicznej Niezależności Danych.** Modyfikacje w logicznej strukturze bazy danych mogą być wykonywane bez wyrejestrowywania się istniejących użytkowników czy zamykania istniejących programów.
10. **Reguła Niezależności Integralności.** Ograniczenia integralności specyficzne dla konkretnej relacyjnej bazy danych muszą być definiowalne w podjęzyku relacyjnym i przechowywane w schemacie bazy a nie w programie aplikacyjnym. Minimum dwa ograniczenia integralności muszą być wpierane:
  - integralność encji: żaden z elementów składowych klucza pierwotnego nie może zawierać wartości pustej,
  - integralność referencyjna: dla każdej różnej, nie pustej wartości klucza obcego musi odpowiadać odpowiednia wartość klucza pierwotnego z tej samej domeny
11. **Reguła Niezależności Dystrybucji.** Niezależność dystrybucji wymusza, że użytkownicy nie powinni martwić się, kiedy baza danych jest dystrybuowana
12. **Reguła Braku Podwersji.** Dostęp na niskim poziomie albo na poziomie rekordu nie może być zdolny do naruszenia systemu, ominięcia reguł integralności lub ograniczeń zdefiniowanych na wyższych poziomach

Do 12 reguł istnieje uzupełnienie znane jako **Reguła zero**:

Dla każdego systemu, który uważany jest za relacyjny musi istnieć możliwość zarządzania danymi wyłącznie poprzez jego relacyjne możliwości.

Do tej pory nie zaimplementowano w pełni funkcjonalnego, wolnego od błędów systemu relacyjnego.

#### 2.5.4. Podsumowanie.

W rozdziale tym zaprezentowałem koncepcje tworzenia relacyjnej bazy danych, oraz nałożone na nią ograniczenia. Zdefiniowałem wiele pojęć które pokazały zasadę działania tej bazy z matematycznego punktu widzenia.

Na modelu relacyjnym oparta jest relacyjna baza danych (RDBMS *ang. Relational Database Management Systems*), w której dane są przedstawione w postaci relacyjnej. Relacja reprezentowana jest przez tablicę (tablica=relacja, stąd nazwa), które są pewnym zbiorem rekordów o identycznej strukturze i wewnętrznie powiązanych za pomocą związków zachodzących pomiędzy danymi. Powoduje to ułatwienie zarządzania bazą danych w stosunku do tradycyjnego podejścia, gdzie dane są przechowywane w postaci strumienia. Takie podejście ułatwia wprowadzania zmian, zmniejsza możliwość pomyłek, jednakowoż dzieje się to kosztem wydajności.

## **2.6. Model relacyjno-obiektowy.**

W ciągu ponad 30 lat od czasu wynalezienia i zaprezentowania przez Codd relacyjnego modelu danych, powstało wiele jego hybryd. Jedną z takich hybryd jest model obiektowo-relacyjny. Za powstaniem tego modelu nie stoi żadna spójna teoria, jednakże wygodnie jest go omawiać jako pewien zbiór mechanizmów stworzonych do operowania na danych. Model obiektowo-relacyjny powstał, gdyż początkowo komputery nie były w stanie sprostać wymaganiom stawianym relacyjnym systemom baz danych. Gdy technologia się rozwinęła, twórcy komercyjnego oprogramowania nie chcieli już rezygnować z pewnych udogodnień systemów hybrydowych.

Systemy relacyjno-obiektowe łączą w sobie zarówno próbę wykorzystania mocnych stron modelu relacyjnego jak i obiektowego do manipulacji danymi.

### **2.6.1. Definicja danych.**

Aby poprawić obiektowość relacyjnego typu danych dodano do niego koncepcję abstrakcyjnego typu danych, oraz relacji zagnieżdżonej.

#### **2.6.1.1. Abstrakcyjne typy danych (ang. ADT).**

Większość istniejących relacyjnych systemów baz danych posiada ograniczoną ilość typów danych, którą może zaoferować użytkownikowi. Wymusza to potrzebę stworzenia nowych typów danych. Daje to użytkownikowi wolną rękę w tworzeniu i definiowaniu dodatkowych abstrakcyjnych typów danych.

Abstrakcyjny typ danych to pewien typ obiektu definiujący dziedzinę wartości oraz zbiór operacji potrzebnych do pracy na tych wartościach. Celem ADT jest tzw. "ukrywanie informacji", co oznacza, że szczegóły jego implementacji są niewidoczne dla wyższych poziomów systemu użytkownika. Jeśli nastąpi zmiana realizacji ADT nie będzie to wpływało na zmianę wyższych warstw systemu.

#### **2.6.1.2. Zagnieżdżone relacje.**

Zwane są one inaczej relacjami w nie pierwszej postaci normalnej (ang. NF 2). Zostały one zaprojektowane do obsługi złożonych obiektów. Złożone obiekty można zdefiniować jako hierarchiczne, których atrybuty mogą być atomowe, bądź złożone. Zatem obiekt może posiadać swoją wewnętrzną złożoność, którą chcemy zdefiniować. Taka wewnętrzna złożoność to nic innego

jak relacja zagnieżdżona. Analogicznie do zwykłych relacji, relacje zagnieżdżone posiadają również listę nazw kolumn. Trzeba jednak pamiętać, że nazwa kolumny w zagnieżdżonej relacji może odnosić się do grupy atrybutów, który może odnosić się do kolejnej grupy, a te do kolejnej itd.

Do pracy na takich relacjach potrzebne są nam dwie wewnętrzne operacje: zagnieżdżenie (nest), oraz rozgnieżdżenie (unnest). Zagnieżdżenie pobiera relację, oraz nazwę atrybutu i zwraca na ich podstawie zagnieżdżoną relację. Rozgnieżdżenie stanowi operację odwrotną i zwraca na podstawie zagnieżdżonej relacji jej relację pierwotną.

### 2.6.1.3. Mechanizmy przechowywania danych.

W obecnie istniejących systemach zarządzania bazami danych istnieją cztery podstawowe mechanizmy przechowywania danych. Są to:

- pliki sekwencyjne
- pliki indeksowo-sekwencyjne,
- pliki haszowane,
- pliki klastrowe.

#### **Pliki sekwencyjne.**

Stosowane są one dla:

- małych tabel
- średnich i dużych, ale w przypadku gdy jest konieczność dostępu do więcej niż 20% wierszy.
- Dowolnych tabel – gdy dostęp do nich jest poprzez zapytanie o niskim priorytecie, bądź też jeśli istnieje potrzeba ich zrealizowania za pomocą przetwarzania wsadowego.

Innym dość istotnym zastosowaniem plików sekwencyjnych jest ładowanie tabel przed ich indeksowaniem.

#### **Pliki indeksowo-sekwencyjne.**

Są bardziej uniwersalną strukturą, ponieważ obsługują wyszukiwanie na podstawie dokładnej zgodności z kluczem, bądź też zgodności ze wzorcem, oraz według zakresu.

Efektywność wyszukiwania w takim pliku pogarsza się gdy nastąpi jego modyfikacja. Związane jest to z tym, że indeks jest tworzony podczas powstawania tego pliku, a potem nie następuje jego modyfikacja.

#### **Pliki haszowane.**

Są najpowszechniej wykorzystywane jeśli chodzi o metodę dostępu do danych za pomocą jednej wartości klucza. Jest on stosowany w takiej sytuacji do budowania głównej ścieżki dostępu do pliku

partej na kluczach głównych.

### **Pliki klastrowe.**

Budowa klastrów ma na celu wymuszenie wykonania pewnych wstępnych złączeń na danych. Porównując to rozwiązanie z innymi, wstępnie złączone pliki klastrowe dają mniejszą efektywność. Ze względu na swoją strukturę mogą wręcz pogorszyć wykonanie aktualizacji

#### 2.6.1.4. Indeksowanie danych

Indeks służy do poprawienia szybkości dostępu do danych, bez zmiany struktury ich przechowywania. Wyróżniamy dwa podstawowe typy indeksów:

- główne,
- wtórne.

Jako absolutne minimum powinno się zakładać istnienie indeksów na kluczach głównych i obcych. Na kluczach głównych takie indeksy nazywamy jednoznacznymi. Wiele relacyjnych systemów zarządzania bazami danych robi to indeksowanie automatycznie.

Teoretycznie istnieje możliwość stworzenia indeksów na wszystkich elementach danej tabeli, jednak w praktyce robi się to rzadko. Najistotniejszym powodem tego jest, że pomimo indeksowanie może znacznie przyspieszyć dostęp do danych, to ma ona negatywny wpływ na wydajność modyfikacji.

Ogólną zasadą jest tworzenie indeksów na średnich i dużych tabelach w celu ułatwienia dostępu do małego procentu wierszy tabeli.

#### 2.6.2. Podsumowanie.

Obiektowo-relacyjny model danych próbuje dodać obiektowości do tablic. Dane są wciąż przechowywane w tabelach, jednak wartości mogą mieć nieco bogatsza niż dotychczas postać - ADT (Abstract Data Type). Pola typu ADT zachowują funkcjonalność zwykłych pól (mogą być używane do indeksowania, wyszukiwania, pobierania lub umieszczania danych) przy nowych zawartościach (jak np. Multimedia). Zatem obiektowo-relacyjny model danych korzysta w dużej mierze ze standardu SQL3.

## **2.7. Model obiektowy ODMG.**

Obowiązujący obecnie standard, opracowany przez ODMG, został opublikowany w roku 1993. Jednym z podstawowych celów modelu obiektowego jest bezpośrednio odwzorowanie obiektów i powiązań między nimi wchodzących w skład aplikacji na zbiór obiektów i powiązań w bazie danych. Dzięki mechanizmom obiektowym można też zwiększyć niezależność danych od aplikacji poprzez przeniesienie procedur obsługi danych (w postaci metod) do systemu zarządzania bazą.

W raporcie OVUM wydanym w 1988 r. zapowiadano, że systemy zarządzania bazami danych oparte na modelu obiektowym, który jest istotnie różny od modelu relacyjnego prześcigną systemy oparte na modelu relacyjnym. Oczywiście tak jednak się nie stało. Nie mamy jednak wątpliwości, że modele obiektowe wywierają duży wpływ na tworzenie i rozwój systemów informatycznych.

We współczesnej informatyce pojęcie obiektowości ma wiele różnych znaczeń.

Termin ten był po raz pierwszy zastosowany w odniesieniu do grupy języków programowania wywodzących się z języka pochodzenia skandynawskiego, znanego jako Simula. Język Simula był pierwszym językiem, który wprowadził pojęcie abstrakcyjnego typu danych jako zintegrowanego pakietu struktur danych i procedur.

Główną różnicą pomiędzy obiektowymi językami programowania, a bazami danych jest to, że obiektowe bazy danych wymagają obiektów trwałych. Obiekty te pozostają zapisane w pamięci pomocniczej przed i po wykonaniu programów.

### **2.7.1. Dlaczego zaistniała potrzeba stworzenia obiektowego modelu danych.**

Relacyjny model ma pewne słabości. Nie zapewnia dobrej reprezentacji “świata rzeczywistego”. Można się nawet pokusić o stwierdzenie, że reprezentacja ta jest słaba. Wynika to z podziału encji lub klas na liczne relacje przez proces normalizacyjny.

Model relacyjny cierpi także na semantyczne przeładowanie. Wynika to z tego, że model danych stosuje jedną konstrukcję dla związków i encji. Relacyjny model ma także trudności z radzeniem sobie ze złożonymi obiektami. Wiąże się to z ograniczeniem relacyjnym do wartości atomowych, a co za tym idzie struktury hierarchiczne, bądź też zagnieżdżone nie są obsługiwane w łatwy sposób.

### **2.7.2. Standard ODMG.**

Nad opracowaniem standardu obiektowego systemu zarządzania bazami danych (OODBMS) zebrali się kilku dostawców oprogramowania tworząc tzw. Object Database Management Group (ODMG). Grupa ta określiła standardy składniowe i semantyczne które pozwalały na przenośność

między różnymi implementacjami ODMG. Pierwsza wersja standardu powstała w roku 1993, natomiast druga w roku 1997.

#### 2.7.2.1. Manifest systemów zarządzania obiektową bazą danych.

Manifest ten proponuje 13 obowiązkowych cech które powinien zawierać taki system. Pierwszych osiem punktów traktuje o zgodności OODBMS z zasadami obiektowości. Ostatnich pięć punktów określa, że OODBMS musi posiadać i obsługiwać kilka cech klasycznego DBMS.

1. Obsługa złożonych obiektów. Musi być możliwość budowania złożonych obiektów przez zastosowanie konstruktorów do obiektów podstawowych takich jak zbiór, krotka i lista.
2. Obsługa identyfikatorów obiektów. Wszystkie obiekty muszą mieć unikatowe identyfikatory, niezależne od wartości ich atrybutów.
3. Obsługa hermetyzacji. Hermetyzacja wymaga, aby programiści mieli dostęp do obiektów poprzez zdefiniowane interfejsy, co nie oznacza, że wiele działających w praktyce systemów zarządzania stawia hermetyzację na pierwszym miejscu. Niektóre wręcz wykorzystują potrzebą zapytań ad-hoc.
4. Obsługa klas obiektów. Schemat w OODBMS musi zawierać zbiór klas.
5. Obsługa dziedziczenia i atrybutów.
6. Obsługa dynamicznego wiązania. Polega to na tym, że DBMS musi wiązać nazwy metod z logiką podczas wykonywania, aby móc obsługiwać nadpisywanie.
7. DML musi być obliczeniowo kompletny, co oznacza, że powinien być językiem programowania ogólnego przeznaczenia.
8. Zbiór typów danych musi być rozszerzalny. Użytkownik musi mieć możliwość budowania nowych typów na podstawie typów zdefiniowanych.
9. Musi być zapewniona trwałość danych. Dane muszą trwać nie tylko podczas działania aplikacji i pracy na danych. Muszą także trwać po zakończeniu działania aplikacji, a Użytkownik nie powinien w sposób jawny wymuszać ich trwałości.
10. DBMS musi być zdolny do zarządzania bardzo dużymi bazami danych, dlatego też OODBMS musi posiadać mechanizmy umożliwiające efektywne działanie i zarządzanie pamięcią pomocniczą (zarządzanie indeksami).
11. DBMS musi zapewnić współbieżny dostęp do danych.
12. DBMS musi mieć możliwość odtworzenia danych po awarii sprzętu, bądź też oprogramowania.
13. DBMS musi zapewnić prosty sposób wyszukiwania danych.

#### 2.7.2.2. Składowe standardu ODMG.

Standard ODMG zawiera cztery podstawowe (główne) elementy:

1. Model obiektowy (OM).
2. Język definiowania obiektów (ODL).
3. Język wyszukiwania obiektów (OQL).
4. Kilka rozwiązań językowych głównie w takich językach jak Java, C++, Smalltalk.

### 2.7.2.3. Model obiektowy.

Definiuje on następujące konstrukcje modelowania. Podstawowymi elementami są obiekty i literały. Te natomiast klasyfikowane są według typów. Zachowanie obiektu jest zdeterminowane przez zbiór pewnych metod związanych z tym obiektem. Każdy obiekt posiada swoje właściwości. Mogą być nimi jego atrybuty, bądź też związki pomiędzy obiektami.

#### **Obiekty.**

Są one definiowane na hierarchii typów zawierającej takie typy abstrakcyjne jak:

- obiekty atomowe,
- obiekty zbiorowe,
- obiekty strukturalne.

W ramach obiektów strukturalnych zdefiniowanych jest kilka typów jednostkowych, czyli takich, które mogą być użyte jako typy podstawowe w bazie danych.

#### **Literały.**

W ramach zbioru literałów znajdują się typ atomowy, zbiorowy, strukturalny, bądź też null. Wartość właściwości literału nie może ulegać zmianie, dlatego też mogą być one używane jako tradycyjne typy danych. Do literałów atomowych zaliczamy np.: integer, char, string, natomiast do literałów strukturalnych np.: datę.

#### **Kolekcje.**

Mogą być literałami bądź obiektami zbiorowymi. Model obiektowy definiuje pięć typów obiektów zbiorowych:

1. zbiór,
2. wielozbiór,
3. lista,
4. tablica,
5. słownik.

**Typy i klasy.**

Typy posiadają jedną specyfikację i jedną implementację. Kombinacja specyfikacji i jednej implementacji nosi nazwę klasy.

**Właściwości.**

W modelu obiektowym mamy dwa podstawowe typy właściwości: relacje i atrybuty. Atrybut jest zdefiniowany na jednym typie obiektu i przyjmuje jako wartość identyfikator obiektu, bądź też literał. Relacje są definiowane pomiędzy określonymi typami i posiadają swoją liczebność.

**Operacje.**

Każda instancja obiektu zachowuje się w określony sposób w zbiorze operacji. Definicja typu obiektu determinuje sygnaturę dla operacji, która zawiera nazwę operacji, parametry i typ każdego parametru, oraz typ zwracanych wartości. Operacja może zwrócić obiekt jako wartość.

## **2.8. Model składu obiektów prof. K. Subiety.**

Istniejące modele obiektowe, w szczególności model ODMG są bardzo złożone. Posiadają one wiele pojęć takich jak: obiekty, literały, typy, interfejsy, dziedziczenie, kolekcje. Jeszcze bardziej złożony jest typ danych SQL-99, ponieważ dokłada on do tego wszystkiego relacje i pojęcie abstrakcyjnych typów danych (ADT).

Dlatego też istnieje potrzeba uproszczenia modeli obiektowych, lub też taka ich abstrakcja, która z jednej strony była by formalnie prosta, z drugiej strony pozwalała na wierne odwzorowanie modeli tworzonych przez praktyków. Właśnie dlatego prostota modelu relacyjnego stała się źródłem jego sukcesu. Jednakże profesor Subieta wyraźnie odrzuca algebry relacyjne, oraz dotychczasowe próby stworzenia algebry obiektowej. Uważa je za błąd w rozumieniu koncepcji języków zapytań oraz samej koncepcji baz danych.

W odróżnieniu od modelu relacyjnego, model obiektowy zawiera o wiele więcej pojęć. Poza tym istnieje w nim różne rozumienie tych samych pojęć (jako przykład można podać pojęci klasy). Zatem trudno zbudować jeden słuszny model, który jednocześnie łączyłby wszystkie sytuacje, które mogą się wydarzyć.

Model swojej obiektowej bazy danych opieram na pojęciach wprowadzonych przez profesora Kazimierza Subietę. Wprowadził on w swojej pracy dotyczącej budowy obiektowych języków zapytań pewien podział, na tak zwane modele składu obiektów w obiektowej bazie danych.

- Model M0: obejmuje on dowolne powiązania hierarchiczne struktur danych. Nie zawiera w sobie takich pojęć jak: klasa, dziedziczenie, interfejs, hermetyzacja. Pozwala on natomiast wyjaśnić semantykę relacyjnych języków zapytań, takich jak np.: SQL.
- Model M1: uzupełnia M0 o pojęcie klasy, dziedziczenia i wielodziedziczenia, natomiast nie obejmuje jeszcze interfejsu i hermetyzacji.
- Model M2: uzupełnia model M1 wprowadzając dziedziczenie i dynamiczne role. Jest on w pewnym sensie modyfikacja modelu M1.
- Model M3: uzupełnia model M1 lub M2 o pojęcie hermetyzacji (podział własności klas i obiektów na prywatne i publiczne)

Oczywiście podane wyżej modele nie odzwierciedlają w pełni możliwości obiektowych baz danych, ponieważ mogą oczywiście istnieć sytuacje, które nie zostały ujęte powyższymi modelami. Inne podejścia do języków zapytań nie różnicują podejścia do składu, oraz jego modelu, co za tym idzie ograniczają się do jednego z nich. Takie podejście do zagadnienia implikuje powstawanie wielu nieuzasadnionych ograniczeń semantycznych. Ograniczenia te następnie owocują w niezgodności natury formalnej, która uniemożliwia precyzyjną specyfikację wielu rozwiązań formalnych.

### 2.8.1. Opis modeli składu obiektów.

W opisie wyróżniamy trzy główne pojęcia:

- wewnętrzny identyfikator obiektu. Nadawany jest bezpośrednio przez system i nie posiada żadnej semantyki w świecie zewnętrznym. Dzięki niemu jesteśmy w stanie identyfikować obiekty znajdujące się wewnątrz pamięci komputera. Zbiór wszystkich identyfikatorów oznaczać będą przez  $I$ .
- zewnętrzna nazwa obiektu. Jest ona nadawana przez projektanta bazy, programistę, bądź też administratora. Powiązana jest ściśle z modelem koncepcyjnym lub biznesowym aplikacji działających na bazie danych. Przykładową taką nazwą może być osoba lub nazwisko. Zewnętrzna nazwa obiektu nie musi być i zwykle nie jest unikalna. Zbiór wszystkich zewnętrznych nazw będą oznaczał przez  $N$ .
- wartość atomowa. Jest to niepodzielna, nie ma żadnych wartości składowych. Zbiór wszystkich wartości atomowych będą oznaczał przez  $V$ .

Identyfikatory będą oznaczał małą literką  $i$ , a nazwy literką  $n$ , natomiast wartości będą oznaczał literką  $v$  (każde z tych oznaczeń będzie uzupełniane w razie potrzeby odpowiednim indeksem).

#### 2.8.1.1. Model składu $M_0$ .

Każdy obiekt posiada unikalny identyfikator zatem można go zidentyfikować jako trójkę postaci  $\langle i, n, v \rangle$ - jeśli jest to obiekt atomowy,  $\langle i_1, n, i_2 \rangle$  - jeśli jest to obiekt pointerowy, oraz  $\langle i, n, T \rangle$ - jeśli jest to obiekt złożony. W obiekcie złożonym zbiór  $T$  jest zbiorem dowolnych obiektów.

Definicja modelu  $M_0$  według profesora Subiety ma postać:

*“W modelu  $M_0$  skład obiektów jest zdefiniowany jako para  $\langle S, R \rangle$ , gdzie  $S$  jest zbiorem obiektów, zaś  $R$  jest zbiorem identyfikatorów określanych przez nas jako “identyfikatory startowe”.”*

Zbiór  $R$  wyznacza punkty wejściowe do składu obiektów, czyli takich miejsc które mogą być początkiem poruszania się po składzie danych.

Przykład:

**S** - Obiekty:

```

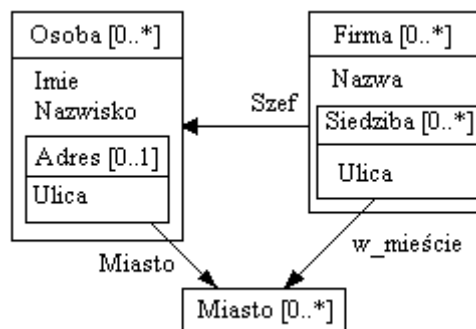
<i1, miasto, "Toruń">
<i2, miasto, "Warszawa">
<i3, osoba, { <i4, imie, "Jan">,
               <i5, nazwisko, "Nowak">,
               <i6, adres, {
                   <i7, miasto, i1>,
                   <i8, ulica, „Szkolniana 12/1”> } > }>
<i9, osoba, { <i10, imie, "Piotr">,
               <i11, nazwisko, "Kowalski">,
               <i12, adres, {
                   <i13, miasto, i2>,
                   <i14, ulica, „Polna 7”> } > }>
<i15, osoba, { <i16, imie, "Anna">,
               <i17, nazwisko, "Pietrzak">,
               <i18, adres, {
                   <i19, miasto, i2>,
                   <i20, ulica, „Różana 102/12”> } > }>
<i21, firma, { <i22, nazwa, "InfoSpex">,
               <i23, siedziba, {
                   <i24, ulica, „Aroniowa 5”>
                   <i25, w_mieście, i1> } >
               <i26, szef, i4> } >
<i27, firma, { <i28, nazwa, "Escal">,
               <i29, siedziba, {
                   <i30, ulica, „Prosta 15/2A”>
                   <i31, w_mieście, i2> } >
               <i32, szef, i15> } >

```

Identyfikatory startowe:  
**R** = {i<sub>1</sub>, i<sub>2</sub>, i<sub>3</sub>, i<sub>9</sub>, i<sub>15</sub>, i<sub>21</sub>, i<sub>27</sub>}

**Rys. 2.8.1.1.** Mała baza danych w modelu M0

Skład obiektów podlega pewnym ograniczeniom. Jak już wspomniałem wcześniej podstawowym ograniczeniem jest to, że każdy obiekt w składzie powinien posiadać unikatowy identyfikator. Kolejne ograniczenie dotyczy obiektów pointerowych. Ponieważ jeśli istnieje obiekt pointerowy postaci  $\langle i_1, n, i_2 \rangle$  to implikuje istnienie obiektu o identyfikatorze  $i_2$ . Warunek ten gwarantuje integralność referencyjną, a co za tym idzie brak zwisających pointerów.



### Rys. 2.8.1.2. Diagram klas z rysunku 2.8.1.1.

Kolejnym istotnym ograniczeniem jest to, że dowolny identyfikator ze zbioru R jest identyfikatorem pewnego istniejącego obiektu znajdującego się w składzie. Obiekty w składzie mogą być osiągnane bezpośrednio, bądź też pośrednio.

Obiekty osiągnane bezpośrednio to obiekty których identyfikator znajduje się w zbiorze R.

Przykład identyfikatorów pośrednich i bezpośrednich:

$\langle i_1, \text{Student}, \{ \langle i_2, \text{nazwisko}, \text{"Nowak"} \rangle, \langle i_3, \text{imie}, \text{"Jarosław"} \rangle, \langle i_4, \text{nr\_indeksu}, 123456 \rangle, \langle i_5, \text{na\_kierunku}, i_{55} \rangle \} \rangle$

Na podstawie tego przykładu jesteśmy w stanie odczytać, że obiekt Student o identyfikatorze startowym  $i_1$  (identyfikator osiągalny bezpośrednio) ma cztery podobiekty o identyfikatorach  $i_2, i_3, i_4, i_5$  (osiągnane pośrednio poprzez identyfikator  $i_1$ ).

### 2.8.1.2. Model składu M1.

W modelu M1 poszerzamy model M0 o klasy i dziedziczenie. Definicja klasy według profesora Subiety:

*“Z formalnego punktu widzenia klasa jest obiektem podobnym do wprowadzonych poprzednio obiektów. Jest przechowywana w składzie obiektów.”*

Czyli obiekty ze składu M0 które przechowują klasy będą wyróżnione jako te zawierające inwarianty innych obiektów. Ta rola jest niezwykle przydatna z punktu widzenia języka. Istotną zmianą dla składu jest wprowadzenie specjalnych powiązań, które mają odwzorować relacje dziedziczenia. Powiązania te są modelowane jako dwie relacje binarne zdefiniowane na obiektach.

**S** – Obiekty i klasy

```

<i3, Osoba, { <i4, Imię, „Jan”>,
               <i5, Nazwisko, „Nowak”>,
               <i6, Adres, {
                   <i7, Miasto, „Toruń”>,
                   <i8, Ulica, „Szklarniana 12/1”> } > }>
<i9, Prac, { <i10, Imię, „Szymon”>,
             <i11, Nazwisko, „Malinowski”>,
             <i12, Zarobek, 2500>,
             <i13, Adres, {
                 <i14, Miasto, „Bydgoszcz”>,
                 <i15, Ulica, „Krótka 7”> } > }>
<i16, Prac, { <i17, Imię, „Zbigniew”>,
             <i18, Nazwisko, „Lipski”>,
             <i19, Zarobek, 2000>,
             <i20, Adres, {
                 <i21, Miasto, „Toruń”>,
                 <i22, Ulica, „Winna 29/12”> } > }>
<i50, KlasaOsoba, {
    <i51, Kod_pocztowy, (...Kod metody Kod_pocztowy)>,
    pola stałe: Nazwa obiektów = „Osoba”,
    pozostałe stałe klasy KlasaOsoba }>
<i60, KlasaPracownik, {
    <i61, Zar_Netto, (...Kod metody Zar_Netto)>,
    <i62, ZmienZar, (...Kod metody ZmienZar)>,
    pola stałe: Nazwa obiektów = „Prac”,
    pozostałe stałe klasy KlasaPracownik }>

```

Identyfikatory startowe:  
**R** = { i<sub>3</sub>, i<sub>9</sub>, i<sub>16</sub> }

Związki dziedziczenia między obiektami:  
**KK** = { < i<sub>50</sub>, i<sub>60</sub> > }

Związki dziedziczenia między obiektami i klasami:  
**OK** = { < i<sub>3</sub>, i<sub>50</sub> >, < i<sub>9</sub>, i<sub>60</sub> >, < i<sub>16</sub>, i<sub>60</sub> > }

**Rys. 2.8.1.3.** Skład obiektów w modelu M1 wraz ze związkami dziedziczenia

Zatem skład w modelu M1 możemy przedstawić jako czwórkę <S, R, KK, OK>.

Zbiór S jest zbiorem obiektów rozszerzonym o specjalne obiekty zwane klasami. W składzie nie różnią się one budową od zwykłych obiektów. Zawierają jedynie informacje (inwariancje) do innych obiektów. R jest zbiorem identyfikatorów startowych. Relacja KK: I x I wyznacza związki dziedziczenia między klasami. Natomiast relacja OK: I x I wyznacza przynależność obiektów do klas.

### 2.8.1.3. Model M2 - modelowanie dynamicznych ról.

Model M2 ma poszerzyć model M1 o wprowadzenie koncepcji dynamicznych ról. Definicja modelu M2 według profesora Subiety:

*“Model M2 jest uporządkowaną piątką  $\langle S, R, KK, OK, OO \rangle$ , gdzie nowa relacja  $OO$   $I \times I$  wyraża dziedziczenie pomiędzy obiektami.”*

Relacja ta posiada podwójną funkcję. Z jednej strony pozwala obiektom dziedziczyć z innych obiektów, na takiej samej zasadzie jak obiekty dziedziczą z klas. Obiekty które dziedziczą z obiektu a nazywamy jego rolami. Istnieje możliwość, że dany obiekt może być dla siebie rolą. Relacja OO ustala także semantykę manipulacji obiektami z dynamicznymi rolami. Co oznacza, że np.: usunięcie obiektu spowoduje usunięcie wszystkich jego ról.

```
S – Obiekty i klasy
<i3, Osoba, { <i4, Rok_ur, 1954>, <i5, Nazwisko, “Nowak”> }>
<i6, Osoba, { <i7, Rok_ur, 1980>, <i8, Nazwisko, “Kowalski”> }>
<i9, Prac, { <i10, Pracuje_w, i27>, <i11, Zarobek, 2500> }>
<i12, Student, { <i10, Wydział, i34>, <i12, Nr_Indeksu, 12 110> }>
<i50, KlasaOsoba, {
    <i51, Kod_pocztowy, (...Kod metody Kod_pocztowy)>,
    pozostałe stałe klasy KlasaOsoba }>
<i60, KlasaPracownik, {
    <i61, Zar_Netto, (...Kod metody Zar_Netto)>,
    <i62, ZmienZar, (...Kod metody ZmienZar)>,
    pozostałe stałe klasy KlasaPracownik }>
<i70, KlasaStudent, {
    <i71, Srednia, (...Kod metody Srednia)>,
    pozostałe stałe klasy KlasaStudent }>

Identyfikatory startowe:
R = { i3, i6, i9, i12 }
Związki dziedziczenia między obiektami:
KK = ∅
Związki dziedziczenia między obiektami i klasami:
OK = { < i3, i50 >, < i6, i50 >, < i9, i60 >, < i12, i70 > }
Związki dziedziczenia między obiektami i obiektami:
OO = { < i9, i3 >, < i12, i6 > }
```

Rys. 2.8.1.4. Model M2 – modelowanie dynamicznych ról

### 2.8.1.4. Model M3 – hermetyzacja i ukrywanie informacji.

Model ten możemy zbudować zarówno w oparciu o model M1 jak i model M2, ponieważ cecha hermetyzacji jest ortogonalna w stosunku do wprowadzonych wcześniej własności. Idea

hermetyzacji polega na tym, aby w określonych sytuacjach zabronić dostępu do pewnych własności obiektów, określanych jako “prywatne”, co nie oznacza, że takie własności nie mają być w ogóle niedostępne.. Mają być dostępne ale jedynie z tzw. wnętrza obiektów, a nie z “zewnątrza”.

*“Model M3 uzupełnia model M1 lub M2 w taki sposób, że każda klasa zostaje wyposażona w dodatkowy inwariant zwany lista eksportową. Jest ona zbiorem nazw własności tej klasy (w szczególności metod) oraz nazw własności jej obiektów (w szczególności atrybutów), które będą widoczne zewnątrz.”*

```

S – Obiekty i klasy
<i3, Osoba, { <i4, Rok_ur, 1956>, <i5, Nazwisko, “Nowak”> }>
<i9, Prac, { <i10, Imię, “Szymon”>,
             <i11, Nazwisko, “Malinowski”8>,
             <i12, Zarobek, 2500> }>
<i16, Prac, { <i17, Imię, “Zbigniew”>,
             <i18, Nazwisko, “Lipski”>,
             <i19, Zarobek, 2000> }>
<i50, KlasaOsoba, {
    <i51, Wiek, (...Kod metody Wiek)>,
    inwariant: Nazwa obiektów = „Osoba”,
    inwariant: Lista eksportowa = { „Nazwisko”, „Wiek” }
    pozostałe inwarianty klasy KlasaOsoba }>
<i60, KlasaPracownik, {
    <i61, Zar_Netto, (...Kod metody Zar_Netto)>,
    <i62, ZmienZar, (...Kod metody ZmienZar)>,
    inwariant: Nazwa obiektów = „Prac”,
    inwariant: Lista eksportowa = { „ZmienZar”, „Zar_Netto” }
    pozostałe inwarianty klasy KlasaPracownik }>

Identyfikatory startowe:
R = { i3, i9, i16 }
Związki dziedziczenia między obiektami:
KK = { < i50, i60 > }
Związki dziedziczenia między obiektami i klasami:
OK = { < i3, i50 >, < i9, i60 >, < i16, i60 > }

```

**Rys. 2.8.1.5.** Model M3 oparty o M1 – uzupełnienie M1 o listy eksportowe

#### 2.8.1.5. Schemat bazy danych dla modeli składu danych.

Schemat bazy danych jest jej obrazem wyrażonym w pewnym sformalizowanym języku.

*“Schemat ogranicza zawartość bazy danych. Formalnie, semantyka schematu bazy danych jest zbiór wszystkich jej dopuszczalnych stanów.”*

### 2.8.2. Podsumowanie.

Obiektowa baza danych przechowuje obiekty w odróżnieniu od wierszy lub krotek przechowywanych w relacyjnych bazach danych. Ponieważ dane przechowywane są w postaci obiektów, mogą być odczytywane tylko przy pomocy metod udostępnianych przez te obiekty.

Obiekty przechowywane w takiej bazie danych są widoczne jako obiekty języka programowania. Ta właściwość nazywana jest transparentną trwałością (ang. transparent persistence).

W połączeniu z obiektowymi językami programowania, obiektowe bazy danych działają szybciej od baz relacyjnych, ponieważ nie ma potrzeby przemapowywania rekordów przechowywanych w tabelach na obiekty (ang. impedance mismatch).

Obiektowe bazy danych rozszerzają obiektowe języki programowania o funkcjonalność zarządzania wielowątkowością, obiektowy język zapytań, funkcje odzyskiwania danych.

### **3. Wykorzystane technologie.**

#### **3.1. Java**

Jest obiektywnym językiem programowania stworzonym przez grupę ludzi z firmy Sun Microsystems pod kierunkiem Jamesa Goslinga. Jest językiem kompilowanym do postaci wykonywanej przez Maszynę wirtualną Java (JVM). Założenia jakie postawiono sobie podczas projektowania tego języka zaczerpnięte zostały z języka Smalltalk oraz C++. Java odziedziczyła ze Smalltalka koncepcję maszyny wirtualnej oraz garbage collector'a natomiast z języka C++ została zaczerpnięta większość składni i słów kluczowych.

Autorzy języka Java określili kilka podstawowych założeń na budowę tego języka, w śród których można wymienić obiektywość, niezależność od architektury, niezawodność, sieciowość i obsługę programowania rozproszonego, oraz bezpieczeństwo.

Podczas tworzenia systemu YODA oparłem się na maszynie wirtualnej Java w wersji 1.5., jednakże system jest zdolny uruchomić się nawet na starszych wersjach maszyny wirtualnej Java. Testowano jego uruchomienie na maszynie Java 1.4.2 i nie stwierdzono żadnych problemów z funkcjonalnością oraz funkcjonowaniem systemu.

#### **Java API for XML Processing.**

W systemie YODA wykorzystano interfejs języka Java (wchodzący w skład pakietu org.w3c.dom) do obsługi dokumentów XML. Składnik Document Object Model Level 2 Core API pozwala nam na dynamiczny dostęp i modyfikacje zawartości dokumentu XML. Definiuje on także w pewien sposób strukturę dokumentu. Dzięki temu pakietowi jesteśmy w stanie przebudować, zmienić usunąć dane zawarte w pliku XML, stosując bardzo proste metody manipulowania danymi. W pracy wykorzystano cztery klasy do operowania na danych w XML-u: Document, Element, Node, NodeList.

#### **Java LinkedList.**

LinkedList jest implementacją zwykłej listy występującej w Javie. LinkedList implementuje wszystkie dodatkowe opcje listy i pozwala na dodanie każdego elementu (włączając w to element null). Wprowadza ona także proste w obsłudze metody do dodawania, usuwania i modyfikacji każdego obiektu znajdującego się na niej. Dzięki tym operacją możemy jej używać jako stosu, kolejki, bądź też podwójnej kolejki.

#### **3.2. XML - Extensible Markup Language.**

Jest to system składni uniwersalnego języka znakujący dane poprzez tagi i atrybuty.

XML powstał jako kolejna generacja standardu po poprzednikach: GML-u (General Markup Language), SGML-u (Structure General Markup Language ISO 8879:1986) zorientowanych do

znakowania tekstu. Obecnie jest wykorzystywany do szerokiego opisu struktur danych, w tym także danych tekstowych, bądź też tworzenia struktur bazodanowych. Formalizm okazał się na tyle silny, że obecnie pierwotne zastosowanie mark-up tekstu jest tylko jednym z możliwych zastosowań. Pierwotnym przeznaczeniem plików XML było ich wykorzystanie do współdzielenia danych na wielu platformach, przy różnych systemach operacyjnych.

XML jest żywym przykładem implementacji modelu hierarchicznego.

### 3.2.1. Modelowanie plików XML.

W swoich początkach postrzegany był jako “lepszy” HTML. Jego zadaniem była strukturalizacja informacji w sieci Web, czyli oddzielenie semantyki danych od formy w jakiej się wyświetlają w przeglądarce. Następnie pliki te wykorzystywano do robienia odpowiednich zestawień giełdowych, modelowania i układania rozkładów lotniczych. Ostatnie nastawienie społeczności informatycznej do wykorzystania XML-a do tworzenia struktur bazodanowych.

XML to język znaczników umożliwiający, podobnie jak SGML, tworzenie swoich własnych znaczników formatujących, definiowanych w DTD dokumencie lub w tzw. schematach XML. W odróżnieniu od SGML-a możliwe jest jednak także stosowanie w XML-u kaskadowych arkuszy stylów CSS, programowalnych arkuszy stylów specjalnie zaprojektowanych dla XML-a o nazwie XSL i innych interaktywnych elementów często stosowanych przy pisaniu stron sieci Web. XML wymaga znacznie większej dyscypliny przy pisaniu dokumentów niż HTML, gdyż zasadą interpretacji XML-a jest najpierw kontrola poprawności składniowej, a dopiero później ew. wyświetlenie/wykonanie dokumentu, tak więc błędnie napisane dokumenty XML nie będą w ogóle wyświetlane przez przeglądarkę. Z drugiej jednak strony, dzięki restrykcyjnej składni, dokumenty XML mogą być automatycznie przekształcane na inne formaty języków za pomocą parserów.

Przykład zawartości dokumentu XML:

```
<toc label="Platform Plug-in Developer Guide">
  <topic label="Programmer's Guide">
    <link> topics_Guide.xml </link/>
  </topic>
  <topic label="Reference">
    <link> topics_Reference.xml </link/>
  </topic>
  <topic label="3.0 Plug-in Migration Guide">
    <link> topics_Porting.xml </link/>
  </topic>
</toc>
```

```
</topic>
<topic label="Examples Guide">
    <link> topics_Samples.xml </link/>
</topic>
<topic label="Index" href="index.html">
    <link> Index.xml </link/>
</topic>
<topic label="Legal" href="notices.html"/>
</toc>
```

### 3.3.2. DTD – Document Type Definition.

Jest to schemat według którego pisany jest dokument XML. Zaczepnięto go jeszcze z wersji SGML-a. Jego struktura nie zmieniła się jeszcze od standardu XML 1.0, zatem podlega on wielu ograniczeniom:

1. Nie zapewnia wsparcia nowszym cechom XML-a, a co najważniejsze nazwą pól.
2. Niektóre ważne aspekty XML-a nie są ujęte w DTD.
3. Używa wybranego, nie związanego z XML-em formatu opisu schematu.

### 3.3.3. Schemat XML.

Jest nowszym językiem opisu od DTD, oraz bardziej rozbudowanym. Posiada bogatszy system opisu danych, co pozwala na zwiększenie dokładności i nałożenie bardziej precyzyjnych ograniczeń na dokument napisany w XML-u. Dzięki niemu możliwe jest lepsze przetwarzanie pliku XML a w połączeniu z implementacją WXS (W3C XML Schema) daje potężne narzędzie służące nie tylko do odczytu danych.

### 3.3.4. Przetwarzanie plików XML.

Dwoma najbardziej powszechnymi interfejsami do obsługi plików XML są SAX oraz DOM.

**SAX** jest leksykalnym, nastawionym na wydarzenia interfejsem dzięki któremu dokument jest czytany seryjnie, a zawartość dokumentu jest raportowana zwrotnie do wielu metod zaprojektowanych przez użytkownika. SAX jest wydajny i szybki w implementacji, ale nieefektywny podczas wybierania danych w sposób losowy z pliku XML. Najlepiej nadaje się do sytuacji kiedy z pliku XML pobierane są informacje w jeden określony sposób, niezależnie od tego gdzie się te informacje w tym pliku znajdują.

**DOM** jest to interfejs, który pozwala na nawigację po całym dokumencie, tak jakby to było drzewo

złożone z wierzchołków na których składowana jest zawartość dokumentu XML. Dokument DOM może zostać utworzony zarówno przez parser działający na pliku XML, jak i przez użytkownika przy pomocy odpowiednich funkcji języka programowania. Aby móc korzystać ze struktury oferowanej przez DOM trzeba pamiętać, potrzebuje ona wczytania całego pliku XML do pamięci. Dopiero po wczytaniu tworzy się struktura drzewiasta, która pozwala nam manipulować na dokumencie XML.

### **3.3.5. Wersje XML.**

Są obecnie dwie wersje standardu XML. Pierwsza, XML 1.0, została pierwszy raz zaprezentowana (oficjalnie zdefiniowana) w 1998 roku. Posiadała ona później wiele pomniejszych poprawek, ale nie były one oznaczane konkretnymi numerami. Jej trzecia poprawka (edycja) została wydana 4 lutego 2004 roku. Jest cały czas implementowana i szeroko rozpowszechniona. W tym samym dniu została również opublikowana wersja standardu XML 1.1. Zawiera ona dodatkowe cechy, które pozwalają na to, że XML staje się prostszy w użyciu. Nie jest ona tak szeroko rozpowszechniona jak wersja 1.0, ale jest polecana dla wielu użytkowników którzy potrzebują specjalnych cech XMLa.

W fazie projektowania istnieje także standard XML 2.0. Istnieją w nim propozycje na eliminację DTD ze składni XML-a, integrację nazw pól, XML Base, oraz XML Information Set do standardu XML-a.

## 4. Architektura klient-serwer.

Jest to asymetryczna architektura, która separuje oprogramowanie klienta od oprogramowania serwera w celu zwiększenia elastyczności oprogramowania, oraz ułatwieniu wprowadzania zmian w każdej z części. Niektóre tego typu architektury posiadają także trzeci komponent, tzw Serwer aplikacji, który spełnia rolę pośrednią pomiędzy klientem a serwerem głównym.

### 4.1. Model klient serwer.

Model klient serwer zapewnia w pełni skalowalną architekturę, w której każdy komputer może uczestniczyć albo jako klient, albo jako serwer. Oprogramowanie serwerowe uruchamiane jest przeważnie na silnych maszynach specjalnie do tego dedykowanych. Istnieje jednak możliwość zainstalowania takiego oprogramowania na komputerach domowych w celu stworzenia prostych baz danych bądź np.: domowych serwerów FTP. Oprogramowanie serwera może być uruchamiane na każdej maszynie wyposażonej w odpowiedni system operacyjny.

W myśl tej koncepcji systemy oparte na tej architekturze podzielono na dwie części. Z jednej strony została wydzielona pewna część systemu (inaczej mówiąc proces) odpowiedzialna za przechowywanie danych i zachowanie ich pełnej spójności. Z drugiej strony wydzielono pewne aplikacje czy procesy, które pobierają dane od użytkownika wyświetlają je i przetwarzają, a następnie albo przesyłają je do serwera w celu zapamiętania, albo generują pewne zapytania w celu uzyskania konkretnych informacji z komputera-serwera. W ten sposób cały proces przetwarzania danych mamy podzielony na dwie części. Z jednej strony mamy *serwer*, który przechowuje dane, ale potrafi także je wyszukiwać z przechowywanej bazy na podstawie zapytań poszczególnych komputerów (*klientów*), a z drugiej strony mamy aplikacje klienta, które tak naprawdę nic nie muszą wiedzieć o fizycznej strukturze danych przechowywanych na serwerze o sposobie ich zarządzania o liczbie użytkowników, a muszą jedynie umieć wysłać zapytanie do bazy, wyświetlić informacje na ekranie lub wysłać do serwera polecenie aktualizujące dane.

### 4.2. Właściwości klienta i serwera.

Serwer:

- pasywny (tzw. slave),
- czeka na zgłoszenia klientów,
- na ich żądania odpowiada wysyłając odpowiednie dane

### **Klient:**

- aktywny (tzw. master),
- wysyła żądania do serwera,
- czeka na odpowiedź serwera.

#### **4.2.1. Typy serwerów.**

- serwer wydruku
- serwer internetowy
- serwer FTP
- serwer baz danych
- serwer autoryzacji (np. LDAP)
- serwer aplikacji.

#### **4.2.2. Podział klientów.**

Klientów można podzielić zarówno na typy serwera z jakim się łączą, jaki na ich własności. Ja chciałbym więcej uwagi poświęcić własnością klientów. Ze względu na to klientów dzielimy na cienkich i grubych.

**Model cienkiego klienta:** klient posiada niezbyt wielką moc przetwarzania, ograniczoną do prezentacji danych na ekranie. Przykładem jest klient w postaci przeglądarki WWW. Jest najczęstszym rozwiązaniem w sytuacji, kiedy system scentralizowany jest zamieniany na architekturę klient-serwer. Wadą jest duże obciążenie serwera i linii komunikacyjnych.

**Model grubego klienta:** klient posiada znacznie bogatsze możliwości przetwarzania, w szczególności może zajmować się nie tylko warstwą prezentacji, lecz także warstwą przetwarzania aplikacyjnego (logiki biznesu). Używa większej mocy komputera klienta do przetwarzania zarówno prezentacji jak i logiki biznesu. Serwer zajmuje się tylko obsługą transakcji bazy danych. Popularnym przykładem grubego klienta jest bankomat. Zarządzanie w modelu grubego klienta jest bardziej złożone.

### **4.3. Serwer iteracyjny a serwer współbieżny.**

**Serwer iteracyjny** to serwer zdolny do obsługi jednego zadania klienta w danym momencie

(okresie) czasu, bądź bardziej precyzyjnie w danej jednostce czasu. Stosuje się je wtedy gdy zapytanie klienta można obsłużyć dokładnie jedną odpowiedzią.

**Serwer współbieżny** to serwer zdolny do obsługi współbieżnej wielu zleceń klientów w danej jednostce czasu. Składa się z wielu wątków, bądź procesów które realizują żądania klientów.

#### **4.4. Obsługa połączeń w modelu klient-serwer.**

Zauważmy że zadania klienta i serwera są asymetryczne zatem obydwa te procesy trzeba zaprogramować inaczej. Serwer uruchamiany jest przeważnie jako pierwszy. Istnieją oczywiście aplikacje klienckie które mogą uruchomić się bez uprzedniego połączenia z serwerem, a dopiero potem nawiązać połączenie, ale nie jest to istotne na tym etapie naszych rozważań.

##### **4.4.1. Etapy nawiązywania komunikacji:**

###### **Proces serwera:**

1. otwiera kanał komunikacyjny oraz informuje swoją lokalną stację o gotowości do przyjmowania żądań wysyłanych przez klientów pod pewien ogólnie znany adres.
2. oczekuje na żądania klientów doręczane pod ten adres.
3. jeśli jest to serwer iteracyjny, to przetwarza żądanie i wysyła odpowiedź. Jeśli jest to serwer współbieżny to tworzy on proces potomny, który ma obsłużyć bieżące żądanie klienta. Po zakończeniu obsługi zamyka kanał łączący go z klientem.
4. przechodzi do punktu 2 i czeka na kolejne żądanie.

###### **Proces klienta:**

1. otwiera kanał komunikacyjny oraz łączy się z określonym serwerem, na określonym adresie.
2. wysyła do serwera komunikat zawierający żądanie, po czym odbiera nadchodzące odpowiedzi. Powtarza tę czynność tak długo, jak to jest konieczne.
3. zamyka kanał komunikacyjny i kończy działanie.

Otwarcie kanału komunikacyjnego przez serwer nazywamy otwarciem **biernym**, natomiast otwarcie kanału komunikacyjnego przez klienta nazywamy otwarciem **czynnym**.

#### **4.5. Metody komunikacji.**

W podrozdziale opiszę tylko dwie najważniejsze metody komunikacji klienta z serwerem, czyli potoki oraz gniazda.

**Potok** jest to łącze komunikacyjne umożliwiające komunikację i przepływ danych tylko w jednym

kierunku. Przykład potoku zilustrowano na rysunku:

**Gniazdo** to pojęcie abstrakcyjne reprezentujące dwukierunkowy punkt końcowy połączenia. Dwukierunkowość oznacza możliwość wysyłania i przyjmowania danych. Wykorzystywane jest do komunikacji przez sieć nie tylko między aplikacjami klienta i serwera, ale również pomiędzy procesami.

Gniazdo posiada trzy główne właściwości:

1. typ gniazda identyfikujący jednoznacznie protokół wymiany danych,
2. lokalny adres np.: adres IP, bądź ethernetowy, jak również adres protokołu IPX.
3. opcjonalny lokalny numer portu identyfikujący proces, który wymienia dane przez gniazdo (jeśli typ gniazda pozwala używać portów)

Gniazdo może posiadać (na czas trwania komunikacji) dwa dodatkowe atrybuty:

1. adres zdalny
2. opcjonalny numer portu identyfikujący zdalny proces

## 5. Baza danych YODA.

W tym rozdziale pragnę przedstawić w pełni funkcjonalny skład danych zaimplementowany w systemie YODA. System ten jest implementacją czysto obiektowej bazy danych opartej na modelu M0, poszerzonym o funkcjonalność pointerów zwrotnych, oraz pojęcie klasy, bez zaimplementowanego dziedziczenia. Podczas mojej pracy implementacyjnej zaprojektowałem i stworzyłem obiektową bazę danych opartą na modelu sieciowym klient-serwer.

Składa się ona z trzech zasadniczych członów. Pierwszy z nich to skład danych, który przechowuje obiekty zarówno na dysku jak i w pamięci operacyjnej komputera.

Prace nad składem danych ewoluowały. Pierwszą implementacją składu był model M0 w jego czystej formie. Opierał się on na założeniach profesora Subiety w jego pracy "Teoria i konstrukcja obiektowych języków zapytań". Do modelu tego wprowadziłem wiele modyfikacji i ulepszeń. Poniższe rozdziały prezentują aktualną wersję systemu YODA.

### 5.1. Skład danych.

Opiera się on na modelu M0. Obiekty składowane są w pamięci operacyjnej komputera. Plik XML służy natomiast do tworzenia back-upu bazy. Zdecydowałem się na umieszczenie całej bazy w pamięci gdyż istniejące do tej pory komputery posiadają już bardzo duże pojemności pamięci. Istotnym otywem jest też problem szybkości dostępu do danych, który jest znacznie krótszy o ile dane przechowywane są w pamięci komputera. Wyeliminowany zostaje wtedy czas dostępu do dysku oraz czas potrzebny na przeszukiwanie w celu dostania się do odpowiednich danych.

Obiekty składowane w pamięci żyją tak długo, jak długo pracuje proces serwera. Z doświadczenia wiadomo, że komputery serwerowe są bardzo zadko wyłączane i pracują 24h na dobę, co daje nam przewagę tej metody przechowywania nad przechowywaniem danych na dysku. Obiekty składowane w pliku XML służą tylko i wyłącznie do zapisu kopii bezpieczeństwa, oraz są wczytywane przy uruchomieniu wątku serwera. Wszystkie niezbędne operacje na danych wykonywane są w pamięci komputera. Chodzi tu oczywiście o dodawanie obiektów, usuwanie i modyfikacje przeprowadzane na obiektach. Danych zapisany w pliku XML są składem biernym, co znaczy, że nie uczestniczą on w procesie przetwarzania danych.

Struktura składu danych podzielona jest pomiędzy obiekty trzech typów:

1. atomowe,
2. złożone,
3. pointerowe.

Dla każdego z tych typów obiektów została zaimplementowana osobna klasa. Klasy te dziedziczą jednak po ich klasie nadrzędnej zwanej *mainObject*.

Obiekty w systemie YODA przechowywane są w dwojaki sposób. Za pomocą składu zapisanego w pliku, bądź też za pomocą listy list w pamięci operacyjnej serwera bazodanowego. Do przechowywania kopii zapasowej składu użyłem plików XML, aby w sposób widoczny zobrazować zapisywanie i odczyt obiektów. Struktura plików XML jest bardzo czytelna i najbardziej funkcjonalna dla potrzeb tego prototypu. Na tym systemie opiera się kilka innych prac magisterskich dlatego też zastosowanie tak przejrzystej struktury pozwala nawet na ręczne modyfikacje pliku, bez zastosowania wyszukanych narzędzi programistycznych. Z drugiej jednak strony zapisany w pliku XML skład może być w łatwy sposób zaprezentowany czytelnikowi.

```
<root>
  <complex0 id="0" nazwa="osoba">
    <atomic0 id="0_0" nazwa="imie" type="String" zawartosc="jan" />
    <atomic0 id="0_1" nazwa="nazw" type="String" zawartosc="Kot" />
    <atomic0 id="0_2" nazwa="zar" type="Integer" zawartosc="100" />
    <pointer0 id="0_3" id_xml="2_1" nazwa="brat" />
    <complex0 id="0_4" nazwa="Adres">
      <atomic0 id="0_4_0" nazwa="miasto" type="String"
        zawartosc="Torun" />
    </complex0>
  </complex0>
  <complex0 id="1" nazwa="osoba">
    <atomic0 id="1_0" nazwa="imie" type="String" zawartosc="Adam" />
    <atomic0 id="1_1" nazwa="nazw" type="String" zawartosc="Nowak" />
    <atomic0 id="1_2" nazwa="zar" type="Integer" zawartosc="130" />
  </complex0>
  <complex0 id="2" nazwa="osoba">
    <atomic0 id="2_0" nazwa="imie" type="String" zawartosc="Adam" />
    <atomic0 id="2_1" nazwa="nazw" type="String"
      zawartosc="Myszowski" />
    <atomic0 id="2_2" nazwa="zar" type="Integer" zawartosc="150" />
  </complex0>
</root>
```

Powyższy przykład pokazuje jak prosty i czytelny jest skład systemu YODA zapisany w pliku XML, oraz przechowywany w pamięci. Do odczytu i zapisu danych wykorzystany jest interfejs DOM, który podczas tworzenia pliku, oraz jego przebudowy stosuje strukturę drzewiastą.

Obiekty przechowywane w pamięci składowane są na liście związanej. Każdy obiekt główny ma swój węzeł na tej liście. Zastosowanie takiego podejścia rozwiązało w bardzo prosty sposób stworzenie listy identyfikatorów startowych. W języku Java lista związana posiada własne

autonumerowanie, zatem numery obiektów na głównej liście to dokładnie zbiór identyfikatorów startowych. Każdy obiekt na liście głównej jeśli jest obiektem typu *complexO* posiada własną listę związaną zawierającą jego podobiekty. Każdy z tych podobiektów, może oczywiście zawierać listę swoich podobiektów, o ile sam również jest obiektem złożonym itd.

Aby poradzić sobie z pointerami zastosowałem system wskaźników (identyfikatorów) zapisanych w pliku XML dla każdego obiektu pointerowego (*id\_xml*). Zapisany jest w nim adres obiektu na który wskazuje dany pointer. Ponieważ w języku Java wszystkie odnoszenia przekazywane są przez referencję, dlatego potrzebna jest nam na starcie zamiana tych adresów na referencje do odpowiednich obiektów. Odpowiedzialna za to jest funkcja *setPointers*, która otrzymuje jako argument listę pointerów. Na liście znajdują się wszystkie pointerzy wczytane z XML-a. Po wykonaniu tej funkcji następuje ustawienie wszystkim pointerów referencji do odpowiednich obiektów.

### 5.1.1. Struktura klas obiektów w systemie YODA

Struktura obiektów w bazie przedstawia się zatem następująco. Jak już wspomniałem klasą nadrzędną jest klasa *mainObject*. Dziedziczą po niej trzy klasy *atomicO*, *pointerO*, oraz *complexO*.

Klasa *mainObject* zawiera metody do manipulowania danymi wspólne dla pozostałych trzech klas. Znajdują się tu metody do usuwania pointerów, sprawdzania typów danych itp., które są przeciążane w pozostałych trzech klasach dziedziczących.

Klasa *atomicO* zawiera definicję obiektu atomowego. Obiekt ten składa się z czterech pól:

- *id* – unikalny identyfikator obiektu,
- *nazwa* – jest to nazwa obiektu w bazie danych np.: nazwisko,
- *typ* – każdy obiekt posiada swój typ, wyróżniamy takie typy jak "String", "Integer", itp.,
- *treść* – inaczej zawartość obiektu np.: "Kowalski"

przykład:

```
<atomicO id="0_2" nazwa="zar" type="Integer" zawartosc="100" />
```

zapisany jest w pamięci jako obiekt na liście obiektu o id 0:

```
<2, zar, Integer, 100>
```

Klasa *pointerO* zawiera definicję obiektu pointerowego. Obiekt taki składa się z trzech pól:

- *id* – unikatowy identyfikator obiektu,

- nazwa – nazwa pointera – dowolny łańcuch znakowy,
- xml\_id - wskaźnik na obiekt znajdujący się w składzie XML, na jego podstawie tworzona jest referencja do obiektu wczytanego do pamięci.

Przykład:

```
<pointer0 id="0_3" id_xml="2_1" nazwa="brat" />
```

zapisany jest w pamięci na liście obiektu o id 0:

```
<3, 2_1, "brat">
```

Klasa *complex0* zawiera definicję obiektu złożonego. Składa się on z trzech pól:

- id – unikatowy identyfikator obiektu,
- nazwa – nazwa obiektu złożonego nie musi być unikatowa, i jest to łańcuch znakowy, np.:osoba,
- lista obiektów – zawiera listę podobiektów powiązanych z tym obiektem

Przykład:

```
<complex0 id="0" nazwa="osoba">
  <atomic0 id="0_0" nazwa="imie" type="String" zawartosc="jan" />
  <pointer0 id="0_1" id_xml="2_1" nazwa="brat" />
</complex0>
```

zapisany jest w pamięci jako:

```
<0, "osoba",{<0, imie, String, "jan">,<1, 2_1, "brat">}>
```

Pomiędzy tymi obiektami istnieje skomplikowana struktura powiązań za pomocą pointerów zwrotnych. Każdy obiekt na którego wskazuje pointer posiada wskaźnik zwrotny, tzw. pointer zwrotny. Dzięki zastosowaniu pointerów zwrotnych przyśpieszyłem metodę usuwania obiektów, a co za tym idzie wydajność całej bazy. Przyśpieszenie to polega na tym, że jeśli mamy obiekt B na którego wskazuje obiekt A, to dzięki pointerowi zwrotnemu na A jestem w stanie w szybki sposób poinformować ten obiekt o tym że nastąpi usunięcie obiektu B. Obiekt a może zatem wykasować swój pointer na obiekt B, a obiekt B może zostać spokojnie usunięty bez naruszania integralności danych znajdujących się w bazie.

### 5.1.2. Typy obiektów.

W systemie YODA wyróżniamy trzy klasy obiektów.

- Obiekty trwałe – istnieją przez cały czas pracy z bazą, oraz zarówno po jak i po skończeniu procesu serwera, oraz przed zainicjowaniem do pracy tego procesu.
- Obiekty tymczasowe – inaczej ulotne, istnieją jako wyniki zapytań, składowane są stosie

ENVS.

- Obiekty lokalne – tak samo składowane na stosie ENVS, ale przechowywane są tylko w najwyższym środowisku tego stosu.

## **5.2. Interakcja użytkownika z bazą.**

Komunikacja użytkownika z bazą danych odbywa się w architekturze klient serwer. Klient przy pomocy odpowiedniego oprogramowania łączy się z serwerem. Jako że w obecnym stadium działania system YODA nie obsługuje uwierzytelniania klientów, zatem dostęp do bazy uzyskuje każdy użytkownik posiadający oprogramowanie klienckie. Po nawiązaniu połączenia klient jest w stanie wysłać do bazy żądania i otrzymać na nie odpowiedź.

Komunikacja pomiędzy klientem a serwerem nawiązuje się za pomocą gniazd. Serwer nasłuchuje na odpowiednim porcie (domyślnie jest to port 12121) zgłoszeń od klienta. Klient natomiast ustanawia połączenie z serwerem poprzez podanie nazwy serwera, bądź też jego adresu IP, oraz portu na którym ma być nawiązane połączenie.

Klient wysyła żądania do serwera w postaci łańcuchów znakowych i w takim samym formacie odbiera odpowiedzi od serwera. Klient nie komunikuje się bezpośrednio z bazą danych (ściślej mówiąc ze składem). Łączy się jedynie instancją serwera, która pośredniczy w procesie komunikacji ze składem. Zatem architektura systemu YODA jest trójwarstwowa (skład danych – interfejs komunikacyjny wraz z serwerem – klient).

## **5.3. Konsola klienta systemu YODA.**

Konsola zawiera przyjazny graficzny interfejs do komunikacji z bazą danych. Jest w stanie połączyć się z każdą działającą instancją serwera. Łącząc się z serwerem podajemy w odpowiednich okienkach adres serwera oraz numer portu. Po nawiązaniu połączenia jesteśmy w stanie wysłać zapytania do serwera i otrzymywać od niego odpowiedzi w odpowiednio sformatowanej formie.

## **5.4. Budowa serwera.**

Serwer systemu YODA składa się z dwóch podstawowych klas. Pierwszą jest klasa *RunMe*, która zawiera klasę główną, dzięki której tworzony jest nasłuch na gniazdach. Drugą jest klasa *BaseSerwer*, która zawiera podstawową obsługę każdego klienta. Klasa *RunMe* to klasyczny serwer współbieżny, który przyjmuje zgłoszenie klienta. Następnie to zgłoszenie klienta jest przesyłane do klasy *BaseSerwer* która obsługuje klienta dzięki temu, że tworzony jest nowy wątek procesu. Proces główny wraca znów do stanu nasłuchu i jest w stanie obsłużyć kolejnego zgłaszającego się klienta. W klasie *RunMe* następuje także wczytanie składu zapisanego kopii zapasowej w pliku XML.

Następuje to podczas startu procesu serwera. Wywołana zostaje wtedy metoda `wczytajZXML()`, która uruchamia funkcje:

```
wczytObjectCom(Element dziecko), wczytObjectAt(Element dziecko),  
wczytObjectPt(Element dziecko).
```

W klasie `BaseSerwer` następuje po każdym prawidłowym obsłużeniu klienta zapis kopii zapasowej składu z pamięci na dysk do pliku XML. Odpowiada za to metoda: `zapiszDoXML()`. Aby jednak był możliwy ten zapis najpierw musi nastąpić zbudowanie poprawnego drzewa XML za pomocą funkcji: `zapisXML()`, która wywołuje dodatkową metodę:

```
dodObjectC(mainObject pom, String i, Element el).
```

Komunikacja z klientem odbywa się za pomocą dwóch zmiennych input oraz output.

```
input = new BufferedReader(new InputStreamReader(  
                                socket.getInputStream()));  
output = new PrintWriter(new BufferedWriter(new OutputStreamWriter(  
                                socket.getOutputStream())), true);
```

Zmienna `input` odpowiada za standardowe wejście do serwera, natomiast zmienna `output` za standardowe wyjście serwera.

Aby zakończyć połączenie klient musi wysłać serwerowi specjalną komendę postaci: `@end`

## 5.5. Uruchomienie systemu YODA

Aby uruchomić system należy najpierw wystartować serwer bazodanowy `RunMe`.

Można to zrobić w dwojaki sposób.

1. Załadować środowisko programistyczne Eclipse, wczytać projekt i z odpowiednimi argumentami odpalić klasę `RunMe`.
2. Wykonać komendę `java RunMe 12121`, znajdując się w katalogu `/bin`.

Po uruchomieniu następuje wczytanie bazy z pliku XML do pamięci przy pomocy interfejsu DOM - skonstruowana zostaje lista list obiektów. Kolejnym krokiem jest ustawienie pointerów, oraz pointerów zwrotnych na odpowiednie obiekty. Ostatnim krokiem jest inicjalizacja nasłuchy procesu serwera na odpowiednim porcie (domyślnie 12121). System YODA jest gotowy do pracy.

### Uruchomienie klienta.

Klienta uruchamia się poprzez wydanie polecenia `java Gconsole [nazwa_serwera] [port]`,

Przykładowe wywołanie tej komendy:

```
java Gconsole \\serwerek 12121
```

co skutkuje nawiązaniem połączenia z instancją serwera postawiona na maszynie o nazwie *serwerek* na porcie *12121*.

## 6. Zawartość dysku CD:

Tekst pracy magisterskiej: pfd and odt:

`/praca magisterska/`

Katalog główny zawierający źródła i skompilowane pliki:

`/workspace/`

Katalog ze źródłami:

`/workspace/src/`

Pliki stworzone przez autora pracy magisterskiej:

`/workspace/sklad.xml`

`/workspace/src/code/objects/atomicO.java`

`/workspace/src/code/objects/complexO.java`

`/workspace/src/code/objects/pointerO.java`

`/workspace/src/code/objects/mainObject.java`

`/workspace/src/code/BaseServer.java`

`/workspace/src/code/DBdriver.java`

`/workspace/src/code/KlientConnect.java`

`/workspace/src/code/Klient.java`

`/workspace/src/code/RunMe.java`

Pliki stworzone przy współpracy z panią Martą Rogińską:

`/workspace/src/code/Binder.java`

## 7. Bibliografia

1. Subieta K. – „Teoria i konstrukcja obiektowych języków zapytań”, Wydawnictwo PJWSTK, Warszawa 2004 r.
2. Subieta K. – „A Critique of Object Algebras”, 1995, niepublikowana, można ją znaleźć na stronie: <http://www.ipipan.waw.pl/~subieta/artykuly/CritiqObjAlg.html>
3. Elmasri R., Navathe S. B. – „Fundamentals of database systems”, Addison Wesley Publishing Company, 1994
4. Beynon-Davies P - “Systemy baz danych”, Wydawnictwo Naukowo-Techniczne, Warszawa 2003 r.
5. [www.odmg.org](http://www.odmg.org) – główna strona grupy ODMG
6. E.F. Codd – "A Relational Model of Data for Large Shared Data Banks", pełny tekst jest dostępny na stronie: <http://www.acm.org/classics/nov95/toc.html>
7. A. Silberschatz, H. F. Korth, S. Sudarshan – “Database System Concepts”, McGraw-Hill Companies, 2001
8. C. J. Date – “Wprowadzenie do systemów baz danych”, Wydawnictwo Naukowo-Techniczne, Warszawa 2001 r.