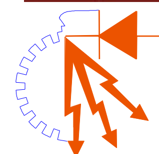
POLITECHNIKA ŁÓDZKA

Wydział Elektrotechniki, Elektroniki,
Informatyki i Automatyki

Katedra Informatyki Stosowanej



Praca dyplomowa magisterska

Aktualizowalne perspektywy w obiektowych bazach danych

Krzysztof Śmiałowicz

Nr albumu: 114900

Opiekun pracy:
prof. dr hab. inż K. Subieta
dr inż. R. Adamus

Łódź, 2007

Podziękowania

Składam serdeczne podziękowania mojemu promotorowi Panu profesorowi Kazimierzowi Subiecie za danie szansy pracy przy rozwijaniu innowacyjnego i przyszłościowego projektu badawczego eGov Bus.

Bardzo dziękuje mojemu opiekunowi Panu doktorowi Radosławowi Adamusowi za życzliwość i cenne uwagi które znacząco pomogły przy pisaniu pracy.

Spis treści

I Część teoretyczna

1 Perspektywy na tle architektury baz danych 8
1.1 Sposoby tworzenia i wykorzystania perspektyw 11
1.2 Aktualizacja perspektywy 12
1.2.1 Kontrola aktualizowanych danych 12
1.2.2 Możliwość dokonania pełnej aktualizacji perspektywy - problem 13
1.2.3 Możliwość rozszerzenia aktualizacji perspektywy 14
2 Język SBQL 16
2.1.1 Modele obiektowe 16
2.1.2 Podejście stosowe dla języka SBQL 17
2.1.3 Stos QRES 17
2.1.4 Stos ENVS 18
2.2 Zapytania 20
Operatory algebraiczne 21
2.2.1 Operatory niealgebraiczne 22
2.3 Przykład wykonania zapytania z uwzględnieniem zawartości stosów 22
2.4 Konstrukcje imperatywne 24
2.4.1 Operacja tworzenia nowego 24
2.4.2 Operacja usuwania obiektu 25
2.4.3 Operacja przypisania 25
2.4.4 Operacja wstawienia 25
2.5 Procedury 25
3 Perspektywy w SBQL 28
3.1 Procedury składowe a perspektywy 28
3.2 Perspektywy, części składowe 31
3.2.1 Procedura on_navigate 32
3.2.2 Procedura obsługi wirtualnego obiektu 33
3.2.3 Wirtualny identyfikator 33
3.2.4 Przepływ sterowania podczas korzystania z perspektyw 34
3.2.5 Podperspektywy 34
3.3 Deklaracja perspektyw 36
3.3.1 Przykłady perspektyw 38
4 Rozszerzenie możliwości perspektyw 40
4.1 Zmienne 40
4.2 Procedury 42
4.2.1 Procedury zwykłe 42
4.2.2 Procedura zwykła - operacje na stosie 44
4.2.3 Procedury wewnętrzne 44
4.2.4 Działanie procedur wewnętrznych dla perspektyw wykorzystujących własność stosu 46
4.2.5 Porównanie procedur 47
4.3 Podsumowanie zmiennych, podperspektyw i procedur 48
4.4 Zasięg widoczności dla zmiennych i procedur 49
4.5 Konstruktor 52

4.5.1 Sposób wprowadzenia zachowania konstruktora do podejścia stosowego............ 56
4.6 Generacja procedury on_retrieve.. 56
4.7 Generacja podperspektyw.. 57
4.8 Problemy mogące wystąpić podczas generowania podperspektyw oraz procedury on_retrieve...60
II Część praktyczna
5 Implementacja perspektywy w systemie Odra...63
5.1 Architektura aplikacji...63
5.3 Skład danych.. 64
Warstwa fizyczna składu danych...65
5.3.1 Warstwa obiektów składu..66
5.3.2 Warstwa obiektów bazy i metabazy.. 67
5.4 Moduły... 67
5.5 Podział na bazę i metabazę...68
5.6 Wewnętrzna budowa wybranych obiektów... 70
5.6.1 Obiekty podstawowe dla wszystkich innych obiektów w bazie i metabazie......... 70
5.6.2 Obiekty dla procedur.. 71
5.7 Proces tworzenia perspektywy... 75
5.7.1 Dodanie perspektywy do modułu... 76
5.7.2 Kompilacja perspektywy... 78
5.7.3 Generowanie on_retrieve...79
5.7.4 Generowanie podperspektyw.. 81
5.8 Użycie perspektywy... 81
5.8.1 Odwołanie do zmiennych i procedur poza wirtualnym obiektem.........................83
5.9 Korzystanie z prototypu.. 83
5.9.1 Tworzenie bazy..84
5.9.2 Uruchamianie procesu serwera..84
5.9.3 Uruchamianie procesu klienta.. 84
6 Dalsze kierunki rozwoju perspektyw...86
6.1 Rozwój klas i perspektyw..86
6.2 Rozwój składni perspektywy - przeniesienie nacisku na wirtualny obiekt.................88
6.2.1 Przykłady różnych wariantów przyszłych perspektyw..92
6.3 Przyjmowanie cech z programowania aspektowego...97
6.3.1 Perspektywy pośredniczące..97
6.3.2 Cechy analogiczne do języka AspectJ...97
Podsumowanie... 98
Bibliografia.. 99

Cel pracy

W większości istniejących bazach danych dominujących obecnie na świecie istnieje możliwość definiowania wirtualnych źródeł danych dających możliwość ukrycia ich wewnętrznej budowy. Możliwość odseparowania wewnętrznej budowy od zewnętrznej reprezentacji danych pozwala na elastyczne podejście do wewnętrznej budowy, a także daje szanse na szybsze i powodujące mniej komplikacji zmiany w samym sercu bazy bez wpływania na zewnętrzne elementy.

Wirtualne źródła danych jakimi są perspektywy w powstałych implementacjach baz danych dają pełną możliwość czerpania danych, jednakże proces odwrotny jakim jest dostarczanie nowych danych, modyfikacje lub ich usuwanie w większości baz danych jest niewystarczające w stosunku do możliwości jakie mogły by zaoferować w przypadku możliwości pełnego dwukierunkowego przepływu danych.

Perspektywa nie posiadająca tego typu ograniczeń w obiektowej bazie danych Odra J2 dla języka SBQL jest celem pracy magisterskiej „Aktualizowalne perspektywy w obiektowych bazach danych" wraz z wykonaniem implementacji w celu udowodnienia możliwości istnienia perspektyw bez niepotrzebnych ograniczeń występujących w większości baz danych.

Zakres pracy

Perspektywy dające możliwość dokonywania aktualizacji danych występują w wielu rodzajach baz danych od relacyjnych przez bazy obiektowo-relacyjne do baz czysto obiektowych bez domieszki relacyjności. Pierwszy rozdział pracy poświęcony jest wprowadzeniu do perspektyw występujących w istniejących obecnie bazach danych. Rozdział ten ukazuje problemy powstające z aktualizacją tych perspektyw oraz sposób rozwiązania.

Po wprowadzeniu do tematyki związanej z perspektywami następne rozdziały poświęcone są perspektywom tylko i wyłącznie dla obiektowej bazy danych Odra J2 korzystającej z języka SBQL. W celu dogłębnego opisania perspektyw dla tej bazy drugi rozdział pracy stanowi wprowadzenie do samego języka SBQL. Tematyka związana z językiem SBQL jest bardzo obszerna i w pracy zostały zawarte tylko i wyłącznie fragmenty potrzebne dla dalszych rozdziałów omawiających perspektywy.

Kolejny, trzeci rozdział skupia się na perspektywach, celach istnienia perspektyw, semantykę oraz na porównaniu mechanizmu wirtualnych wskaźników z sposobami niepełnego rozwiązania w innych językach obiektowych.

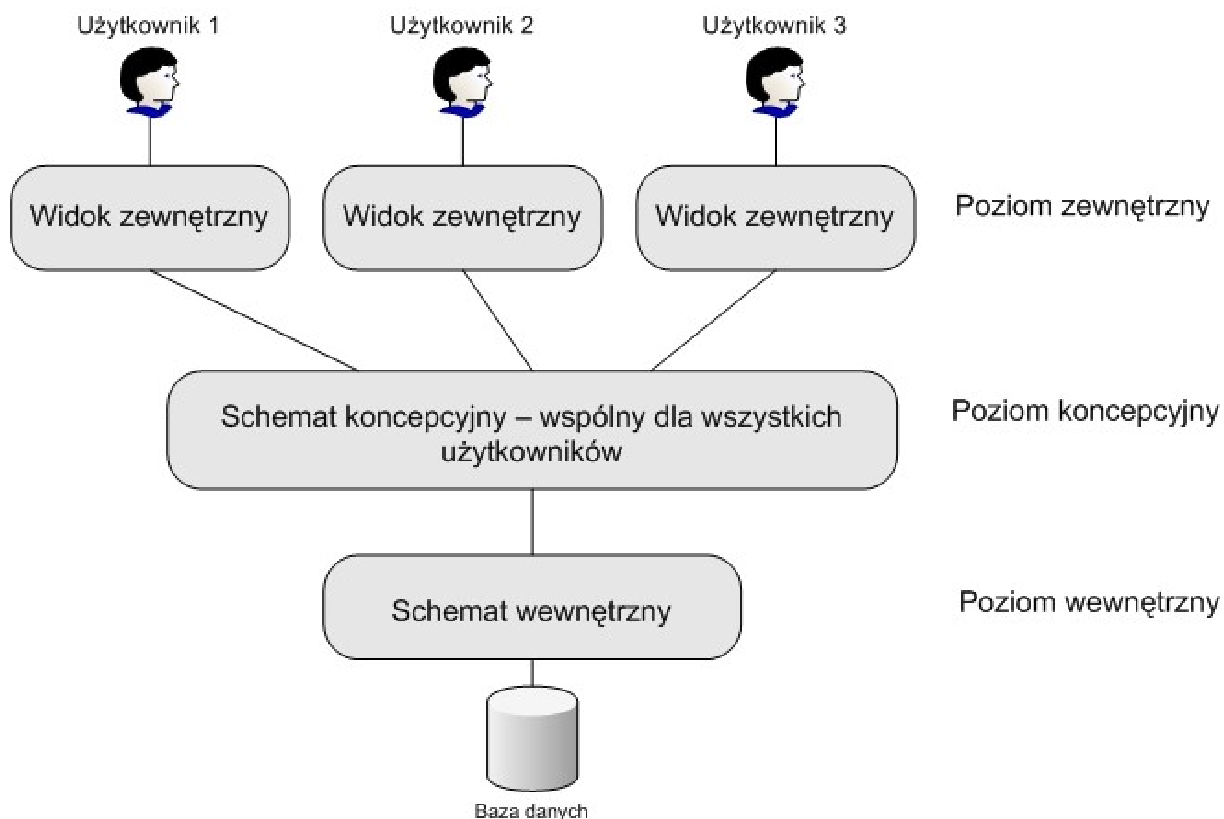
Czwarty rozdział prezentuje dodatkowe rozszerzenia możliwości perspektyw rozszerzające samą perspektywę o dodatkowe cechy istniejące w innych implementacjach języka SBQL jak i również nowatorskie rozwiązania powstałe podczas pisania tej pracy.

Następnym elementem pracy jest opis części praktycznej jakim jest zaimplementowanie perspektywy dla języka SBQL w obiektowej bazie danych Odra J2.

Ostatnie rozdziały prezentują dalsze możliwości rozwoju perspektyw a także podsumowują ich obecny kształt oraz opisują możliwe kierunki rozwoju.

I Część teoretyczna

1 Perspektywy na tle architektury baz danych



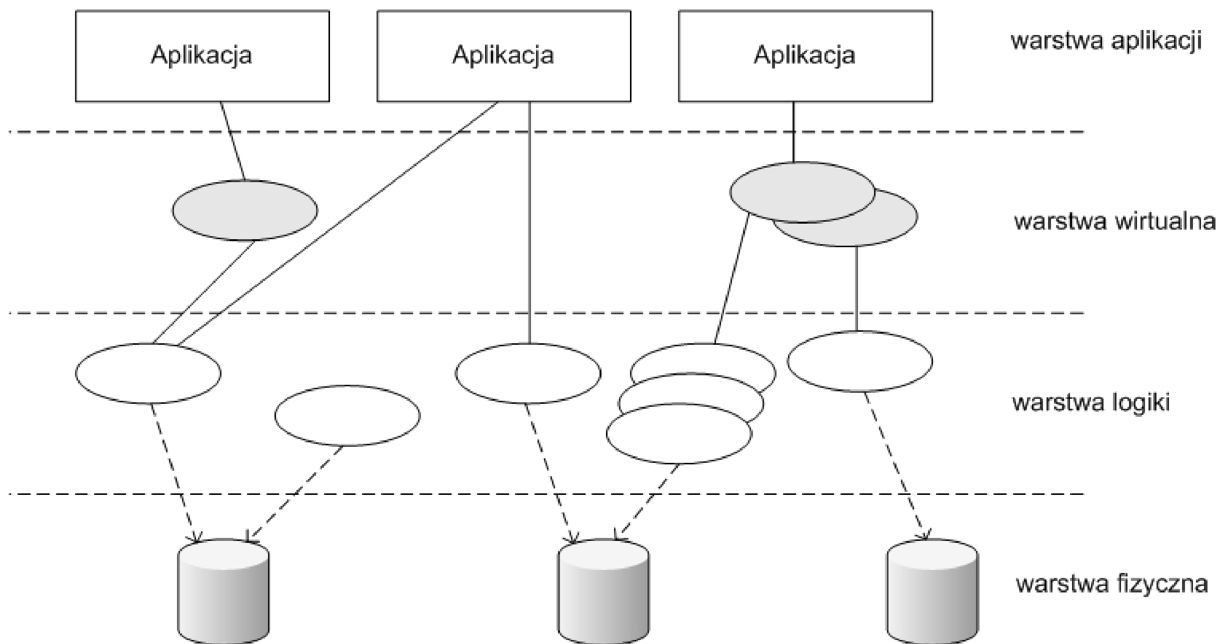
Rysunek 1: Architektura ANSI/SPARC

Architektura dla baz danych jako trójwarstwowa została zaproponowana przez komitet ANSI/SPARC (American National Standards Institute/Standards Planning and Requirements Committee). Dzieli się ona na trzy poziomy/schematy(idąc od najniższego poziomu):

- wewnętrzny
- koncepcyjny
- zewnętrzny

Schemat wewnętrzny najniżej występujący ze wszystkich schematów, definiuje sposób przechowywania danych, ich fizyczną organizację oraz sposób dostępu do tych danych. Drugi schemat natomiast definiuje struktury z punktu widzenia globalnego użytkownika, tabele, klasy, atrybuty oraz ograniczenia. Definiuje ponadto sposób ich połączenia pomiędzy sobą. Najwyższa warstwa jest tworzona pod kątem pewnej grupy użytkowników, służy do zaprezentowania im danych pod kątem różnych aspektów. Ta warstwa może być tworzona

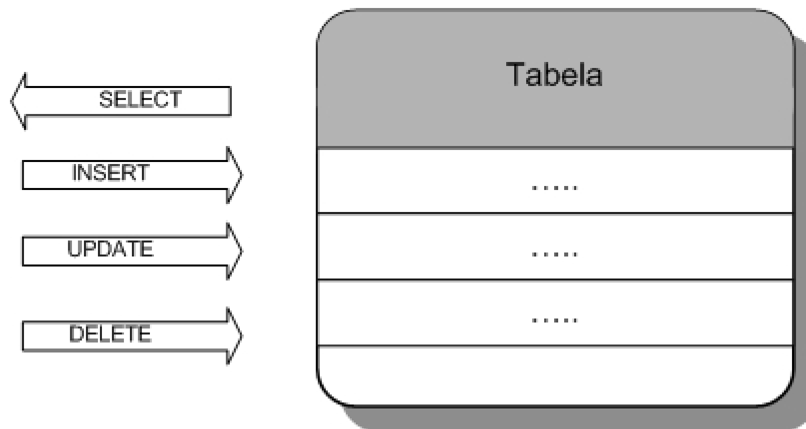
przez zastosowanie perspektyw które pośredniczą między aplikacją końcową a warstwą wewnętrzna. Głównym celem tej architektury jest zapewnienie niezależności tych schematów od siebie czyli też niezależność fizyczną jak i logiczną danych oraz możliwość niezależności aplikacji i danych. Inną zaletą oprócz separacji warstw od siebie jest możliwość przypisania zarządzania nimi innym osobom. I tak dla najniższej warstwy jest to administrator systemowy, środkowej programista systemowy a dla ostatniej programista aplikacji.



Rvsunek 2: Warstwv danvch

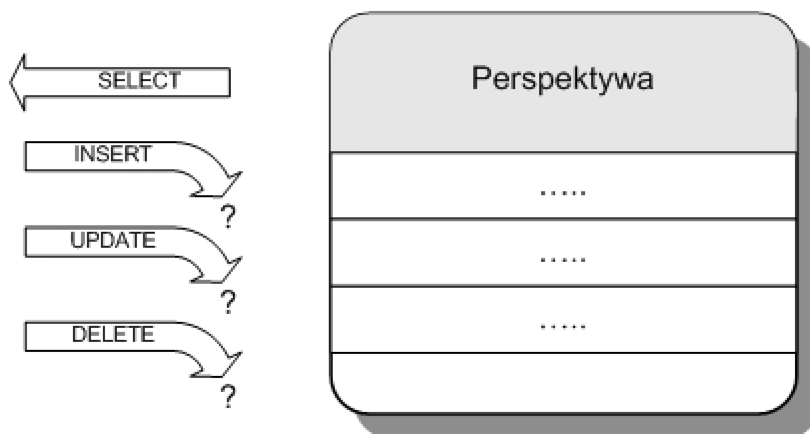
W przypadku zmiany schematu bazy danych lub też dodanie dodatkowych źródeł danych jak dodanie nowego serwera kluczową role ma warstwa wirtualna. Pozwala ona na dokonanie zmian w sposób niezauważalny dla warstwy aplikacji. W warstwie wirtualnej znaczącą role odgrywają perspektywy - umożliwiają one na dokonywanie zmian w sposób przezroczysty.

W relacyjnych bazach danych oraz relacyjno-obiektowych podstawowym źródłem informacji są tabele, w obiektowych bazach są to zbiory obiektów. Można na nich wykonywać operacje pobierania danych (SELECT) czy też ich aktualizacji jak wstawianie (INSERT), aktualizowanie (UPDATE) czy usuwanie (DELETE).



oraz relacyjno-obiektowych bazach danych

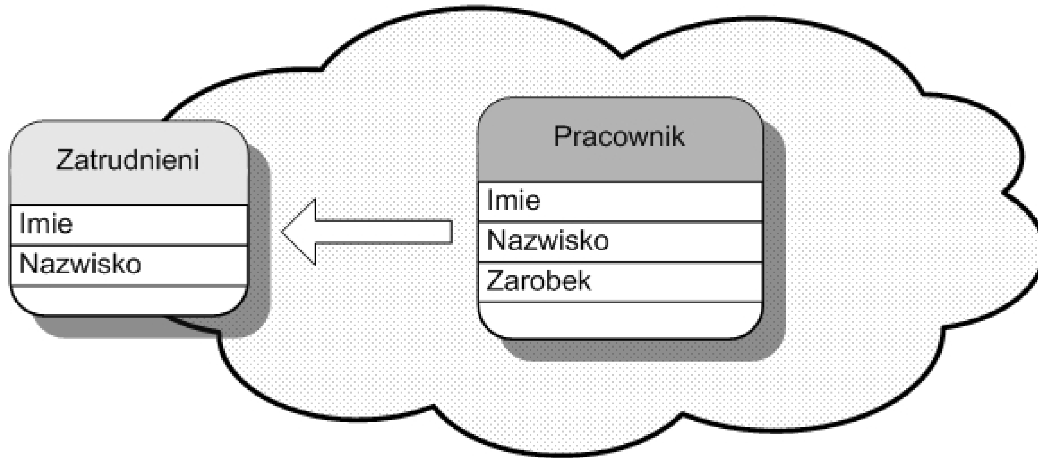
W powyższych bazach danych w pośredniczeniu do źródła informacji jakimi są tabele mogą występować perspektywy. Perspektywa dla bazy danych jest to logiczna struktura podobna do wirtualnej tabeli generowana w locie w oparciu o zapytanie podane w trakcie jej tworzenia. Pobieranie danych z perspektyw jest wykonywane w taki sam sposób jak w przypadku tabel, perspektywa pod tym względem jest przezroczysta. Natomiast aktualizacja danych dla perspektyw nie zawsze jest możliwa. Nie które bazy danych nie pozwalają na ich aktualizację lub tylko zezwalają na częściową aktualizację.



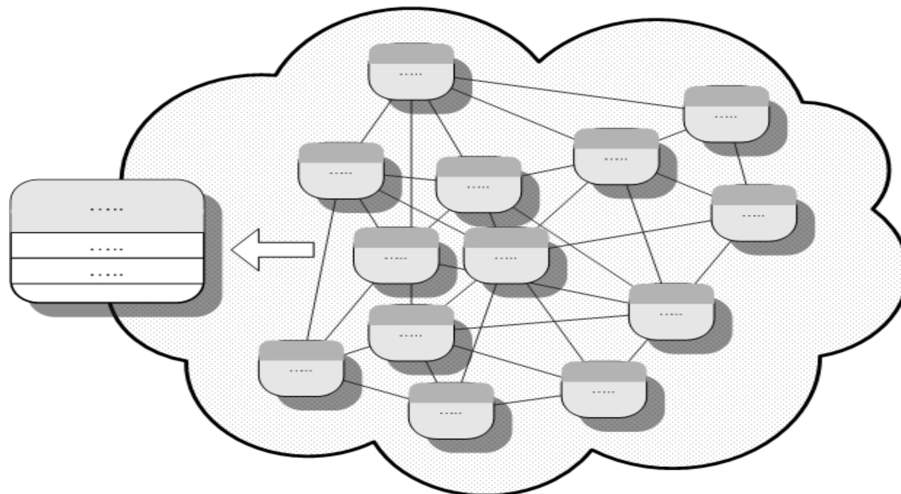
Rysunek 4: Operacje wykonywane na perspektywie w relacyjnych i relacyjno-obiektowych bazach danych, operacje związane z aktualizacją nie zawsze są możliwe

Perspektywy dają dodatkowe możliwości względem tabel takie jak możliwość ukrycia części informacji czy też zdefiniowanie interfejsu dla widoczności na zewnątrz systemu

do mogącego się zmienić systemu, co powoduje także modularyzacje systemu - efekt w większych systemach wymagany do możliwości zapanowania nad złożonością skomplikowanego systemu.



Rysunek 5: Ukrycie zarobku pracowników przez perspektywę



Rysunek 6: Modularyzacja systemu przez zastosowanie perspektywy jako interfejsu do niego

1.1 Sposoby tworzenia i wykorzystania perspektyw

Perspektywa jest tworzona na podstawie zapytania któremu przypisywana jest specjalna nazwa. Dla baz relacyjnych i relacyjno-obiektowych zapytania w języku SQL

tworzące perspektywę mają format:

```
CREATE VIEW nazwa AS zapytanieTworzące ;
```

gdzie *nazwa* to nazwa nowo utworzonej perspektywy a *zapytanieTworzące* to zapytanie będące źródłem dla perspektywy.

Przykład tworzenia perspektywy o nazwie *PanstwaRozszerzone* :

```
CREATE VIEW PanstwaRozszerzone AS
SELECT Id, Nazwa, Ludnosc, Powierzchnia, Ludnosc/Powierzchnia AS Gestosc FROM Panstwa;
```

na podstawie tabeli generowanej zapytaniem:

```
CREATE TABLE Panstwa(Id number PRIMARY KEY, Nazwa varchar(25), Ludnosc number, Powierzchnia number);
```

Z perspektywy utworzonej powyżej możemy korzystać tak jak z zwykłej tabeli do pobierania danych:

```
SELECT * FROM PanstwaRozszerzone;
```

Jeśli baza danych wspiera tworzenie perspektyw to dostęp do pobierania danych jest zawsze bezproblemowy.

1.2 Aktualizacja perspektywy

Problem aktualizacji perspektywy można podzielić na dwa podproblemy:

- kontrolę aktualizowanych danych w perspektywie
- brak możliwości dokonania aktualizacji dla dowolnej perspektywy

1.2.1 Kontrola aktualizowanych danych

Jeśli dla perspektywy istnieje pełna możliwość modyfikacji oraz wstawiania danych do perspektywy to istnieje możliwość takiej aktualizacji bądź wstawienia które spowoduje, że dane na których wykonano którąś operację choć były wcześniej osiągalne przez perspektywę to po dokonaniu aktualizacji już nie będą dostępne dla perspektywy.

Przykład:

Dla perspektywy:

```
CREATE VIEW DuzePanstwa AS
SELECT * FROM Panstwa WHERE Ludnosc > 1000000;
```

Dokonajmy zmiany ilości ludności dla konkretnego państwa poniżej miliona mieszkańców. Zapytanie zostanie wykonane poprawnie co w efekcie zaowocuje, że dane państwo nie będzie już dostępne przez perspektywę DuzePanstwa.

W celu zabezpieczenia przed powyższymi skutkami w języku SQL [Date00] można zastosować klauzulę WITH CHECK OPTION przy tworzeniu perspektywy. Po zastosowaniu tej klauzuli po każdej operacji aktualizacji danych bądź wstawiania będzie sprawdzany warunek czy te operację dają w efekcie dane osiągalne przez perspektywę. W przypadku negatywnego rozpatrzenia warunku dane nie zostaną zaktualizowane.

Przykład:

Perspektywa z opcją sprawdzania

```
CREATE VIEW DuzePanstwa AS
SELECT * FROM Panstwa WHERE Ludnosc > 1000000
WITH CHECK OPTION;
```

1.2.2 Możliwość dokonania pełnej aktualizacji perspektywy - problem

Operacje takie jak wstawianie, modyfikowanie czy usuwanie nie zawsze są możliwe dla wszystkich możliwych perspektyw.

Przykład:

Aktualizacja perspektywy na podstawie wstawiania nowych danych do niej. Perspektywa zawiera kolumny takie jak Id, Nazwa, Ludnosc, Powierzchnia, Gestosc.

Wstawy dane do perspektywy wykorzystując tylko kolumnę Nazwa:

```
INSERT INTO PanstwaRozszerzone(Nazwa) VALUES('USA');
```

Jeśli baza danych pozwala na aktualizację perspektyw to ta operacja zostanie zakończona sukcesem.

Szczególną uwagę należy zwrócić na kolumny Ludnosc, Powierzchnia, Gestosc -

zależą one od siebie. Dokładnie o ich relacjach względem siebie wie osoba tworząca perspektywę. Natomiast dla osoby korzystającej te kolumny wydają się jednakowe. Przy wstawieniu kolejnego państwa:

```
INSERT INTO PanstwaRozszerzone(Nazwa,Ludnosc, Powierzchnia)
VALUES('Polska',38189000,322577);
```

Operacja też się powiedzie. Następna operacja wstawiania wykorzystuje też dwie z tych współzależnych kolumn:

```
INSERT INTO PanstwaRozszerzone(Nazwa, Ludnosc, Gestosc)
VALUES('Watykan', 932, 2118);
```

Zakończy się niepowodzeniem. Dla osoby z zewnątrz te dwa powyższe zapytania niewiele się różnią, mogła przewidywać że oba zakończą się sukcesem. Niestety przewidywania okazały się błędne ponieważ kolumna Gestosc jest tworzona na podstawie:

$$Gestosc = \frac{Ludnosc}{Powierzchnia}$$

Jest to kolumna wirtualna generowana z dwóch pozostałych kolumn. Bazy danych nie potrafią w tym przypadku wstawić wartości do takiej kolumny.

1.2.3 Możliwość rozszerzenia aktualizacji perspektywy

Należy jednak nadmienić, że dwie bazy danych którymi są Microsoft SQL Server oraz baza firmy Oracle potrafią wykonać powyższe wstawienie rekordu jeśli zdefiniuje się specjalny trigger typu INSTEAD OF. Poniżej znajduje się przykład takiego trigger-a dla języka PL/SQL sprawdzony na bazie Oracle 9i :

```
CREATE TRIGGER trgPanstwaRozszerzone_insert
INSTEAD OF INSERT ON PanstwaRozszerzone
DECLARE
lud NUMBER;
pow NUMBER;
BEGIN
```

```
lud:=:new.Ludnosc;
pow:=:new.Powierzchnia;
IF(lud IS NULL) THEN
lud:=:new.Gestosc * :new.Powierzchnia;
END IF;
IF(pow IS NULL) THEN
pow:=:new.Ludnosc / :new.Gestosc;
END IF;
INSERT INTO Panstwa VALUES(:new.Id, :new.Nazwa, lud, pow);
END;
```

Powyższy trigger jest typu INSTEAD OF czyli trigger który jest specjalnie dla perspektyw. Wykonuje się wraz z zdarzeniem którym jest aktualizacja perspektywy *PanstwaRozszerzone*. Jest on konkretnie dla zdarzenia jakim jest wstawianie (INSERT) nowych rekordów. Triggery tego typu mogą też obsługiwać jeszcze zdarzenia związane z aktualizacją (UPDATE) oraz z usuwaniem (DELETE). W ciele triggera mogą występować dwa specjalne słowa: *new* i *old*, new jest nową wprowadzaną wartością a old starą już istniejącą. Nie dla każdej operacji triggera każde słowo jest wykorzystywane.

Operacja	***Czy wykorzystywane***	
	new	***old***
INSERT	tak	nie
UPDATE	tak	tak
DELETE	nie	tak

Wykorzystanie new i old dla konkretnych operacji dla triggera INSTEAD OF

2 Język SBQL

Język SBQL(Stack Base Query Language) jest nowoczesnym językiem obiektowym bazującym na pojęciu stosu. Język jest przystosowany do przetwarzania i manipulowania na złożonych strukturach danych ze szczególnym uwzględnieniem struktur bazujących na XML. Sam język został zaprojektowany w sposób zachowujący takie elementy jak uniwersalność, naturalność, niezależność od fizycznej organizacji danych, koncepcyjna spójność, łatwość użycia, ortogonalność oraz duże możliwości optymalizacji zapytań.

2.1.1 Modele obiektowe

Obecnie istniejące modele obiektowe są skomplikowane. Dlatego model obiektowy dla SBQL został podzielony na kilka modeli. Każdy kolejny model zawiera w sobie poprzednie przez co stopień skomplikowania pojęciowego rośnie wraz z kolejnym modelem, niemniej jednak daje to efekt rozszerzenia złożoności tylko o pojęcia wprowadzone w tym modelu.

Modele obiektowe:

- M0 - dane półstrukturalne
- M1 - klasy i dziedziczenie
- M2 - interfejsy, dynamiczne role
- M3 - hermetyzacja

M0

Ten model pozwala wyrażać hierarchiczną strukturę danych. Można w nim korzystać z struktur jak i kolekcji. Nie ma w nim natomiast klas oraz rzeczy związanych z dziedziczeniem czy z hermetyzacją. Umożliwia także wykorzystywanie danych półstrukturalnych takich jak w XML-u. Struktury danych występujące w modelu relacyjnym są szczególnym przypadkiem tego modelu.

M1

Model ten uzupełnia model M0 o pojęcie klasy, dziedziczenia w tym wielodziedziczenia. Natomiast nie wnosi hermetyzacji czy pojęcia interfejsów które znajdują się w kolejnych modelach.

M2

Na tym poziomie zostają dodane dynamiczne role oraz interfejsy. Obiektom mogą być dynamicznie dodawane role jak i usuwane. Konkretna rola może powtórzyć się więcej niż raz.

M3

Model ten mógłby zostać zamieniony miejscami z modelem M2. Pojęcia które wnosi wymagają tylko modelu M1 a nie M2. Hermetyzacja dodana na tym poziomie powoduje że do pewnych własności obiektów nie jesteśmy wstanie się dostać z zewnątrz.

2.1.2 Podejście stosowe dla języka SBQL

W języku SBQL są używane dwa stosy w celu wykonania zapytań:

- QRES - stos rezultatów
- ENVS - stos środowiskowy

Pierwszy z nich QRES jest znany z innych języków programowania, na nim są odkładane wyniki operacji oraz używane w kolejnych, natomiast stos ENVS jest odpowiedzialny za wiązanie nazw.

2.1.3 Stos QRES

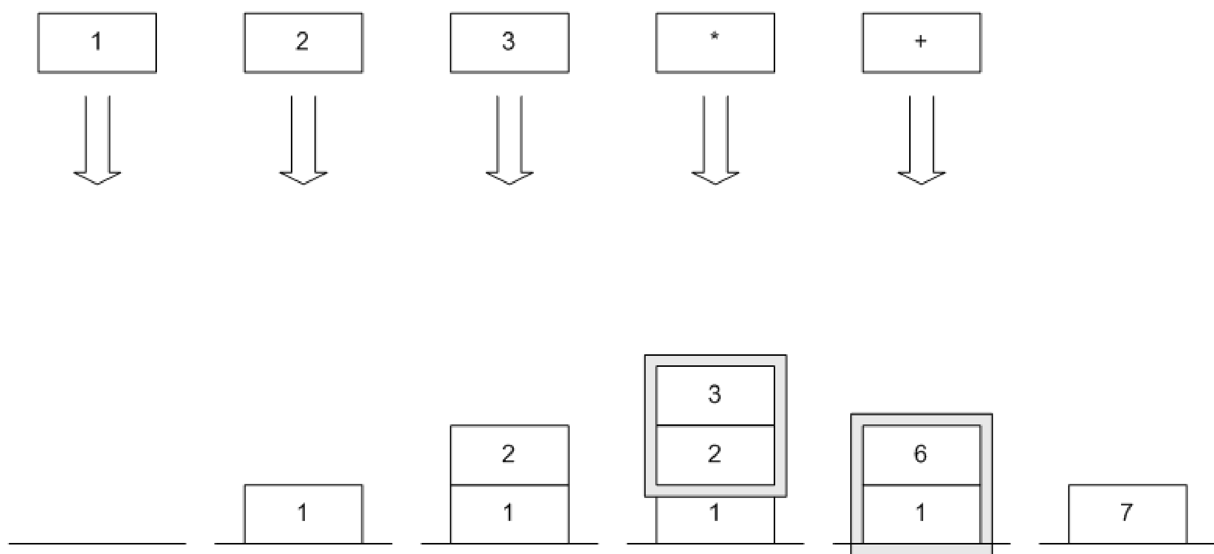
Nazwa QRES zdefiniowane w [Subieta04] pochodzi od angielskich słów **Q**uery **RE**sult **S**tack. Na tym stosie odkładane są wszelkie wyniki operacji, także wyniki pośrednie. Stos ten jest uogólnieniem stosu arytmetycznego używanego w innych językach programowania. Stos ten dostarcza operacji:

- push - kładzie nowy element na stosie
- pop - pobiera element z wierzchołka stosu
- top - jak pop ale bez zdejmowania elementu

- empty - sprawdza czy stos jest pusty

Przykład dla wykorzystywania stosu rezultatów dla prostego wyrażenia *1 + 2 * 3*

Po przetworzeniu do formatu w odwrotnej notacji polskiej *1 2 3 * +* na stos są kładzione kolejne elementy wraz z odpowiednimi dla nich akcjami. Na początku stos jest pusty, zostają kolejno po sobie położone(push) elementy 1, 2, 3. Następnie jest wykonywany operator mnożenia, powoduje on zdjęcie(pop) z stosu dwóch elementów wykonanie na nich swojej operacji jaką jest mnożenie i odłożenie wyniku tej operacji na stos. Po operatorze mnożenia wykonywany jest operator dodawania, który zdejmuje dwa elementy stosu, dodaje je do siebie i wynik umieszcza z powrotem na stosie. W efekcie na zakończenie na stosie otrzymujemy rezultat końcowy całego wyrażenia.



Rysunek 7: Mechanizm działania stosu rezultatów

Elementami stos QRES będą rezultaty zapytań, a także wszelkie pomocnicze elementy potrzebne do prawidłowego wykonywania się zapytań jak np. liczniki pętli.

2.1.4 Stos ENVS

Nazwa ENVS zdefiniowana w [Subieta04] pochodzi od angielskich słów **ENV**ironment **S**tack. Stos ten jest odpowiedzialny za wiązanie nazw. Składa się z bloków które zawierają bindery potrzebne do wiązania nazwy z potrzebnym obiektem.

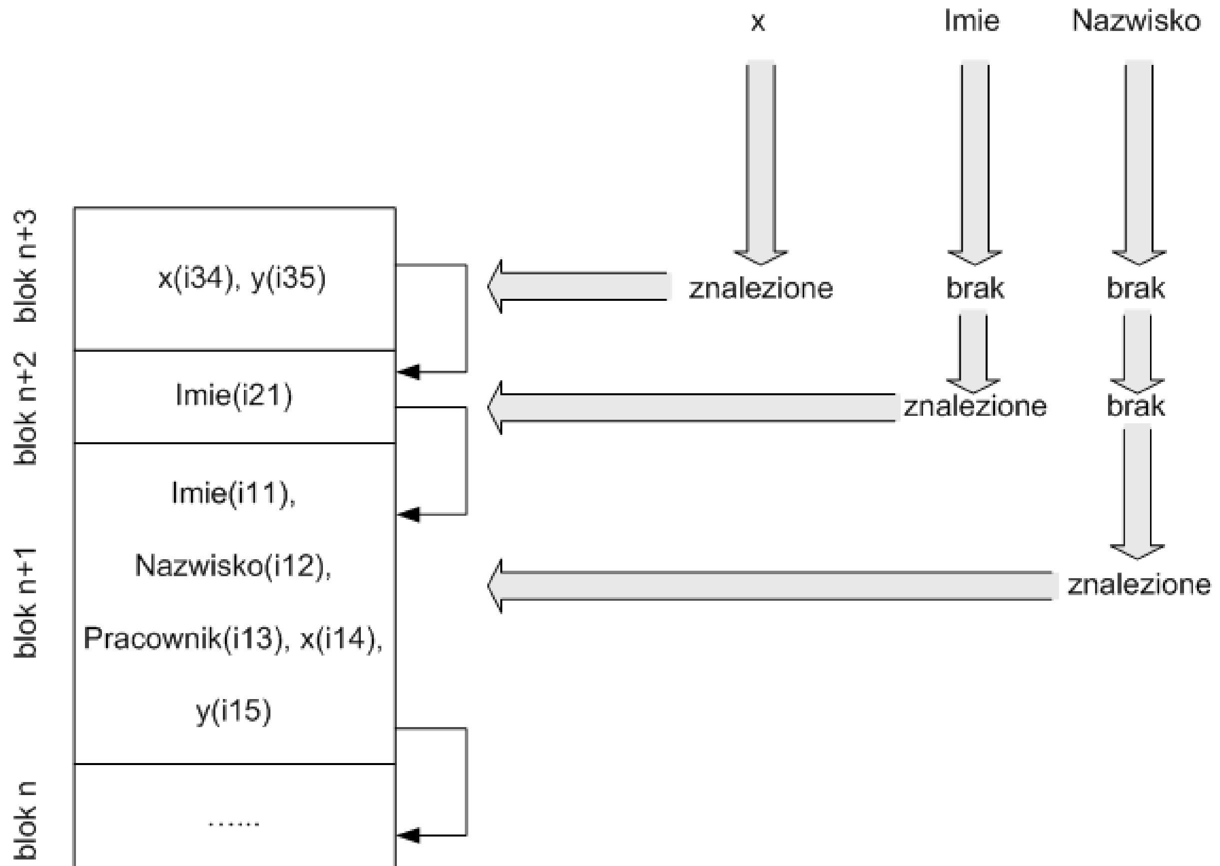
Binder zdefiniowany w [Subieta04] jest strukturą przechowującą połączenie nazwy zewnętrznej oraz obiektu przyporządkowanego do danej nazwy. Nazwa zewnętrzna może być nazwą zmiennej, stałej, procedury, funkcji, metody, klasy, perspektywy lub dowolnego

obiektu.

Stos jest przeszukiwany zawsze od bloku położonego najwyżej, następnie jeśli tam nie zostanie odnaleziona szukana nazwa są przeszukiwane kolejne bloki. Stworzenie nowego bloku na szczycie stosu może powstać dzięki wykonaniu funkcji nested. Ta funkcja dla danego składu i dla konkretnego obiektu generuje wewnętrzne środowisko tego obiektu, które można umieścić na wierzchołku stosu.

Przykład wiązania nazw:

Dla stosu pokazanego na rysunku 8 próbujemy związać nazwę *x*. Przeszukujemy pierwszy blok. Tam znajduje się binder wiążący szukaną nazwę *x* z obiektem co kończy wiązanie choć dwa bloki niżej występuje też binder wiążacy nazwę *x*. Kolejno chcemy związać nazwę *Imie*, przeszukujemy pierwszy blok i bindery z tego bloku tej nazwy nie zawierają więc przeskakujemy do bloku niżej, tam wśród binderów znajdujemy poszukiwany. Ostatnią nazwę którą chcemy związać jest *Nazwisko*, przeszukujemy pierwszy blok -nie znajdujemy tam bindera łączącego obiekt z szukaną nazwą. Analogicznie postępujemy z każdym następnym blokiem. W kolejnym bloku znowu nie znajdujemy szukanego bindera, ale w jeszcze następnym już istnieje binder łączący obiekt z nazwą *Nazwisko* - co kończy proces wiązania nazwy.



Rysunek 8: Przykład wiązania nazw

2.2 Zapytania

Zapytania występujące w języku SBQL mogą być reprezentowane przez:

- literały reprezentujące wartości atomowe np: 2, "Ala ma kota", true
- dowolne zmienne np: x, y
- dowolne nazwy użyte do konstrukcji schematu

Powyższe zapytania mogą być łączone w złożone zapytania za pomocą operatorów. Wyróżniamy dwa typy operatorów:

- algebraiczne
- niealgebraiczne

Przykłady zapytań:

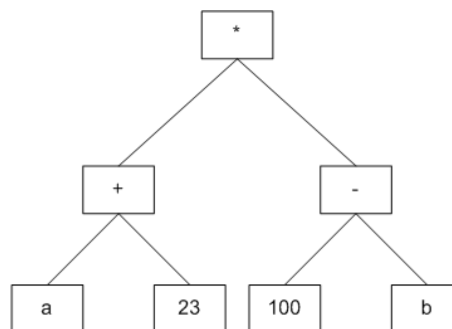
- "Ala ma kota"

- (a + 23) * (100 - b)
- a <100
- a = 23
- a := 23 +100
- Osoba.Nazwisko
- Osoba where (wydatki > (zarobek + premia)* 1.5)

Operatory algebraiczne

Operatory typu algebraicznego nie zmieniają stanu stosu środowisk. Powoduje to iż dla operatora Δ dla zapytania x_1 Δ x_2 kolejność wykonania podzapytań x_1 i x_2 może być dowolna ponieważ są od siebie niezależne. Możliwość takiej zamiany kolejności daje możliwości językowi zapytań takie jak możliwość lepszej optymalizacji, redukcji martwych podzapytań czy wykorzystania przetwarzania równoległego dla zapytania.

Przykładowo dla zapytania (a + 23) * (100 - b) operacje algebraiczne jak dodawanie i odejmowanie mogą zostać wykonane w dowolnej kolejności ponieważ nie zmieniają one stanu stosu.



*Rysunek 9: Drzewo dla zapytania (a + 23) * (100 - b)*

Dodatkową zaletą zapytań w SBQL jest posiadanie semantyki niezależnej od kontekstu[1]

Przykłady operatorów algebraicznych zdefiniowanych w [OdraManual] : +, -, *, /, %, and, or, not, =, <>, >, <, >=, <=

1 Zależności zapytań od kontekstu występują w języku SQL

2.2.1 Operatory niealgebraiczne

W przeciwieństwie do operatorów algebraicznych dla operatorów niealgebraicznych kolejność wykonania ma istotne znaczenie ponieważ każdy z nich modyfikuje stan stosu, co implikuje możliwość różnego zachowania się w zależności od kolejności ich wykonania.

Dla zapytania $x_1 \Delta x_2$ przy wykonaniu zapytania w kolejności x_1, x_2 stan stosu oznaczany jako s poddawany jest modyfikacją:

$s_1 := x_1(s_0)$

$s_2 := x_2(s_1)$

W przypadku gdy kolejność to x_2, x_1

$s_3 := x_2(s_0)$

$s_4 := x_1(s_3)$

Ponieważ operatory są niealgebraiczne to stan stosu po wykonaniu w różnej kolejności nie musi być taki sam jak to miało zawsze miejsce dla operatorów algebraicznych.

Przykłady operatorów niealgebraicznych zdefiniowanych w [OdraManual]: .,join,where,forall,forsome,orderby, closeby

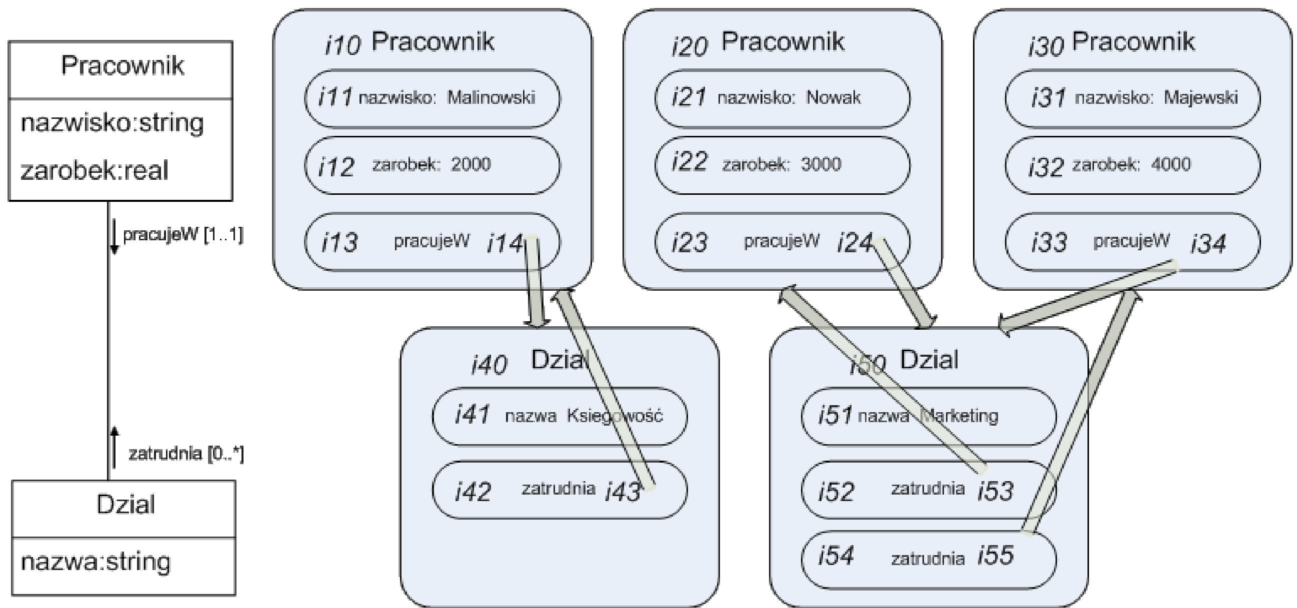
2.3 Przykład wykonania zapytania z uwzględnieniem zawartości stosów

Prześledźmy wykonanie wraz ze zmianami na stosach QRES i ENVS przy zapytaniu[2]:

```
Dzial join avg((zatrudnia.Pracownik).zarobek)
```

Przy założeniu iż w bazie danych znajdują się poniższe obiekty:

2 Na podstawie przykładu z [Subieta04]



Samo zapytanie możemy rozłożyć następująco:

zapytanie := q1 **join** q2

q1 := Dzial

q2 := **avg**(q3)

q3 := q4 **.** q5

q4 := q6 **.** q7

q5 := zarobek

q6 := zatrudnia

q7 := Pracownik

Przed rozpoczęciem wykonania zapytania stos QRES jest pusty, na stosie ENVS znajdują się bindery do Pracownik(i10), Pracownik(i20), Pracownik(i30), Dzial(i40), Dzial(i50). Rozpoczyna się wykonanie zapytania - operator join wykonuje zapytanie q1 w efekcie czego na stosie QRES pojawia się bag który zawiera identyfikatory do obiektów Dzial. Następnie operator join wykonuje dla każdego elementu tego bag-u zapytanie q2 w środowisku wewnętrznym dla tego elementu. Nim zapytanie q2 rozpocznie liczenie średniej wykonywane jest zapytanie q3 które wykonuje najpierw zapytanie q4. Zapytanie q4 wykonuje zapytanie q6 które umieszcza na stosie QRES identyfikatory do obiektów zatrudnia. Sterowanie wraca do zapytania q4 które dla każdego elementu odłożonego poprzednio tworzy jego środowisko

wewnętrzne i w nim wykonuje zapytanie q7. Zapytanie q7 zwraca bag z identyfikatorami do obiektów Pracownik. Kończy się wykonanie q4, sterowanie wraca do zapytania q3 które dla każdego identyfikatora wykonuje w jego środowisku wewnętrznym zapytanie q5. Zapytanie q5 zwraca bag z identyfikatorami do zarobek. Kończy się wykonywanie zapytania q3 i jego wynik w postaci bag-u z identyfikatorami trafia do wykonania w zapytaniu q2 liczenia średniej. Policzona średnia jest następnie łączona z wynikami powstałymi po wykonaniu zapytania q1 w efekcie czego na stos rezultatów trafia bag z strukturami zawierającymi działy wraz z średnią zarobków.

2.4 Konstrukcje imperatywne

Język SBQL oprócz cech typowych dla języków deklaratywnych jest wyposażony w możliwości tworzenia konstrukcji imperatywnych podobnych do innych języków obiektowych takich jak C++, Java, C#. W efekcie możliwość tworzenia zapytań zostaje rozszerzona o konstrukcje dające możliwość zmiany stanu bazy danych.

Operacje imperatywne o jakie został rozszerzony SBQL to:

- tworzenie obiektu
- usuwanie obiektu
- przypisanie
- wstawianie nowego obiektu do już istniejącego obiektu

2.4.1 Operacja tworzenia nowego

Podczas korzystania z tej operacji tworzony jest nowy obiekt oraz dodawany do składu obiektów. Stos ENVS zostaje uaktualniony o nowopowstaly obiekt w sposób zależny od podanego typu utrwalania.

```
create [typ_utrwalania] zapytanie
```

Możliwe typy utrwalania to:

- local
- temporary
- permanent

Jeśli nie podano typu przyjmowany jest automatycznie typ permanent. Typ local jest dozwolony tylko dla stosowania wewnątrz procedur. Tworzy zmienne lokalne, które są usuwane po opuszczeniu procedury. Typ temporary powoduje utworzenie obiektu tymczasowego, który nie jest dodawany do składu obiektów trwałych tak jak w przypadku obiektów typu permanent.

2.4.2 Operacja usuwania obiektu

Operację usunięcia obiektu powoduje usunięcie jego nazwy ze stosu ENVS, oraz usunięcie samego obiektu w składzie obiektów. Powyższa operacja jest dokonywana za pomocą zapytania o składni:

```
delete [typ_utrwalania] zapytanie
```

2.4.3 Operacja przypisania

Operacja przypisania powoduje zmianę zawartości obiektu zwróconego przez zapytanie $zapytanie_1$ przez wartość otrzymaną przez zapytanie $zapytanie_2$.

```
zapytanie1  := zapytanie2.
```

2.4.4 Operacja wstawienia

Operacja wstawienia powoduje wstawienie istniejącego obiektu definiowanego przez $zapytanie_2$ do innego obiektu jako jego podobiekt definiowanego przez zapytanie $zapytanie_1$.

```
zapytanie1  :< zapytanie2.
```

2.5 Procedury

SBQL również istnieje takie pojęcie jak procedury. Są to nazwane kawałki kodu zapytań SBQL-a którym można przekazywać parametry jak i otrzymać wartość zwracaną przez nie. Deklaracja procedury zawiera części:

- nazwę procedury
- typ zwracany
- parametry
- ciało procedury

Nazwa procedury powinna być unikatowa w obrębie swojego otoczenia tzn. jeśli procedura jest deklarowana w module to w tym module nie powinno być innego obiektu noszącego tą nazwę. Jednakże inny obiekt o tej samej nazwie może wystąpić w innych modułach czy wręcz w tym samym module jednak pod warunkiem, że będzie on we wnętrzu innego obiektu np: w klasie.

Typ zwracany przez procedurę jest typem jaki będzie miał wynik zwrócony przez kod zapytań znajdujący się w ciele procedury. Jeśli typ zwracany zostanie pominięty przyjmuje się że typem zwracanym jest typ pusty czyli *void*.

Parametry procedury pozwalają zapytaniom w kodzie ciała procedury korzystać z nich jak z lokalnych zmiennych jednakże wartość początkowa tych zmiennych jest ustalane w momencie wywoływania procedury przez zewnętrznie napisany kod który w ten sposób może zmienić zachowanie procedur poprzez parametry a nie tylko poprzez modyfikację zmiennych globalnych[3]

Ciało procedury zawiera kod zapytań przetwarzany w momencie wywołania procedury. Wartość zwracana z procedury jest za pomocą wyrażenia w kodzie zapytania *return* które także dodatkowo kończy działanie procedury.

Składnia dla procedur:

```
proc            ::=     name ( arg_opt ) ret_type_opt { body_opt }
                ;
ret_type_opt    ::=
                        |       : type optcard

                ;
arg_opt         ::=
                |       arg_list
                ;
```

3 Technika programowania w oparciu o wykorzystanie zmiennych globalnych do przekazywania informacji do procedur jest przejawem złego stylu programowania, może on powodować niespodziewane efekty zwane efektami ubocznymi w przypadku wykonywania kodu przez jeden wątek/zapytanie. Natomiast gdy zachodzi sytuacja że wiele wątków/zapytań wykonuje się równocześnie takie podejście do przekazywania parametrów jest zupełnie niedopuszczalne.

```
arg_list        ::=     proc_arg:d
                |       arg_list ; arg
                ;
arg             ::=     name optcard : type
                ;
```

Przykład procedur

Deklaracja:

```
delta(a:real; b:real; c:real; ):real {
        return b*b - 4*a*c;
}
```

Wywołanie:

```
wynik := delta(1.0; 2.3; 4.4);
```

3 Perspektywy w SBQL

W poprzednim rozdziale uwaga była skupiona na fragmentach języka SBQL stanowiących wstęp do perspektyw. W tym rozdziale zostaną przedstawione perspektywy korzystające z elementów opisanych w poprzednim rozdziale.

3.1 Procedury składowe a perspektywy

W języku SQL definicja perspektywy powstawała poprzez podanie specjalnego zapytania z którego pobierane były wszystkie obiekty które zwracała perspektywa. Jeśli rozpatrzymy perspektywę bez możliwości jej aktualizacji to można by było zamienić perspektywę w procedurę której kod zawierał by zapytanie poprzednio używane w definicji perspektywy. Co więcej w takim przypadku kod wykonywany wewnątrz procedury mógłby składać się więcej niż z jednego zapytania co automatycznie rozszerza możliwości dla stosowania procedur. Jednakże przy procedurach składowych pojawiają się problemy z aktualizacją danych.

Połączenie wywoływania procedur oraz traktowanie perspektywy w sposób przezroczysty jest obecne w SBQL-u. Dla każdej operacji możliwej do wykonania na perspektywie jest możliwość napisania specjalnej procedury która będzie wykonywała się w miejsce tej operacji przy jednoczesnym zachowaniu przezroczystości z punktu widzenia kodu wywołującego perspektywę.

Podejście wykorzystywania wywołania procedur w miejsce dokonywanej operacji znane jest też między innymi z właściwości(ang. property) z obiektowych języków jak C++ Builder[4] oraz C#. Jako iż SBQL kładzie nacisk na przetwarzanie kolekcji perspektywy w SBQL-u mają więcej operacji niż właściwości w wymienionych językach. Własności mogą mieć tylko dwie operacje[5]:

- set/read
- get/write

Pierwsza z nich jest wywoływana w momencie gdy w kodzie następuje pobieranie zawartości

4 C++ Builder jest produktem który używa języka C++ rozszerzonym o pewne dodatkowe możliwości, w pracy podkreślane jest że to nie jest język C++ tylko C++ plus dodatkowe rzeczy przez pisanie tego języka jako C++ Builder a nie język C++ którym jest w 100%

5 Dla C++ Builder-a dochodzą specyficzne dodatkowe opcje takie jak wartość domyślna czy opcja co do przechowywania informacji jednakże one nie zwiększają ilości operacji ponadto.

własności, natomiast druga operacja w momencie ustawiania własności.

Przykład dla C#

```
class KlasaCsharp{
        private int _cena;

        public int cena{
                get{
                        return _cena;
                }
                set{
                        _cena = value;
                }
        }
}
```

Wykorzystanie

```
obj.cena = 42;
int x = obj.cena;
```

Przykład dla C++ Builder

```
class CppBuilder{
private:
        int Fcena;

        void __fastcall SetCena(const int value){
                Fcena = value;
        }

        int __fastcall GetCena(){
                return Fcena;
        }

__published:
        __property int cena = {
```

```
                        read = GetCena,
                        write = SetCena};
};
```

Wykorzystanie

```
obj->cena = 42;
int x = obj->cena;
```

Każde z powyższych rozwiązań ma specyficzne warunki do tego aby móc być stosowanym jednakże idee wykonania tej wirtualizacji są te same. Każde odwołanie do własności w C++ Builderze [Mischel97] oraz w C# [Liberty06] zostaje przetłumaczone na wywołanie odpowiedniej funkcji w ich miejsce. Czyli

```
obj->cena = 42;        ⇒    obj->SetCena(42);
int x = obj->cena;     ⇒    int x = obj->GetCena();
```

Kompilator C++ Builder-a dokonuje tego już w momencie kompilacji kodu, co niestety powoduje brak możliwości pobrania referencji do własności gdyż otrzymano by referencję do funcji a nie do zmiennej. W efekcie nie można mówić o pełnej przezroczystości.

Sam proces tłumaczenia można podzielić na kategorie w których tłumaczenie następuje:

- na etapie kompilacji - wczesne
- w czasie wykonania - późne
- mieszane

Proces tłumaczenia wczesnego niesie ze sobą wady opisane powyżej natomiast daje minimalny dodatkowy narzut przez używanie właściwości. Późne tłumaczenie natomiast wymaga oznaczenia że dany obiekt nie jest zwykłym obiektem, ale wirtualnym. To powoduje iż referencja/identyfikator obiektu musi zawierać dodatkowe dane że jest wirtualny i należy go traktować w sposób szczególny. Uzyskujemy wtedy pełną przezroczystość jednak musimy liczyć się z dodatkowymi narzutami związanymi z obsługą takiej wirtualności. Sposób mieszany natomiast polega na wykorzystaniu wszędzie tam gdzie jest możliwe wersji dla wczesnego tłumaczenia, a w każdym innym miejscu wersji dla późnego tłumaczenia.

Perspektywy w języku SBQL wymagają aby proces tłumaczenia był późny lub mieszaną wersją, uzyskuje się to opcje przez stosowanie wirtualnego OID-u[6].

---

6 OID - z ang. Object IDentificator

3.2 Perspektywy, części składowe

W tej części zostaną przedstawione perspektywy dla modelu M0. W następnym rozdziale będą podane rozwinięcia możliwości dla perspektyw.

Na samą perspektywę mogą składać się elementy takie jak:

- nazwa schematu perspektywy
- nazwa wirtualnego obiektu wraz z procedurą obsługi wirtualnego obiektu
- procedura on_retrieve- opcjonalna
- procedura on_new - opcjonalna
- procedura on_update - opcjonalna
- procedura on_delete - opcjonalna
- procedura on_navigate - opcjonalna
- podperspektywy - opcjonalnie
- inne dodatkowe elementy opisane w dalszych rozdziałach

Każda perspektywa posiada swoją nazwę jak i nazwę obiektu wirtualnego. Istnieje propozycja konwencji nazywania nazw zarządczych perspektyw przez dodawanie na końcu nazwy obiektu wirtualnego członu Def np:

- *BogatyPracDef* - obiekt wirtualny *BogatyPrac*
- Czerwony*SamochodDef* - obiekt wirtualny *CzerwonySamochod*
- *XxxxDef* - obiekt wirtualny *Xxxx*

Przez nazwę obiektu wirtualnego użytkownik może odwoływać się do perspektywy w sposób przezroczysty dla niego. Całą perspektywę definiujemy podanie słowa kluczowego view, po nim nazwy zarządczej perspektywy a następnie ciała perspektywy w nawiasach klamrowych. W ciele perspektywy znajduje się deklaracja obiektu wirtualnego z procedurą obsługi wirtualnego obiektu oraz mogą być deklaracje specjalnych procedur: on_retrieve, on_new, on_update, on_delete które będą wykonane w przypadku zaistnienia któreś sytuacji: dereferencja, wstawienie nowego elementu, modyfikacja, usuwanie. Jeśli któreś z procedur jest brak to operacja związana z nią się nie da wykonać. W perspektywie można deklarować podperspektywy. Oprócz tego możliwe jest definiowanie dodatkowych procedur, zmiennych w ciele perspektywy.

Akcja	*Działanie*	*Przykład*
on_retrieve	dereferencja	**on_retrieve** : string {

		return deref(osoba.imie); }

on_new	nowy	on_new(value: string){
		kto := value;
		}

on_update	aktualizacja	on_update(value: string){
		kto := value;
		}

on_delete	usunięcie	on_delete{
		delete osoba;
		}

3.2.1 Procedura on_navigate

Oprócz procedur on_retrieve, on_new, on_update, on_delete można dla perspektyw deklarować jeszcze jedną specjalną procedurę on_navigate. Procedura on_navigate potrzebna jest w momencie gdy potrzebny jest wirtualny wskaźnik wskazujący na obiekt w kontekście wykonania operatorów niealgebraicznych. Składniowo procedura on_navigate jest taka sama jak on_retrieve z tym że ma zwrócić referencje.

Przykład:

```
view BogatyPracownikDef{
	virtual objects BogatyPracownik:record{p:Pracownik;}{
		return (Pracownik where zarobek > 5000) as p;
	}
	view pracujeWDef{
		virtual objects pracujeW:record{d:Pracownik.nazwaDzialu;}{
			return p.nazwaDzialu as d;
		}
		on_navigate:Dzial {
			return (Dzial where nazwa = d);
		}
```

```
            on_retrieve:string {
                return deref(d);
            }
    }
    .....
}
```

3.2.2 Procedura obsługi wirtualnego obiektu

W ciele każdej perspektywy dla obiektu wirtualnego znajduje się procedura obiektu wirtualnego generująca ziarna dla danej perspektywy. Procedura generuje ziarna na których będą dalej wykonywane operacje definiowane przez specjalne procedury które będą wykonane w miejsce tych że operacji w kontekście dla wygenerowanych ziaren. Procedura obsługi wirtualnego obiektu nie przyjmuje żadnych parametrów, jedynie zwraca ziarna dla danej perspektywy.

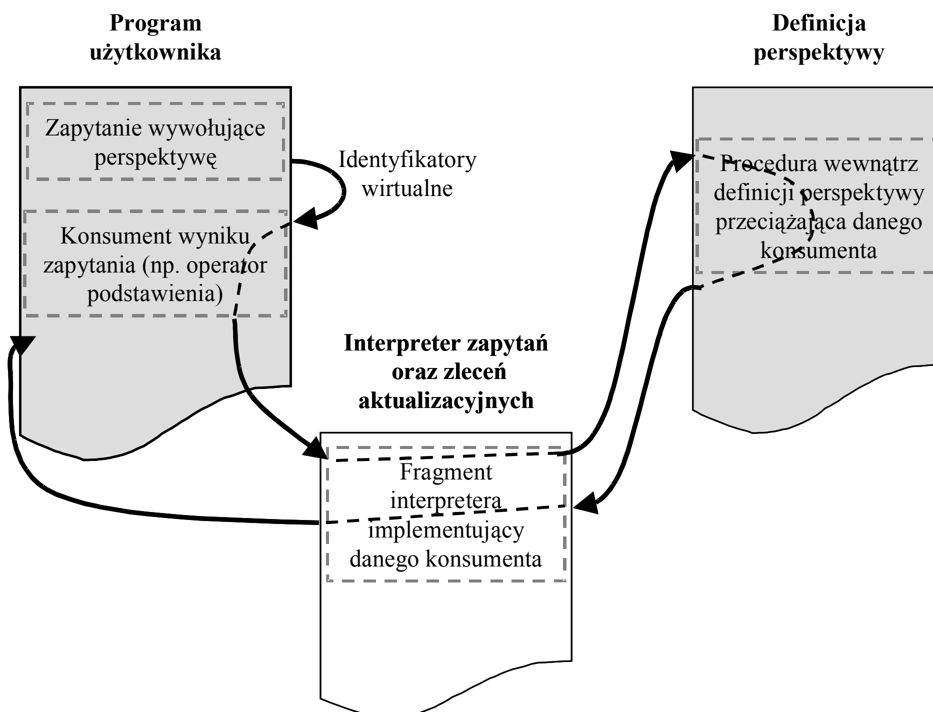
3.2.3 Wirtualny identyfikator

Procedura obsługi wirtualnego obiektu zwraca wynik jakim jest wirtualny identyfikator. Problem jaki wystąpił by w momencie gdyby ta procedura zwróciła zwykłą wartość spowodował by iż nie wiadomo w dalszym toku wykonania zapytania informacji o pochodzeniu tej że wartości z perspektywy. Rozwiązaniem problemu jest wirtualny identyfikator [Subieta04]. Wirtualny identyfikator oznacza iż takiego obiektu nie należy traktować jak zwykły obiekt lecz dla odpowiednich operacji trzeba wykonywać specjalne operacje działające na stowarzyszonych z tym wirtualny identyfikator ziarnami. Dzięki takiemu specyficznemu traktowaniu obiekty wirtualne wygenerowane w procedurze obsługi wirtualnego obiektu po zwróceniu do dalszego przetwarzania mają możliwość odwołania do macierzystej perspektywy z której pochodzą. W celu umożliwienia odróżnienia wirtualnego identyfikatora od innych, wirtualny identyfikator ma postać rozszerzonego identyfikatora i może składać się z: *<flagaJestemWirtualny, nazwaZarządcza, ziarno>* lub *<flagaJestemWirtualny, referencjaDoPerspektywy, ziarno>*. Pierwszy element jest potrzebny do zaznaczenie iż jest to wirtualny identyfikator, kolejny służy do utrzymania więzi z perspektywą w celu dania możliwości wykonywania specjalnych procedur operacyjnych

zawartych w tej perspektywie na ostatnim elemencie w miejsce zwykłych operacji.

3.2.4 Przepływ sterowania podczas korzystania z perspektyw

W celu wykorzystania perspektywy zapytanie wywołuje wirtualny obiekt. Interpreter wykonuje funkcję oznaczoną przez virtual objects co w efekcie daje wirtualne identyfikatory. Dalsze działania są wykonywane na obiektach wirtualnych, na każdym z nich jest wywoływana właściwa procedura dla danej operacji z schematu zarządczego perspektywy gdzie parametry dla danej operacji są przekazywane poprzez odpowiednie akcje na stosie ENVS. Po jej zakończeniu program wraca do dalszej pracy i jeśli procedury zwracały jakieś wyniki to są one dalej uwzględniane przy wykonywaniu dalszej części zapytania.



Rysunek 10: Przepływ sterowania podczas korzystania z perspektyw, schemat wg. [Subieta04]

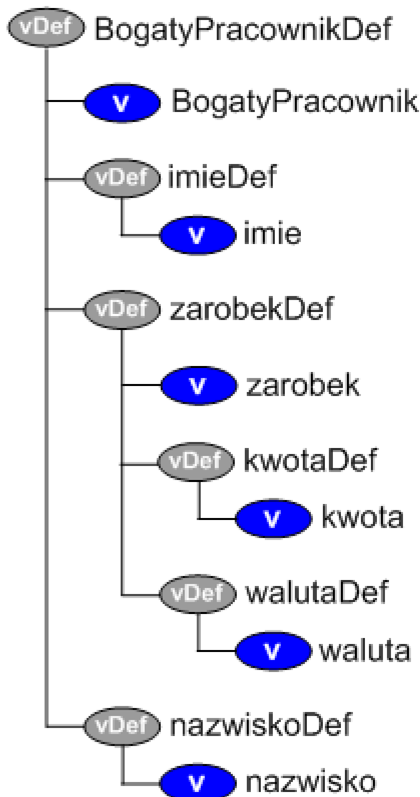
3.2.5 Podperspektywy

Każda perspektywa może zawierać dowolną ilość podperspektyw. Każda taka podperspektywa jest deklarowana w taki sam sposób jak perspektywy które nie są podperspektywami. Jedyne co różni podperspektywę od perspektyw to fakt iż mają dostęp do ziarna obiektu wirtualnego perspektyw znajdujących się nad nimi.

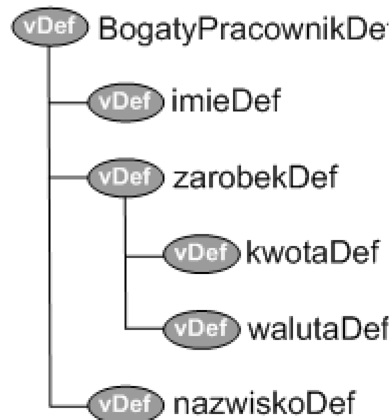
Przykład:

Dla:

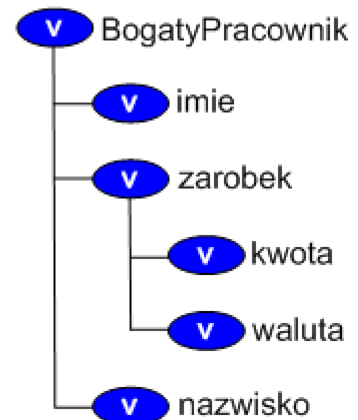
```
view BogatyPracownikDef{
        virtual objects BogatyPracownik:record{p:Pracownik;}{ ... }
        view imieDef{
                virtual objects imie:record{i:string;} { ... }
        }
        view zarobekDef{
                virtual objects zarobek :record{z:Zarobek;}{ ... }
                view kwotaDef{
                        virtual objects kwota:record{k:real;} { ... }
                }
                view walutaDef{
                        virtual objects waluta:record{w:string;}{ ... }
                }
        }
        view nazwiskoDef{
                virtual objects nazwisko:record{n:string;}{ ... }
        }
}
```



Rysunek 11: Struktura/rozmieszczenie w kodzie tworzącym perspektywę z podperspektywami



Rysunek 12: Struktura schematu zarządczego dla definiowanej powyższej perspektywy z podperspektywami



Rysunek 13: Struktura wirtualnych obiektów dla powyższej perspektywy z podperspektywami

3.3 Deklaracja perspektyw

Składnia deklaracji perspektyw wymaga podania po słowie *view* nazwy perspektywy, następnie pomiędzy nawiasami klamrowymi znajduje się ciało perspektywy. W ciele perspektywy mogą występować sekcje typu:

- procedury wirtualnego obiektu
- procedury: on_retrieve, on_new, on_update, on_delete, on_navigate
- podperspektyw
- zmiennych
- procedur
- procedur wewnętrznych

Wszystkie sekcje które zostaną zadeklarowane w perspektywie mogą występować w dowolnej kolejności. Obowiązkowo dla perspektywy musi wystąpić jeden raz sekcja dla wirtualnego obiektu. Sekcje dla procedur on_retrieve, on_new, on_update, on_delete, on_navigate nie są obowiązkowe, ale sekcja danego typu może wystąpić tylko raz. Sekcje dla podperspektyw mogą występować dowolną ilość razy jednakże pod warunkiem iż nazwy dla podperspektyw są różne od innych nazw występujących w perspektywie czyli nazwy: podperspektyw, zmiennych procedur oraz wewnętrznych procedur. Sekcje dla zmiennych, procedur i procedur wewnętrznych zostaną omówione w następnym rozdziale.

Składnia deklaracji perspektywy bez sekcji zmiennych, procedur i procedur wewnętrznych:

```
view                    ::=   VIEW name { view_body }
                        ;

view_body               ::=   view_body_sections view_body_section
                        ;

view_body_sections      ::=
                        |     view_body_sections view_body_section
                        ;

view_body_section       ::=   virtual_objects
                              |       on_retrieve
                              |       on_update
                              |       on_delete
                              |       on_new
                              |       on_navigate
                              |       subview_field
                              |       view_body_other
                              ;

virtual_objects         ::=   VIRTUAL OBJECTS name  : type { stmt_list_opt  }
                              ;

on_retrieve             ::=   ON_RETRIEVE : type  { stmt_list_opt }
```

```
                                        ;

on_retrieve                     ::=     ON_NAVIGATE : type  { stmt_list_opt }
                                        ;

on_update                       ::=     ON_UPDATE ( name : type )  { stmt_list_opt }
                                        ;

on_new                          ::=     ON_NEW ( name : type )  { stmt_list_opt }
                                        ;

on_delete                       ::=     ON_DELETE { stmt_list_opt }
                                        ;

subview_field                   ::=     view_decl
                                        ;
```

3.3.1 Przykłady perspektyw

Przykład 1:

```
view BogatyPracownikDef{
        virtual objects BogatyPracownik:record{p:Pracownik;}[0..*] {
                return (Pracownik where zarobek > 5000.0 ) as p;
        }
        on_retrieve:record{imie:string; nazwisko:string; zarobek:real;} {
                return (deref(p.imie) as imie, deref(p.nazwisko) as nazwisko,
                                deref(p.zarobek) as zarobek);
        }
}
```

Przykład 2:

```
view KubaDef {
```

```
        virtual objects Kuba:integer {
                return 0;
        }
        on_retrieve:string {
                return "Republika Kuby";
        }
        view prezydentDef{
                virtual objects prezydent:integer {
                        return 0;
                }
                on_retrieve:string {
                        return "Fidel Castro";
                }
                on_update(value:string) {
                        System.log(System.getCurrentUser() +
                                        " chciał  dokonać zmiany prezydenta" );
                }
                on_delete {
                        System.log(System.getCurrentUser() + " chciał  usunąć
prezydenta" );
                }
        }
}
```

4 Rozszerzenie możliwości perspektyw

W poprzednim rozdziale została zdefiniowana idea perspektywy oraz sposób jej użytku dla rzeczy stanowiących podstawowe możliwości perspektywy. W niniejszym rozdziale zostaną zaprezentowane rozszerzenia oryginalnie pochodzące z [Subieta04] oraz zupełnie nowe. Rozszerzenia oryginalne:

- zmienne
- procedury - ale nie procedury wewnętrzne

Rozszerzenia nowe:

- procedury wewnętrzne
- konstruktor
- generacja on_retrieve
- generacja podperspektyw

4.1 Zmienne

Sposób oraz działanie zmiennych jest takie samo jak dla zmiennych deklarowanych w module. Różnica polega na widoczności danej zmiennej. Zmienna jest własnością schematu zarządczego perspektywy i poprzez wykonanie operatora kropki na obiekcie jakim jest schemat zarządczy perspektywy można uzyskać dostęp do jego podobiektów - zmiennych. Dodatkowo dla ułatwienia pracy osoby piszącej kod obsługi perspektywy przez procedurę wirtualnego obiektu, procedur on_xxxx i pozostałych procedur wraz z wewnętrznymi, w nich wszystkich są dostępne zmienne dla danego schematu zarządczego bez potrzeby wykorzystywania schematu zarządczego perspektywy w celu dostępu do zmiennych.

Należy zwrócić uwagę iż zmienne należą do schematu zarządczego perspektywy a nie do samego wirtualnego obiektu oraz na fakt iż jeśli występują zmienne w podperspektywach to dostęp do nich z poziomu aktualnie rozpatrywanej perspektywy lub nawet z zewnątrz możliwy jest poprzez podanie jeszcze dodatkowo nazwy schematu zarządczego podperspektywy.

Składnia deklaracji zmiennej[7]:

7 Dla sekcji perspektywy została zdefiniowana jeszcze jedna sekcja view_body_other która zawierać będzie wszelkie dodatkowe elementy w stosunku do rdzenia samej gramatyki dla składni perspektywy

```
view_body_other   ::=   variable
                  |     ...
                  ;
variable          ::=   name optcard : type
                  ;
```

Przykład:

Deklaracja:

```
view BogatyPracownikDef{
        virtual objects BogatyPracownik:record{p:Pracownik;}[0..*] {
                return (Pracownik where zarobek > limit ) as p;
        }
        on_retrieve:record{imie:string; nazwisko:string; zarobek:real;} {
                return (deref( p.imie) as imie, deref(p.nazwisko) as nazwisko,
                 deref(p.zarobek) as zarobek);
        }
        limit:real;
}
```

Dostęp z poza perspektywy:

```
BogatyPracownikDef.limit := 5000.0;
```

Wewnątrz perspektywy BogatyPracownikDef:

```
limit := 5000.0;
```

Natomiast zabronione jest takie odwołanie niezależnie od miejsca występowania czy w perspektywie czy też poza nią:

```
BogatyPracownik.limit:=1000;
```

4.2 Procedury

W ciele perspektywy mogą być deklarowane procedury. Są one używane w taki sam sposób jak procedury deklarowane poza perspektywami. Jednakże znaczenie ich dla perspektyw jest znacznie większe od procedur z poza perspektyw. Procedury deklarowane w ciele perspektywy są z nią związane poprzez sposób ich wywołania, ale również przynależą ze względu na funkcję jaką powinny spełniać dla perspektywy - dostarczać metod operacji na perspektywie.

Dla perspektyw istnieją dwa typy procedur - oba typy deklaruje się wewnątrz perspektywy podobnie jak dla zmiennych.

4.2.1 Procedury zwykłe

Procedury zwykłe to odpowiedniki procedur deklarowanych poza perspektywą. Mogły by być deklarowane nie w perspektywie jednakże takie podejście powoduje iż procedura jest odseparowana od samej perspektywy. Związek iż powinna należeć ona do perspektywy przy takim podejściu jest gubiony, perspektywa lub procedura mogą zostać usunięte z pozostawieniem drugiej. W efekcie w bazie danych zaczną zostawać obiekty które utraciły swoją przydatność a nie zostały usunięte. W celu powiązania procedur z perspektywami których dotyczą zostało założone iż w perspektywach można deklarować procedury.

Deklaracja procedury w perspektywie:

```
view_body_other   ::=    proc
                  |      ...
                  ;
proc              ::=    name ( arg_opt ) ret_type_opt { body_opt }
                  ;
```

Procedury zwykłe dla perspektyw są odpowiednikiem metod statycznych dla klas w takich językach jak Java czy C++.

W języku Java można zapisać taką procedurę jako:

```
class Test{
        public static int dodaj(int a, int b){
```

```
                return a + b;
        }
}
```

Natomiast dla perspektywy w SBQL-u:

```
view TestDef{
        virtual objects test  ...
        dodaj(a:integer; b:integer):integer{
                return a + b;
        }
}
```

Sposób deklaracji w obu przypadkach jest podobny, jednakże w językach takich jak Java możliwe jest zapisanie takiego odwołania:

```
Test obj = new Test();
obj.dodaj(2,2);
```

Natomiast w perspektywach dla SBQL-a jest zabroniony taki zapis

```
test.dodaj(2;2);
```

ponieważ SBQL bazuje na innym mechanizmie niż Java. W przypadku gdyby przyjąć powyższy zapis za poprawny to dla stu wirtualnych obiektów procedura została wykonała przez interpreter sto razy czyli uzyskalibyśmy efekt inny niż oczekiwany[8].

Prawidłowe korzystanie z procedur:

```
TestDef.dodaj(2;2);
```

oraz wewnątrz kodu innych procedur zdefiniowanych w perspektywie:

```
dodaj(2;2);
```

8 Wskazane sposób działania należy też uznać za sensowny, jednakże takie wywoływanie procedur za pomocą nazwy wirtualnego obiektu powoduje wykonanie procedury wirtualnego obiektu która zwróci pewną liczbę wirtualnych obiektów których zawartość była by zupełnie obojętna, a liczyła by się tylko ich ilość. W efekcie wykonanie procedury wirtualnego obiektu staje się nadmiarowe. Wskazany sposób wywołania procedur też nie jest pozbawiony sensu, jednakże powyższy sposób wywoływania procedur oryginalnie został zastrzeżony dla procedur wewnętrznych które mają możliwość wykorzystania pełnych informacji powstałych w takim wywołaniu.

4.2.2 Procedura zwykła - operacje na stosie

Próba wywołania zapytania odwołującego się do procedury zwykłej poprzez nazwę schematu zarządczego przebiega następująco: na stosie ENVS znajduje się binder do schematu zarządczego perspektywy, po znalezieniu jego na stosie QRES identyfikator do niego. Następnie wykonywany jest operator kropki, który dostarcza na stos ENVS elementy składowe perspektywy takie jak procedury zwykłe i zmienne. W przedostatniej ostatniej fazie z wierzchołka stosu znajdowany jest binder odpowiadający wywoływanej procedurze. Dalsze działania na stosie jak przekazywanie parametrów i zwracanego wyniku są takie same jak dla procedur niestowarzyszonych z perspektywą.

4.2.3 Procedury wewnętrzne

Obok procedur zwykłych istnieją procedury wewnętrzne. Deklarowane są one w sposób podobny do zwykłych procedur jednakże poprzedzone słowem *internal* mówiącym że dana procedura nie jest zwykłą procedurą.

Deklaracja procedury wewnętrznej w perspektywie:

```
view_body_other   ::=   iproc
                  |     ...
                  ;
iproc             ::=   INTERNAL name ( arg_opt ) ret_type_opt { body_opt }
                  ;
```

Przykład:

```
view BogatyPracownikDef{
        virtual objects BogatyPracownik:record{p:Pracownik;}[0..*] {
                return (Pracownik where zarobek > limit ) as p;
        }
        on_retrieve:record{imie:string; nazwisko:string; zarobek:real;impremia:real;} {
                return (deref(p.imie) as imie, deref(p.nazwisko) as nazwisko,
                        deref(p.zarobek) as zarobek, obliczPremie() as impremia);
```

```
        }
        limit:real;
        internal obliczPremie():real{
                return obecnaPremia() * p.zarobek;
        }                                              ;
        obecnaPremia():real{
                return 0.1;
        }
}
```

Procedury wewnętrzne porównując do języków takich jak Java, C++ można porównać do prywatnych metod w klasie. Procedury wewnętrzne mają dostęp do ziarna które gra rolę jak dla wyżej wymienionych języków *this* który jest referencją do obiektu.

Procedury wewnętrzne w przeciwieństwie do zwykłych procedur mogą być wywoływane tylko i wyłącznie w procedurach on_retrieve, on_update, on_delete, on_navigate. Za cenę tego ograniczenia jednak procedury wewnętrzne otrzymują dodatkową możliwość jaką jest dostęp do ziarna wirtualnego obiektu. Perspektywy zwykłe nie mogą mieć dostępu ponieważ mogą one być wołane przez nazwę schematu zarządczego perspektywy, w takim wypadku ziarno obiektu wirtualnego nie istnieje co uniemożliwia zwykłym procedurom dostęp do ziarna. Procedury wewnętrzne nie mogą być też wywoływane w procedurze wirtualnego obiektu z powodu iż jeszcze ziarno jeszcze nie istnieje.

Powyższe ograniczenie procedur wewnętrznych można jednak złagodzić o możliwość ich wywoływania na wirtualnym obiekcie. Jednak to rozwiązanie choć prawidłowe może zaciemniać różnicę pomiędzy procedurami zwykłymi a wewnętrznymi co spowodowało zabronienie takich konstrukcji w celu zmniejszenia niejasności. Celem istnienia procedur zwykłych jest możliwość dokonywania operacji z zewnątrz, natomiast wewnętrznych tylko wewnątrz perspektywy.

Należy zaznaczyć że, specjalne procedury[9] jakimi są: on_retrieve, on_new, on_update,

9 powyższe procedury mimo iż są procedurami wewnętrznymi nie są poprzedzane słowem *internal* ponieważ mają specjalne zadania kluczowe dla perspektyw to jednak próbuje się w ten sposób oddzielić je od całej swojej rodziny a także ze względu na ich liczne występowanie powoduje to skrócenie zapisu.

on_delete, on_navigate są procedurami wewnętrznymi, choć są zawsze wywoływane niejawnie.

W miejsce każdego wywołania perspektywy dla otrzymania wartości interpreter dokonuje zamiany z:

```
x := view;
```

w sposób przezroczysty na:

```
x:= view.on_retrieve();
```

W taki sam sposób postępuje dla innych specjalnych procedur. To co robi interpreter w sposób przezroczysty można wprowadzić do języka jako możliwość ręcznego nakazania wykonania specjalnej operacji np:

```
view.on_delete();
```

Jednakże stwierdzono iż taki zapis nie będzie możliwy dla perspektyw dla specjalnych procedur ze względów na trzymanie się wersji iż te procedury są tylko i wyłącznie dla akcji wykonywanych przez interpreter.

Decyzja dotycząca powyższych specjalnych procedur powoduje iż taka sama zasada będzie stosowana do pozostałej części rodziny procedur wewnętrznych i nie będą one możliwe do wywołania w innych miejscach niż tylko procedury wewnętrzne.

4.2.4 Działanie procedur wewnętrznych dla perspektyw wykorzystujących własność stosu

Procedura wewnętrzna może zostać wywołana według przyjętych wcześniej ograniczeń tylko wewnątrz innych procedur wewnętrznych, bądź funkcyjnych.

W momencie odwołania do perspektywy wykonywana jest procedura wirtualnego obiektu, która zwraca wirtualny identyfikator. Ponieważ procedury wewnętrzny są też podobiektami perspektywy również trafiają na stos ENVS. W momencie gdy zajdzie potrzeba wykonania dereferencji lub któreś z operacji aktualizacji wykonywana jest procedura specyficzna dla danej operacji. W kodzie zapytań możliwe jest korzystanie z procedur wewnętrznych. W momencie próby wywołania ich wewnątrz innych procedur wyszukiwane są na stosie i uruchamiane tak jak każde inne procedury – czyli można przekazywać im parametry jak i otrzymywać zwrócony wynik ich działania. To co jest specyficzne

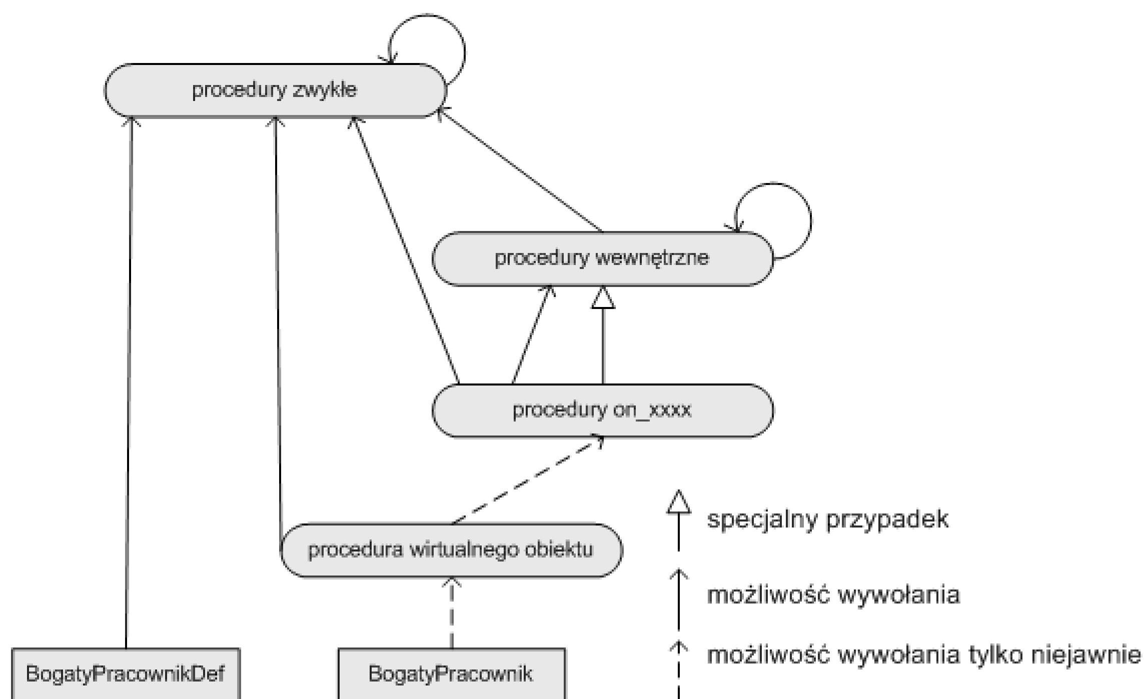
dla procedur wewnętrznych to iż kod zawarty w nich jest wywoływany w specyficznych warunkach, tzn. na stosie odłożone są już wszystkie podobiekty perspektywy wraz z ziarnami i procedury wewnętrzne mają możliwość korzystania z tak przygotowanego środowiska. Ponieważ procedury wewnętrzne mają gwarancję iż na stosie na pewno będą ziarna to mogą z nich korzystać w kodzie ich zapytań.

4.2.5 Porównanie procedur

Możliwości wywołania procedury:

Gdzie	Procedura zwykła	Procedura wewnętrzna
`w wirtualnym obiekcie`	tak	-
on_retrieve, on_new, ...	tak	tak
`w zwykłej procedurze`	tak	-
`w wewnętrznej procedurze`	tak	tak
`przez definicję schematu`	tak	-
`przez nazwę wirtualnego obiektu`	-	*[10]

10 W obecnej wersji przyjęto iż jest to zabronione jednakże nie ma technicznych przeciwwskazań do zezwolenia na takie wywołanie



Rysunek 14: Możliwości wywoływania procedur, niejawne wywoływanie zastrzeżone dla interpretera

Inne różnice:

Cecha	Procedura zwykła	Procedura wewnętrzna
`dostęp do ziarna`	nie	tak

4.3 Podsumowanie zmiennych, podperspektyw i procedur

Każda perspektywa posiada swoją nazwę zarządczą oraz wirtualny obiekt. Oboje mogą posiadać elementy które są nakierowane na przechowywanie stanu czyli zmienne dla schematu zarządczego perspektywy czy symulujące zachowanie zmiennych - dynamiczny stan poprzez podperspektywy dla wirtualnego obiektu.

Oprócz elementów nakierowanych na przechowywanie stanu mogą występować elementy posiadające wykonywalny kod - procedury. Służą one do wykonywania dodatkowych akcji przewidzianych przez programistę. I tak dla schematu zarządczego perspektywy są to procedury zwykłe, natomiast dla wirtualnego obiektu są to procedury wewnętrzne.

„Właściciel”	Przechowanie stanu	Przechowanie kodu procedur

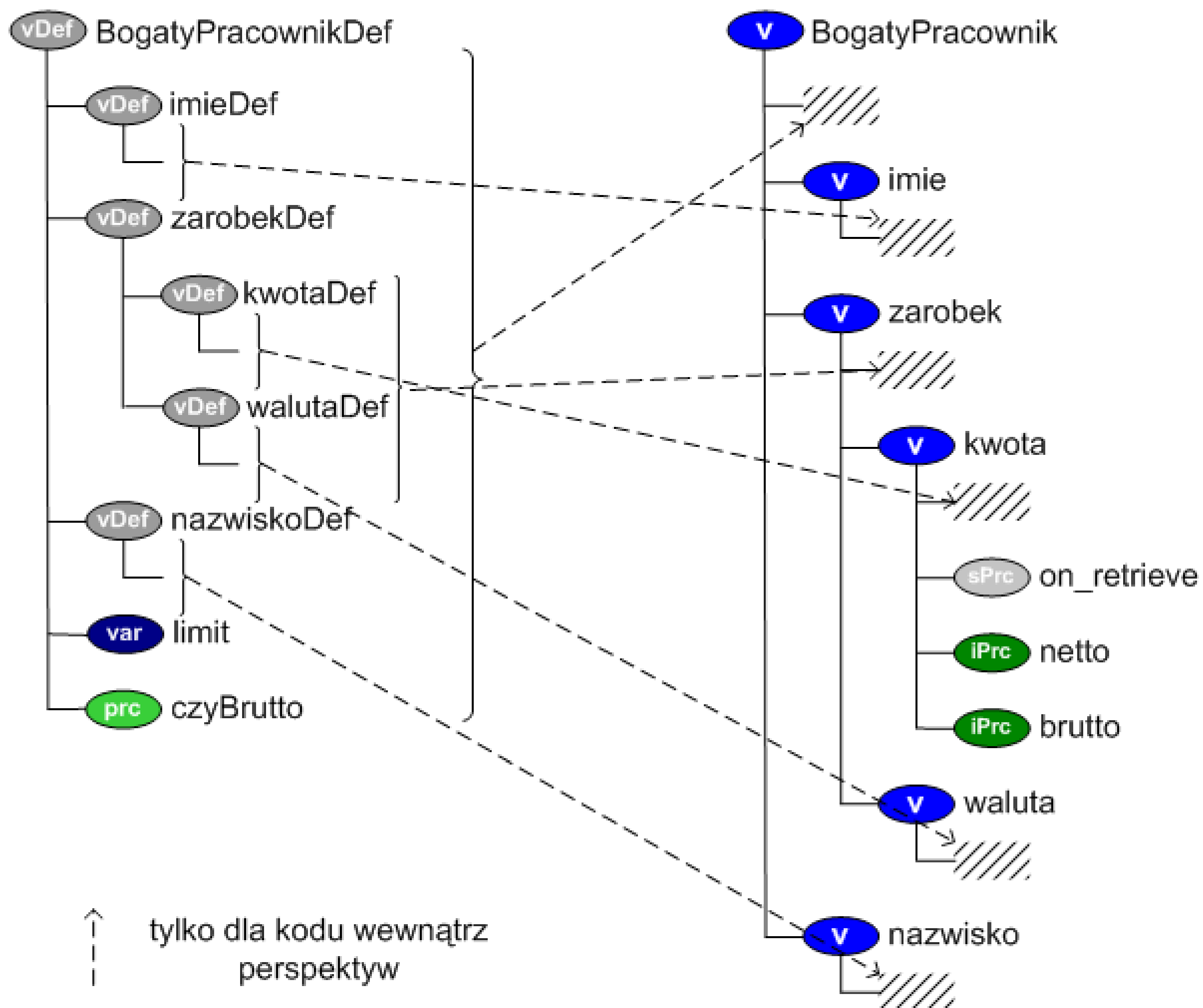
definicja schematu	zmienne	procedury zwykłe
wirtualny obiekt	podperspektywy	procedury wewnętrzne

4.4 Zasięg widoczności dla zmiennych i procedur

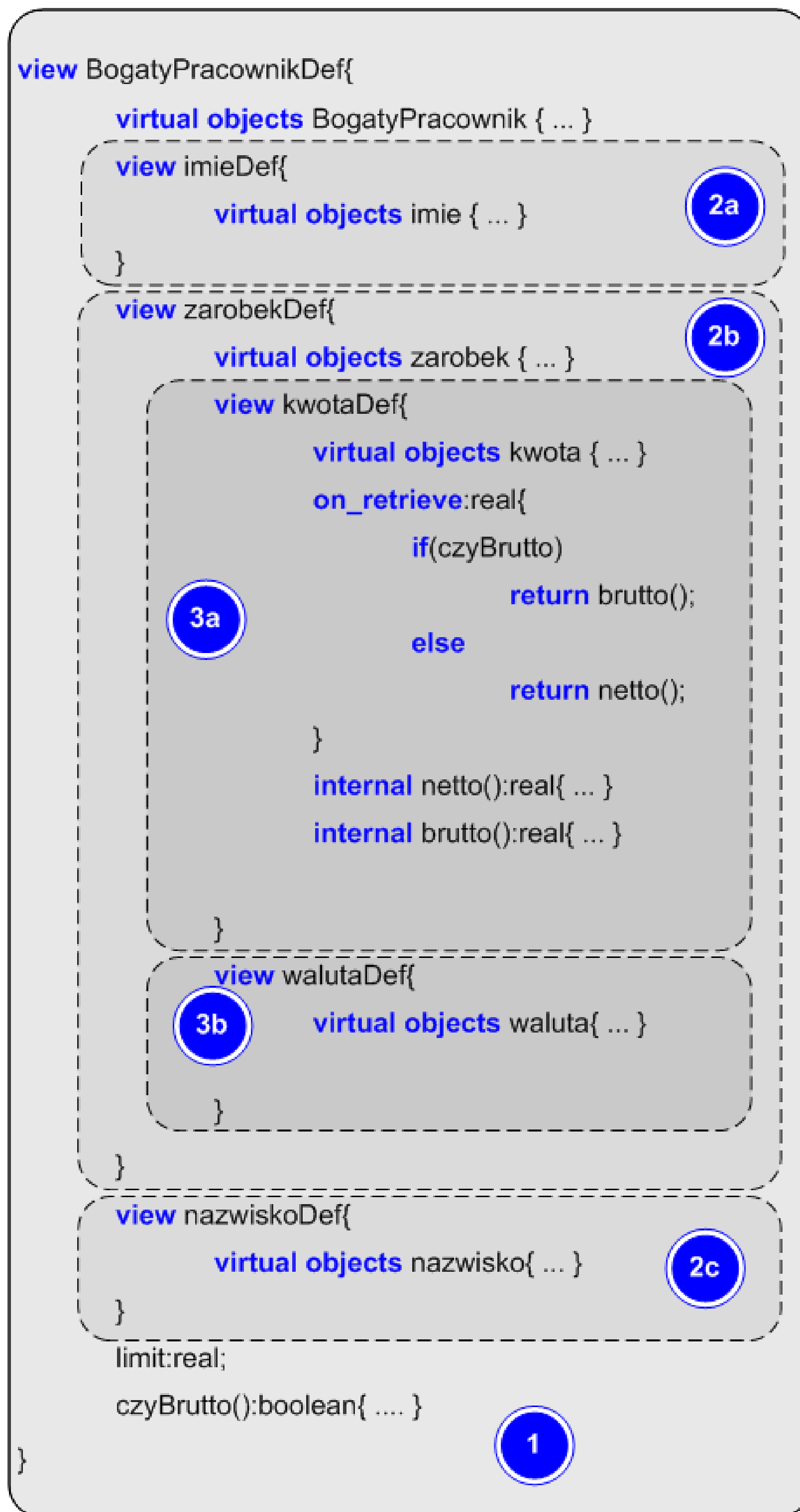
W przypadku gdy występują w perspektywach podperspektywy wszystkie elementy można przydzielić poziomy widoczności. Dla kodu istniejącego poza perspektywami przyjmujemy poziom zerowy. Dla elementów pierwszej perspektywy przyjmujemy poziom pierwszy. Dla każdej z jej podperspektyw postępujemy analogicznie jednakże zwiększając poziom o jeden. Elementy z dalszych poziomów mają automatycznie dostęp do poziomów o niższych numerach. Jeśli element powtarza się na dalszym poziomie to jego nazwa powinna być przyporządkowana dla elementu występującego na dalszym poziomie. Należy zwrócić szczególną uwagę iż struktura dla perspektyw jest w dwóch rodzajach:

- schematów zarządczych perspektyw
- wirtualnych obiektów

Elementy z poziomów dla struktury schematu zarządczego perspektywy nie mają dostępu do struktury wirtualnych obiektów. Natomiast z punktu widzenia struktury wirtualnych obiektów istnieje dostęp do elementów struktury schematu zarządczego perspektywy, jednakże jest on dozwolony tylko i wyłącznie dla kodu zawartego wewnątrz procedury wirtualnego obiektu, specjalnych procedur on_xxxx oraz procedur wewnętrznych. Wszystkie wymienione przypadki mają poziom widoczności zawsze większy od zera. Z zakresu o widoczności zero ten dostęp jest niemożliwy. Każdy wirtualny obiekt dla powyższych przypadków ma dla swojego zakresu widoczności dodane elementy z struktury jego odpowiadającej schematu zarządczego.



Rysunek 15: Dwie struktury dla schematu zarządczego perspektyw oraz wirtualnych obiektów dla poniższego przykładu



Rysunek 16: Przykład poziomów widoczności

Przykład dostępu do elementów:

Z poza perspektywy:

BogatyPracownikDef.limit := 5000.0;

Ponieważ powyższy kod znajduje się poza perspektywą to jego zakres widoczności wynosi zero. W celu dostępu do potrzebnego elementu używamy nazwy schematu zarządczego perspektywy dzięki czemu wchodzimy na poziom pierwszy. W zakresie dla tego poziomu są elementy: imieDef, zarobekDef, nazwiskoDef, limit, czyBrutto. W efekcie znajdujemy szukany element limit.

W kodzie procedury czyBrutto aby dostać się do tego samego elementu nie trzeba pisać *BogatyPracownikDef.limit*, wystarczy samo odwołanie się do elementu ponieważ procedura już jest na poziomie pierwszym.

Jeśli znowu potrzebujemy dostępu do elementu *limit*, ale tym razem z procedury wirtualnego obiektu BogatyPracownik to możemy zapisać tak jak byśmy byli na poziomie widoczności zero, ale także mamy możliwość skrótu ponieważ dla wirtualnych obiektów są dołączane elementy zawierane przez odpowiadające im schematy zarządcze - co skutkuje krótkim zapisem do szukanego elementu *limit*.

4.5 Konstruktor

Perspektywa może posiadać zmienne, których wartości można ustalić poprzez wykonanie zapytań ustalających ich wartość. Jeśli z założeń zmienne mają mieć inne wartości niż domyślne to kod inicjujący można zamknąć w procedurze. Jednakże taka procedura była by niepowiązana z perspektywą dla której istnieje. Nasuwającym się automatycznie krokiem dalej jest wstawienie tej procedury do właściwej perspektywy. Jednakże użytkownik może nadal odwołać do zmiennych perspektywy nim procedura zostanie wywołana. W tym celu należy zapewnić iż zawsze przed daniem możliwości odwoływania się do zmiennych procedury zostanie wywołana procedura inicjująca - procedura ta będzie nazywana konstruktorem ze względu na podobieństwo do konstruktów w językach takich jak Java, C++.

Konstruktory dla perspektyw są podobne do konstruktorów w powyższych językach lecz występuje pewna różnica w stosunku do nich. Konstruktor klasy w dla języka Java:

```
class Test{
```

```
        // konstruktor
        public Test(){ .... }
}
```

Utworzenie obiektu, podczas tworzenia obiektu wykonywany jest konstruktor

```
Test obj = new Test();
```

W perspektywach wirtualne obiekty są tworzone w momencie wykonania procedury wirtualnego obiektu, niemniej jednak lepszym określeniem niż są tworzone jest słowo „pozyskiwane”. Mimo iż tak naprawdę wirtualnych obiektów nie było przed ich pozyskaniem, pojęcia jakie mają nam modelować wirtualne obiekty mogły istnieć już wcześniej - sam fakt stworzenia wirtualnych obiektów modelujących te pojęcia można uznać za użycie tych pojęć za pomocą wirtualnych obiektów niż ich stworzenie.

Dodatkowo oprócz faktu że dla pojęć modelowanych przez wirtualne obiekty nie jesteśmy w stanie z poziomu perspektywy podać początku ich istnienia to dodatkowo zmienne które chcemy inicjować są własnością nie wirtualnych obiektów a schematu zarządczego perspektywy. Konstruktory perspektyw porównując z językiem Java są jak konstruktory klas. Java nie posiada takiego pojęcia jak konstruktor klasy jednakże można dokonać inicjalizacji zmiennych statycznych[11] klasy za pomocą poniższego zapisu:

```
class Test{
        public static int x;
        static{
                x = 100;
        }
}
```

Powyższy kod znajdujący się w bloku static zostanie wykonany w momencie ładowania klasy przez wirtualną maszynę javy.

Składnia dla deklarowania konstruktora jest taka sama jak dla procedur zwykłych wewnątrz perspektyw, ale z ograniczeniem co do nazwy - nazwa musi brzmieć *constructor*

11 Można także przypisać im wartość
public static int x = 100;

oraz konstruktor nie może posiadać parametrów. Typ zwracany przez konstruktor jest dowolny lub konstruktor może nic nie zwracać.

Przykład:

```
view TestDef{
        ....
        x:integer;
        constructor(){
                x := 100;
        }
}
```

Konstruktory dla perspektyw działają analogicznie jednakże sam moment wykonania kodu zawartego w konstruktorze może być uzależniony od innych perspektyw. Jeśli zestawimy proces wywoływania konstruktów to można wyróżnić trzy różne podejścia:

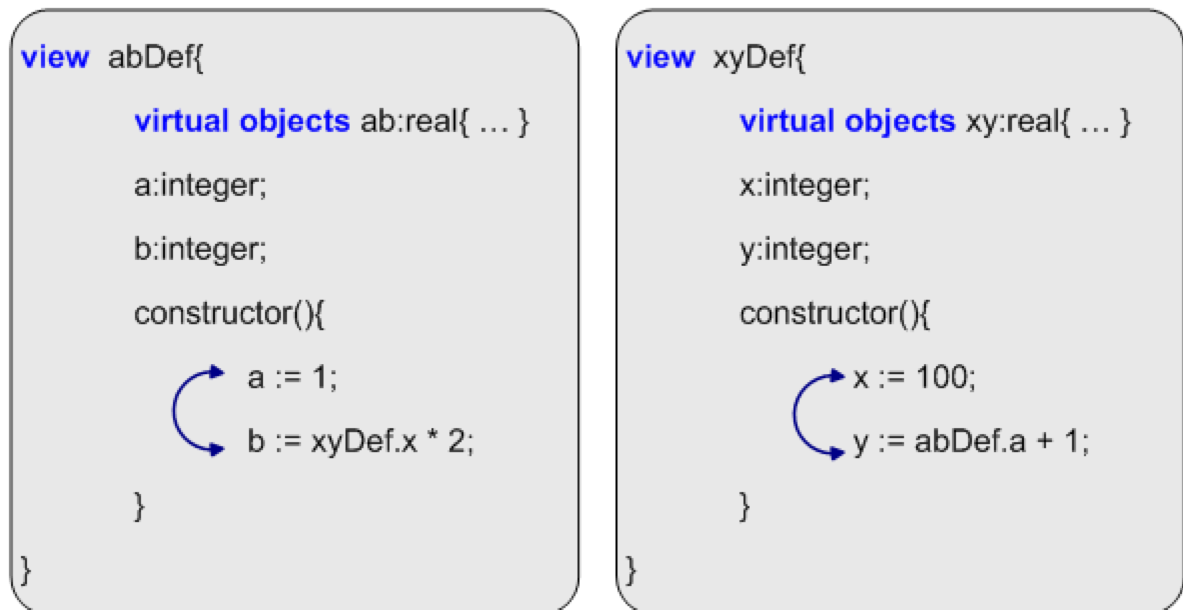
- proste
- z właściwą kolejnością
- opóźnione

Podejście proste polega na wykonaniu wszystkich konstruktorów perspektyw zaraz po utworzeniu modułu w którym znajdują się perspektywy. To podejście ma jednak wadę, mogą istnieć tak skonstruowane konstruktory które powinny być wykonane w odpowiedniej kolejności np. gdy konstruktor korzysta z zmiennej perspektywy którą ustawia konstruktor.

Istotą ulepszonego podejście prostego jest wykonywanie konstruktorów po kolei chyba że następuje odwołanie do perspektywy która ma konstruktor, wtedy przerywane jest wykonanie pierwszego konstruktora i wykonywany drugi. Podejście to choć rozwiązuje poniższy przykład widoczny na rysunku 17, to nie jest i tak w stanie poradzić gdy zostaną zamienione linie oznaczone strzałkami[12].

Przykład:

12 Rozwiązać problem tego typu musi programista



Rysunek 17: Problem wraz z rozwiązaniem dwóch konstruktorów zależnych od siebie

Ostatni sposób postępowania z konstruktorami jest rozwinięciem sposobu poprzedniego, jednakże sam proces wywołania konstruktora zostaje przeniesiony z momentu utworzenia modułu w którym się znajduje do momentu gdy użytkownik będzie chciał skorzystać z perspektywy. Co więcej zostaną wykonane tylko te konstruktory których perspektywy są obecnie potrzebne, reszta będzie czekała aż one też będą potrzebne.

Zaletą wersji opóźnionej jest możliwość zostawienia wykonania konstruktorów na termin późniejszy, niestety pojawiają się też wady:

- błąd który nastąpi podczas wykonania kodu konstruktora zostanie dopiero wykryty w momencie próby wykorzystania perspektywy
- kod konstruktora potrzebuje uprawnień w bazie danych takich jak jego twórca, natomiast użytkownik który spowodował jego wywołanie nie posiada potrzebnych praw

Problem wystąpienia błędu jest na tyle istotny iż jego wykrycie może nastąpić nawet miesiące, lata po stworzeniu perspektywy jeśli przez ten czas nikt nie miał potrzeby jej użyć. Dodatkowo takie nieprawidłowe działanie konstruktora powoduje, że perspektywa nie będzie zachowywała się zgodnie z przewidzianym założeniem. Możliwe są dwa podejścia:

- mimo błędów można korzystać z perspektywy choć może ona dawać informacje niezgodne z założeniami

- zablokować jakikolwiek dostęp do perspektywy

Natomiast problem z brakiem uprawnień można rozwiązać poprzez pozostawienie dodatkowej informacji kto jest twórcą danej perspektywy. Gdy będzie potrzeba wywołania konstruktora do incjalizacji perspektywy to zostanie wywołany konstruktor z prawami twórcy.

Na zakończenie o konstruktorach należy podkreślić że konstruktor to zwykła procedura. Można ją też wywołać jak inne zwykłe procedury. Sensowność takiego wywołania konstruktora na perspektywie która została już zainicjalizowana może być chęć zresetowania perspektywy.

4.5.1 Sposób wprowadzenia zachowania konstruktora do podejścia stosowego

W celu prawidłowego działania konstruktora opóźnionego musi on zostać wywołany przed pierwszą dowolną operacją która jest stowarzyszona z daną perspektywą. W celu uzyskania takiego efektu należy wykorzystać moment odkładania na stosie bindera schematu zarządczego oraz wirtualnego obiektu perspektywy. Wykonanie konstruktora tego typu musi zostać odnotowane w bazie w celu uniknięcia wielokrotnego nieprawidłowego wywołania konstruktora.

4.6 Generacja procedury on_retrieve

Deklaracje perspektyw w języku SBQL wykazują się dużą ilością słów do napisania w celu powstania najprostszej perspektywy. W celu skrócenia zapisu - możliwości perspektywy zostały poszerzone o możliwość korzystania z elips. Jedną z najprostszych elips jest możliwość automatycznego tworzenia procedury on_retrieve w swej najprostszej postaci. Wygenerowana procedura on_retrieve będzie zwracała wszystkie elementy ziaren wirtualnego obiektu, jednocześnie dokonując na nich dereferencji jeśli to jest konieczne. W przypadku chęci zwrócenia innych wartości lub dodaniu jakiś innych elementów do kodu zapytania trzeba własnoręcznie napisać całą procedurę bez korzystania z automatycznej generacji procedury on_retrieve.

W celu skorzystania z automatycznego generowania on_retrieve należy w ciele perspektywy wstawić tekst:

```
on_retrieve;
```

Przykład:

```
view BogatyPracownikDef{
                virtual objects BogatyPracownik:record{p:Pracownik;}[0..*] {
                        return (Pracownik where zarobek > limit ) as p;
                }
                on_retrieve;
                limit:real;
}
```

W efekcie zostanie wygenerowana procedura on_retrieve na podstawie typu Pracownik. Jeśli przyjąć że Pracownik został zadeklarowany jako:

```
Pracownik[0..*]:record{ imie:string; nazwisko:string; zarobek:real; }
```

Procedura przyjmie postać:

```
on_retrieve:record{ imie:string; nazwisko:string; zarobek:real; }{
        return (deref(p.imie) as imie,
                deref(p.nazwisko) as nazwisko,
                deref(p.zarobek) as zarobek );
}
```

4.7 Generacja podperspektyw

Oprócz możliwości generowania automatycznego procedury on_retrieve istnieje możliwość automatycznej generacji podperspektyw. Mechanizm jest zbliżony do generowania on_retrieve jednakże zamiast tworzyć procedurę która zwraca strukturę z każdym elementem ziaren tworzy ona dla każdego elementu ziaren podperspektywę jeśli miała być wygenerowana. Użytkownik podczas deklaracji perspektywy definiuje które podperspektywy mają być wygenerowane. Nazwy podperspektyw muszą być takie same jak elementy ziaren. Zamiast podawać wszystkie elementy można skorzystać z znaku „*” która oznacza wszystkie elementy z ziaren wirtualnego obiektu.

Przykład:

```
view BogatyPracownikDef with imie, nazwisko { ... }
```

Wszystkie pola

```
view BogatyPracownikDef with * { ... }
```

Jeśli w perspektywie występuje już podperspektywa o danej nazwie a miała zostać ona wygenerowana to nie zostanie ona wygenerowana ponieważ już istnieje. Jeśli generowana ma być podperspektywa której nazwy nie ma żaden z elementów ziaren wirtualnego obiektu to jest uznawane to za błąd.

Sposób postępowania prowadzący do ustalenia które podperspektywy mają być wygenerowane:

1. Pobierz listę wszystkich elementów ziaren wirtualnego elementu perspektywy w której mają być dokonane wygenerowanie podperspektyw
2. Z listy podperspektyw do generowania w deklaracji perspektywy weź wszystkie elementy z punktu 1 jeśli na liście jest „*” w przeciwnym wypadku weź tylko elementy wspólne dla obu list.
3. Jeśli któryś z elementów z punktu 1 nie znajduje się na liście powstałej dla punktu 2 to zgłoś błąd i zakończ.
4. Dla listy podperspektyw do generowania powstałej w punkcie 2 usuń podperspektywy które użytkownik sam zadeklarował.

Kiedy faktyczna lista podperspektyw do generacji zostaje ustalona to dla każdej z nich zostaje wygenerowana odpowiadająca jej podperspektywa z procedurą wirtualnego obiektu pobierającą z ziarna nadperspektywy element skojarzony dla perspektywy.

Przykład:

```
Pracownik[0..*]:record{ imie:string; nazwisko:string; zarobek:real; }
view BogatyPracownikDef with * {
        virtual objects BogatyPracownik:record{p:Pracownik;}[0..*] {
                return (Pracownik where zarobek > limit ) as p;
        }
```

```
		limit:real;
}
```

Da w efekcie

```
Pracownik[0..*]:record{ imie:string; nazwisko:string; zarobek:real; }
view BogatyPracownikDef with * {
		virtual objects BogatyPracownik:record{p:Pracownik;}[0..*] {
			return (Pracownik where zarobek > limit ) as p;
		}
		limit:real;
		// wygenerowane
		view imieDef{
			virtual objects imie:record{@imie:strring;}{
				return (p.imie) as @imie;
			}
			on_retrieve;
		}
		view nazwiskoDef{
			virtual objects nazwisko:record{@nazwisko:strring;}{
				return (p.nazwisko) as @nazwisko;
			}
			on_retrieve;
		}
		view zarobekDef{
			virtual objects zarobek:record{@zarobek:real;}{
```

```
                    return (p.zarobek) as @zarobek;
                }
                on_retrieve;
        }
}
```

4.8 Problemy mogące wystąpić podczas generowania podperspektyw oraz procedury on_retrieve

W obu przypadkach automatycznego generowania podperspektyw i on_retrieve może wystąpić problem dla ziaren które posiadają elementy o takich samych nazwach. Powyższe algorytmy postępowania podczas generacji kodu dla tych specyficznych przypadków wytworzą procedury, które mają dwa lub więcej parametrów o tej samej nazwie w jednej procedurze co uniemożliwia sensowne ich rozgraniczenie.

Przykład:

```
Emp[0..*]:record{ name:string; salary:real; }
Company[0..*]:record{ name:string;  budget:real; }
view EmpCompanyDef{
        virtual objects  EmpCompany:record{p:Emp; c:Company;}[0..*] { .... }
        on_retrieve;

}
```

Procedura przyjmie błędną postać:

```
on_retrieve:record{ name:string; salary:real; name:string; budget:real; }{ .... }
```

Problemem w tej procedurze jest wystąpienie konfliktu nazw poprzez podwójne wystąpienie parametru – *name*. Powyższy problem można rozwiązać poprzez modyfikację algorytmów odpowiedzialnych za generowanie kodu poprzez:

- zabronienie automatycznej generacji procedur dla takich sytuacji konfliktowych

- generowanie procedury z pominięciem elementów dla których występują konflikty
- zmiana nazwy dla elementów powodujących konflikty

Pierwsze rozwiązanie jest najbardziej radykalne w rozwiązywaniu konfliktów, jednakże kolejne rozwiązania są bardziej przyjazne dla programisty. Drugi sposób rozwiązania problemu zachowuje co prawda część z swojej przyjazności poprzez brak potrzeby pisania procedur dla przypadków trywialnych, jednakże uniemożliwia dostęp do elementów dla których wystąpił konflikt co jest rozwiązaniem niepełnym. Ostatnie rozwiązanie daje pełną możliwość z korzystania ze wszystkich elementów jednakże konflikt może zostać jedynie rozwiązany poprzez zmianę nazwy na unikalną dla elementów powodujących konflikt. Najlepszym rozwiązaniem jest ustalenie jasnych reguł jakie nazwy dostawać będą elementy dla których wystąpił konflikt. Innym rozwiązaniem jest dodanie specjalnej składni, która definiowała nowe nazwy dla konfliktów przez programistę, jednakże takie dodatkowe konstrukcje dla języka mogą dać w efekcie zmniejszenie czytelności kodu przez co sposób automatycznego nadawania nazw jest zalecany. W przypadku zapotrzebowania na nadanie nazw dla elementów które powodują konflikt niezgodnych z regułami automatycznej zmiany nazw, programista będzie zmuszony do rezygnacji z automatycznej generacji procedur ponieważ jego potrzeby wybiegają poza możliwości takiego rozwiązania.

II Część praktyczna

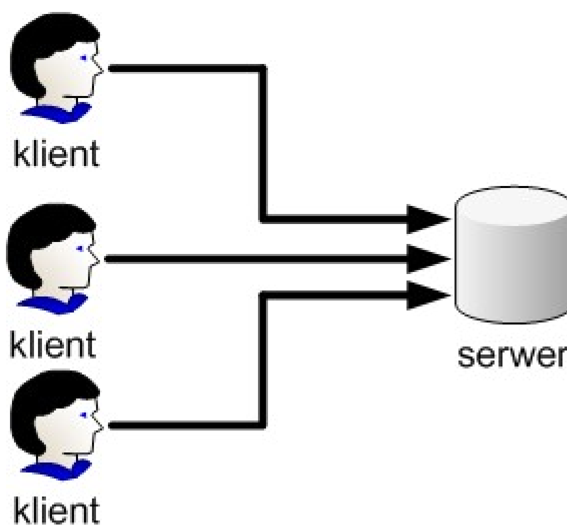
5 Implementacja perspektywy w systemie Odra

W tym rozdziale zostanie zaprezentowana implementacja perspektyw dla języka SBQL, które posiadają pełną możliwość aktualizacji oraz dodatkowe rozszerzenia opisane w poprzednich rozdziałach.

Implementacja prototypu została wykonana za pomocą języka Java SE w wersji 6 oraz za pomocą parsera CUP[CUP] oraz leksera JFlex[JFlex]. Dla dokonania kompilacji kodów źródłowych należy użyć narzędzia jakim jest Ant[Ant]. Kod implementacji perspektywy oparty jest o źródła obiektowej bazy danych Odra J2 tworzonego w ramach projektu eGov-Bus finansowanego w ramach szóstego programu ramowego Unii Europejskiej.

5.1 Architektura aplikacji

Aplikacja jest wykonana w architekturze klient-serwer. Komunikacja pomiędzy klientem a serwerem możliwa jest poprzez sieć oraz aplikacja umożliwia obsługę wielu klientów w tym samym czasie.



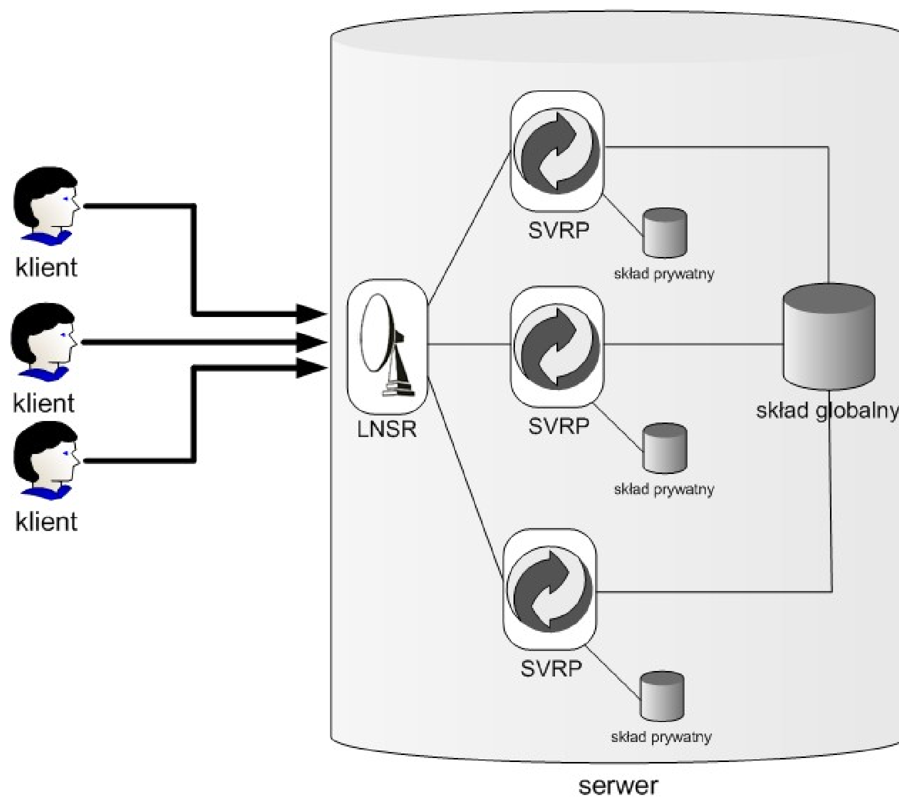
Rysunek 18: Architektura aplikacji

Część aplikacji jaką jest klient złożona jest z tekstowego interfejsu w którym użytkownik dostaje możliwość dokonania połączenia z bazą wykonywania zapytań na tej bazie oraz dokonywanie innych czynności. Działanie klienta opiera się na wysłaniu żądania wykonania zapytania bądź innej czynności a następnie na pokazaniu wyniku przesłanego przez serwer.

Część serwera natomiast zbudowana jest z dwóch części[DocOdra]:

- LSNR - proces komunikacji sieciowej
- SVRP - proces serwera

LSNR jest procesem oczekującym na przychodzące połączenia z sieci od klientów. Po połączeniu klienta z serwerem proces LSNR tworzy dla każdego połączenia nowy proces SVRP reprezentujący klienta po stronie serwera. Dalsza komunikacja klient serwer odbywa się tylko poprzez proces SVRP. Każdy proces SVRP oprócz tego iż pośredniczy w dostępie do bazy danych która jest wspólna dla wszystkich procesów SVRP, posiada również prywatny skład danych do którego tylko on ma dostęp. W tym składzie obiektów mogą znaleźć się obiekty tymczasowe, obiekty sesyjne, obiekty lokalne itd.



Rysunek 19: Procesy SVRP i LNSR

5.2

5.3 Skład danych

Składy danych występujący w prototypie aplikacji Odra J2 można podzielić na dwa typy składu:

- skład trwały
- skład nietrwały

Skład trwały jest to skład globalny dostępny dla każdego procesu klienta, można go utożsamiać z właściwą bazą danych. Natomiast w składach nietrwałych są przechowywane dane które nie będą zapamiętane w sposób trwały np. obiekty sesyjne po zakończeniu sesji użytkownika muszą zostać usunięte.

Każdy skład podzielony jest na trzy warstwy:

- warstwa obiektów bazy i metabazy
- warstwa obiektów składu
- warstwa fizyczna składu danych

Warstwa fizyczna składu danych

Najniższa warstwa składu danych czyli warstwa fizyczna może być kawałkiem pamięci - skład nietrwały lub plikiem zamontowanym w pamięci przez system - skład trwały. W przypadku mapowanego pliku całe zarządzanie pamięcią zostaje przeniesione na system.

Aplikacja zakłada agresywne korzystanie z pamięci operacyjnej tzn. zakłada się iż większość bazy danych jest obecna w pamięci i system decyduje które części bazy mają zostać przesłane do jak i z dysku do pamięci.

Plik bazy danych składa się z nagłówka zawierającego cztery znaki tworzące słowo ODBF i liczby mówiącej o długości całego pliku oraz z ciała które można wykorzystać w dowolny sposób.

nagłówek		ciało
znaki 'O','D', 'B', 'F'	długość	do wykorzystania
4 bajty	4 bajty	n bajtów

Alokację pamięci dokonuje się za pomocą funkcji malloc a zwalnia za pomocą funkcji free. Pamięć jest zarządzana wg. zmodyfikowanego algorytmu sequential-fit [DocOdra].

Dla składu obiektów wielkości kilkuset megabajtów powyższy sposób realizacji jest wystarczający. Prototyp ma ograniczenie wynikające z powyższych założeń co do wielkości bazy danych - wielkość bazy danych musi mieścić się w liczbie czterobajtowej.

5.3.1 Warstwa obiektów składu

Pośrednią warstwą bazującą na warstwie fizycznej jest warstwa odpowiedzialna za obiekty proste, złożone i obiekty referencyjne.

Powyższe obiekty mają format:

rodzaj	nazwa	obiekt nadrzędny	referencje zwrotne	wartość
1 bajt	4 bajty	4 bajty	4 bajty	1, 2, 4 lub 8 bajtów

Gdzie rodzaj to liczba określająca typ obiektu. Nazwa to liczba odpowiadająca nazwie danego obiektu. Liczbę można uzyskać przez dodanie nazwy do indeksu nazw, jeśli nazwa już była wcześniej dodana to zostanie zwrócona liczba przyjęta dla poprzedniego wywołania metody dodawania nazwy. Obiekt nadrzędny to obiekt do którego należy dany obiekt. Pole referencje zwrotne jest wskaźnikiem wskazującym na blok zawierający referencje zwrotne dla danego obiektu. Ostatnie pole wartość w zależności od rodzaju obiektu może mieć różną długość.

rodzaj	wartość	opis
STRING_OBJECT	wskaźnik	Napis, wartość wskazuje na blok pamięci zawierający zarówno długość napisu jak i sam napis
INTEGER_OBJECT	4 bajty	Zawiera liczbę typu integer
DOUBLE_OBJECT	8 bajtów	Zawiera liczbę typu double
BOOLEAN_OBJECT	1 bajt	Prawda lub fałsz
BINARY_OBJECT	wskaźnik	Ciąg bajtów, wartość wskazuje na blok pamięci zawierający zarówno długość jak i sam blok danych
COMPLEX_OBJECT	wskaźnik	Obiekt złożony z serii podobiektów
REFERENCE_OBJECT	wskaźnik	Zawiera wskaźnik do obiektu, rozwiązuje problem wiszących wskaźników
POINTER_OBJECT	wskaźnik	Zawiera wskaźnik do obiektu
AGGREGATE_OBJECT	wskaźnik	Służy do modelowania kolekcji obiektów o tej samej nazwie

Obiekty typu REFERENCE_OBJECT oraz POINTER_OBJECT są obiektami wskazującymi na inny obiekt. Różnica występująca między nimi widoczna staje się w momencie skasowania obiektu do którego prowadzą wskaźniki zapamiętane w obiektach. W przypadku POINTER_OBJECT obiekt wskazuje na nieistniejący obiekt - powstaje problem z zwisającym wskaźnikiem, czyli wskaźnikiem który nigdzie nie prowadzi, albo prowadzi do innych danych które nie powinny być wskazywane przez ten wskaźnik. Ten problem nie występuje w przypadku obiektów REFERENCE_OBJECT. W momencie skasowania obiektu należy usunąć wszystkie referencje wskazujące na niego.

5.3.2 Warstwa obiektów bazy i metabazy

W tej warstwie znajdują się obiekty występujące w bazie i metabazie modułów. Składają się one z obiektów występujących warstwę niżej. Obiekty jakie występują na tym poziomie to zmienne, typy, perspektywy, klasy itd.

5.4 Moduły

W celu zaprowadzeniu porządku oraz dla podzielenia na mniejsze spójne z sobą jednostki baza danych podzielona jest na moduły[13].

Każdy moduł jest niezależną jednostką programistyczną, wspomaga enkapsulację oraz zwiększa możliwości ponownego użycia kodu zawartego wewnątrz modułu. Moduł posiada nazwę, listę importową a także ciało modułu w którym zawarte są wszystkie elementy wchodzące w skład modułu. Celem listy importowej jest danie możliwości korzystania z elementów znajdujących się w innych modułach w danym module.

Moduł może zawierać w sobie inne moduły tworząc w ten sposób strukturę drzewiastą.

Pierwszym modułem jest moduł *sys* wszystkie inne moduły importują go w sposób niejawny. Moduł sys pełni także rolę biblioteki standardowej[14] dostarczając w ten sposób innym modułom elementy systemowe.

Następnie jako moduły dodane do modułu *sys* są dodawane moduły reprezentujące schematy baz danych poszczególnych użytkowników.

---

13 Jest to rozwiązanie podobne do pakietów stosowanych w języku Java

14 Pod tym względem moduł *sys* można porównać do modułu system w języku Pascal.

Rysunek 20:
Przykład modułów

5.5 Podział na bazę i metabazę

Każdy moduł występujący w aplikacji Odra J2 musi składać się z dwóch elementów zawierające informacje dotyczących:

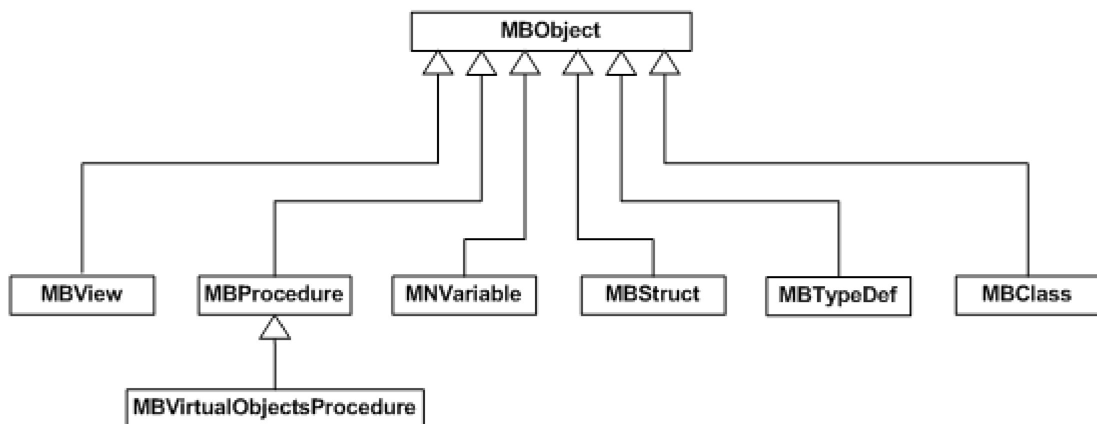
- bazy
- metabazy

Metabaza zawiera informacje potrzebne do przeprowadzenia procesu kompilacji, a także dla dokonania kontroli typologicznej dokonywanej przez typchecker. Baza natomiast zawiera informacje - dane wykorzystywane podczas wykonywania zapytań.

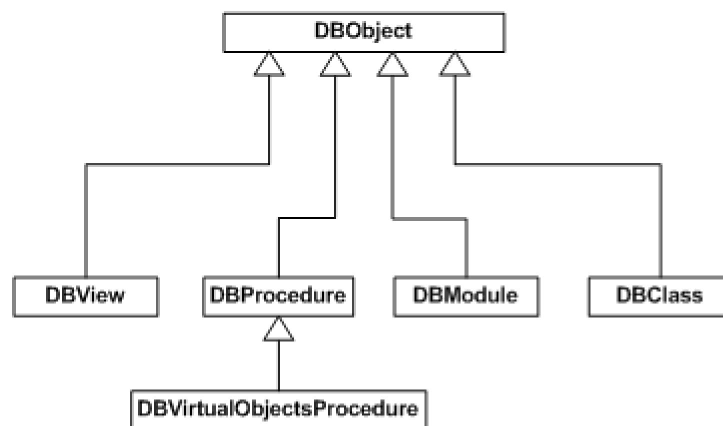
W bazie i metabazie występują obiekty(najważniejsze obiekty):

w bazie	w metabazie	opis
MBObject	DBObject	Wszystkie obiekty po nim dziedziczą.
MBVariable	-	Zmienna
MBStruct	-	Struktura
MBTypeDef	-	Deklaracja typu dla zmiennych
MBClass	DBClass	Klasa

MBProcedure	DBProcedure	Procedura
MBVirtualObjects Procedure	DBVirtualObjects Procedure	Procedura wirtualnego obiektu
MBView	DBView	Perspektywa
-	DBModule	Moduł



Rysunek 21: Obiekty występujące w metabazie



Rysunek 22: Obiekty występujące w bazie

Obiekty występujące w bazie i w metabazie modelują struktury na jakich będzie operował krańcowy użytkownik. Wszystkie te obiekty pośredniczą w dostępie do elementów warstwy pośredniej - wzorzec wrapper[Gamma05].

Obiekty typu MBObject oraz DBObject są korzeniem dla hierarchii wszystkich obiektów dla metabazy i bazy. Każdy z nich przechowuje typ jakim jest obiektem przez co

możliwe jest ustalenie typu z otrzymanych danych z składu.

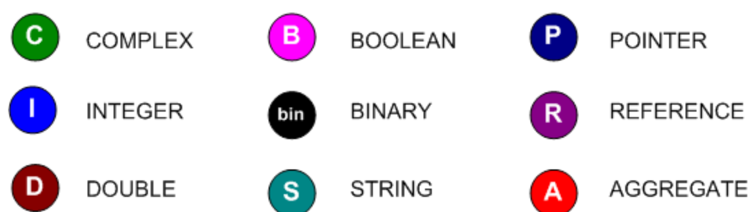
Obiekty typu DBModule modelują moduły, które występują tylko dla bazy gdyż kontrola typologiczna nie obejmuje samego modułu. Moduł daje dostęp dla części bazy i metabazy związanej z elementami występującymi w module. To te elementy podlegają kontroli typologicznej.

Obiekty MBVariable, MBStruct, MBTypeDef są potrzebne tylko do kontroli typologicznej więc występują bez swoich odpowiedników w bazie. W bazie natomiast istnieją dla nich obiekty składu których typy są przedstawiane za pomocą tych obiektów.

Obiekty MBProcedure i DBProcedure modelują procedury których celem istnienia jest przechowywanie kodu zapytań a także umożliwienie jego wykorzystania poprzez wykonanie.

Obiekty modelujące procedury zostają rozszerzone o potrzebne elementy do działania perspektyw, w ten sposób powstają procedury dla wirtualnego obiektu MBVirtualObjectsProcedure w metabazie i DBVirtualObjectsProcedure w bazie. Sama zaś perspektywa jest modelowana za pomocą obiektów MBView w metabazie oraz DBView w bazie.

5.6 Wewnętrzna budowa wybranych obiektów



Rysunek 23: Konwencja co do oznaczeń typów

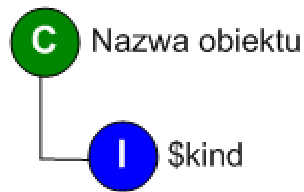
Przy opisie kolejnych obiektów bazy i metabazy będziemy stosować konwencje co do typów z których złożony jest obiekt:

Obiekty systemowe niedostępne wprost dla użytkownika mają w nazwie jako pierwszy znak $ [DocOdra].

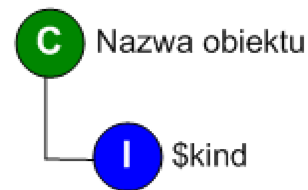
5.6.1 Obiekty podstawowe dla wszystkich innych obiektów w bazie i metabazie

Są to obiekty MBObject i DBObject które nie występują nigdy w bazie. Każdy z obiektów jest obiektem złożonym, zawierającym element typu integer o nazwie $kind

przenoszący liczbę. Powyższa liczba informuje jaki konkretnie jest typ dla danego obiektu. Wszystkie inne obiekty rozszerzają któryś z tych obiektów.



Rysunek 24: Obiekt podstawowy w bazie



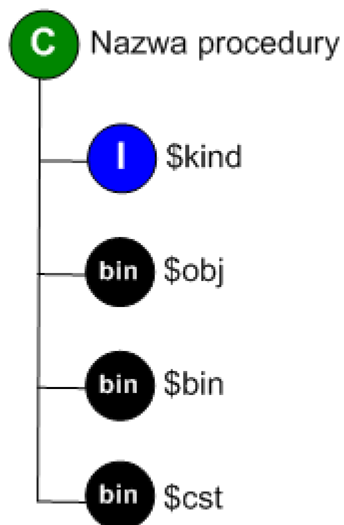
Rysunek 25: Obiekt podstawowy w metabazie

5.6.2 Obiekty dla procedur

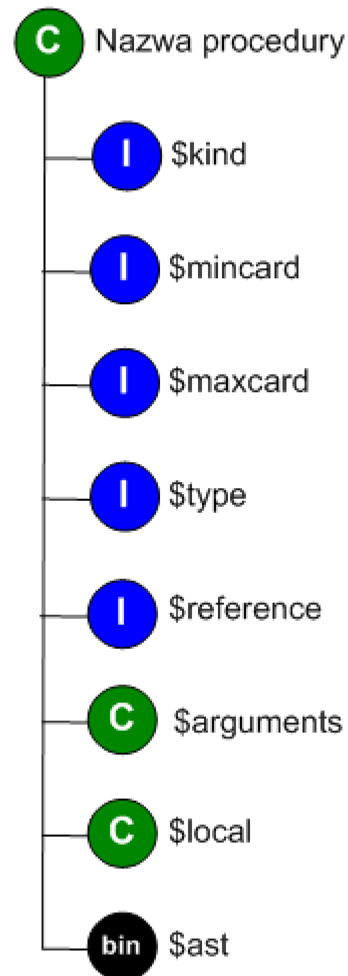
Obiekty modelujące procedury w metabazie przechowują ponad elementy występujące w obiektach podstawowych elementy dotyczące:

- typu zwracanego oraz liczności - $mincard, $maxcard, $type, $reference
- parametrów procedury - $arguments
- zmiennych lokalnych - $local
- serializowany kod procedury potrzebny do procesu kompilacji

Natomiast obiekty dla bazy danych w stosunku do obiektów podstawowych zawierają kod powstały po kompilacji $bin wraz z stałymi $cst oraz mogą zawierać kod pośredni $obj możliwy do wykorzystania w przyszłych implementacjach do optymalizacji kodu.



Rysunek 26: Procedura w bazie



Rysunek 27: Procedura w metabazie

Obiekty związane z perspektywą

Obiekty związane z perspektywą występują oprócz w wersji dla metabazy i bazy w dwóch oddzielnych strukturach:

- w procedurze wirtualnego obiektu
- w schemacie zarządczym perspektywy

Zawsze oba obiekty muszą być powiązane z sobą poprzez wskazywanie na siebie poprzez odpowiednie wskaźniki zarówno w bazie jak i w metabazie. Tymi wskaźnikami są elementy w procedurze wirtualnego obiektu $view, a w schemacie zarządczym perspektywy wskaźnikiem jest $vopproc.

Elementy procedury wirtualnego obiektu zarówno dla bazy jak i metabazy zostały rozszerzone dodatkowo o element $view.

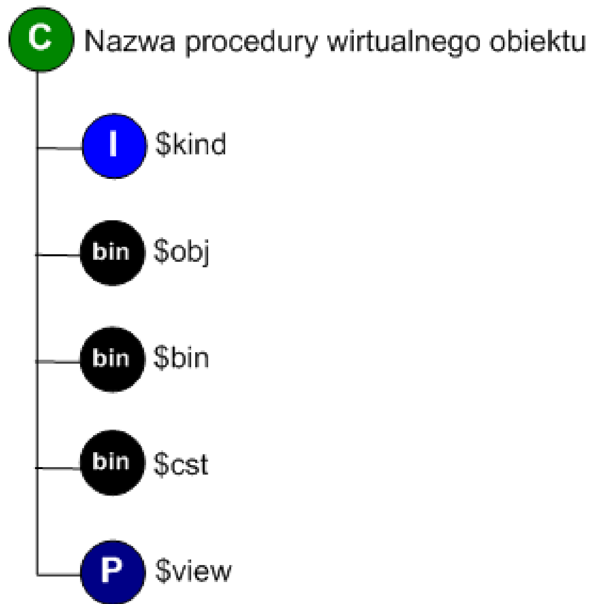
Budowa schematu zarządczego perspektywy w bazie i metabazie ma podobną strukturę z wyjątkiem ostatniego elementu. Ostatnim elementem w bazie jest element $state wskazujący obecny stan w którym znajduje się perspektywa - jest to związane z konstruktorem. Możliwe stany perspektywy:

- UNREADY - perspektywa przed wywołaniem konstruktora
- READY - perspektywa po pomyślnym wywołaniu konstruktora
- FAILED - perspektywa w stanie niemożliwym do użytku, nastąpił błąd podczas wykonania kodu konstruktora danej perspektywy

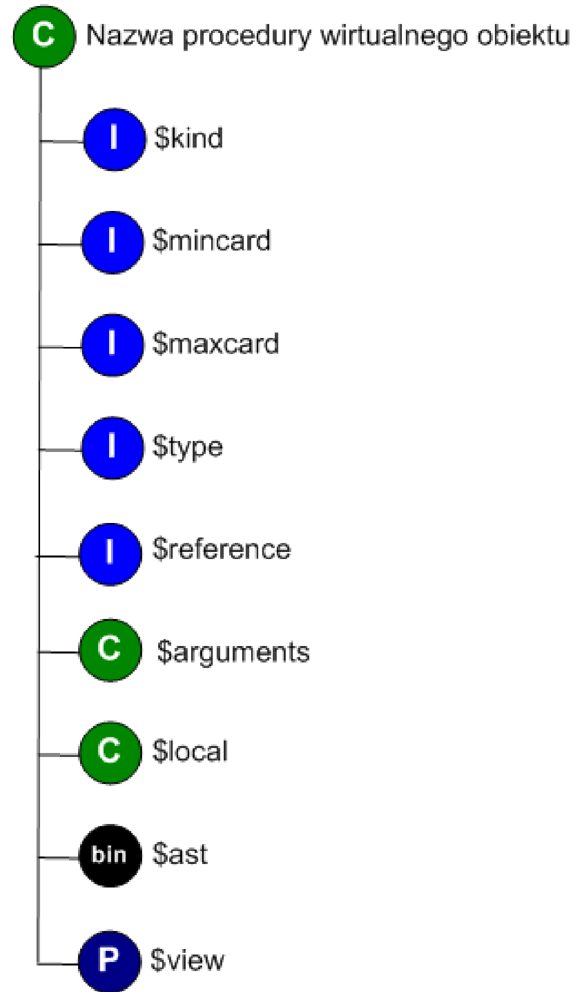
Ostatnim elementem w metabazie jest $genExtra jest to obiekt złożony z obiektów zawierających łańcuch znaków wykorzystywanych do generowania trywialnych podperspektyw.

Część wspólna dla bazy i metabazy dla schematu zarządczego perspektyw zawiera rozszerzenie ponad elementy z obieków podstawowych:

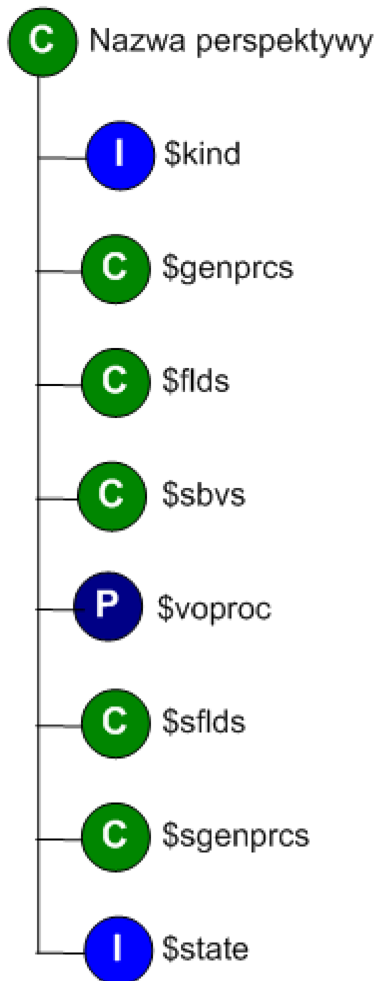
- wskaźnik do procedury wirtualnego obiektu - $voproc
- obiekt złożony przechowujący procedury wywoływane do operacji na wirtualnym obiekcie oraz procedury wewnętrzne - $genprcs
- obiekt złożony przechowujący procedury wirtualnego obiektu dla podperspektyw - $flds
- obiekt złożony przechowujący podperspektywy - $sbvs
- obiekt złożony przechowujący zmienne należące do schematu zarządczego - $sflds
- obiekt złożony przechowujący procedury należące do schematu zarządczego - $sgenprcs



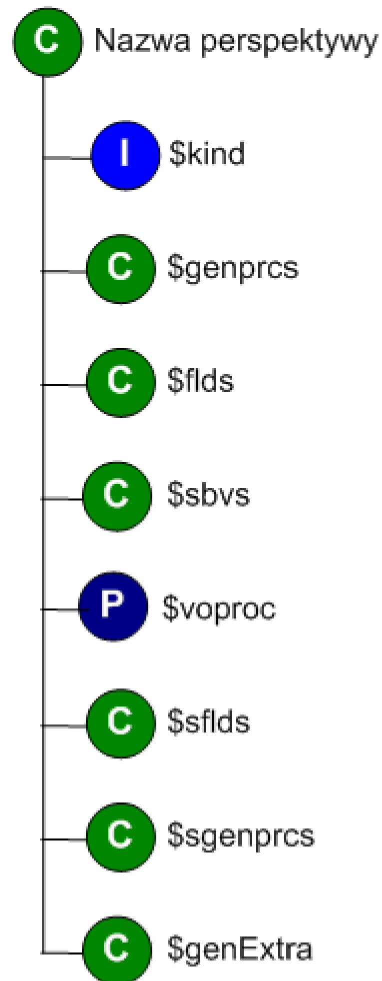
Rysunek 28: Procedura wirtualnego obiektu w bazie



Rysunek 29: Procedura wirtualnego obiektu w metabazie



Rysunek 30: Perspektywa w bazie



Rysunek 31: Perspektywa w metabazie

5.7 Proces tworzenia perspektywy

Tworzenie perspektywy gotowej do użytku dla końcowego użytkownika podzielone jest na dwa etapy:

- Utworzenie perspektywy w module bez dokonania procesu kompilacji dla kodu procedur należących do perspektywy
- Kompilacja kodu procedur zawartych w procedurach występujących w perspektywie

Tworzenie perspektyw możliwe jest za pomocą dwóch dróg:

- wraz z modułem

- dynamiczna

Pierwsza droga wiedzie poprzez tworzenie perspektywy wraz z momentem tworzenia modułu. Następnym etapem wywołanym przez użytkownika powinien być etap kompilacji modułu a wraz z nim także i perspektywy. Jeśli użytkownik zechce wykorzystać dowolny element należący do modułu to przed jego użyciem zostanie skompilowany moduł.

Druga droga - dynamiczna pozwala na dodanie perspektywy do istniejącego modułu poprzez poprzedzenie kodu zapytania tworzącego perspektywę słowem kluczowym *create*. Jeśli moduł nie był wcześniej kompilowany to perspektywa jest tylko dodawana, jeśli moduł został skompilowany to proces kompilacji zostaje ponowiony wraz z dodaniem perspektywy.

5.7.1 Dodanie perspektywy do modułu

Perspektywa jest dodawana do modułu poprzez utworzenie dwóch obiektów złożonych w bazie dla danego modułu oraz dwóch obiektów złożonych w metabazie modułu. Nazwą dla pierwszego obiektu jest nazwa schematu zarządczego perspektywy, nazwą drugiego nazwa wirtualnego obiektu.

Oba obiekty zarówno w bazie jak i metabazie są tworzone tak jak zostało to opisane w części opisujących wewnętrzną strukturę tych obiektów.

Postępowanie dla pary obiektów występujących w metabazie

Struktura utworzonego wirtualnego obiektu w metabazie jest wypełniana odpowiednią wartością dla elementu $kind oznaczającym jego tożsamość. Obiekt wirtualny jest traktowany jak zwykła procedura - tzn. zostają ustawione liczności oraz typ zwracanej wartości, a także kod procedury dla wirtualnego obiektu .

Pary obiektów - schemat zarządczy perspektywy i wirtualny obiekt ustawiane są tak aby wzajemnie na siebie wskazywały przez elementy $view i $voproc.

Struktura utworzonego schematu zarządczego perspektywy w metabazie jest wypełniana odpowiednią wartością dla elementu $kind oznaczającym jego tożsamość Do obiektu złożonego $genprcs dodawane są procedury on_retrieve, on_new, on_update, on_delete, on_navigate w przypadku gdy istnieją one dla danej perspektywy a także procedury wewnętrzne. Jeśli procedura on_retrieve miała zostać wygenerowana to jest ona dodawana z typem zwracanym void oraz nie jest ustalany element $ast dla tej procedury.

Normalne procedury dodawane są do obiektu złożonego $sgenprcs. Zmienne należące do perspektywy są wstawiane do obiektu złożonego $sflds. Do obiektu złożonego $genExtra są dodawane nazwy wszystkich podperspektyw jakie mają zostać utworzone bądź znak * jeśli mają być wygenerowane wszystkie podperspektywy.

Podperspektywy danej perspektywy są tworzone analogicznie jak perspektywy , jednakże zamiast być tworzone w przestrzeni metabazy dla modułu definicje podperspektyw są dodawane do obiektu złożonego $sbvs a wirtualne obiekty tych podperspektyw są dodawane do obiektu złożonego $flds. Dla podperspektyw znajdujących się na kolejnych poziomach zagłębienia sposób postępowania dalej jest analogiczny.

Postępowanie dla pary obiektów występujących w bazie

Struktura utworzonego wirtualnego obiektu w bazie jest wypełniana odpowiednią wartością dla elementu $kind oznaczającym jego tożsamość. Obiekt wirtualny jest traktowany jak zwykła procedura wszystkie elementy binarne są ustawiane jako puste.

Pary obiektów - schemat zarządczy perspektywy i wirtualny obiekt ustawiane są tak aby wzajemnie na siebie wskazywały przez elementy $view i $voproc.

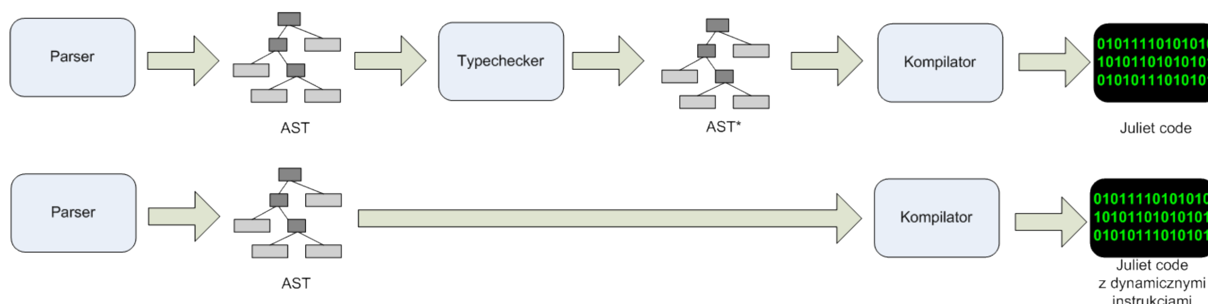
Struktura utworzonego schematu zarządczego perspektywy w bazie są wypełniane odpowiednią wartością dla elementu $kind oznaczającym jego tożsamość Do obiektu złożonego $genprcs dodawane są procedury on_retrieve, on_new, on_update, on_delete, on_navigate w przypadku gdy istnieją one dla danej perspektywy a także procedury wewnętrzne. Jeśli procedura on_retrieve miała zostać wygenerowana to jest ona też dodawana. Normalne procedury dodawane są do obiektu złożonego $sgenprcs. Zmienne należące do perspektywy są wstawiane do obiektu złożonego $sflds, jeśli liczność minimalna dla danej zmiennej jest większa od zera to następuje próba utworzenia tylu zmiennych o danej nazwie ile wskazuje liczność minimalna, których wartości są przyjmowane jako wartości domyślne. Dla schematu zarządczego perspektywy w bazie element $state ustawiany jest na wartość UNREADY.

Podperspektywy danej perspektywy są tworzone analogicznie jak perspektywy , jednakże zamiast być tworzone w przestrzeni metabazy dla modułu schematy zarządcze podperspektyw są dodawane do obiektu złożonego $sbvs a wirtualne obiekty tych podperspektyw są dodawane do obiektu złożonego $flds. Dla podperspektyw znajdujących się na kolejnych poziomach zagłębienia sposób postępowania dalej jest

analogiczny.

5.7.2 Kompilacja perspektywy

W prototypie kompilacja zapytań przebiega według poniższego schematu:



Rysunek 32: Kompilacja zapytań

Kod zapytania zostaje przetworzony przez parser do postaci abstrakcyjnego drzewa składni. Następnie zostaje wykonywana kontrola typów przez typechecker w efekcie drzewo składni zostaje sprawdzone czy nie zawiera błędnych konstrukcji. Jeśli jest wersja dynamiczna to etap z typecheker-em zostaje pominięty i wykonywany jest następny etap kompilacja. Etap właściwej kompilacji generuje na wyjściu dla wirtualnej maszyny kod asemblera - Julietcode[JulietCode]. W zależności od poprzedniego etapu jeśli nie pracował na drzewie typchecker to kod który powstaje zawiera dynamiczne instrukcje, które nie precyzują konkretnie typów, dopiero w czasie wykonania będzie możliwość sprawdzenia ich. W przeciwnym razie dynamiczne instrukcje nie są potrzebne, wszystkie typy są znane.

Proces kompilacji perspektyw zostaje rozpoczęty w momencie gdy proces kompilacji modułu dojdzie do schematu zarządczego perspektywy znajdującej się w metabazie. Dla danego schematu zarządczego perspektywy wyszukiwany jest schemat zarządczy perspektywy w bazie. Tworzone są pary dla danych schematu zarządczego perspektyw z metabazy i bazy. Proces kompilacji danej pary przebiega w trzech etapach:

- kompilacja procedury wirtualnego obiektu
- kompilacja procedur, procedur wewnętrznych a także procedur on_xxxx
- kompilacji podperspektyw

Procedura wirtualnego obiektu nigdy nie jest kompilowana z poziomu kompilacji modułu. Zawsze proces kompilacji procedury wirtualnego obiektu inicjuje kompilacja stowarzyszonego z nią schematu zarządczego perspektywy. Po inicjacji kompilacji procedury

wirtualnego obiektu dokonywana jest kontrola typologiczna drzewa składni pobranego z procedury wirtualnego obiektu znajdującego się w metabazie w elemencie $ast. Podczas kontroli w procedurze jest dozwolone wywoływanie procedur zwykłych oraz dostęp do zmiennych należących do stowarzyszonego schematu zarządczego perspektywy. Ziarna zwracane przez tą procedurę zostają zapamiętane do dalszej kontroli typologicznej innych elementów, a przetworzone drzewo zostaje skompilowane i wynikowy kod trafia do elementu $bin, stałe dla wygenerowanego kodu do elementu $cst w procedurze wirtualnego obiektu w bazie.

W dalszej części proces kompilacji dla pozostałych elementów perspektywy jest kontynuowany. Zostają sprawdzone pod kątem typologicznym przez typechecker oraz skompilowane procedury zwykłe występujące w elemencie $sgenprcs, procedury wewnętrzne i istniejące procedury typu on_xxxx będące w elemencie $genprcs w schemacie zarządczym perspektywy w metabazie do analogicznych elementów znajdujących się w schemacie zarządczym perspektywy w bazie. Dla każdej procedury pobierany jest kod zawarty w jej elemencie $ast, dokonywana jest kontrola typologiczna z uwzględnieniem możliwości korzystania z zmiennych, procedur zwykłych a dla procedur wewnętrznych i on_xxxx korzystania dodatkowo z ziaren wirtualnego obiektu i wywoływania procedur wewnętrznych należących do danej perspektywy. Jeśli element $ast jest pusty a procedura której to dotyczy jest on_retrieve to dokonywane jest generowanie procedury on_retrieve. Drzewo po przejściu przez typecheker jest kompilowane a kod wynikowy trafia do elementu $bin, stałe dla wygenerowanego kodu do elementu $cst.

Ostatnim etapem jest kompilacja dla podperspektyw. Jednak najpierw zostaje wykonana generacja trywialnych podperspektyw jeśli jest to wymagane. Następnie dla każdej podperspektywy znajdującej się w elemencie z $sbvs jest dokonywany analogiczny proces kompilacji w sposób rekurencyjny. Jednak dla procedur znajdujących się w podperspektywach oraz dla procedury wirtualnego obiektu, ale bez procedur zwykłych dodatkowo istnieje możliwość korzystania z ziaren wszystkich perspektyw które są ponad daną podperspektywą.

5.7.3 Generowanie on_retrieve

Podczas procesu kompilacji może wystąpić potrzeba wygenerowania kodu procedury on_retrieve z elementów zawieranych przez ziarna wirtualnego obiektu dla danej

perspektywy, a także ustawienie właściwego typu zwracanego przez procedurę on_retrieve. Porces generacji polega na ustaleniu wszystkich elementów wchodzących do generacji poprzez sprawdzenie typu jaki jest zwracany przez procedurę wirtualnego obiektu, powiązanie każdego z ziaren z tego typu z elementami. Jeśli ziarno jest typem prostym to do tej listy jest dodawane jako puste ziarno z nazwą tego ziarna i z typem ziarna.

Przykład takiej listy dla typu zwracanego przez procedurę wirtualnego obiektu:

```
record{z1:Pracownik; z2:Dzial; opis:string; }
```

gdzie

```
Pracownik[0..*]:record{ imie:string; nazwisko:string; zarobek:real; }
Dzial[0..*]:record{ nazwa:string; gdzie:string; }
```

ziarno	nazwa	typ
z1	imie	string
z1	nazwisko	string
z1	zarobek	real
z2	nazwa	string
z2	gdzie	string
-	opis	string

Kiedy powstanie taka lista tworzony jest nowy typ który jest ustawiany jako typ zwracany przez procedurę on_retrieve. Typ ten jest rekordem i zawiera wszystkie elementy z listy o nazwach takich jakie są w liście i o typach występujących wraz z nimi na liście.

Dla powyższej listy z przykładu będzie to:

```
record{ imie:string; nazwisko:string; zarobek:real; nazwa:string; gdzie:string; opis:string; }
```

Kolejnym krokiem jest wygenerowanie kodu procedury. Kod procedury musi mieć postać:

```
return ( E1 , E2 , E3 .... );
```

lub w przypadku jednego elementu

```
return ( E1 );
```

gdzie każde wyrażenie Ex jest budowane z elementów listy według zasady:

```
deref( ziarno . nazwa ) as nazwa
```

W przypadku elementów prostych można pominąć *deref* oraz nazwę ziarna wraz z kropką.

Gotowy kod zapytania w postaci drzewa zostaje zapisany w elemencie $ast procedury on_retrieve w metabazie. Następnie wymuszany jest ponowny proces linkowania modułu, a po nim kompilacja jest dalej kontynuowana po procesie generacji on_retrieve.

5.7.4 Generowanie podperspektyw

Generacja podperspektyw jest podobna do generowania procedury on_retrieve. Różnica polega na tym, że dla generacji podperspektywy dla jej procedury wirtualnego obiektu jest zwracany jeden element tożsamy dla danej podperspektywy, ustawiany jest typ zwracany przez procedurę wirtualnego obiektu na typ rekordu zawierającego dany elementu, jednak o nazwie jak dany element jednak poprzedzonej znakiem @. Dodatkowo generacja podperspektywy dokonuje też generacji procedury on_retrieve dla podperspektywy.

Dla elementu z listy

ziarno	nazwa	typ
z1	imie	string

zostanie wygenerowany typ zwracany przez procedurę wirtualnego obiektu:

```
record{ @imie:string;}
```

a kod zawarty w tej procedurze przyjmie postać tak jak w on_retrieve wystąpił jeden element jednakże z różnicą iż zamiast nazwy występującej po operatorze *as* zostanie wstawiona nazwa poprzedzona znakiem @.

```
return ( deref( ziarno . nazwa ) as @nazwa );
```

5.8 Użycie perspektywy

W momencie odwoływania do perspektywy poprzez nazwę wirtualnego obiektu zostaje wyszukany w bazie wirtualny obiekt o poszukiwanej nazwie. Następnym etapem jest wykonanie procedury zawartej w wirtualnym obiekcie. W momencie rozpoczęcia wykonania procedury wirtualnego obiektu na stos ENVS zostają odłożone struktury[15] dające dostęp

15 Opisane struktury są kolekcjami binderów

do wszystkich elementów odpowiedzialnych za zmienne oraz procedury zwykłe.

Podczas pobierania tych że elementów zostaje sprawdzone czy dla danej perspektywy był już uruchamiany konstruktor poprzez sprawdzenie pola $state dla schematu zarządczego perspektywy znajdującego się w bazie. Jeśli wartość pola jest READY to wykonanie jest dalej kontynuowane, gdy UNREADY to w nowym środowisku wykonywana jest procedura o nazwie constructor, jeśli taka procedura istnieje. W momencie gdy pole zawierało wartość FAILED lub w trakcie działania konstruktora został zgłoszony wyjątek to pole $state jest ustawiane w stan błędu - wartość FAILED, a w miejsce skąd nastąpiło wywołanie do procedury wirtualnego obiektu jest zgłaszany wyjątek[16]. W innym przypadku ustawiane jest pole $state na wartość READY i kontynuowane wykonanie procedury wirtualnego obiektu.

Dalszy przebieg korzystania z perspektywy przebiega za pomocą wykonania specjalnych kawałków kodu - subroutines pomagających przy wykonaniu specjalnych operacji na zwróconych obiektach przez procedurę wirtualnego obiektu. Odpowiednia subroutine dla danej operacji ma na celu wykonanie specjalnej procedury przechowywanej przez schemat zarządczy perspektywy w bazie a wynik jeśli dana procedura zwraca przekazać do dalszego wykonania kodu gdzie nastąpiło odwołanie do nazwy wirtualnego obiektu. W przypadku braku specjalnej procedury dla danej operacji zgłaszany jest błąd o niemożliwości wykonania tej operacji. W momencie rozpoczęcia wykonywania specjalnej procedury zostają odłożony na stos ENVS struktury zawierające kolekcje elementów dostępne dla procedury. I tak dla procedur potrzebnych do wykonania specjalnych operacji odkładane są elementy zawierające zmienne, procedury zwykłe, procedury wewnętrzne. Dodatkowo zostają także odłożone elementy dla podperspektyw, zarówno nazw wirtualnych obiektów jak i dla nazw schematu zarządczego perspektyw.

Podczas korzystania z procedur zwykłych istnieje możliwość skorzystania z elementów zawierających:

- zmienne należące do danej perspektywy
- procedury zwykłe zawarte wewnątrz perspektywy
- schematów zarządczych podperspektyw danej perspektywy

---

16 Obecnie jest zaimplementowany wyjątek dla języka Java, jednakże w przyszłości wraz z dodaniem obsługi wyjątków do SBQL ten że wyjątek powinien zostać częścią wyjątków języka SBQL

W przypadku korzystania z procedur wewnętrznych ponad to co jest osiągalne dla procedur zwykłych jest możliwe skorzystanie z elementów zawierających:

- procedury wewnętrzne
- ziarna perspektywy jak i ziarna wszystkich perspektyw w których dana perspektywa jest zagnieżdzona

Wszystkie struktury zawierające powyższe elementy są odkładane na stos ENVS w momencie gdy istnieje możliwość odwołania do tych elementów z poziomu kodu znajdującego się wewnątrz procedur zawartych w perspektywie.

5.8.1 Odwołanie do zmiennych i procedur poza wirtualnym obiektem

Odwołanie do elementów zawartych w perspektywie poprzez wykorzystanie nazwy schematu zarządczego perspektywy jest możliwe do:

- zmiennych
- procedur zwykłych
- schematów zarządczych podperspektyw

W momencie odwoływania przez kod zapytania do nazwy schematu zarządczego perspektywy jest wykonywane sprawdzenie czy nastąpiła konstrukcja perspektywy - analogicznie jak dla przypadku kiedy odwołanie następowało poprzez nazwę wirtualnego obiektu.

W dalszym toku dostępu do elementów zawartych w perspektywie na stos zostają odłożone specjalne struktury zawierające te elementy. Poprzez wykorzystanie nazw schematów zarządczych podperspektyw możliwy jest dostęp do elementów zawartych w podperspektywach.

5.9 Korzystanie z prototypu

W celu skorzystania z aplikacji należy utworzyć bazę danych, uruchomić proces serwera dla danej bazy a następnie uruchomić proces klienta i wykonać połączenie z serwerem.

Do wykonania każdej z powyższych czynności należy uruchomić odpowiednią klasę javy wraz z potrzebnymi parametrami przekazanymi przez linię poleceń. Podczas

uruchamiania danej klasy muszą być dostępne dla niej biblioteki zawarte w podkatalogu lib: jodra.jar, cli.jar, common.jar, xmlfilter.jar, cup-runtime.jar, xom-1.1.jar, commons-httpclient-3.0.1.jar, http.jar, wsdl4j.jar, serializer.jar, activation.jar. Powyższe pakiety można dodać na trzy sposoby dla wykorzystania przy potrzebnych czynnościach poprzez:

- dodanie do wywołania uruchomienia klasy javy dodatkowego parametru -cp wraz z ścieżką do potrzebnych pakietów
- dodanie ścieżki do potrzebnych pakietów do zmiennej systemowej CLASSPATH
- skopiowanie potrzebnych pakietów do katalogu lib\ext będącego w katalogu środowiska uruchomieniowego javy

5.9.1 Tworzenie bazy

Tworzenie bazy wykonywane jest poprzez uruchomienie klasy odra.system.Main wraz z parametrami --create *nazwa wielkość* gdzie parametry nazwa i wielkość odnoszą się do nazwy pliku tworzonej bazy danych oraz jego wielkości.

Przykład:

```
java odra.system.Main --create test.dbf 3000000
```

5.9.2 Uruchamianie procesu serwera

Uruchomienie procesu serwera dla bazy danych wykonywane jest poprzez uruchomienie klasy odra.system.Main wraz z parametrem --start *nazwa*gdzie parametr nazwa jest nazwą pliku zawierającego bazę danych.

Przykład:

```
java odra.system.Main --start test.dbf
```

5.9.3 Uruchamianie procesu klienta

Uruchomienie procesu klienta wykonywane jest poprzez uruchomienie klasy odra.cli.CLI.

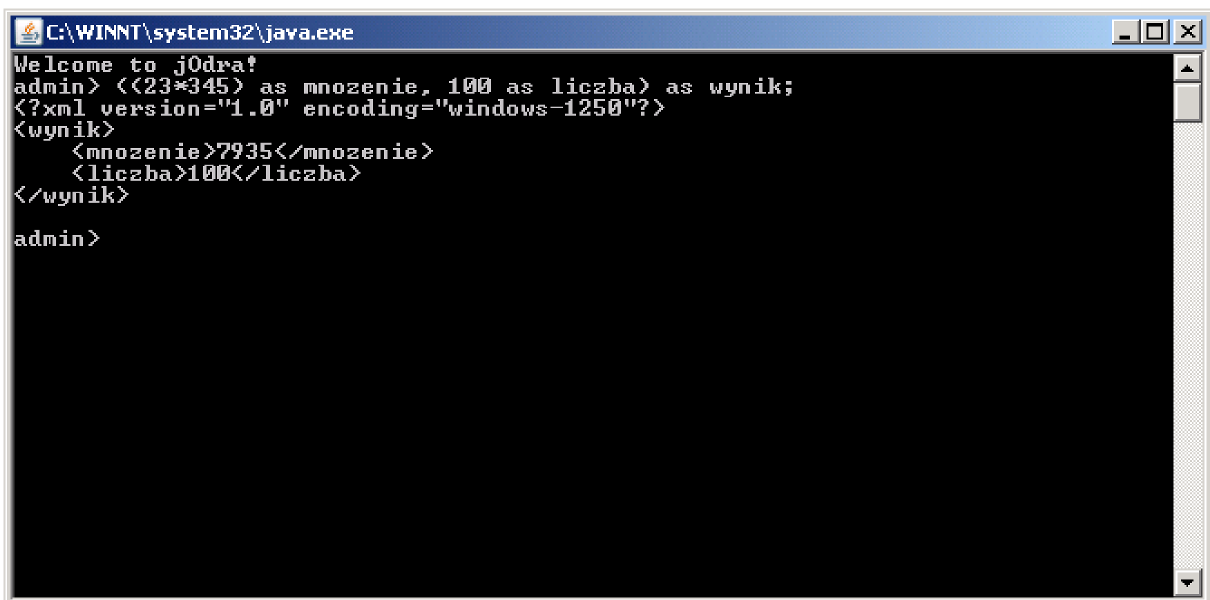
Następnie istnieje możliwość dokonania połączenia z bazą danych poprzez wydanie polecenia wewnątrz powłoki klienta connect *użytkownik*/*hasło@serwer*:*por*t gdzie parametry serwer i port określają nazwę serwera oraz port z którym zostanie wykonanie połączenie jako

podany użytkownik z podanym hasłem.

Do innych najważniejszych poleceń możliwych do wywołania wewnątrz powłoki klienta:

- disconnect - rozłącza połączenie z serwerem
- ls - pokazuje obiekty w bieżącym module
- cm - zmienia bieżący moduł, w przypadku podania zamiast nazwy symboli .. zostaje zmieniony bieżący moduł na moduł będący w hierarchii wyżej
- pwm - wypisuje bieżącą ścieżkę dla bieżącego modułu
- add module - dodaje nowy moduł
- compile - dokonuje kompilacji modułu
- batch - wykonuje kod zapytań zawartych w pliku
- config - dokonanie wewnętrznej konfiguracji serwera
- help - pokazuje pomoc

Oprócz wykonywania poleceń powłoki klienta istnieje możliwość wykonywania zapytań w procesie klienta poprzez wpisanie zapytania z poziomu klienta.



Rysunek 33: Wynik wykonania prostego zapytania

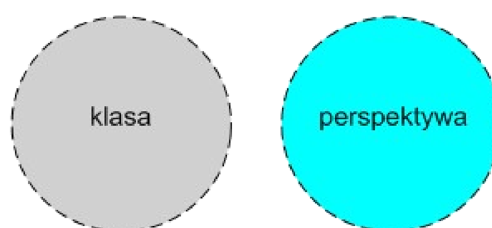
6 Dalsze kierunki rozwoju perspektyw

Dalszy rozwój perspektyw bazujących na podejściu stosowym wykorzystującym implementację projektu Odra J2 może wykazywać przyjmowanie następujących cech:

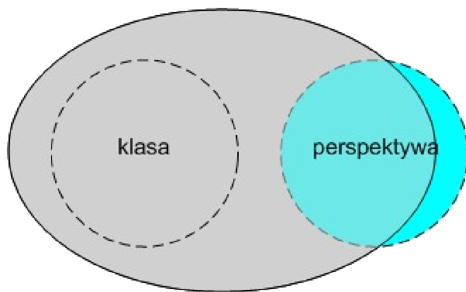
1. przejmowanie cech klasy przez perspektywę
2. asymilacja perspektyw przez klasy
3. zmiana/przeniesienie nacisku w deklaracji z perspektywy na wirtualny obiekt
4. przyjmowanie cech z programowania aspektowego

6.1 Rozwój klas i perspektyw

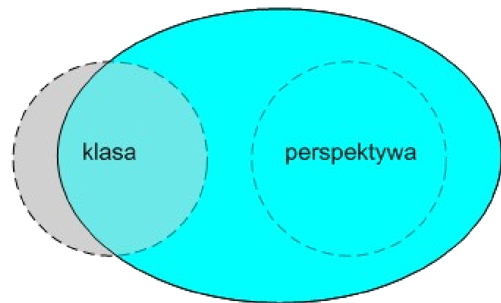
Dwa pierwsze kierunki rozwoju w przyszłości staną są do siebie wzajemnie przeciwstawne. Jeśli na początku swojej ewolucji klasa zacznie przejmować cechy perspektywy a perspektywa klasy to po dłuższym etapie takiej ewolucji powstaną dwa nowe twory niewiele różniące się od siebie. Z powodu niewielkiej różnicy ta różnorodność pojęciowa będzie powodowała konflikt - niepotrzebną redundancję co w efekcie powinno zostać zakończone wybraniem jednego a odrzuceniem drugiego pojęcia. W przypadku kiedy tylko jedno z powyższych zacznie przyjmować cechy drugiego natomiast drugie nie będzie się rozwijało w kierunku pierwszego to pierwsze pojęcie wchłonie drugie. Możliwa jest też droga mniej burzliwego rozwoju, a mianowicie gdy poszerzanie obszaru pojęciowego klas, perspektyw nie następuje w sposób wystarczająco szeroki do zaistnienia powyższych sytuacji.



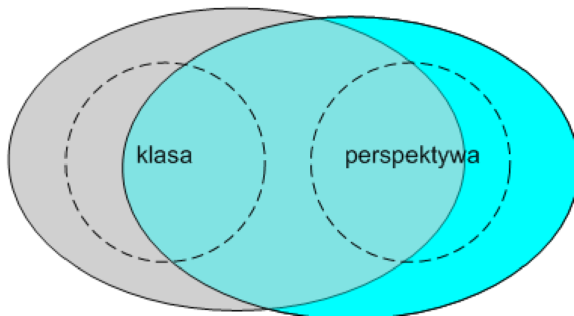
Rysunek 34: Początkowy zasięg cech pojęć



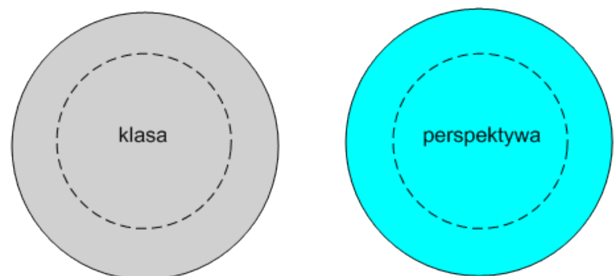
Rysunek 36: Rozrost cech klas - wchłonięcie perspektywy przez klasę



Rysunek 35: Rozrost cech perspektywy - wchłonięcie klasy przez perspektywę



Rysunek 37:Rozrost cech klasy i perspektywy - konflikt



Rysunek 38: Rozrost cech klasy i perspektywy - niezagrażający sobie nawzajem

Możliwy rozwój klas w kierunku perspektyw to możliwość posiadania wirtualnych zmiennych. Wirtualne zmienne mogły by być tym samym co dla innych języków np. C# jest własność(ang. property). Takie wirtualne zmienne działają w podobny sposób jak perspektywy czyli dla odpowiednich operacji na nich są wywoływane specjalne procedury przez interpreter w sposób przezroczysty z punktu widzenia kodu który się do nich odwołuje.

Rozwój perspektyw poprzez nabieranie cech klas już następuje. W pierwotnym założeniu perspektyw perspektywy mogły zawierać zmienne jak i procedury lecz możliwość ich wykorzystania była tylko i wyłącznie poprzez wykorzystanie schematu zarządczego perspektywy. Natomiast w implementacji posunięto się o krok dalej, można je wywoływać w wnętrzach procedur perspektywy z pominięciem schematu zarządczego perspektywy, ponadto procedury wewnętrzne mają dostęp do ziarna co powoduje analogię do metod obiektów gdzie ziarno jest wskaźnikiem do obiektu zapisywanym jako this. Procedury wewnętrzne dodatkowo także zachowującą się jak by miały cechę z modelu M3 czyli występowała hermetyzacja z widoczność ustawioną na prywatną dla procedur wewnętrznych.W klasach w języku SBQL nie występuje konstrukcja taka jak konstruktor dla perspektyw. Niemniej

jednak patrząc na inne języki obiektowe takie jak Java, C++ itd. w klasach mamy możliwość deklaracji konstruktora dla obiektu danej klasy. Pokazuje to że cecha jaką jest konstruktor przeszła nie tyle od klas istniejących w języku SBQL co od klas z innych obiektowych języków do perspektywy. Działanie samego konstruktora w perspektywach w stosunku do innych języków różni się, niemniej jednak pokazuje jak cechy klas nie tylko SBQL-a przechodzą do perspektyw. Najważniejszą cechą jaką mogą przejąć perspektywy z klas jest możliwość dziedziczenia lub wielodziedziczenia co w efekcie znacząco zbliżyło by pojęcie perspektyw do pojęcia klas tak że perspektywa stała by się realnym konkurentem dla zastosowań klas.

6.2 Rozwój składni perspektywy - przeniesienie nacisku na wirtualny obiekt

W składni deklaracji perspektyw pierwszoplanowo jest deklarowana nazwa schematu zarządczego perspektywy, natomiast nazwa wirtualnego obiektu jest deklarowana wewnątrz ciała deklaracji perspektywy. Taka kolejność wskazuje ważność deklaracji, schematu zarządczego perspektywy jest ważniejsza od wirtualnego obiektu. Przy podejściu odwrotnym ważniejszy staje się wirtualny obiekt. Takie podejście kładzie większy nacisk na użytkowanie wirtualnego obiektu niż na zadania związane z zarządzaniem perspektywą. Przy próbie pisania implementacji która próbuje domyślnie przyjmować którąś z nazw: wirtualnego obiektu lub nazwę schematu zarządczego perspektywy - czytelniejsze jest generowanie nazwy dla drugorzędnej definicji.

```
virtual BogatyPracownik{
        from:record{p:Pracownik;} {
                return Pracownik where zarobek>2000 as p;
        }
        on_retrieve:record{imie:string;nazwisko:string;} {
                return (deref(p.imie) as imie, deref(p.nazwisko) as nazwisko );
        }
        on_update(value:record{imie:string;nazwisko:string;}) {
                ......
```

```
	}
}
```

Kolejną z możliwości dalszego ulepszenia perspektywy jest ustalenie typów wszystkich procedur wykonywanych zamiast operacji na perspektywie. Ten typ zostaje zadeklarowany w jednym miejscu i jest uważany za typ wszystkich powyższych procedur które zwracają dane oraz typ dla procedur on_update i on_new typ parametru którego nazwa zostaje automatycznie nadana jako value. To podejście ogranicza możliwości samej perspektywy niemniej jednak wymusza konsekwentne trzymanie się typu pobieranego z perspektywy jak i we wszystkich procedurach przesyłających dane do perspektywy dzięki czemu jest możliwe spełnienie zasady przezroczystości - nie da się odróżnić po typach czy mamy dostęp do zmiennej czy perspektywy. Jednakże ten problem z ograniczeniem można zniwelować przez przyjęcie założenia: tak gdzie nie jest deklarowany typ jest brany typ ustalony dla wszystkich procedur a tak gdzie jest podany typ tam wspólny typ jest pomijany i wstawiany zadeklarowany.

Rysunek 39: Przesyłanie i odbieranie danych dla zwykłego obiektu oraz perspektywy, perspektywa może zwracać inny typ niż przesyłany do niej

```
virtual BogatyPracownik:record{imie:string;nazwisko:string;} {
	from:record{p:Pracownik;} {
		return Pracownik where zarobek>2000 as p;
	}
	on_retrieve{
		......
	}
	on_update{
```

```
            ......
        }
}
```

Kolejnym miejscem gdzie można dokonać zmian jest procedura wirtualnego obiektu. Jest ona deklarowana w powyższy sposób:

```
        virtual objects  BogatyPracownik:record{p:Pracownik;} {
                return Pracownik where zarobek>2000 as p;
        }
```

lub wg. propozycji

```
        from:record{p:Pracownik;} {
                return Pracownik where zarobek>2000 as p;
        }
```

Obie powyższe deklaracje charakteryzują się tym że zwracają `record{p:Pracownik;}` gdzie ziarnem jest p. Dla każdego wirtualnego obiektu musi być zdefiniowane w ten sposób ziarno, jeśli nie będzie podane w powyższy sposób tylko jako zwrócony typ to nie będzie możliwe odwoływanie się do niego w innych częściach perspektywy. Dla każdej perspektywy oprócz wyjątku powyższego będą więc co najmniej trzy nazwy:

- schematu zarządczego perspektywy
- nazwy wirtualnego obiektu
- nazwę ziarna dla wirtualnego obiektu

Nazwę o nazwie schematu zarządczego perspektywy było mówione wcześniej, według przyjętej konwencji nazwa definicji jest złożeniem nazwy wirtualnego obiektu oraz sufiksu Def. Dla nazw ziaren nie ma konwencji zapisu, są one dowolne przez co patrząc w kodzie SBQL-a w pierwszym momencie nie można dojść po samej nazwie czym jest ziarno. Proponowana jest konwencja dla pojedynczych ziaren aby nazwa ich była taka jak nazwa wirtualnego obiektu ale poprzedzona znakiem @.

Przykład:

wirtualny obiekt	ziarno (wg konwecji)
BogatyPracownik	@BogatyPracownik
CzerwonySamochod	@CzerwonySamochod

Można do kwestii nazywania ziarna podchodzić w inny sposób - nazwa ziarna jest stała np: seed. Niestety takie podejście powoduje iż w podperspektywach nie mielibyśmy dostępu do ziarna perspektyw które są wyżej ponieważ nazwa seed była by wiązana zawsze do podperspektywy - co implikuje odrzucenie tej drogi. Jedynym rozwiązaniem tego problemu dla tego podejścia to możliwość dodania tej opcji jako uzupełnienia już innego sposobu dostawania się do ziarna. W takim wypadku tam gdzie nie można dostać się przez nazwę seed można by uzyskiwać dostęp drogą alternatywną.

Powyższa konwencja nie obejmuje przypadków w których ziaren jest więcej niż jedno:

```
virtual objects  BogatyPracownik:record{i:string; n:string;} {
                return (Pracownik where zarobek>2000).( imie as i, nazwisko as n);
}
```

Przy stosowaniu tej konwencji można doprowadzić do skrócenia zapisu procedury wirtualnego obiektu przy założeniach że występuje tylko jedno ziarno dla orginalnego zapisu:

```
virtual objects  BogatyPracownik:record{@BogatyPracownik:Pracownik;} {
        return Pracownik where zarobek>2000 as @BogatyPracownik;
}
```

do postaci

```
virtual objects  BogatyPracownik:Pracownik {
                return Pracownik where zarobek>2000;
}
```

Powyższą konwersje można dokonać za pomocą dwóch dróg:

- przez przetwarzanie składni
- przez zmianę działania iterpretera

Istota pierwszej drogi polega na:

- zmianie typu procedury wirtualnego obiektu z delkarowanego typu TYP na typ **record**{ @nazwaZiarna:TYP;}
- zmiany wszędzie we wnętrzu procedury z **return** wyrażenie na **return** wyrażenie **as** @nazwaZiarna

Zmiana działania iterpretera natomiast polega nie na zmienianiu procedury wirtualego obiektu, ale na zmianie zachowania się po zwróceniu wartości przez procedurę, wartość uzyskana zostaje nazwana @nazwaZiarno przez co uzyskujemy efekt analogiczny do zmiany składni z tym że zostaje to przeniesione do czasu wykonania.

Łącząc wszystkie opisane możliwe rozszerzenia wraz z możliwością generowania on_retrieve możemy uzyskać poniższy przykład

```
virtual BogatyPracownik:record{imie:string; nazwisko:string;zarobek:real;}{
        from:Pracownik {
                return Pracownik where zarobek>2000;
        }

        on_retrieve;  // wygeneruje
//      on_retrieve { return (deref(@BogatyPracownik.imie) as imie,
//                      deref(@BogatyPracownik.nazwisko) as nazwisko,
//                      deref(@BogatyPracownik.zarobek) as zarobek );
//      }
}
```

6.2.1 Przykłady różnych wariantów przyszłych perspektyw

W efekcie ilość terminali dla poszczególnych pokazywanych poprzednio rozwiązań prezentuje się następująco:

A) 16 - SQL[17]

17 perspektywy dla języka SQL zostały wstawione tylko i wyłącznie dla porównania z konstrukcjami obiektowej bazy ponieważ obecnie na świecie takie perspektywy są wykorzystywane. Należy pamiętać że perspektywy w SQL-u zawsze będą krótsze ponieważ w SBQL-u dochodzi kontrola typów co zwiększa

```
CREATE VIEW BogatyPracownik AS SELECT imie, nazwisko, zarobek FROM
Pracownik WHERE zarobek > 2000
```

B) 74 - obecna implementacja

```
view BogatyPracownikDef{
	virtual objects BogatyPracownik:record{p:Pracownik;}{
		return Pracownik where zarobek>2000 as p;
	}
	on_retrieve:record{imie:string; nazwisko:string;zarobek:real;}{
		return (deref(p.imie) as imie, deref(p.nazwisko) as nazwisko,
		deref(p.zarobek as zarobek );
	}
}
```

C) 28 - obecna implementacja z generowaniem on_retrieve

```
view BogatyPracownikDef{
	virtual objects BogatyPracownik:record{p:Pracownik;}{
		return Pracownik where zarobek>2000 as p;
	}
	on_retrieve;
}
```

D) 67 - obecna implementacja + zmiana w procedurze wirtualnego obiektu

```
view BogatyPracownikDef{
	virtual objects BogatyPracownik:Pracownik{
		return Pracownik where zarobek>2000;
	}
	on_retrieve:record{imie:string; nazwisko:string;zarobek:real;}{
```

długość kodu

```
                return (deref(@BogatyPracownik.imie) as imie,
                        deref(@BogatyPracownik.nazwisko) as nazwisko,
                        deref(@BogatyPracownik.zarobek) as zarobek );
        }
}
```

E) 20 - obecna implementacja z generowaniem on_retrieve + zmiana w procedurze wirtualnego obiektu

```
view BogatyPracownikDef{
        virtual objects BogatyPracownik:Pracownik{
                return Pracownik where zarobek>2000;
        }
        on_retrieve;
}
```

F) 75 - obecna implementacja + wspólny typ dla procedur on_xxxx

```
view BogatyPracownikDef:record{imie:string; nazwisko:string;zarobek:real;}{
        virtual objects BogatyPracownik:record{p:Pracownik;}{
                return Pracownik where zarobek>2000 as p;
        }
        on_retrieve{
                return (deref(p.imie) as imie,
                        deref(p.nazwisko) as nazwisko,
                        deref(p.zarobek) as zarobek );
        }
}
```

G) 44 - obecna implementacja z generowaniem on_retrieve + wspólny typ dla procedur on_xxxx

```
view BogatyPracownikDef:record{imie:string; nazwisko:string;zarobek:real;}{
        virtual objects BogatyPracownik:record{p:Pracownik;}{
                return Pracownik where zarobek>2000 as p;
        }
        on_retrieve;
}
```

H) 67 - obecna implementacja + zmiana w procedurze wirtualnego obiektu, wspólny typ dla procedur on_xxxx

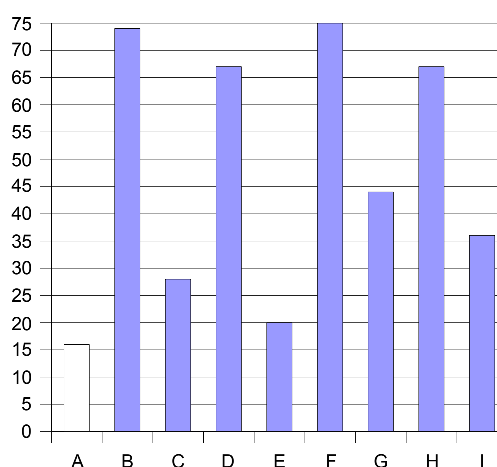
```
view BogatyPracownikDef:record{imie:string; nazwisko:string;zarobek:real;}{
        virtual objects BogatyPracownik:Pracownik{
                return Pracownik where zarobek>2000;
        }
        on_retrieve{
                return (deref(@BogatyPracownik.imie) as imie,
                        deref(@BogatyPracownik.nazwisko) as nazwisko,
                        deref(@BogatyPracownik.zarobek) as zarobek );
        }
}
```

I) 36 - obecna implementacja z generowaniem on_retrieve + zmiana w procedurze wirtualnego obiektu, wspólny typ dla procedur on_xxxx

```
view BogatyPracownikDef:record{imie:string; nazwisko:string;zarobek:real;}{
        virtual objects BogatyPracownik:Pracownik{
                return Pracownik where zarobek>2000;
        }
        on_retrieve;
}
```

Zestawienie wyników:

wersja	ilość	generowanie on_retrieve	zmiana w proc. wirt. obiektu	wspólny typ dla proc.
A	16	-	-	-
B	74	nie	nie	nie
C	28	tak	nie	nie
D	67	nie	tak	nie
E	20	tak	tak	nie
F	75	nie	nie	tak
G	44	tak	nie	tak
H	67	nie	tak	tak
I	36	tak	tak	tak



Rysunek 40: Ilość terminali dla różnych wersji perspektyw

W powyższym zestawieniu najlepszy wynik uzyskuje perspektywa w SQL, jednakże należy pamiętać że nie ma w niej kontroli typów przez co perspektywy SBQL-a z kontrolą typu są zawsze dłuższe. W rozwiązaniach z generacją on_retrieve udało się zbliżyć do perspektyw SQL-owych. W szczególności rozwiązania C i E uzyskały długość którą można przyjąć za porównywalna z SQL choć w języku SBQL występuje kontrola typów. Generowanie on_retrieve może występować tylko dla przypadków trywialnych, niemniej jednak dla innych przypadków perspektywa SQL-a jest poza możliwościami, a perspektywa SBQL-a mimo że w niej nie można wygenerować automatycznie on_retrieve to daje możliwość implementacji tego przypadku. Zmiana nazwy ziarna dla procedury wirtualnego obiektu nie przynosi dużych

zmian w długości perspektywy natomiast wpływa na trudno mierzalny czynnik jakim jest czytelność zapisu. Natomiast wspólny typ dla procedur on_xxxx przynosi skrócenie zapisu perspektywy dopiero gdy tych procedur występuje więcej niż jedna.

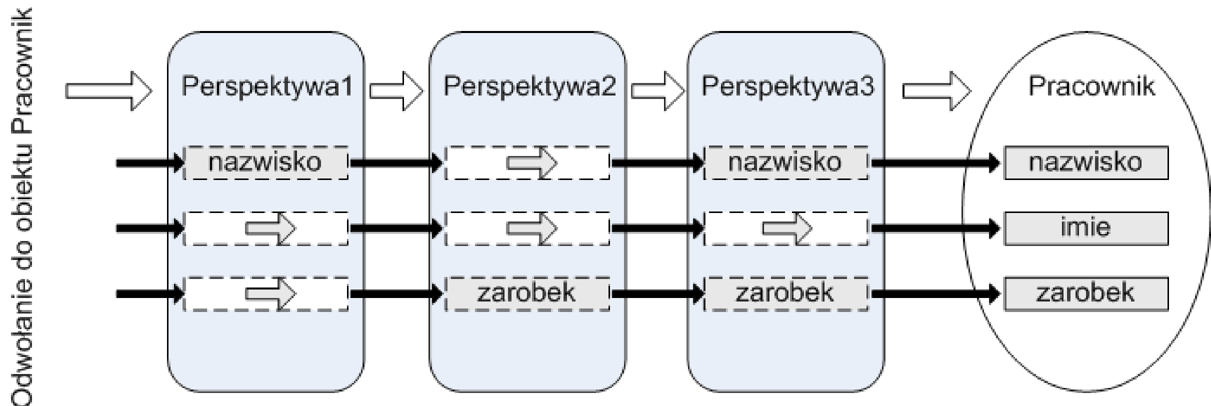
6.3 Przyjmowanie cech z programowania aspektowego

Cechy perspektywy mogą zostać rozszerzone o cechy z programowania aspektowego. Możliwymi drogami rozwoju w kierunku aspektowym są:

- łańcuch perspektyw pośredniczących
- model aspektowości analogiczny do AspectJ

6.3.1 Perspektywy pośredniczące

Rozwój w tym kierunku został szczegółowo opisany w [Adamus05]. Perspektywy są pośrednikami w dostępie do danych, podczas próby odwołania do danych interpreter musi przejść przez cały łańcuch perspektyw pośredniczących. Perspektywy pośredniczące mają za zadanie każda odwzorowanie jednego aspektu np.: transakcji, bezpieczeństwa, logowania etc.



Rysunek 41: Przykład łańcucha perspektyw pośredniczących do obiektu Pracownik wg [Adamus05].

6.3.2 Cechy analogiczne do języka AspectJ

Inną drogą dalszego rozwoju jest rozszerzenie możliwości SBQL-a a wraz z nim perspektyw o elementy aspektowości tak jak język Java został poszerzony do języka AspectJ. W języku AspectJ[AspectJ] istnieje możliwość definiowania punktów przecięć i złączeń dla wybranych aspektów. W sposób analogiczny istnieje możliwość wprowadzenia do perspektyw takiego mechanizmu.

Podsumowanie

W niniejszej pracy magisterskiej zostały opisane perspektywy mające możliwość pełnej aktualizacji - od sposobu rozwiązania problemu aktualizacji dla perspektyw w bazie relacyjno-obiektowej firmy Oracle poprzez zastosowanie specjalnego trigger-a INSTEAD OF aż do implementacji perspektywy zaimplementowanej w ramach pracy magisterskiej „Aktualizowalne perspektywy w obiektowych bazach danych”. W ramach pracy wykonano działającą implementacje perspektywy dającą możliwość aktualizacji wraz z dodatkowymi elementami nadającymi nowe cechy pojęciu perspektywy. W efekcie perspektywa stała się narzędziem mogącym rozwiązywać problemy w sposób bardziej przyjazny dla użytkownika. Dalsze możliwości zmian w celu poprawienia sposobu korzystania z perspektywy przez programistę zostały zawarte w ostatnim rozdziale. Dalszy rozwój perspektywy w którym kolwiek z określonych kierunków opisanych w ostatnim rozdziale może przynieść znaczącą poprawę odbioru przez użytkownika końcowego, co w sposób efektywniejszy może pomóc wykorzystać pełną moc narzędzia jakim są perspektywy. Ze względów objętościowych niniejszej pracy w ostatnim rozdziale uwaga została skupiona tylko na krótkim wstępie dla każdej drogi rozwoju i dalsze rozwinięcie może zostać kontynuowane w dalszych pracach badawczych bazujących na pracy magisterskiej „Aktualizowalne perspektywy w obiektowych bazach danych”.

Bibliografia

Adamus R. (2005): Programming in Aspect-Oriented Databases, Praca doktorska.

Date C. J. (2000): SQL Omówienie standardu języka Wydawnictwo Naukowo-Techniczne, Warszawa.

Eckel B. (2003) Thinking in Java 3-cia edycja, Helion.

Gamma E.,Helm R., Johnson R., Vlissides J. (2005): Wzorce projektowe. Wydawnictwo Naukowo-Techniczne, Warszawa.

Kozankiewicz H. (2004): Updatable Object Views. Praca doktorska.

Liberty J. (2006): C# Programowanie, Helion.

Mischel J. Duntemann J. (1997): Borland C++ Builder, Wydawnictwo ZNI Mikom, Warszawa.

Stencel K. (2006): Półmocna kontrola typów w językach programowania baz danych.

Stroustrup B. (2002): Język C++ WNT wydanie szóste, Warszawa.

Subieta K. (2004): Teoria i konstrukcja obiektowych języków zapytań. Wydawnictwo PJWSTK, Warszawa.

Źródła internetowe:

- [Ant] http://ant.apache.org/
- [AspectJ] http://www.eclipse.org/aspectj/
- [CUP] http://www.cs.princeton.edu/~appel/modern/java/CUP/
- [DocOdra] Odra J2 Dokumentacja techniczna: http://iolab.pjwstk.edu.pl:8081/forum/image.aspx?a=77
- [JFlex] http://www.jflex.de/
- [JulietCode] Adamus R. svn://212.191.89.27:3691/documentation
- [LINQ] http://msdn2.microsoft.com/en-us/netframework/aa904594.aspx

- [OdraManual] http://iolab.pjwstk.edu.pl:8081/forum/image.aspx?a=239
- [SBQL] Subieta K. SBQL: http://www.sbql.pl/