Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Adam Dąbrowski
Nr albumu: 197852

Interfejsy i schematy zewnętrzne w LoXiM

Praca magisterska
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem
dr hab. Krzysztof Stencel
Instytut Informatyki

Styczeń 2009

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Data Podpis kierującego pracą

Oświadczenie autora (autorów) pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data Podpis autora (autorów) pracy

Streszczenie

W pracy przedstawiam implementację schematów zewnętrznych i niezbędnych do ich funkcjonowania interfejsów w obiektowym systemie zarządzania bazami danych LoXiM. Zrealizowane przeze mnie schematy zewnętrzne pozwalają na odwzorowanie przestrzeni nazw i obiektów bazy danych na byty dostępne programistom aplikacyjnym korzystającym z LoXiM. Dzięki temu możliwe jest ograniczanie dostępu poszczególnym aplikacjom oraz zapewnienie, dzięki wykorzystaniu interfejsów do klas i perspektyw systemu, dowolnego odwzorowania schematów zewnętrznych na obiekty bazy danych.

Słowa kluczowe

SBQL, podejście stosowe, bazy danych, architektura trójwarstwowa, interfejsy, schematy zewnętrzne, LoXiM

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

Klasyfikacja tematyczna

H. Information System
H.2 Database Management
H.2.3 Languages
H.2.4 Systems

Tytuł pracy w języku angielskim

Interfaces and outer schemas in LoXiM

Spis treści

Rozdział 1

Wprowadzenie

1.1. Podejście stosowe

Podejście stosowe zostało przedstawione w monografii Kazimierza Subiety [Sub04]. Na potrzeby mojej pracy krótko opiszę, na czym ono polega. Podejście stosowe pozwala spojrzeć na język zapytań z punktu widzenia języka programowania. Nie ma w tym podejściu rozróżnienia na zapytania i byty programistyczne, takie jak procedury, funkcje, typy czy klasy. W ten sposób rozwiązywany jest powszechny w systemach zarządzających trwałymi danymi problem łączenia dwóch technologii: odpowiedzialnego za działanie systemu języka programowania i języka zapytań, który dotyczy składu danych. Do realizacji widoczności bytów w podejściu stosowym służy stos środowisk, na którym wiązane są nazwy wystepujące w zapytaniu. Podczas nawigacji po danych widoczność ta jest regulowana poprzez mechanizm przesłaniania sekcji stosu.

1.2. Język SBQL

Stack-Based Query Language to język zapytań, będący jednocześnie językiem programowania, stworzony zgodnie z podejściem stosowym. W SQBL nie ma koncepcyjnej różnicy między wyrażeniami, takimi jak *2*4* i zapytaniami w rodzaju *(Prac where Nazwisko = Kowalski).Zarobki*. Realizacja SBQL wymaga składu obiektów, stosu wyników i stosu środowisk. Dokładny opis języka SBQL można znaleźć na stronie [SQBL], a jego implementacja w LoXiM jest omówiona w drugim rozdziale pracy Dariusza Gryglasa [Gryg08].

1.3. LoXiM

Powstający pod opieką dra hab. Krzysztofa Stencla i realizowany przez studentów Uniwersytetu Warszawskiego obiektowy, półstrukturalny system zarządzania bazami danych. Oparty na podejściu stosowym, wykorzystuje własny wariant języka SBQL, w założeniu będzie rozszerzony również o SQL i XQuery. Choć LoXiM jest systemem eksperymentalnym, na którym są tworzone i testowane nowe rozwiązania w technologii baz danych, planowane są także stabilne wydania. Więcej informacji można znaleźć na stronie projektu [LoXiM].

1.3.1. Elementy systemu LoXiM związane z tematem pracy

- **klasa w LoXiM** - specjalny obiekt składu bazy danych i abstrakcja w pewnych aspektach podobna do klasy w obiektowych językach programowania. Relacja między klasami

i obiektami w LoXiM jest dynamiczna - obiekty mogą stawać się instancjami klasy i przestawać nimi być. Z tymi operacjami nie są wiązane dodatkowe czynności, innymi słowy nie ma konstruktorów i destruktorów - bycie instancją klasy jest rozumiane jako cecha obiektu. Klasa określa jego właściwości, definiując pola i metody, które można na nim wywoływać. W LoXiM zaimplementowany został graf klas, przy pomocy którego realizowane są mechanizmy dziedziczenia, takie jak zastępowalność na poziomie kolekcji. Obiekt może być instancją wielu niezależnych od siebie w hierarchii klas. Istnieje możliwość wywołania metody z konkretnej nadklasy, co rozwiązuje część problemów stwarzanych przez wielodziedziczenie. Autorem implementacji klas w LoXiM jest Sebastian Kaczanowski [Kacz07].

- **perspektywa w LoXiM** - jest to aktualizowalna, obiektowa perspektywa, definiująca zapytanie o tzw. wirtualne obiekty oraz określająca sposób ich odczytywania, tworzenia, usuwania oraz nawigacji po ich podobiektach. Perspektywa w LoXiM jest w pełni wirtualna (nie podlega materializacji). Problemy z aktualizacją wirtualnych obiektów zostały rozwiązane poprzez przeciążenia operacji imperatywnych, jakie są na tych obiektach wykonywane - takie przeciążenia definiuje twórca perspektywy przy pomocy specjalnej składni. Złożone wirtualne obiekty tworzone są przy użyciu podperspektyw, możliwe jest ich zagnieżdżanie. Definicja perspektywy jest specjalnie oznaczonym obiektem składu bazy danych. Autorem implementacji perspektyw w LoXiM jest Dariusz Gryglas [Gryg08].

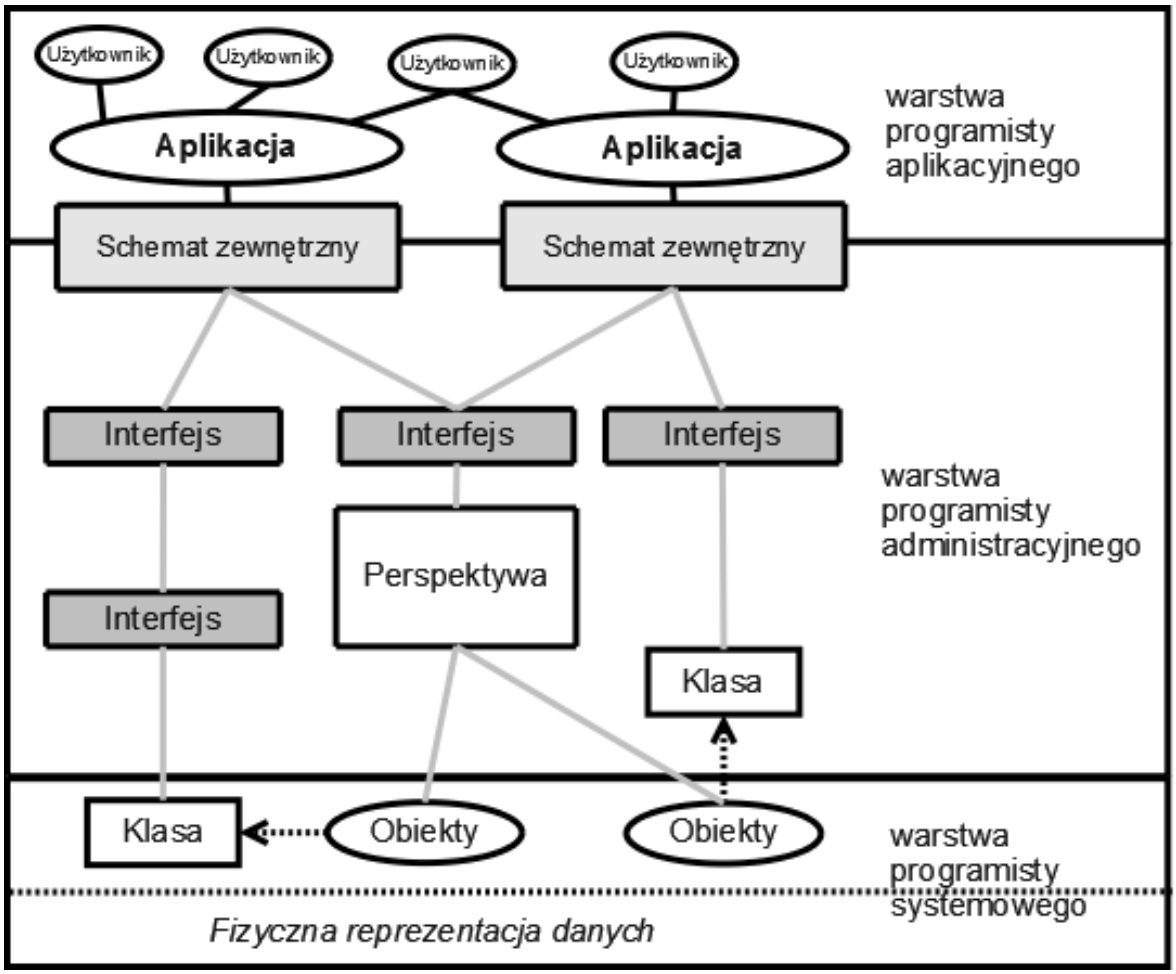
- **nested** - funkcja aparatu wykonawczego systemu LoXiM odpowiadająca za obliczenie nowej sekcji, jaka ma być włożona na stos środowisk, kiedy przetwarzany jest jakikolwiek operator niealgebraiczny. Ta nowa sekcja zawiera wnętrze obiektu, które musi być widoczne np. w kontekście prawego zapytania operatora nawigacji (kropki), aby nazwy podobiektów poprawnie wiązały się na stosie. Jeśli funkcja *nested* wywołana jest dla obiektu atomowego, jej wynik jest pusty. Dla obiektu złożonego *nested* zwraca zbiór wiązań (nazwanych *binderami*) do wszystkich jego bezpośrednich podobiektów.

- **deref** - unarna operacja odczytu obiektów w LoXiM, przyjmująca jako argument zapytanie i zastępująca wszystkie jego referencje do podobiektów ich rzeczywistymi wartościami. Te wartości mogą zawierać kolejne referencje, których odczyt wymaga kolejnego wywołania funkcji *deref*.

1.4. Model architektury trójwarstwowej

Model architektury trójwarstwowej dla obiektowej bazy danych został zaprezentowany w pracy Piotra Habeli, Krzysztofa Stencela i Kazimierza Subiety [HSS06]. W tym modelu zostały zostały wyróżnione trzy role związane z bazą danych - programisty systemowego, administracyjnego i aplikacyjnego. Odpowiadają im poszczególne warstwy BD, które reprezentują zainteresowanie i zakres obowiązków właściwe każdej z tych ról. Z punktu widzenia mojej pracy najistotniejsza jest rola programisty administracyjnego.

Warstwy i role w architekturze trójwarstowej przedstawia poniższy schemat, który stworzyłem na podstawie diagramu przedstawionego w pracy [HSS06].



Rysunek 1.1: Schemat modelu architektury trójwarstowej

W przedstawionym modelu programista administracyjny definiuje klasy, perspektywy i interfejsy, a także powiązania między nimi. Następnie tworzy schematy zewnętrzne, nadaje do nich prawa i przekazuje je programiście aplikacyjnemu, który wykorzystuje je, pisząc kod aplikacji. W rezultacie wszyscy użytkownicy aplikacji korzystają z bazy danych za pośrednictwem schematów, które ograniczają i przekształcają przestrzeń obiektów bazy danych ściśle na potrzeby konkretnych zastosowań. Takie rozwiązanie zawiera w sobie pełny mechanizm kontroli dostępu. W modelu architektury trójwarstwowej te same obiekty warstwy programisty systemowego mogą być widoczne przez kilka różnych interfejsów wykorzystywanych w schematach dla poszczególnych aplikacji. Oprócz ograniczenia widoczności podobiektów możliwe jest też wyspecyfikowanie na poziomie interfejsu, jakie operacje są na prezentowanych danych dozwolone. Wynika z tego między innymi to, że jedna aplikacja może modyfikować pola obiektu, których druga nawet nie widzi. W pracy dotyczącej modelu architektury trójwarstwowej [HSS06] zaproponowana została semantyka, w której dostęp do każdego pola obiektu definiowany jest oddzielnie. W swojej pracy zdecydowałem się na prostsze rozwiązanie, w którym specyfikacja praw odbywa się na poziomie całego interfejsu.

Rozdział 2

Interfejsy

2.1. Założenia towarzyszące implementacji

Implementacja interfejsów w LoXiM stanowi dużą część mojej pracy. Zdecydowałem, że obiekty interfejsów będą w pełni wirtualne - będą istnieć tylko w wyobraźni użytkownika, podczas gdy rzeczywiste operacje w bazie danych będą wykonywane na obiektach związanej implementacji. Przemawiały za tym względy estetyczne i problemy w zachowaniu spójności w rozwiązaniu alternatywnym. Operacje na obiektach poprzez interfejs powinny być przy tym całkowicie przezroczyste w kilku aspektach: składni zapytań, postaci zwracanych rezultatów oraz ograniczeniu dostępu. To założenie również udało mi się zrealizować. Składnia zapytań i postać rezultatów są identyczne jak w przypadku bezpośrednich manipulacji, a pola niedostępne są nieodróżnialne od nieistniejących. Kolejnym wyzwaniem była wydajność zapytań wykonywanych na obiektach interfejsów - w tym celu zaimplementowałem szybkie struktury pomocnicze, które znacznie zmniejszają dodatkowy narzut czasowy. Zadbałem również o utrzymanie spójności, zarówno w odniesieniu do hierarchii interfejsów, jak i ich wiązań z implementacjami.

2.2. Konstrukcja interfejsu

Interfejs może zostać zdefiniowany przez uprawnionego użytkownika przy pomocy następującej składni:

```
create interface [nazwa interfesju] extends [nazwa interfejsu rodzica], ...
{
  instance [nazwa obiektów]:
  {
    [nazwa pola];
    ...
  }
  procedure [nazwa procedury] ( [nazwa parametru], ... )
  ...
}
```

Pola, metody, nazwa obiektów i lista rodziców są opcjonalne. Interfejs stworzony bez elementu składni **instance** nie będzie reprezentował żadnych obiektów. Brak tego elementu składni oznacza, że taki interfejs może co prawda służyć jako szczebel w hierarchii dziedziczenia, ale nie można używać go do operowania na danych. Nazwa interfejsu i nazwa obiektów muszą być

unikalne w całej bazie danych, nazwy pól i metod o określonej liczbie parametrów w obrębie tej definicji nie mogą się powtarzać. Istniejący już interfejs można zmodyfikować przy pomocy składni **update**:

```
update interface (...)
```

O ile **create** nie powiedzie się (i zasygnalizuje błąd), jeśli interfejs o podanej nazwie już istnieje, **update** zadziała dokładnie odwrotnie: uaktualni wskazaną definicję interfejsu albo zwróci błąd w przypadku jej braku. Połączeniem pozytywnych ścieżek dla obu tych opcji jest składnia **create or update**.

Dla potrzeb dalszej analizy podaję poniżej prosty przykład definicji dwóch klas i odpowiadających im interfejsów:

```
create class COsoba
{
  instance OOsoba:
  {
    imię;
    nazwisko;
    dataUrodzenia
  }
}

create class CPrac extends COsoba
{
  instance OPrac:
  {
    dział;
    stanowisko;
    zarobki
  }
  procedure dajPodwyżkę(kwota) {zarobki := zarobki + kwota; }
  procedure zmieńDział(nowyDział) {dział := nowyDział; }
}

create interface IOsoba
{
  instance OIOsoba:
  {
    imie;
    nazwisko;
    dataUrodzenia
  }
}

create interface IPrac extends IOsoba
```

```
{
  instance OIPrac:
  {
    dział;
    stanowisko
  }
  zmieńDział(nowyDział)
}
```

2.2.1. Składowanie obiektów definicji

W wyniku polecenia **create interface** do składu danych zostanie dodany specjalnie oznaczony obiekt definicji interfejsu, który można potem odczytać, usunąć, bądź zmodyfikować - tak jak każdy inny obiekt. Składowanie realizowane jest podobnie jak w przypadku klas, interfejsom odpowiada jednak oddzielny plik, *sbinterfaces* i odmienna struktura pomocnicza. Jest nią zbiór słowników, otoczony klasą C++ InterfaceMaps. Jej struktury są inicjowane przy starcie serwera. Ich istnienie uzasadnione jest potrzebą zwiększenia wydajności operacji często wykonywanych na interfejsach, takich jak zapytania o uwzględniającą dziedziczenie listę pól i metod danego interfejsu. Struktury te uczestniczą w większości opisanych dalej operacji, przyspieszając je i ułatwiając ewentualne przyszłe modyfikacje.

2.2.2. Dziedziczenie

Dla interfejsu można zdefiniować rodziców przy pomocy elementu składni **extends**. Wszystkie nazwy wymienione w liście następującej po tym słowie kluczowym muszą odpowiadać obiektom już utworzonym. W powyższym przykładzie najpierw musimy zdefiniować IOsoba, a następnie IPrac, w przeciwnym wypadku próba utworzenia definicji IPrac zakończy się błędem. Wszystkie pola i metody rodzica będą odziedziczone przez potomka - w tym przypadku interfejs IPrac odziedziczy pola *imię*, *nazwisko* i *dataUrodzenia*.

Elementem mechanizm dziedziczenia klas jest zastępowalność na poziomie kolekcji. Oznacza to, że zapytanie o obiekty OOsoba klasy COsoba zwróci również obiekty OPrac jej nadklasy CPrac. Hierarchia i dziedziczenie w klasach odzwierciedlają nasz sposób myślenia o modelowanej rzeczywistości - każdy pracownik jest również osobą. Więcej o mechanizmach dziedziczenia w LoXiM można przeczytać w pracy Sebastiana Kaczanowskiego opisującej implementację klas [Kacz07]

Interfejsy nie realizują zastępowalności na poziomie kolekcji. Choć można wskazać sytuacje, w których rozwiązanie to byłoby koncepcyjnie korzystne, nie jest ono wskazane z kilku przyczyn. Jedną z nich jest redundancja takiego mechanizmu. W zaprezentowanym przykładzie zapytanie *OIOsoba* zwróci wszystkie, odpowiednio ograniczone przez interfejs, obiekty OOsoba i OPrac, wykorzystując mechanizm zastępowalności właściwy klasom. Jeśli rezultat zapytania *OIOsoba* miałby być rozszerzony o wynik polecenia *OIPrac*, obiekty OPrac występowałyby w nim podwójnie. Innym powodem jest swoboda i wygoda w wykorzystywaniu dziedziczenia interfejsów bez potrzeby kopiowania pól. Rozważmy rozszerzenie przykładu o perspektywę i interfejs do niej:

```
create view BogaciPracDef
```

```
{

  virtual_objects BogaciPrac()
  {
    return (OPrac where zarobki > 15000) as bp
  }

  procedure on_retrieve()
  {
    return deref(bp)
  }

}

create interface IBogaciPrac extends IOsoba
{
  instance OIBogaciPrac:
  {
    dział;
    stanowisko
  }
}
```

Po związaniu interfejsu IBogaciPrac z perspektywą BogaciPracDef zapytanie *OIBogaci-Prac* będzie zwracać obiekty określone w elemencie składni **virtual_objects** definicji perspektywy, czyli BogaciPrac. Hierarchia dziedziczenia interfejsów nie odzwierciedla dziedziczenia w domenie implementacji - a jednak użycie składni **extends** przy definiowaniu IBogaciPrac jest naturalne i wygodne. Więcej o perspektywach można przeczytać w pracy Dariusza Gryglasa opisującej ich implementacje w LoXiM [Gryg08].

Dziedziczenie dla interfejsów jest mechanizmem ułatwiającym projektowanie i zapobiegającym redundancji. Rodzic występuje tu jedynie w roli zbioru wyodrębnionych pól i metod. Hierarchia dziedziczenia interfejsów może odpowiadać strukturze klas i zawierać wiązania na róznych jej poziomach, ale nie musi.

2.2.3. Spójność hierarchii interfejsów

Implementacja interfejsów w LoXiM bierze pod uwagę problem spójności hierarchii. W trakcie pracy z bazą danych nie można dopuścić do powstania pętli w dziedziczeniu, ani do sytuacji, w której rodzic któregoś z interfejsów nie istniałby w składzie.

Za zachowanie spójności hierarchii odpowiadają struktury i metody klasy InterfaceMaps. Próba wykonania operacji rozspójniającej hierarchię zakończy się błędem. Weryfikacja dotyczy operacji dodawania, modyfikowania i usuwania interfejsów. W dwóch pierwszych przypadkach istnieje zagrożenie powstaniem pętli w dziedziczeniu, w trzecim należy zapobiec usunięciu czyjegoś rodzica.

2.3. Wiązanie nazw i weryfikacja

Po stworzeniu definicji interfejsu poleceniem **create interface** opisanym powyżej, obiekt tej definicji zostaje zapisany w bazie danych, ale nie jest jeszcze związany z żadnymi prawdziwymi danymi. Zapytanie o obiekty tego interfejsu zwróci pusty zbiór wynikowy. Aby wykorzystać interfejs do odczytywania i manipulacji danymi, należy go z nimi związać.

2.3.1. Składnia wiązania

Wiązanie interfejsu z implementacją (bądź innym interfejsem) odbywa się przy pomocy polecenia programisty administracyjnego:

```
[Interfejs] shows [Implementacja]
```

Obie abstrakcje mogą być wskazane zarówno poprzez ich nazwę, jak i nazwę ich obiektów. Skoro IPrac jest nazwą interfejsu z obiektami OIPrac, które chcemy związać z obiektami OPrac klasy CPrac, możemy to zrobić na jeden z czterech sposobów:

```
IPrac shows CPrac
OIPrac shows CPrac
IPrac shows OPrac
OIPrac shows OPrac
```

Efekt tych poleceń będzie taki sam - od tej chwili zapytanie o obiekty OIPrac zwróci obiekty OPrac widoczne w specjalny sposób, przez interfejs IPrac. Zapytania IPrac i CPrac oczywiście zwrócą obiekty definicji interfejsu i klasy, a zapytanie OPrac zwróci obiekty klasy CPrac, dokładnie tak, jak przed wiązaniem.

Po lewej stronie polecenia (zapytania) **shows** musi stać nazwa istniejącego interfejsu lub nazwa obiektów istniejącego interfejsu. Po prawej stronie aparat wykonawczy oczekuje interfejsu, klasy, bądź perspektywy, wyspecyfikowanych analogicznie przez ich nazwę bądź przez nazwę ich obiektów. Zauważmy tutaj, że choć można zdefiniować klasy bądź interfejsy bez elementu składni **instance: [nazwa obiektów]** - takie byty są zupełnie na miejscu w świetle dziedziczenia - to nie mają one żadnego sensu jako implementacje. Jeśli ich nazwy wystąpią w poleceniu **shows**, zostanie zgłoszony błąd na poziomie aparatu wykonawczego.

Aby podane w przykładzie interfejsy IOsoba, IPrac, IBogaciPrac zostały związane, wystarczą trzy polecenia programisty administracyjnego, na przykład takie:

```
IOsoba shows COsoba
IPrac shows CPrac
IBogaciPrac shows BogaciPracDef
```

2.3.2. Weryfikacja i dynamika wiązań

Aby nastąpiło wiązanie na poziomie bazy danych, musi ono najpierw zostać pozytywnie zweryfikowane przez aparat wykonawczy. Jeśli obie nazwy w poleceniu **shows** zostaną poprawnie rozpoznanie i mają znaczenia spełniające wymagania opisane w powyższym rozdziale, następuje weryfikacja wiązania. Odpowiadają za to metody klasy InterfaceMatcher, w obrębie której tworzone są jednolite reprezentacje różnych abstrakcji (interfejsu, klasy, widoku), dziedziczące pola i metody rodziców. Dla każdego pola interfejsu musi istnieć pole implementacji;

podobnie dla każdej metody interfejsu musi istnieć metoda implementacji o identycznych parametrach. Jeśli te warunki nie są spełnione, główna metoda klasy InterfaceMatcher zwraca błąd.

Wiązania z widokami nie są weryfikowane. Definicja wirtualnych obiektów perspektywy podlega składni procedur SBQL i daje ogromne możliwości, które utrudniają określenie, jakie obiekty są zwracane w wyniku. Można by obejść to ograniczenie wykonując zapytanie o wirtualne obiekty i analizując ich strukturę, ale na przeszkodzie stoi inne cecha implementacji perspektyw w LoXiM: mogą one być stanowe. Perspektywa ze stanem w momencie weryfikacji może zwracać obiekty pewnej klasy, a kiedy jej stan ulegnie zmianie - obiekty innej klasy, o zupełnie odmiennej strukturze. Perspektywy ze stanem opisuje rozdział **4.5** pracy Dariusza Gryglasa [Gryg08]. W przypadku wiązania interfejsu z widokiem programista administracyjny nie dysponuje mechanizmem automatycznej weryfikacji. Jeśli popełni błąd i nieprawidłowo wyspecyfikuje podzbiór pól i funkcji w interfejsie, obiekty zwracane w wyniku zapytań wykonywanych przez ten interfejs będą nadmiernie ograniczone, albo nawet zupełnie puste.

Aby system pozostał spójny podczas operacji na interfejsach i klasach, zaimplementowałem odpowiednie mechanizmy, które w takich sytuacjach są wywoływane. Po usunięciu poleceniem **delete** interfejsu, klasy bądź widoku, w słownikach wyszukiwane są wiązania dla tych nazw. Wszystkie dotknięte zmianą wiązania są następnie zrywane. W przypadku, gdy interfejs bądź klasa ulegają modyfikacji, każde z wiązań wykorzystujące te abstrakcje jest weryfikowane ponownie. Jeśli wiązanie nie jest już prawidłowe, tj. nie spełnia wymagań wymienionych w poprzednim akapicie, jest ono zrywane. W ten sposób zapewniony jest niezmiennik, że jeśli wiązanie wykorzystujące klasę bądź interfejs istnieje, to jest ono prawidłowe.

2.4. Realizacja zapytań poprzez interfejsy w LoXiM

Obiekty interfejsów są w pełni wirtualne, zapytania ich dotyczące operują więc na powiązanych obiektach implementacji. Przy definiowaniu schematu można określić, jakie operacje chcemy poprzez dany interfejs umożliwić. Opisałem to w następym rozdziale. Aby odróżnić takie operacje od bezpośrednich manipulacji danymi, wprowadziłem do systemu dodatkową informację. Referencje do obiektów uzyskiwanych w wyniku wykonania zapytań realizowanych przez interfejsy są specjalnie oznaczane, co na dalszych etapach manipulacji tymi obiektami umożliwia ograniczenie widoczności i dostępu do pól oraz metod. Mechanizmy te zostały zaimplementowane na kilku poziomach.

2.4.1. Odczytywanie

Kiedy na takiej specjalnie oznaczonej referencji wykonywany jest odczyt obiektu (tzn. operacja deref), klasa InterfaceMaps odpytywana jest o listę pól widocznych przez interfejs właściwy tej referencji. Ta operacja wykonywana jest bardzo szybko - działa na słownikach i dodatkowo korzysta z pamięci podręcznej, która dla danego intefejsu zawiera uwzględniającą dziedziczenie listę jego pól i naturalnie jest unieważniana w przypadku z założenia rzadkich operacji administracyjnych na interfejsach (dodawanie, usuwanie, modyfikacja obiektu definicji). Podobiekty zwracane przez funkcję deref() są następnie filtrowane względem tej listy, dzięki czemu niewidoczne w interfejsie pola nie zostają udostępnione użytkownikowi. Analogiczny mechanizm jest wykorzystywany w funkcji nested(), co gwarantuje odpowiednie

ograniczenie nawigacji po podobiektach.

Dostęp do metod obiektów w LoXiM został zrealizowany poprzez przypisanie poszczególnym sekcjom stosu środowisk listy widocznych w ich obrębie klas tych obiektów. Wołanie metody powoduje wyszukanie jej ciała w klasie. Ten proces jest dokładniej opisany w [Kacz07]. Aby ograniczyć dostęp do metod, które nie są widoczne przez interfejs, dodałem nową informację do sekcji stosu środowisk. Dzięki niej sytuacja, w której metodę klasy wołamy poprzez interfejs, jest odróżniana i następuje odpowiednie odfiltrowanie metod niewidocznych dla użytkownika.

W każdym z powyższych przypadków wykonujący zapytanie poprzez interfejs nie może odróżnić pola niedostepnego od zwyczajnie nie istniejącego.

2.4.2. Tworzenie

Konstrukcja realnych obiektów implementacji przy pomocy zapytań wykorzystujących wirtualne obiekty interfejsu tworzy zasadniczy problem koncepcyjny. Należy zdecydować, jakie wartości będą miały te pola realnego obiektu, które przez interfejs nie są widoczne. Tej decyzji nie można rozwiązać dodatkową składnią, nie może jej bowiem podjąć wykonujący zapytanie - z założenia nie powinien on nic wiedzieć o niedostępnych polach.

Jednym z rozważanych przeze mnie rozwiązań było wprowadzenie domyślnych wartości pól, które można by określić np. w specjalnej funkcji on_create definiowanej przez administratora przy tworzeniu interfejsu. Innym rozwiązaniem mogłoby być zrezygnowanie z możliwości tworzenia obiektów poprzez interfejs - do tego mogłyby służyć np. metody statyczne. Trzecia, zrealizowana przeze mnie możliwość, to po prostu tworzenie obiektów bez niedostepnych pól - mogą one np. być następnie uzupełniane o te pola w kolejnym, wykorzystującym inny schemat i inne interfejsy przypadku użycia. Takie podejście wydało mi się najlepsze, gdyż jest najbardziej naturalne w procesie przepływu informacji - za zarządzaniem różnymi częściami obiektu mogą odpowiadać różne osoby o niejednakowych uprawnieniach.

2.4.3. Usuwanie i modyfikowanie

Przy pomocy polecenia **delete**, wywołanego na wirtualnych obiektach intefejsu, można usuwać związane z nimi obiekty implementacji (o ile ma się do tego prawo). Widoczne pola tych obiektów mogą być również modyfikowane.

Rozdział 3

Schematy zewnętrzne

3.1. Wprowadzenie

Schemat zewnętrzny to zbiór interfejsów zewnętrznych pośredniczący w interakcji między system LoXiM, a aplikacjami z niego korzystającymi. Występuje w dwóch postaciach, odpowiadającym poziomom i rolom w modelu architektury trójwarstwowej. Programista administracyjny operuje na schemacie zewnętrznym zapisanym w bazie danych jako obiekt i elementy struktur pomocniczych. Programista aplikacyjny wykorzystuje wyeksporowaną postać tekstową, w której dokładnie wyspecyfikowana jest budowa dostępnych interfejsów.

Ze strony programisty administracyjnego, schemat zewnetrzny umożliwia manipulację zbiorem nazw widocznych i dostępnych programiście aplikacyjnemu. Taka warstwa pośrednicząca uniezależnia kod aplikacji od struktury obiektów, klas i perspektyw tworzonych i zmienianych przez programistę systemowego. Pozwala na zablokowanie dostepu do danych, które w założeniu mają być niedostepne, bądź po prostu są nieistotne dla użytkowników aplikacji. Dzięki wykorzystaniu interfejsów do perspektyw, schematy zewnętrzne umożliwiają dowolnie skomplikowane odwzorowania obiektów bazy danych na obiekty, na których w warstwie aplikacji wykonywane są zapytania.

Z punktu widzenia programisty aplikacyjnego, tekstowa postać schematu zewnętrznego, dostarczana w postaci pliku, zawiera strukturę obiektów, na których są dozwolone określone operacje. Ten plik powinien zostać wykorzystany do odnalezienia błędów w kodzie na etapie kompilacji. Schemat zewnętrzny jest dla programisty aplikacyjnego rodzajem promesy: jeśli kod wykorzystujący ten schemat się skompiluje, a schemat jest spójny ze swoim odpowiednikiem w bazie danych, nie będzie błędnych ani niedozwolonych odwołań podczas działania aplikacji.

Za zachowanie spójności między dwiema postaciami schematu zewnętrznego odpowiada programista administracyjny. Jeśli definicja schematu zewnętrznego lub jego interfejsów ulegnie zmianie, powinien wyeksportować schemat ponownie i dostarczyć stworzony w ten sposób plik programiście aplikacyjnemu. Taki plik jest w założeniu tylko do odczytu, jakiekolwiek zmiany na nim, np. rozszerzenie o kolejne nazwy, nie dadzą żadnych efektów poza błędami, kiedy nastąpi do nich odwołanie.

Obie postacie schematu zewnętrznego są konieczne. Gdyby istniał tylko jako obiekt definicji w bazie danych, programista aplikacyjny nie miałby możliwości kompilacji kodu bez

odpytania bazy danych. Gdyby wykorzystywana była tylko postać tekstowa, możliwy byłby nieuprawniony dostep poprzez rozszerzenie zawartej w niej przestrzeni nazw.

3.2. Konstrukcja schematu zewnętrznego

Schemat zewnętrzny możemy utworzyć na trzy sposoby. Jednym z nich jest import z pliku, dwa pozostałe odpowiadają alternatywnym wariantom polecenia **create schema**. Podobnie jak w przypadku interfejsów, można określić zachowanie w sytuacji, gdy obiekt o wskazanej nazwie istnieje. W tym celu zamiast słówka **create** można użyć składni **update** lub **create or update**.

3.2.1. Składnia z listą nazw

```
create schema [nazwa schematu]
{
  [nazwa interfejsu 1] [CRUD]
  ...
}
```

Nazwa schematu musi być unikalna w całej bazie danych, a lista nazw interfejsów nie może zawierać powtórzeń. Nie byłyby one szkodliwe z punktu widzenia logiki wykorzystania schematu, ale mogą być sygnałem, że programista administracyjny popełnił błąd. CRUD definiuje listę operacji, którą można przez interfejs wykonywać na obiektach i wszystkich ich polach określonych w tym interfejsie. Składa się z dowolnego podzbioru zbioru następującego zbioru słów: **create read update delete**. Kiedy nie jest wyspecyfikowany, przyjmuje domyślnie wartość **read**, która oznacza możliwość odczytu wszystkich pól obiektu i wykonania wszystkich metod jego klasy, które są widoczne przez interfejs. Uprawnienie **create** pozwala tworzyć obiekty, **delete** pozwala je kasować, a **update** umożliwia modyfikację pól, łącznie z prawem do ich dodawania i usuwania. Uprawnienia te są realizowane przez mechanizm kontroli dostępu do obiektów. Jeśli powyższe zapytanie powiedzie się, w bazie zostanie utworzony obiekt o wskazanej nazwie, reprezentujący schemat zewnętrzny.

3.2.2. Składnia z pełnym wyspecyfikowaniem interfejsów

```
create schema [nazwa schematu]
{
  [definicja interfejsu 1] [CRUD]
  ...
}
```

Składnia definicja interfejsu jest identyczna ze składnią jego konstrukcji, z tym, że nie występuje w niej słowo kluczowe **create**. To polecenie ma inną semantykę - powoduje mianowicie utworzenie wymienionych interfejsów, jeśli takie nie istnieją jeszcze w bazie danych. W takim przypadku, jeśli utworzenie któregoś z nich nie powiedzie się, zapytanie **create schema** zwraca błąd.

3.2.3. Składowanie obiektów definicji schematów zewnętrznych

Na poziomie Store'a zarządzanie tymi obiektami jest zrealizowane analogicznie jak w przypadku interfejsów. Schematom zewnęrznym odpowiada plik *sbschemas* i struktury pomocnicze

oraz metody zawarte w klasach OuterSchema oraz OuterSchemas. Te struktury inicjowane są przy starcie serwera.

3.3. Poprawność i weryfikacja

Przechowywany w bazie schemat zewnętrzny określa cecha poprawności. W przeciwieństwie do wiązania schemat zewnętrzny nie jest usuwany, jeśli na skutek zmian stanie się niepoprawny. W takiej sytuacji próba wykorzystania schematu do interakcji z bazą danych zakończy się odpowiednim błędem. Odpowiedzialność za poprawność schematów ponosi programista administracyjny.

Schemat jest zbiorem interfejsów, do których jego użytkownikowi przysługuje określony dostęp. Aby weryfikacja schematu zakończyła się pozytywnie, każdy z interfejsów musi istnieć bazie. Co więcej, jeśli zawiera element składni **instance**, czyli definiuje nazwę obiektów, musi być związany z ostateczną implementacją. To oznacza, że jeśli interfejs A, występujący w schemacie, jest związany z interfejsem B, który jednak nie jest związany dalej z żadną abstrakcją, schemat zostanie oceniony jako niepoprawny. Takiego schematu nie można wyeksportować.

Pilnowanie, by ocena poprawności zawsze była aktualna, zostało zrealizowane przy wykorzystaniu słownika tłumaczącego nazw abstrakcji na nazwy schematów, które te abstrakcje wykorzystują. Jeśli interfejs zostanie zmodyfikowany, wszystkie schematy, które go udostępniają, weryfikowane są ponownie. Jeśli wiązanie zostało zerwane (np. na skutek usunięcia klasy implementacji), sprawdzenie poprawności przeprowadzane jest analogicznie dla wszystkich dotkniętych tą zmianą interfejsów.

3.4. Ograniczenie dostępu do nazw

Celem istnienia schematów zewnętrznych jest ograniczenie i przekształcenie przestrzeni obiektów bazy danych tak, by odpowiadała potrzebom konkretnej aplikacji. Potrzebny jest zatem mechanizm kontroli dostępu do obiektów. Funkcjonalność taka wydawała się być zaimplementowana w LoXiM, jednak okazało się, że rozwiązanie to od bardzo dawna nie było wspierane w rozwijającym się kodzie, nigdy też niestety nie działało tak, jak tego wymaga semantyka ograniczenia dostępu przez schematy. Chcielibyśmy, żeby dostęp do obiektu implikował dostęp do jego podobiektów. W tym celu musiałem zmodyfikować mechanizm kontroli dostępu w LoXiM.

Aby osiągnąć wymaganą semantykę kontroli dostępu do obiektów, zaimplementowałem klasę AccessMap. Jej struktury określają, jakie nazwy są dostępne w obecnym kontekście i jakie są do nich uprawnienia. Kiedy do bazy loguje się nowy użytkownik, znajdywany jest przypisany mu schemat, a na jego podstawie generowany jest podstawowy zbiór uprawnień. Podczas wykonywania zapytania ten zbiór jest dynamicznie rozszerzany i ograniczany, tak aby następujące zasady były przestrzegane:

- Jeśli użytkownik ma prawo odczytu lub modyfikacji obiektu, to ma takie same prawo względem jego podobiektów

- Jeśli użytkownik ma prawo modyfikacji obiektu, to dodatkowo ma prawo do tworzenia i usuwania jego podobiektów
- Każda metoda wołana na obiekcie ma pełny dostęp do wszystkich pól i metod tego obiektu, nawet tych niewidocznych przez interfejs (oczywiście wołana metoda sama musi być widoczna)

3.5. Eksportowanie i importowanie schematu zewnętrznego

Jak wspomniałem, schemat zewnętrzny w LoXiM występuje w dwóch postaciach: jako obiekt w bazie danych i jako tekstowy plik dostarczany programiście aplikacyjnemu. Do tłumaczenia między tymi postaciami i zachowania spójności służą polecenia eksportu i importu schematu zewnętrznego. Jeśli schemat nosi nazwę SFinance, składnia tych poleceń będzie następująca:

```
export SFinance
import SFinance
```

Wygenerowany plik będzie nosił taką samą nazwę, jak schemat. Importowanie schematu zachowuje się analogicznie do tworzenia go przy pomocy składni nr 2. Wygenerowany plik jest tylko do odczytu.

3.6. Przykład zastosowania

Niech poniższe definicje klasy stanowią część struktury danych związanych z pewnym systemem. W bazie trzymane są dane dotyczące pacjentów i lekarzy. Niektóre obiekty złożone, takie jak kalendarzWizyt, nie mają wyszczególnionej struktury, nietrudno jednak się domyśleć, jaka ona mogłaby być. Celem niniejszego przykładu nie jest stworzenie działającego systemu dla ośrodka medycznego, a jedynie zilustrowanie istotnych aspektów zastosowania schematów i interfejsów.

```
create class COsoba
{
  instance OOsoba:
  {
    imie;
    nazwisko;
    dataUrodzenia;
    adres;
  }
}

create class CPacjent extends COsoba
{
  instance OPacjent:
  {
    kalendarzWizyt;
    historiaChorób;
```

```
    rodzajKarty;
    historiaWpłat;
  }
  procedure dodajDiagnoze(diagnoza, data) {...}
}

create class CLekarz extends COsoba
{
  instance OLekarz:
  {
    specjalności;
    kalendarzGodzinPracy;
    kalendarzWizyt;
    zarobki;
  }
  procedure dajPowyżkę(kwota) {zarobki := zarobki + kwota; return}
  procedure wolneTerminy(odDnia, dniWPrzód) { ... }
  procedure dodajWizytę(data) { ... }
  procedure odwołajWizytę(data) { ... }
  procedure przełóżWizytę(dataPrzed, dataPo) { odwołajWizytę(dataPrzed);
                                               return dodajWizytę(dataPo) }
}
```

Programista administracyjny ma za zadanie zdefiniować interfejsy i schematy do trzech aplikacji, z których jedną będą wykorzystywać osoby pracujące w recepcji, druga przeznaczona jest dla lekarzy, a trzeciej używają pracownicy działu promocji w Warszawie, np. do zapraszania posiadaczy złotych kart na wydarzenia kulturalne w tym mieście.

Najpierw wygodnie jest zdefiniować interfejs do klasy COsoba, aby wykorzystać mechanizm dziedziczenia:

```
create interface IOsoba
{
  {
    imie;
    nazwisko;
    dataUrodzenia;
    adres;
  }
}
```

Tak zdefiniowany interfejs nie ma wyspecyfikowanej nazwy obiektu - żadna z docelowych aplikacji nie będzie przetwarzać danych lekarzy i pacjentów razem w kontekście osób. Następnie programista administracyjny tworzy interfejsy dla aplikacji wykorzystywanej w recepcji. Ma ona umożliwiać sprawdzanie grafików lekarzy, umawianie, odwoływanie i przekładanie wizyt, a także dostęp do danych pacjenta, z wyłączeniem danych o chorobach. Te dane, oraz dane o zarobkach lekarzy, powinny być niewidoczne dla aplikacji używanej w recepcji.

```
create interface ILekarz extends IOsoba
{
  instance OILekarz:
  {
    specjalności;
    kalendarzGodzinPracy;
    kalendarzWizyt;
  }
  procedure wolneTerminy(odDnia, dniWPrzód)
  procedure dodajWizytę(data)
  procedure odwołajWizytę(data)
  procedure przełóżWizytę(dataPrzed, dataPo)
}

create interface IPacjent extends IOsoba
{
  instance OIPacjent:
  {
    kalendarzWizyt;
    rodzajKarty;
    historiaWpłat
  }
}

ILekarz shows CLekarz;
IPacjent shows CPacjent;

create schema SRecepcja
{
  interface ILekarz;
  interface IPacjent
}
```

Informacje o zarobkach lekarzy i chorobach pacjentów są niedostępne w schemacie SRecepcja. Programiście administracyjnemu pozostaje stworzyć nowego użytkownika bazy danych, określając login i hasło, a następnie nadać mu uprawnienia do schematu SRecepcja i wyeksportować ten schemat. Programista aplikacyjny otrzyma schemat i zgodnie z nim napisze kod aplikacji dla recepcji.

Schemat dla aplikacji dla lekarzy będzie wykorzystywał rozszerzony interfejs do klasy pacjenta:

```
create interface IPacjentChoroby extends IPacjent
{
  instance OIPacjentChoroby:
  {
    historiaChorób;
  }
```

```
  procedure dodajDiagnozę(diagnoza, data)
}

IPacjentChoroby shows CPacjent;

create schema SLekarze
{
  interface IPacjentChoroby;
}
```

Aplikacja przeznaczona dla działu promocji w Warszawie powinna mieć dostęp do danych pacjentów posiadających złotą kartę, którym będą wysyłane propozycje i oferty. Doskonale sprawdzi się tutaj perspektywa i interfejs do niej. Do ich zdefiniowania można użyć następujących poleceń:

```
create view ZłociPacjenciZWarszawyDef
{
  virtual_objects ZłociPacjenciZWarszawy()
  {
    return (OPacjent where rodzajKarty = "złota" and adres.miasto = "Warszawa") as pa
  }

  procedure on_retrieve()
  {
    return deref(pa)
  }
}

create interface IZłociPacjenciZWarszawy extends IPacjent
{
  instance OIZłociPacjenciZWarszawy:
  {
  }
}

IZłociPacjenciZWarszawy shows ZłociPacjenciZWarszawyDef;

create schema SPromocja
{
  interface IZłociPacjenciZWarszawy;
}
```

Przestrzeń obiektów reprezentujących pacjentów w bazie danych została w ten sposób przekształcona na potrzeby konkretnego zastosowania.

Rozdział 4

Podsumowanie

W wyniku mojej pracy system LoXiM został rozszerzony o interfejsy i schematy zewnętrzne. Implementacja daje szerokie możliwości wynikające z wykorzystania modelu trójwarstwowego. Używanie interfejsów w interakcji z bazą danych jest wygodne i przezroczyste, nie narusza też w żaden sposób spójności danych. Programista administracyjny pracujący z systemem LoXiM dysponuje teraz silnymi metodami przekształcania przestrzeni obiektów w bazie na potrzeby poszczególnych aplikacji. Rozwiązany został również problem kontroli dostępu. Implementacja pokrywa wszystkie zaplanowane aspekty wykorzystania interfejsów i schematów, ich tworzenie, usuwanie, modyfikację, wiązanie (interfejsów), importowanie i eksportowanie (schematów).

Realizując zadanie stworzenia interfejsów i schematów wewnętrznych w LoXiM zwracałem uwagę na to, by łatwo można było ich implementację modyfikować i rozszerzać. Jest ona podzielona na klasy C++, które izolują poszczególne abstrakcje i związane z nimi funkcjonalności.

Ze względu na brak możliwości określania typów pól w klasach zdecydowałem się zrezygnować z takich definicji również w interfejsach. Pociąga to za sobą pewne niedogodności związane z odwoływaniem się przez interfejs do pól będących obiektami złożonymi. W podanym przeze mnie przykładzie takim polem jest np. *adres* w klasie COsoba. W schemacie, przez interfejs (ani w samej klasie) nie jest widoczna struktura tego obiektu, nie koniecznie więc musi być wiadomo, jakie podobiekty wchodzą w jego skład. Nie ma w obecnej implementacji możliwości określenia, że pole adres jest np. obiektem pewnej klasy CAdres, choć programista administracyjny może tak przypuszczać, sugerując się nazwą.

Możliwe jest rozszerzenie klas i interfejsów o typy pól i argumentów funkcji. W ten sposób rozwiązany zostanie problem z nieznaną strukturą złożonego podobiektu, opisany powyżej. Dokonując takiej zmiany, należy między innymi zadbać o sprawdzanie zgodności typów przy wiązaniu interfejsu z implementacją, a także odpowiednio rozszerzyć struktury reprezentujące klasy i interfejsy. Takie modyfikacje są koncepcyjnie nietrudne.

Obecną implementację interfejsów można w przyszłości rozszerzyć o składowe statyczne. Klasy w LoXiM pozwalają na definiowanie statycznych pól i metod, do których nie ma bezpośredniego dostępu z poziomu interfejsu. Dodanie składowych statyczych do struktury interfejsu pozwoliłoby na pełniejsze wykorzystanie możliwości, jakie oferują klasy.

Wiązania interfejsów z widokami nie są obecnie weryfikowane z powodu problemu z określeniem postaci obiektów wirtualnych, zwracanych jako rezultat wykonania kodu perspektywy. Można wprowadzić kontrolę poprawności dla tych wiązań, na przykład poprzez dodanie opcjonalnego elementu składni definicji widoku, który określałby, jak zbudowane są jego wirtualne obiekty. Taka struktura byłaby w procesie weryfikacji porównywana z interfejsem.

Dodatek A

Przykładowa sesja z LoXiM

A.1. Administrator tworzy schematy zewnętrzne

Poniższy zapis sesji z systemem LoXiM ilustruje wykorzystanie schematów zewnętrznych i interfejsów. Pierwszą serię zapytań wykonuje administrator, tworząc obiekty definicji, określając wiązania interfejsów i przydzielając schematy zewnętrzne pozostałym użytkownikom.

```
[adabrowski@localhost Programs]$ ./lsbql
Welcome to loxim client.

Login: root
Password: Connected.
Type $help to see the available commands
Authorized.
> begin;
void
> $slash
Switching to slash mode
> --Wykonane polecenie ustawiło odpowiedni wariant składni
\ --Tworzymy klasę COsoba.
\ --Wynikiem tego zapytania jest referencja do obiektu definicji klasy.
\ create class COsoba
\ {
\   instance OOsoba:
\   {
\     imie;
\     nazwisko;
\     rokUrodzenia;
\     adres
\   }
\ }
\ /
ref(14297)
> --Tworzymy klasę CPrac, która dziedziczy po klasie COsoba.
\ create class CPrac extends COsoba
\ {
\   instance OPrac:
```

```
\   {
\     zarobki
\   }
\   procedure dajPodwyzke()
\   {
\     zarobki := 1.1 * zarobki
\   }
\ }
\ /
ref(14304)
> --Tworzymy dwa obiekty należące do klasy CPrac.
\ --Wynikiem zapytania będą referencje do stworzonych obiektów.
\ CPrac includes
\ (
\   create
\   (
\     "Jan" as imie,
\     "Kowalski" as nazwisko,
\     (
\       "Marszalkowska" as ulica,
\       12 as numerDomu,
\       3 as numerMieszkania,
\       "Warszawa" as miasto
\     ) as adres,
\     5000 as zarobki
\   ) as OPrac
\ )
\ /
Bag
  ref(14313)
> CPrac includes
\ (
\   create
\   (
\     "Jolanta" as imie,
\     "Nowak" as nazwisko,
\     (
\       "Warszawska" as ulica,
\       1 as numerDomu,
\       12 as numerMieszkania,
\       "Szczecin" as miasto
\     ) as adres,
\     5500 as zarobki
\   ) as OPrac
\ )
\ /
Bag
  ref(14322)
> --Definiujemy interfejs IOsoba.
```

```
\ --Interfejs ten nie pozwala na dostęp do obiektów, gdyż nie definiuje
\ --nazwy niezmienniczej (brak składni instance). Można natomiast po nim
\ --dziedziczyć i do tego w tym przykładzie służy.
\ --Wynikiem zapytania jest referencja do obiektu definicji interfejsu.
\ create interface IOsoba
\ {
\   {
\     imie;
\     nazwisko
\   }
\ }
\ /
ref(14328)
> --Tworzymy dwa rożne interfejsy dziedziczące po interfejsie IOsoba.
\ create interface IPrac1 extends IOsoba
\ {
\   instance OIPrac1:
\   {
\     adres;
\     rokUrodzenia
\   }
\ }
\ /
ref(14335)
> create interface IPrac2 extends IOsoba
\ {
\   instance OIPrac2:
\   {
\     zarobki
\   }
\   dajPodwyzke()
\ }
\ /
ref(14342)
> --Definiujemy perspektywę PracZWarszawyDef.
\ --Umożliwia ona odczyt danych wszystkich pracowników z Warszawy.
\ create view PracZWarszawyDef
\ {
\   virtual objects PracZWarszawy()
\   {
\     return OPrac where adres.miasto = "Warszawa" as pw
\   }
\   procedure on_retrieve()
\   {
\     return deref(pw)
\   }
\ }
\ /
ref(14348)
```

```
> --Do perspektywy PracZWarszawyDef wstawiamy procedurę pomocniczą
\ PracZWarszawyDef :<
\   procedure dorosly(rokUrodzenia)
\   {
\     if 2009 - rokUrodzenia >= 18 then
\       return true
\     else
\       return false
\     fi
\   }
\ /
void
> --Do perspektywy PracZWarszawyDef wstawiamy kolejne podperspektywy.
\ --Ma to na celu umożliwienie dostępu do pól jej wirtualnych obiektów.
\ PracZWarszawyDef :<
\   view imieDef
\   {
\     virtual objects imie()
\     {
\       return pw.imie as i
\     }
\     procedure on_retrieve()
\     {
\       return deref(i)
\     }
\     procedure on_update(new_val)
\     {
\       i := new_val
\     }
\   }
\ /
void
> PracZWarszawyDef :<
\   view nazwiskoDef
\   {
\     virtual objects nazwisko()
\     {
\       return pw.nazwisko as n
\     }
\     procedure on_retrieve()
\     {
\       return deref(n)
\     }
\     procedure on_update(new_val)
\     {
\       n := new_val
\     }
\   }
\ /
```

```
void
> PracZWarszawyDef :<
\   view rokUrodzeniaDef
\   {
\     virtual objects rokUrodzenia()
\     {
\       return pw.rokUrodzenia as r
\     }
\     procedure on_retrieve()
\     {
\       return deref(r)
\     }
\     procedure on_update(new_val)
\     {
\       if dorosly(new_val) then
\         n := new_val
\       fi
\     }
\   }
\ /
void
> PracZWarszawyDef :<
\   view adresDef
\   {
\     virtual objects adres()
\     {
\       return pw.adres as a
\     }
\     procedure on_retrieve()
\     {
\       return deref(a)
\     }
\     procedure on_update(new_val)
\     {
\       a := new_val
\     }
\   }
\ /
void
> --Poleceniem shows wiążemy interfejs IPrac1 z widokiem PracZWarszawyDef
\ --Wynikiem jest potwierdzenie utworzenia wiązania
\ IPrac1 shows PracZWarszawyDef
\ /
bound
> --Interfejs IPrac2 związany zostaje z klasą CPrac
\ IPrac2 shows CPrac
\ /
bound
> --Definiujemy schemat zewnętrzny SWarDane.
```

```
\ --Ten schemat zezwala na korzystanie tylko z interfejsu IPrac1.
\ --Po nazwie interfejsu wymienione są dozwolone operacje.
\ create schema SWarDane
\ {
\   IPrac1 read update delete
\ }
\ /
ref(14390)
> --Tworzymy schemat SZarobki, wykorzystujący interfejs IPrac2
\ --Dostęp odczytu do IPrac2 pozwala na wykonywanie jego metod
\ create schema SZarobki
\ {
\   IPrac2 read
\ }
\ /
ref(14393)
> --Schematy są eksportowane
\ --W wyniku tych poleceń tworzone są pliki SWarDane.sch i SZarobki.sch
\ export SWarDane
\ /
exported
> export SZarobki
\ /
exported
> --Tworzymy użytkownika wardane i przypisujemy mu schemat SWarDane
\ create user wardane passwd haslo
\ /
1
> grant read on SWarDane to wardane
\ /
1
> --Tworzymy użytkownika szarobki i przypisujemy mu schemat SZarobki
\ create user szarobki passwd haslo
\ /
1
> grant read on SZarobki to szarobki
\ /
1
> --Koniec sesji administratora
\ end
\ /
void
> quit
Server disconnected
```

A.2. Sesja użytkownika ze schematem SWarDane

W dalszej części przykładu dostęp do bazy LoXiM uzyskuje inny, nie uprzywilejowany użytkownik. Jego dostęp do danych jest ściśle określony przypisanym mu wcześniej przez administratora schematem zewnętrzny.

```
[adabrowski@localhost Programs]$ ./lsbql
Welcome to loxim client.

Login: wardane
Password: Connected.
Type $help to see the available commands
Authorized.
> begin;
void
> $slash
Switching to slash mode
> --Zapytanie bezpośrednio o obiekty OPrac nie zwróci żadnych danych, bo
\ --użytkownik nie ma do nich prawa. Może jedynie korzystać z interfejsów
\ --należących do przypisanego mu schematu zewnętrznego
\ OPrac
\ /
Bag
> --Poniższe zapytanie nie da żadnych rezultatów z podobnych przyczyn
\ PracZWarszawyDef
\ /
Bag
> --Dostęp do obiektów OIPrac1 interfejsu IPrac1 jest zgodny ze schematem
\ --Operacja odczytu na tym obiekcie powiedzie sie, zwracając jego strukturę
\ deref(OIPrac1)
\ /
Struct
  imie=>
    Jan
  nazwisko=>
    Kowalski
  adres=>
    Struct
      ulica=>
        Marszalkowska
      numerDomu=>
        12
      numerMieszkania=>
        3
      miasto=>
        Warszawa
> --Przy pomocy interfejsu IPrac1 można rownież zmieniać pola tej struktury:
\ (OIPrac1 where nazwisko = "Kowalski").imie := "Jaroslaw"
\ /
```

```
void
> --Pole odpowiedniego obiektu OPrac zostało zmienione
\ deref(OIPrac1)
\ /
Struct
  imie=>
    Jaroslaw
  nazwisko=>
    Kowalski
  adres=>
    Struct
      ulica=>
        Marszalkowska
      numerDomu=>
        12
      numerMieszkania=>
        3
      miasto=>
        Warszawa
> --Możemy zmienić rownież adres, wstawiając obiekt złożony
\ (OIPrac1 where nazwisko = "Kowalski").adres
\ := ("Staromiejska" as ulica,
\     2 as numerDomu,
\     15 as numerMieszkania,
\     "Krakow" as miasto)
\ /
void
> --Kiedy teraz spróbujemy odczytać OPrac przez interfejs OIPrac1, Kowalski
\ --nie zostanie wypisany, bo tylko pracownicy z Warszawy są widoczni.
\ deref(OIPrac1)
\ /
Bag
> --Koniec sesji użytkownika wardane
\ end
\ /
void
> quit
Server disconnected
```

A.3. Sesja użytkownika ze schematem SZarobki

W ostatniej części przykładu zapytania wykonuje kolejny użytkownik, dysponujący schematem umożliwiającym wykonanie metody dajPodwyzke na obiektach reprezentujących pracowników.

```
[adabrowski@localhost Programs]$ ./lsbql
Welcome to loxim client.

Login: szarobki
```

```
Password: Connected.
Type $help to see the available commands
Authorized.
> begin;
void
> $slash
Switching to slash mode
> --Analogicznie jak dla poprzedniego użytkownika, bezpośrednie zapytanie
\ --o obiekty OPrac zwróci pusty wynik
\ OPrac
\ /
Bag
> --Zapytanie przez interfejs IPrac2 pokaże odpowiednio ograniczone
\ --obiekty OPrac
\ deref(OIPrac2)
\ /
Bag
  Struct
    imie=>
      Jaroslaw
    nazwisko=>
      Kowalski
    zarobki=>
      5000
  Struct
    imie=>
      Jolanta
    nazwisko=>
      Nowak
    zarobki=>
      5500
> --Możemy wykorzystać dostepną przez interfejs metodę dajPodwyzkę.
\ --Wywoła się na wszystkich obiektach spełniających warunek where
\ (OIPrac2 where zarobki < 6000).dajPodwyzke()
\ /
Bag
> --Sprawdzamy efekt wykonania metody
\ deref(OIPrac2.zarobki)
\ /
Bag
  5500
  6050
> --Próbujemy skasować obiekty OPrac przez interfejs IPrac2.
\ --Ze względu na brak uprawnień pozostanie to bez efektu.
\ delete OIPrac2
\ /
void
> OIPrac2
\ /
```

```
Bag
  ref(14313)
  ref(14322)
> --IPrac2 nie pozwala na modyfikację pól. Próba bezpośredniej zmiany
\ --pola zarobki nie powiedzie się
\ (OIPrac2 where imie = "Jaroslaw").zarobki := 200
\ /
void
> deref(OIPrac2.zarobki)
\ /
Bag
  5500
  6050
> --Koniec sesji użytkownika szarobki
\ end
\ /
void
> quit
Server disconnected
```

Bibliografia

[HSS06] Piotr Habela, Krzysztof Stencel, Kazimierz Subieta, *Three-Level Object-Oriented Database Architecture Based on Virtual Updateable Views*, Warszawa 2006

[Sub04] Kazimierz Subieta, *Teoria i konstrukcja obiektowych języków zapytań*, PJWSTK, Warszawa 2004

[Gryg08] Dariusz Gryglas, *Implementacja aktualizowalnych perspektyw w systemie LoXiM*, Warszawa 2008

[Kacz07] Sebastian Kaczanowski, *Implementacja klas w systemie LoXiM*, Warszawa 2007

[LoXiM] strona WWW systemu LoXiM: *http://loxim.mimuw.edu.pl*

[SQBL] strona WWW przedstawiająca podejście SBA i język SBQL: *http://www.sbql.pl*