Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Tomasz Rosiek
Nr albumu: 201089

Przezroczyste odwzorowanie semistrukturalnej bazy danych na Javę

Praca magisterska
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem
dra hab. Krzysztofa Jana Stencla
Instytut Informatyki

Sierpień 2007

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

Celem pracy jest zaprojektowanie i zaimplementowanie narzędzia umożliwiającego przeglądanie i modyfikowanie obiektów w bazie danych bez potrzeby jawnego wywoływania zapytań. Dodatkowo narzędzie ma umożliwiać przezroczyste dla programisty odwzorowanie obiektów z bazy semistrukturalnej na interfejsy Javy i ukryć przed nim mechanizmy dostępu do bazy danych, pozwalając skupić się na implementowaniu logiki aplikacji z wykorzystaniem obiektowych metod projektowania.

Słowa kluczowe

SBQL, odwzorowanie, semistrukturalne bazy danych, obiektowe bazy danych, Java

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

Klasyfikacja tematyczna

D. Software
D.3 PROGRAMMING LANGUAGES
D.3.3 Language Constructs and Features
Data types and structures

Tytuł pracy w języku angielskim

Transparent mapping of a semistructural database to Java

Spis treści

Wprowadzenie

Baza danych traktowana jest zazwyczaj jako zewnętrzny zasób, któremu można zadawać polecenia pobrania lub modyfikacji danych (zapytania). Po wykonaniu zapytania otrzymujemy wynik zawierający żądane dane bądź rezultat operacji. Na tej zasadzie przebiega współpraca z najpopularniejszymi w tej chwili relacyjnymi bazami danych opartymi na języku zapytań SQL. Tworzenie programów korzystających z bazy danych wiąże się z koniecznością jawnego wykonywania operacji pobierania i modyfikacji danych, interpretowania otrzymanych wyników i konwersji ich do struktur specyficznych dla języka programowania.

W pewnym momencie pojawił się pomysł aby zintegrować bazę danych z językiem programowania. Chodziło o to, aby cecha trwałości obiektu była ortogonalna w stosunku do logiki programu. Powstało kilka propozycji rozszerzeń języków programowania pozwalających operować na trwałych obiektach bezpośrednio z poziomu języka – dla języków strukturalnych (Ada95[6], DBPL bazujący na Moduli2[9]) czy funkcyjnych (rozszerzenie Common LISP[7]).

Możliwość pracy z obiektami w bazie danych w taki sam sposób jak z lokalnymi obiektami języka programowania miała uprościć kod programu, a także zwiększyć jego uniwersalność. Użytkownik mógłby skupić się na logice programu a nie na niuansach dostępu do bazy. Ułatwione jest także tworzenie i testowanie kodu, który można efektywnie rozwijać bez podłączenia do bazy danych. Wydzielenie z kodu programu mechanizmów dostępu do danych zmniejsza także ilość używanych w kodzie technologii – logika programu nie musi już być przeplatana operacjami dostępu do danych (często specyficznymi dla konkretnej technologii).

Rozwiązania umożliwiające nadanie obiektom trwałości dostępne są również dla większości popularnych języków obiektowych (takich jak Ruby, Python, C# czy Smalltalk). Istnieje także kilka tego typu narzędzi dla języka Java. Opierają się one na relacyjnych bazach danych i stanowią swego rodzaju warstwę abstrakcji pomiędzy czystym językiem SQL a językiem obiektowym. Oczywiście dodatkowa warstwa powoduje wymierny narzut wydajnościowy i w wielu sytuacjach ma ograniczoną funkcjonalność w stosunku do bezpośredniego dostępu do bazy danych. Z drugiej strony uzyskujemy znaczne uproszczenie i poprawę czytelności kodu, a implementacja pewnych rozwiązań (jak np. buforowanie) jest łatwiejsza. Podczas prac nad systemem zarządzania bazami danych LoXiM[3] pojawił się pomysł zaprojektowania narzędzia analogicznego do opisanych powyżej, pozwalającego na nadanie obiektom języka Java cechy trwałości i przechowywanie ich w bazie semistrukturalnej, a z drugiej strony na operowanie na obiektach z bazy semistrukturalnej jak na zwykłych obiektach Javy. Model obiektów używany przez bazę semistrukturalną jest zbliżony do modelu wykorzystywanego w języku Java – wydaje się więc, że integracja między powyższymi środowiskami może przebiec łatwo i bez nadmiernego zawężania funkcjonalności.

Celem tej pracy jest wykonanie narzędzia umożliwiającego odwzorowanie obiektów bazy danych LoXiM na obiekty języka programowania Java i w ten sposób pozwalającego na nadanie obiektom Javy cechy ortogonalnej trwałości. Otrzymane narzędzie ma służyć jako wygodny interfejs pozwalający w praktyce wykorzystać system zarządzania bazami danych LoXiM.

Rozdział 1

Cele i założenia

1.1. Cele

Celem tej pracy jest zaprojektowanie narzędzia umożliwiającego odwzorowanie obiektów semistrukturalnej bazy danych na obiekty języka programowania Java. Narzędzie to pozwalałoby na dokonywanie operacji na obiektach w bazie a także na nadawanie obiektom Javy cechy trwałości i późniejsze pobieranie utrwalonych obiektów z bazy danych i modyfikowanie ich.

Narzędzie ma operować na dwóch poziomach abstrakcji. W niższej warstwie każdy obiekt bazodanowy ma być reprezentowany przez obiekt specjalnej klasy Javy (Node). Klasa ta definiuje metody pozwalające na dokonywanie operacji na obiekcie w bazie. W warstwie wyższej rozwiązanie to daje możliwość odwzorowania na bazę danych dowolnych stworzonych przez użytkownika klas. Użytkownik, po określeniu sposobu odwzorowania, jest w stanie utrwalić obiekty własnych klas w bazie danych i operować na nich zarówno za pomocą narzędzi bazy (takich jak język zapytań SBQL), jak i za pomocą języka programowania. Funkcjonalność powyższych warstw abstrakcji zostanie zaimplementowana jako dwa oddzielne moduły:

- Moduł nawigacji po bazie danych (warstwa niższa)
- Moduł przezroczystego odwzorowania (warstwa wyższa)

1.2. Moduł nawigacji po bazie danych

Moduł nawigacji po bazie danych ma umożliwiać przeglądanie drzewa obiektów w bazie oraz tworzenie i modyfikację poszczególnych obiektów. Szczegółowa funkcjonalność modułu nawigacji obejmuje:

- Wykonanie zapytania i zwrócenie obiektów bazodanowych (obiektów Node)
- Pobranie obiektu o określonym identyfikatorze
- Pobranie dzieci danego obiektu
- Usunięcie obiektu
- Dodanie/usunięcie obiektu podrzędnego do obiektu złożonego
- Zmiana wartości obiektu prostego

1.3. Moduł przezroczystego odwzorowania

Moduł przezroczystego odwzorowania ma umożliwiać odwzorowanie obiektów bazodanowych na interfejsy Javy zgodne ze standardem JavaBeans[5]. Ma on udostępniać następującą funkcjonalność:

- Odwzorowanie obiektów bazy semistrukturalnej na hierarchię zwykłych klas Javy (klasy zgodne ze standardem JavaBeans[5])
- Utrwalenie w bazie danych obiektu klasy posiadającej odwzorowanie
- Pobranie z bazy danych obiektów Javy określonych przez zapytanie
- Przezroczyste dla użytkownika modyfikowanie obiektów w bazie danych, gdy modyfikowane są reprezentujące je obiekty Javy
- Usuwanie obiektów z bazy danych
- Odwzorowanie podstawowych kolekcji występujących w języku Java

1.4. Baza danych LoXiM

Narzędzie implementowane w ramach tej pracy ma współpracować z systemem zarządzania bazami danych LoXiM[3]. LoXiM obsługuje bazy semistrukturalne, przechowujące dane w postaci hierarchii obiektów. Dostęp do danych odbywa się za pomocą stosowego języka zapytań SBQL [10]. Podczas realizacji pracy wykorzystana została wersja szbd LoXiM z dnia 17.07.2007. Posiada ona następujące cechy:

- Przechowywanie obiektów zgodnie z modelem obiektowości M0
- Implementacja języka zapytań SBQL
- Transakcyjność
- Komunikacja z serwerem bazy danych pomocą protokołu TCP
- W zasadzie brak schematu bazy danych – możliwe jest dołączenie schematu do bazy, natomiast jest on używany tylko w celach optymalizacyjnych i nie ogranicza struktury danych.
- Brak procedur składowanych i metod w obiektach

Rozdział 2

Istniejące rozwiązania

2.1. Odwzorowanie obiektowo-relacyjne

Istnieje wiele popularnych narzędzi pozwalających na odwzorowanie informacji z baz danych w formie obiektów języka programowania. Są one przeznaczone głównie do baz relacyjnych (stąd nazwa ORM [Object-Relational Mapping]). Ich przydatność wynika z faktu, że przy wielkiej popularności obiektowego podejścia do projektowania systemów informatycznych, wciąż powszechnie stosowane są relacyjne bazy danych.

Aby czytelnie operować danymi z baz relacyjnych w obiektowo zaprojektowanej aplikacji, musimy mieć narzędzie, które pozwoli na zapisanie danych obiektowych wraz z ich specyficznymi cechami (dziedziczenie, kompozycja obiektów) w bazie relacyjnej. Dzięki narzędziom ORM programista aplikacji nie musi interesować się sposobem zapisu danych – może po prostu operować na abstrakcyjnych obiektach, a operacje te zostaną automatycznie przełożone na odpowiednie zapytania dla bazy danych. Istnieje wiele tego typu narzędzi dla języka Java. Do najpopularniejszych należą Hibernate, JDO, iBatis czy Java Persistence API (część standardu Java EE).

2.1.1. Typowy scenariusz pracy z narzędziami ORM

Tworzenie aplikacji korzystającej z narzędzia ORM do dostępu do bazy danych wygląda z reguły następująco:

- Projektujemy obiektowy model danych
- Projektujemy aplikację w oparciu o powyższy model
- Definiujemy odwzorowanie obiektowego modelu danych na tabele w bazie relacyjnej
- Implementujemy aplikację, operując na obiektach z bazy danych tak, jak na obiektach Javy

Co istotne – zarówno projektant jak i programista mogą operować na modelu obiektowym i zależnościach między obiektami, a nie na związkach między encjami w tabelach. Można też wyeliminować niektóre sztuczne pojęcia i operacje charakterystyczne dla baz relacyjnych (na przykład złączenie tabel, konieczność definiowania dodatkowej tabeli dla relacji wiele do wielu). Narzędzia odwzorowania obiektowo-relacyjnego pozwalają nam także na realizację dziedziczenia obiektów przechowywanych w bazie danych – odbywa się to niezależnie od ewentualnych mechanizmów dziedziczenia występujących w używanym systemie zarządzania bazami danych.

2.1.2. Wady narzędzi ORM

Narzędzia odwzorowania obiektowo-relacyjnego, choć znacząco ułatwiają projektowanie i implementację aplikacji, mają jednak kilka istotnych ograniczeń i wad. Oto najistotniejsze:

- Niedopasowanie impedancji – odwzorowanie danych z bazy na obiekty nie jest pełne. Praktycznie w każdym z opisanych powyżej rozwiązań istnieje wiele możliwości dokoniania nieprzewidzianej przez twórców narzędzia zmiany stanu obiektu bazodanowego. Zmiana taka może powodować nieoczekiwane efekty i utratę integralności danych. W szczególności każde z narzędzi ORM narzuca pewne ograniczenia na formę obiektów, które mogą być utrwalone w bazie. Na przykładzie narzędzia Hibernate:
  - Brak obsługi kolejek i stosów.
  - Nieefektywna obsługa list (Nie można pobrać pojedynczego elementu bez pobrania pozostałych).
  - Brak obsługi zagnieżdżonych kolekcji (np. Set< Set<Integer> >).
  - Każdy obiekt musi mieć zdefiniowany atrybut/atrybuty odwzorowywane na klucz główny.
- Brak efektywnego języka zapytań – języki zapytań stosowane przez narzędzia ORM mają dużo mniejsze możliwości wyrazu niż język SQL i trudniej optymalizować w nich zapytania. Zapytania w natywnym języku narzędzia odwzorowującego jest tłumaczone na SQL. Wymaganie translacji do SQL determinuje formę obiektowych języków zapytań – języki te są niespójne i mają wiele nielogicznych ograniczeń. Język HQL używany przez popularne środowisko Hibernate obarczone jest kilkoma powaznymi błędami.
- Mniejsza efektywność niż w przypadku bezpośredniego wykonywania zapytań SQL – korzystanie z narzędzi ORM ułatwia projektowanie aplikacji, jednakże ogranicza naszą kontrolę nad tym jakie konkretnie zapytania do bazy zostaną wykonane w momencie pobierania bądź modyfikacji obiektów. W przypadku krytycznych fragmentów kodu może się okazać, że narzut na niepotrzebne operacje wykonywane przez narzędzia odwzorowujące jest tak duży, że opłaca się wykonywać ręczne zapytania SQL bądź wywołać procedurę składowaną.

2.2. Narzędzia do operowania na danych XML

Dane XML mają, podobnie jak dane w bazie LoXiM, strukturalny charakter. W obu środowiskach mamy do czynienia z drzewem nazwanych obiektów. Typy obiektów są podobne – w LoXiM prosty, złożony, referencja; w XML między innymi element złożony (*Element*), tekst (*Text*, *CData*). W typowych zastosowaniach dane w formacie XML dostarczone są w formie pliku tekstowego. Zwykle używamy XML do wymiany danych, nie zaś do ich przechowywania i swobodnej manipulacji. Wiążą się z tym pewne ograniczenia narzędzi operujących na XML: ewentualne modyfikowanie danych sprowadza się do odtworzenia struktury pliku wejściowego w pamięci, zmodyfikowania tej struktury i utworzenia pliku ponownie na podstawie danych z pamięci. Istnieją również rozwiązania będące czymś pomiędzy źródłem danych XML a bazami danych – repozytoria XML. Nie zdobyły one szerszej popularności.

Wraz ze standardem XML zdefiniowane zostały dwa modele pracy z danymi – DOM (Document Object Model), pozwalający na swobodne przeglądanie i modyfikację danych XML, oraz SAX (Simple API for XML) – model oparty o zdarzenia, pozwalający na sekwencyjne przetwarzanie danych.

Istnieje wiele innych narzędzi do pracy z XML – języków zapytań (XPath, XQuery), narzędzi do przetwarzania plików (XSLT) oraz interfejsów programistycznych. Warto przyjrzeć się istniejącym rozwiązaniom pod kątem ewentualnych mechanizmów, które możnaby wykorzystać. Z uwagi na podobieństwo zastosowań do narzędzia odwzorowywującego opiszę dwa rozwiązania – DOM oraz JAXB.

2.2.1. DOM

DOM (Document Object Model) jest standardowym interfejsem programistycznym pozwalającym na przeglądanie danych w formacie XML i ich modyfikację. Uznany za standard przemysłowy, doczekał się implementacji dla praktycznie każdej popularnej platformy tworzenia oprogramowania – od języka C do JavaScriptu. Co ciekawe, interfejsy dla poszczególnych platform wyglądają dokładnie tak samo, nawet z dokładnością do nazw metod i typów danych.

W interfejsie DOM dane strukturalne reprezentowane są przez drzewo węzłów. Możemy nawigować między węzłami, a także tworzyć nowe węzły oraz dodawać i usuwać dzieci istniejących węzłów.

O ile sama idea DOM sprawdza się doskonale, o tyle pewne szczegóły implementacyjne (chociażby brak kompatybilności z podsystemem kolekcji w Javie) nie są wygodne w praktyce. Na bazie DOM powstały więc bardziej funkcjonalne interfejsy takie jak JDOM.

2.2.2. JAXB

Przyglądając się dokumentowi XML, jasno widzimy, że jego węzły możemy potraktować jako obiekty, natomiast dzieci tych węzłów jako atrybuty obiektów. W dużej części przypadków potrzebujemy po prostu przepisać dane z pliku XML do odpowiadającej mu struktury obiektów. Kod realizujący tę funkcjonalność będzie bardzo powtarzalny i w przypadku skomplikowanych danych dosyć obszerny. W celu wyeliminowania tej żmudnej czynności twórcy Javy zaproponowali standard JAXB (Java API for XML Binding), który definiuje narzędzie pozwalające na powiązanie danych XMLowych z klasami Javy. Implementacje JAXB pozwalają na wygenerowanie hierarchii klas w oparciu o schemat dokumentów XML (DTD, XML Schema). Następnie możliwe jest wczytanie dokumentu XML do pamięci w postaci obiektów wygenerowanych wcześniej klas. Powstałe w ten sposób obiekty możemy modyfikować a następnie zapisać zmiany do dokumentu XML. Istotny jest tu fakt, że nie musimy znać hierarchii dokumentu XML. Nie musimy nawet wiedzieć, że pracujemy na dokumencie XML, ponieważ otrzymujemy zwykłe obiekty Javy. Taka sytuacja jest bardzo wygodna z punktu widzenia programisty, który nie musi przejmować się szczegółami dostępu do danych i może się skupić na rozwiązywanym problemie.

Poważną wadą mechanizmu JAXB jest ścisłe związanie schematu danych z kodem programu. Jakakolwiek zmiana w schemacie XML pociąga za sobą konieczność zmian w kodzie i potrzebę ponownej kompilacji programu.

2.3. Wnioski

Jak widać, powstało wiele ciekawych narzędzi pozwalających na połączenie koncepcji programowania obiektowego z relacyjnymi bazami danych. Są one z powodzeniem wykorzystywane przy tworzeniu systemów informatycznych różnych zastosowań i rozmiarów. Co prawda, aplikacje napisane z wykorzystaniem narzędzi ORM są często mniej wydajne niż analogiczne aplikacje bezpośrednio odwołujące się do baz danych. Mogą natomiast być bardziej niezawodne, łatwe w rozwoju i testowaniu.

Dla narzędzi do obróbki XML – DOM i JAXB specyficzny jest fakt, że wszelkie operacje odbywają się na zlokalizowanej w pamięci kopii danych – dyskwalifikuje to te narzędzia do operowania na bardzo dużych zbiorach danych. Co prawda mechanizm JAXB na pierwszy rzut oka wygląda na bardzo podobny do tworzonego narzędzia odwzorowującego do LoXiM, jednakże po bliższym przyjrzeniu widać pewne ograniczenia – oprócz koncepcji pracy na kopii danych przechowywanej w pamięci problematyczna jest konieczność rekompilacji programu w związku ze zmianą modelu danych.

Podsumowując, wydaje się, że mechanizm odwzorowania danych semistrukturalnych na obiektowe będzie dużo prostszy i bardziej czytelny niż mechanizm odwzorowania obiektowo-relacyjnego. Wynika to z faktu, że obiekty bazy LoXiM mają strukturę podobną do obiektów z obiektowych języków programowania. Możemy łatwo odwzorowywać obiekty LoXiM na obiekty Javy. Nie mamy też problemu z wykonywaniem zapytań. W przeciwieństwie do baz relacyjnych wynikiem zapytania w języku SBQL jest po prostu zbiór obiektów.

Z drugiej strony należy zwrócić uwagę, że XML to również dane semistrukturalne: możemy zatem oprzeć koncepcję dostępu do danych z bazy LoXiM na sprawdzonych w praktyce standardach (np. DOM) i, być może, tylko zmodyfikować pomysł dla naszych potrzeb. W realizowanym projekcie do tworzenia modułu nawigacji została z tego powodu zastosowana koncepcja bardzo podobna do DOM.

Rozdział 3

Odwzorowanie obiektów Javy na obiekty LoXiM

3.1. Podobieństwa i różnice

Chcąc stworzyć efektywny mechanizm wzajemnego odwzorowania obiektów Javy na obiekty LoXiM musimy przyjrzeć się definicji obiektu w obu środowiskach, znaleźć wszelkie podobieństwa i różnice, a następnie określić w jaki sposób cechy obiektów w jednym środowisku mają być odwzorowane w drugim. Należy pamiętać, że każda różnica semantyki obiektów niemożliwa do pogodzenia między Javą a LoXiM powoduje ograniczenie funkcjonalności narzędzia odwzorowującego.

Przyjrzyjmy się wspólnym cechom obu środowisk:

Pojęcie obiektu Zarówno w Javie jak i w bazie LoXiM poprzez obiekt rozumiemy byt, który posiada identyfikator oraz określoną wartość. W obu środowiskach możemy zdefiniować dwa rodzaje obiektów – obiekty proste i obiekty złożone.

Typy proste/typy złożone W środowisku Javy jak i w bazie LoXiM możemy stosować obiekty typów prostych takie jak liczba, wartość logiczna czy znakowa. W obu tych środowiskach semantyka obiektów prostych jest podobna – możemy dokonywać na nich operacji arytmetycznych i logicznych, a dwa różne obiekty atomowe o tej samej wartości są sobie równe. Obiekt typu złożonego posiada nazwane atrybuty oraz metody. W przypadku języka Java jako typ złożony rozumiemy wszystkie obiekty nie będące prymitywami. Baza LoXiM posiada typ obiektu złożonego, którego wartością jest zbiór innych obiektów. O ile w przypadku bazy LoXiM podział na obiekty proste/złożone jest częścią struktury środowiska, o tyle w Javie podział na obiekty proste (klas Boolean, Integer, String) oraz złożone (klasy stworzone przez użytkownika, zgodne ze specyfikacją JavaBeans) jest arbitralny – określony pod kątem narzędzia odwzorowującego.

Referencje W obu środowiskach możemy operować referencją do obiektu. Jeśli ją znamy, możemy zarówno dokonywać operacji na obiekcie, jak i użyć go jako argumentu operacji.

Z naszego punktu widzenia istotne są następujące kwestie w różnicy obiektowości bazy LoXiM i Javy :

Hierarchia obiektów W Javie nie istnieje żadna hierarchia obiektów: dostęp do poszczególnych obiektów odbywa się poprzez referencję. Z punktu widzenia programisty obiekt, do którego nie ma dostępu poprzez ścieżkę referencji, nie istnieje. Obiekt w Javie nie

posiada nazwy – ma ją co najwyżej referencja do obiektu. Tymczasem baza LoXiM przechowuje obiekty w drzewie, a każdy obiekt ma określoną nazwę.

Schemat Język Java wymaga ścisłego zdefiniowania modelu obiektów, których możemy używać (klasy). Co prawda baza danych LoXiM może zawierać metadane zawierające strukturę bazy, jednakże informacje te są opcjonalne, używane tylko w celach optymalizacyjnych i nie ograniczają struktury obiektów przechowywanych w bazie danych.

Usuwanie obiektów a referencje W bazie danych możliwe jest usuwanie obiektów, wówczas wszystkie referencje do usuwanego obiektu też są usuwane. W Javie w ogóle nie istnieje operacja usunięcia obiektu – obiekt istnieje dopóki prowadzi do niego jakaś referencja.

Obiekt składowy Java dysponuje tylko jednym rodzajem relacji między obiektami – atrybutem obiektu może być referencja do innego obiektu. W przypadku LoXiM możliwe jest również tworzenie obiektów składowych, co wpływa na semantykę niektórych operacji. Na przykład usunięcie obiektu powoduje usunięcie wszystkich jego obiektów składowych – w Javie podobny efekt nie występuje, co może sprawiać kłopoty w odwzorowaniu.

Kolekcje Język Java posiada kilka zdefiniowanych rodzajów kolekcji: listę, zbiór, mapę. W bazie LoXiM nie występuje pojęcie kolekcji, natomiast obiekt może posiadać dowolną, zmienną liczbę podobiektów. Dzięki temu implementacja różnego rodzaju kolekcji nie powinna stanowić problemu. Pojawia się jednak pytanie, jak odwzorować poszczególne rodzaje kolekcji Javy w bazie danych.

Metody Obiekt w bazie danych LoXiM, w przeciwieństwie do obiektu Javy nie posiada metod, a jedynie atrybuty. Z punktu widzenia narzędzia odwzorowującego oznacza to tyle, że ewentualne metody obiektu nie będą miały reprezentacji w bazie danych, a ich ewentualne wykonanie może następować po stronie aplikacji Javy.

Wartość zerowa W bazie LoXiM nie istnieje pojęcie wartości zerowej. W Javie mamy do dyspozycji specjalną wartość *null*. Z uwagi na większą elastyczność struktur LoXiM i brak modelu możemy odwzorować wartość zerową po prostu jako brak obiektu.

3.2. Identyfikacja obiektów

Potrzebujemy mechanizmu jednoznacznej identyfikacji każdego obiektu w bazie danych. Stanowi to pewien problem, ponieważ za pomocą języka zapytań SBQL nie możemy pobrać identyfikatora obiektu, ani obiektu o określonym identyfikatorze. Dysponujemy jedynie operatorem == pozwalającym na sprawdzenie identyczności obiektów w ramach pojedynczego zapytania. Mimo wszystko sytuacja jest lepsza niż w przypadku relacyjnych baz danych, gdzie nie jesteśmy w stanie rozróżnić dwóch identycznych encji w danej relacji.

W przypadku baz relacyjnych, aby umożliwić jednoznaczną identyfikację encji, możemy dla każdej tabeli zdefiniować klucz główny – określić zbiór atrybutów, które będą jednoznacznie identyfikować każdą encję. Opisane wcześniej narzędzia do odwzorowania obiektowo-relacyjnego bazują istnieniu klucza głównego. To na jego podstawie są w stanie zidentyfikować konkretną encję w bazie danych, która jest reprezentowana przez pewien obiekt pośredniczący. W praktyce bez jawnego określenia kluczy głównych nie jesteśmy w stanie efektywnie korzystać z relacyjnej bazy danych. Przyjrzyjmy się bazie semistrukturalnej. Widzimy, że zamiast składać relacje posługując się kluczami tak jak w bazach relacyjnych, możemy po prostu

jawnie nawigować po wskaźnikach i pobierać obiekty pochodne od obiektu bazowego, zatem jawne definiowanie kluczy nie jest konieczne. Z punktu widzenia osoby piszącej pojedyncze zapytanie w języku SBQL mamy co najmniej taką samą siłę wyrazu jak języku SQL. Problem pojawia się gdy chcemy śledzić stan konkretnych obiektów w bazie danych w sytuacji, kiedy obiekty te mogą się zmieniać w czasie. Rozważmy dwa rozwiązania tego problemu:

- Możemy zdefiniować identyfikatory obiektów jako jawne atrybuty, na tej samej zasadzie jak w przypadku baz relacyjnych. Po pierwsze jest to rozwiązanie nadmiarowe – wiemy, że każdy obiekt w bazie ma już unikalny identyfikator. Po drugie, skomplikuje się proces dodawania danych – potrzebne będą mechanizmy przyznawania i sprawdzania unikalności identyfikatorów, twórca aplikacji korzystającej z bazy danych musi wiedzieć jakie atrybuty zawierają identyfikator. Problem pojawia się też gdy obserwowany obiekt zmieni lokalizację w drzewie obiektów, wówczas go „zgubimy": na podstawie samej wartości identyfikatora nie będziemy go w stanie zlokalizować

- Gdyby język zapytań bazy danych pozwolił nam na pobranie identyfikatora obiektu, oraz późniejsze pobranie obiektu o określonym identyfikatorze, wówczas na tej podstawie bylimyśmy w stanie śledzić dowolne obiekty niezależnie od zmian jakie w nich zaszły i niezależnie od ich lokalizacji. Problemy jakie pojawiają się w tym rozwiązaniu to przede wszystkim konieczność dodania pewnych rozszerzeń do języka SBQL, a dodatkowo konieczność zapewnienia przez system zarządzania bazą danych, że identyfikatory obiektów będą unikalne w ramach bazy, oraz problem zapewnienia niezmienności identyfikatorów obiektów przez cały czas w jakim chcemy śledzić ich stan.

3.3. Szczegółowa propozycja odwzorowania obiektów Java - LoXiM

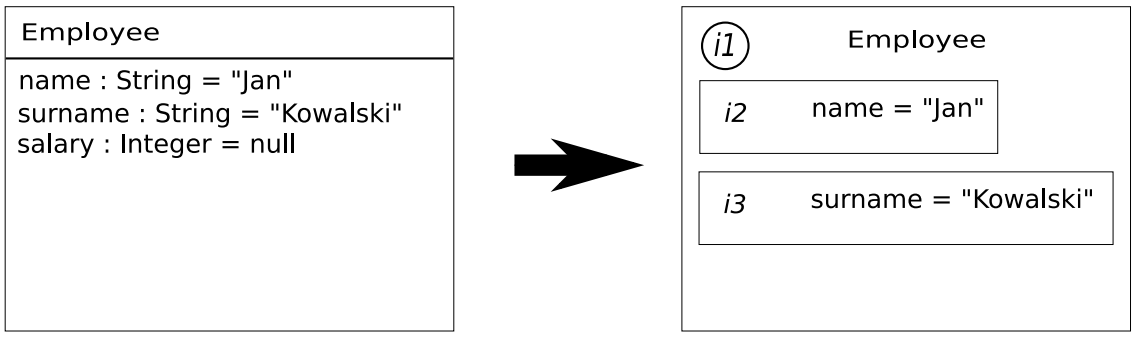
Poniżej przedstawiłem propozycję odwzorowania poszczególnych struktur języka Java w bazie danych LoXiM. Z uwagi na to, że pewne elementy języka można odwzorować na kilka sposobów posiadających swoje zalety i wady, będziemy musieli umożliwić użytkownikowi narzędzia odwzorowującego wybór właściwego rozwiązania.

3.3.1. Obiekt Javy z prostymi atrybutami

Obiekt Javy posiadające proste atrybuty możemy odwzorować jako złożony obiekt LoXiM posiadający proste podobiekty reprezentujące poszczególne atrybuty. Nazwy obiektu złożonego jak i obiektów reprezentujących atrybuty mogą zostać zdefiniowane przez użytkownika, natomiast domyślnie nazwa obiektu złożonego będzie taka jak nazwa klasy obiektu Javy, natomiast nazwy obiektów reprezentujących atrybuty będą takie same jak nazwy reprezentowanych atrybutów. Przykład odwzorowania obiektu prostego znajduje się na rysunku 3.1.

3.3.2. Obiekt Javy z innym obiektem jako atrybutem

Odwzorowanie atrybutów obiektu typu innego niż prosty (Integer, String, Boolean) możliwe jest tylko wtedy, gdy typ atrybutu również jest odwzorowywalny w bazie LoXiM. W takiej sytuacji wartość atrybutu zostanie odwzorowana na dodatkowy obiekt w bazie danych. Powiązanie obiektu reprezentującego atrybut z obiektem nadrzędnym możemy zrealizować na dwa sposoby:



Rysunek 3.1: Odwzorowanie obiektu Javy z prostymi atrybutami.

Poprzez referencję – W tej sytuacji obiekt będący wartością atrybutu musi byś obiektem trwałym – znajdować się już gdzieś w bazie danych. Wartość atrybutu będzie miała postać składowej referencyjnej obiektu złożonego (rysunek 3.2).

Poprzez obiekt składowy – W tej sytuacji wartość atrybutu zostanie odwzorowana jako podobiekt złożony (rysunek 3.3).

3.3.3. Zbiór

Struktura bazy danych stosowana przez LoXiM nie objemuje pojęcia kolekcji, natomiast obiekt w bazie danych może zawierać dowolną ilość podobiektów o tej samej nazwie. Stąd możliwe jest odwzorowanie atrybutu typu zbiór (*Set*) na podobiekty o takiej nazwie jak nazwa atrybutu. Wówczas dodanie nowego elementu do zbioru będzie polegać na dodaniu nowego podobiektu, a pobranie wszystkich elementów zbioru – na pobraniu wszystkich podobiektów obiektu o danej nazwie. Z punktu widzenia odwzorowania możemy wyróżnić trzy rodzaje kolekcji w języku Java.

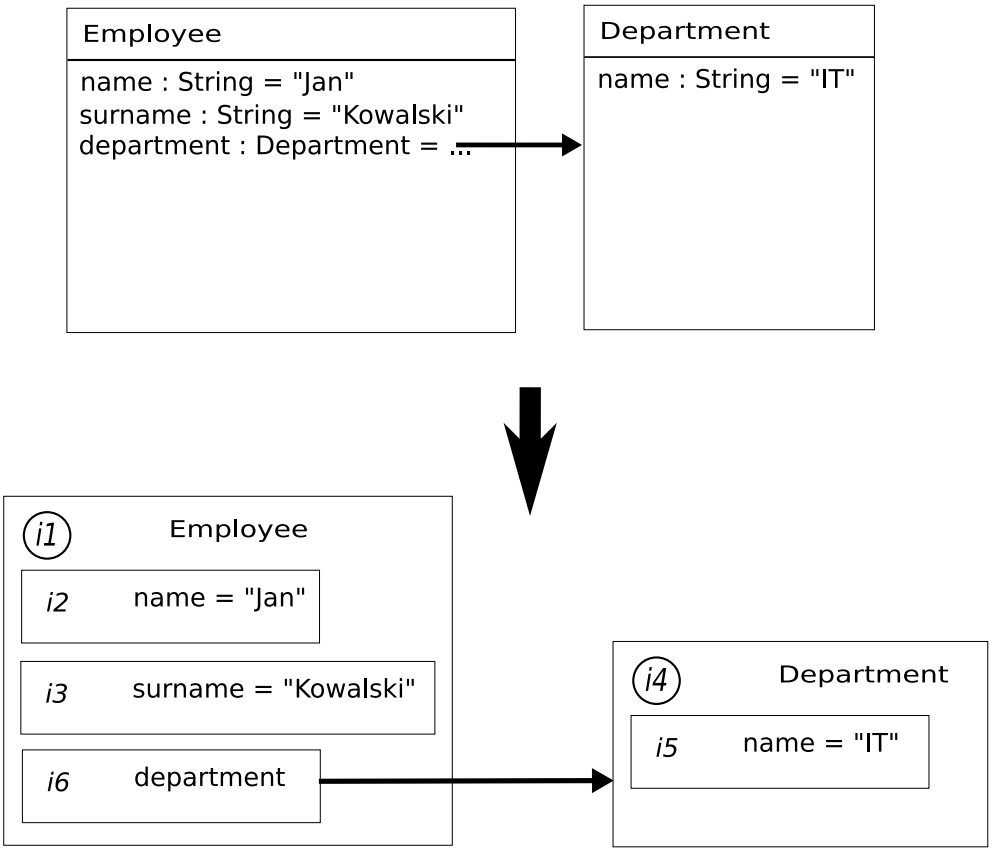
Zbiór referencji do obiektów – Atrybut zostanie odwzorowany na zbiór podobiektów typu referencyjnego – analogicznie do atrybutów typu referencyjnego.

Zbiór obiektów złożonych – Atrybut zostanie odwzorowany na zbiór podobiektów typu złożonego – analogicznie do atrybutów odwzorowanych poprzez obiekt składowy.

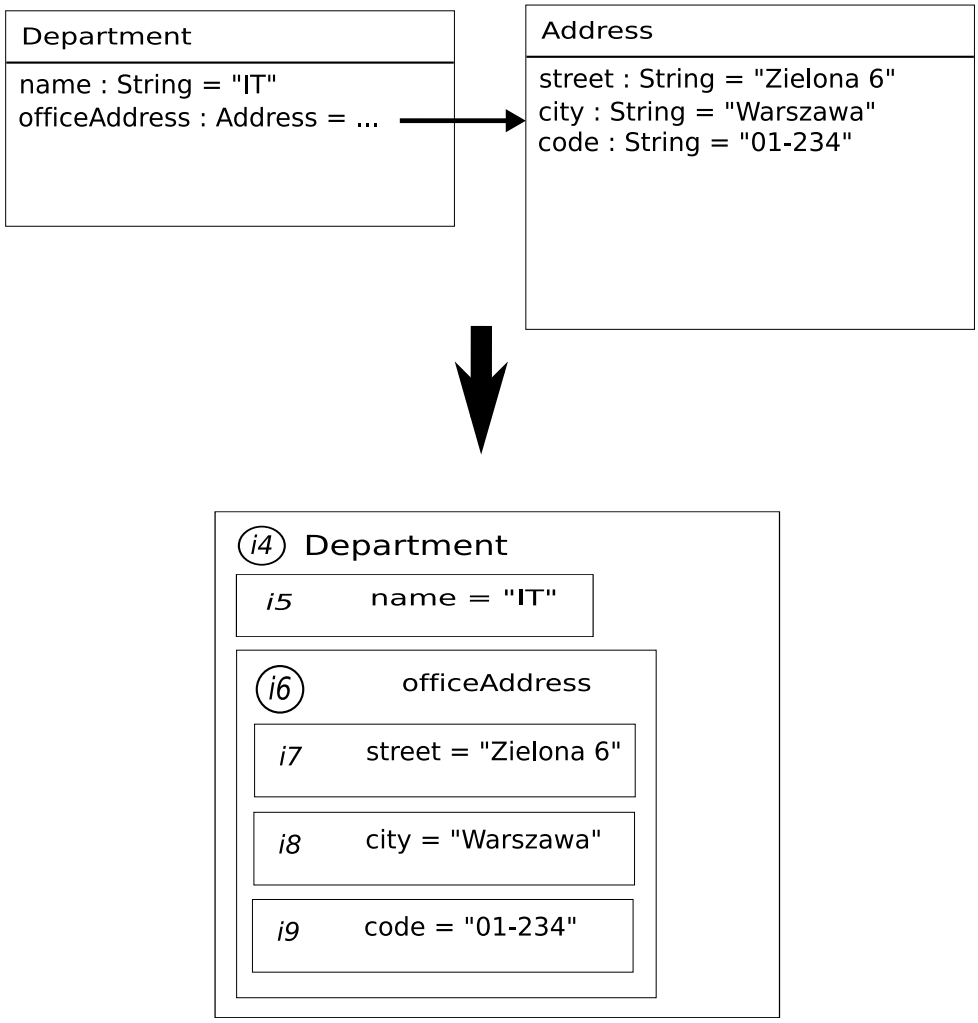
Zbiór obiektów prostych – Atrybut zostanie odwzorowany na zbiór podobiektów typu prostego (INTEGER, BOOLEAN, STRING) – analogicznie do atrybutów odwzorowanych poprzez obiekt składowy (przykład na rysunku 3.4).

3.3.4. Wartość zerowa

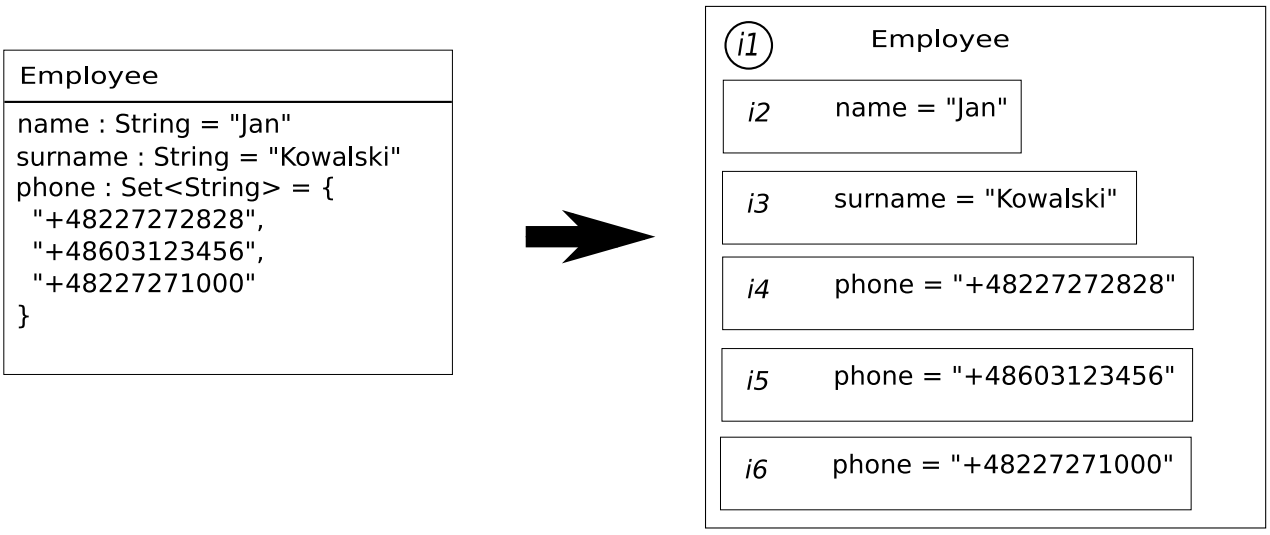
W języku Java występuje specjalną wartość zerowa *null* – atrybut obiektu może oczywiście przyjąć taką wartość. W bazie LoXiM wartość zerowa nie występuje. Z uwagi na to, że w LoXiM nie mamy żadnych ograniczeń modelu, zatem reprezentacja obiektu z atrybutem o wartości *null* może po prostu nie zawierać obiektu podrzędnego reprezentującego ten atrybut. W drugą stronę – brak obiektu reprezentującego atrybut będzie równoznaczny z wartością *null* atrybutu.



Rysunek 3.2: Odwzorowanie obiektu Javy ze złożonym atrybutem odwzorowanym poprzez referencję.



Rysunek 3.3: Odwzorowanie obiektu Javy ze złożonym atrybutem odwzorowanym poprzez obiekt składowy.



Rysunek 3.4: Odwzorowanie obiektu Javy ze atrybutem typu zbiór wartości prostych.

3.4. Semantyka operacji na obiektach

3.4.1. Utrwalenie obiektu nie posiadającego zależności do innych obiektów

W przypadku gdy utrwalamy obiekt, który posiada tylko atrybuty proste, nieutrwalone obiekty składowe bądź referencje do już utrwalonych obiektów, operacja zapisu jest jednoznaczna. W tej sytuacji po prostu tworzymy semistrukturalne odwzorowanie utrwalanego drzewa obiektów i wykonujemy zapytanie zapisujące drzewo do bazy.

3.4.2. Dodanie podobiektu do obiektu

Sytuacja jest analogiczna do powyższej. Z tą różnicą, że po utworzeniu podobiektu przenosimy go z korzenia do obiektu nadrzędnego.

3.4.3. Dodanie utrwalonego podobiektu do obiektu

Użytkownik podstawia utrwalony obiekt A jako atrybut podobiektowy do utrwalonego obiektu B. W tej sytuacji pojawia się konflikt. Obiekt A nie może jednocześnie istnieć w dwóch lokalizacjach – dotychczasowej i jako składowej obiektu B. Możliwe są dwa rozwiązania powyższego problemu:

- Usunięcie obiektu A z dotychczasowej lokalizacji.
- Skopiowanie wartości obiektu A.
- Zgłoszenie wyjątku

Z uwagi na to iż żadne z powyższych rozwiązań nie jest doskonałe w każdej sytuacji, powinniśmy pozwolić użytkownikowi na wybór właściwej semantyki.

3.4.4. Utrwalenie obiektu posiadającego referencje do nieutrwalonych obiektów

W momencie natrafienie na referencję do nieutrwalonego obiektu możemy postąpić na dwa sposoby:

- Zgłosić wyjątek
- Kaskadowo utrwalić wskazywany obiekt

Rozwiązanie z wyjątkiem jest oczywiście najprostsze do zaimplementowania, jednak kaskadowe utrwalanie może znacznie uprościć pisanie kodu aplikacji przez użytkownika narzędzia odwzorowującego. Utrwalanie kaskadowe wiąże się z pewnymi problemami, a mianowicie może zdarzyć się, że wskazywany obiekt zawiera referencje do kolejnych nieutrwalonych obiektów. Zadecydowałem że narzędzie odwzorowujące będzie stosowało pierwsze rozwiązanie - zgłaszanie wyjątku.

3.4.5. Zmiana wartości atrybutu prostego

Aby zmienić wartość atrybutu prostego wystarczy tylko zmienić wartość obiektu w bazie reprezentującego dany atrybut, a w przypadku, gdy obiekt ten nie istniał (z punktu widzenia Javy wartość *null*, stworzyć nowy obiekt. Jeśli nowa wartość atrybutu to *null*), wówczas usuwamy obiekt reprezentujący starą wartość.

3.4.6. Zmiana wartości atrybutu będącego obiektem składowym

W przypadku, gdy ustalamy nową wartość atrybutu będącego obiektem składowym, musimy rozważyć dwa przypadki:

- Nowa wartość to obiekt lokalny Javy – w takiej sytuacji możemy go po prostu utrwalić w bazie danych jako podobiekt obiektu nadrzędnego
- Nowa wartość to obiekt trwały, istniejący już gdzieś w hierarchii obiektów – w tym momencie proste przeniesienie tego obiektu w inne miejsce hierarchii może spowodować nieoczekiwany efekt: obiekt przestanie istnieć w dotychczasowym miejscu.

  Istnieją trzy możliwe rozwiązania tego problemu:

  – Zabronić operacji podstawienia obiektu trwałego jako atrybutu odwzorowanego jako obiekt składowy
  – Przenieść obiekt składowy w nowe miejsce w drzewie
  – Wykonać kopię obiektu składowego

  Narzędzie odwzorowujące obsługuje wszystkie trzy rozwiązania – wybór właściwego w danej sytuacji należy do użytkownika.

Jeśli modyfikujemy atrybut odwzorowany na obiekt składowy, który posiada już jakąś wartość, wówczas pojawia się problem co zrobić z ową wartością. Tu możemy:

- Usunąć obiekt składowy reprezentujący tą wartość
- Przenieść obiekt składowy reprezentujący wartość do korzenia

Podobnie jak wcześniej – narzędzie odwzorowujące będzie pozwalało na wybór rozwiązania przez użytkownika.

3.4.7. Zmiana wartości atrybutu będącego referencją

Rozwiązanie jest analogiczne do zmiany wartości atrybutu prostego. W przypadku gdy podstawiamy nową wartość atrybutu – do obiektu w bazie LoXiM będącego reprezentantem danego obiektu Javy dodawany jest podobiekt typu referencyjnego zawierający referencję do nowej wartości atrybutu. Wymagamy aby nową wartością był obiekt uprzednio utrwalony w bazie.

Rozdział 4

Architektura rozwiązania

4.1. Wstęp

Narzędzie składa się z trzech głównych modułów – połączenia z bazą danych, przeglądania bazy i przezroczystego odwzorowania. Oprócz tego z warstwą przezroczystego odwzorowania związanych jest kilka komponentów pomocniczych:

- Komponent odpowiedzialny za tworzenie obiektów trwałych i zarządzający cyklem ich życia (*TransparentProxyFactory*)
- Komponent odpowiedzialny za zdefiniowanie i przechowywanie modelu danych (*ModelRegistry*)
- Komponent odpowiedzialny za obsługę poszczególnych typów danych (*AccessorRegistry*)

Podczas tworzenia narzędzia odwzorowującego w miarę możliwości wykorzystywałem wzorzec projektowy *Wstrzykiwanie zależności*[8]. Pozwoliło to na czytelne odseparowanie od siebie komponentów realizujących poszczególne zadania i powinno ułatwić ewentualny rozwój narzędzia. Uruchomienie odwzorowującego sprowadza się do stworzenia obiektów realizujących poszczególne zadania, a następnie skonfigurowaniu ich i połączeniu ze sobą.

4.2. Moduł połączenia z bazą danych

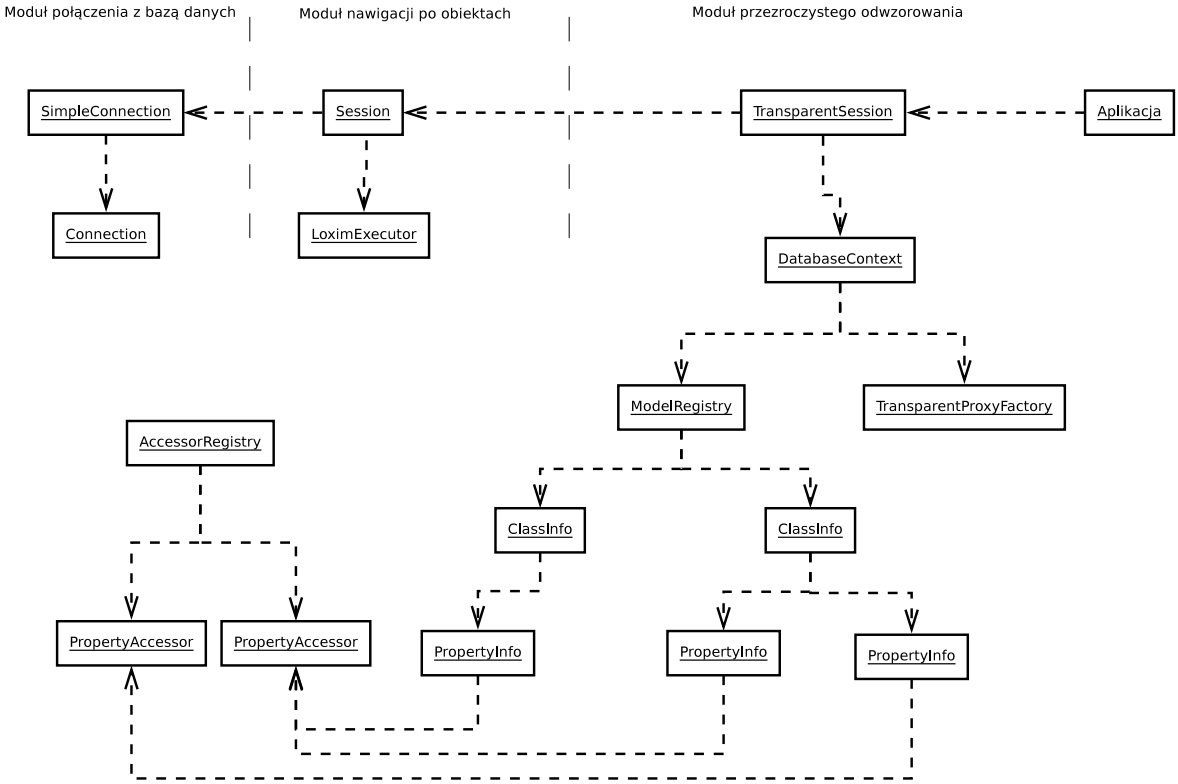
4.2.1. Zadanie

Warstwa ta będzie odpowiedzialna za połączenie z serwerem bazy danych i wykonywanie zapytań.

4.2.2. Implementacja

W chwili pisania tej pracy protokół komunikacyjny w systemie LoXiM był w trakcie sporych zmian i najprawdopodobniej docelowo do połączenia z bazą zostanie wykorzystany nowy sterownik stworzony przez Piotra Tabora [11].

Podstawowym celem przy projektowaniu interfejsu programistycznego do łączenia z bazą danych było maksymalne uproszczenie operacji na bazie, tak aby kod wykonujący zapytanie był jak najbardziej zwięzły.



Rysunek 4.1: Architektura narzędzia odwzorowującego.

4.3. Moduł nawigacji po obiektach

4.3.1. Zadanie

Warstwa ta ma być odpowiedzialna za pobieranie obiektów z bazy. Dane z bazy będą reprezentowane w Javie poprzez specjalne obiekty wskaźnikowe. Warstwa ma też pozwalać na modyfikowanie zawartości bazy poprzez wykonywanie operacji na obiektach wskaźnikowych.

4.3.2. Reprezentacja danych w Javie

Podstawowym bytem na jakim będzie operować warstwa nawigacji jest obiekt LoXiM znajdujący się w bazie danych. Dla uproszczenia będziemy go dalej nazywać węzłem. Węzły reprezentowane są przez obiekty klasy LoximNode, które stanowią wskaźnik do węzła znajdującego się w bazie danych lub wartość nowego węzła, który chcemy zapisać w bazie danych. Obiekt LoximNode może znajdować się w dwóch stanach:

Podłączonym Obiekt klasy LoximNode jest wskaźnikiem do obiektu znajdującego się w bazie danych i przechowuje tylko nazwę tego obiektu oraz jego identyfikator. Każda operacja odczytu bądź modyfikacji wartości obiektu wiąże się z wykonaniem operacji na bazie danych

Odłączonym Obiekt reprezentuje węzeł, który nie znajduje się w bazie danych, natomiast przechowuje informację o swojej własnej wartości. Taki obiekt możemy podłączyć do bazy danych, czyli zapisać go w bazie. Wówczas otrzyma identyfikator i wszelkie dalsze operacje na obiekcie będą wiązały się wykonaniem operacji na bazie danych

4.3.3. Przeglądanie węzłów

Podczas próby pobrania wartości węzła (obiektu LoximNode) wywoływane jest zapytanie, które pobiera wartość obiektu z bazy danych o referencji wskazywanej przez LoximNode:

```
deref($nodeRef)
```

Na podstawie wyniku zapytania tworzony jest obiekt typu ObjectValue.

4.3.4. Tworzenie węzłów

W warstwie przeglądania obiektów potrzebujemy funkcjonalności tworzenia nowych obiektów bazodanowych (w postaci struktury obiektów Javy) i zapisywania ich w bazie danych.

W tym celu dodana została możliwość tworzenia obiektów klasy LoximNode zawierających pewne wartości aczkolwiek nie związanych jeszcze z bazą danych. Obiekty klasy LoximNode nie podłączone do bazy danych nazywać będziemy odłączonymi węzłami.

Odłączone węzły zawierają po prostu wartość obiektu, który chcemy zapisać, w przypadku gdy jest to obiekt prymitywny będzie to jego wartość atomowa, w przypadku gdy jest to obiekt złożony będziemy mieli dostęp do listy dzieci. Przy tworzeniu odłączonego węzła mamy całkowitą swobodę modyfikacji nazwy, wartości oraz dzieci obiektu.

Po stworzeniu obiektu oczywiście chcielibyśmy zapisać go w bazie. Do tego celu służy metoda *attachObject* w klasie *LoximExcecutor*, która zapisuje do bazy odłączony węzeł utrwalając go rekurencyjnie wraz z obiektami składowymi. Tworzenie obiektów zawierających podobiekty przebiega zgodnie z poniższym algorytmem:

- tworzymy obiekt złożony nie zawierający dzieci, poleceniem

```
create $arg0
```

gdzie arg0 to pusta struktura[1].

- tworzymy rekurencyjnie wszystkie dzieci obiektu – początkowo w korzeniu
- przenosimy utworzone dzieci do utworzonego wcześniej obiektu złożonego, poleceniem

```
$parent <: $child
```

Ta metoda utrwalania obiektów może powodować duży narzut z racji konieczności wykonania co najmniej jednego zapytania dla każdego utrwalanego węzła. Język SBQL pozwala co prawda utworzyć hierarchię obiektów za pomocą jednego zapytania, przykładowo:

```
create  (
        "Jan" as name,
        "Kowalski" as surname,
        4500 as salary,
        ref(Department where name="IT") as department
) as Employee
```

Zapytanie to tworzy obiekt złożony *Part* zawierający cztery atrybuty proste i referencję do już istniejącego obiektu *Department*. To podejście ma oczywiście wiele zalet, natomiast z naszego punktu widzenia ma dwie wady: po pierwsze długość pojedynczego zapytania jest liniowo zależna od ilości utrwalanych obiektów – stąd przy utrwalaniu dużych struktur możemy przekroczyć ograniczenie na maksymalną długość zapytania. Z drugiej strony – tworząc wiele obiektów w jednym zapytaniu tracimy informację o referencjach do utworzonych podobiektów, a ta informacja jest nam potrzebna.

4.3.5. Sesje

Praca z warstwą nawigacji odbywa się w ramach pojedynczej transakcji. Sesję pracy w ramach transakcji reprezentuje obiekt klasy *Session*. Obiekt ten zawiera między innymi informację o obiektach pobranych z bazy w ramach sesji. W momencie, gdy użytkownik będzie chciał ponownie pobrać ten sam obiekt z bazy – otrzyma utworzony wcześniej wskaźnik do niego. Od momentu, gdy sesja zostanie zamknięta – transakcja zakończona, połączenie z bazą zwrócone do obiektu *datasource*, wszelkie próby operacji na obiektach pobranych w ramach tej sesji zgłoszą wyjątek (*SessionClosedException*). Powiązanie pobranych obiektów z sesją pozwala na ewentualne rozszerzenie narzędzia odwzorowującego o buforowanie.

4.4. Moduł przezroczystego odwzorowania

4.4.1. Zadanie

Warstwa ta ma umożliwiać użytkownikowi utrwalanie w bazie danych zwykłych obiektów Javy (POJO). Użytkownik będzie mógł stworzyć dowolny obiekt Javy przy użyciu zwykłego konstruktora, następnie za pomocą odpowieniej operacji będzie mógł utrwalić ten obiekt – od

[1] Używana wersja bazy LoXiM nie pozwala na stworzenie pustego obiektu złożonego, stąd narzędzie odwzorowujące tworzy obiekty złożone zawierające obiekt prosty *bugfix(1)*.

tego momentu obiekt wraz ze wszystkimi jego atrybutami zostanie zapisany w bazie danych, a wszelkie zmiany atrybutów obiektu będą pociągały za sobą zmiany w bazie. Z powodu specyfiki języka Java konieczne będzie narzucenie pewnych niewielkich ograniczeń na kształt utrwalanych obiektów. Dostęp do atrybutów obiektów musi być zrealizowany zgodnie ze standardem JavaBeans[5].

Przed użyciem narzędzia konieczne będzie określenie sposobu odwzorowania konkretnych klas na struktury bazodanowe. Do tego celu zdefiniowane zostaną specjalne adnotacje w języku Java, które będą pozwalały na oznaczenie klasy jako poddającej się utrwaleniu oraz określające sposób zapisu obiektów tej klasy w bazie.

4.4.2. Tworzenie dynamicznych obiektów pośredniczących w Javie

Podczas dodawania cechy trwałości do istniejącego obiektu natrafiamy na problem w jaki sposób w trakcie wykonania programu zmienić semantykę tego obiektu. Dopóki obiekt był zwykłym obiektem Javy, wszystkie operacje wykonywane na nim wynikały z kodu programu. Po utrwaleniu obiektu chcielibyśmy, aby dodatkowo każda operacja odnosząca się do atrybutów obiektu powodowała zapis lub odczyt z bazy danych. Oczywiście moglibyśmy modyfikować przed kompilacją kod klas, których obiekty chcemy utrwalać. Takie rozwiązanie jest jednak niewygodne.

Na szczęście Java udostępnia kilka mechanizmów pozwalających kształtować zachowanie obiektów w trakcie wykonania programu. Pierwszy mechanizm, który może nam pomóc to technologia dynamicznych klas pośredniczących[4], która umożliwia nam dynamiczne stworzenie obiektu implementującego dowolne interfejsy poprzez określenie jakie interfejsy powinien on implementować oraz dostarczenie kodu obsługującego wywołania metod tego obiektu (w formie obiektu typu *InvocationHandler*). Kod obsługujący wywołania metod obiektu proxy może wyglądać następująco:

```
public Object invoke(Object proxy, Method m, Object[] args)
        throws Throwable {

        System.out.println(
                "Wywołano metode " + m.getName() + "z parametrami:");

        for (Object arg : args) {
                System.out.println(arg.toString());
        }
}
```

Jak widzimy, możemy w dowolny sposób kształtować działanie obiektu – w momencie pisania procedury obsługi nie musimy znać nazw metod oraz ich parametrów.

Mechanizm klas pośredniczących ma dwie zasadnicze wady. Po pierwsze, co z resztą można wywnioskować z powyższego kodu, jest on mało wydajny. Przy każdym wywołaniu metody obiektu proxy musimy dokonać wywołania metody *invoke* w *InvocationHandler*, kod tej metody za pomocą mechanizmów refleksji sprawdza jaka metoda została wywołana i dopiero na tej podstawie wykonuje właściwą operację. Drugi problem związany jest z faktem, że za pomocą mechanizmu *Dynamic Proxy Classes* możemy tylko tworzyć obiekty implementujące interfejsy. Przydałaby się nam możliwość tworzenia dynamicznych obiektów udających istniejące w kodzie programu klasy. W przypadku stosowania mechanizmu DPC, nie jesteśmy w stanie stworzyć obiektów pewnej klasy o zmienionym zachowaniu. Możemy co najwyżej stworzyć interfejs, który ta klasa będzie implementować i stworzyć obiekty implementujące ten

interfejs o zachowaniu zmienionym w stosunku do klasy początkowej. Gdybyśmy mieli możliwość zmiany zachowania obiektów danej klasy, bądź tworzenia obiektów klasy, natomiast o semantyce innej niż wynika z kodu klasy, wówczas zwiększylibyśmy elastyczność naszego narzędzia, a liczba kodu, który musi zostać napisany przez użytkownika zmniejszyłaby się.

Jedną z zalet architektury języka Java jest możliwość dynamicznego doładowywania nowych klas w trakcie pracy programu. Dzięki temu mechanizmowi możliwe jest zbudowanie, kodu klasy Javy w trakcie wykonania programu, po czym załadowanie go jako nowej klasy. Widać, że daje nam to duże możliwości. Powstało kilka narzędzi które pozwalają generować klasy czy to na podstawie dostarczonego kodu maszyny wirtualnej Javy, czy poprzez modyfikację wcześniej załadowanych klas, czy też interpretujących języki wyższego poziomu. Pokrótce można tu wymienić ObjectWeb ASM[2], czy CGlib[1].

Ponieważ z punktu widzenia narzędzia odwzorowującego standardowy mechanizm klas pośredniczących w języku Java okazał się wystarczający, zastosowałem właśnie jego.

4.4.3. Sesje. Zarządzanie cyklem życia obiektów

Podobnie jak w przypadku warstwy przeglądania, w warstwie przezroczystego odwzorowania wszelkie operacje wykonywane są w ramach sesji, która jest równoważna transakcji bazodanowej. Również obiekty trwałe są podpięte do sesji w której zostały pobrane lub utrwalone i nie działają po jej ukończeniu. Sesja warstwy przezroczystego odwzorowania przechowuje informacje o wszystkich obiektach trwałych które zostały w jej ramach pobrane. W przypadku gdy użytkownik po raz kolejny pobierze obiekt z bazy, nie zostanie utworzony nowy obiekt pośredniczący Javy reprezentujący ten obiekt – zapytanie zwróci poprzednio pobrany obiekt. Powyższy mechanizm może zostać rozszerzony o możliwość odłączania obiektów z sesji i późniejszego podłączenia do nowej sesji.

4.4.4. Spójność danych

W momencie gdy pobieramy odwzorowanie pewnego obiektu z bazy danych, pojawiają się dwie kopie tej samej informacji – jedna w bazie, druga w środowisku wykonania aplikacji Java. Warto przypomnieć, iż system zarządzania bazą danych LoXiM na bazie którego realizowany jest projekt, zapewnia spójność danych w ramach transakcji, zatem możemy być pewni, że stan bazy w ramach pojedynczej transakcji będzie niezmienny, o ile sami nie dokonamy modyfikacji danych.

Przyjrzyjmy się dokładniej problemowi spójności danych przy modyfikacji. Z założenia zmiana stanu obiektu Javy spowoduje natychmiastowe wykonanie zapytania uaktualniającego bazę danych. Problem spójności jest w narzędziu odwzorowującym łatwy do obsługi, ponieważ obiekty Javy reprezentujące węzły w bazie danych zawierają tylko wskaźnik do węzła. Właściwe pobranie aktualnej danej następuje dopiero w momencie wywołania akcesora (metody *getXxx*). Pozostaje tylko problem próby dostępu do atrybutów obiektu, który nie istnieje (na przykład został usunięty). W takiej sytuacji fakt nieistnienia obiektu zostanie stwierdzony podczas wykonania akcesora, po czym zgłoszony zostanie wyjątek DeletedException.

4.4.5. Zgodność danych z modelem

Model danych jest zdefiniowany tylko w Javie – nawet w przypadku gdy dane w bazie nie zgadzają się ze zdefiniowanym modelem, narzędzie odwzorowujące będzie próbowało je odwzorować zgodnie z modelem, gdyż nie jest w stanie stwierdzić ich poprawności bądź nie. Możliwe efekty uboczne takiego zachowania są następujące:

- W bazie danych nie ma obiektów (n.p. atrybutów) zdefiniowanych w modelu – narzędzie odwzorowujące uzna, że atrybut, dla którego nie ma obiektu w bazie, ma wartość *null*
- W bazie danych istnieją obiekty dla których nie zdefiniowano odwzorowania – w takiej sytuacji obiekty te zostaną zignorowane
- Występuje niezgodność typów danych zdefiniowanych w modelu z typami danych w bazie – w takiej sytuacji narzędzie odwzorowujące zgłosi wyjątek.

Pobranie obiektu trwałego może nastąpić w wyniku wykonania zapytania lub w wyniku pobrania obiektu o zadanej referencji. To użytkownik deklaruje, jakiego typu wynik zwraca zapytanie (parametr *desiredClass*). Możliwe jest, że w wyniku zapytania użytkownik pobierze obiekt niewłaściwego typu. Jedyną możliwością sprawdzenia typu obiektu jest skonfrontowanie lokalizacji obiektu w hierarchii bazy danych z definicją modelu. Niestety język zapytań SBQL nie pozwala nam na sprawdzenie lokalizacji obiektu w bazie – możliwe jest jedynie pobranie nazwy obiektu na podstawie referencji (dodatkowa operacja *nameof*).

Podczas projektowania narzędzia odwzorowującego przyjąłem założenie, że za poprawność bazy danych i zapytań na niej wykonywanych odpowiada użytkownik i narzędzie nie będzie dodatkowo weryfikować poprawności pobieranych danych i wykonywanych zapytań.

W momencie pobierania danych (wykonywania zapytania bądź próby pobrania obiektu o zadanym identyfikatorze) pobierane są tylko identyfikatory wynikowych obiektów – na podstawie samych identyfikatorów nie jesteśmy w stanie stwierdzić, jakiego rodzaju obiekt został pobrany. Problem niezgodności danych zawartych w bazie ze zdefiniowanym modelem nastąpi dopiero podczas próby odczytania bądź modyfikacji atrybutów obiektu trwałego.

4.5. Warstwa definicji modelu danych

Ponieważ odwzorowanie obiektów Javy na obiekty bazy LoXiM nie jest jednoznaczne, a i chcielibyśmy dać użytkownikowi możliwość dostosowania rozwiązania do konkretnych potrzeb, konieczne było opracowanie mechanizmu definiowania sposobu odwzorowania. Brałem pod uwagę kilka możliwości konfiguracji. Pierwsza z nich miała opierać się na zewnętrznych plikach konfiguracyjnych (w formacie XML) z definicją odwzorowań dołączonych do aplikacji. Druga, ostatecznie wybrana, bazować miała na, wprowadzonej w wersji 5 języka Java możliwości opisywania fragmentów kodu (klas, metod, atrybutów) za pomocą metadanych (annotations).

Każde z dwóch powyższych rozwiązań ma swoje wady i zalety. Konfiguracja przechowywana w zewnętrznym pliku pozwala na odizolowanie kodu obiektów od cechy trwałości – możemy zdefiniować parametry odwzorowania obiektu bez zmiany kodu źródłowego klasy tego obiektu. Z drugiej strony mechanizm oparty o adnotacje minimalizuje ilość plików jakie musimy stworzyć aby skorzystać z odwzorowania, oraz powoduje, że wszystkie informacje o modelu są w jednym miejscu – w klasach obiektów należących do modelu danych. Z dwóch możliwych rozwiązań wybrałem wariant drugi. Oczywiście nie stanowi przeszkód zaimplmentowanie własnego mechanizmu definiowania odwzorowania opierającego się na innym źródle danych – na przykład plików konfiguracyjnych.

4.5.1. Definiowanie sposobu odwzorowania obiektów za pomocą adnotacji

Mechanizm adnotacji w języku Java 5 pozwala na dodanie do kodu programu specjalnych metadanych, które później mogą zostać odczytane w momencie wykonania. Adnotacjami możemy opisywać klasy, metody i zmienne. Przykładowa metoda udekorowana adnotacjami ma następującą postać:

```
@Transactional(readOnly=true)
void getAmount();
```

Adnotacja jest określonego typu (w tym przypadku *Transactional*) i może mieć dodatkowo pewne atrybuty (powyżej atrybut *readOnly*). Lista adnotacji zaproponowanych do definiowania odwzorowania LoXiM-Java przedstawia się następująco:

Adnotacje odnoszące się do atrybutów

Adnotacje odnoszące się do poszczególnych atrybutów (aby adnotacja dotyczyła atrybutu musimy oznaczyć albo getter, albo setter danego atrybutu)

@Persistent - tą adnotacją musi być oznaczony każdy atrybut, co do którego chcemy aby był utrwalany w bazie danych. Adnotacja zawiera opcjonalny parametr *nodeName* definiujący nazwę węzła lub węzłów składowych reprezentujących wartość tego atrybutu w bazie danych. Domyślna wartość tego parametru *nodeName* to nazwa atrybutu

@Component - atrybut oznaczony tą adnotacją zostanie utrwalony w bazie danych jako obiekt składowy

@Reference - atrybut oznaczony tą adnotacją zostanie utrwalony w bazie danych jako nowy obiekt umieszczony w korzeniu do którego utworzona zostanie odpowiednia referencja (domyślny)

@ComponentSet - atrybut oznaczony tą adnotacją zostanie utrwalony w bazie danych jako zbiór węzłów złożonych. adnotacja może być stosowana tylko do atrybutów typu *java.util.Set* zawierających odwzorowywalne obiekty. Adnotacja zawiera obowiązkowy atrybut *itemType* definiujący typ obiektów przechowywanych w zbiorze – możliwe wartości to odwzorowywalne typy danych.

@ReferenceSet - atrybut oznaczony tą adnotacją zostanie utrwalony w bazie danych jako zbiór węzłów zawierających referencje. adnotacja może być stosowana tylko do atrybutów typu *java.util.Set* zawierających utrwalone obiekty. Adnotacja zawiera obowiązkowy atrybut *itemType* definiujący typ obiektów przechowywanych w zbiorze – możliwe wartości to odwzorowywalne typy danych.

@PrimitiveSet - atrybut oznaczony tą adnotacją zostanie utrwalony w bazie danych jako zbiór węzłów zawierających prymitywy. adnotacja może być stosowana tylko do atrybutów typu *java.util.Set* zawierających prymitywy. Adnotacja zawiera obowiązkowy atrybut *itemType* definiujący typ obiektów przechowywanych w zbiorze – możliwe wartości to *Integer*, *String* i *Boolean*

4.6. Dodatkowe operacje w języku SBQL

Podczas tworzenia narzędzia odwzorowującego natknąłem się na ograniczenia języka zapytań SBQL uniemożliwiające pełną implementację narzędzia odwzorowującego. W tej sytuacji konieczne było rozszerzenie języka zapytań o dodatkową operację i zaimplementowanie jej obsługi w systemie zarządzania bazą danych LoXiM. Poniżej dodatkowe operacje:

4.6.1. Operacja Nameof

Operacja *nameof(referencja do obiektu)* zwraca nazwę obiektu o podanej referencji. W przypadku, gdy obiekt o podanej referencji nie istnieje zwraca *VOID*.

Rozdział 5

Instrukcja obsługi

5.1. Warstwa połączenia z bazą danych

Warstwa połączenia z bazą danych odpowiada za komunikację z serwerem i wykonywanie zapytań. Połączenie z bazą danych jest reprezentowane przez obiekt typu *SimpleConnection* pobrany z fabryki połączeń – obiektu *Datasource*. Fabrykę połączeń tworzymy następująco:

```
LoximDatasourceImpl ds = new LoximDatasourceImpl();
ds.setHost("127.0.0.1");
ds.setPort(6543);
ds.setLogin("root");
ds.setPassword("");
```

Posiadając przygotowaną fabrykę połączeń możemy rozpocząć połączenie:

```
Connection connection = ds.getConnection();

/* Operacje na bazie danych */

ds.release(connection);
```

W ramach połączenia z bazą danych możemy:

- Rozpocząć/zakończyć transakcję

  ```
  connection.begin();
  try {
          /* Operacje na bazie danych */

          connection.commit();

  } catch (Exception e){
          connection.rollback();
          throw e;
  }
  ```

- Wykonać zapytanie

  ```
  Result result = connection.execute("Employee where name='Stefan'");
  ```

Wynikiem zapytanie będzie drzewo obiektów typu *Result*. Dokładna dokumentacja znajduje się w załączniku A.

- Wykonać zapytanie sparametryzowane. Możemy stosować zapytania z parametrami nazwanymi:

```
Map<String, Result> params = new HashMap<String, Result>();
params.put("name", new ResultString("Stefan"));
params.put("surname", new ResultString("Nowak"));

Result result = connection.executeNamed(
        "Employee where name=:name and surname=:surname",
        params);
```

Możemy też używać zapytań z nienazwanymi parametrami:

```
Result result = connection.executeParam(
        "Employee where name=? and surname=?",
        new ResultString("Stefan"),
        new ResultString("Nowak"));
```

W przypadku wystąpienia błędu wykonania zapytania lub błędu serwera bazy danych warstwa połączenia zgłasza wyjątek typu *SBQLException*. Możliwe wyjątki to:

SBQLIOException - błąd wejścia/wyjścia, powodowany przez standardowy wyjątek *IOException*

SBQLProtocolException - błąd protokołu komunikacyjnego. Warstwa połączenia otrzymała nieoczekiwane dane od serwera

SBQLServerException - błąd zgłoszony przez serwer bazodanowy. Obiekt wyjątku zawiera informację o module, w którym wystąpił błąd (metoda *getModuleCode()*) oraz kod błędu (metoda *getDetailCode()*). Szczegółowe znaczenie poszczególnych kodów opisane jest w kodzie źródłowym systemu LoXiM.

5.2. Moduł nawigacji po obiektach

Warstwa nawigacji po obiektach na przeglądanie hierarchii obiektów w bazie danych i ich modyfikację za pomocą standardowego interfejsu *Node*. Praca z warstwą nawigacji odbywa się w ramach sesji, która jest równoważna z transakcją w bazie danych. Po zakończeniu sesji praca z obiektami utworzonymi/pobranymi w ramach niej nie jest możliwa. Rozpoczęcie sesji polega na utworzeniu obiektu klasy *LoximSession*

```
Session session = new LoximSession(datasource);
```

gdzie parametr *datasource* to źródło połączeń z bazą danych. Po otwarciu sesji możemy pobrać z bazy danych obiekty określone przez zapytanie SBQL

```
Collection<Node> nodes = session.find("Employee where surname='Kowalski'");
```

Obiekt klasy *Node* reprezentuje konkretny trwały obiekt znajdujący się w bazie danych. Ważne jest, że obiekty Node są tylko wskaźnikami do obiektów w bazie i nie zawierają one wartości obiektów w bazie.

Mając obiekt *node* możemy wykonać na nim następujące operacje:

5.2.1. Pobranie nazwy obiektu

```
String nodeName = node.getName()
```

5.2.2. Pobranie identyfikatora obiektu

```
String nodeId = node.getReference()
```

5.2.3. Pobranie wartości obiektu

```
ObjectValue value = node.getValue()
```

Wartość obiektu może być jednego z następujących typów:

SimpleValue Wartość prymitywna (*int*, *String*, *boolean*). Typ wartości możemy pobrać za pomocą metody:

```
SimpleValue.Type type = ((SimpleValue)node.getValue()).getType();
```

Konkretną wartość możemy pobrać za pomocą metody:

```
Integer value = (Integer)((SimpleValue)node.getValue()).getValue();
```

ReferenceValue Referencja do innego obiektu. Obiekt klasy *Node* reprezentujący cel referencji możemy pobrać w następujący sposób:

```
Node targetNode = ((ReferenceValue)node.getValue()).getTargetNode()
```

CompositeValue Wartość złożona – zbiór obiektów. W przypadku takiej wartości metoda *getValue()* zwróci specjalny obiekt klasy *ComplexValue* nie zawierający żadnej informacji. Aby pobrać obiekty składowe wartości złożonej musimy wykonać jedną z następujących metod.

Aby pobrać obiekty składowe o nazwie *childName* wywołujemy metodę:

```
Collection<Node> nodes = node.getChildNodes(childName);
```

W przypadku gdy zakładamy, że dany obiekt ma tylko jedno dziecko o danej nazwie, możemy skorzystać z metody:

```
Node surname = node.getUniqueChildNode("surname");
```

Metoda zwróci ewentualne dziecko o danej nazwie lub *null* gdy dziecko nie istnieje. Użycie tej metody upraszcza kod programu – nie musimy wyłuskiwać pojedynczego obiektu z kolekcji.

Aby pobrać wszystkie obiekty składowe, wywołujemy metodę:

```
Collection<Node> nodes = node.getAllChildNodes(childName);
```

5.2.4. Modyfikowanie wartości obiektów

Jeśli dany obiekt jest wartości prymitywnej (*Boolean*, *Integer*, *String*) lub jest obiektem referencyjnym, wówczas możemy nadać mu nową wartość. Robimy to w sposób następujący:

```
node.setValue(new SimpleValue("Kowalski"));
```

5.2.5. Tworzenie nowych obiektów

Dodanie nowego obiektu do baziy danych sprowadza się do:

- Utworzenia odłączonego obiektu bazy LoXiM,
- Nadania mu oczekiwanej wartości (obiekty odłączone przechowują swoje wartości – wartości te nie są związane z zawartością bazy danych)
- Podłączenia go do korzenia bazy danych, albo jako dziecko istniejącego obiektu.

```
/* Tworzymy odłączony obiekt */
Node newEmployee = new LoximNode("Employee", new ComplexValue());

/* Dodajemy do niego odłączone dzieci */
newEmployee.addChild(new LoximNode("name", new SimpleValue("Jan")));
newEmployee.addChild(new LoximNode("surname", new SimpleValue("Kowalski")));

/* Zapisujemy hierarchię obiektów w korzeniu */
session.addToRoot(newPart);

/* Tworzymy odłączony obiekt */
Node salary = new LoximNode("salary", new SimpleValue(2200));

/*
 Podłączamy odłączony obiekt salary jako dziecko istniejącego w
 bazie obiektu newEmployee
*/

newEmployee.addChild(salary);
```

5.2.6. Usuwanie obiektów

Aby usunąć obiekt z bazy danych wywołujemy metodę *delete()* klasy *Node*:

```
employee.delete();
```

Możemy też za pomocą jednego zapytania usunąć wszystkie dzieci danego obiektu złożonego posiadające tą samą nazwę:

```
department.removeAllChildren("employee");
```

Próba późniejszego odwołania się do obiektu klasy *Node* reprezentującego skasowany obiekt spowoduje zgłoszenie wyjątku *DeletedException*.

5.3. Moduł przezroczystego odwzorowania

Moduł przezroczystego odwzorowania pozwala na utrwalanie w bazie danych obiektów Javy spełniających standard JavaBeans, i późniejsze pobieranie tych obiektów za pomocą zapytań. W celu skorzystania z modułu odwzorowania musimy uprzednio zdefniować model danych.

5.3.1. Definiowanie modelu

Obiekty, które mogą zostać utrwalone w bazie danych muszą spełniać część założeń standardu JavaBeans, a przede wszystkim:

- Posiadać bezargumentowy konstruktor
- Dla każdego atrybutu *xxx* obiektu muszą istnieć publiczne metody:

  getXxx() (dla argumentów typu boolean isXXX()) zwracająca wartość atrybutu
  setXxx(T value) zmieniająca wartość atrybutu

Atrybuty obiektów należących do modelu mogą mieć następujące typy:

- Typy proste obsługiwane przez LoXiM (*boolean*, *int*, *String*).
- Klasy należace do modelu (odwzorowane jako referencja lub obiekt składowy).
- Zbiór obiektów prostych lub obiektów klas należących do modelu (w tym przypadku konieczne jest określenie szczegółowego sposobu odwzorowania zbioru za pomocą adnotacji).

Dodatkowo dla każdej klasy należącej do modelu musi istnieć interfejs udostępniający opisane powyżej metody. Interfejs ten może zostać dodatkowo opisany adnotacjami Javy definiującymi sposób w jaki obiekty LoXiM mogą być na niego odwzorowane. Szczegółowy opis adnotacji znajduje się punkcie 4.5.1.

Przykładowy model danych wygląda następująco:

```
/* Interfejs */
public interface Employee {

        public String getName();

        public void setName(String name);

        public String getSurname();

        public void setSurname(String surname);

        @Persistent(nodeName="dept")
        @Reference
```

```
        public Department getDepartment();

        public void setDepartment(Department department);


}

/* Implementacja */
public class EmployeeImpl implements Employee {

    private String name;

    private String surname;

    private Department department;

    public Department getDepartment() {
        return department;
    }

    public void setDepartment(Department department) {
        this.department = department;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getSurname() {
        return surname;
    }

    public void setSurname(String surname) {
        this.surname = surname;
    }

}
```

5.3.2. Rozpoczęcie pracy

Aby rozpocząć pracę z warstwą przezroczystego odwzorowania musimy stworzyć i skonfigurować szereg obiektów pomocniczych odpowiedzialnych za właściwą pracę narzędzia, w szczególności:

- Obiekty odpowiedzialne za przekształcenia atrybutów Javy na obiekty bazodanowe

- Obiekty przechowujące informacje o modelu danych
- Fabrykę przezroczystych obiektów – tworzącą obiekty Javy na podstawie danych pobranych z bazy
- Obiekt datasource reprezentujący fabrykę połączeń do bazy danych.

Za wykonanie wszystkich powyższych działań odpowiedzialna jest klasa *SimpleTransparentSessionFactoryImpl*. Utworzenie i skonfigurowanie środowiska przezroczystego odwzorowania wygląda następująco:

```
LoximDatasource ds = ...;

SimpleTransparentSessionFactoryImpl sessionFactory =
        new SimpleTransparentSessionFactory();

/* Źródło danych */
sessionFactory.setDatasource(ds);

/* Klasy należące do modelu */
sessionFactory.setClasses(new Class[] {
        pl.tzr.model.Employee.class,
        pl.tzr.model.Department.class
});
```

Mając skonfigurowaną fabrykę sesji możemy rozpocząć sesję:

```
TransparentSession session = sessionFactory.getSession();
```

Sesję kończymy wykonując operację:

```
session.commit();
```

lub

```
session.rollback();
```

5.3.3. Utrwalanie obiektów

Utwórzmy obiekt jednej z klas należących do modelu:

```
Employee emp = new EmployeeImpl();
emp.setName("Jan");
emp.setSurname("Kowalski");
```

Aby utrwalić go w bazie danych wykonujemy operację:

```
Employee persistentEmp = session.persist(emp);
```

Wartość obiektu *emp* zostanie zapisana do bazy danych, a my otrzymamy obiekt *persistentEmp* – reprezentację obiektu *emp* w bazie danych. Od tego momentu powinniśmy posługiwać się referencją *persistentEmp* a nie *emp*. *Emp* pozostaje zwykłym lokalnym obiektem Javy, natomiast *persistentEmp* to obiekt posiadający cechy trwałości.

5.3.4. Modyfikacja obiektów

Wszelkie operacje na obiektach trwałych, jak na przykład poniższa, pociągają za sobą zmiany w bazie danych:

```
persistentEmp.setSurname("Nowak");
```

W przypadku gdy mamy do czynienia z typami prostymi, sprawa jest oczywista. Gdy modyfikujemy atrybuty złożonych typów, semantyka operacji ustawienia atrybutu zależy od ustawień odwzorowania. Spróbujmy ustawić atrybut złożony, który jest odwzorowany w LoXiM jako referencja:

```
persistentEmp.setDepartment(dept);
```

W takim przypadku narzędzie odwzorowujące będzie oczekiwać, że *dept* to obiekt trwały i zapisze referencję do niego jako atrybut trwałego obiektu *persistentEmp*. W przypadku gdy dept nie jest obiektem trwałym zgłoszony zostanie wyjątek *ObjectNotPersistentException*.

Rozważmy przypadek, gdy atrybut złożony jest odwzorowany w LoXiM jako obiekt składowy:

```
Address addr = new Address();
addr.setStreet("Zielona");
addr.setCity("Warszawa");
persistentEmp.setAddress(addr);
```

W powyższej sytuacji obiekt *addr* zostanie odwzorowany jako obiekt składowy obiektu *persistentEmp*. Możemy jednakże natknąć się na pewien problem.

Może się zdarzyć, że chcielibyśmy podstawić obiekt trwały jako wartość atrybut złożony odwzorowanego jako obiekt składowy. Jak wiadomo dany obiekt trwały może istnieć tylko w jednym miejscu w drzewie obiektów bazy danych. Wykonując powyższą operację chcielibysmy umieścić go w innym miejscu drzewa. W tym momencie działanie narzędzia odwzorowującego zależy od ustawien odwzorowania atrybutu, a konkretnie od właściwości adnotacji *@Component* o nazwie *onBind*. Możliwe działania to:

BIND - przeniesienie obiektu trwałego reprezentującego nową wartośc atrybutu w nowe miejsce.

COPY - skopiowanie wartości obiektu trwałego do nowego obiektu i podpięcie kopii w nowe miejsce.

DENY - zgłoszenie wyjątku.

W przypadku gdy ustawiamy wartość atrybutu odwzorowanego na obiekt składowy na *null*, wówczas w zależności ustawień odwzorowania (właściwość adnotacji@Component o nazwie *onRemove*) wykonywana jest następująca operacja:

DELETE – Usunięcie obiektu trwałego reprezentującego dotychczasową wartość atrybutu.

MOVE_TO_ROOT – Przeniesienie obiektu trwałego reprezentującego dotychczasową wartość atrybutu do korzenia.

5.3.5. Operacje na zbiorach

Gdy pobierzemy wartość atrybutu typu zbiorowego obiektu trwałego, wówczas otrzymamy obiekt implementujący interfejs *java.util.Set*, operujący na zawartości bazy danych (dla uproszczenia nazwijmy go *zbiorem trwałym*) – wszelkie operacje zdefiniowane w interfejsie *Set* będą operować na danych w bazie.

```
Employee persistentEmp = ...;

Set<Address> persistentAddresses = persistentEmp.getAddresses();

Address newAddress = new Address();
newAddress.setStreet("Zielona 6");
newAddress.setCity("Warszawa");

persistentAddresses.add();
```

Wszystkie operacje na zbiorze trwałym (dodawanie, usuwanie, część wspólna, iterowanie itd.) działają zgodnie z założeniami interfejsu *Set*.

Atrybuty zbiorowe podobnie jak atrybuty proste, mogą zostać odwzorowane na trzy sposoby:

- obiekty proste (gdy typ obiektów przechowywanych w zbiorze jest prosty)
- obiekty składowe
- referencje

Sposób odwzorowania określamy definiując model danych. Oczywiście każdy z powyższych sposobów niesie za sobą ograniczenia:

- Jeśli elementy zbioru są odwzorowane jako referencje, wówczas do zbioru trwałego możemy dodać tylko obiekty trwałe, gdy spróbujemy dodać obiekt lokalny wówczas zgłoszony zostanie wyjątek
- Jeśli elementy zbioru są odwzorowane jako obiekty składowe, dodanie do zbioru obiektu trwałego jak i usunięcie elementu ze zbioru będzie niosło za sobą konsekwencje analogiczne do opisanych w punkcie 5.3.4. Oczywiśćie tu również, podobnie jak w przypadku atrybutów odwzorowywanych na obiekty składowe, mamy możliwość wyboru właściwej semantyki.

W przypadku, gdy podstawimy lokalny obiekt klasy *Set*, jako wartość atrybutu zbiorowego obiektu trwałego, wówczas wszystkie elementy należące do lokalnego zbioru zostaną utrwalone w bazie jako wartość atrybutu. Gdy następnie pobierzemy wartość tego atrybutu, otrzymamy w wyniku zbiór trwały.

5.3.6. Zapytania

Po utrwaleniu obiektów w bazie danych możemy je pobierać korzystając z zapytań w języku SBQL

```
Collection<Employee> janki =
        session.find("Employee where name='Janek'", Employee.class);
```

Pierwszym metody *find* jest zapytanie, natomiast drugim jest oczekiwany typ obiektu trwałego zwróconego przez zapytanie. Zapytanie powinno zwrócić zbiór referencji do obiektów trwałych – w przypadku, gdy zwrócone zostaną inne wartości niż referencje, metoda zgłosi wyjątek.

Narzędzie będzie interpretować zbiór referencji jako zbiór identyfikatorów obiektów odwzorowanych na klasę *Employee*. Nawet gdy referencje wskazują na obiekty innych klas bądź na obiekty nie odwzorowane na Javę, narzędzie będzie próbować stworzyć na ich podstawie obiekty klasy *Employee* – zgodnie z regułami odwzorowania tej klasy. Możliwe efekty uboczne takiej operacji opisane są w punkcie 4.4.5.

Oczywiście możliwe jest też wywołanie zapytania sparametryzowanego. Zapytania możemy parametryzować wartościami prymitywnymi:

```
int minSalary = 1000;
Collection<Employee> underpaid = session.findWithParams(
        "Employee where salary > ?",
        Employee.class, minSalary);
```

Możemy też parameteryzować je obiektami trwałymi. W takiej sytuacji w miejsce argumentu w zapytaniu wstawiona zostanie referencja do obiektu trwałego.

```
Department itDepartment = ...
Collection<Employee> itGuys = session.findWithParams(
        "Employee where department = ?",
        Employee.class, itDepartment);
```

Oczywiście możemy stosować dowolną liczbę parametrów obu rodzajów.

Rozdział 6

Praktyczne użycie

6.1. Prosta aplikacja WWW – działy i pracownicy

Prawdziwym sprawdzianem dla wszelkich narzędzi programistycznych jest oczywiście użycie ich w praktyce. Aby sprawdzić, na ile efektywne jest korzystanie z narzędzia odwzorowującego stworzyłem przykładową aplikację WWW opartą o bazę danych LoXiM. Aplikacja pozwala na ewidencjonowanie pracowników i działów pewnej firmy oraz przydzielanie pracowników do działów. Dostępne operacje to:

- Przeglądanie listy pracowników.
- Przeglądanie listy działów.
- Przeglądanie listy pracowników konkretnego działu.
- Dodawanie/edycja/usuwanie działu.
- Dodawanie/edycja/usuwanie pracownika.
- Zmiana przynależności pracownika do działu.

Aplikacja zrealizowana została przy użyciu następujących technologii:

Spring Framework 2.0 - szkielet aplikacji, środowisko Model-View-Controller, obsługa transakcji.

Java Server Pages - interfejs użytkownika.

Apache Maven 2 - narzędzie do budowania projektu.

Apache Tomcat - serwer aplikacji.

6.2. Obsługa transakcji w Springu

Podczas projektowania przykładowej aplikacji zintegrowałem obsługę transakcji na bazie LoXiM z modułem obsługi transakcji biblioteki Spring. Umożliwiło to obsługę transakcji na bazie w sposób typowy dla Springa, w tym wykorzystanie transakcji deklaratywnych. Mechanizm transakcji deklaratywnych, oparty na programowaniu aspektowym, pozwala nam uniknąć jawnego wplatania w logikę aplikacji operacji rozpoczynania, zatwierdzania i anulowania transakcji. Zamiast tego wystarczy tylko wskazać kod, który musi wykonać się w

ramach transakcji (czy to za pomocą adnotacji, czy poprzez zewnętrzny plik konfiguracyjny). Zaletą zunifikowanego kodu zarządzającego transakcjami jest możliwość łatwego podmienienia go kodem współpracującym z inną bazą – logika aplikacji się w tej sytuacji nie zmieni. Przykładowy kod korzystający z deklaratywnych transakcji wygląda następująco:

```
@Transactional
void transferMoney(Account source, Account destination, BigDecimal amount) {

  if (source.getAmount().compareTo(amount) < 0) throw
        new NotEnoughMoneyException();

        source.removeAmount(amount);
        destination.addAmount(amount);
}
```

Domyślnie Spring zatwierdza transakcję w momencie ukończenia kodu wymagającego transakcyjności, natomiast w sytuacji gdy zostanie zgłoszony wyjątek, wówczas transakcja zostanie anulowana.

Rozdział 7

Podsumowanie

Celem pracy było stworzenie przyjaznego, efektywnego interfejsu programistycznego pozwalającego łatwo i intuicyjnie używać bazy danych w praktycznych projektach.

Przyglądając się kodowi źródłowemu przykładowej aplikacji można stwierdzić, że zamierzony cel został osiągnięty. Trwałość obiektów jako cecha niezależna od logiki programu czyni kod prostym i czytelnym. Znacznie upraszcza to pracę nad projektowaniem aplikacji i dalszym jej rozwojem.

Narzędzie to jest w tej chwili tylko prototypem. Już na tym etapie widać jednak jak atrakcyjne może być zastosowanie semistrukturalnych baz danych do realizacji przezroczystej trwałości obiektów w języku Java. Istnieje wiele potencjalnych kierunków rozwoju. Z jednej strony można zaadaptować narzędzie do zastosowań praktyczno-komercyjnych: rozbudować możliwości konfiguracyjne, dodać obsługę nowych typów danych (listy, mapy, typy wyliczeniowe). Z drugiej zaś, rozwój systemu LoXiM otworzy nowe możliwości ścisłej integracji obu środowisk.

Istotną i bardzo interesującą kwestią (która nie została rozwinięta w tej pracy) jest optymalizacja operacji na bazie danych i zmniejszenie ilości wykonywanych zapytań. Tutaj pojawia się ciekawy problem buforowania danych pobranych z bazy LoXiM, który może być tematem dalszych badań.

Podczas pracy nad narzędziem odwzorowującym dużo uwagi poświęciłem modularności. Daje to wiele możliwości adaptacji rozwiązania do nowych celów. Z pewnością interesujące byłoby zastosowanie narzędzia do pracy z innym systemem zarządzania bazami danych lub z danymi w formacie XML. Ciekawe też byłoby wypróbowanie innych niż *DynamicProxyClasses* mechanizmów modyfikowania semantyki obiektów[2, 1].

7.1. Możliwe rozszerzenia

Tworzenie obiektów pośredniczących za pomocą innych mechanizmów - eliminacja konieczności tworzenia interfejsu dla każdej klasy obiektu przechowywanego w bazie.

Optymalizacja komunikacji z bazą - zmniejszenie liczby zapytań wykonywanych przez narzędzie odwzorowujące.

Buforowanie danych - wprowadzenie mechanizmu buforowania zmniejszającego liczbe zapytań wykonywanych przez narzędzie.

Obsługa odłączania i ponownego podłączania obiektów do sesji - wprowadzenie możliwości modyfikowania obiektów trwałych bez konieczności połączenia z bazą danych.

Obsługa innych rodzajów kolekcji niż *Set* - obsługa słowników i list.

Możliwość wywoływania metod obiektów bazodanowych - przyszłe wersje bazy LoXiM będą obsługiwać metody obiektów. Wprowadzenie możliwośći przezroczystego – zdalnego wywoływania tych metod.

Dodatek A

Przegląd klas narzędzia odwzorowującego

A.1. Warstwa sterownika

A.1.1. pl.tzr.driver.loxim.Connection

Interfejs reprezentujący połączenie z semistrukturalną bazą danych

Metody

- `Result execute(String query) throws SBQLException`

  Wykonuje zapytanie *query*. Zwraca wynik zapytania.

- `long parse(String query) throws SBQLException`

  Przygotowuje do wykonania sparametryzowane zapytanie *query*. Miejsca na parametry w zapytaniu powinny mieć postać *:nawaParametru*. Zwraca identyfikator przygotowanego zapytania pozwalający je później wykonać za pomocą metody *execute(statementId, params)*.

- ```
  Result execute(
          long statementId,
          Map<String, Result> params)
  throws SBQLException;
  ```

  Wykonuje uprzednio przygotowane zapytanie sparametryzowane. Argument *statementId* powinien zawierać identyfikator przygotowanego zapytania, a *params* listę nazwanych parametrów.

- `close() throws SBQLException`

  Zamyka połączenie z bazą danych, przerywając ewentualną transakcję

A.1.2. pl.tzr.driver.loxim.TcpConnection

Implementacja interfejsu *pl.tzr.driver.loxim.Connection* realizująca dostęp do bazy danych LoXiM za pomocą protokołu TCP/IP.
  ```

Konstruktory

- `TcpConnection(final Socket socket) throws IOException`

  Rozpoczyna sesję z bazą danych w oparciu o otwarte połączenie TCP/IP reprezentowane przez *socket*.

A.1.3. pl.tzr.driver.loxim.SimpleConnection

Rozszerzenie interfejsu *pl.tzr.driver.loxim.Connection* o wysokopoziomowe operacje takie jak obsługa transakcji czy prostsze wykonywanie zapytań sparametryzowanych.

Metody

- ```
  Result executeNamed(
          String query,
          Map<String, Result> params)
  throws SBQLException;
  ```

  Przygotowuje i wykonuje zapytanie sparametryzowane *query* z nazwanymi parametrami. Miejsca na parametry w zapytaniu powinny mieć postać *:nawaParametru*. Wartości argumentów zapytania powinny zostać umieszczone w *params*.

- ```
  Result executeParam(
          String query,
          Result... params)
  throws SBQLException
  ```

  Przegotowuje i wykonuje zapytanie sparametryzowane *query* z nienazwanymi parametrami. Miejsca na parametry w zapytaniu powinny mieć postać znaku zapytania (*?*). Wartości argumentów zapytania powinny zostać umieszczone jako kolejne parametry metody (*params*)

- `void beginTransaction() throws SBQLException`

  Rozpoczyna transakcję

- `void commitTransation() throws SBQLException`

  Zatwierdza aktualnie wykonywaną transakcję

- `void rollbackTransaction() throws SBQLException`

  Przerywa aktualnie wykonywaną transakcję

A.1.4. pl.tzr.driver.loxim.SimpleConnectionImpl

Implementacja interfejsu *pl.tzr.driver.loxim.SimpleConnection* obudowująca istniejące połączenie typu *pl.tzr.driver.loxim.Connection* o nową funkcjonalnośc. Klasa realizuje również logowanie do bazy danych.

Konstruktory

- `SimpleConnectionImpl(`
  `final Connection connection,`
  `final String login,`
  `final String password)`
  `throws SBQLException`

  Dokonuje logowania do bazy danych reprezentowanej przez *connection* i tworzy obiekt typu reprezentujący zautoryzowane połączenie implementujący interfejs *SimpleConnection*.

A.1.5. pl.tzr.driver.loxim.LoximDatasource

Interfejs reprezentujący fabrykę połączeń do określonej bazy danych. Pozwala na pobranie połączenia do bazy danych i jego późniejszy zwrot. Możliwe są rózne implementacje tego interfejsu dla różnych rodzajów baz danych, a także realizujących różne sposoby przydzielania połączeń – praca na jednym połączeniu, tworzenie połączeń na żądanie, pula połączeń i tym podobne.

Metody

- `SimpleConnection getConnection() throws SBQLException`

  Pobiera połączenie do bazy danych.

- `void release(SimpleConnection connection) throws SBQLException`

  Zwalnia połączenie (*connection*) z bazą danych.

A.1.6. pl.tzr.driver.loxim.LoximDatasourceImpl

Implementacja interfejsu *LoximDatasource* pozwalająca na tworzenie połączeń do bazy danych LoXiM – wywołanie metody *getConnection()* powoduje utworzenie nowego połączenia z bazą, a metoda *release(connection)* zamyka połączenie.

Metody

- `void setHost(String host)`

  Określa adres TCP/IP używanego serwera bazy danych. Ustawienie atrybutu *host* jest konieczne przed korzystaniem z fabryki połączeń.

- `void setPort(int port)`

  Określa port TCP/IP, na którym nasłuchuje używany serwer bazy danych. Ustawienie atrybutu *port* jest konieczne przed korzystaniem z fabryki połączeń.

- `void setHost(String host)`

  Określa nazwę użytkownika bazy. Ustawienie atrybutu *login* jest konieczne przed korzystaniem z fabryki połączeń.

- `void setPassword(String password)`

  Określa hasło użytkownika bazy. Ustawienie atrybutu *host* jest konieczne przed korzystaniem z fabryki połączeń.

A.1.7. pl.tzr.driver.loxim.result.Result

Ogólny interfejs dla obiektów zwracanych przez system LoXiM w zapytaniu. Możliwe obiekty to:

ResultBag - bag

ResultBinder - para <nazwa, obiekt>

ResultBool - wartość logiczna

ResultDouble - wartość zmiennopozycyjna

ResultError - błąd

ResultInt - wartość całkowita

ResultReference - referencja do obiektu w bazie

ResultString - wartość znakowa

ResultSequence - sekwencja

ResultStruct - struktura obiektów

ResultVoid - wartość pusta

Poszczególne implementacje interfejsu dla konkretnych typów danych zawierają metody pozwalające na pobranie wartości obiektu.

A.1.8. pl.tzr.driver.loxim.Package

Reprezentuje pakiet danych przesyłany między serwerem a klientem. Poszczególne implementacje tego interfejsu pozwalają na serializację i deserializację pakietów zgodnie z protokołym używanym przez system LoXiM. Protokół ten przewiduje następujące rodzaje pakietów:

SIMPLEQUERY - pakiet zawierający treść zapytania. Przesyłany od klienta do serwera.

PARAMQUERY - pakiet zawierający treść zapytania sparametryzowanego. Przesyłany od klienta do serwera.

STATEMENT - pakiet zawierający identyfikator przygotowanego zapytania sparametryzowanego. Przesyłany od serwera do klienta po przygotowaniu zapytania.

PARAMSTATEMENT - pakiet zawierający identyfikator oraz parametry zapytania sparametryzowanego. Przesyłany od klienta do serwera w celu wykonania zapytania sparametryzowanego.

SIMPLERESULT - pakiet zawierający wynik wykonanego zapytania skonwertowany do postaci obiektu klasy *Result*. Przesyłany od serwera do klienta po pomyślnym wykonaniu zapytania.

ERRORRESULT - pakiet przesyłany z serwera do klienta po wystąpieniu błędu bazy danych. Zawiera szczegółowe informacje o błędzie.

Metody

- `int serialize(OutputStream stream) throws SBQLException`

  Serializuje pakiet do strumienia wyjściowego *stream*. Zwraca wielkość wysłanego pakietu.

- `void deserialize(InputStream stream, int size) throws SBQLException`

  Wczytuje pakiet z podanego strumienia wyjściowego *stream*. Argument *size* zawiera zadeklarowaną wielkość wczytywanego pakietu.

A.2. Warstwa przegladania

A.2.1. pl.tzr.browser.session.Session

Interfejs reprezentujący sesję pracy z warstwą przeglądania (pojedynczą transakcję w bazie danych). Obiekt sesji jest odpowiedzialny za przechowywanie informacji o pobranych obiektach oraz zarządzanie połączeniem. Obiekty z bazy danych pobrane w ramach sesji są ważne tylko w ramach tej sesji.

Metody

- ```
  Collection<Node> find(
          String query,
          ObjectValue... params)
  throws SBQLException
  ```

  Wykonuje w ramach sesji zapytanie *query* z ewentualnymi nienazwanymi parametrami o wartościach *params*. Oczekiwany wynik zapytania to bag referencji do obiektów znajdujących się w bazie, który jest konwertowany na kolekcję obiektów *Node*.

- ```
  Collection<SimpleValue> findPrimitive(
          String query,
          ObjectValue... params)
  throws SBQLException
  ```

  Wykonuje w ramach sesji zapytanie *query* z ewentualnymi nienazwanymi parametrami o wartościach *params*. Oczekiwany wynik zapytania to bag prostych wartości (integer, string, boolean) konwertowany na obiekty klasy *SimpleValue*.

- `void addToRoot(Node node) throws SBQLException`

  Utrwala obiekt o wartości reprezentowanej przez *node* w bazie danych i umieszcza go w korzeniu hierarchii obiektów. Obiekt *node* musi być w stanie rozłączonym.

- `Node fetchNode(final String ref)`

  Zwraca wskaźnik do obiektu w bazie o identyfikatorze *ref*, zwraca null jeśli obiekt o tym identyfikatorze nie istnieje.

- `void commit()`

  Zatwierdza aktualną transakcję i kończy sesję.

- `void rollback()`

  Przerwa aktualną transakcję i kończy sesję.

- `boolean isActive()`

  Zwraca *true* jeśli sesja jest aktywna.

A.2.2. pl.tzr.browser.session.LoximSession

Implementacja interfejsu *Session* korzystająca z bazy danych LoXiM.

Konstruktory

- `LoximSession(final LoximDatasource datasource)`

  Tworzy nową sesję warstwy przeglądania korzystając z połączenia utworzonego za pomocą fabryki *datasource*.

Metody

- `SimpleConnection getConnection()`

  Zwraca połączenie z bazą LoXiM używane przez daną sesję.

- `Map<String, Node> getFetchedNodes()`

  Zwraca mapę obiektów bazodanowych pobranych w ramach tej sesji. Kluczem w mapie jest identyfikator obiektu, wartością jest obiekt wskaźnikowy *Node*.

- `LoximExecutor getExecutor()`

  Zwraca obiekt odpowiedzialny za wykonywanie i interpretację zapytań.

- `Node createNode(final String ref, final String name)`

  Tworzy obiekt klasy Node reprezentujący obiekt o identyfikatorze *ref* i nazwie *name* istniejący w bazie danych. Jeśli w ramach sesji utworzony już został reprezentant tego obiektu w bazie danych, zwraca go zamiast tworzyć nowy.

A.2.3. pl.tzr.browser.store.node.Node

Interfejs reprezentujący wskaźnik do obiektu w bazie semistrukturalnej. Pozwala na dokonywanie modyfikacji obiektu oraz zarządzanie jego ewentualnymi obiektami składowymi. Obiekt *Node* może być w dwóch stanach – połączonym (związany z obiektem w bazie danych) i odłączonym (zawiera wartość obiektu ale nie jest związany z bazą danych).

Metody

- `String getReference()`

  Zwraca identyfikator wskazywanego obiektu.

- `String getName()`

  Zwraca nazwę wskazywanego obiektu.

- `ObjectValue getValue()`

  Zwraca wartość wskazywanego obiektu.

- `void setValue(ObjectValue value)`

  Ustala nową wartość wskazywanego obiektu – w przypadku gdy jest to obiekt prosty bądź referencja. W przypadku gdy wskazywany obiekt jest typu złożonego, zgłasza wyjątek.

- `Collection<Node> getChildNodes(String propertyName)`

  Zwraca dzieci wskazywanego obiektu o nazwie *propertyName* – w przypadku gdy wskazywanym obiektem jest obiekt złożony. W przeciwnym wypadku zgłasza wyjątek.

- `Collection<Node> getAllChildNodes()`

  Zwraca wszystkie dzieci wskazywanego obiektu – w przypadku gdy wskazywanym obiektem jest obiekt złożony. W przeciwnym wypadku zgłasza wyjątek.

- `Node getUniqueChildNode(String propertyName)`

  Zwraca obiekt składowy o nazwie *propertyName* wskazywanego obiektu. W przypadku, gdy nie istnieje obiekt składowy o takiej nazwie zwraca *null*. W przypadku gdy istnieje wiele obiektów składowych o takiej nazwie, zgłasza wyjątek.

- `void addChild(Node child)`

  Podłącza obiekt wskazywany przez *child* jako dziecko aktualnego obiektu. W przypadku gdy aktualny obiekt nie jest typu złożonego, zgłasza wyjątek.

- `void delete()`

  Usuwa wskazywany obiekt z bazy danych.

- `boolean isDetached()`

  Zwraca *true*, jeśli obiekt *Node* jest w stanie odłaczonym – to znaczy, że zawiera (oprócz referencji) wartość obiektu i może być użytkowany poza sesją.

- `void markAttached(LoximSession session, String ref)`

  Oznacza odłaczony obiekt jako podłączony i podpina go do sesji *session* oraz identyfikatora *ref*. Metoda jest wywoływana przez *Executor* po zapisaniu odłączonego obiektu w bazie danych i otrzymaniu przyznanego mu identyfikatora.

- `boolean isChild(Node childNode)`

  Zwraca *true* jeśli dany obiekt jest obiektem składowym obiektu *childNode*.

- `int childAmount(String name)`

  Zwraca liczbę obiektów składowych wskazywanego obiektu o nazwie *name*. Zgłasza wyjątek jeśli wskazywany obiekt nie jest złożony.

- `void removeAllChildren(String childName)`

  Usuwa wszystkie dzieci wskazywanego obiektu posiadające nazwę *childName*. Zgłasza wyjątek jeśli wskazywany obiekt nie jest złożony.

- `Collection<Node> getChildrenWithValue(String name, ObjectValue value)`

  Zwraca wszystkie dzieci wskazywanego obiektu posiadające wartość *value* i nazwę *name*.

A.2.4. pl.tzr.browser.store.node.LoximNode

Implementacja interfejsu *node* przeznaczona do współpracy z bazą danych LoXiM.

Konstruktory

- `LoximNode(LoximSession loximSession, String ref, String name)`

  Tworzy wskaźnik do obiektu w bazie danych o nazwie *name* i identyfikatorze *ref* w ramach sesji *loximSession*. Utworzony wskaźnik jest w stanie podłączonym.

- `LoximNode(String name, ObjectValue value)`

  Tworzy odłączony wskaźnik do nowego obiektu o nazwie *name* i wartości *value* nie mający odpowiednika w bazie danych.

A.2.5. pl.tzr.browser.store.node.ObjectValue

Wspólny interfejs dla możliwych wartości obiektu w bazie LoXiM. Konkretne wartości obiektu reprezentowane są przez klasy *SimpleValue*, *ReferenceValue* oraz *ComplexValue*

A.2.6. pl.tzr.browser.store.node.SimpleValue

Klasa reprezentująca wartość obiektu prostego.

Konstruktory

- `SimpleValue(boolean value)`

  Tworzy reprezentację wartości obiektu o wartości logicznej *value*

- `SimpleValue(int value)`

  Tworzy reprezentację wartości obiektu o wartości całkowitej *value*

- `SimpleValue(String value)`

  Tworzy reprezentację wartości obiektu o wartości znakowej *value*

Metody statyczne

- `static SimpleValue build(Object item)`

  Tworzy reprezentację wartości obiektu o wartości *item*. Typ wartości jest zalezny od typu argumentu *item*

Metody

- `Type getType()`

  Zwraca typ wartości przechowywanej przez obiekt. Możliwe typy to:

  - BOOL
  - INT
  - STRING

- `Boolean getBoolean()`

  Zwraca wartość typu *Boolean*

- `Integer getInteger()`

  Zwraca wartość typu *Integer*

- `String getString()`

  Zwraca wartość typu *String*

- `Object getValue()`

  Zwraca wartość obiektu

A.2.7. pl.tzr.browser.store.ReferenceValue

Klasa służąca do przechowywania wartości typu referencyjnego.

Konstruktory

- `ReferenceValue(final Node targetNode)`

  Tworzy wartość będącą referencją do obiektu *targetNode*

Metody

- `Node getTargetNode()`

  Zwraca wskaźnik do celu referencji.

A.2.8. pl.tzr.browser.store.ComplexValue

Klasa służąca do reprezentowania wartości obiektów złożonych. Obiekty tej klasy nie przechowują żadnych informacji. Dostęp do dzieci obiektu złożonego odbywa się za pomocą odpowiednich metod klasy *Node*.

A.2.9. pl.tzr.browser.store.LoximExecutor

Klasa odpowiedzialna za wykonywanie wszelkich zapytań do bazy danych związanych z warstwą przeglądania i interpretację ich wyników. Poszczególne węzły oraz sesje delegują operacje manipulacji bazy danych do *LoximExecutor*.

Konstruktory

- `LoximExecutor(LoximSession loximSession)`

  Tworzy obiekt klasy executor związany z sesją *loximSession*.

Metody

- `void addChild(String parentRef, Node child)`

  Podłacza odłaczony obiekt reprezentowany przez *node* jako dziecko obiektu złożonego o identyfikatorze *parentRef*. Zgłasza wyjątek, jeśli obiekt o identyfikatorze *parentRef* nie istnieje albo nie jest typu złożonego.

- `Node loadObject(String ref)`

  Zwraca wskaźnik do obiektu o identyfikatorze *ref* lub *null* jeśli obiekt o takim identyfikatorze nie istnieje.

- `Node createSimpleObject(String name, ObjectValue value)`

  Tworzy prosty obiekt w korzeniu bazy danych o nazwie *name* i wartości *value*.

- `void attachObject(Node node)`

  Zapisuje w bazie odłączony obiekt *node* i oznacza go jako podłączony.

- `void deleteObject(String ref)`

  Usuwa z bazy danych obiekt o identyfikatorze *ref*.

- `ObjectValue getValue(String ref)`

  Zwraca wartość obiektu o identyfikatorze *ref*.

- `Collection<Node> getChildNodes(String parentRef, String propertyName)`

  Pobiera obiekty składowe o nazwie *propertyName* obiektu złożonego o identyfikatorze *parentRef*.

- `void setValue(String ref, ObjectValue value)`

  Ustala wartość obiektu prostego lub referencji o identyfikatorze *ref* na *value*. W przypadku gdy nowa wartość jest typu *CompositeValue*, zgłasza wyjątek.

- `Collection<Node> find(String query, ObjectValue... objectValues)`

  Wykonuje sparametryzowane zapytanie *query* z parametrami *objectValues* i zwraca wynik.

- `boolean isChild(String parentRef, Node childNode)`

  Zwraca *true*, jeśli obiekt *childNode* jest dzieckiem obiektu o identyfikatorze *parentRef*.

- `int childAmount(String parentRef, String childName)`

  Zwraca liczbę dzieci o nazwie *childName* obiektu o identyfikatorze *parentRef*.

- `void removeAllChildren(String parentRef, String childName)`

  Usuwa wszystkie dzieci o nazwie *childName* obiektu o identyfikatorze *parentRef*.

- ```
  Set<Node> findChildrenOfValue(
          String parentRef,
          String name,
          ObjectValue value)
  ```

  Zwraca wszystkie dzieci o nazwie *name* obiektu o identyfikatorze *parentRef* posiadające wartość *value*.
  ```

A.3. Warstwa przezroczystego odwzorowania

A.3.1. pl.tzr.transparent.TransparentSession

Interfejs sesji połączenia z bazą danych w warstwie przezroczystego odwzorowania. Sesja jest równoważna transakcji z bazą danych.

Metody

- ```
  <T> Collection<T> find(
          String query,
          Class<T> desiredClass
  ) throws SBQLException
  ```

  Wykonuje zapytanie *query* na bazie danych i zwraca kolekcję wyników w formie obiektów trwałych. Zapytanie powinno zwrócić listę referencji do obiektów, dla których zdefiniowano odwzorowanie na Javę. Argument *desiredClass* określa jakiego typu powinny być obiekty zwrócone przez zapytanie.

- ```
  <T> Collection<T> findWithParams(
          String query,
          Class<T> desiredClass,
          Object... params
  )  throws SBQLException
  ```

  Wykonuje sparametryzowane zapytanie *query* na bazie danych i zwraca kolekcję wyników w formie obiektów trwałych. Miejsca z parametrami w zapytaniu powinny zostać oznaczone znakiem „*?*”. Zapytanie powinno zwrócić listę referencji do obiektów, dla których zdefiniowano odwzorowanie na Javę. Argument *desiredClass* określa jakiego typu powinny być obiekty zwrócone przez zapytanie. Dopuszczalnymi wartościami parametrów są obiekty trwałe oraz wartości typów prostych (*boolean*, *int*, *String*).

- ```
  Collection<Object> findPrimitiveWithParams(
              String query,
              Object... params
  ) throws SBQLException
  ```

  Wykonuje sparametryzowane zapytanie *query* na bazie danych i zwraca kolekcję wyników w formie obiektów prostych. Miejsca z parametrami w zapytaniu powinny zostać oznaczone znakiem „?”. Zapytanie powinno zwrócić listę prostych wartości (*boolean*, *int*, *String*). Dopuszczalnymi wartościami parametrów są obiekty trwałe oraz wartości typów prostych.

- `Object persist(Object object)`

  Utrwala wartość lokalnego obiektu *object* w bazie danych. Zwraca obiekt trwały zapisany w bazie danych.

- `void delete(Object object)`

  Usuwa obiekt trwały *object* z bazy danych.

- `boolean isActive()`

  Zwraca true, jeśli sesja połączenia z bazą danych jest aktywna.
  ```

- `void commit()`

  Zatwierdza transakcję i kończy sesję.

- `void rollback()`

  Przerywa transakcję i kończy sesję.

- `String getId(Object object) throws ObjectNotPersistentException`

  Zwraca identyfikator w bazie danych obiektu trwałego *object*. Zgłasza wyjątek *ObjectNotPersistentException* w momencie, gdy jako argument podano obiekt lokalny.

- `Object getById(String id, Class desiredClass)`

  Zwraca obiekt trwały o identyfikatorze *id*. Obiekt jest odwzororowany na interfejs *desiredClass*. W przypadku, gdy obiekt trwały o tym identyfikatorze nie istnieje, zwraca *null*

- `Session getSession()`

  Zwraca aktualnie używaną sesję warstwy przeglądania bazy danych.

- `DatabaseContext getDatabaseContext()`

  Zwraca obiekt zawierający konfigurację przezroczystego odwzorowania.

A.3.2. pl.tzr.transparent.TransparentSessionFactory

Interfejs dla fabryk inicjujących nowe sesje połączenia z bazą w warstwie przezroczystego odwzorowania.

Metody

- `TransparentSession getSession()`

  Rozpoczyna nową sesję pracy z warstwą przezroczystego odwzorowania.

A.3.3. pl.tzr.transparent.SimpleTransparentSessionFactoryImpl

Implementacja fabryki sesji w warstwie przezroczystego odwzorowania. Pozwala tworzyć połaczenia z bazą LoXiM. Aby stworzyć sesję trzeba uprzednio ustawić następujące atrybuty fabryki:

- *datasource* - źródło połączeń do bazy danych
- *classes* lub *classNames* - klasy składające się na model danych

Metody

- `void setDatasource(LoximDatasource datasource)`

  Określa źródło połączeń do bazy danych którego będzie używać fabryka sesji.

- `void setClasses(Class[] classes)`

  Określa jakie klasy mają składać się na model danych. Klasy podane są jako tablica obiektów *Class*.

- `void setClassNames(Collection<String> classNames)`

  Alternatywny do metody setClasses sposób określenia klas składających się na model danych. Argumentem metody jest kolekcja pełnych nazw klas należących do modelu

A.3.4. pl.tzr.transparent.DatabaseContext

Obiekt przechowujący konfigurację warstwy przezroczystego odwzorowania – wszystkie statyczne informacje na temat modelu danych, typów danych i sposobów odwzorowania.

Metody

- `ModelRegistry getModelRegistry()`

  Zwraca obiekt przechowujący informacje o modelu danych.

- `void setModelRegistry(ModelRegistry registry)`

  Ustala obiekt przechowujący informacje o modelu danych.

- `TransparentProxyFactory getTransparentProxyFactory()`

  Zwraca obiekt zarządzający cyklem życia obiektów trwałych.

- ```
  void setTransparentProxyFactory(
          TransparentProxyFactory transparentProxyFactory)
  ```

  Określa obiekt zarządzający cyklem życia obiektów trwałych.

A.3.5. pl.tzr.transparent.TransparentProxyFactory

Interfejs obiektów odpowiedzialnych za tworzenie trwałych obiektów Javy na podstawie obiektu klasy *Node* zwróconego jako wynik zapytania oraz zarządzanie cyklem życia tych obiektów.

Metody

- ```
  Object createProxy(
          Node node,
          Class desiredClass,
          TransparentSession session)
  ```

  Tworzy obiekt Javy reprezentujący obiekt w bazie danych wskazywany przez *node*. Obiekt bazodanowy jest odwzorowany na interfejs *desiredClass*. Powstały obiekt Javy jest podpięty do sesji *session*

- `Object createRootProxy(Node node, TransparentSession session)`

  Tworzy obiekt Javy reprezentujący obiekt w bazie danych wskazywany przez *node*. Obiekt bazodanowy musi znajdować się w korzeniu hierarchii obiektów. Interfejs powstałego obiektu Javy jest określany na podstawie nazwy obiektu bazodanowego. Powstały obiekt Javy jest podpięty do sesji *session*

- `boolean isProxy(Object object)`

  Zwraca *true* jeśli obiekt *object* jest obiektem trwałym zarządzanym przez dany komponent *TransparentProxyFactory*

- `Node getNodeOfProxy(Object object)`

  Zwraca wskaźnik do obiektu bazodanowego reprezentującego obiekt trwały *object*. Zgłasza wyjątek jeśli obiekt *object* nie jest obiektem trwałym.

A.3.6. pl.tzr.transparent.JavaTransparentProxyFactory

Implementacja klasy *TransparentProxyFactory* korzystająca z mechanizmu dynamicznych klas pośredniczących języka Java w celu tworzenia trwałych obiektów.

A.4. Warstwa przezroczystego odwzorowania – obiekty pośredniczące

A.4.1. pl.tzr.transparent.proxy.JavaTransparentProxyHandler

Klasa obsługująca wywołania metod trwałych obiektów Javy. Wywołanie metody na obiekcie trwałym jest przekazywane do jednej z metod tej klasy.

Konstruktory

- ```
  JavaTransparentProxyHandler(
          final Node node,
          final Class entityClass,
          final TransparentSession session)
  ```

  Tworzy obiekt obsługujący wywołania metod trwałego obiektu Javy reprezentującego obiekt w bazie wskazywany przez *node*. Obsługiwany trwały obiekt Javy będzie typu *entityClass* i będzie podpięty do sesji *session*.

Metody

- ```
  Object invokeGetter(
          Object transparentProxy,
          String propertyName,
          Class returnType)
  ```

  Operacja wykonywana w momencie wykonania wywołania metody pobierającej wartość atrybutu *propertyName* (*getXxx* lub *isXxx*) na obiekcie trwałym *transparentProxy*. Oczekiwany typ wyniku metody to *returnType*.

- ```
  Object invokeSetter(
          Object proxy,
          String propertyName,
          Object arg)
  ```

  Operacja wykonywana w momencie wykonania wywołania metody ustalającej wartość atrybutu *propertyName* (*setXxx*) na obiekcie trwałym *proxy*. Argument metody ustawiającej przekazywany jest jako *arg*.

- `Object invokeEquals(Object proxy, Object arg)`

  Operacja wykonywana w momencie wywołania metody *equals(arg)* na obiekcie trwałym *proxy*.
  ```

- `Object invokeToString(Object proxy)`

  Operacja wykonywana w momencie wywołania metody *toString()* na obiekcie trwałym *proxy*.

- `Object invokeHashCode(Object proxy)`

  Operacja wykonywana w momencie wywołania metody *hashCode()* na obiekcie trwałym *proxy*.

A.4.2. pl.tzr.transparent.proxy.handler.PropertyAccessor

Interfejs dla klas definiujących semantykę odczytu i zapisu atrybutów obiektów. Dla każdego możliwego typu atrybutu konieczna jest implementacja klasy *PropertyAccessor*. Narzędzie odwzorowujące posiada następujące implementacje tego interfejsu:

BooleanPropertyAccessor – obsługa atrybutów typu *Boolean*

ComponentPropertyAccessor – obsługa atrybutów złożonych odwzorowanych na obiekty składowe

IntegerPropertyAccessor – obsługa atrybutów typu *Integer*

ReferencePropertyAccessor – obsługa atrybutów złożonych odwzorowanych na referencje

StringPropertyAccessor – obsługa atrybutów typu *String*

PrimitiveSetAccessor – obsługa atrybutów typu zbiór (*java.util.Set*) obiektów prostych

ComponentSetAccessor – obsługa atrybutów typu zbiór obiektów złożonych odwzorowanych na obiekty składowe

ReferenceSetAccessor – obsługa atrybutów typu zbiór obiektów złożonych odwzorowanych na referencje

Możliwe jest zdefiniowane nowych typów atrybutów. Wystarczy stworzyć klasę implementującą interfejs *PropertyAccessor* i zarejestrować ją w używanym rejestrze dostępnych typów (obiekt klasy *AccessorRegistry*).

Metody

- ```
  T retrieveFromBase(
          Node parent,
          PropertyInfo propertyInfo,
          TransparentSession session)
          throws SBQLException, DeletedException
  ```

  Dla obiektu określonego przez *parent* pobiera wartość atrybutu określonego przez *propertyInfo*. Pobranie danych odbywa się w ramach sesji *session*. W przypadku, gdy obiekt *parent* nie istnieje, zgłasza wyjątek *DeletedException*

- ```
  void saveToBase(
          T data,
          Node parent,
          PropertyInfo propertyInfo,
  ```

```
TransparentSession session)
throws SBQLException, DeletedException
```

Dla obiektu określonego przez *parent* ustala wartość atrybutu określonego przez *propertyInfo* na *data*. Modyfikacja danych odbywa się w ramach sesji *session*. W przypadku, gdy obiekt *parent* nie istnieje, zgłasza wyjątek *DeletedException*

A.4.3. pl.tzr.transparent.proxy.handler.registry.AccessorRegistry

Obiekt przechowujący rejestr definicji typów atrybutów i sposobie odczytu/zapisu tych atrybutów (obiekty implementujące interfejs *PropertyAccessor*).

- `void registerHandler(Class clazz, PropertyAccessor handler)`

  Rejestruje sposób obsługi atrybutów typu *class*. Obiekt *handler* powinien zawierać metody odczytu/zapisu tego typu atrybutów.

- `boolean isHandlerAvailable(Class clazz)`

  Zwraca *true* jesli w rejestrze dostępny jest sposób obsługi atrybutów typu *clazz*.

- `PropertyAccessor getHanlder(Class clazz)`

  Zwraca obiekt zawierający metody obsługi atrybutów typu *clazz*.

A.4.4. pl.tzr.transparent.proxy.collection.PersistentSet

Nadklasa róznych implementacji standardowych zbiorów Javy (*java.util.Set*), odwzorowanych na bazę danych. Wraz z narzedziem odwzorowującym dostarczone są trzy implementacje zbiorów:

PersistentPrimitiveSet – Zbiór obiektów prostych (*Integer*, *Boolean*, *String*)

PersistentComponentSet – Zbiór obiektów złożonych odwzorowanych na obiekty składowe

PersistentReferenceSet – Zbiór obiektów złożonych odwzorowanych na referencje

Klasa abstrakcyjna *PersistentSet* oraz jej podklasy implementują wszystkie metody interfejsu *java.util.Set*

A.5. Definiowanie przezroczystego odwzorowania

A.5.1. pl.tzr.transparent.structure.model.ModelRegistry

Obiekty tej klasy przechowują informacje o modelu danych używanym przez aplikację.

Metody

- `void registerClass(ClassInfo classInfo)`

  Rejestruje definicję odwzorowania klasy zawartą w *classInfo*

- `ClassInfo getClassInfo(Class clazz)`

  Zwraca informację o sposobie odwzorowania klasy *clazz* zawartą w rejestrze

- `ClassInfo getClassInfo(String entityName)`

  Zwraca zawartą w rejestrze informację o sposobie odwzorowania klasy, której obiekty umieszczone w korzeniu bazy danych mają nazwę *entityName*.

- ```
  Node createNodeRepresentation(
          Object object,
          TransparentSession session)
  ```

  Zapisuje obiekt lokalny *object* ze wszystkimi jego atrybutami w bazie danych (w ramach sesji *session*), w korzeniu. Zwraca wskaźnik do utworzonego obiektu bazodanowego.

- ```
  Node createNodeRepresentation(
          Object object,
          String name,
          TransparentSession session)
  ```

  Zapisuje obiekt lokalny *object* ze wszystkimi jego atrybutami w bazie danych (w ramach sesji *session*), w korzeniu, pod nazwą *name* . Zwraca wskaźnik do utworzonego obiektu bazodanowego.

A.5.2. pl.tzr.transparent.structure.model.ClassInfo

Obiekty tej klasy zawierają informację o sposobie odwzorowania pewnej klasy Javy na obiekty w bazie danych.

Konstruktory

- ```
  ClassInfo(
          final String entityName,
          final Class clazz,
          final Map<String, PropertyInfo> properties
  )
  ```

  Tworzy obiekt z definicją odwzorowania klasy *clazz*. Odwzorowanie atrybutów określone jest przez mapę *properties* (kluczem jest nazwa atrybutu, wartością definicja odwzorowania). Obiekty klasy *clazz* zostaną odwzorowane na obiekty bazodanowe o nazwie *entityName*

Metody

- `String getEntityName()`

  Zwraca nazwę używaną obiekty LoXiM reprezentujące odwzorowywaną klasę.

- `Class getClazz()`

  Zwraca klasę modelu której dotyczy ta definicja.

- `PropertyInfo getPropertyInfo(String name)`

  Zwraca definicję odwzorowania atrybutu o nazwie *name*.

- `Collection<PropertyInfo> getProperties()`

  Zwraca kolekcję odwzorowań wszystkich odwzorowań atrybutów.
  ```

A.5.3. pl.tzr.transparent.structure.model.PropertyInfo

Obiekty tej klasy zawierają informację o sposobie odwzorowania pewnego atrybutu klasy na obiekty w bazie danych.

Konstruktory

- ```
  PropertyInfo(
          final String propertyName,
          final String nodeName,
          final Class clazz,
          final PropertyAccessor accessor
  )
  ```

  Tworzy obiekt z definicją atrybutu o nazwie *propertyName* typu *clazz*. Obiekty w bazie danych reprezentujące wartośc tego atrybutu będą miały nazwę *nodeName*. Za operacje odczytu/zapisu wartości atrybutu będzie odpowiedzialny obiekt *accessor*.

Metody

- `String getPropertyName()`

  Zwraca nazwę atrybutu

- `String getNodeName()`

  Zwraca nazwę używaną do nazywania obiektów w bazie danych reprezentujących wartości atrybutu.

- `Class getClazz()`

  Zwraca typ atrybutu

- `PropertyAccessor getAccessor()`

  Zwraca obiekt odpowiedzialny za zapis/odczyt wartości atrybutu

- `boolean isValueCacheable()`

  Zwraca *true*, jeśli wartości tego atrybutu mogą być buforowane przez warstwę przezroczystego odwzorowania.

A.5.4. pl.tzr.transparent.structure.model.CollectionPropertyInfo

Podklasa klasy *PropertyInfo* zawierająca informacje o sposobie odwzorowania atrybutu będącego kolekcją.

Konstruktory

- ```
  CollectionPropertyInfo(
          final String propertyName,
          final String nodeName,
          final Class clazz,
          final Class itemClass,
          final PropertyAccessor itemAccessor
  )
  ```

Tworzy obiekt z definicją atrybutu-kolekcji o nazwie *propertyName* typu *clazz*, mogącego zawierać elementy typu *itemClass*. Obiekty w bazie danych reprezentujące wartośc tego atrybutu będą miały nazwę *nodeName*. Za operacje odczytu/zapisu elementów atrybutu będzie odpowiedzialny obiekt *itemAccessor*.

Metody

- `Class getItemClass()`

  Zwraca typ elementów przechowywanych w kolekcji

A.5.5. pl.tzr.transparent.structure.model.ModelRegistryFactory

Wspólny interfejs dla klas mogących stworzyć obiekt typu *ModelRegistry* zawierający informacje o modelu danych używanym przez aplikacje.

Metody

- `ModelRegistry getModelRegistry()`

  Tworzy obiekt zawierający informacje o modelu danych

pl.tzr.transparent.structure.model.SimpleAnnotatedRegistryFactory

Implementacja interfejsu ModelRegistryFactory tworząca informacje o modelu danych na podstawie struktury zadanego zbioru klas modelu oraz adnotacji zawartych w kodzie tych klas. Analiza klas modelu odbywa się za pomocą mechanizmów refleksji języka Java. Do poprawnej klasy fabryki modelu potrzebne jest określenie klas należących do modelu (metoda *setClasses(classes[])*), oraz podanie komponentu zarządzającego dostępnymi typami danych (metoda *setHandlerRegistry(registry)*)

Metody

- `void setClasses(Class[] classes)`

  Ustala zbiór klas jakie mają należeć do modelu danych.

- `void setHandlerRegistry(AccessorRegistry registry)`

  Ustala komponent zarządzający dostępnymi typami danych, używany podczas pracy z modelem.

Dodatek B

Opis zawartości płyty dołączonej do pracy

Do pracy załączona została płyta CD zawierająca

- Treść pracy w formie elektronicznej (katalog `/praca`)
- Kod źródłowy bazy LoXiM w wersji używanej przez narzędzie odwzorowujące (katalog `/loxim`)
- Kod źródłowy narzędzia odwzorowującego (katalog `/mapper`)
- Testy automatyczne sprawdzające działanie narzędzia odwzorowującego
- Przykładowa aplikacja WWW opisana w punkcie 6.1 (katalog `/mapper-testapp`)
- Niezbędne biblioteki i narzędzia (katalog `/tools`)[1]
  - Apache Tomcat 5.5
  - Maven 2.0.6
  - Biblioteka Commons BeanUtils 1.1
  - Biblioteka Commons Logging 1.1
  - JUnit 4.3.1

Do kompilacji i uruchomienia bazy LoXiM potrzebne jest następujące środowisko:

- System operacyjny Linux
- Kompilator GNU C++
- Narzędzie GNU Make
- Narzędzie GNU Bison
- Narzędzie GNU Flex
- Narzędzie GNU Sed

[1] Wszystkie załączone narzędzia i biblioteki dostępne są zgodnie z ustaleniami licencji Apache License, Version 2.0 (JUnit w oparciu o licencję Common Public License, vol 1.0)

Do kompilacji i uruchomienia narzędzia odwzorowującego i aplikacji przykładowej konieczny jest system z dostępnym środowiskiem Java Developement Kit w wersji 1.5 lub nowszej (testowane 1.5 i 1.6).

Szczegółowe informacje o sposobie uruchomienia i instalacji dostarczonego oprogramowania znajdują się w plikach tekstowych *install.txt* znajdującymi się w katalogach z poszczególnymi modułami.

Bibliografia

[1] Code Generation Library. http://cglib.sourceforge.net/.

[2] ObjectWEB Asm. http://asm.objectweb.org/.

[3] System zarządzania bazami danych LoXiM. http://loxim.mimuw.edu.pl.

[4] Technologia Dynamic Proxy Classes. http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html.

[5] Technologia JavaBeans. http://java.sun.com/products/javabeans/.

[6] S. Crawley and M. Oudshoorn. Persistence extensions to ada, 1995.

[7] J. H. Jacobs and Mark R. Swanson. UCL+P - defining and implementing persistent Common Lisp. *Lisp and Symbolic Computation*, 10(1):5–38, 1997.

[8] Robert C. Martin. The dependency inversion principle. *C++ Report*, 8, May 1996.

[9] J.W. Schmidt and F. Matthes. The DBPL project: Advances in modular database programming. *Information Systems*, 19(2):121–140, 1994.

[10] K. Subieta. *Teoria i konstrukcja obiektowych języków zapytań*. Wydawnictwo PJWSTK, 2004.

[11] Piotr Tabor. Loxim - projekt protokołu komunikacyjnego klient-serwer.