Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Sebastian Kaczanowski
Nr albumu: 153407

Implementacja klas w systemie LoXiM

Praca magisterska
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem
dr. hab. Krzysztofa Stencela
Instytut Informatyki

Wrzesień 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

Niniejsza praca opisuje implementację klas w systemie zarządzania obiektową bazą danych LoXiM. System jest tworzony pod opieką promotora niniejszej pracy dr. hab. Krzysztofa Stencela na Wydziale Matematyki Informatyki i Mechaniki Uniwersytetu Warszawskiego w oparciu o podejście stosowe przedstawione w monografii Kazimierza Subiety [Sub04].

Implementacja polegała na rozszerzeniu istniejącego już interpretera SBQL o możliwości związane z obsługą klas. Oprócz tego obejmowała też rozszerzenie modelu składu danych z M0 do M1. Wynika to stąd, że klasy w podejściu stosowym są przechowywane jako zwykłe obiekty, podlegające takim operacją jak: usuwanie, dodawanie, czy zmienianie. Natomiast przejście z modelu M0 do modelu M1 wymaga dodania do modelu M0 informacji o dziedziczeniu dla klas i informacji o przynależności do klas dla obiektów. Implementacja nie obejmowała zagadnień związanych z pakietami, ani przestrzeniami nazw, opierała się jednak na założeniu unikatowości nazw klas.

Słowa kluczowe

system zarządzania bazą danych, podejście stosowe, klasa, dziedziczenie

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

Klasyfikacja tematyczna

H. Information Systems
H.2 DATABASE MANAGEMENT
H.2.3 Languages

Tytuł pracy w języku angielskim

Classes implementation in LoXiM system

Spis treści

Rozdział 1

Wprowadzenie

Niniejsza praca opisuje implementację klas w systemie zarządzania obiektową bazą danych LoXiM. System jest tworzony pod opieką promotora niniejszej pracy dr. hab. Krzysztofa Stencela na Wydziale Matematyki Informatyki i Mechaniki Uniwersytetu Warszawskiego w oparciu o podejście stosowe przedstawione w monografii Kazimierza Subiety [Sub04].

Implementacja polegała na rozszerzeniu istniejącego już interpretera SBQL o możliwości związane z obsługą klas. Oprócz tego obejmowała też rozszerzenie modelu składu danych z M0 do M1. Wynika to stąd, że klasy w podejściu stosowym są przechowywane jako zwykłe obiekty, podlegające takim operacją jak: usuwanie, dodawanie, czy zmienianie. Natomiast przejście z modelu M0 do modelu M1 wymaga dodania do modelu M0 informacji o dziedziczeniu dla klas i informacji o przynależności do klas dla obiektów. Implementacja nie obejmowała zagadnień związanych z pakietami, ani przestrzeniami nazw, opierała się jednak na założeniu unikatowości nazw klas.

1.1. Podejście stosowe

Model obiektowości będący konsekwencją tej implementacji ma kilka ciekawych własności. Przede wszystkim obiekty są bytami niezależnymi od klas, oznacza to, że mogą istnieć i mieć właściwości nie będąc jednocześnie instancją żadnej klasy. Dodatkowo obiekt może w dowolnej chwili zarówno stać się instancją klasy, jak i przestać nią być. Oprócz tego obiekt może być instancją wielu klas. Gdy zebrać te spostrzeżenia, to nasuwa się wniosek, że ten model wspiera dodawanie zobowiązań. Jest tak w istocie, mówi o tym podrozdział 5.6.4.

To, że obiekt jest niezależny od klasy, ma swoje ważne konsekwencje dla modelu wielokrotnego dziedziczenia. Inaczej niż w C++ tutaj klasa nie tworzy właściwości obiektu, czyli obiekt, który dziedziczy z dwóch klas nie musi mieć dla każdej klasy oddzielnego zestawu pól. Może mieć jedną właściwość współdzieloną między klasami, a jak zostało pokazane w podrozdziale 5.6.4 można w tym modelu również sprawić, żeby każda z klas miała oddzielny zestaw właściwości.

Samo podejście stosowe w oparciu, o które, jak już wspomniano, budowany jest LoXiM, jest, między innymi, bardzo ciekawym spojrzeniem na jedno z najpowszechniej występujących zagadnień z jakimi trzeba się zmierzyć implementując większość systemów informatycznych, chodzi oczywiście o problem trwałości. We wszystkich powszechnie używanych językach programowania ten problem jest uznawany za coś, co wymaga rozwiązań zewnętrznych wobec języka, co skutkuje tym, że tworzenie systemów wymagających utrwalania danych rozwarstwia się na dwie płaszczyzny technologiczne. Pierwszą z nich jest działanie systemu, za które odpowiedzialny jest język programowania, a drugą stanowi język zapytań odpowiedzialny za

zarządzanie danymi trwałymi. Rozwarstwienie to nie znika nawet, gdy używa się zrębów takich jak Hibernate, które wprawdzie łagodzą różnicę między paradygmatami obowiązującymi w składzie danych i w języku programowania, jednak nadal wymagają użycia języka zapytań, który jest niezależny od języka programowania. Podejście stosowe to propozycja, która pokazuje, że nie trzeba oddzielać od siebie języka programowania i języka zapytań i przedstawia mechanizm stosu środowisk jako abstrakcyjny model, za pomocą którego można opisać działanie obiektowego języka zapytań, który jednocześnie jest językiem programowania.

System LoXiM dysponuje już językiem, który jest językiem zapytań i językiem programowania, a o implementacji klas w tym systemie mówi ta praca.

1.2. Struktura pracy

W podrozdziale 1.3 zostały przedstawione podstawowe pojęcia i użyte konwencje. W rozdziale 2 można znaleźć informacje o tym jak został zmieniony model składu w związku z dodaniem klas, jakie konsekwencje wynikają z przyjętego rozwiązania i jakie rozwiązania alternatywne były rozważane. Następny rozdział (3) został poświęcony temu w jaki sposób klasa jest składowana w składzie obiektów. Zawiera on też opis zarządzania składowymi klasowymi. Kluczowa struktura w całej opisywanej tu implementacji, czyli graf klas, została opisana w rozdziale 4. Można tam znaleźć informacje o jej konstrukcji, wymieniono tam też podstawowe możliwości tej struktury oraz założenia, na których oparto się przy jej tworzeniu. Niewątpliwie najistotniejszą część pracy stanowi rozdział 5, w którym opisano semantykę i sposób realizacji wszystkich funkcjonalności, dodanych w prezentowanej tu implementacji, wraz z konstrukcjami składniowymi stworzonymi na ich potrzeby. Znajdują się tu również informacje o wadach i zaletach niektórych rozwiązań i sposobach ich wykorzystania. W rozdziale podsumowującym znalazły się refleksje dotyczące rezultatów pracy i propozycje dotyczące dalszych prac nad systemem LoXiM. Oprócz powyższego praca zawiera trzy dodatki: dodatek A opisujący zawartość płyty dołączonej do pracy, dodatek B zawierający przykład użycia funkcjonalności dodanych w opisywanej tu implementacji, dodatek C z niepublikowany maszynopisem Krzysztofa Stencela *Klasy w ODRA*, z którego korzysta się w tej pracy oraz załącznik w postaci płyty zawierającej system LoXiM w wersji, której dotyczy ta praca.

1.3. Podstawowe pojęcia

Najwięcej problemów, związanych z terminologią używaną w tej pracy, powoduje fakt, że opisuje ona implementację języka obiektowego w języku obiektowym. Ten problem dotyczy w największym stopniu pojęcia *klasa*, które może oznaczać klasę implementowanego języka lub klasę implementacji tego języka. W całości pracy przyjęto konwencję, według której termin *klasa* jest używany w odniesieniu do klas implementowanego tutaj języka SBQL, w odniesieniu do klas zawartych w implementacji jest używany termin *klasa implementacji*. W niektórych, szczególnie wrażliwych na to zamieszanie terminologiczne, częściach pracy używa się terminu *klasa SBQL*, żeby podkreślić, iż chodzi o klasę implementowanego języka. Analogicznie do terminu *klasa*, także inne terminy związane z nomenklaturą programowania obiektowego mogą zostać uzupełnione o słowo „implementacji" lub „SBQL".

1.3.1 podejście stosowe — *SBA (Stack Based Approach) to sposób przetwarzania danych, w którym zakres zmiennych i danych wejściowych, zależy od stanu stosu. Na przykład każda zmienna odnosi się do danych położonych wyżej na stosie i oznaczonych nazwą tej zmiennej. To pozwala zarządzać zakresem w prosty sposób i przetwarzać dane poprzez wkładanie środowisk*

na stosu i zdejmowanie ich ze stosu (Strona systemu LoXiM — http://loxim.mimuw.edu.pl/-pages/sbql.html) [LoXiM].

1.3.2 SBQL (Stack Based Query Language) — *Jak można przeczytać na stronie http://-sbql.pl/SBAoverview.3/SBA.3.htm [SBQL] podejście stosowe jest formalną metodologią opisu obiektowych języków zapytań i obiektowych języków programowania. Stąd też SBQL jest (wbrew nazwie) nie tylko językiem zapytań, ale też językiem programowania opartym na podejściu stosowym.*

1.3.3 M0, M1 — *Modele składu danych półstrukturalnych. Model M0 przypomina strukturą XML, model M1 wzbogaca model M0 o klasy*[1].

1.3.4 zastępowalność na poziomie kolekcji — *polega na tym, że jeśli obiekt jakiejś klasy K ma nazwę taką jak niezmiennicza nazwa obiektów tej klasy, to jest on też obiektem zwracanym w trakcie wykonywania zapytań, których celem jest pobranie obiektów o nazwach takich jak niezmiennicze nazwy obiektów nadklas klasy K. Innymi słowy taki obiekt dochowuje się tak jakby miał tyle nazw ile jest niezmienniczych nazw nadklas klasy, której jest instancją.*[2]

[1] Dokładny opis modeli M0 i M1 można znaleźć w [Sub04] s. 144 – 169
[2] Definicja na podstawie [ODRA]

Rozdział 2

Zmiany w sposobie składowania obiektów związane z dodaniem klas

Za zapisywanie różnego rodzaju artefaktów w pamięci trwałej w systemie LoXiM odpowiedzialny jest moduł o nazwie „Store" (na poziomie kodu źródłowego w C++ ten moduł jest katalogiem i jednocześnie przestrzenią nazw). Do zadań tego modułu należy również reprezentowanie obiektów składu w pamięci ulotnej, a więc w tym module znajdują się też klasy implementacji, które reprezentują te obiekty. Konkretnie mówiąc, obiekty składu są reprezentowane przez instancje klasy implementacji `Store::ObjectPointer`, której najważniejszymi składowymi są: identyfikator obiektu oraz instancja klasy implementacji `Store::DataValue`, która zawiera informacje o zawartości obiektu, a także o jego rodzaju i typie. W szczególności klasa implementacji `Store::DataValue` ma właściwość `subtype`, dzięki której można wyróżnić spośród składowanych elementów te, które są klasami.

Do tego by uzyskać model M1 trzeba było jeszcze składować informację o związkach dziedziczenia między klasami oraz o związkach dziedziczenia między obiektami. W przyjętym rozwiązaniu te relacje są zawarte w obiektach, które zostały odpowiednio w tym celu rozbudowane. Do klasy implementacji `Store::DBDataValue` dodano właściwość `classMarks`, która jest zbiorem (`std::hash_set`) identyfikatorów obiektów. Gdy obiekt jest klasą, `classMarks` jest interpretowana jako zbiór identyfikatorów jej nadklas, gdy jest to obiekt innego typu, ta właściwość jest rozumiana, jako zbiór identyfikatorów klas, których instancją jest dany obiekt.

Przedstawione wyżej rozwiązanie ma następujące konsekwencje:

- klasa nie może być instancją innej klasy,
- klasa nie może wielokrotnie bezpośrednio dziedziczyć po innej klasie,
- obiekt może być instancją wielu klas,
- instancjami klas mogą być obiekty atomowe i wskaźniki.

Z punktu widzenia specyfikacji, `classMarks` wystarcza do utworzenia modelu składu odpowiadającego modelowi M1, jednak z przyczyn związanych z wydajnością zostały dodane dwa ważne usprawnienia.

Pierwsze z tych usprawnień, to dodanie do klasy implementacji `Store::DBDataValue` właściwości `subclasses`, która jest wykorzystywana jedynie przez klasy. To pole zawiera zbiór identyfikatorów podklas danej klasy. W chwili obecnej[1] jest wykorzystywane przy zmienianiu

[1] Gdy system LoXiM będzie miał możliwość kompilowania kodu, będzie można wykorzystać `subclasses` do tego, by podklasy zmienianej lub usuwanej klasy oznaczać jako przeznaczone do re-kompilacji.

i usuwaniu klasy, do tego by uaktualnić właściwość `classMarks` podklas. Oczywiście można by przeprowadzić taką operację wykorzystując w tym celu jedynie `classMarks`, jednak trzeba by w takim wypadku przejrzeć wszystkie przechowywane aktualnie obiekty.

Drugim usprawnieniem jest specjalny plik, w którym przechowywane są podstawowe informacje o wszystkich klasach jakie zostały dodane do składu. Te informacje to:

- identyfikator klasy,
- unikatowa nazwa klasy,
- niezmiennicza nazwa obiektów klasy.

Plik ten jest obsługiwany przez klasę `Store::Classes`, która udostępnia możliwość wyszukania identyfikatorów klas o zadanej nazwie lub o zadanej niezmienniczej nazwie obiektów. Obecnie z pliku tego korzysta się wyłącznie przy budowaniu grafu klas, co pozwala, przy tej okazji, uniknąć konieczności przeglądania wszystkich składowanych obiektów. Znaczenie tego usprawnienia niewątpliwie wzrośnie, gdy zaistnieje potrzeba leniwego ładowania klas lub gdy trzeba będzie zmniejszyć zużycie pamięci usuwając z niej aktualnie nieużywane klasy.

Przedstawiona w tym rozdziale implementacja nie była jedynym pomysłem jaki brano pod uwagę przy przekształcaniu modelu składu do modelu M1. Innym rozwiązaniem, którego wdrożenie rozważano, był pomysł, ażeby trzymać dane o dziedziczeniu jako zwyczajne podobiekty obiektów, których te dane dotyczyły. Ważną zaletą takiego podejścia byłaby łatwa implementacja. W szczególności nie byłyby konieczne żadne zmiany w klasie implementacji `Store::DBDataValue`. Jednak system LoXiM z tak wprowadzonymi klasami miałby poważną wadę, mianowicie obiekty atomowe (takie jak: napis, liczba całkowita, liczba rzeczywista itp.) i wskaźniki nie mogłyby być instancjami klas, ponieważ obiekty tego typu nie mogą zawierać podobiektów. To zdecydowało o odrzuceniu tego rozwiązania. Nie było poważnie rozważane rozwiązanie, w którym dane o dziedziczeniu są przechowywane niezależnie od obiektów. Jego implementacja byłaby bardzo trudna, a jego właściwości wydają się nieatrakcyjne. Już choćby to, że pobranie obiektu nie wystarcza do tego, żeby mieć wszystkie potrzebne informacje o nim (bo nie wiadomo od razu, czy jest instancją jakiejś klasy) wydaje się być wystarczająco zniechęcające.

Rozdział 3

Struktura obiektu klasowego

Jak wspomniano już wcześniej, klasy są składowane dokładnie tak samo jak obiekty innego rodzaju (np. metody, perspektywy, liczby całkowite, napisy, wskaźniki itp.). Poniższy rozdział opisuje strukturę obiektu, w którym przechowywana jest klasa.

W klasie przechowywane są następujące składowe:

- niezmiennicza nazwa obiektów,
- nazwy właściwości instancyjnych,
- wskaźniki do metod instancyjnych,
- nazwy właściwości klasowych,
- wskaźniki do metod klasowych.

Fakt, iż składowane w LoXiM obiekty mają nazwy, został wykorzystany i nazwa składowanego obiektu jest jednocześnie nazwą klasy. Takie rozwiązanie jest do przyjęcia, póki w systemie nie ma pakietów, gdy będą one dodawane, może się okazać, iż również nazwę wygodnie będzie składować w obiekcie.

Żadna z wymienianych składowych nie musi wystąpić, w szczególności klasa może być pusta. Niezmiennicza nazwa obiektu może wystąpić najwyżej raz. Nazwy pól instancyjnych są jedynie po to, by ułatwić dodanie do systemu kontroli typów i obecnie nie są do niczego wykorzystywane, dba się jedynie, by nazwy były unikatowe. Wskaźniki do metod instancyjnych są domyślnie[1] jedynym sposobem na odczytanie zawartości tych metod, gdyż metody instancyjne są tworzone jako lokalne obiekty, czyli wskaźniki do nich nie są umieszczane wśród identyfikatorów startowych. Samo tworzenie metod instancyjnych i klasowych przebiega dokładnie tak samo, jak tworzenie procedur. W przeciwieństwie do metod instancyjnych, wskaźniki do metod statycznych nie są domyślnie jedyną drogą odnalezienia ich, dlatego że tego rodzaju metody są umieszczane w zbiorze identyfikatorów startowych. Wskaźniki te wykorzystuje się do tego, by móc usunąć wskazywane przez nie metody klasowe w momencie gdy klasa jest usuwana lub zmieniana. Warto dodać, że metody klasowe są tworzone w momencie utworzenia klasy i nie można ich utworzyć kiedy indziej. Składowe klasowe (metody i pola) różnią się od innych obiektów składu nazwą. Ich nazwy - w stosunku, do których w dalszej części, będzie używany termin: nazwa rozszerzona - poprzedzane są nazwą klasy i separatorem złożonym ze znaków, które nie mogą wchodzić w skład normalnej nazwy (obecnie: ::). Ten fakt pomógł zapobiec temu, by składowe statyczne były umieszczane poza zbiorem identyfikatorów

[1] Ponieważ jednak klasa to zwyczajny obiekt istnieje możliwość umieszczenia wskaźników do jej metod poza nią.

startowych. Informacja o nazwach pól klasowych pozwala kontrolować tworzenie obiektów o nazwach rozszerzonych, tak by można było tworzyć jedynie obiekty o nazwach odpowiednich dla pól statycznych klas. Same pola klasowe mogą zostać zainicjowane dopiero po stworzeniu klasy. Dostęp do zbioru pól statycznych jest potrzebny również podczas zmieniania i usuwania klasy. W trakcie tych operacji wszystkie pola statyczne klasy są usuwane.

Same klasy tworzone są domyślnie jako identyfikatory startowe ale oczywiście można je wstawiać do innych obiektów. Jednak w takiej sytuacji należy pamiętać, że nazwa klasy pozostaje ta sama, w związku z czym nadal nie będzie można utworzyć klasy o nazwie identycznej z tą, której identyfikator przestał być identyfikatorem startowy.

Poniższy przykład pokazuje zawartość przykładowej klasy.

Przykład 3.0.1 *Załóżmy, że do składu obiektów dodano klasę, za pomocą następującego zapytania*[2]*:*

```
create class Przyklad {
  instance przyklad : {x; y}
  static {X; Y}

  static procedure d(y) { y := Y }
  procedure getA() { return a}
  procedure setA(_a) { a := _a }
}
```

W celu wyświetlenia struktury klasy można wykonać zapytanie: `deref(Przyklad)`*, którego wynik będzie następujący:*

```
< invariant(przyklad)
field(x) field(y)
sfield(X) sfield(Y)
method(ref(31)) method(ref(35))
smethod(ref(39)) >
```

Jak widać przechowywane są nazwy właściwości instancyjnych w polach `field`*, nazwy właściwości klasowych w polach* `sfield`*, niezmiennicza nazwa obiektów klasy w polu* `invariant`*, wskaźniki do metod instancyjnych w polach* `method` *i metod klasowych w polach* `smethod`*.*

Trzeba w tym miejscu zastrzec, że powyższa struktura może się zmieniać. Nazwy pól powinny zostać skrócone, gdy wydajność zacznie odgrywać w LoXiMie kluczową rolę, póki co czytelność jest może nawet ważniejsza niż wydajność.

[2]Informacje o składni tworzenia klas można znaleźć w rozdziale 5.

Rozdział 4

Graf klas

W opisywanej tutaj implementacji klas SBQL w systemie LoXiM, graf klas SBQL (nazywany tutaj po prostu „grafem klas") jest strukturą danych stworzoną po to, żeby wspomagać i umożliwiać operacje na klasach SBQL takie jak ich dodawanie, zmienianie i usuwanie oraz operacje wykonywane z użyciem informacji zawartych w klasach SBQL. Dobrymi przykładami operacji tego drugiego rodzaju są: wywołanie metody, podmienianie na poziomie kolekcji, rzutowanie.

Ten rozdział opisuje jak graf klas został skonstruowany, co zawiera, jak jest przygotowywany do użycia i w jaki sposób jest modyfikowany.

4.1. Założenia

Graf klas został zaimplementowany w oparciu o wymienione poniżej założenia.

1. Graf klas jest jeden współdzielony między sesjami użytkowników.
2. Graf klas jest ładowany przy starcie serwera.
3. Graf klas jest modyfikowany przy każdej operacji na klasach.
4. Nie jest brane pod uwagę bezpieczeństwo transakcyjne operacji na grafie klas. Założono, że są one rzadkie i że, w warunkach produkcyjnych, nie będzie ich wykonywać więcej niż jedna osoba. To co w tym zakresie graf klas zapewnia to to, że jeśli kod klas SBQL zostanie wykonany bez błędów i transakcja, w której wykonywano ten kod zostanie zatwierdzona, to będzie istniała pełna zgodność między tym co zawiera skład obiektów i tym co zawiera graf klas. Taką pewność daje również przebudowanie grafu klas. Odnosząc to do cech transakcji należy stwierdzić, że graf klas:
   - nie zachowuje atomowości transakcji, gdyż nie można wycofać operacji wykonanej na grafie klas (można co najwyżej przebudować graf klas), więc jeśli klasa SBQL zostanie dodana w jakiejś transakcji, która zostanie potem wycofana, w grafie klas ta klasa pozostanie aż do momentu przebudowania grafu klas;
   - zachowuje spójność transakcji, bo wykonanie transakcji nie może uszkodzić grafu klas;
   - nie jest zachowana izolacja transakcji, wszystkie transakcje widzą od razu wprowadzane przez siebie zmiany w grafie klas;
   - jest zachowana trwałość transakcji, po ponownym uruchomieniu systemu zawartość grafu klas jest zgodna z zawartością składu.

4.2. Podstawowe klasy implementacji grafu klas

Celem tego podrozdziału nie jest szczegółowy opis metod i pól wszystkich klas implementacji, to zadanie spełnia dołączona do kodu źródłowego dokumentacja. Ma on (podrozdział) służyć przedstawieniu cech grafu klas koniecznych do zrozumienia jego roli w implementacji podstawowych funkcjonalności związanych z działaniem klas SBQL w systemie LoXiM.

Generalna uwaga dotycząca słowników i zbiorów we wszystkich klasach implementacji grafu klas jest taka, że są to struktury nieuporządkowane wykorzystujące haszowanie, czyli w typowej sytuacji zapewniające stały czas wyszukiwania.

4.2.1. QExecutor::Method

Ta klasa implementacji jest prostym „pojemnikiem" na metodę SBQL. Zawiera identyfikator metody SBQL, nazwy jej parametrów i kod w postaci napisu.

W czasie ładowania klasy SBQL do grafu klas, wszystkie jej metody są ładowane w postaci instancji klasy implementacji `Method`.

4.2.2. QExecutor::ClassGraphVertex

Ta klasa implementacji reprezentuje wierzchołek grafu klas, a więc zawiera informacje o klasie SBQL. Znajdujemy tu pola jednoznacznie kojarzące się, z tym co przedstawiono w rozdziale 3, czyli:

- `extends` — zbiór identyfikatorów klas SBQL, z których klasa SBQL dziedziczy,
- `subclasses` — zbiór identyfikatorów klas SBQL, które dziedziczą z klasy SBQL,
- `fields` — zbiór nazw pól instancyjnych klasy SBQL,
- `staticFields` — zbiór nazw pól statycznych klasy SBQL,
- `invariant` — niezmiennicza nazwa obiektów klasy SBQL,
- `name` — nazwa klasy SBQL.

Rola tych pól została opisana w rozdziale 3. Brakuje w tej liście spisów metod SBQL, a to z tego względu, że nie ma potrzeby przechowywania w grafie klas żadnych informacji o metodach klasowych, a co do metod instancyjnych, to właściwość, która je przechowuje, jest tutaj używana nie tylko do celów opisanych w rozdziale 3.

Struktura danych, w której przechowywane są metody instancyjne, czyli typ pola `methods`, to słownik, którego kluczami są nazwy metod, a wartościami są słowniki, których z kolei kluczami są ilości argumentów metod, a wartościami obiekty typu `Method`. Oto jak wygląda deklaracja typu `NameToArgCountToMethodMap` (szczegóły techniczne zostały pominięte):

```
typedef hash_map<unsigned int, Method*> ArgsCountToMethodMap;
typedef hash_map<string, ArgsCountToMethodMap*, ...> NameToArgCountToMethodMap;
```

Główną zaletą takiej struktury jest możliwość szybkiego sprawdzenia, czy klasa SBQL ma metodę o podanej nazwie z podaną ilością argumentów i dostarczenia tej metody. Do tego służy metoda `int getMethod(const string& name, unsigned int argsCount, Method*& method, bool& found)`, która na podstawie nazwy metody (`name`) i ilości argumentów (`argsCount`), podaje informację o tym czy taka metoda jest (`found`), zwracając ją jednocześnie (`method`).

Inną ważną funkcjonalnością dostarczaną w opisywanej tu klasie implementacji, jest możliwość zainicjowania obiektu tej klasy implementacji na podstawie obiektu typu `ObjectPointer`, który, mówiąc ogólnie, stanowi reprezentację obiektu przechowywanego w bazie danych. W ten sposób nie są inicjowane tylko pola `extends` i `subclasses`, za ich inicjację odpowiada klasa implementacji `ClassGraph` opisana w podrozdziale 4.2.3.

4.2.3. QExecutor::ClassGraph

Podstawową właściwością tej klasy implementacji jest właściwość `classGraph`, czyli słownik wierzchołków grafu (instancji `ClassGraphVertex`), którego klucze to identyfikatory obiektów, w których składowane są klasy SBQL. Słownik ten zawiera wszystkie klasy SBQL. Mimo swej nazwy nie jest on oczywiście grafem, graf dziedziczenia uzyskuje się dopiero przez połączenie tego słownika ze zbiorami identyfikatorów `extends` i `subclasses`, które znajdują się w wierzchołkach grafu (klasa implementacji `ClassGraphVertex` opisana w rozdziale 4.2.2).

Poruszanie się po grafie dziedziczenia wygląda następująco. Zaczyna się zawsze od identyfikatora jakiejś klasy SBQL i wyszukuje się w `classGraph` instancji `ClassGraphVertex`, która mu odpowiada. W tej instancji na podstawie zbioru `extends` albo `subclasses` wyszukujemy kolejnych wierzchołków grafu w `classGraph`. Teraz bierzemy z kolei ich właściwości `extends` albo `subclasses` i postępujemy jak poprzednio i tak dalej, aż zbiór nadklas SBQL albo podklas SBQL dla danego wierzchołka okaże się pusty.

Do tego, by podany wyżej algorytm przechodzenia grafu zawsze się zatrzymywał, potrzebne jest, aby był zachowany niezmiennik, polegający na tym, że nie zachodzi taka sytuacja, w której jakaś klasa SBQL dziedziczy po swojej podklasie SBQL. Oczywiście usunięcie klasy SBQL nie zagraża temu niezmiennikowi. Łatwo też zauważyć, że dodanie klasy SBQL nie stanowi problemu, ponieważ żadna z istniejących klas SBQL po niej nie dziedziczy, co wystarczy, by niezmiennik był zachowany. Jedyny problem stanowi modyfikacja klasy SBQL, gdyż może ona polegać między innymi na tym, że zmieni się zestaw klas SBQL, z których dana klasa SBQL dziedziczy i wtedy może nastąpić próba dziedziczenia po swojej podklasie SBQL. Dlatego też przed dokonaniem modyfikacji klasy SBQL należy się przekonać, czy taka modyfikacja jest możliwa. Służy do tego metoda implementacji `checkExtendsForUpdate`.

Dwie inne właściwości tej klasy implementacji: `invariants` i `nameIndex`, mają znaczenie z punktu widzenia wydajności. Znacznie poprawiają szybkość działania niektórych operacji, kosztem stosunkowo niedużego zużycia pamięci. Pole `invariants` jest słownikiem zbiorów identyfikatorów klas SBQL, którego klucze to niezmiennicze nazwy obiektów tych klas. Ta struktura umożliwia wyszukiwanie klas na podstawie ich niezmienniczych nazw obiektów w czasie $O(1)$. `nameIndex` jest odwzorowaniem (słownikiem) z nazwy klasy w jej identyfikator, co umożliwia wyszukiwanie klas po nazwach w czasie $O(1)$.

Taka konstrukcja grafu klas — czyli połączenie klas implementacji: `Method`, `ClassGraphVertex`, `ClassGraph` — umożliwiła wydajną realizację, między innymi, wymienionych poniżej funkcjonalności.

- Dodawanie klasy SBQL do grafu klas — metoda `addClass`.
- Usuwanie klasy SBQL z grafu klas — metoda `removeClass`.
- Modyfikacja klasy SBQL w grafie — metoda `updateClass`.
- Odnajdowanie na podstawie niezmienniczej nazwy obiektów klasy SBQL (oznaczmy tę nazwę przez n) wszystkich niezmienniczych nazw podklas SBQL i nadklas SBQL, tych klas SBQL, których niezmienniczą nazwą obiektów jest n. Odpowiadają za to metody implementacji: `fetchSubInvariantNames` i `fetchExtInvariantNames`.

- Wyszukanie wszystkich niezmienniczych nazw obiektów nadklas SBQL danej klasy SBQL — metoda fetchExtInvariantNamesForLid.
- Wyszukiwanie metody instancyjnej SBQL — `findMethod`.
- Sprawdzanie czy pole klasowe o podanej nazwie istnieje — `staticFieldExist`.
- Sprawdzenie czy rzutowanie jest możliwe — `isCastAllowed`.
- Wymieniane już sprawdzenie, czy można dokonać modyfikacji danej klasy SBQL — metoda `checkExtendsForUpdate`.

4.3. Inicjowanie grafu klas

Do inicjowania grafu klas służy bezparametrowa metoda klasowa `ClassGraph::init`. Zakłada ona, że został zainicjowany menadżer transakcji. Działanie metody `ClassGraph::init` polega na tym, że pobiera ona wszystkie klasy SBQL ze składu i na tej podstawie buduje graf klas, po czym zapisuje tak powstałą instancję klasy `ClassGraph` na zmiennej klasowej `ClassGraph::handle`, do pobrania której służy metoda `int getHandle(ClassGraph*& cg)`. Niszczenie grafu klas wymaga wywołania metody klasowej `void shutdown()`.

Rozdział 5

Składnia, semantyka i realizacja kluczowych funkcjonalności

W tym rozdziale zostaną opisane wszystkie kluczowe funkcjonalność zrealizowane podczas implementacji opisywanej w tej pracy realizacji klas w systemie LoXiM. Ten opis będzie się koncentrował głównie, na tym co zostało dodane. Oznacza to między innymi, że konstrukcje składniowe, które istniały przed stworzeniem opisywanej tu implementacji, będą traktowane jako już znane i nie wymagające opisu.

Wszystkie formuły składniowe zapisywane są według następującego schematu:

- elementy należące do języka SBQL są zapisywane kursywą, przy czym:
  - małymi literami zapisuje się słowa kluczowe,
  - dużymi literami zapisuje się konstrukcje i całostki składniowe (jak listy nazw, czy nazwy),
- czcionką o stałej szerokości zapisuje się te znaki, które opisują cechy składniowe:
  - nawiasy zwykłe oznaczają grupowanie,
  - nawiasy kwadratowe oznaczają opcjonalność,
  - symbol * oznacza opcjonalność i możliwość powtórzenia danej konstrukcji wiele razy,
  - symbol | oddziela elementy zbioru, spośród których wybiera się jeden.

5.1. Tworzenie klasy

Składnię tworzenia klas pokazuje poniższa formuła, w której należy na początku użyć słowa kluczowego „create" lub "create or update", jeśli klasa o podanej nazwie nie istnieje.

5.1.1 *Składnia tworzenia i zmieniania klas.*

(*create* | *update* | *create or update*) *class NC* [*extends LNRK*] *{*
  [*instance* [*NNO*] *: {* [*LNWI*] *}*]
  [*static { LNWK }*]
  ([*static*] *DP*)*
}

Skróty mają następujące znaczenie:

- *NC — nazwa tworzonej klasy,*
- *LNRK — lista nazw rozszerzanych,*
- *NNO — niezmiennicza nazwa obiektów klasy,*
- *LNWI — lista nazw właściwości instancyjnych,*
- *LNWS — lista nazw właściwości klasowych,*
- *DP — definicja procedury.*

Zanim klasa zostanie dodana następuje sprawdzenie, czy klasa o takiej nazwie już istnieje, jeśli tak proces dodawania kończy się z błędem. Podczas dodawania klasy, poza czynnościami opisanymi w rozdziale 3 związanymi z dodawaniem obiektu składu, który zawiera klasę, wykonywane są czynności związane z dodaniem klasy do grafu klas. Te czynności to dodanie nowego wierzchołka w grafie klas i powiązanie go z wierzchołkami klas, po których nowo utworzona klasa dziedziczy.

Po wykonaniu tych czynności można używać klasy od razu we wszystkich sesjach, które są aktywne.

5.2. Usuwanie klasy

Na poziomie składniowym klasy usuwa się tak samo jak normalne obiekty (poleceniem „delete"). Jednak, jak już wspomniano w rozdziale 3, podczas usuwania klasy, oprócz jej samej, usuwane są również jej składowe klasowe. Konieczna jest też aktualizacja grafu klas, polegająca na usunięciu wierzchołka usuwanej klasy i wszystkich powiązań, które do niego prowadzą, czyli tych z podklas i tych z nadklas. Nie usuwane są natomiast powiązania między usuwaną klasą a obiektami, głównie ze względu na to, że takie rozwiązanie wymagałoby przejrzenia wszystkich obiektów. Zadbano jednak o to, żeby obiekty, w których mamy do czynienia z powiązaniami z klasami, które nie istnieją, zachowywały się tak jak obiekty bez tych powiązań.

5.3. Modyfikacja klasy

Składnię używaną do modyfikacji klasy przedstawia formuła 5.1.1, z zastrzeżeniem, że na początku powinno znaleźć się słowo kluczowe „update", choć modyfikacja nastąpi również gdy na początku wystąpi „create or update" i klasa o nazwie „NC" istnieje.

Modyfikacja klasy polega na jej usunięciu (tak jak to opisano w rozdziale 5.2), dodaniu (opis w rozdziale 5.1) ale w taki sposób, by nie zmienił się identyfikator klasy i by spis jej podklas się nie zmienił. Sprawdza się też, czy klasa nie dziedziczy po swoich podklasach i jeśli tak jest procedura modyfikacji klasy kończy się błędem. Po dodaniu klasy odtwarza się jej połączenia z podklasami w grafie klas.

Po tej operacji obiekty, które przedtem były instancjami modyfikowanej klasy są nimi nadal, podobnie klasy, które dziedziczyły po modyfikowanej klasie, nadal po niej dziedziczą. Tak jak w wypadku operacji dodawania oraz usuwania zmiany są widoczne od razu we wszystkich aktualnych sesjach.

5.4. Zastępowalność na poziomie kolekcji

Do realizacji zastępowalności na poziomie kolekcji wystarczy jeśli: „dowolny binder $n(i)$ wynikający ze zbioru R identyfikatorów startowych jest uzupełniany poprzez bindery $n1(i)$, $n2(i)$, ..., gdzie $n1$, $n2$,... są nazwami ustalonymi jako inwarianty klas nadrzędnych w stosunku do klasy, do której należy obiekt i"[1]. Algorytm, który się narzuca jako rozwiązanie tego problemu, to dodanie do składu obiektów tylu nowych binderów ile potrzeba, przy każdym wskazaniu, że obiekt jest instancją klasy. Jest to jednak rozwiązanie mało wydajne i komplikujące takie operacje jak usuwanie obiektu, czy wykluczanie obiektu ze zbioru instancji klasy.

Inne rozwiązanie zaproponowano w artykule „Klasy w odra" ([ODRA]). Tam bindery, potrzebne do poprawnego działania zastępowalności na poziomie kolekcji, są tworzone w momencie, gdy obiekt jest wkładany na stos środowisk. W systemie LoXiM ta strategia nie mogła być zastosowana do wszystkich sekcji stosu środowisk, dlatego że sekcja bazy danych nie jest wkładana na stos środowisk, jednak jest ona (strategia) wykorzystywana we wszystkich sekcjach powyżej sekcji bazy danych.

Realizacja tej strategii w LoXiM przebiegła w ten sposób, że zmodyfikowano procedurę wkładania na stos środowisk (czyli metodę `QExecutor::EnvironmentStack::push`), tak by bindery były wkładane wraz ze związanymi z nimi binderami wymaganymi z punktu widzenia zastępowalności na poziomie klas. Właśnie do realizacji tego zadania została stworzona metoda grafu klas `fetchExtInvariantNamesForLid`, która na podstawie identyfikatora obiektu, dostarcza wszystkie wymagane tutaj nazwy.

Co do sekcji bazy danych to wprawdzie nie jest ona wkładana na stos środowisk, ale dostępne są metody wyszukiwania umieszczonych tam binderów po nazwach. Dodatkowo łatwo zauważyć, że jeśli następuje wyszukiwanie według zasady zastępowalności na poziomie kolekcji, to najpierw należy wyszukać wszystkich obiektów normalnie, a potem, jeśli szukana nazwa okaże się niezmienniczą nazwą jakiejś klasy K, trzeba wyszukać wszystkich obiektów o nazwach identycznych z niezmienniczymi nazwami podklas klasy K i spośród nich wykluczyć te obiekty, które mają odpowiednie nazwy, ale nie są instancjami odpowiednich klas. Obie te rzeczy pozwoliły stworzyć algorytm wyszukiwania w sekcji bazy danych.

W celu wdrożenia tego algorytmu należało zmodyfikować metodę `EnvironmentStack::bindName` i do wyników zwykłego wyszukiwania w sekcji bazy danych dodać wyniki uzyskane w opisany wyżej sposób. Do tego celu posłużyły dwie metody grafu klas. Metoda `fetchSubInvariantNames` na podstawie wyszukiwanej nazwy N dostarcza zbiór niezmienniczych nazw obiektów klas, które dziedziczą z klas o niezmienniczej nazwie obiektu N, przy czym samo N jest eliminowane z tego zbioru w celu uniknięcia pobrania jednego obiektu wiele razy. Zbiór uzyskany za pomocą tej metody jest podstawą następnych wyszukiwań w sekcji bazy danych. Efekty tych wyszukiwań są kumulowane a następnie filtrowane za pomocą metody `belongsToInvariant` z grafu klas. Metoda ta, na podstawie nazwy $N2$ i identyfikatora, sprawdza, czy podany identyfikator wskazuje na obiekt, który jest instancją podklas jakiejś klasy o niezmienniczej nazwie $N2$ i czy nazwa tego obiektu jest taka sama jak niezmiennicza nazwa obiektów tej podklasy. Wszystkie obiekty, które pomyślnie przejdą test przeprowadzony za pomocą `belongsToInvariant` są dodawane do ostatecznego wyniku wyszukiwania.

[1] [Sub04] s. 249

5.5. Wywołanie metod

W podejściu stosowy, po to żeby znaleźć się w kontekście jakiegoś obiektu, należy włożyć jego zawartość na stos środowisk. Do tego służy operacja *nested*[2]. Jednak metody instancyjne nie znajdują się w obiekcie tylko w klasie, której instancją jest obiekt. Oznacza to, że wprowadzenie klas wymaga takiej modyfikacji *nested*, by w kontekście obiektu znalazły się metody instancyjne.

Oto sposób w jaki można zmodyfikować *nested*. Jeżeli przetwarzany jest „obiekt z identyfikatorem i należący do klasy $K1$, która ma kolejne nadklasy $K2$, $K3$, ..., wówczas na wierzchołek stosu środowisk wkłada się po kolei, poczynając od wierzchołka: $nested(i)$, bindery do właściwości $K1$, bindery do właściwości $K2$, bindery do właściwości $K3$ itd."[3].

Taka modyfikacja byłaby bardzo niewydajna, dlatego też w LoXiM zastosowano inne rozwiązanie. Zamiast wkładania na stos środowisk klas, do każdej sekcji stosu środowisk dodano informację o klasach instancyjnych związanych z tą sekcją (czyli z obiektem, na którym wykonano *nested*). Do tego służy właściwość `classesPerSection` w klasie implementacji `QExecutor::EnvironmentStack`.

Oprócz tego zmieniono też algorytm wyszukiwania nazw procedur, dodając do niego wyszukiwanie metod w klasach związanych z sekcją, która jest aktualnie przetwarzana w poszukiwaniu nazwy procedury. Zmiany te dotyczyły metody `bindProcedureName` w klasie implementacji `QExecutor::EnvironmentStack`. Oczywiście wyszukiwanie metody ma miejsce, tylko wtedy gdy nie znaleziono wcześniej procedury[4]. Za samo wyszukiwanie metod w odpowiedniej kolejności — czyli począwszy od podklas, przechodząc do nadklas — odpowiada metoda grafu klas `findMethod`.

Taka implementacja zapewnia działanie wywoływania metod zgodnie z paradygmatem programowania obiektowego, czyli umożliwia przedefiniowywanie metod w podklasach. Co do składni wywoływania metod instancyjnych na obiektach, to wygląda ona tak samo jak wywoływanie procedur, które obiekt zawiera[5]. Wywołanie metody instancyjnej z wnętrza innej metody instancyjnej wygląda tak samo jak wywołanie procedury.

5.5.1. self

Modyfikacja *nested* została też wzbogacona o operację wstawienia do sekcji, do której są wstawiane informacje o aktualnym obiekcie, bindera o nazwie `self` z identyfikatorem tego obiektu. Dzięki temu metody mają dostęp do identyfikatora obiektu, na którym operują. Ten dostęp odbywa się dokładnie tak jakby obiekt miał podobiekt o nazwie `self`.

5.5.2. Wywoływanie metod z nadklas

Ponieważ w realizowanym w tej implementacji modelu składu M1 mamy do czynienia z wielokrotnym dziedziczeniem, czyli też z potencjalnymi konfliktami nazw metod, trzeba było dać możliwość radzenia sobie z tym problemem. Dodatkowo ewentualne rozwiązanie musiało być zgodne z podejściem stosowym.

Pomysł tu wykorzystany polegał na użyciu nazw rozszerzonych[6] i składniowo jest identyczny z rozwiązaniem przyjętym w C++, gdzie wywołanie bezparametrowej metody o nazwie `metoda` z nadklasy o nazwie `K` zapisuje się tak `K::metoda()`. Idea jego realizacji zgodnej z

[2]Dokładny opis *nested* można znaleźć w [Sub04] s. 190 – 192
[3][Sub04] s. 249
[4]Obiekty mogą zawierać procedury, gdyż te są zwyczajnymi obiektami.
[5]Czyli np. tak: `obiekt.metoda()`.
[6]Termin wprowadzony w rozdziale 3.

podejściem stosowym i przy założeniu, że *nested* działa wkładając na stos zawartość klas, a nie z wykorzystanie grafu klas[7], polega na tym, by oprócz normalnych nazw metod wkładać ich nazwy rozszerzone. Do tego, żeby taki mechanizm działał sensownie, czyli nie pozwalał wywoływać metod z nadklas, po których klasa nie dziedziczy, konieczne byłoby stosowanie przysłaniania sekcji stosu środowisk. Trzebaby zapamiętywać, z której sekcji pochodzi wykonywana aktualnie metoda i przy wywołaniu metody za pomocą nazwy rozszerzonej przysłaniać te sekcje stosu środowisk z klasami przetwarzanego aktualnie obiektu, które nie są nadklasami klasy, której metody są umieszczone w zapamiętanej sekcji. Przysłanianie było już wykorzystywane w LoXiM przy realizacji procedur, a co za tym idzie również metod. Najogólniej mówiąc, przesłanianie polega na tym, że w pewnych momentach niektóre sekcje stosu środowisk nie są widoczne.

Jak widać wykorzystanie nazw rozszerzonych do wywoływania metod z nadklas jest zgodne z podejściem stosowym, jednak i w tym miejscu wykorzystanie grafu klas okazuje się dużo efektywniejsze i chyba prostsze zważywszy na dość skomplikowane operacje przysłaniania, których trzebaby dokonywać przyjmując rozwiązanie z wkładaniem rozszerzonych nazw na stos środowisk.

W ostatecznej implementacji okazało się konieczne dodanie do stosu środowisk informacji o klasie, z której pochodzi aktualnie wykonywana metoda (właściwość `actualBindClassLid` w klasie implementacji `QExecutor::EnvironmentStack`) oraz rozszerzenie protokołu wywołania i powrotu z procedury o obsługę tej informacji (przed wywołaniem metody zapamiętywana jest stara wartość ze stosu środowisk i wstawiana wartość odpowiednia dla obsługiwanej metody, po wywołaniu przywracana jest stara wartość). Została też rozszerzona funkcjonalność metody `bindProcedureName` z klasy implementacji `QExecutor:EnvironmentStack` i na dodatkowym parametrze zwraca ona identyfikator klasy, w której znaleziono metodę przeznaczoną do wywołania. Również funkcjonalność metody grafu klas `findMethod` musiała zostać rozbudowana. W obecnej postaci ma ona możliwość obsługi nazw rozszerzonych, dzięki temu, że przyjmuje identyfikator klasy, która wyznacza podgraf, w którym należy szukać klasy wyznaczonej przez nazwę rozszerzoną, a w niej z kolei metody, też wyznaczonej przez tę nazwę. `findMethod` zwraca też, na dodatkowym parametrze, identyfikator klasy, w której znaleziono metodę. Dodatkowo `findMethod` obsługuje też rozszerzone nazwy metody postaci `super::nazwaMetody`, w ten sposób, że pomija pierwszy poziom wyszukiwania, czyli zaczyna szukanie od nadklasy metody, z której wywołanie tej postaci nastąpiło, co sprawia, że działa podobnie do tego, jak tego typu wywołania działają w języku Java. Oznacza to, że w systemie LoXiM działa składnia `super`, tyle że wielokrotne dziedziczenie ogranicza jej użyteczność do sytuacji w których nie występuje konflikt nazw metod w nadklasach.

Warto podkreślić, że wywołania metod z nadklas działają również na obiektach, a więc można wywołać na obiekcie metodę z jakiejś (lub ze wskazanej) nadklasy, którejś z jego klas instancyjnych. Tak więc następujące konstrukcje składniowe są w systemie LoXiM poprawne i sensowne: `obiekt.Klasa::metoda()`, `obiekt.super::metoda()`.

5.6. Operacje dotyczące jednocześnie klas i obiektów

Wszystkie opisane w tym rozdziale operatory mają po dwa argumenty, jednym jest zapytanie zwracające obiekty, a drugim zapytanie zwracające klasy. Choć takie rozwiązanie może się wydawać zaskakujące, zwłaszcza w wypadku takich operatorów jak rzutowanie oraz sprawdzanie, czy obiekt jest instancją klasy, to jest ono konsekwencją tego, że obiekt może być instancją wielu niedziedziczących po sobie klas. Można też uznać to rozwiązanie za dostosowanie ope-

[7] patrz 5.5

racji znanych z języków programowania do potrzeb języków zapytań. Ma ono również cechy uogólnienia pewnych operatorów, pozwalającego rozszerzyć ich działanie z pojedynczych artefaktów na kolekcje artefaktów.

5.6.1. Operacje na zbiorach instancji klas

Przykład 5.6.1 *Operator dołączający obiekty do zbiorów instancji klas*

ZZK includes ZZO

Znaczenie skrótów:

- *ZZK — zapytanie zwracające klasy,*
- *ZZO — zapytanie zwracające obiekty.*

Przykład 5.6.2 *Operator usuwający obiekty ze zbiorów instancji klas*

ZZK excludes ZZO

Znaczenie skrótów identyczne jak w poprzednim przykładzie:

- *ZZK — zapytanie zwracające klasy,*
- *ZZO — zapytanie zwracające obiekty.*

Jak widać na tych przykładach (5.6.1 i 5.6.2) operatory `includes` i `excludes` mają bardzo podobną składnię, również ich semantyka jest ściśle ze sobą powiązana. Operator `includes` działa w ten sposób, że wszystkie obiekty zwrócone przez zapytanie „ZZO" stają się instancjami wszystkich klas zwracanych przez zapytanie „ZZK". Przy czym jeśli obiekt był instancją klas nie zwracanych przez „ZZK" to pozostaje nią nadal. Jeżeli połączyć ten fakt z tym, że instancje klas są zgromadzone w zbiorze (czyli każdy obiekt jest co najwyżej jednokrotną instancją klasy), to można opisać znaczenie `includes` jako sumę zbiorów, której jednym składnikiem jest zbiór dotychczas istniejących instancji klas, a drugim zbiór instancji klas wynikających z wejścia operatora. Wtedy działanie operatora `excludes` da się opisać jako różnicę tych zbiorów. Konsekwentnie nie zwracany jest błąd, gdy następuje próba usunięcia obiektu ze zbioru instancji klasy, której obiekt nie był instancją.

Implementacja obu tych operatorów wymagała dodania do klasy `Store::DBDataValue` operacji na właściwości `classMarks`, która, jest zbiorem identyfikatorów klas obiektu (w wypadku gdy mamy do czynienia z obiektem składu, który nie jest klasą, dokładny opis w rozdziale 2). Dla `includes` była to operacja dodania do tego zbioru, a dla drugiego operatora usuwania ze zbioru. Samo działanie operatorów przebiega tak, że najpierw zbierane są wyniki wykonania „ZZK" do tymczasowego zbioru, potem dla każdego obiektu wykonywana jest odpowiednia operacja (dodawanie lub usuwanie z `classMarks`), biorąca pod uwagę ten zbiór. Zarówno w wypadku, gdy obiekt (który nie jest klasą) jest zwracany przez zapytanie „ZZK", jak i w sytuacji gdy klasa jest zwracana przez zapytanie „ZZO" zgłaszany jest błąd i to nie zależnie od tego, czy któryś z wyników wykonania „ZZK" i „ZZO" był pusty.

5.6.2. instanceof

Przykład 5.6.3 *Operator sprawdzający, czy każdy podany obiekty jest instancją wszystkich podanych klas*

ZZO instanceof ZZK

Znaczenie skrótów identyczne jak w przykładach 5.6.1 i 5.6.2:

- *ZZK — zapytanie zwracające klasy,*
- *ZZO — zapytanie zwracające obiekty.*

Jak już wspomniano na początku tego rozdziału znaczenie operatora `instanceof` różni się od znanego z języka Java, choć wynikiem jego działania też jest wartość logiczna. Operator ten zwraca wartość logiczną „prawda", jeśli wszystkie obiekty zwrócone przez zapytanie „ZZO" są instancjami każdej klasy zwracanej przez zapytanie „ZZK" lub jej podklasy. W szczególność, gdy wynikiem przynajmniej jednego z zapytań jest zbiór pusty zwracana jest „prawda". W przeciwnym do opisanego wyżej przypadku zwracana jest wartość logiczna „fałsz".

Kluczowe znaczenie dla zrealizowania tego operatora miała metoda grafu klas `isCastAllowed`, która biorąc obiekt i identyfikator klasy stwierdza, czy dany obiekt jest instancją tej klasy lub którejś z jej podklas. Zastosowano też optymalizację, dzięki której sprawdzanie jest kończone w momencie, gdy okaże się, że pierwszy z obiektów nie przejdzie testu wykonywanego przez `isCastAllowed`.

5.6.3. Rzutowanie

Przykład 5.6.4 *Operator wybierający te spośród zadanych obiektów, które można rzutować na podane nadklasy*

cast(ZZO as ZZK)

Znaczenie skrótów identyczne jak w przykładach 5.6.1 i 5.6.2 i 5.6.3:

- *ZZK — zapytanie zwracające klasy,*
- *ZZO — zapytanie zwracające obiekty.*

Różnica między rzutowaniem tu przedstawianym, a rzutowaniem znanym z języków programowania takich jak C++ i Java polega na trzech rzeczach:

- rzutowanie jest przeprowadzane z kolekcji obiektów na kolekcję obiektów[8];
- wynikiem rzutowania jest kolekcja obiektów, której elementy należą do kolekcji wejściowej;
- gdy obiektu nie można rzutować, to nie jest zgłaszany błąd, ale obiekt nie jest dodawany do kolekcji wynikowej.

Oznacza to, że rzutowanie jest rodzajem selekcji elementów na podstawie ich klas. W skład wyniku działania tego operatora wchodzą te obiekty zwrócone przez „ZZO", które są instancjami każdej klasy zwracanej przez zapytanie „ZZK" lub jej podklasy. Gdy wynik zapytania „ZZK" jest pusty, to zwracany jest wynik „ZZO".

Podobnie jak w wypadku realizacji `instanceof`, tak i w wypadku rzutowania kluczowe znaczenie miała metoda `isCastAllowed`, z tą różnicą, że tutaj służyła ona jako filtr.

[8] Ta cecha odróżnia to rzutowanie od rzutowania opisanego w [ODRA].

5.6.4. Możliwe zastosowania i ewentualne rozszerzenia

Dynamiczne dodawanie zobowiązań i konstruktory

Wydaje się, że moment, w którym jest wykonywany operator `includes` jest dobrą chwilą, by uruchomić konstruktor odpowiedni dla danej klasy. Zaś moment, w którym wykonywany jest operator `excludes` jest dobry do tego, by uruchomić destruktor. Takie wprowadzenie tych konstrukcji byłoby dość nietypowe, ponieważ obiekt, który jest dołączany do zbioru instancji jakiejś klasy już istnieje, a po odłączeniu go od niej nadal istnieje, więc nazwy konstruktor i destruktor należałoby zapewne zmienić.

Z drugiej strony nie trudno zauważyć, że operatory `includes` i `excludes` świetnie się nadają do dodawania i usuwania zobowiązań do obiektu. O ile w wypadku obiektów atomowych ewentualne nowe zobowiązania mają raczej ograniczone możliwości ze względu na to, że nie mogą dodać do obiektu nowych pól, o tyle w wypadku obiektów złożonych nie napotykamy tego typu ograniczeń. Klasa może mieć metodę inicjującą, której uruchomienie powoduje dodanie do obiektu pól, które są potrzebne do prawidłowego działania klasy i metodę usuwającą te pola przed odłączeniem obiektu od klasy. Najlepiej byłoby, żeby te metody były uruchamiane automatycznie i tu wracamy do idei konstruktorów i destruktorów.

Przedstawiony wyżej mechanizm dynamicznego dodawania zobowiązań ma wadę polegającą na tym, że nie ma w nim żadnych zabezpieczeń chroniących przed konfliktem nazw pól. Jednak jego możliwość wyglądają obiecująco.

Sposób na konflikt nazw właściwości przy wielokrotnym dziedziczeniu?

Przykład 5.6.5 *Ten znany przykład pokazuje użyteczność wielokrotnego dziedziczenia i jednocześnie problemy z nim związane.*

Mamy klasę `Obiekt-Poruszający-Się`*, z niej dziedziczą dwie klasy:* `Samochód` *i* `Łódź`*, z nich z kolej dziedziczy klasa* `Amfibia`*.*

Klasa `Obiekt-Poruszający-Się` *ma właściwość* `szybkość-maksymalna`*, która jest liczbą kilometrów na godzinę i bezparametrową metodę* `zwiększ-szybkość`*, która zwiększa wartość właściwości* `szybkość-maksymalna` *o jeden.*

Gdybyśmy zaimplementowali podany przykład w C++, interpretacja jaką narzuciłby nam język jest taka, że `Amfibia` ma dwie właściwości `szybkość-maksymalna` mianowicie: `Samochód::szybkość-maksymalna` i `Łódź::szybkość-maksymalna`. Implementacja w LoXiM narzuciłaby inną interpretację, a to taką, że jest jedna właściwość `szybkość-maksymalna`. W LoXiM istniałaby możliwość wywołania metody `zwiększ-szybkość` na instancji klasy `Amfibia`, w C++ należałoby wskazać, z której nadklasy jest to metoda.

Obie te interpretacje mogą być przydatne w praktycznych zastosowaniach powstaje więc pytanie, czy można w LoXiMie stworzyć kod, który interpretowałby podany przykład podobnie do C++? Podany niżej sposób jest dowodem na to, że tak, choć nie jest to tak proste jak w C++.

Rozwiązanie tego problemu wymaga dwóch dodatkowych pustych klas: `Pola-Samochodu` i `Pola-Łodzi`. Wszystkie ewentualne operacje wykonywane na właściwości `szybkość-maksymalna` w klasie `Samochód` musiałyby być wykonywane po rzutowaniu tej właściwości na klasę `Pola-Samochodu`[9], a w obiekcie, który staje się instancją tej klasy, należałoby dodać tą właściwość i dołączyć ją do instancji klasy `Pola-Samochodu`. Podobnie należałoby zrobić z klasami

[9] To nie musi być bardzo uciążliwe, wystarczy w tym celu użyć akcesorów.

`Pola-Łodzi` i `Łódź`. Przy dodawaniu obiektu do klasy `Amfibia` musiałyby być wywołane metody dodające pola z jej nadklas[10]. Oczywiście można uniknąć niepotrzebnego dodawania właściwości do obiektu, ale w tym celu klasy `Pola-Samochodu` i `Pola-Łodzi` powinny dziedziczyć z jednej nadklasy np. `Pola`. Wtedy przed dodaniem właściwości należałoby sprawdzić, czy nie istnieje już właściwość o tej nazwie. Jeśli taka właściwość istnieje, i jest instancją pożądanej podklasy klasy `Pola` to nie trzeba nic robić. Jeśli istnieje taka właściwość i nie jest instancją podklasy klasy `Pola` należy ją uczynić instancją pożądanej podklasy klasy `Pola`. W trzecim możliwym przypadku należy dodać właściwość.

Tak można uzyskać dwa pola oddzielnie obsługiwane przez każdą z nadklas klasy `Amfibia`. Powstaje teraz inny problem. Nadklasa `Obiekt-Poruszający-Się` zamiast jednego obiektu ma do czynienia z kolekcją obiektów, co prowadzi do wniosku, że należałoby to przewidzieć już na etapie projektowania tej klasy i odpowiednio ją zaimplementować (w podanym przykładzie takie problemy implementacyjne stwarzałaby metoda `zwiększ-szybkość`). Jednak ten wysiłek dałby efekt pod pewnym względem lepszy niż w C++. Przy takim rozwiązaniu wywołanie metody nie wymagałoby wskazywania nadklasy, z której ma być metoda wywołana.

Trzeba zaznaczyć, że nadal istniałaby różnica w interpretacjach, co widać w momencie gdy chce się przedefiniować metodę `zwiększ-szybkość` w klasie `Amfibia` tak, żeby zwiększała ona tylko wartość właściwości związanej z klasą `Samochód`. W C++ można do tego celu użyć, w nowo tworzonej metodzie, metody `Samochód::zwiększ-szybkość`, w LoXiM nie ma takiej możliwość. Można wyobrazić sobie usprawnienie, które mogłoby pomóc uporać się i z tym problemem. Wydaje się, że wystarczyłoby wprowadzić do systemu możliwość oznaczania przynależności właściwości do klas. To pozwoliłoby uniknąć konieczności tworzenia pustych klas i mogłoby posłużyć do tego, by wywołanie metody z określonej nadklasy mogło użyć właściwości przeznaczonych dla tej nadklasy. To rozwiązanie byłoby zgodne z podejściem stosowym, bo można by je sobie wyobrażać, jako stworzenie nowej sekcji na stosie środowisk, zawierającej właściwości przeznaczone dla danego wywołania metody.

5.7. Składowe klasowe

Rozdział 2 omawia sposoby tworzenia i usuwania właściwości i metod statycznych. W tym rozdziale skupimy się na tym, jak ich używać. Powiemy też o problemach związanych z przyjętym tu rozwiązaniem i zaproponujemy sposób ich rozwiązania.

Używanie składowych statycznych bardzo przypomina używanie zwykłych obiektów, których identyfikatory są wśród identyfikatorów startowych, bo takimi obiektami składowe klasowe de facto są. Dzięki temu stają się one rodzajem zmiennych lub procedur globalnych dostępnych z dowolnego miejsca w kodzie programu. Jedynym co je wyróżnia z grona innych obiektów jest użycie nazwy rozszerzonej i to niezależnie od tego, czy pisany kod jest w klasie, do której należą składowe statyczne, czy nie. Innymi cechami, które wyróżniają składowe klasowe od innych obiektów, są, opisane w rozdziale 2, zachowania związane z tworzeniem tych składowych, oraz z brakiem możliwości umieszczania ich pod nazwami rozszerzonymi poza zbiorem identyfikatorów startowych.

5.7.1. Problemy i możliwe usprawnienia

Niestety z tego, że składowe klasowe umieszczane są wśród identyfikatorów startowych wynikają pewne problemy. Naturalnym rozwiązaniem problemu przestrzeni nazw i pakietów w podejściu stosowym byłoby umieszczenie klas w odpowiednich obiektach, taka operacja w

[10] Znowu przydałby się konstruktor.

podejściu stosowym powinna być niezależna od tego, jak działają klasy. Jednak umieszczenie składowych klasowych wśród identyfikatorów startowych spowodowało, że nazwy klas muszą być unikatowe, czyli muszą zawierać w sobie nazwy pakietów i przestrzeni nazw, do których klasa należy. Oznacza to, że albo miejsce składowania klasy nie będzie miało nic wspólnego z tym do jakiego pakietu klasa należy, albo nazwa klasy będzie tworzona na podstawie tego, gdzie ta klasa się znajduje. Oba rozwiązania zdają się być wadliwe. Pierwsze dlatego, że jest niezgodne z podejściem stosowym. Drugie dlatego, że powoduje trudności z odwoływaniem się z metod zawartych w klasie do składowych statycznych tej klasy, ponieważ podczas tworzenia kodu klasy nie wiadomo jaka jest jej nazwa.

Wszystko wskazuje na to, że pierwszym krokiem, na drodze do implementacji składowych statycznych o właściwościach lepszych niż opisywana tutaj, powinno być umieszczenie tych składowych poza zbiorem identyfikatorów startowych. Mogłoby się wydawać, że powinny one być przechowywane w tym miejscu gdzie klasa i powinny nadal używać nazw rozszerzonych, jednak w takiej sytuacji wystąpiłby problem z tym, że podczas tworzenia klasy nie wiadomo gdzie będzie przechowywana w związku z tym nie byłoby jak dostać się z wnętrza klasy do jej statycznych składowych. Może jednak lepszym pomysłem byłoby przechowywanie statycznych składowych w klasie. Wtedy byłyby one wkładane na stos środowisk wraz z nią i dostęp do nich byłby możliwy z tej klasy. Nie trzeba by też używać nazw rozszerzonych do przechowywania tych składowych, należałoby wtedy rozwiązać problem przechowywania innych informacji o klasie (takich jak pola instancyjne, niezmiennicza nazwa obiektów itp.), to jednak nie jest duży problem, wystarczy umieścić je w podobiekcie o zabronionej nazwie np. ::.

Ale zanim podejmie się jakiekolwiek próby zmieniania tej implementacji, należy się zastanowić, czy w praktyce ewentualna niezgodność z podejściem stosowym konstrukcji takich jak przestrzenie nazw i pakiety stanowi jakiś problem. Wydaje się wysoce wątpliwe, by można było tworzyć sensowne oprogramowanie dynamicznie zmieniając przestrzenie nazw i pakiety klas, a to dlatego, że klasy korzystające z tych klas zakładają coś o nazwach ich pakietów i nazwach ich przestrzeni nazw. Co więcej wydaje się, że oddzielenie nazwy pakietu od położenia w bazie danych może być rozwiązaniem dużo bardziej elastycznym, właśnie dlatego, że takie klasy można umieścić gdzie się chce, a wszystko i tak będzie działać. Tak więc wbrew pozorom, może się okazać, że pomysł, w którym pakiet stanowi część nazwy klasy jest wystarczająco dobrym rozwiązaniem.

Podsumowanie

Możliwe, że najciekawszym rezultatem osiągniętym w tej pracy jest stworzenie zalążków spójnego systemu zarządzania instancjami klas. Również rozwiązania pozwalające wywołać metodę z konkretnej nadklasy mają szansę okazać się dość przydatne, gdyż pomagają one w rozwiązywaniu problemów stwarzanych przez wielokrotne dziedziczenie. Pomimo wątpliwości, związanych ze zgodnością z paradygmatem podejścia stosowego, także składowe stałe nie wyglądają na konstrukcję skazaną na bezrefleksyjne usunięcie z systemu. Jeśli nawet ostatecznie zostaną usunięte, to sama próba ich realizacji i jej analiza może się przydać jako punkt wyjścia do dalszej pracy lub przestroga przed jej podjęciem[11].

Opisywana tutaj implementacja wprowadziła do systemu LoXiM klasy w podstawowym zakresie. Dalsze prace w tym kierunku, z punktu widzenia logiki tej implementacji, powinny skupiać się na stworzeniu pakietów i ewentualnie przestrzeni nazw, co wiąże się z rozważeniem czy i w jaki sposób powinny być zaimplementowane składowe stałe. Także w pierwszej kolejności należałoby rozważyć stworzenie konstruktorów i destruktorów wyzwalanych odpowiednio w momencie kiedy obiekt staje się instancją klasy i kiedy przestaje nią być.

[11] Choć autor tej pracy ma nadzieję, że ostatnia z możliwości jest jedynie hipotetyczna.

Dodatek A

Dołączona płyta

A.1. Zawartość dołączonej płyty

W katalogu głównym znajduje się plik o nazwie „sk153407_pracamgr.pdf", z którego wydrukowano tę pracę. W katalogu „/szbd/Testy" znajduje się plik o nazwie „test_klas.sbql", którego zawartość wydrukowano w dodatku „B". Katalog „/szbd" zawiera pliki źródłowe wersji systemu LoXiM opisywanej w tej pracy.

A.2. Instalacja

LoXiM działa w systemie operacyjnym Linux. W celu zainstalowania systemu LoXiM należy przegrać zawartość katalogu „/szbd" z załączonej płyty na twardy dysk. Następnie z katalogu „szbd" uruchomić kompilację poleceniem `make`. Po zakończeniu procesu kompilacji z katalogu „szbd/Server" uruchamiamy serwer poleceniem `./Listener`. Następnie należy uruchomić program kliencki z katalogu „szbd/SBQLCli" poleceniem `./SBQLCli`, po czym należy zalogować się do systemu jako użytkownik „root" z pustym hasłem i rozpocząć transakcję poleceniem `begin`. Teraz już można wykonać kod w SBQLu zawarty w dodatku „B".

Dodatek B

Przykład użycia

```
-- Uwaga interpretacje konstrukcji składniowych pojawiają się tylko
-- przy ich pierwszym wystąpieniu.

-- Nadklasa wszystkich rodzajów produktów.
-- Nazwa obiektów: produkt
-- Wymagane pola: nazwa, cena
create class Produkt {
  instance produkt : {nazwa; cena}
}
/

-- Płyty Audio.
-- Rozszerza klasę Produkt.
create class PlytaAudio extends Produkt {
  instance plytaAudio : {wykonawca}
}
/

-- Produkty Elektroniczne
-- porownaj(v1, v2) - Metoda statyczna do porównywania parametrów produktów.
-- NOTEQAL, EQUAL - właściwości statyczne służące za wyniki porównania.
-- porownaj( prodEl ) - metoda do porównywania parametrów bieżącego obiektu
--                      z obiektem prodEl.
create class ProduktElektroniczny extends Produkt {
  instance produktElektroniczny : { poborMocy }
  static {NOTEQUAL; EQUAL}

  static procedure porownaj(v1, v2) {
    if v1 > v2 then
      return deref(ProduktElektroniczny::NOTEQUAL)
    else if v1 = v2 then
      return deref(ProduktElektroniczny::EQUAL)
    fi fi;
    return (-ProduktElektroniczny::NOTEQUAL)
  }
```

```
  procedure porownaj( prodEl ) {
    return ProduktElektroniczny::porownaj(prodEl.poborMocy, poborMocy)
  }
}
/

-- Inicjowanie właściwości statycznych

create 1 as ProduktElektroniczny::NOTEQUAL
/

create 0 as ProduktElektroniczny::EQUAL
/

-- Odtwarzacz CD.
-- porownaj( cd ) - przedefiniowanie porównanie uwzględniające też parametr: naIlePlyt
create class OdtwarzaczCD extends ProduktElektroniczny {
  instance odtwarzaczCD : {naIlePlyt}

  procedure porownaj( cd ) {
    return super::porownaj(cd) + ProduktElektroniczny::porownaj(naIlePlyt, cd.naIlePlyt)
  }
}
/

-- Tuner.
-- porownajSamTuner( t ) - metoda porównująca tuner bez porównania
--                         parametru poborMocy z nadklasy
-- porownaj( t ) - przedefiniowanie porównanie uwzględniające też parametr: naIleStacji
create class Tuner extends ProduktElektroniczny {
  instance tuner : {naIleStacji}

  procedure porownajSamTuner( t ) {
    return ProduktElektroniczny::porownaj(naIleStacji, t.naIleStacji)
  }

  procedure porownaj( t ) {
    return porownajSamTuner( t ) + super::porownaj(t)
  }
}
/

-- Wzmacniacz.
-- porownaj( wzm ) - przedefiniowanie porównanie uwzględniające też parametr: mocMuzyczna
create class Wzmacniacz extends ProduktElektroniczny {
  instance wzmacniacz : {mocMuzyczna}

  procedure porownaj( wzm ) {
```

```
    return super::porownaj(wzm) +
      ProduktElektroniczny::porownaj(mocMuzyczna, wzm.mocMuzyczna)
  }
}
/


-- Amplituner.
-- Klasa dziedziczy po klasach Tuner i Wzmacniacz.
-- porownaj( at ) - przedefiniowanie porównanie, które porównuje amplitunery
--                  (czyli wzmacniacz i tuner), gdy at jest amplitunerem,
--                  w p.p. porównuje siebie w zależności od klasy at
--                  albo z tunerem albo ze wzmacniaczem.
create or update class Amplituner extends Wzmacniacz, Tuner {
  instance amplituner : {}

  procedure porownaj( at ) {
    if at instanceof Amplituner then
      return Wzmacniacz::porownaj(at) + Tuner::porownajSamTuner(at)
    fi;
    return - at.porownaj(at)
  }
}
/

-- Klasa umożliwiająca obsługę wypożyczania produktów.
-- Brak nazwy obiektów.
-- konstruktor() - metoda, którą należy wywołać, gdy dołącza się obiekt do tej klasy.
--                 Dodaje ona właściwości potrzebne do działania klasy.
-- destruktor() - metoda, którą należy wywołać, gdy się chce odłączyć obiekt
--                od klasy. Usuwa ona właściwości tej klasy i wyłącza obiekt
--                spośród instancji DoWypozyczenia.
-- Metody konstruktor() i destruktor() zostały upodobnione pod względem tego,
-- co zwracają, do operatorów includes i excludes ze względu na to,
-- iż będą wykorzystywane w identycznych sytuacjach jak te operatory.
create or update class DoWypozyczenia {
  instance : {wypozyczony}

  procedure moznaWypozyczyc() {
    if(wypozyczony = 0) then return self fi
  }

  procedure konstruktor() {
    self :< 0 as wypozyczony;
    return self
  }

  procedure destruktor() {
    delete self.wypozyczony;
```

```
    return DoWypozyczenia excludes self
  }

  procedure wypozycz() {
    if wypozyczony = 0 then
      wypozyczony := 1;
      return 1
    else
      return 0
    fi
  }

  procedure oddaj() {
    wypozyczony := 0
  }
}
/

-- Dodawanie przykładowych obiektów.

PlytaAudio includes create (
  "Ipecac" as producent,
  "thank you for giving me your valuable time" as nazwa,
  "Kaada" as wykonawca,
  40 as cena
) as plytaAudio
/

PlytaAudio includes create (
  "Nonesuch" as producent,
  "Blues Dream" as nazwa,
  "Bill Frisell" as wykonawca,
  45 as cena
) as plytaAudio
/

PlytaAudio includes create (
  "epic" as producent,
  "time's up" as nazwa,
  "Living Color" as wykonawca,
  50 as cena
) as plytaCD
/

Amplituner includes create (
  "Dobra Firma" as producent,
  "AT 1" as nazwa,
  1000 as cena,
  130 as poborMocy,
```

```
  50 as mocMuzyczna,
  99 as naIleStacji
) as amplituner
/

Amplituner includes create (
  "Firma" as producent,
  "super AT 1000" as nazwa,
  400 as cena,
  100 as poborMocy,
  40 as mocMuzyczna,
  9 as naIleStacji
) as amplituner
/

Tuner includes create (
  "Firma" as producent,
  "super T 1000" as nazwa,
  150 as cena,
  10 as poborMocy,
  9 as naIleStacji
) as tuner
/

Tuner includes create (
  "Dobra Firma" as producent,
  "T 3" as nazwa,
  300 as cena,
  10 as poborMocy,
  999 as naIleStacji
) as tuner
/

Wzmacniacz includes create (
  "Dobra Firma" as producent,
  "A 1" as nazwa,
  1000 as cena,
  150 as poborMocy,
  60 as mocMuzyczna
) as wzmacniacz
/

Wzmacniacz includes create (
  "Firma" as producent,
  "super A 1000" as nazwa,
  350 as cena,
  180 as poborMocy,
  60 as mocMuzyczna
) as wzmacniacz
```

```
/

OdtwarzaczCD includes create (
  "Dobra Firma" as producent,
  "CD 1" as nazwa,
  1000 as cena,
  150 as poborMocy,
  1 as naIlePlyt
) as odtwarzaczCD
/

-- Koniec dodawania obiektów (obiektów jest 10 w tym: 2 * tuner, 2 * wzmacniacz,
-- 2 * amplituner, 2 * plytaAudio, 1 * plytaCD, 1 * odtwarzaczCD).

--
-- Zastępowalności na poziomie kolekcji.
--

count(produkt)
/

-- Wynik: 9, bo obiekt o nazwie "plytaCD" nie podlega zasadzie zastępowalności
-- na poziomie kolekcji, gdyż niezmiennicza nazwa obiektów klasy to: "plytaAudio"

count(tuner)
/

-- Wynik: 4, zgodnie z zasadą zastępowalności, amplitunery też zostały
-- policzone jako tunery.

count(produktElektroniczny)
/

-- Wynik: 7, bo PlytaAudio nie dziedziczy po ProduktElektroniczny.

--
-- Wywoływanie metod.
--

(amplituner where nazwa = "AT 1").porownaj(amplituner where nazwa = "super AT 1000")
/

-- Wynik: { 1 }, bo "AT 1" ma wprawdzie większy pobór mocy, ale też większą
-- moc muzyczną i większą ilość stacji, więc wynik jest poprawny.

(amplituner where nazwa = "AT 1").porownaj(wzmacniacz where nazwa = "A 1")
/

-- Wynik: { 0 }
```

```
(wzmacniacz where nazwa = "A 1").porownaj(amplituner where nazwa = "AT 1")
/

-- Wynik: { 0 }

(wzmacniacz where nazwa = "super A 1000").porownaj(wzmacniacz where nazwa = "A 1")
/

-- Wynik { -1 }


--
-- Modyfikowanie klasy.
--

amplituner.cenaWGroszach()
/

-- Wynik: { }, ponieważ nie ma metody cenaWGroszach()

-- Dodanie metody cenaWGroszach() do klasy Produkt (dodano też pole producent)
update class Produkt {
  instance produkt : {producent; nazwa; cena}

  procedure cenaWGroszach() {
    return 100 * cena
  }
}
/

amplituner.cenaWGroszach()
/

-- Wynik: { 100000 40000 }, modyfikacja klasy się powiodła, co więcej
-- można powtórzyć z takim samym wynikiem wszystkie poprzednie testy.

--
-- Dodawanie zobowiązań.
--

-- Rzutowanie wybiera te produkty, które są jednocześnie
-- instancjami klasy DoWypozyczenia.
-- Operator "deref" jest prostym sposobem na wypisanie zawartości obiektów.
deref(cast(produkt as DoWypozyczenia))
/

-- Wynik: { }, nie ma produktów klasy DoWypozyczenia.
```

```
-- Przeznaczenie amplitunerów do wypożyczenia.
(DoWypozyczenia includes amplituner).konstruktor()
/

deref(cast(produkt as DoWypozyczenia))
/

-- Wynik: {
-- < producent(Dobra Firma)  nazwa(AT 1)  cena(1000)  poborMocy(130)
--   mocMuzyczna(50)  naIleStacji(99)  wypozyczony(0)  >
-- < producent(Firma) nazwa(super AT 1000)  cena(400)  poborMocy(100)
--   mocMuzyczna(40)  naIleStacji(9)  wypozyczony(0)  >  }
-- Amplitunery mogą zostać wypożyczone. Zauważmy właściwość "wypozyczony",
-- dodaną przy pomocy konstruktora do obu amplitunerów.

-- Wypożyczamy jeden z amplitunerów.
(amplituner where producent = "Firma").wypozycz()
/

-- Wynik: { 1 }, udało się.

-- Wypożyczamy ten sam amplituner powtórnie.
(amplituner where producent = "Firma").wypozycz()
/

-- Wynik: { 0 }, nie udało się bez oddania wypożyczyć.

-- Uruchomienie destruktora z dokładnie wskazanej klasy.
-- To, że destruktor zwraca zmieniane obiekty zostało wykorzystane
-- do ich wypisania przez "deref".
deref(cast(produkt as DoWypozyczenia).DoWypozyczenia::destruktor())
/

-- Wynik: {
-- < producent(Dobra Firma)  nazwa(AT 1)  cena(1000)  poborMocy(130)
--   mocMuzyczna(50)  naIleStacji(99)  >
-- < producent(Firma)  nazwa(super AT 1000)  cena(400)  poborMocy(100)
--   mocMuzyczna(40)  naIleStacji(9)  >  }
-- Jak widać nastąpiło usunięcie właściwości: "wypozyczony".

deref(cast(produkt as DoWypozyczenia))
/

-- Wynik: { }, czyli tak jak przed wywołaniem konstruktorów,
-- nie ma produktów klasy DoWypozyczenia.


--
-- Usuwanie klas.
```

```
--

delete Produkt
/

count(produkt)
/

-- Wynik: 0, bo nie ma klasy Produkt.

count(tuner)
/

-- Wynik: 4.

count(produktElektroniczny)
/

-- Wynik: 7, inne klasy jak widać działają.

(amplituner where nazwa = "AT 1").porownaj(wzmacniacz where nazwa = "A 1")
/

-- Wynik: { 0 }, wywołania metod też działają.
-- Uwaga, po usunięciu nadklasy hierarchia jest nieodwracalnie uszkodzona,
-- a to dlatego, że składnia dodawania klasy nie przewiduje możliwości
-- wskazania jej podklas.
```

Dodatek C

Krzysztof Stencel: *Klasy w ODRA*

Dodatek zawiera niepublikowany maszynopis Krzysztofa Stencela *Klasy w ODRA*.

C.1. Krzysztof Stencel: *Klasy w ODRA*

C.1.1. Niezmienniki

Przyjmujemy, że klasa w ODRA jest miejscem przechowywania niezmienników pewnej grupy obiektów. Tymi niezmiennikami są:

1. Typ (struktura danych).
2. Metody (w pierwszej wersji nie przewidujemy przeciążania metod).
3. Nazwa (opcjonalnie).

C.1.2. Niezmiennicza nazwa obiektów klasy

Klasy mogą wprowadzać niezmienniczą nazwę obiektów, ale nie muszą. W przypadku gdy klasa nie wprowadzi takiej nazwy, obiekty tej klasy ta nie podlegają zasadzie zastępowalności na poziomie kolekcji obiektów. To oznacza np., że instancje klasy nie posiadającej niezmiennika nie będą zwracane, gdy pojawi się w zapytaniu niezmiennicza nazwa którejś z jej podklas.

Deklaracja klasy nie jest deklaracją obiektów tej klasy. Obiekty klasy są deklarowane osobno. Jeśli klasa definiuje niezmienniczą nazwę swoich obiektów, można zadeklarować jej obiekty pod inną nazwą. Takie obiekty nie podlegają zasadzie zastępowalności na poziomie kolekcji obiektów. Obiekt klasy wprowadzającej niezmienniczą nazwę swoich obiektów zachowuje się tak, jakby miał wiele nazw, tj. ma on wszystkie nazwy wprowadzane przez wszystkie nadklasy swojej klasy, które wprowadzają niezmiennicze nazwy swoich obiektów. Gdy na stos środowisk ma trafić binder do obiektu takiej klasy (do sekcji bazowej modułu lub np. w wyniku nawigacji po asocjacji), do odpowiedniej sekcji bazowej stosu środowisk jest więc wkładane tyle binderów z nim w środku, ile ma on nazw ustalonych przez jego klasę i jej nadklasy.

Przykład

Przypuśćmy, że mamy klasy *PersonClass* o niezmienniczej nazwie obiektów *Person* oraz dziedziczącą po niej klasę *EmployeeClass* o niezmienniczej nazwie obiektów *Employee*. Zadeklarowanie i powołanie do życia trzech obiektów o nazwie *Employee* (załóżmy, że ich identyfikatory wewnętrzne to $i1$, $i2$ oraz $i3$) powoduje dodanie do sekcji bazowej stosu środowisk sześciu binderów: $Employee(i1)$, $Employee(i2)$, $Employee(i3)$, $Person(i1)$, $Person(i2)$,

$Person(i3)$. Dalsze wiązanie nazwy przebiega wg normalnej mechaniki stosowej. Jeśli zadeklarowano i powołano by te same obiekty pod inną nazwą np. Emp, to wówczas nie obowiązuje zastępowalność i do sekcji bazowej dodane zostaną tylko trzy bindery $Emp(i1)$, $Emp(i2)$, $Emp(i3)$.

Uwaga

Może to prowadzić do pewnych paradoksów. Załóżmy że w globalnej przestrzeni nazw będą obecne powyżej opisane klasy *PersonClass* i *EmployeeClass* oraz ich obiekty *Person* i *Employee*. I teraz w lokalnym środowisku procedury zadeklarujemy obiekty lokalne *Employee* klasy *EmployeeClass*, to wiązanie nazwy *Employee* zwróci tylko obiekty lokalne *Employee*. (ich bindery będą w wyższej sekcji niż bindery obiektów globalnych — w sekcji lokalnego środowiska procedury). Co więcej, wiązanie nazwy *Person* też zwróci tylko obiekty lokalne *Employee* (bo na stosie w rekordzie aktywacji procedury znajda się też bindery *Person* dla lokalnych obiektów!). Nic w tym nadzwyczajnego, bo taka jest mechanika stosowa i wynikające z niej przesłanianie nazw.

C.1.3. Składnia deklaracji

```
class nazwaKlasy [extends listaNazwKlas]? {
  instance [nazwaObiektówKlasy]? : { [deklaracjaAtrybutu]* }
  [deklaracjaMetody]*
}
```

Metasymbole są jak w XML DTD: pytajnik = 0..1, gwiazdka = 0..*. Opcjonalne są więc lista nadklas, nazwa obiektów klasy. Można też nie deklarować metod ani atrybutów w konkretnej klasie. Oto przykład deklaracji

```
class EmployeeClass extends PersonClass {
  instance Employee : {
    salary : integer;
    works_in : &Department;
    prev_job : record {
      comp_name : string:
      from : integer;
      till : integer;
    }
  }
  changeSalary(amount : integer){
    salary := amount;
  }
  moveEmployee(newDept : &Department){
    ...
  }
}
```

C.1.4. Deklaracja obiektów klasy

Składnia jest taka jak deklaracja obiektów o ustalonym typie. Przy czym zamiast typu używamy nazwy klasy:

```
nazwaObiektów : nazwaKlasy liczebność;
```

Przykład:

```
Employee : EmployeeClass [0..*];
```

Lub

```
Emp : EmployeeClass [0..*];
```

W tym wypadku zadeklarowano obiektu pod inną nazwą niż niezmiennik nazwowy klasy. W związku z czym obiekty o nazwie *Emp* nie będą zastępowalne na poziomie kolekcji: wiązanie nazw *Employee* oraz *Person* nie zwróci referencji do obiektów zadeklarowanych powyżej jako *Emp*.

C.1.5. Otwarcie zakresu na $ENVS$ w czasie wykonania

Wykonanie operatora niealgebraicznego na obiekcie należącym do klasy powoduje włożenie na stos środowisk kilku sekcji. Przypuśćmy, że przetwarzany obiekt o referencji r należy do klasy K_0, która ma n nadklas K_1, K_2,..., K_n. Klasy te uporządkowane są w ten sposób, że dla żadnego $a < b$ klasa K_a nie jest nadklasą klasy K_b. To oznacza, że żadna klasa nie poprzedza swojej nadklasy. W przypadku wielodziedziczenia takich porządków może być wiele. Implementacja może wybrać dowolny.

Zakładamy więc, że w przypadku wystąpienia konfliktu dziedziczenia, do wykonania może zostać wybrana dowolna metoda spośród tych o właściwej nazwie znajdujących się w grafie dziedziczenia. W takiej sytuacji i tak nie da się zachować zasady zastępowalności.

Na czubek stosu środowisk wkładane są nowe sekcje w następującej kolejności (czubek stosu jest na górze kartki):

bindery do podobiektów składowych w r
bindery do metod klasy K_0
bindery do metod klasy K_1
bindery do metod klasy K_2
...
bindery do metod klasy K_n

C.1.6. Otwarcie zakresu na $ENVS_S$ w czasie kontroli typów

Kontrola statyczna operatora niealgebraicznego na sygnaturze referencyjnej odwołującej się do węzła klasy w metabazie musi symulować to samo obliczenie z czasu wykonania. Przypuśćmy, że przetwarzana sygnatura dotyczy węzła metabazy klasy K0, która ma n nadklas K_1, K_2,..., K_n. Klasy te uporządkowane są w tej sposób, że dla żadnego $a < b$ klasa Ka nie jest nadklasą klasy K_b. To oznacza, że żadna klasa nie poprzedza swojej nadklasy. W przypadku wielodziedziczenia takich porządków może być wiele. Kontroler typów może wybrać dowolny. Wskazane jest aby wybrał ten sam porządek co interpreter czasu wykonania.

Na czubek statycznego stosu środowisk wkładane są nowe sekcje w następującej kolejności (czubek stosu jest na górze kartki):

bindery statyczne do atrybutów K_0, K_1, K_2,..., K_n
bindery statyczne do metod klasy K_0
bindery statyczne do metod klasy K_1
bindery statyczne do metod klasy K_2
...
bindery statyczne do metod klasy K_n

C.1.7. Reprezentacja klas w metabazie

Szkielet implementacji klas w metabazie jest już wykonany. Klasa w metabazie będzie reprezentowana przez obiekt złożony o rodzaju (pierwszy podobiekt o nazwie $*kind*) równym `MetaObjectKind.CLASS_OBJECT`. Następne podobiekty będą miały kolejno następujące nazwy i znaczenia:

- $*extends*: lista pointerów do węzłów metabazy reprezentujących bezpośrednie nadklasy.
- $*structure*: pointer do węzła metabazy reprezentującego typ rekordowy złożony z atrybutów własnych danej klasy.
- $*methods*: lista pointerów do węzłów metabazy reprezentujących metody własne danej klasy.
- $*objectName*: napis reprezentujący niezmienniczą nazwę obiektów danej klasy (jeśli napis jest pusty, to tej nazwy nie podano).

Klasa `odra.db.objects.meta.MBClass` udostępnia API do obiektów metabazy reprezentujących klasy.

C.1.8. Reprezentacja klas w składzie obiektów

Szkielet implementacji klas w składzie obiektów jest już wykonany (choć znacznie słabiej niż w metabazie). Klasa w składzie obiektów będzie reprezentowana przez obiekt złożony o rodzaju (pierwszy podobiekt $*kind*) równym `DataObjectKind.CLASS_OBJECT`). Następne podobiekty będą miały kolejno następujące nazwy i znaczenia:

- $*methods*: lista pointerów do obiektów składu obiektów będących procedurami, które są metodami własnymi danej klasy.

Klasa `odra.db.objects.data.DBClass` udostępnia API do obiektów składu obiektów reprezentujących klasy.

C.1.9. Rzutowanie (cast)

Należy zaimplementować operację rzutowania na klasę. Operacja ta będzie makroskopowa i dynamiczna, tzn. jej argumentem jest bag referencji do obiektów. Wynikiem jest podbag tej kolekcji, który zawiera referencje tych i tylko tych, obiektów, które należą do klasy, na którą rzutujemy. Jeśli rzutowanie jest „nieudane", to zwracany jest pusty bag, a nie błąd luba jakaś wartość niezdefiniowana.

Składnia rzutowania będzie taka jak w C/C++: `(NazwaKlasy) zapytanie` Do zaimplementowania są: *downcast* (rzutowanie na podklasę), *upcast* (rzutowanie na nadklasę) oraz *crosscast* (rzutowanie na klasę, która nie jest przodkiem ani potomkiem w hierarchii dziedziczenia).

Downcast (przykład): `(StudentClass) (Person where age > 30)`

Wynikiem zapytania jest kolekcja obiektów klasy StudentClass mających wiek większy od 30. Rzutowanie działa więc tu także jak selekcja — odrzuca obiekty nie należące do StudentClass. Jeśli nie będzie żadnego takiego obiektu wynikiem tego zapytania jest pusty bag.

Upcast (przykład): `((EmpClass)((PersonClass)((StudentClass) (Person where age > 30)))`

To zapytanie zwróci studentów będących jednocześnie pracownikami. Po downcaście na *StudentClass* dokonujemy *upcastu* na *PersonClass*, żeby zrobić *downcast* na *EmpClass*, wyłuskując tych ponadtrzydziestoletnich studentów, którzy są jednocześnie pracownikami.

Crosscast (przykład): `((EmpClass) ((StudentClass) (Person where age > 30))`

To zapytanie ma taki sam wynik, jak poprzednie, ale posługuje się crosscastem. Zrobiliśmy tu bezpośredni cast na klasę *EmpClass*.

Bibliografia

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

[ODRA] Krzysztof Stencel, *Klasy w ODRA*, Dodatek C

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

[SBQL] *http://www.sbql.pl*, strona WWW języka SBQL