



**Polsko-Japońska Wyższa Szkoła
Technik Komputerowych**

Katedra Inżynierii Oprogramowania

INŻYNIERIA OPROGRAMOWANIA I BAZ DANYCH

Kamil Wysocki

Nr albumu 5558

Procedury szablonowe w systemie ODRA

Praca magisterska napisana pod kierunkiem
prof. dr hab. inż. Kazimierza Subiety

Warszawa, czerwiec 2009

SPIS TREŚCI

1. WSTĘP.....	4
1.1 CEL PRACY	4
1.2 REZULTAT PRACY.....	4
1.3 ORGANIZACJA PRACY	5
1.4 OD AUTORA.....	5
2. SZABLONY FUNKCJI – WSTĘP	6
2.1 PRZECIĄŻENIE NAZW FUNKCJI.....	7
2.2 PRZECIĄŻENIE A REFERENCJA.....	9
2.3 KONWERSJA NIEJAWNA	10
2.4 KIEDY PRZECIĄŻAĆ?.....	11
3. ARCHITEKTURA STOSOWA I SYSTEM ODRA	12
3.1 CO TO JEST SBA?	12
3.2 OBIEKT I JEGO TOŻSAMOŚĆ	13
3.3 MODELE DANYCH.....	13
3.4 STOSY.....	15
3.5 SBQL W SYSTEMIE ODRA.....	17
3.5.1 <i>Moduły</i>	17
3.5.2 <i>Procedury</i>	18
4. DEKLARACJA SZABLONÓW W SYSTEMIE ODRA.....	19
4.1 TYP SZABLONOWY.....	19
4.2 SKŁADNIA PROCEDUR SZABLONOWYCH	20
4.3 DEKLARACJA SZABLONU	22
5. PROCES TWORZENIA SZABLONÓW W SYSTEMIE ODRA	25
5.1 WYBÓR FAZY KOMPILACJI DO ROZPOCZĘCIA ANALIZY	25
5.2 ALGORYTM WYBORU WŁAŚCIWEGO SZABLONU	27
5.3 ANALIZA ARGUMENTÓW AKTUALNYCH WOŁANIA.....	32
5.4 GENEROWANIE PROCEDURY - UNIFIKACJA SZABLONU.....	35
5.5 SZABLONY W TRADYCYJNYCH JĘZYKACH PROGRAMOWANIA.....	39
6. SZABLONY – UZUPEŁNIENIE I ROZSZERZENIE OPISU.....	42
6.1 PROCEDURY SPECJALIZOWANE.....	42
6.2 DWA SZABLONY, JEDEN SZCZEGÓLNYM PRZYPADKIEM DRUGIEGO.....	44
6.3 PARAMETR USTALAJĄCY TYP REZULTATU FUNKCJI	45
6.4 TYPY WBUDOWANE A TYPY ZDEFINIOWANE.....	46

6.5	MECHANIZM SZABLONÓW DLA KLAS.....	49
7.	PODSUMOWANIE	52
8.	BIBLIOGRAFIA.....	54
9.	SPIS RYSUNKÓW I TABEL	55
10.	DODATEK A - PRZYKŁADY KODU SBQL	56
10.1	DEKLARACJA SZABLONU	57
10.2	WYBÓR ODPOWIEDNIEGO SZABLONU	58
10.3	ZAPYTANIA JAKO ARGUMENTY AKTUALNE	59
10.4	WOŁANIE SZABLONU W CIELE INNEGO SZABLONU.....	61
10.5	WYWOŁANIA REKURENCYJNE PROCEDUR SZABLONOWYCH	62
10.6	PROCEDURA SPECJALIZOWANA	63
10.7	JEDEN SZABLON SZCZEGÓLNYM PRZYPADKIEM DRUGIEGO	65
11.	DODATEK B – OPIS TECHNICZNY	66
12.	DODATEK C - SŁOWNIK UŻYTEJ TERMINOLOGII I SKRÓTÓW.....	67
13.	DODATEK D – PLIKI ZMIENIONE LUB DODANE DO KODU ŹRÓDŁOWEGO SYSTEMU ODRA	68

1. WSTĘP

Prezentowana praca dotyczy inżynierii oprogramowania i baz danych. Konkretny problem rozpatrywany w pracy dotyczy budowy obiektowego języka programowania w podejściu stosowym. Praca rozszerza możliwości systemu zarządzania bazą danych ODRA, skonstruowanego przez naukowców z Polsko-Japońskiej Wyższej Szkoły Technik Komputerowych. Zaproponowane rozszerzenia dotyczą m.in. możliwości używania konstrukcji szablonowych do generowania nowych procedur w języku SBQL zaimplementowanym w systemie ODRA. Rozszerzenia wprowadzają także niezbędne dodatkowe mechanizmy, takie jak przeciążanie nazw funkcji i prekompilator.

1.1 Cel pracy

Głównym celem pracy jest implementacja mechanizmu procedur szablonowych w systemie ODRA. Powodem, dla którego autor podjął się niniejszego zadania jest brak podobnej implementacji we wspomnianym systemie oraz chęć uczestnictwa i nabycia doświadczenia w pracy przy zaawansowanym projekcie łączącym inżynierię oprogramowania, bazy danych i języki programowania.

1.2 Rezultat pracy

Praca składa się z 3 części:

1. Część implementacyjna – obejmuje źródła systemu ODRA rozszerzone o mechanizm generowania procedur szablonowych (folder „kod źródłowy”) wraz z plikami wynikowymi¹.
2. Część merytoryczna – to niniejszy dokument powstały po fazie implementacji, dokładnie opisujący wszystkie aspekty związane z mechanizmem szablonów.

¹ Uruchomienie plików wynikowych oraz szerszy opis techniczny znajduje się w dodatku „Opis techniczny”

3. Przykłady użycia – wszystkie omawiane zagadnienia odnośnie procedur szablonowych w systemie ODRA są potwierdzone praktycznymi przykładami użycia, które czytelnik może uruchomić na dołączonej wersji systemu (folder „Przykłady” oraz dodatek „Przykłady kodu SBQL”).

1.3 Organizacja pracy

Praca rozpoczyna wyjaśnienie, czym jest mechanizm szablonów w językach programowania. Wiele języków jest wyposażonych w mechanizm funkcji szablonowych, ale rozwiązania w tym zakresie nieco się różnią. Dlatego w celu zrozumienia ogólnej idei oprzemy się o konkretne rozwiązanie znane z języka C++.

Następnie, omówimy podstawowe pojęcia z zakresu architektury stosowej i systemu ODRA, dla którego autor tej pracy pisał rozszerzenie.

Opis implementacji został podzielony na trzy rozdziały. Pierwszy opisuje szablony w sposób ogólny. W szczególności omawia jakie nowe elementy musimy wprowadzić do systemu oraz jak utworzyć przykładowy szablon. W drugim rozdziale jest omówiony w szczegółowy sposób proces generowania procedur na podstawie zdefiniowanego szablonu. Następnie są rozpatrzone dalsze aspekty stosowania omawianego mechanizmu oraz zaproponowane są dalsze możliwości rozwoju. Trzeci rozdział stanowi podsumowanie całej pracy, w którym omówione są zalety oraz wady opracowanego rozwiązania, a także wnioski autora.

1.4 Od autora

Niniejsza praca nie byłaby możliwa, gdyby nie wykłady: „Języki i środowiska programowania baz danych” oraz „Konstrukcja obiektowych rozproszonych baz danych”, których autorem jest prof. dr hab. inż. Kazimierz Subieta w Polsko-Japońskiej Wyższej Szkole Technik Komputerowych. Były one inspiracją do podjęcia niniejszego tematu. Praca jest dopełnieniem i praktycznym wykorzystaniem wiedzy zdobytej na wykładach a jej realizacja znacznie poszerzyła umiejętności autora w zakresie szeroko rozumianej inżynierii oprogramowania i baz danych.

2. SZABLONY FUNKCJI – WSTĘP

Szablon (ang. *template*) funkcji lub funkcja szablonowa definiuje kod, który może być realizowany dla różnych typów danych. Innymi słowy (jak sama nazwa wskazuje), dzięki szablónowi możemy stworzyć rodzinę funkcji różniącą się tylko typem argumentów formalnych i zmiennych wewnątrz ciała. Koncepcja ta po raz pierwszy została zaimplementowana w języku ADA¹ w roku 1970, lecz dopiero język C++ przyniósł jej popularność. W języku tym jest to jeden z dwóch rodzajów szablónów, dzięki któremu możliwe jest automatyczne generowanie nowych funkcji (drugi sposób dotyczy klas). Reprezentacja ta przypomina zwykłą funkcję języka C++, ale niektóre jej elementy są zastąpione przez parametry. Najlepiej wytłumaczymy to na prostym przykładzie (język C++). Poniższy kod stanowi deklarację szablónu funkcji, zwracającej większą z dwóch przekazanych do niej wartości.

```
template <typename T>
T max (T a, T b)
{
    return a > b ? a : b;
}
```

Szablon ten definiuje rodzinę funkcji, które zwracają większą z dwóch przekazanych do nich wartości (przekazanie odbywa się za pośrednictwem parametrów wywołania funkcji: a i b). Typ parametrów funkcji nie został jawnie określony i występuje w szablonie jako parametr szablónu T . Przykład pokazuje, że parametry szablónu muszą być podane w ramach następującej konstrukcji składniowej:

```
template < lista_parametrow >
```

Nazwą parametru typu w naszym przykładzie jest T . Nazwa ta może być dowolnym ciągiem dopuszczalnym z punktu widzenia zasad konstruowania identyfikatorów języka. Dla parametru typu przyjęło się stosować nazwę T . Parametr

¹ Ada to język programowania opracowany przez Jean Ichbiaha w latach 70-tych XX wieku. Język ten wygrał konkurs zorganizowany przez Departament Obrony USA, pokonując 19 innych projektów.

typu reprezentuje typ umowny precyzowany przez programistę dopiero w miejscu wywołania funkcji. Wywołanie może określać dowolny typ (prymitywny, klasę itp.) pod warunkiem, że dla tego typu zdefiniowane są wszystkie operacje wykorzystywane w szablonie funkcji.

W prezentowanym przykładzie typ ten musi więc obsługiwać operator relacji „większe niż” (>), ponieważ parametry przekazywane do funkcji są porównywane właśnie za pomocą tego operatora.

Sygnalem dla kompilatora, by wygenerował daną funkcję szablonową, jest miejsce w programie, gdzie taką funkcję wywołujemy. Wywołanie funkcji szablonowej wygląda tak, jakbyśmy robili to dla zwykłej funkcji. Zatem wywołanie: `max(1, 3)` spowoduje wygenerowanie i wykonanie następującej funkcji:

```
int max (int a, int b)
{
    return a > b ? a : b;
}
```

`max(3.4, 4.5)` wyprodukuje i wykona funkcje dla wartości *double*:

```
double max (double a, double b)
{
    return a > b ? a : b;
}
```

Jak widać, uzyskaliśmy sytuację, gdy w tym samym zakresie ważności nazw mamy więcej niż jedną funkcję o tej samej nazwie. Aby obsłużyć tę sytuację musimy skorzystać z mechanizmu przeciążenia nazw funkcji.

2.1 Przeciążenie nazw funkcji

W języku angielskim przeciążenie (*overloading*) jakiegoś słowa oznacza, że ma ono więcej niż jedno znaczenie. Zjawisko to występuje także w niektórych językach programowania, gdzie można przeciążać nazwy funkcji, w szczególności, w PL/I, C++, Forth, Object Pascal, Java, C# i wielu innych językach.

Przeciążenie nazw funkcji polega na tym, że w danym zakresie ważności nazw jest więcej niż jedna funkcja o takiej samej nazwie. To, która z nich zostanie

w danym wypadku uaktywniona, zależy od typów argumentów wywołania. Funkcje takie mają tę samą nazwę, ale muszą się różnić liczbą lub typem poszczególnych argumentów.

Przykład (język SBQL):

```
ustaw_współrzędne(x : real; y : real; nazwa : string) { . . . }  
ustaw_współrzędne(xy : Współrzędna; nazwa : string) { . . . }
```

Oczywiście błędem byłoby zdefiniowanie dwóch procedur o takich samych nazwach i takiej samej liście argumentów. Przykładowo, jeżeli do powyższego kodu dodamy:

```
ustaw_współrzędne(wysokość : real; średnica : real; kod:string):  
boolean
```

wówczas kompilator zgłosi nam błąd kompilacji, ponieważ procedura o takiej nazwie i takich parametrach już istnieje. Natomiast, jeżeli w deklaracji przedstawimy kolejność parametrów:

```
ustaw_współrzędne(kod : string; wysokość : real; średnica:real):  
boolean
```

uzyskamy poprawną sytuację.

Zatem widzimy, że kolejność parametrów jest istotna w kontekście przeciążeń, co daje kolejną regułę ustalania nazwy procedury. Podsumowując stwierdzamy, że identyfikacja procedury to nazwa, typy jej argumentów formalnych i ich kolejność. Zauważmy, że w regule tej nie jest ujęty zwracany typ (w naszych przykładach niektóre procedury nic nie zwracają, inne zaś zwracają *boolean*). Powód, dla którego tak się dzieje, staje się trywialny jeżeli ustalimy, w jaki sposób kompilator interpretuje napotkaną nazwę procedury. Nie ma w tym szczególnej finezji, jest to proste sklejenie łańcuchów znakowych, czyli nazwy procedury z nazwami typów z listy argumentów. Przykładowo dla procedur:

```
akcja(x : integer; y : real) {...}  
akcja(x : real; y : integer) {...}
```

zostaną wygenerowane nazwy:

- *akcja\$args_integer_real*
- *akcja\$args_real_integer*

W rezultacie, w programie są teraz procedury o zupełnie innych nazwach. Rozumiemy teraz, dlaczego przeciążone procedury nie mogą się różnić jedynie typem zwracanym - informacja o typie zwracanym nie jest konkatelowana do nazwy. Powodem dla którego ignorujemy zwracany rezultat jest fakt, że dozwolone

jest wywoływanie procedur bez zapamiętywania rezultatu. Załóżmy na chwilę, że możliwe jest rozróżnianie procedur po samym zwracanym typie. Zdefiniujmy dwie procedury o identycznych nazwach i argumentach formalnych, z tym, że jedna zwraca typ *real* a druga *integer*:

```
akcja(x : integer; y : real) : real {...} // niedozwolona
akcja(x : integer; y : real) : integer {...} // konstrukcja
```

Która z nich ma być użyta w przypadku wywołania, w którym nie zapamiętujemy zwracanej wartości:

```
akcja(22; 22.5);
```

Niestety, nie wiadomo. Programista musiałby zakomunikować (np. operatorem rzutowania), o którą wersję mu chodzi. Nie mniej jednak istnienie takiej konstrukcji byłby wątpliwy i mógłby doprowadzić do błędnych sytuacji w programie.

2.2 Przeciążenie a referencja

Rozważmy następujące procedury o nazwach (w komentarzach nazwy zapamiętane w programie):

```
Daj_podwyżkę(dział : ref Dept; ile : real) : boolean
Daj_podwyżkę(pracownik : Emp; ile : real) : boolean
```

Mimo tego, że argument *Dept* w pierwszej jest przekazany przez referencje a *Emp* przez wartość, to nazwy będą utworzone tą samą regułą, bez uwzględniania faktu referencji. W powyższym przypadku:

Daj_podwyżkę $\$args_Dept_real\$$ - argument *Dept* przekazany przez referencje

Daj_podwyżkę $\$args_Emp_real\$$ - argument *Emp* przekazany przez wartość

Zatem nie jest możliwe przeciążenie dwóch funkcji różniących się jedynie sposobem przekazania wartości, np.:

```
func(a : Typ);
func(a : ref Typ);
```

Oprócz bezpieczeństwa (niewątpliwie istnienie dwóch takich procedur budzi wątpliwości), uzyskaliśmy również jasną sytuację w przypadku tworzenia nazwy procedury na podstawie wołania z argumentem referencyjnym. Zawsze ustalamy typ referencji. Przeanalizujmy wołanie:

```
Daj_podwyżkę(Dept where dName = "IT"; 2000);
```

posiadające argumenty aktualne:

- **referencje** do obiektu *Dept*,
- *integer* równy 2000.

W celu wygenerowania nazwy procedury, musimy ustalić, z jakimi typami mamy do czynienia. O ile w przypadku *integer* nie musimy wykonywać żadnych specjalnych zabiegów, o tyle w przypadku referencji musimy przeprowadzić operację dereferencji i sprawdzić z jakim obiektem pracujemy. W dalszych krokach postępujemy jak wcześniej, czyli nazwę typów używamy do produkcji nazwy wywołanej procedury. W tym przypadku *Dept*, zatem finalnie wywołamy: *Daj_podwyżkę\$args_Dept_integer\$*.

2.3 Konwersja niejawna

Konwersja niejawna to konstrukcja programistyczna dostępna w określonym języku programowania, umożliwiająca traktowanie danej pewnego, konkretnego typu jako danej innego typu. Odwołując się do języka Java, jeżeli funkcja przyjmuje argument *double*, a my wywołamy ją z argumentem *integer*, to nastąpi niejawna konwersja z *integer* na *double*.

Rozważmy powyższą sytuację w SBQL:

```
Daj_podwyżkę(dział : Dept; ile : real): boolean { ... }
Daj_podwyżkę(Dept where dName = "IT"; 2000);
```

Wywołanie takiej nazwy spowoduje błąd, ponieważ nie mamy procedury o parametrach *Dept*, *integer*. Poprawne wywołanie jest następujące:

```
Daj_podwyżkę(Dept where dName = "IT"; (real)2000);
```

Konwersja niejawna tutaj nie zachodzi. Należy explicite rzutować wartość na żądany typ. Co musielibyśmy zrobić, aby konwersja niejawna zachodziła? Przede wszystkim zdefiniować, jakie typy mogą w bezpieczny, niejawny sposób przejść proces konwersji, bo o ile konwersja *integer* w *double* jest operacją bezpieczną, o tyle odwrotna sytuacja jest niedopuszczalna (przynajmniej w niejawnej wersji). Również musimy wziąć pod uwagę konwersję typu zdefiniowanego przez programistę, np. *Student* w *Osoba*, który jest klasą, abstrakcyjną, odziedziczoną przez *Student*. Oprócz określenia takich reguł, musimy również zmodyfikować mechanizm generowania nazw na podstawie wołania. Zatem próbujemy najpierw

operację *bind* dla nazwy wygenerowanej na podstawie argumentów, jeżeli dostaniemy pusty rezultat, to zamienimy nazwę typu na tę, w którą może niejawnie przejść. Na nowo wygenerowanej nazwie wykonujemy operację *bind*.

Całość wydaje się stosunkowo prosta, jednakże może powodować trudności. Przeanalizujmy przykładową funkcję i jejwołanie:

```
func(x : real; y : real; z : real; height : real; width : real)
{...}
func(1; 2; 1; 100; 50)
```

W pesymistycznym przypadku musimy zbadać aż 32 kombinacji typów *real* i *integer*. Ogólnie, jest to złożoność obliczeniowa rzędu $O(n^m)$, gdzie n jest liczbą reguł konwersji a m ilością konwertowanych typów (w powyższym przypadku $n=2$, zaś $m=5$). Dla bardziej skomplikowanych funkcji, używanych np. w grafice możemy mieć problem z nadmierną liczbą kombinacji do zbadania.

Z tego powodu nie pozwalamy na niejawne konwersje, wymagając od programisty jawnych rzutowań. Jednakże nie jest to decyzja ostateczna i z tego wymagania możemy zrezygnować.

2.4 Kiedy przeciążać?

Przeciążenia nazw funkcji są cennym udogodnieniem. Szczególnie warto z nich korzystać wtedy, gdy procedury posiadają pewną cechę wspólną (np. kilka procedur wyświetlających informację na ekranie), a mają działać na różnych typach obiektów. Procedury w sensie abstrakcyjnym wykonują analogiczną czynność, która w języku naturalnym ma tę samą nazwę (np. dodawanie: liczb rzeczywistych, liczb zespolonych, wektorów, macierzy, itp.). Jak już wcześniej wspomnieliśmy, przeciążeń używamy *implicite*, gdy korzystamy z zaimplementowanego mechanizmu szablonów.

3. ARCHITEKTURA STOSOWA I SYSTEM ODRA

Rozdział ten ma na celu przybliżenie czytelnikowi podstawowych koncepcji, na bazie których powstała niniejsza praca. Zostanie tu omówiona architektura stosowa (SBA - Stack Based Architecture) oraz system ODRA (Object Database for Rapid Application development), który autor rozszerzył o możliwość stosowania funkcji szablonowych.

Podejście stosowe do języków zapytań oraz język SBQL zostało opisane na podstawie książki K. Subiety „*Teoria i konstrukcja obiektowych języków zapytań*”, jak również na podstawie rozprawy doktorskiej M. Lentnera „*Integracja danych i aplikacji przy użyciu wirtualnych repozytoriów*”. Również bardzo pomocne były treści wykładów z przedmiotu „*Języki i środowiska programowania baz danych*”, prowadzone przez K. Subietę. W rozdziale tym omówimy tylko podstawowe pojęcia, na których opiera się praca. Po dokładniejsze informacje odsyłam do wyżej wymienionej książki.

3.1 Co to jest SBA?

Podejście stosowe (SBA - Stack-Based Approach) do języków zapytań zakłada, że języki zapytań są szczególną formą języków programowania. SBA jest semantyczną ramą umożliwiającą budowę mocnego języka dla praktycznie dowolnego modelu danych, niezależnie od tego czy jest to model ustrukturalizowany (np. relacyjne bazy danych), nieustrukturalizowany (np. dane multimedialne) czy półstrukturalny (np. XML). Przykładowo, obsługa danych zawartych w dokumencie XML jest możliwa poprzez zaimportowanie zawartości pliku XML do bazy danych, a następnie wyrażenie jego treści w terminach obiektów tej bazy danych. Na danych takich możemy pracować w ten sam sposób jak na danych zdefiniowanych za pomocą języka definiowania danych.

Co więcej, w podejściu stosowym, dzięki przyjęciu założenia, że zapytania traktowane są w ten sam sposób jak wyrażenia w tradycyjnych językach programowania, udało się zbudować kompletny język programowania bezszwowo zintegrowany z konstrukcjami języka zapytań. Zatem semantyka zapytań na równi

z semantyką wyrażen jest oparta na takich mechanizmach jak stos środowiskowy i stos rezultatów, oraz podlega tym samym regułom zakresów nazw oraz ich wiązania. Krokiem dalej jest precyzyjne określenie semantyki operacyjnej języków zapytań z uwzględnieniem cech obiektowości (np. klasy), konstrukcji imperatywnych i abstrakcji programistycznych (np. moduły, perspektywy, procedury).

3.2 Obiekt i jego tożsamość

„Obiekt (object) jest abstrakcyjnym bytem reprezentującym lub opisującym pewną rzecz lub pojęcie obserwowane w świecie rzeczywistym. Obiekt jest odróżnialny od innych obiektów, ma nazwę i dobrze określone granice.” [K. Subieta 2004]. Przykładami obiektów są: pracownik John Smith, komputer PC, książka o ISBN 978-0-12-345678-9, rozdział „O autorze”, hasło „obiektość” w encyklopedii, wynik losowania loterii z dnia 2009-06-12 itp.

Model obiektowy w przeciwieństwie do relacyjnego nie zakłada konieczności nadawania obiektowi atrybutu (lub zbioru atrybutów), które jednoznacznie go identyfikują, czyli tzw. „klucza głównego”. W obiektości zakładamy, że każdy obiekt posiada swoją „tożsamość”, tzn. możemy mieć dwa obiekty o takich samych wartościach, ale oba „odróżniają się” od siebie, gdyż posiadają unikalne „wewnętrzne identyfikatory” (object identifier, *OID*). Inaczej mówiąc, nie mogą istnieć dwa obiekty posiadające ten sam identyfikator. *OID* jest nadawany przez system automatycznie w momencie powstawania obiektu, niezależnie od woli programisty czy projektanta. Identyfikator może być zwrócony jako wynik zapytania, procedury a następnie użyty np. jako argument operacji „usuń” lub zapamiętany jako wartość zmiennej lub obiektu.

3.3 Modele danych

Podejście stosowe jest przystosowane do wielu uniwersalnych modeli składu danych. W zależności od stopnia złożoności wyróżniamy następujące modele: M0, M1, M2, M3, gdzie każdy kolejny typ dodaje nowe właściwości do poprzednika.

M0 – Model ten jest zbudowany zgodnie z zasadą relatywizmu oraz wewnętrznej identyfikacji. Model ten jest bardzo prosty, gdyż każdy obiekt składa

się z wewnętrznego identyfikatora (ukrytego przed programistą), zewnętrznej nazwy (dostępnej dla programisty) oraz wartości. W zależności od charakteru przechowywanej wartości mamy trzy rodzaje obiektów:

- atomowe – $\langle ii, n, v \rangle$ - gdzie ii należy do zbioru dopuszczalnych identyfikatorów wewnętrznych, n należy do zbioru dopuszczalnych nazw zewnętrznych obiektów, a v należy do zbioru wszystkich dopuszczalnych wartości prostych, takich jak liczby, itp. Są to najprostsze rodzaje obiektów.
- referencyjne $\langle ii, n, ij \rangle$ - gdzie ii i n oznaczają to samo, co wyżej, a wartością ij jest identyfikator innego obiektu, wskazywanego przez ten obiekt. Służą do odwzorowania powiązań pomiędzy obiektami.
- złożone - $\langle ii, n, O \rangle$ - gdzie ii i n są zdefiniowane jak wyżej, a wartość O jest zbiorem obiektów należących do modelu M0. Służą do modelowania zagnieżdżonych obiektów.

Przykład:

```
<i0, entry,  
  <i1, Student,  
    <i4, Imię, "O. Holy">  
    <i5, Rok, 1>  
    <i99, Kierunek, i3>  
  >  
  <i2, Student,  
    <i6, Imię, "K. Chief">  
    <i7, Rok, 5>  
    <i100, Kierunek, i3>  
  >  
  <i3, Kierunek,  
    <i8, Nazwa, "Informatyka">  
    <i9, Uczelnia, "P JWSTK">  
  >  
>
```

M1 – uzupełnia M0 o pojęcia klasy, dziedziczenia, jak również relacji przynależności obiektu do klasy.

M2 – wprowadza pojęcie dynamicznej roli obiektu, czyli w czasie wykonania obiekt może zyskiwać nowe role i tracić stare, zaś dziedziczenie pomiędzy rolami ma charakter dynamiczny.

M3 – uzupełnia model M1, M2 o pojęcie hermetyzacji (dzieląc własności obiektów na publiczne i prywatne).

3.4 Stosy

Istotą podejścia stosowego jest zastosowanie dwóch stosów do objaśnienia semantyki języka zapytań, czyli stosu rezultatów oraz stosu środowiskowego. Pierwszy z nich, stos rezultatów (QRES - Query Result Stack), służy do przechowywania tymczasowych oraz końcowych rezultatów zapytań. Np. wyrażenie arytmetyczne: $((14 + 6) / 10 + (14 - 4) * 10) / 2$, którego postać w ONP możemy zapisać jako: $14\ 6\ +\ 10\ /\ 14\ 4\ -\ 10\ *\ +\ 2\ /\$, będziemy mieli następujące rezultaty ewaluacji zapytania pojawiające się na QRES:

						4 (-)		10 (*)				
	6(+)		10(/)		14	14	10	10	100(+)		2(/)	
14	14	20	20	2	2	2	2	2	2	102	102	101

Rysunek 1. Stan stosu rezultatów dla wyrażenia arytmetycznego

Do zbioru wszystkich możliwych rezultatów, które może przechowywać QRES mogą należeć następujące elementy:

- wartość atomowa (literały, np. "napis", 33.3, *false*) – referencja do obiektu (czyli identyfikator wewnętrzny każdego obiektu, OID).
- binder – czyli para $\langle n, x \rangle$, gdzie n jest nazwą obiektu, zaś x jest bytem czasu wykonania (zazwyczaj referencją). Binder jest podstawową strukturą przechowywaną na stosie. Binder $\langle n, x \rangle$ będziemy zapisywać w postaci $n(x)$.
- struktura – ciąg pojedynczych rezultatów, którą możemy utworzyć konstruktorem $struct\{x_1, x_2, x_3, \dots\}$ lub za pomocą samego przecinka (x_1, x_2) . W strukturze kolejność występowania elementów jest istotna.
- kolekcje rezultatów – elementem kolekcji może być dowolny rezultat za wyjątkiem innej kolekcji. Kolekcje najczęściej powstają jako wynik tworzenia nazw lub też jako wynik operatorów zbiorowych (np. *union*).

Mamy dwa rodzaje kolekcji: $bag\{x_1, x_2, x_3, \dots\}$, która nie zachowuje kolejności występowania lub $sequence\{x_1, x_2, x_3, \dots\}$, która zachowuje kolejność występowania.

Przykład rezultatów zapytań:

- atomowe: "Ola", 20.24, *true*, i_{12} , $x(i_{12})$
- złożone:
 - $struct\{i_{12}, i_{23}, 9\}$,
 - $sequence\{i_1, i_2, i_3\}$,
 - $bag\{struct\{i_{12}, i_{23}, 9\}, struct\{i_{13}, i_{23}, 20\}, struct\{i_{14}, i_{23}, 33\}, \}$,
 - $bag\{struct\{student("Wysocki"), Rok(5), Uczelnia(i_{100})\}\}$.

Drugim stosem jest stos środowiskowy (ENVS - Environment Stack), który w szczególności służy do kontrolowania zakresów nazw. Na stosie umieszczane są sekcje, w skład których wchodzi bindery, informujące nas o aktywnych bytach programistycznych czasu wykonania (obiektach, procedurach, zmiennych), dostępnych w danym punkcie sterowania programu komputerowego.

Oczywiście środowisko obliczeniowe nie jest stałe. Przyjmuje się, że jest ono podzielone na pod-środowiska (są to wspomniane sekcje). Na początku ewaluacji zapytania stos składa się z tak zwanej sekcji bazowej, w której przechowujemy bindery do obiektów korzeniowych. W trakcie ewaluacji zapytania, na stosie pojawiają się i znikają dodatkowe sekcje, ale sekcja bazowa zawsze na nim zostaje.

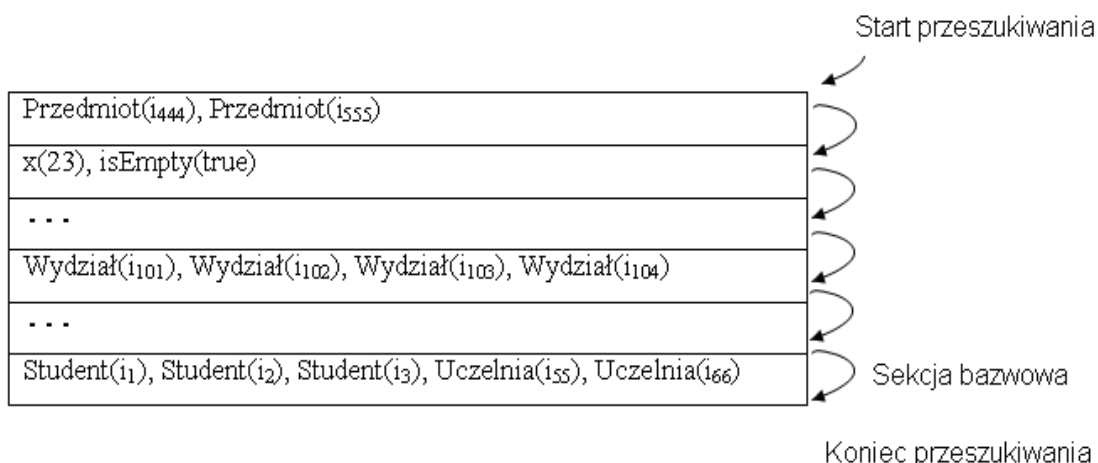
Wiązanie nazw, czyli operacja **bind**, realizowane jest na stosie środowiskowym. Polega ono na przeszukiwaniu stosu w kierunku od czubka w dół w celu znalezienia pierwszej takiej sekcji, w której występuje przynajmniej jeden binder o szukanej nazwie. W przypadku, gdy w napotkanej sekcji znajduje się więcej niż jeden binder odpowiadający wiązanej nazwie, zwracamy kolekcję złożoną z wyszukanych binderów. W szczególności, jeżeli nic nie znaleźliśmy w żadnej sekcji to zwracamy pustą kolekcję (lub podnosimy błąd typologiczny).

Przykład: Dla stosu przedstawionego na Rys.2 operacja *bind* zwróci następujące wyniki:

$bind("Wydział") = bag\{i_{101}, i_{102}, i_{103}, i_{104}\}$

$bind("x") = 23 = bag\{23\}$

$bind("kot") = bag\{\}$



Rysunek 2. Kolejność przeszukiwania ENVS podczas wiązania nazwy

3.5 SBQL w systemie ODRA

Przykładem obiektowego języka programowania zbudowanego zgodnie z architekturą stosową, z mocno zintegrowanym językiem zapytań jest język SBQL. Stanowi on jądro, wokół którego zbudowany jest system zarządzania obiektową bazą danych ODRA. System ten jest przykładem udanej implementacji (w języku Java) teorii opartej na SBA. Głównym celem projektu ODRA jest stworzenie nowej idei tworzenia aplikacji bazodanowych przez podnoszenie poziomu abstrakcji, na którym pracują programiści. Zapytania przyjmują formę wyrażeń, a powiązania między konstrukcjami imperatywnymi a deklaratywnymi można określić jako bezszwowe.

3.5.1 Moduły

Podobnie jak w klasycznych językach programowania, moduł jest podstawową jednostką organizacyjną programu, grupującą byty programistyczne i posiadającą dobrze zdefiniowany interfejs. Po zakończeniu pracy przez jednego programistę inni programiści mogą używać dowolnych publicznych właściwości danego modułu, ale nie mogą używać i zmieniać właściwości odnoszących się do jego wnętrza. Z punktu widzenia koncepcji obiektowości, moduł jest obiektem zawierającym w sobie inne obiekty oraz inne właściwości, takie jak typy lub klasy.

3.5.2 Procedury

Procedury w systemie ODRA mają charakter typowy dla języków programowania. Głównym celem procedur jest hermetyzacja dowolnie skomplikowanych obliczeń, ponieważ wewnątrz procedur jest niedostępne z zewnątrz. Procedury mogą być wywoływane z wielu miejsc, zaś ich przystosowanie do konkretnego celu najczęściej następuje poprzez określenie ich parametrów lub przez efekty uboczne. Podobnie jak większość elementów w module, procedury w systemie ODRA są bytami pierwszej kategorii, co oznacza, że możemy ich używać, dodawać i modyfikować w trakcie działania programu. Procedury możemy podzielić na procedury właściwe i procedury funkcyjne (zwane funkcjami, jako naturalne skojarzenie z terminem matematycznym). Pierwsze z nich nie zwracają wyniku, zatem nie mogą być używane jako składnik wyrażenia, zaś procedury funkcyjne zwracają wynik i przez to mogą być używane jako składniki wyrażen. W niniejszej pracy przyjmujemy uproszczone nazewnictwo, stosując określenia „procedura” lub „funkcja” jako jej równoważnik.

4. DEKLARACJA SZABLONÓW W SYSTEMIE ODRA

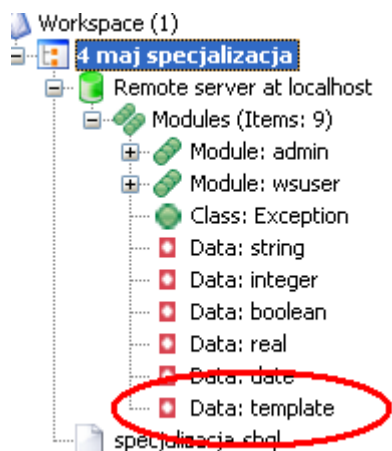
Niniejszy rozdział jest wstępem do szerszego omawiania szablonów w systemie ODRA. W poniższych podrozdziałach omówimy sposób deklaracji szablonu oraz pojęcia i czynności niezbędne przy tym procesie.

4.1 Typ szablonowy.

Przed dokładnym omówieniem deklaracji szablonu potrzebujemy zdefiniować pojęcie typu szablonowego. Typ szablonowy (ang. *template type*) jest umownym typem znajdującym się tylko i wyłącznie w definicji szablonu. Jego deklaracja znajduje się w nagłówku szablonu, dzięki czemu możemy go używać w liście argumentów formalnych, w ciele, czy jako definicji zwracanych rezultatów. Użycie go w szablonie jako typu nie implikuje sprawdzenia poprawności użycia lub kontroli typów podczas kompilacji. Kiedy go używamy, nie wiemy z jakim typem mamy dokładnie do czynienia. Dlatego na zmiennej typu szablonowego możemy wywoływać dowolne metody składowe lub wykonywać dowolne operacje – typ szablonowy na to pozwala.

Podczas generowania procedury na podstawie szablonu następuje podmiana typu szablonowego na konkretne typy podane podczas wołania proceduryⁱ.

Na potrzeby prac implementacyjnych został dodany typ szablonowy o nazwie *template* jako typ prymitywny.



Rysunek 3. Typ szablonowy jako prymityw w systemie ODRA

ⁱ Proces ten zostanie dokładnie omówiony w następnym rozdziale

Każde użycie przez programistę w kodzie słowa kluczowego *template* jako typu:

```
var : template;
```

będzie się kończyć błędem parsowania:

„*Unexpected token (template) var : template;*”

Dzieje się tak dlatego, że słowo kluczowe *template* celowo nie zostało uwzględnione w parserze, gdyż nie jest ono dostępne programiście *explicite*. Typ szablonowy możemy deklarować tylko w określony sposób wewnątrz systemu ODRA, podczas deklaracji samego szablonu i tylko na jego potrzeby.

4.2 Składnia procedur szablonowych

Składnia deklaracji szablonów przypomina składnię deklaracji znanych nam już procedur, z tą różnicą, że dodajemy nagłówek ze słowem kluczowym *template* oraz listą parametrów, które określają typy szablonowe, te zaś mogą wystąpić w liście argumentów formalnych, ciele szablonu, czy jako zwracany rezultat.

W wołaniu procedury można umieszczać zmienne dowolnego typu, zgodne z logiką przeznaczenia szablonu. Wołanie procedury szablonowej niczym się nie różni od wołania zwykłej procedury. Przyjrzyjmy się gramatyce konstrukcji umożliwiającej definiowanie szablonów:

Definicja szablonu:

```
procedura_szablonowa ::=
    nagłówek_szablonowy procedura

nagłówek_szablonowy ::=
    template(parametrySzab)

parametrySzab ::=
    parametrSzab | parametrSzab; parametrySzab

parametrSzab ::=
    type nazwa

procedura ::=
    nazwaProc() { instrukcje }

procedura ::=
    nazwaProc() : typZwracany { instrukcje }
```

procedura	::=	nazwaProc(parFormalne) { instrukcje }
procedura	::=	nazwaProc(parFormalne):typZwracany { instrukcje }
typZwracany	::=	nazwa
nazwaProc	::=	nazwa
parFormalne	::=	parFormalny parFormalny; parFormalne
parFormalny	::=	nazwa in nazwa out nazwa
instrukcje	::=	instrukcje_SBQL return [zapytanie]

Przykład:

```

template (type T)
suma(skladnik1 : T; skladnik2 : T): T
{
    Sum : T;
    sum := skladnik1 + skladnik2;
    return sum;
}

```

Wolanie szablonu:

wolanieProc	::=	nazwaProc () nazwaProc (parAktualne) ;
parAktualne	::=	parAktualny parAktualny; parAktualne
parAktualny	::=	zapytanie

Przykład:

```
suma (12; 22) ;
```

4.3 Deklaracja szablonuⁱ

Jak widać, deklaracja procedury szablonowej z dokładnością do nagłówka szablonu i typu *template* jest bardzo podobna do deklaracji zwykłej procedury. W implementacji w systemie ODRA skorzystamy z tego faktu. Przeanalizujemy to na przykładowo zdefiniowanych szablonach:

```
template (type T, type R)
procedura_szablonowa(a : T): T
{ ... }

template (type T, type R)
procedura_szablonowa(a : T; b : R): T
{ ... }
```

Deklarację szablonu możemy podzielić na trzy etapy:

Etap 1. Deklaracja typu szablonowego (template).

Pierwszą czynnością, o której musimy pamiętać jest konieczność zakomunikowania kompilatorowi faktu, że typy T i R są typami szablonowymi w ramach deklarowanego szablonu. Jest to robione w sposób automatyczny, bez wiedzy programisty deklarującego szablon. Posłużymy się do tego istniejącą konstrukcją z systemu ODRA, a mianowicie:

```
type T is template;
```

Konstrukcja ta oznacza, że od tej pory nasz typ szablonowy T będzie traktowany na równi z typem *template*.

Jak pamiętamy, użycie takiego wyrażenia w kodzie SBQL zakończyłoby się błędem parsowania – nie możemy użyć w sposób *explicite* słowa *template*. Jednakże wewnątrz, podczas konstrukcji szablonu, możemy wywołać taką deklarację typu szablonowego. Zatem, dla parametrów T oraz R wykonujemy powyższą instrukcję, dzięki czemu możemy używać nazwy T jako typu – kompilator wie już o nich dostatecznie dużo. Gdybyśmy takiej instrukcji wewnątrz systemu nie wykonali, to

ⁱ Ponieważ w systemie ODRA deklaracja procedury/szablonu jest ściśle powiązana z definicją, zatem będziemy posługiwać się tylko pierwszym określeniem.

użycie typu *T* w ciele szablonu, liście argumentów formalnych, czy też jako zwracany rezultat, zakończyło by się błędem kompilacji.

Etap 2. Generowanie nazwy.

Mimo dużego podobieństwa między zwykłą procedurą a szablonem, musimy w pewien sposób oznaczyć fakt, że dana procedura jest szablonowa. Możemy to zrobić na kilka sposobów, np. ustawić specjalną „flagę” w obiekcie procedury mówiącą o „rodzaju” procedury. Jednakże wewnątrz systemu ODRA istnieje kilka klas odpowiedzialnych za obiekty procedur: *ProcedureDeclaration*, *OdraProcedureSchema*, *DBProcedure*, *MBProcedure*, itp. Dlatego więc, kłopotliwe może się okazać powielanie tej informacji na wszystkie dotychczasowe klasy, a w przypadku definicji nowych wariantów może się okazać, że zapomnimy o nowej właściwości. Zatem zdecydujemy się na rozwiązanie bardziej uniwersalne, a mianowicie do nazwy procedury będziemy konkatenuować jako prefiks słówko *\$TEMPLATE\$*. Każda z wyżej wymienionych klas zawiera nazwę procedury, a więc jak raz naznaczymy naszą procedurę, że jest szablonowa, to informacja ta będzie powielać się po całym systemie. Nadawanie nazwy w ten sposób również upraszcza operację wołania procedur i szukania odpowiedniego szablonu, o czym szerzej powiemy w podrozdziale poświęconym wołaniu.

Kolejną różnicą w generowaniu nazwy jest brak wymienionych argumentów formalnych w nazwie procedury. Powód jest oczywisty: w momencie deklaracji szablonu nie wiemy, z jakimi typami będziemy pracować (również szczegółowo omówimy to we wspomnianym podrozdziale). Nazwy wygenerowane dla naszych przykładowych szablonów to odpowiednio:

\$TEMPLATE\$procedura_szablonowa\$0

\$TEMPLATE\$procedura_szablonowa\$1

Etap 3. Tworzenie szablonu.

Możemy teraz przejść do stworzenia szablonu, który do facto jest pewnego rodzaju procedurą (wzorcem) – dlatego generujemy ją w ten sam sposób, jak zwykłą procedurę. Rezultatem będzie utworzenie specjalnego obiektu w bazie, jak i metabazie.

Musimy pamiętać o wyłączeniu kontroli typów dla szablonu. Szablon traktujemy jako narzędzie do produkcji funkcji i bez konkretnej unifikacji typów nie możemy stwierdzić czy:

- jest poprawny typologicznie,
- można dane operacje wykonać,
- mamy dostęp do danych lub metod w składzie,
- spełnione są inne wymagania kontroli typów.

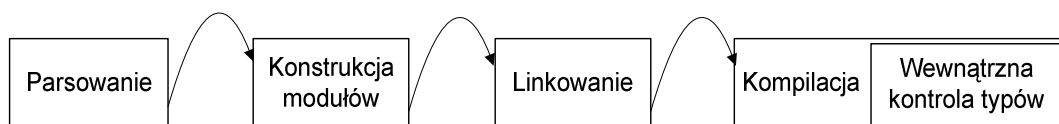
Przykład kodu w systemie ODRA ilustrujący omawiane zagadnienie:
Dodatek A.10.1 (lub dołączony do pracy plik Przykłady/deklaracja.sbql)

5. PROCES TWORZENIA SZABLONÓW W SYSTEMIE ODRA

W kolejnych podrozdziałach omówimy dokładnie sposób realizacji mechanizmu szablonów w systemie ODRA. Zaczniemy od wyboru odpowiedniego „miejsca”, w którym możemy umieścić elementy reagujące na wołania do szablonów, następnie podamy algorytm wyboru właściwego wzorca, a na końcu zamieścimy sposób tworzenia nowej procedury.

5.1 Wybór fazy kompilacji do rozpoczęcia analizy

Proces budowy modułów w systemie ODRA jest procesem złożonym, z którego czerpią informację różne mechanizmy, w szczególności statyczna kontrola typów oraz optymalizacja zapytań. Przedstawimy tę konstrukcję w nieco uproszczony sposób. W pierwszej kolejności kod źródłowy programu trafia do parsera, którego zadaniem jest wyodrębnienie podstawowych jednostek organizacyjnych (np. modułów) oraz wykrycie wszystkich bytów programistycznych należących do nich wraz z zamodelowaniem ich w postaci drzewa AST. Struktura ta jest podstawowym elementem podczas każdej kolejnej fazy. Drzewo AST nadaje się bardzo dobrze do analizy kodu programu, dlatego bez trudu możemy przejść do fazy konstrukcji modułów i bytów w nim zawartych. Całość należy zlinkować i przekompilować. W trakcie kompilacji dochodzi do wewnętrznej kontroli typów. Poniższy schemat przedstawia kolejności wykonywania opisanych etapów:



Rysunek 4. Etapy budowania modułów

Pierwszą kwestią przy produkcji procedury szablonowej jest wyszukanie w kodzie SBQL (drzewie AST) odpowiednich instrukcji, które zainicjalizują mechanizm generacji. Dzieje się to oczywiście podczas zwykłego wołania procedury, w przypadku gdy wołany obiekt nie istnieje w bazie danych, a jesteśmy

w posiadaniu szablonu, który umożliwia daną produkcję. Innymi słowy, aby wygenerować procedurę szablonową, musimy wyszukać wszystkie wołania odnoszące się do nieistniejących procedur i rozstrzygnąć, czy dostępne są narzędzia zdolne do dalszej produkcji. Zastanówmy się, w którym miejscu możemy zainicjalizować taki proces? Aby rozstrzygnąć daną kwestię, omówimy kilka podejść wypróbowanych podczas implementacji w systemie ODRA.

Oczywiście istnieje możliwość zrobienia całości, to jest generowanie procedur z szablonów w podobny sposób, jak w języku C++, czyli poprzez mechanizm prekompilacji opartej na operacjach na ciągach znakowych. Analizator skupia się tylko na kodzie programu jeszcze przed parsowaniem, a procedury są generowane przez zastosowanie prostych technik do obróbki tekstu. Podejście oparte na tej technice nie było jednak celem tej pracy.

Pierwszym podejściem była próba umieszczenia analizatora drzewa AST zaraz po zakończeniu procesu parsowania. Podejście to, w głównej mierze opierało się na generacji nowych węzłów do już istniejącego drzewa AST. Jednakże rozwiązanie to okazało się kłopotliwe i mało elastyczne, między innymi z powodu konieczności wielokrotnego przechodzenia drzewa w celu ustalenia istniejących procedur, wołań, a także problemów z ustaleniem typów i zakresów, w jakich występują (np. w różnych klasach – w przypadku wizji szablonowych metod składowych).

„Podłączenie się” do analizatora wewnętrznej kontroli typów w fazie kompilacji z początku owocowało bardzo obiecującymi rezultatami. Moment kontroli typów jest momentem po konstrukcji modułu i linkowaniu, kiedy cała baza danych jest już praktycznie gotowa. Łatwo nam jest sprawdzać w metabazie istniejące typy, procedury i je stosować. Jednak po operacji dodania nowej procedury zachodzi potrzeba wykonania wszystkich kroków, które normalnie przechodzi kod wprowadzony do systemu ODRA, czyli dodanie, linkowanie metabazy oraz kompilowanie. Kłopot w tym, że chcielibyśmy to zrobić w fazie kompilacji. Dokładniej rzecz ujmując, problem tkwi w linkowaniu metabazy, która w systemie ODRA jest wykonywana na poziomie obiektu (głównie modułu) i polega na inwalidacji wszystkich powiązań (referencji do obiektów reprezentujących drugi koniec metareferencji wyliczonych na podstawie nazwy tego obiektu), a następnie ich regeneracji. Teoretycznie rozwiązanie danego problemu możemy zrealizować zmieniając sposób linkowania tak, aby można było realizować pojedyncze metareferencje. Nie wiadomo jednak czy jest to realizowalne przy obecnych

założeniach systemu ODRA. Natomiast rezygnacja z fizycznego linkowania zapisanego w metabazie i oparcie się o dynamiczne linkowanie w fazie kompilacji jest jak najbardziej realne i potencjalnie uelastyczniające cały system ODRA. Oznacza to zrezygnowanie z fazy linkowania i pozostawienie tylko fazy kompilacji. Jednakże zadanie to jest zagadnieniem bardzo pracochłonnym i tym samym wychodzi poza zakres niniejszej pracy - autor przedstawia je tylko jako teoretyczne rozważanie.

Mając na uwadze powyższy problem z powtórным linkowaniem, spróbujmy zatem wykonać analizę drzewa AST tuż przed tym procesem. Podobnie jak wcześniej, mamy dostęp do obiektów metabazy, z której możemy pobrać informację o istniejących procedurach i trochę uboższą informację o typach (która jest mimo tego wystarczająca). Po generacji całość zostanie razem zlinkowana i przekompilowana, wraz kontrolą typologiczną. To podejście okazało się sukcesem, a rozwiązanie problemu miało kluczowe znaczenie w finalizacji implementacji. Dodatkowo, z inżynierskiego punktu widzenia, lepiej jest stworzyć osobny etap prekompilacji, niż podłączać się do już istniejących procesów i narażać się (z wysokim prawdopodobieństwem) na ich zaburzenie.

Implementacja omawianego prekompilatora¹ znajduje się w dołączonych do pracy rozszerzonych źródłach systemu ODRA:

Kod źródłowy/EGB/src/odra/sbql/precompiler/SBQLPrecompiler.java

5.2 Algorytm wyboru właściwego szablonu

Proces wywoływania procedury jest dla nas najistotniejszym etapem, gdy mówimy o mechanizmie szablonów. To właśnie w momenciewołania procedury następuje generowanie jej na podstawie szablonu, więc tutaj powinniśmy skupić naszą uwagę. Nie byłoby w tym nic szczególnego, gdybyśmy mieli jeden rodzaj szablonu o tej samej nazwie. Jednakże szablony, tak samo jak procedury, podlegają przeciążeniu nazw. Dlatego dopuszczamy możliwość istnienia kilku szablonów o tej samej nazwie, różniące się tylko argumentami formalnymi. Algorytm wyboru właściwego szablonu prześledzimy na poniższym przykładzie:

¹ Precyzyjniejszą nazwą byłaby nazwa ‘prelinkier’, jednakże słowo ‘precompiler’ jednoznacznie kojarzy się z czynnościami wykonywanymi przed kompilacją.

```

//nr 1
    template (type T) // $TEMPLATE$proc$0
    proc(int1 : integer; t1 : T; t2 : T; int2: integer; int3 :
integer): string
    { return "proc$args_integer_T_T_integer_integer$"; }

//nr 2
    template (type T, type R) // $TEMPLATE$proc$1
    proc(str1 : R; int2 : integer; t1 : T; str1 : string; t2 :
T): string
    { return "proc$args_R_integer_T_string_T$"; }

//nr 3
    template (type T) // $TEMPLATE$proc$2
    proc(t1 : T; t2 : T; t3 : T; str1 : string; str2 : string):
string
    { return "proc$args_T_T_T_string_string$"; }

//nr 4
    template (type T) // $TEMPLATE$proc$3
    proc(t1 : T; int1: integer; ok : boolean): string
    { return "proc$args_T_integer_boolean$"; }

//nr 5
    template (type T) // $TEMPLATE$proc2$0
    proc2(i : integer) : string
    { return "proc2$args_integer$"; }

```

Zdefiniowaliśmy cztery szablony o nazwie *proc* (w komentarzu w kodzie ich pełne nazwy, jakie są przechowywane w systemie ODRA) i jeden o nazwie *proc2*. Każdy z nich teoretycznie może nam posłużyć do produkcji różnych zestawów funkcji. Wygenerowane procedury będą zwracać informację, z jakiego szablonu zostały stworzone. Aby zainicjalizować algorytm założmy, że podczas kompilacji natrafiamy na wołanie:

```
proc(3; 4; 5; "Kim"; "Ola");
```

Który szablon pasuje do wołanej procedury? Dla większej czytelności zapiszmy je w postaci tabeli:

Tabela 1. Próba dopasowania wołania do wariantów istniejących szablonów

Lp.	Nazwa	Parametr 1	Parametr 2	Parametr 3	Parametr 4	Parametr 5
1	proc	<i>integer</i>	<i>T</i>	<i>T</i>	<i>integer</i>	<i>integer</i>
2	proc	<i>R</i>	<i>integer</i>	<i>T</i>	<i>string</i>	<i>T</i>
3	proc	<i>T</i>	<i>T</i>	<i>T</i>	<i>string</i>	<i>string</i>
4	proc	<i>T</i>	<i>integer</i>	<i>boolean</i>		
5	proc2	<i>integer</i>				
Wywołujemy:						
	proc	<i>integer</i>	<i>integer</i>	<i>integer</i>	<i>string</i>	<i>string</i>

Na pierwszy rzut oka może się wydawać, że można dopasować procedurę nr 2, bo przecież za typ *T* i *R* możemy wstawić dowolne typy. Nie jest to jednak prawda, gdyż w całej procedurze szablonowej dany typ szablonowy może być unifikowany tylko i wyłącznie na jeden typ w ramach całego szablonu.

W momencie, kiedy raz ustalimy, że *T* przechodzi w *integer*, to w całej procedurze zachodzi taka unifikacja. Ustalenie tego faktu zachodzi podczas analizy argumentów aktualnych wołanej procedury.

Prześledźmy wybór szablonu dla naszego wywołania. W pierwszym kroku konstruujemy nazwę wołanej procedury. Zgodnie z zasadami przeciążeń nazw funkcji wygląda ona następująco:

```
proc$args_integer_integer_integer_string_string$
```

Sprawdzamy czy już nie istnieje taka procedura za pomocą operacji *bind*. Wiemy, że może istnieć procedura specjalizowanaⁱ lub już wcześniej mogliśmy mieć do czynienia z podobnym wołaniem i interesująca nas procedura została wygenerowana.

Jeżeli nie istnieje, to sprawdzamy czy nie istnieje odpowiednia procedura szablonowa. Dzięki specjalnemu nazewnictwu szablonów, czyli bez nazw argumentów w nazwie procedury, możemy to zrobić budując nazwę *\$TEMPLATE\$proc\$0* (nazwa budowana zgodnie z regułą nazewnictwa dla szablonów), jeżeli znaleźliśmy taką, to zapamiętujemy w tablicy identyfikator wewnętrzny (OID) tego szablonu. Następnie powtarzamy ten krok dla kolejnej

ⁱ Dokładniej o procedurach specjalizowanych w systemie ODRA powiemy w osobnym podrozdziale.

nazwy: $\$TEMPLATE\$proc\$1$. Robimy tak dopóty, dopóki **nie** uda nam się operacja *bind* dla kolejno wygenerowanej nazwy. Teraz widzimy korzyści płynące z nazewnictwa szablonów bez argumentów jedynie jako konkatencja: $\$TEMPLATE\$ + nazwa_proc + \$ + kolejna_liczba$. Możemy w szybki sposób wyselekcjonować wszystkie szablony o nazwie użytej podczas wołania. W naszym przypadku dostaniemy tablicę z czterema obiektami OID, czyli de facto z czterema szablonami, które spełniły warunki selekcji po nazwie. Szablon nr 5 - *proc2* słusznie nie zostanie zakwalifikowany.

Jeżeli w tablicy jest przynajmniej jedna referencja (OID) do szablonu, to przechodzimy do etapu dopasowywania typów. Potrzebujemy do tego tablicy asocjacyjnej, która będzie przechowywała elementy: klucz – nazwa typu szablonowego, wartość – nazwa typu, w który przechodzi. Taką tablicę nazwijmy **tablicą przejść** (transition table).

Tablica przejść zostanie utworzona na podstawie analizy argumentów szablonu. Jeżeli argument formalny jest typu szablonowego, sprawdzamy jaki argument aktualny odpowiada mu z wołanej procedury, prześledźmy to na naszym przykładzie:

```
proc(3; 4; 5; "Kim"; "Ola");
```

Tabela 2. Analiza parametrów formalnych szablonu nr 1

1	proc	<i>integer</i>	<i>T</i>	<i>T</i>	<i>integer</i>	<i>integer</i>
---	------	----------------	----------	----------	----------------	----------------

- *integer* nie jest typem szablonowym,
- *T* jest typem szablonowym i odpowiada mu *integer* (4) z wołanej procedury. Zatem *T* przechodzi na *integer*. Zapisujemy to w tablicy przejść dla danego szablonu,
- następny argument to również *T*, jednakże mamy go już zapisanego w tablicy, więc nic nie robimy,
- kolejne dwa argumenty nie są typu szablonowego.

Zatem w rezultacie dostaniemy:

Tabela 3. Tablica przejść dla szablonu nr 1

Typ szablonowy	Przechodzi na typ
<i>T</i>	<i>integer</i>

Po unifikacji nazwa naszej procedury będzie wyglądać następująco:

Tabela 4. Nazwa procedury możliwej do wygenerowania z szablonu nr 1

1	proc	<i>integer</i>	<i>integer</i>	<i>integer</i>	<i>integer</i>	<i>integer</i>
Czyli: proc\$args_integer_integer_integer_integer_string\$						

Porównujemy ją do nazwy wołanej procedury:

proc\$args_integer_integer_integer_string_string\$

Widzimy, że nie są sobie równe, zatem odrzucamy dany OID szablonu z listy szablonów pasujących do naszego wołania.

Wykonujemy kroki 4.1 – 4.4 dla kolejnego szablonu:

Tabela 5. Analiza parametrów formalnych szablonu nr 2

2	Proc	<i>R</i>	<i>integer</i>	<i>T</i>	<i>string</i>	<i>T</i>
---	------	----------	----------------	----------	---------------	----------

- *R* jest typem szablonowym, przechodzi w *integer*,
- *integer* nie jest typem szablonowym,
- *T* jest typem szablonowym, przechodzi w *integer*,
- *string* nie jest typem szablonowym,
- *T* już istnieje w tablicy przejść.

Zatem w rezultacie dostaniemy:

Tabela 6. Tablica przejść dla szablonu nr 2

Typ szablonowy	Przechodzi na typ:
<i>R</i>	<i>integer</i>
<i>T</i>	<i>integer</i>

Po unifikacji nazwa naszej procedury będzie wyglądać następująco:

Tabela 7. Nazwa procedury możliwej do wygenerowania z szablonu nr 2

2	Proc	<i>integer</i>	<i>integer</i>	<i>integer</i>	<i>string</i>	<i>integer</i>
Czyli: proc\$args_integer_integer_integer_string_integer\$						

Również nie pasuje do naszego wołania, jakim jest:

proc\$args_integer_integer_integer_string_string\$

Przechodzimy do kolejnego szablonu:

Tabela 8. Analiza parametrów formalnych szablonu nr 3

3	Proc	<i>T</i>	<i>T</i>	<i>T</i>	<i>string</i>	<i>string</i>
---	------	----------	----------	----------	---------------	---------------

- *T* jest typem szablonowym, przechodzi w *integer*, zapisujemy w tablicy przejść i pomijamy sprawdzanie kolejnych dwóch typów *T*,
- *string* nie jest typem szablonowym,

Zatem w rezultacie dostaniemy:

Tabela 9. Tablica przejść dla szablonu nr 3

Typ szablonowy	Przechodzi na typ:
<i>T</i>	<i>integer</i>

Po unifikacji nazwa naszej procedury będzie wyglądać następująco:

Tabela 10. Nazwa procedury możliwej do wygenerowania z szablonu nr 3

2	proc	<i>integer</i>	<i>integer</i>	<i>integer</i>	<i>string</i>	<i>string</i>
Czyli: proc\$args_integer_integer_integer_string_string\$						

Tym razem nazwa pasuje do naszego wołania:

proc\$args_integer_integer_integer_string_string\$

Szablon nr 4 odrzucamy z powodu nierównej liczby argumentów formalnych i aktualnych. Zatem, znaleźliśmy dokładnie jedną procedurę szablonową dla naszego wołania. W przypadku gdybyśmy znaleźli dwa lub więcej pasujących szablonów, wówczas skutkowałoby to podniesieniem błędu kompilacji z informacją o niemożliwości jednomyślnego wyboru szablonu.

Przykład kodu w systemie ODRA ilustrujący omawiane zagadnienie:

Dodatek A.10.2 (lub dołączony do pracy plik Przykłady/wybor_szablonu.sbnl)

5.3 Analiza argumentów aktualnych wołania

W poprzednim rozdziale omówiliśmy algorytm wyboru odpowiedniego szablonu na podstawie wołania. W naszym przykładzie analizowaliśmy wołanie:

```
proc(3; 4; 5; "Kim"; "Ola");
```

Użyte w nim argumenty aktualne są typami prymitywnymi, więc nie ma problemu z identyfikacją ich typów. W rzeczywistości w programach bardzo rzadko mamy do czynienia z przypadkiem, aby argumenty aktualne pojawiały się w postaci literałów.

Zazwyczaj argumenty są wyliczane dynamicznie. Najczęściej mamy do czynienia z wołaniami typu:

```
Proc(zm_x; zm_y; zm_z)
```

Dlatego musimy przeanalizować wszystkie argumenty aktualne w celu precyzyjnego ustalenia ich typów. W tym celu musimy dostać się do informacji o ich deklaracjach (pamiętajmy, że jesteśmy w fazie poprzedzającej linkowanie). Możemy sobie z tym poradzić, przekazując do analizatora deklaracje zmiennych lokalnych bądź globalnych, dostępnych w danym środowisku. Ciekawszy problem stanowi przypadek, w którym mamy do czynienia z zapytaniami, np.:

```
max_val(Student where nazw = "Chief"[1]; Student where nazw =  
"Holy"[1]);
```

Rozważmy to na przykładzie:

```
class StudentClass  
{  
    instance Student:  
    {  
        imie : string;  
        nazw : string;  
        nrIndexu : string;  
    }  
}  
Student : StudentClass [0..*];  
  
template(type T)  
max_val(a : T; b : T): T  
{  
    if(a.age >= b.age)  
        return a;  
    else return b;  
}
```

Dodatkowo zakładamy, że dodajemy do bazy dwa trwałe obiekty typu *Student*:

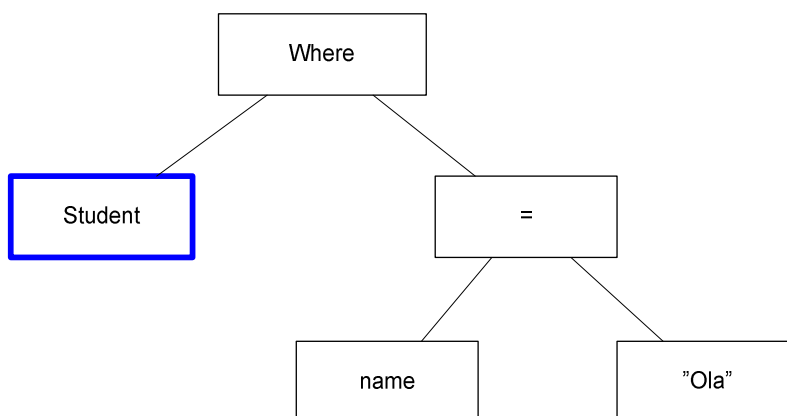
```
p1 : ref Student;  
p1 := create permanent Student  
(  
    "Kim" as imie,  
    "Chief" as nazw,  
    "5558" as nrIndexu  
);  
  
p2 : ref Student;  
p2 := create permanent Student  
(  
    "Ola" as imie,  
    "Holy" as nazw,  
    "6744" as nrIndexu  
);
```

Natrafiamy na wyżej wspomniane wywołanie procedury *max_val*:

```
max_val(Student where nazw = "Chief"[1]; Student where nazw =  
"Holy"[1]);
```

Czy możemy dowiedzieć się przed fazą linkowania, jaki będzie rezultat zapytań? Czyli innymi słowy, czy możemy bez „zaglądania do bazy” stwierdzić, że zwracana referencja prowadzi do klasy *StudentClass*, będąc w połowie kompilacji?

Odpowiedź brzmi: tak. Wiemy to, bez „zaglądania do bazy”. Wystarczy spojrzeć na zapytanie i już wiemy, jaki będzie rezultat:



Rysunek 5. Drzewo AST dla konstrukcji ‘where’

Oczywiście zwrot „wystarczy spojrzeć” dla kompilatora oznacza „wystarczy zbudować analizator drzewa AST” na nową okoliczność, który analizując wyrażenia będzie zwracał nam typ rezultatu. W naszym przypadku, wiedząc, że będzie to instancja *Student*, możemy zajrzeć do metabazy i sprawdzić, jakiego jest typu - na tym etapie kompilacji mamy do niej dostęp. Na zakończenie otrzymamy typ *StudentClass*.

Dla każdego wyrażenia, które może pojawić się w wywołaniu musimy w podobny sposób przebadac typ zwracanego wyniku. Dotyczy to także takich prostych wyrażeń jak ‘2+2’.

Przykład kodu w systemie ODRA ilustrujący omawiane zagadnienie:

Dodatek A.10.3 (lub dołączony do pracy plik Przykłady/Expression.sqql)

Przykład implementacji omawianego analizatora znajduję się w dołączonych do pracy rozszerzonych źródłach systemu ODRA:

Kod źródłowy/EGB/src/odra/sqql/precompiler/SBQLPreTypeInfo.java

5.4 Generowanie procedury - unifikacja szablonu

W momencie, gdy wiemy już, kiedy i którego szablonu użyjemy do produkcji naszej procedury i posiadamy tablice przejść dla typów szablonowych, przyszedł czas na generowanie procedury. Celowo użyłem słowa generowanie, aby przypomnieć, że nie używamy szablonu, aby do niego wstawić typy i używać go jak zwykłą procedurę. Używamy go, jako „wzór” i na jego podstawie stworzymy własną procedurę dostosowaną do odpowiednich typów z wywołania. Miejsca, w których typ szablonowy może wystąpić to:

- zwracany rezultat,
- lista argumentów,
- ciało procedury.

Zatem tworzymy nowy obiekt procedury i dokładamy do niego nowo stworzone elementy.

Rezultat procedury - Bierzemy z szablonu typ zwracanego rezultatu, sprawdzamy czy znajduje się w *tablicy przejść* jako klucz. Jeżeli tak, to oznacza to, że mamy do czynienia z typem szablonowym. Pobieramy przypadającą mu wartość z *tablicy przejść* i dokonujemy unifikacji typów. Tak powstały obiekt rezultatu wstawiamy do naszej procedury.

Lista argumentów - Pobieramy listę argumentów formalnych z szablonu. Dla każdego elementu listy robimy dokładnie to samo, jak w powyższym przypadku. Powstałą listę argumentów wstawiamy do naszej procedury.

Ciało procedury – Ciało procedury zapisane jest w postaci drzewa AST w szablonie. Typ szablonowy może występować wszędzie tam, gdzie normalnie możemy używać innych typów, np. w deklaracji zmiennych czy w operacji rzutowania. Aby dokonać unifikacji, musimy przeanalizować węzły drzewa AST, w przypadku natrafienia na typ szablonowy podmieniamy go zgodnie z informacją zawartą w *tablicy przejść*.

Szczególnym przypadkiem jest wołanie procedury szablonowej w ciele innego szablonu. Oczywiście zgodnie z zasadą ortogonalności, nie możemy zabronić takiej sytuacji, ale aby taką sytuację obsłużyć, musimy zaraz po unifikacji typów przeanalizować ciało na wypadek wystąpienia wołania do procedury szablonowej. Jeżeli takie napotkamy, to tworzymy nową procedurę, wg tego samego schematu. Zatem rekurencyjny algorytm postępowania możemy zapisać następująco:

Krok 1.

Analiza kodu SBQL na wypadek odwołania się do szablonu.

Krok 2.

W przypadku nie znalezienia wołania kończymy postępowanie.
W przypadku odnalezienia wołania dokonujemy unifikacji typów zwracanego rezultatu, argumentów formalnych oraz ciała szablonu.

Krok 3.

Dodatkowo dla zunifikowanego ciała szablonu powtarzamy algorytm zaczynając od kroku 1.

Krok 4.

Generujemy procedurę.

Przykład: Załóżmy, że kod SBQL składa się z dwóch szablonów (*Proc1*, *Proc2*). W ciele jednego (*Proc2*) znajduje się wołanie do drugiego szablonu (*Proc1*). Mamy jeszcze jedną procedurę (*init*), która dzięki temu, że w jej ciele występuje odwołanie do procedury szablonowej, zainicjalizuje cały proces.

```

template (type T)
Proc1(a : T; b : T)
{
    . . . // krok 3.2, 3.2.1, 3.4
}

template (type T)
Proc2(a : T; b : T)
{
    Proc1(a; b); // krok 2, 3.1, 4
}

init()
{
    Proc2(100; 200); // krok 1
}

```

Krok 1.

Analizator trafia na wywołanie *Proc2* w ciele procedury *init*, odnoszące się do szablonu.

Krok 2.

Dokonujemy unifikacji szablonu *Proc2*.

Krok 3.

Rozpoczynamy algorytm od kroku 1 dla ciała procedury *Proc2*.

Krok 3.1

Natrafiamy na wołanie do szablonu *Proc1*.

Krok 3.2

Dokonujemy unifikacji szablonu *Proc1*.

Krok 3.3

Rozpoczynamy algorytm od kroku 1 dla ciała procedury *Proc1*.

Krok 3.3.1

Analizator nie natrafia na odwołanie do szablonu, zatem nie następuje dalsza rekurencja.

Krok 3.4

Generujemy procedurę:

```
Proc1(a : integer; b : integer)
{ ... }
```

Krok 4.

Generujemy procedurę:

```
Proc2(a : integer; b : integer)
{ Proc1(a; b); }
```

W ten sposób wygenerowaliśmy dwie procedury szablonowe, awołanie *Proc2* nie zakończyło się błędem. Musimy jeszcze rozważyć sytuację, w której w ciele szablonu znajduje się rekurencyjne wywołanie do samej siebie, np.

```
template (type T)
Proc1(a : T; b : T)
{
    if(. . . )
        Proc1(a; b);
}

Procedura()
{ Proc1(100; 200); }
```

W tym przypadku efektem wykonywania naszego algorytmu byłoby „zapętlenie się” mechanizmu generacji na etapie analizy ciała, unifikacji i powtórnej analizy ciała, itd. W celu uniknięcia tego należy do analizy sprawdzającej ciało przekazać informację, że: właśnie jesteśmy w trakcie tworzenia procedury o danej nazwie (w naszym przypadku *Proc1\$_args\$integer_integer\$*), powstałej z danego szablonu. Jeżeli zamierzamy generować taką samą procedurę, to nie pozwalamy algorytmowi dalej postępować.

Przykład kodu w systemie ODRA, ilustrujący omawiane zagadnienie:

- unifikacja szablonu (wołanie procedury szablonowej z szablonu):

Dodatek A.10.4

(lub dołączony do pracy plik Przykłady/generowanie_proc.sq1)

- wywołania rekurencyjne z ciała szablonu:

Dodatek A.10.5

(lub dołączony do pracy plik Przykłady/rekurencja.sqql)

Przykład implementacji omawianego analizatora odpowiedzialnego za unifikację znajduje się w dołączonych do pracy rozszerzonych źródłach systemu ODRA:

Kod źródłowy/Odra/EGB/src/Odra/sqql/pretypechecker/SBQLTemplateUnification.java

5.5 Szablony w tradycyjnych językach programowania

Zaproponowane rozwiązanie na pewno warto porównać z kolejną implementacją tego samego rozszerzenia a dokładnie z podejściem stosowanym w tradycyjnych językach programowania np. w języku C++, czyli operacjach na łańcuchach tekstowych. Wiąże się to z przeniesieniem całego mechanizmu prekompilacji przed fazę parsowania, łącznie z mechanizmem przeciążeń nazw funkcji. Jednakże, wcale nie jest powiedziane, że to podejście będzie prostsze w implementacji. Trudność będzie polegać na ustaleniu typów podczas wołania. Przypadek, gdzie mamy:

```
Max(1; 3);
```

jest oczywiście trywialny, parser bez trudu rozpoznaje typy *integer*, ale już przypadek:

```
Max(zmienna_x; zmienna_y);
```

wymaga od nas znalezienia deklaracji zmiennych i ustalenia typów, czyli zdefiniowania „przeszukiwaczy”, które w danym zakresie odnajdą deklaracje obu zmiennych, a wszystko to przez operacje na łańcuchach tekstowych. Jeszcze większy problem mamy w przypadku wyrażenia występującego w formie zapytania (oczywiście niedostępne w C++), którego rezultatem jest sama referencja do obiektu znajdującego się w bazie, np.:

```
Max(Emp where name="Chief"; Emp where name="Holy");
```

Załóżmy, że wołanie znajduje się w module:

```
class StudentClass
{
    instance Student:
    {
        imie : string;
        nazw : string;
        nrIndexu : string;
    }
}
Student : StudentClass [0..*];

template (type T)
max_val(a : T; b : T): T
{
    if(a.age >= b.age)
        return a;
    else return b;
}
```

Ustalenie typów na etapie „przed parsowaniem” będzie uciążliwe. Patrząc na sam tekst wywołania (nie drzewo AST!), nie jesteśmy w stanie powiedzieć, jaki typ nam zwraca. A nawet gdyby było to możliwe, to jest to słowo *Student* a nie *StudentClass*. To tylko pokazuje, że bez parsowania kodu źródłowego do obiektu drzewa AST nie jesteśmy w stanie zrobić prostej (już tylko z nazwy) operacji prekompilacji. A nawet jak utworzymy drzewo AST, to nadal potrzebujemy mechanizmu na wzór metabazy, w której oprócz informacji o znajdujących się obiektach w module, potrzebujemy jeszcze informacji o ich zakresach, czyli w jakich sekcjach programu są dostępne, a w jakich nie. To tylko utwierdza nas w słuszności wyboru miejsca analizy wywołań szablonowych, prezentowanego w niniejszej pracy.

Jednocześnie, wnioski te pokazują nam istotę głównego problemu, dotyczącego mechanizmu szablonów w systemie ODRA, jakim jest miejsce wyboru generacji nowych procedur. Gdyby nie omawiany w pracy problem z linkowaniem, generacja procedur szablonowych odbywałaby się w fazie wewnętrznej kontroli typów. Pierwsza próba implementacji przeprowadzona była właśnie w niej,

co skutkowało szybkim i bezproblemowym dostępem do wszystkich informacji odnośnie typów. Jest to najlepsze miejsce do ustalania konkretnych typów z wołania, czyli de facto, ustalania nazwy, jakie będą przyjmować nowe procedury. Niewątpliwie oparcie mechanizmu linkowania o sposób dynamiczny umożliwiłoby to podejście.

6. SZABLONY – UZUPEŁNIENIE

I ROZSZERZENIE OPISU

W kolejnych podrozdziałach szerzej omówimy szczególne przypadki lub niejasności związane z używaniem procedur szablonowych. Omówimy również możliwości dalszego rozszerzenia i porównamy z podobnymi mechanizmami w tradycyjnych językach programowania.

6.1 *Procedury specjalizowane*

Szablony, mimo swojej elastyczności, nie zawsze do końca umożliwiają nam zrealizowanie wszystkich aspektów. Niestety taka jest ich natura, podobnie jest w życiu, gdy nie zawsze jesteśmy zadowoleni z „szablonowych rozwiązań”. Jeżeli chcemy mieć coś oryginalnego, to musimy to zamówić „na miarę”, z czym na co dzień się spotykamy w branży IT. Możemy powiedzieć, że jedno i drugie zjawisko istnieją obok siebie zgodnie, bo przecież godzimy się na kilka wersji Windows-a dystrybuowanego na cały świat, ale dopuszczamy również możliwość wyspecjalizowania produktu, czego przykładem jest opracowanie przez firmę Microsoft specjalnej, bezpiecznej wersji Windows-a XP dla amerykańskich sił powietrznych¹.

Podobnie rzecz się ma z szablonami w językach programowania. Zdarza się, że chcielibyśmy mieć szablon do produkcji zestawu funkcji (szablonowych), jednakże w trakcie pisania widzimy, że zupełnie nie będzie się on nadawał do pewnego typu zmiennej. Co możemy zrobić? Otóż musimy w pewien sposób zakomunikować kompilatorowi, że ma oto szablon do produkcji funkcji szablonowej, ale jeśli chodzi o jakiś konkretny typ, to nie chcielibyśmy tej szablonowej wersji funkcji, ale raczej specjalną – taką, jaką zaraz mu zdefiniujemy. Funkcję taką będziemy nazywać **funkcją specjalizowaną**.

¹ Dzięki wdrożeniu nowego, bezpiecznego XP, US Air Force zaoszczędziła, między innymi, wiele godzin serwisowania oraz 100 milionów dolarów dzięki konsolidacji 30 innych kontraktów.

W praktyce oznacza to, że zamieszczamy w tym samym zakresie (np. w module) definicję szablonu i funkcji specjalizowanej. Oczywiście nazwa funkcji specjalizowanej jest taka sama jak nazwa funkcji szablonej.

Przykład:

```
template (type T)
max_val(a : T; b : T): T
{
    if(a >= b)
        return a;
    else
        return b;
}

max_val(a : Osoba; b : Osoba): Osoba
{
    if (a.wiek() >= b.wiek())
        return a;
    else
        return b;
}
```

Kompilator natrafiając na wywołanie:

```
max_val(student; emeryt);
```

z argumentami aktualnymi typu *Osoba*, najpierw poszuka czy funkcja już nie istnieje – sprawdzi czy nie zrealizowaliśmy jej jako funkcji specjalizowanej. Jeżeli znajdzie definicję funkcji specjalizowanej, to owo wywołanie potraktuje jako wywołanie tejże funkcji specjalizowanej. Jeżeli nie znajdzie takiej definicji, a ma zdefiniowany tylko szablon, to za pomocą tego szablonu wyprodukuje funkcję. Będzie to znaczyć, że na okoliczność takiego wywołania funkcji nie przewidujemy wersji specjalizowanej – wystarczy nam powstała z szablonu. Taka technika jest niewątpliwie uelastycznieniem mechanizmu szablonów.

Przykład kodu w systemie ODRA ilustrujący omawiane zagadnienie:

Dodatek A.10.6 (lub dołączony do pracy plik Przykłady/specjalizacja.sbnl)

6.2 Dwa szablony, jeden szczególnym przypadkiem drugiego

Założmy, że w kodzie SBQL zdefiniowaliśmy dwa szablony zdolne do generowania funkcji o nazwie *suma* wywoływanej z dwoma argumentami. Funkcja pierwsza przyjmuje parametry identycznego typu, a funkcja druga parametry dwóch różnych typów:

```
template (type T)
suma(a: T; b: T): T
{ instrukcje }

template (type T, type R)
suma (a: T; b: R): T
{ instrukcje }
```

Rozważmy, czy dwa takie szablony mogą pojawić się w tym samym module programu? Nie narusza to żadnych zasad konstrukcyjnych. Problem polega na tym, że szablon o dwóch parametrach w pewnych przypadkach nadaje się równie dobrze do wyprodukowania funkcji jak pierwszy szablon. Przypadek ten zachodzi wtedy, gdy dwa parametry T , R oznaczają ten sam typ. Wtedy szablon z dwoma „różnymi” parametrami jest jakby szczególnym przypadkiem tego z dwoma takimi samymi parametrami. Zatem mimo tego, że szablony takie mogą istnieć obok siebie, kompilator natrafiający na wywołanie funkcji:

```
suma(1; 2)
```

będzie w konfuzji, nie wiedząc, którego z szablonów należy użyć. Oba są jednakowo dobre, bo jak już wcześniej ustaliliśmy, typ zwracanego rezultatu nie wpływa na nazwę funkcji. Zatem, nie pozostaje nam nic innego, jak podnieść błąd kompilacji z komunikatem o dwuznaczności co do wyboru szablonu. Oczywiście kompilator mógłby użyć pierwszego znalezionej szablonu, lecz wybór taki może oznaczać „zgadywanek”, a niczego nieświadomy programista może nawet nie zorientować się, że wywołuje zupełnie inną funkcję, niż zamierzał.

Tak więc, tworząc w tym samym module dwa szablony o tej samej nazwie, należy zwrócić uwagę, aby jeden nie był szczególnym przypadkiem drugiego.

Przykład kodu w systemie ODRA, ilustrujący omawiane zagadnienie:
Dodatek A.10.7 (lub dołączony do pracy plik Przykłady/ambigu_call..sbql)

6.3 Parametr ustalający typ rezultatu funkcji

Rozważmy następujący przykład. Załóżmy, że chcemy utworzyć szablon do generowania procedur:

```
procedura(a1 : typ_1; a2 : typ_2): typ_3
```

Zauważmy, że zwracany rezultat *typ_3* nie występuje na liście parametrów formalnych, a więc jest to zupełnie inny typ, który jednak byśmy chcieli dynamicznie generować. Nie chodzi nam o procedurę, której zwracany rezultat jest określony na „sztywno”, np.:

```
procedura(a1 : typ_1; a2 : typ_2): boolean
```

To oczywiście jesteśmy w stanie zrobić, ale nas interesuje, aby typ rezultatu dało się niezależnie określić podczas wywołania procedury szablonowej. Jak napisać taki szablon?

Jak pamiętamy, parametrami szablonu mogą być tylko te typy, które występują jako typy argumentów formalnych danego szablonu. Dlatego parametrem szablonu funkcji nie możemy uczynić typu rezultatu, jeżeli ten typ jest odmienny od wszystkich ewentualnych argumentów funkcji. Innymi słowy nie możemy mieć takiego szablonu:

```
template (type szablonowy1, type szablonowy2, type szablonowy3)  
procedura (a : szablonowy1; b : szablonowy2): szablonowy3  
{ ... }
```

Dlaczego nie? Powód jest następujący, gdybyśmy napisali takie wywołanie funkcji:

```
procedura(2; 3.3);
```

czyli wywołanie, w którym nie dbamy o to, co się stanie z rezultatem, wówczas co kompilator miałby przyjąć za typ rezultatu; i jaką funkcję szablonową dla nas wygenerować?

Zasada, która tu obowiązuje, jest następująca. Przy dopasowywaniu wywołań do odpowiednich szablonów, nie jest brany pod uwagę typ rezultatu funkcji, zatem parametr szablonu nie może posłużyć do określenia jedynie samego rezultatu

funkcji. W przeciwnym wypadku moglibyśmy produkować funkcje różniące się tylko typem zwracanym, a to jest sprzeczne z zasadą przeciążeń nazw funkcji, o której pisaliśmy wcześniej.

Czy zatem nie da się nic zrobić, gdy potrzebujemy takiego szablonu? Możemy nasz żądany typ rezultatu umieścić jako dodatkowy argument formalny naszego szablonu:

```
template (type szablonowy1, type szablonowy2, type szablonowy3)
procedura (a : szablonowy1; b : szablonowy2; c : szablonowy3):
szablonowy3
{ ... }
```

Wówczas wywołanie:

```
procedura(2; 3.3; "typu string");
```

spowoduje wygenerowanie procedury:

```
procedura ( a : integer; b : real; c : string): string
{ ... }
```

Zatem dokładnie to, o co nam chodziło – możemy dynamicznie generować typ rezultatu szablonu. Jednakże musimy pamiętać, że jest to sztuczka, która wprowadzając niepotrzebny w ciele funkcji parametr, mija się trochę ze sztuką dobrego programowania, zatem nie powinna być zalecana.

6.4 Typy wbudowane a typy zdefiniowane

Uniwersalność szablonów polega na tym, że mogą służyć do produkcji procedur przyjmujących argumenty dowolnego typu. Pamiętajmy jednak, że nie każdy szablon można sprowadzić do dowolnego typu. Jeżeli budujemy szablon funkcji służący np. do rejestracji osób na wybory, to oczekujemy, że argumentem funkcji będzie obiekt typu *Osoba*, gdyż spodziewamy się użyć takich metod składowych, jak *wiek()*, czy też dostępu do pól takich, jak *narodowość*, czy *nr Dowodu Osobistego*. Zatem, nie jest możliwe wykorzystanie tego szablonu dla prymitywnego typu *integer*, gdyż rzeczą oczywistą jest, że nie posiada on wyżej wymienionych właściwości. Procedura powstała dla parametru *integer* nie przejdzie pomyślnie etapu kontroli typów.

Tym samym, dochodzimy do stwierdzenia, że nie każdy szablon funkcji nadaje się do argumentów dowolnego typu. Najczęściej tworząc go, sami dokładnie wiemy, do czego chcemy go używać. Na przykład, powinien nadawać się dla obiektów typu *Student*. Inne typy argumentów nas nie interesują. Wtedy testowanie szablonu można ograniczyć tylko do interesujących nas typów – czyli w naszym przypadku do samych obiektów klasy *Student*, ale skoro tylko do jednego typu, to czy teraz nie mówimy już o normalnej funkcji – nieszablonowej?

Twórcy języka Java poradzili sobie z tym problemem. Programista w trakcie tworzenia szablonu może określić warunek minimum dziedziczenia dla parametru *T*.
Przykład:

```
public static <T extends Osoba> T zapiszNaWybory(T
obywatel)
{
    if (obywatel.wiek() > 18
        && obywatel.narodowosc.equals("PL"))
        Rejestruj(obywatel.nrDowoduOsobistego);
    return obywatel;
}
```

Założmy istnienie klas:

```
public abstract class Osoba{ ... }
public class Prezydent extends Osoba{ ... }
public class Student extends Osoba{ ... }
```

Teraz nie grozi nam błąd z powodu wspomnianego problemu. Dostęp do odpowiednich metod i pól składowych zostaje zagwarantowany przez dziedziczenie, które zadeklarowaliśmy przy parametrze *T*. W ten sposób mamy zapewnione, że argumenty aktualne (typu *Student*, czy *Prezydent*) mają dostęp do właściwości i metod składowych, odziedziczonych (z klasy *Osoba*). Co by się stało, gdybyśmy chcieli skorzystać z metody składowej *podajNumerIndeksu()*, dostępnej w klasie *Student*, ale niedostępnej w klasie *Osoba*? Dostalibyśmy błąd kompilacji, gdyż możemy posługiwać się tylko właściwościami z klasy *Osoba*. Zastanówmy się, czy prostszy zapis nie daje nam tego samego efektu. Zamiast szablonu użyjmy procedury z parametrem abstrakcyjnej klasy *Osoba*:

```

public static Osoba zapiszNaWybory(Osoba obywatel)
{
    if (obywatel.wiek() > 18
        && obywatel.narodowosc.equals("PL"))
        Rejestruj(obywatel.nrDowoduOsobistego);
    return obywatel;
}

```

Funkcja ta, wykonuje dokładnie to samo i również jest bezpieczna, gdyż do wywołania tej samej funkcji nieszablonowej, możemy używać obiektów typu *Prezydent* lub *Student*. Jaka zatem jest różnica, oprócz czytelności kodu, która na pierwszy rzut oka jest na korzyść dla funkcji opartej o klasę abstrakcyjną? Przewaga tkwi w wydajności rozwiązania szablonowego. Przeanalizujemy poniższy kod:

```

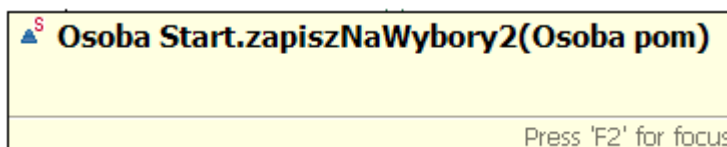
Student stud = new Student("Kamil");
Prezydent prez = new Prezydent("Roosevelt");

//Wywołanie funkcji, gdzie argumentem jest obiekt klasy
//abstrakcyjnej:

    zapiszNaWybory2(stud); //func. oparta o klasę abstr
    zapiszNaWybory2(prez); //func. oparta o klasę abstr

```

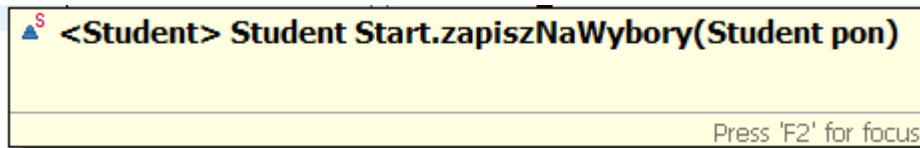
Kompilator w obu przypadkach podpowie nam, że mamy do czynienia z tą samą funkcją, przyjmującą argument abstrakcyjnej klasy, jak i zwracającą go:



Rysunek 6. Informacja o funkcji wygenerowana przez Eclipse IDE

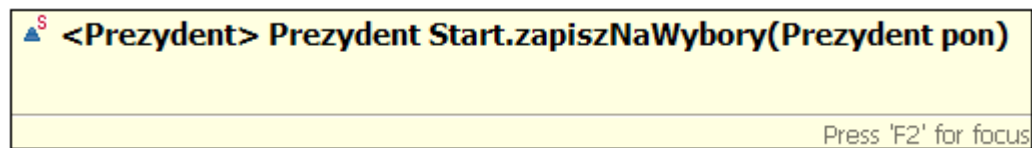
Wszystko odbywa się tak, jak to zdefiniowaliśmy. Natomiast wywołanie funkcji opartej o szablon, doprowadzi do **stworzenia** dwóch funkcji: jednej z parametrem typu *Student*, drugiej z parametrem typu *Prezydent*. Zwracany rezultat obu tych funkcji będzie już konkretnego typu, zatem zaoszczędziliśmy na operacji rzutowania. Czytelność kodu (a raczej pracy z nim) również ulega poprawie, bowiem w miejscu wołania dostajemy jasne informacje, z jaką funkcją pracujemy:

```
zapiszNaWybory(stud); // funkcja wygenerowana z szablonu
```



Rysunek 7. Informacja o funkcji wygenerowana przez Eclipse IDE

```
zapiszNaWybory(prez); // funkcja wygenerowana z szablonu
```



Rysunek 8. Informacja o funkcji wygenerowana przez Eclipse IDE

Podsumowując stwierdzamy, że wprowadzenie dziedziczenia do typów szablonowych, mimo poprawy bezpieczeństwa, mocno ograniczyło uniwersalność samych funkcji szablonowych. Z drugiej jednak strony, szablon nie może być uniwersalny, musi być dobry i o tym przede wszystkim należy pamiętać. W niniejszej pracy, w części implementacyjnej, autor nie zdecydował się wprowadzać wyżej wymienionej właściwości dla typu szablonowego.

6.5 Mechanizm szablonów dla klas

Warto zauważyć, że mając mechanizm do produkcji procedur szablonowych naturalnym kierunkiem jest dążenie do zrealizowania **klas szablonowych**. Zasada działania jest bardzo podobna jak w przypadku procedur. Skoro mamy napisać parę podobnych klas, różniących się tylko typem danej składowej, to zamiast tego, możemy pokazać kompilatorowi jak powinien to wykonać za nas, czyli wskazać mu szablon, według którego to zrobi.

Przykład:

```
class schowekClass
{
    instance schowek :
    {
        Sejf : integer;
```

```

    }
    Schowaj(x : integer) {Sejf := x; }
    Oddaj():integer { return Sejf; }
}

```

Jak widać, klasa ma jedną daną składową na przechowanie liczby typu *integer* i dwie metody składowe do jej obsługi. Gdybyśmy chcieli zdefiniować podobną klasę do przechowywania łańcucha znakowego, to musielibyśmy zdefiniować:

```

class schowek_na_stringClass
{
    instance schowek_na_string :
    {
        Sejf : string;
    }
    Schowaj(x : string) {Sejf := x; }
    Oddaj():string { return Sejf; }
}

```

Widzimy, że zmieniliśmy typy i nazwę klasy, ale poza tym wszystko wygląda bardzo podobnie. Możemy więc przypuszczać, że dało by się ten proces zautomatyzować. Zobaczmy zatem, jakby teoretycznie wyglądała klasa szablonowa dla tego przypadku:

```

template(type T)
class schowekClass
{
    instance schowek :
    {
        Sejf : T;
    }
    Schowaj(x : T) {Sejf := x; }
    Oddaj():T { return Sejf; }
}

```

Podobnie jak wcześniej, definicje rozpoczyna słowo *template*, po którym deklarujemy, jak się będą nazywały nasze typy szablonowe, których użyjemy wewnątrz klasy. Tym samym „nauczylismy” kompilator, jak ma produkować nowe klasy w chwili, kiedy będą nam potrzebne. Jeżeli gdzieś w programie zapagniemy mieć schowki przechowujące liczbę *integer* lub *string*, to napiszemy instrukcję:

```
schowekNaInteger : SchowekClass<integer> [0..*];  
schowekNaReal : SchowekClass<real> [0..*];
```

Kompilator dokona unifikacji typów i wygeneruje dwie całkiem nowe klasy. Jeżeli chodzi o samo wołanie, to w porównaniu do procedur, widzimy pewne różnice. O ile podczas wołania procedury szablonowej

```
Max(3; 4);
```

programista używa tej samej składni, co w przypadku zwykłej procedury (nie musi wiedzieć o istnieniu szablonu, kompilator sam dopasowywał odpowiednie typy na podstawie wołania), o tyle podczas tworzenia klas szablonowych trzeba nieco zmodyfikować składnię, definiując *explicite* w co ma przejść typ szablonowy.

Również, podobnie jak w przypadku procedur, potrzebujemy rozbudować mechanizm generowania nazwy klasy na podstawie typu. W naszym przypadku będzie to np. *SchowekClass\$integer*, *SchowekClass\$real*.

Oczywiście szablony klas wykraczają poza temat niniejszej pracy, ale niewątpliwie jest to naturalne uzupełnienie mechanizmów z rodziny *template*. W klasycznych językach programowania bardzo dobrze sprawdzają się w „klasach pojemnikach”, czyli abstrakcyjnych strukturach danych, takich jak lista, kolejka, stos, czy drzewo. Dzięki temu, że określamy typ na którym pracujemy, wystarczy nam jeden szablon do pracy z nieznanymi nam jeszcze typami, a także osiągamy lepszą wydajność, niż gdybyśmy chcieli zaprojektować pojemnik oparty na składowej będącej klasą abstrakcyjną.

7. PODSUMOWANIE

Praca udowadnia, że osiągnięcie celu, jakim jest implementacja mechanizmów szablonu w systemie ODRA, jest jak najbardziej możliwa do zrealizowania, a sam język SBQL został w naturalny sposób rozszerzony o nowe właściwości. Mechanizmy te są dobrze znane programistom tradycyjnych języków programowania, takich jak C++, JAVA, C#, dlatego wprowadzenie ich do systemu ODRA nie jest wydarzeniem rewolucyjnym, lecz raczej oczekiwanym i oczywistym w rozwoju systemu.

Dzięki wprowadzeniu szablonów zaistniała również potrzeba wprowadzenia mechanizmu przeciążeń nazw funkcji, co niezależnie od głównego tematu pracy podniosło możliwości języka, a tym samym systemu ODRA. Niewątpliwie jest to często używany mechanizm przez programistów, również obecny w szerokim gronie współczesnych języków programowania.

Oprócz podstawowego modelu wykorzystania szablonów, autor niejednokrotnie porusza, mniej lub bardziej kłopotliwe przypadki użycia, jak: przeciążenia szablonów, „szablon będący szczególnym przypadkiem innego”, rekurencja, procedury specjalizowane i inne, oraz rozstrzyga, jak powinien zachować się kompilator i jaki powinien być wynik. Znajduje to potwierdzenie w licznych przykładach kodu SBQL (gotowego do użycia) dołączonego do pracy.

Mimo prostego założenia szablonów, od strony technicznej wymagało to budowy dość rozbudowanego mechanizmu (4 analizatory drzew AST, w tym 3 zupełnie nowe, które wywołują się rekurencyjnie), który dokonuje zmian w procesie kompilacji. Właśnie za sprawą rozbudowanej implementacji może się okazać, że mimo dołożonych wszelkich starań w procesie testowaniu i poprawiania napotkanych błędów, pewne procesy uległy niekorzystnym zmianom. Głównie wiąże się to z tym, że został zmieniony sposób nadawania nazw procedur, jak i ich wywoływania, czyli w obu przypadkach konieczność generacji pełnej nazwy na podstawie wołania. Same procedury są rozpięte po całym systemie, wchodząc w skład wielu konstrukcji, o których autor mógł nawet nie wiedzieć lub niedostatecznie je przetestować. Co więcej, w przypadku dalszych rozszerzeń, jest to miejsce, o którym należy pamiętać. Na dzień dzisiejszy nie odnotowano żadnych negatywnych skutków ubocznych danej implementacji.

Niemniej jednak autor wierzy, że obrany kierunek w pracy magisterskiej jest kierunkiem słusznym i wyraża zadowolenie, że prace implementacyjne zostały sfinalizowane, a całość można prezentować w formie praktycznej, a nie tylko opisowej.

Dalszym kierunkiem rozwoju mechanizmów z rodziny *template* jest stworzenie w analogiczny sposób klas szablonowych, które reprezentują zestaw (rodzinę) klas, mogących współpracować z różnymi typami danych. Klasy takie będą wyposażone w metody szablonowe i dane składowe typu szablonowego. Obie te rzeczy są już zrealizowane. Pozostaje jedynie rozbudować mechanizm nazywania klas na podstawie typu oraz zaprojektować proces generacji nowej klasy na podstawie wzorca. Z bardziej praktycznego punktu widzenia, szablony klas są znacznie przydatniejsze i częściej stosowane niż szablony funkcji. Typowym ich zastosowaniem są klasy pojemnikowe, czyli znane i lubiane przez programistów struktury danych - a one obok algorytmów, są według klasyków informatyki podstawowymi składnikami programówⁱ.

ⁱ Autor nawiązuje do równania: Algorytmy + struktury danych = programy, będącego jednocześnie tytułem słynnej książki Niklausa Wirtha.

8. BIBLIOGRAFIA

1. K. Subieta “*Teoria i konstrukcja obiektowych języków zapytań*”, wydawnictwo PJWSTK, Warszawa 2004
2. M. Lentner Rozprawa doktorska „*Integracja danych i aplikacji przy użyciu wirtualnych repozytoriów*”, PJWSTK, Warszawa 2008.
3. J. Grębosz – „*Symfonia C++*” wydanie 5, wydawnictwo Edition 2000, Kraków.
4. J. Grębosz – „*Pasja C++*” wydanie 3, wydawnictwo Edition 2000, Kraków.
5. M. Trzaska - ODRA-IDE, strona WWW: „Pliki : PJWSTK - Mariusz Trzaska”, <http://www.pjwstk.edu.pl/~mtrzaska/odra-ide>
6. strona WWW: „ODRA manual”,
http://www.sbql.pl/various/ODRA/ODRA_manual.html
7. strona WWW: „Kazimierz Subieta: Słownik terminów z zakresu obiektowości”, http://www.ipipan.waw.pl/~subieta/artykuly/slownik_obiektowosci/hasla_slownika.html
8. strona WWW: “Template (programming) - Wikipedia, the free encyclopedia”, (wersja z dnia 10-II-2009)
[http://en.wikipedia.org/wiki/Template_\(programming\)](http://en.wikipedia.org/wiki/Template_(programming))
9. strona WWW: “SBA and SBQL description”,
<http://www.sbql.pl/>
10. strona WWW: „Generic programming - Wikipedia, the free encyclopedia”, (wersja z dnia 11-II-2009)
http://en.wikipedia.org/wiki/Generic_programming
11. strona WWW: „Generics in C#, Java, and C++”
<http://www.artima.com/intv/generics2.html>
12. strona WWW: „Język programowania – Wikipedia, wolna encyklopedia” (wersja z dnia 28-I-2009)
http://pl.wikipedia.org/wiki/Język_programowania

9. SPIS RYSUNKÓW I TABEL

Spis rysunków:

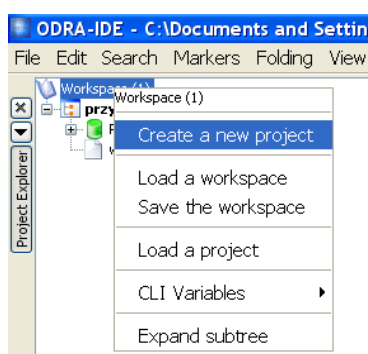
Rysunek 1. Stan stosu rezultatów dla wyrażenia arytmetycznego	15
Rysunek 2. Kolejność przeszukiwania ENVŚ podczas wiązania nazwy	17
Rysunek 3. Typ szablonowy jako prymityw w systemie ODRA	19
Rysunek 4. Etapy budowania modułów	25
Rysunek 5. Drzewo AST dla konstrukcji ‘where’	34
Rysunek 6. Informacja o funkcji wygenerowana przez Eclipse IDE	48
Rysunek 7. Informacja o funkcji wygenerowana przez Eclipse IDE	49
Rysunek 8. Informacja o funkcji wygenerowana przez Eclipse IDE	49
Rysunek 9. Tworzenie nowego projektu w ODRA-IDE	56
Rysunek 10. Połączenie z serwerem.....	56
Rysunek 11. Tworzenie nowego pliku	56
Rysunek 12. Kompilacja projektu	57
Rysunek 13. Inicjalizacja modułu.....	57
Rysunek 14. Zintegrowane środowisko programistyczne Odra-IDE	66
Rysunek 15. Przykład zmian odczytanych za pomocą TortoiseSVN.....	68

Spis tabel:

Tabela 1. Próba dopasowania wołania do wariantów istniejących szablonów.....	29
Tabela 2. Analiza parametrów formalnych szablonu nr 1	30
Tabela 3. Tablica przejść dla szablonu nr 1	30
Tabela 4. Nazwa procedury możliwej do wygenerowania z szablonu nr 1	31
Tabela 5. Analiza parametrów formalnych szablonu nr 2	31
Tabela 6. Tablica przejść dla szablonu nr 2	31
Tabela 7. Nazwa procedury możliwej do wygenerowania z szablonu nr 2.....	31
Tabela 8. Analiza parametrów formalnych szablonu nr 3	32
Tabela 9. Tablica przejść dla szablonu nr 3	32
Tabela 10. Nazwa procedury możliwej do wygenerowania z szablonu nr 3.....	32

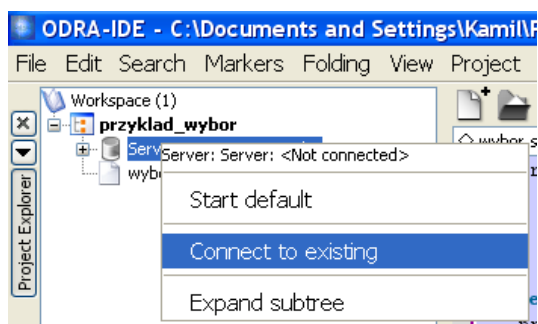
10. Dodatek A - Przykłady kodu SBQL

W dodatku tym zamieszczone są wszystkie przykłady do których odnosiliśmy się w pracy. Wszystkie prezentowane przykłady szablonów można uruchomić w systemie ODRA. Aby tego dokonać, należy uruchomić zintegrowane środowisko wraz z serwerem ODRA (dokładny opis tej czynności znajduje się w Dodatku B), wybrać opcję „Create a new project” z menu kontekstowego pozycji „Workspace” i podać nazwę projektu.



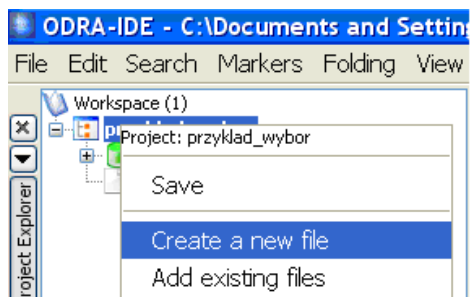
Rysunek 9. Tworzenie nowego projektu w ODRA-IDE

Łączymy się z serwerem poprzez „Connect to existing”



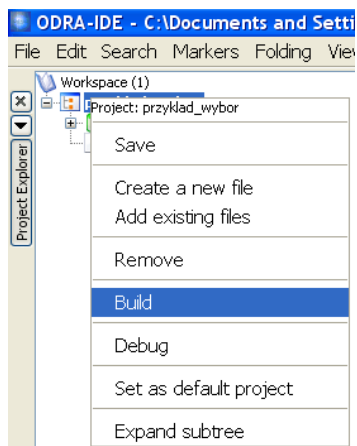
Rysunek 10. Połączenie z serwerem

Z menu kontekstowego projektu wybieramy „Create a new file” aby utworzyć nowy plik w projekcie w którym zapiszemy ciało modułu.



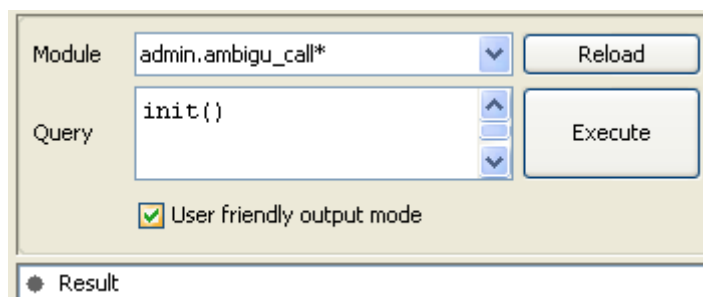
Rysunek 11. Tworzenie nowego pliku

W nowo powstałym pliku umieszczamy treść przykładu i uruchamiamy kompilację za pomocą opcji „Build”.



Rysunek 12. Kompilacja projektu

Inicjalizujemy działanie modułu za pomocą wywołania (przycisk „Execute”) odpowiednich procedur (zwykle w nazwie zawiera słowo init).



Rysunek 13. Inicjalizacja modułu

Wszystkie przykłady znajdują się również na płycie CD dołączonej do pracy (folder Przykłady).

10.1 Deklaracja szablonu

```
module def
{
  template (type T, type R)
  procedura_szablonowa(a : T): T
  { return a; }
```

```

template (type T, type R)
procedura_szablonowa(a : T; b : T): T
{ return b; }

template (type T, type R)
procedura_szablonowa(a : T; b : T; c : T): T
{ return c; }

init1(): integer
{ return procedura_szablonowa(3);}

init2(): integer
{ return procedura_szablonowa(4; 4); }

init3(): integer
{ return procedura_szablonowa(5; 5; 5); }

}

```

10.2 Wybór odpowiedniego szablonu

```

module wybor_template
{
    template (type T) // $TEMPLATE$proc$0
    proc(int1 : integer; t1 : T; t2 : T; int2: integer; int3 :
integer): string
    { return "proc$_args$integer_T_T_integer_integer"; }

    template (type T, type R) // $TEMPLATE$proc$1
    proc(str1 : R; int2 : integer; t1 : T; str1 : string; t2 :
T): string
    { return "proc$_args$R_integer_T_string_T"; }

    template (type T) // $TEMPLATE$proc$2

```

```

proc(t1 : T; t2 : T; t3 : T; str1 : string; str2 : string):
string
{ return "proc$_args$T_T_T_string_string"; }

template (type T) // $TEMPLATE$proc$3
proc(t1 : T; int1: integer; ok : boolean): string
{ return "proc$_args$T_integer_boolean"; }

template (type T) // $TEMPLATE$proc2$0
proc2(i : integer): string
{ return "proc2$_args$integer"; }

init1(): string
{ return proc(3; 4; 5; "Kim"; "Ola"); }

intit2(): string
{ return proc(3; 4; 5; "Kim"; 6); }

/* istnienie takiego szablonu oznaczałoby niemożliwość
jednomyślnego wyboru
template (type T, type R) // $TEMPLATE$proc$4
proc(str1 : T; int2 : T; t1 : T; str1 : R; t2 : R): string
{ return "proc_args$T_T_T_R_R"; }
*/
}

```

10.3 Zapytania jako argumenty aktualne

```

module spec
{
  class PersonClass
  {
    instance Person :
    {
      fName: string;
      lName: string;
    }
  }
}

```

```

        sex: string;
        age: integer;
    }
    getLastName(): string { return lName;}
    getFullName(): string { return lName + " " + fName;}
}

Person : PersonClass [0..*];

template (type T)
max_val(a : T; b : T): T
{
    if(a.age >= b.age)
        return a;
    else return b;
}

template (type T)
max_val(a : T; b : T; x : integer): T
{
    if (a.age >= b.age)
        return a;
    else return b;
}

createDatabase()
{
    p1: ref Person;
    p1 := create permanent Person
    (
        "Kim" as fName,
        "Chief" as lName,
        "M" as sex,
        25 as age
    );

    p2: ref Person;

```

```

    p2 := create permanent Person
    (
        "Ola" as fName,
        "Holy" as lName,
        "W" as sex,
        20 as age
    );
}

init_database()
{ createDatabase(); }

init_max_val_Person_int(): PersonClass
{
    return max_val(Person where fName = "Kim";
        Person where fName = "Ola"; 2+2);
}

init_max_val_Person(): PersonClass
{
    return max_val(Person where fName = "Kim";
        Person where fName = "Ola");
}
}

```

10.4 Wołanie szablonu w ciele innego szablonu

```

module generowanie_proc
{
    template (type T)
    _porownaj(ob1 : T; ob2:T): boolean
    { return ob1 = ob2; }
    template (type T)
    porownaj(obj1 : T; obj2 : T): string
    {

```

```

object1 : T;
object2 : T;

object1 := obj1;
object2 := obj2;
if (_porownaj(object1; object2))
    return "są sobie równe";
else
    return "są sobie nierówne";
}

init(): string
{ return porownaj(33; 44); }
}

```

10.5 Wywołania rekurencyjne procedur szablonowych

```

module rekurencja
{
    template (type T)
    suma(t : T; i: integer): T
    {
        pom : integer;
        pom := (i+(-1)); // UWAGA w tym miejscu jest zrobiona
        // sztuczka z odejmowaniem. Z niewiadomych przyczyn
        // operator '-' może być tylko używany jako
        // operator unarny!
        if (pom = 0)
            return t;
        else
            return t + suma(t; pom);
    }

    init_procInteger(): integer
    { return suma(3; 10); }
}

```

```

init_procString(): string
{ return suma("kot "; 10); }

init_procReal(): real
{ return suma(3.14; 10); }
}

```

10.6 Procedura specjalizowana

```

module spec
{
    class PersonClass
    {
        instance Person :
        {
            fName: string;
            lName: string;
            sex: string;
            age: integer;
        }

        getLastname():string { return lName;}
        getFullName():string { return lName + " " + fName;}
    }

    Person: PersonClass [0..*];

    template (type T)
    max_val(a : T; b : T): T
    {
        if (a >= b)
            return a;
        else return b;
    }
}

```

```

/***** procedura specjalizowana *****/
max_val(a : Person; b : Person): Person
{
    if (a.age >= b.age)
        return a;
    else return b;
}

init_run_Person(): Person
{
    p1 : ref Person;
    p1 := create permanent Person
    (
        "Kim" as fName,
        "Chief" as lName,
        "M" as sex,
        25 as age
    );

    p2 : ref Person;
    p2 := create permanent Person
    (
        "Ola" as fName,
        "Holy" as lName,
        "W" as sex,
        20 as age
    );

    // wołanie procedury wygenerowanej z szablonu
    return max_val(p1; p2);
}

// wołanie procedury specjalizowanej
init_run_integer(): integer
{ return max_val(1; 3); }
}

```

10.7 Jeden szablon szczególnym przypadkiem drugiego

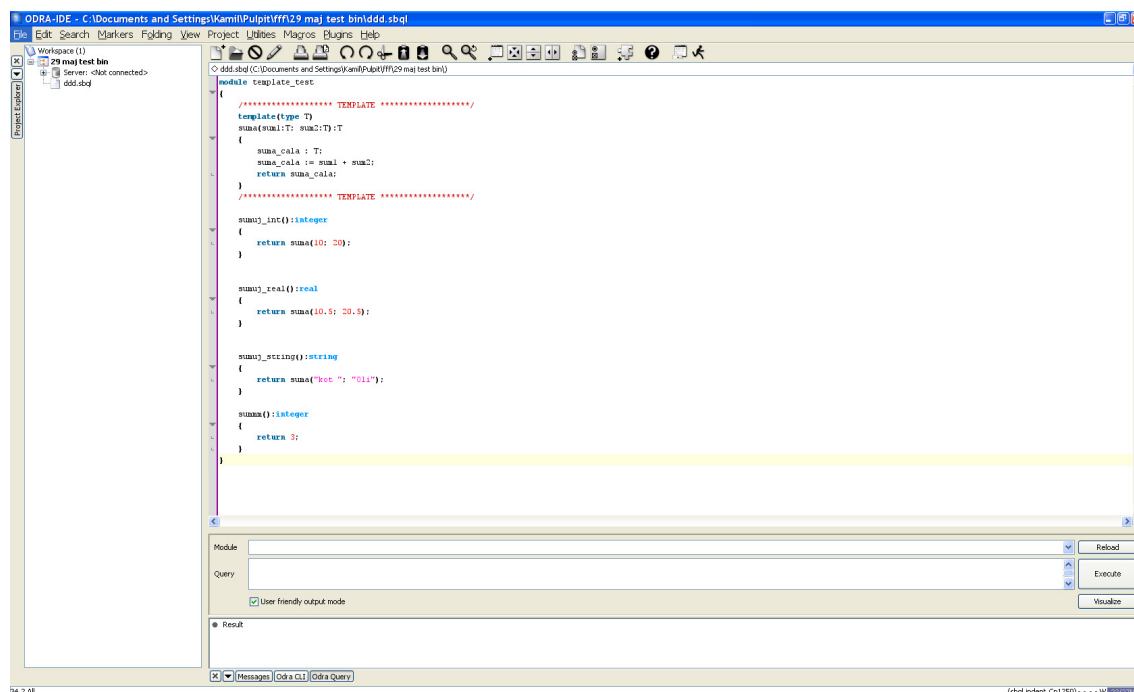
```
module ambigu_call
{
    template (type T)
    suma(a : T; b : T): T
    { return a + b; }

    template (type T, type R)
    suma (a : T; b : R): T
    { return a + b; }

    init()
    {
        // takie wywołanie spowoduje błąd kompilacji
        // "Two or more templates exist for one call of
        // procedure"
        suma(1; 3);
    }
}
```

11. DODATEK B – OPIS TECHNICZNY

Do niniejszej pracy został dołączony rezultat implementacji w postaci kodu źródłowego oraz plików wynikowych. Aby uruchomić pliki wynikowe wystarczy uruchomić skrypt „server recreate.bat” znajdujący się w katalogu „Odra z szablonami”. Aby podłączyć się do uruchomionego serwera należy włączyć klienta „cli.bat”. Jednakże, aby pracować w bardziej efektywny sposób z systemem ODRA należy uruchomić zintegrowane środowisko programistyczne (Odra-IDE – stworzone przez dr M. Trzaskę), dołączone do niniejszej pracy (katalog „ODRA-IDE_070906-01”).



Rysunek 14. Zintegrowane środowisko programistyczne Odra-IDE

12. DODATEK C - SŁOWNIK UŻYTEJ TERMINOLOGII I SKRÓTÓW

SBA (Stack Based Approach) - podejście stosowe

SBQL (Stack Based Query Language) - język oparty na podejściu stosowym

ODRA (Object Database for Rapid Application development) – implementacja obiektowej bazy danych zgodnej z teorią SBA.

ENVS (Enviromental Stack) – stos środowiskowy

QRES (Query Result Stack) - stos rezultatów

SZBD – System zarządzania bazą danych

AST (Abstract Syntax Tree) - rodzaj drzewa, które przedstawia strukturę programu komputerowego

ONP (Odwrotna Notacja Polska) - sposób zapisu wyrażeń arytmetycznych

XML (Extensible Markup Language) - uniwersalny język formalny przeznaczony do reprezentowania różnych danych w ustrukturalizowany sposób.

IDE (Integrated Development Environment) - zintegrowane środowisko programistyczne

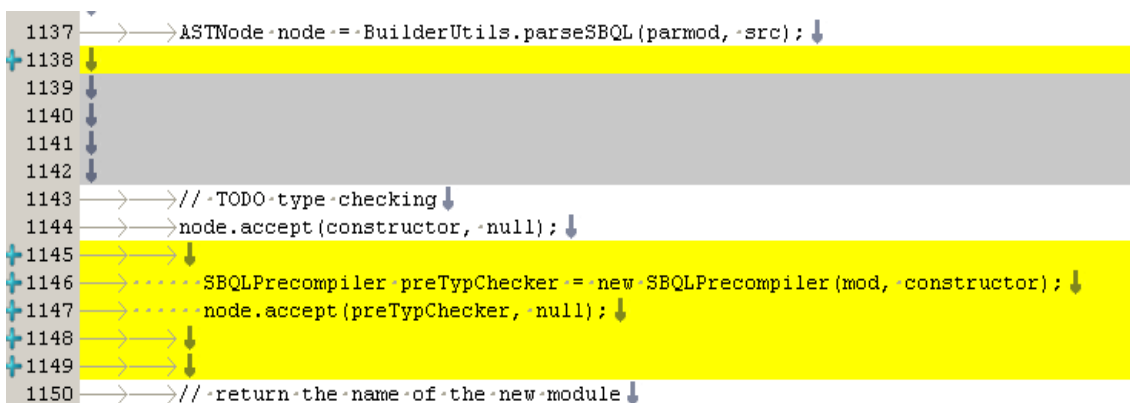
Tablica asocjacyjna (Associative Array) –abstrakcyjny typu danych, który przechowuje pary (unikalny klucz, wartość) i umożliwia dostęp do wartości poprzez podanie klucza.

Ada - język programowania opracowany w latach 70. Nazwa języka, pochodzi od imienia lady Augusty Ady Lovelace, uważanej za pierwszą programistkę w historii.

13. DODATEK D – PLIKI ZMIENIONE LUB DODANE DO KODU ŹRÓDŁOWEGO SYSTEMU ODRA

Główną częścią niniejszej pracy jest implementacja mechanizmu szablonów w systemie ODRA, która znajduje się w folderze „kod źródłowy” dostępnym na dołączonej płycie CD. Aby w szybki i wygodny sposób zobaczyć listę zmian dokonanych w porównaniu do oryginalnej wersji, należy zainstalować narzędzie TortoiseSVN dostępną na stronie <http://tortoisesvn.tigris.org/>. Narzędzie to wyselekcjonuje zmienione lub dodane pliki a także, pozwala z dokładnością do jednego znaku podać listę zmian uczynionych w projekcie.

Przykładowo:



```
1137 → → ASTNode·node·:=·BuilderUtils.parseSQL (parmod,·src); ↓
1138 ↓
1139 ↓
1140 ↓
1141 ↓
1142 ↓
1143 → → //·TODO·type·checking ↓
1144 → → node.accept (constructor,·null); ↓
1145 → → ↓
1146 → → .....SQLPrecompiler·preTypChecker·:=·new·SQLPrecompiler (mod,·constructor); ↓
1147 → → .....node.accept (preTypChecker,·null); ↓
1148 → → ↓
1149 → → ↓
1150 → → //·return·the·name·of·the·new·module ↓
```

Rysunek 15. Przykład zmian odczytanych za pomocą TortoiseSVN

Lista plików zmienionych lub dodanych przez autora.

res/lexer.lex

res/parser.cup

src/odra/cli/gui/components/ast/NodePanel.java

src/odra/db/IDataStore.java

src/odra/db/links/RemoteDefaultStore.java

src/odra/db/objects/data/DBSystemModule.java

src/odra/db/objects/meta/MetabaseManager.java

src/odra/db/objects/meta/MetaObjectKind.java

src/odra/db/objects/meta/PrimitiveTypeKind.java

src/odra/db/schema/OdraProcedureSchema.java

src/odra/db/schema/OdraViewSchema.java

src/odra/db/StdEnvironment.java
src/odra/dbinstance/processes/ServerProcess.java
src/odra/sbql/ast/ASTAdapter.java
src/odra/sbql/ast/ASTVisitor.java
src/odra/sbql/ast/declarations/VariableDeclaration.java
src/odra/sbql/ast/statements/VariableDeclarationStatement.java
src/odra/sbql/ast/terminals/Name.java
src/odra/sbql/builder/BuilderUtils.java
src/odra/sbql/builder/DatabaseManager.java
src/odra/sbql/builder/ModuleCompiler.java
src/odra/sbql/builder/ModuleConstructor.java
src/odra/sbql/builder/ModuleOrganizer.java
src/odra/sbql/emitter/OpCodes.java
src/odra/sbql/emitter/SafeJulietCodeGenerator.java
src/odra/sbql/results/compiletime/util/ValueSignatureInfo.java
src/odra/sbql/results/compiletime/util/ValueSignatureType.java
src/odra/sbql/typechecker/SBQLTypeChecker.java
src/odra/sbql/typechecker/SBQLTypeCheckerHelper.java
src/odra/sbql/typechecker/StaticEnvironmentManager.java
src/odra/store/DefaultStore.java
src/odra/store/sbastore/ObjectManager.java
src/odra/store/sbastore/ODRAObjectKind.java
res/SBQLLexer.java
src/odra/db/objects/meta/MBTemplate.java
src/odra/sbql/ast/declarations/SequentialTemplateArgumentDeclaration.java
src/odra/sbql/ast/declarations/SingleTemplateArgumentDeclaration.java
src/odra/sbql/ast/declarations/TemplateArgumentDeclaration.java
src/odra/sbql/ast/declarations/TemplateDeclaration.java
src/odra/sbql/ast/declarations/TemplateFieldDeclaration.java
src/odra/sbql/ast/declarations/TemplateHeaderDeclaration.java
src/odra/sbql/ast/expressions/TemplateExpression.java
src/odra/sbql/precompiler
src/odra/sbql/precompiler/PrecompilerException.java
src/odra/sbql/precompiler/SBQLPrecompiler.java

src/odra/sbql/precompiler/SBQLPrecompilerHelper.java
src/odra/sbql/precompiler/SBQLPreTypeInfo.java
src/odra/sbql/precompiler/SBQLTemplateUnification.java
src/odra/sbql/results/compiletime/MetaTypeSignature.java