(cz. I)", Software nr 11, listopad 1997
(cz. II)", Software nr 12, grudzień 1997
Zygmunt BOK
Zakłady Tworzyw Sztucznych 'NITRON' S.A.
ul. Zawadzkiego 1, 42-693 Krupski Młyn

ANALIZA MODELU WYMIANY DANYCH

POMIĘDZY BAZAMI I APLIKACJAMI ZORIENTOWANYMI OBIEKTOWO W ŚRODOWISKU OBJECT PASCAL

Streszczenie. W artykule przedstawiono analizę zaimplementowanego w środowisku *Object Pascal* modelu mechanizmu strumieni w procesie wymiany danych w postaci obiektów pomiędzy bazami i aplikacjami zorientowanymi obiektowo. W szczególności przedstawiono charakterystyczne dla tej implementacji mechanizmy rejestracji typów obiektowych w systemie strumieni, jak również ich składowania i odczytywania.

ANALYSIS OF THE DATA EXCHANGE MODEL

BETWEEN OBJECT-ORIENTED DATABASES AND APPLICATIONS IN OBJECT PASCAL ENVIRONMENT

Summary. In this article an analysis of the streams mechanism model implemented in the *Object Pascal* environment in the process of the object data exchange between object-oriented databases and applications has been presented. In particular, an characteristic to this implementation object types registration mechanisms with streams and their backup and restore mechanisms have been presented.

1. W P R O W A D Z E N I E

Technika programowania zorientowanego obiektowo, zaimplementowana w środowisku *Object Pascal* (*Turbo Vision, Object Windows, Delphi, Lazarus*) daje silne mechanizmy do hermetyzacji (*encapsulation*) kodu i danych jak również do budowy wzajemnie powiązanych struktur obiektów.

Jedną z najważniejszych zasad w programowaniu zorientowanym obiektowo jest zasada, że programista podczas projektowania programu powinien myśleć o kodzie oraz o danych jednocześnie, bowiem dane kierują przepływem wykonywanego kodu, natomiast kod manipuluje postacią i wartościami danych.
Według autorów [**3,4,7,8**], hermetyzacja to stopienie (połączenie) kodu i danych do postaci obiektu oraz ukrycie szczegółów implementacyjnych. Korzyści jakie daje hermetyzacja to modularność oraz izolacja kodu zaimplementowanego w obiekcie od kodu innych obiektów.

Co jednak zrobić w przypadku, kiedy zachodzi konieczność wykonania prostej czynności jak np. zapamiętania kilka obiektów na dysku, stanowiącej jeden z podstawowych elementów systemu zarządzania obiektowymi bazami danych typu *OO-DBMS* (*Object-Oriented Database Managament System*)? Można oczywiście odseparować dane od obiektów i zapisać je w pliku na dysku, lecz byłby to krok wstecz. W tym przypadku z pomocą przychodzi mechanizm strumieni (*streams*).
Strumieniem w środowisku *Object Pascal* jest kolekcja obiektów na swojej drodze do jakiegoś miejsca przeznaczenia: zwykle do pliku, do EMS, do portu szeregowego lub do innych urządzeń. Strumienie obsługują operacje we/wy nie na poziomie danych, lecz na poziomie obiektów, zachowując wszystkie zalety związane z hermetyzacją.

2. STRUMIENIE A PROBLEM ZARZĄDZĄNIA DANYMI

2.1. SPOSOBY ZARZĄDZANIA DANYMI

Obecnie, wg autora [**4**], istnieją trzy zasadnicze podejścia stosowane do zarządzania danymi:

* *Zwykły plik* - zarządzanie za pomocą zwykłego pliku zapewnia podstawową obsługę plików i sortowanie,

* *System zarządzania relacyjną bazą danych* - opierający się na teorii relacyjnej,

* *System zarządzania obiektowymi bazami danych*. Systemy zarządzania obiektowymi bazami danych typu *OO-DBMS*, są najnowszą technologią implementacji. Produkty handlowe pojawiły się po raz pierwszy w 1986 roku. Producenci przyjmują jedno z dwóch głównych rozwiązań - *rozszerzone relacyjne* i *rozszerzony obiektowo* zorientowany język programowania typu *OOPL* (*Object Oriented Programming Language*). Dostępne na rynku *rozszerzone systemy relacyjne* wzbogacają system zarządzania relacyjnymi bazami danych dodając do nich abstrakcyjne typy danych oraz dziedziczenie,

które według [**7**] jest mechanizmem do wyrażania podobieństwa pomiędzy obiektami lub inaczej - współdzielenia [**8**] atrybutów oraz metod, jak również kilka usług ogólnego przeznaczenia do tworzenia obiektów i manipulowania nimi.
W drugim rozwiązaniu, *rozszerzony obiektowo OOPL* wzbogacony jest o składnię i możliwości zarządzania odczytem i zapamiętywaniem obiektu w bazie danych. Może to dać twórcy oprogramowania jednolite spojrzenie na wszystkie obiekty.
Obecnie większość obiektowych schematów zarządzania danymi korzysta z rozwiązania 'kopiowania obiektów' - zapisując wartości obiektu, a później tworząc jego kopię. Rozszerzony *OOPL OO-DBMS,* w przeciwieństwie do tego rozwiązania, może zrobić jednak więcej, zapewniając 'trwałe obiekty' - gdzie zapisuje się dokładnie ten sam obiekt wraz z jego wewnętrznym identyfikatorem, a nie kopię obiektu. W ten sposób, gdy obiekt zostanie odtworzony, jest identyczny z obiektem, który istniał uprzednio. „Trwałe obiekty" stanowią również podstawę do współdzielonego dostępu do pojedynczego obiektu z serwera obiektów w systemach wielodostępnych.

2.2. PROBLEMY WYSTĘPUJĄCE W OPERACJACH WE/WY

Mechanizm strumieni w *Object Pascal* został zaimplementowany w celu przezwyciężenia dwóch podstawowych problemów;

- *Problem 1*. Jest zasadą, że przed wykonaniem operacji we/wy do pliku, kompilator musi znać typ danych zapisywanych do pliku, natomiast typ pliku musi być określony w czasie kompilacji. Ponadto wszystkie elementy pliku również muszą być tego samego typu. Chociaż język *Turbo Pascal* zaproponował obejście tej zasady poprzez wprowadzenie plików bez typu, tzw. pliki blokowe z procedurami *BlockRead* oraz *BlockWrite*, lecz brak sprawdzania zgodności typów podczas kompilacji, pozwalający co prawda na wykonanie bardzo szybkich binarnych operacji we/wy, powoduje jednak nałożenie dodatkowej odpowiedzialności w tym zakresie na programistę. Co jednak zrobić w przypadku, kiedy zaistnieje potrzeba zachowania obiektów różnych typów?

- *Problem 2*. Brak możliwości użycia pliku do bezpośredniego zapisywania obiektów. *Turbo Pascal* nie zezwala na utworzenie pliku typu *object*, ponieważ zapisywane obiekty mogą zawierać metody wirtualne, których adresy określone są w czasie wykonania programu (*at Run-time*) na podstawie specjalnej tablicy *VMT* [**2,7**], zatem zachowywanie informacji z tych metod poza programem jest bezcelowe. Odczytywanie takich informacji jest jeszcze gorsze.

2.3. POLIMORFIZM STRUMIENI

Na podstawowym poziomie można myśleć o strumieniach w podobny sposób jak o plikach *Pascal*'owych. Pliki te mogą być prostymi sekwencyjnymi urządzeniami we/wy. Strumienie natomiast są polimorficznymi sekwencyjnymi urządzeniami we/wy, co oznacza, że zachowują się bardzo podobnie do plików sekwencyjnych z możliwością czytania i zapisywania obiektów różnych typów, których typ nie musi być określony w czasie kompilacji. Jak długo bowiem strumienie operują na obiektach pochodnych, wyprowadzonych dzięki mechanizmowi dziedziczenia z najbardziej podstawowego obiektu *TObject*, tak długo obiekty te mogą być zapisywane do tego samego strumienia, jako grupa identycznych lub całkowicie

różnych typów. Definicja obiektu *TObject* przedstawiono na Wydr. 1., z którego widać, że obiekt ten zapewnia podstawowe metody, których implementację przedstawiono na Wydr. 2.

```
PObject = ^TObject;
 TObject = object
  constructor Init;
  procedure Free;
  destructor Done; virtual;
 end;
```

Wydr. 1. Definicja obiektu *TObject*
List. 1. *TObject* object definition

```
constructor TObject.Init;
type
   Image = record
             Link: Word;
             Data: record end;
           end;
begin
 {$IFNDEF Windows}
     FillChar(Image(Self).Data, SizeOf(Self) - SizeOf(TObject), 0);
 {$ENDIF}
end;

procedure TObject.Free;
begin
 Dispose(PObject(@Self), Done);
end;

destructor TObject.Done;
begin
end;
```

Wydr. 2. Podstawowe metody obiektu *TObject*
List. 2. Basic methods of the *TObject* object

2.4. REJESTRACJA OBIEKTÓW

Mając na uwadze polimorfizm strumieni, nasuwają się następujące pytania:

- w jaki sposób ten sam strumień może czytać i zapisywać tak różne obiekty jak np. *TWindow*, *TDialog* czy *TCollection*, nie potrzebując nawet wiedzieć, w trakcie kompilacji, z jakimi obiektami będzie miał do czynienia?

- W jaki sposób strumień może obsługiwać nowe typy obiektów, które nie były jeszcze utworzone, w momencie kiedy strumień był kompilowany?

Odpowiedzią na te pytania jest *rejestracja* (*registration*). Przedstawiając zagadnienie w pewnym skrócie można powiedzieć, że każdy typ obiektu, jak również wyprowadzone nowe typy z typów już istniejących, posiada przyporządkowany, za pomocą procedury *RegisterType*, unikalny numer rejestracyjny. Ten unikalny numer zostaje wpisany do strumienia w procesie zapisywania obiektu (np. do pliku na dysku) przed danymi pochodzącymi z zapisywanego obiektu.

W przypadku procesu odwrotnego, tj. odczytywania obiektu z pliku, strumień najpierw pobiera numer identyfikacyjny odczytywanego obiektu, a następnie bazując na tej informacji, odczytuje właściwą liczbę bajtów danych oraz właściwe metody wirtualne, które dołącza do odczytanych danych.

2.5. INICJALIZACJA STRUMIENIA

W celu efektywnego wykorzystania strumienia w procesie zapisu i odczytu obiektów z pliku, należy dokonać jego inicjalizacji. Mechanizm strumieni oparty jest o obiekt *TStream,*

którego definicję przedstawiono na Wydr. 3. Jak widać, *TStream* jest obiektem potomnym, wywodzącym się z obiektu nadrzędnego podstawowego *TObject.* Jest to obiekt abstrakcyjny wraz ze swoimi najbardziej podstawowymi metodami tj. *Get, Put* oraz *Error.* Do inicjacji strumienia należy zatem użyć obiektu potomnego. Dla programisty dostępne są następujące strumienie: *TMemoryStream, TEmsStream* oraz *TDosStream*, z którego wyprowadzono *TBufStream*. Metody *Get, Put*, z grubsza biorąc, odpowiadają *Pascal*'owym procedurom *Read* oraz *Write*, które są używane w zwykłych plikowych operacjach we/wy. Przykładowo, w celu inicjalizacji buforowanego strumienia *TBufStream*, zapewniającego buforowane dyskowe operacje we/wy i użytecznego w przypadku czytania i pisania dużej ilości lecz małych porcji informacji na dysk, służącego do transmisji kolekcji obiektów pomiędzy programem a plikiem, należy w programie umieścić deklaracje i polecenia, przedstawionym na Wydr. 4.

```
PStream = ^TStream;
TStream = object( TObject )
   Status: Integer;
   ErrorInfo: Integer;
   constructor Init;
   procedure CopyFrom(var S: TStream; Count: Longint);
   procedure Error(Code, Info: Integer); virtual;
   procedure Flush; virtual;
   function Get: PObject;
   function GetPos: Longint; virtual;
   function GetSize: Longint; virtual;
   procedure Put(P: PObject);
   procedure Read(var Buf; Count: Word); virtual;
   function ReadStr: PString;
   procedure Reset;
   procedure Seek(Pos: Longint); virtual;
   function StrRead: PChar;
   procedure StrWrite(P: PChar);
   procedure Truncate; virtual;
   procedure Write(var Buf; Count: Word); virtual;
   procedure WriteStr(P: PString);
end;
```

Wydr. 3. Definicja obiektu *TStream*
List. 3. *TStream* object definition

```
var
  BufDosStream: TBufStream;
begin
  BufDosStream.Init('Kolekcja.DAT', stOpen, 1024);
  .
  .
end.
```

Wydr. 4. Inicjalizacja buforowanego strumienia *TBufStream*
List. 4. *TBufStream* buffered stream initialization

2.6. CZYTANIE I ZAPISYWANIE OBIEKTÓW DO STRUMIENIA

Z każdym obiektem wywodzącym się z obiektu podstawowego *TObject*, tworzona jest automatycznie przez kompilator tablica *VMT* (*Virtual Method Table*) dla wszystkich typów obiektów, lecz nie dla ich wystąpień (instancji), zawierających lub dziedziczących metody wirtualne, tzn. jedna tablica *VMT* na jeden typ obiektu.
Zdefiniujmy przykładowy, przedstawiony na Wydr. 5., obiekt *TMojObject*, pochodzący od obiektu podstawowego *TObject*, o którym zakłada się, że był wcześniej zarejestrowany, natomiast *BufDosStream* jest zainicjowanym strumieniem (obiektem) pochodzącym od obiektu *TStream*.

```
PMojObject = ^TMojObject;
TMojObject = object( TObject )
   constructor Init;
   destructor Done; virtual;
end;
```

Wydr. 5. Definicja obiektu *TMoj Object*
List. 5. *TMojObject* object definition

Przyjrzyjmy się bliżej metodzie *Put,* ze strumienia *BufDosStream*, której składnia jest następująca: *BufDosStream.**Put**(PMojObiekt).* Strumień *BufDosStream*, bazując na informacji odczytanej ze skojarzonej z obiektem *PMojObiekt* tablicy *VMT*, jest w stanie określić:

- jakiego typu jest to obiekt,
- jaki jest jego numer identyfikacyjny (*ID*)
- wielkość obiektu w bajtach (dane oraz kod)
- rzeczywiste adresy metod, w szczególności adresy metod *Get* oraz *Put.*

Na podstawie powyższych informacji strumień 'wie', jaki numer identyfikacyjny *ID* zapisać do strumienia oraz ile bajtów informacji ma zapisać po nim, używając właściwej metody *Put*, której adres, według [**2,7**], umieszczony jest w tej tablicy podczas kompilacji, dając w ten sposób wiązanie dynamiczne.

Na specjalną uwagę zasługuje fakt, że w przypadku wykonania metody *Put* dla złożonego obiektu rodzicielskiego (*parent*) zawierającego szereg obiektów podrzędnych (*child*), wówczas wszystkie obiekty podrzędne zostaną zapisane do pliku automatycznie. Kiedy program zostanie ponownie uruchomiony i dokona się odczytu obiektu złożonego z pliku, wówczas będzie on w tym samym stanie, w jakim był w chwili jego zapisywania.

W celu odczytywania obiektów z pliku za pomocą metody *Get*, której ogólna składnia jest następująca: *BufDosStream.**Get**(PMojObiekt)*, zwracany jest wskaźnik do odczytanej informacji. W tym przypadku, ile bajtów informacji zostanie odczytanych oraz jaki typ *VMT* zostanie przypisany do odczytanej informacji, określane jest nie z typu obiektu *PMojObiekt*, lecz z typu obiektu odczytywanego ze strumienia. Tak więc, jeśli obiekt będący na bieżącej pozycji strumienia *BufDosStream* nie jest tego samego typu co typ obiektu *PMojObiekt*, wówczas odczytana informacja będzie zafałszowana, zaśmiecona (*garbled*).

2.7. TWORZENIE NOWYCH OBIEKTÓW WSPÓŁPRACUJĄCYCH ZE STRUMIENIAMI

Wszystkie standardowe obiekty zdefiniowane w środowisku *Turbo Vision*, *Object Windows, Delphi*, gotowe są do ich zapisania w pliku, bądź odczytania, za pomocą zaimplementowanego mechanizmu strumieni. Co jednak zrobić w przypadku utworzenia nowego obiektu pochodzącego od jednego ze standardowych obiektów, który dodatkowo został wzbogacony o jedno lub kilka nowych pól. Jak poinformować system strumieni o jego istnieniu?

Aktualnie, czytanie i zapisywanie obiektów do strumieni obsługiwane jest przez metody *Load* oraz *Store*. Aby każdy z obiektów mógł być obsługiwany przez strumienie musi posiadać te metody; nie są one jednak nigdy wywoływane bezpośrednio, lecz za pomocą metod *Get* oraz *Put.*

Wszystko co programista musi zrobić, żeby odczytać lub zapisać obiekt do strumienia, to upewnić się, czy nowy obiekt '*wie'*, jak '*wysłać się*" do strumienia, kiedy przyjdzie na to czas. Dzięki mechanizmom dziedziczenia własności od swoich poprzedników, nowy obiekt musi tylko wiedzieć, jak odczytywać i zapisywać do strumienia dodatkowo dodane informacje (pola) - pozostałe pola obsługiwane są automatycznie dzięki odziedziczonym metodom.

Przykładowo, załóżmy, że wyprowadzono nowy obiekt *TMyWindow* , przedstawiony na Wydr. 6., z obiektu *TWindow,* do którego dodano nowe pole, np. *field1*.

```
type
  TMyWindow = object( TWindow )
    field1: integer;
    constructor Load(var S: Tstream);
    procedure   Store(var S: Tstream);
    .
    .
  end;
```

Wydr. 6. Definicja obiektu *TMyWindow* z dodatkowym polem
List. 6. *TMyWindow* object definition with additional field

Poniżej, na Wydr. 7, przedstawiono procedurę zapisywania i odczytywania obiektu *TMyWindow*, za pomocą metody *Store* oraz *Load*, z dodatkowym polem *field1*. Aby zapisać obiekt *TMyWindow* do strumienia, należy zapisać standardowy obiekt *TWindow*, następnie zaś zapisać dodatkowe pole *field1*. Tak samo należy postąpić przy odczytywaniu obiektu *TMyWindow*; najpierw odczytać standardowy obiekt *Twindow*, a potem pole *field1*.

```
constructor TMyWindow.Load(var S: Tstream);

procedure TMyWindow.Store(var S: Tstream);
begin
  inherited Store(S);                    {zapisanie typu przodka 'ancestor'}
  S.Write(field1, SizeOf( field1 ));     {zapisanie dodatkowego pola field1}
end;

begin
  inherited Load(S);                     {odczytanie typu przodka 'ancestor'}
  S.Read(field1, SizeOf( field1 ));      {odczytanie dodatkowego pola field1}
end;
```

Wydr. 7. Procedura zapisywania i odczytywania obiektu z dodatkowym polem
List. 7. The object save and read procedures with an additional field

Ostatnią czynnością, którą programista musi wykonać, jest poinformowanie system strumieni o nowym obiekcie. Należy zatem:

1. zdefiniować rekord rejestracyjny strumieni typu *TStreamRec* dla tego obiektu, który pokazano na Wydr. 8.
2. przekazać go do procedury *RegisterType*, którą pokazano na Wydr. 9.

W omawianym przykładzie można, np. zdefiniować rekord rejestracyjny *RMyWindow* oraz wykonać procedurę *RegisterType*.

```
PStreamRec = ^TStreamRec;
TStreamRec = record
   ObjType: Word;
   VmtLink: Word;
   Load: Pointer;
   Store: Pointer;
   Next: Word;
end;
```

Wydr. 8. Rekord rejestracyjny strumieni typu *TStreamRec*
List. 8. Registration record of streams of the type *TStreamRec*

```
const

RMyWindow: TStreamRec=( ObjType: 100;
          VmtLink: Ofs(TypeOf(TMyWindow)^);
          Load:    @TMyWindow.Load;
          Store:   @TMyWindow.Store );

RegisterType( RMyWindow );
```

Wydr. 9. Rekord rejestracyjny *RMyWindow*. oraz procedura *RegisterType*
List. 9. Registration *RMyWindow* record and *RegisterType* procedure

gdzie pole:

ObjType - przechowuje unikalny numer identyfikacyjny nowego typu obiektu, który nadawany jest przez programistę. Środowisko *Turbo Vision* oraz *Object Windows* rezerwuje numery od 0 do 99 dla standardowych obiektów, natomiast programista swoim typom obiektowym może przydzielać liczby z zakresu 100 do 65535,

VmtLink - przechowuje offset nowego typu obiektu, który stanowi połączenie (*link*) rekordu rejestracyjnego z jego tablicą *VMT* zawartą w segmencie danych programu, inicjowaną przez konstruktora obiektu,

Load, Store - zawiera odpowiednio adres metody *Load* oraz *Store* nowego obiektu.

Next - przechowuje liczbę przyporządkowaną przez procedurę *RegisterType* do wewnętrznego wykorzystania połączonej listy rekordów rejestracyjnych przez strumienie. Nie wymaga żadnej obsługi ze strony programisty

Od tej pory, po wykonaniu powyższych czynności, wystąpienia (instancje) nowo zdefiniowanego obiektu *TMyWindow* można zapisywać i odczytywać za pomocą standardowych strumieni.

2.7.1. PROCEDURA REGISTRTYPE()

Po zdefiniowaniu rekordów rejestracyjnych, w celu ich rejestracji w systemie strumieni, należy przekazać je do procedury rejestracyjnej *RegistrerType(var S: TStreamRec)*.
Zadaniem tej procedury, pokazanej na Wydr. 10 i zaimplementowanej w języku assembler, jest przypisanie odpowiednim polom przekazanego rekordu typu *TStreamRec* właściwych wartości oraz utworzenie w pamięci *RAM* listy połączonych wzajemnie ze sobą, za pomocą pola *Next*, rekordów rejestracyjnych.

```
procedure RegisterType(var S: TStreamRec); assembler;
asm
        MOV     AX,DS
        CMP     AX,S.Word[2]
        JNE     @@1
        MOV     SI,S.Word[0]
        MOV     AX,[SI].TStreamRec.ObjType
        OR      AX,AX
        JE      @@1
        MOV     DI,StreamTypes
        MOV     [SI].TStreamRec.Next,DI
        JMP     @@3
@@1:    JMP     RegisterError
@@2:    CMP     AX,[DI].TStreamRec.ObjType
        JE      @@1
        MOV     DI,[DI].TStreamRec.Next
@@3:    OR      DI,DI
        JNE     @@2
        MOV     StreamTypes,SI
end;
```

Wydr. 10. Implementacja procedury *RegistrerType* w języku assembler
List. 10. Implementation of the *RegistrerType* procedure in the assembler language

2.8. MIEJSCE STRUMIENI NA TLE OBIEKTOWEGO SYSTEMU ZARZĄDZANIA DANYCH

W artykule [**8**] zacytowano szereg ogólnych obowiązkowych cech, które system *OODBS* powinien wykazywać, aby mógł być uznany jako obiektowo zorientowany. Biorąc te cechy pod uwagę widać wyraźnie, że mechanizm strumieni może stanowić podstawę do zbudowania, na jego na jego osnowie, obiektowego systemu zarządzania danych. W stanie obecnym, ze względu na brak chociażby takich cech jak:

- języka manipulowania danymi w bazie danych,
- mechanizmów transakcji, jak również ich współbieżnego wykonywania,
- języka zapytań,

mechanizm strumieni może być wykorzystany przez programistę do tworzenia oraz podstawowej obsługi prostych obiektowych baz danych.

3. STRUMIENIE - mechanizmy - struktura wewnętrzna

3.1. OBIEKTY - WEWNĘTRZNY FORMAT DANYCH

Wewnętrzny format, lub struktura danych, obiektu przypomina strukturę rekordu. Pola obiektu zapamiętywane są w porządku ich deklaracji, jako ciągła sekwencja zmiennych. Każde pole, odziedziczone po swoim przodku, zapamiętywane jest przed polami zdefiniowanymi w nowym typie potomnym. Jeśli w typie obiektowym zdefiniowano metody wirtualne, konstruktory lub destruktory, wówczas kompilator alokuje również dodatkowe pole w tym typie, tj. w sekwencji zmiennych. Jest to 16-bitowe pole zwane polem tablicy metod virtualnych *VMTF* (*Virtual Method Table Field*), które używane jest do zapamiętania *offset*'u obiektowej tablicy *VMT* w segmencie danych programu. Pole *offset*'u tablicy *VMT* następuje zaraz za zwykłymi polami zdefiniowanymi w jakimś typie obiektowym (patrz Rys. 1). W sytuacji, kiedy definiowany typ obiektowy dziedziczy po swoim przodku metody wirtualne, konstruktory lub destruktory, wówczas dziedziczy również pole *VMT* i żadne dodatkowe pola nie są już alokowane. Inicjalizacja pola *VMT* wystąpienia (instancji) jakiegoś typu obiektowego, obsługiwana jest przez konstruktora(ów) tego obiektu. Program nigdy bezpośrednio nie inicjalizuje, ani nie ma do niego dostępu. Przykładowo, rozważmy deklaracje typów obiektowych *Lokacja*, *Punkt* i *Okrąg* pokazanych na Wydr. 11, a na Rys. 1 zarys ich wystąpień (instancji). Każda pozycja odpowiada jednemu słowu pamięci (2 bajty). Ponieważ w przedstawionej hierarchii, typ obiektowy *Punkt* jest pierwszym typem w którym wprowadzono metody wirtualne, zatem pole *VMT* jest zaalokowane tuż za polem *Kolor*.

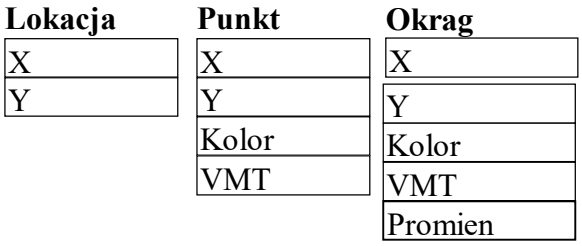
```
type

PLokacja = ^Lokacja
Lokacja = object
   X, Y: Integer;
   constructor  Init(LX, LY: Integer);
   function GetX: Integer;
   function GetY: Integer;
end;

PPunkt = ^Punkt;
Punkt = object( Lokacja )
   Kolor: Integer;
   constructor Init(LX, LY, LKolor: Integer);
   destructor Done;      virtual;
   procedure Show;       virtual;
   procedure Hide;       virtual;
   procedure MoveTo;     virtual;
end;

POkrag = ^Okrag;
Okrag = object( Punkt )
  Promien: Integer;
  constructor Init(LX, LY, LKolor, LPromien: Integer);
  procedure Show;        virtual;
  procedure Hide;        virtual;
  procedure Fill;        virtual;
end;
```

Wydr.11. Deklaracje typów obiektowych *Lokacja*, *Punkt*, *Okrag*
List. 11. Declarations of object types Location, Point, Okrag



Rys.1. Zarys instancji typów obiektowych *Lokacja*, *Punkt*, *Okrag*
Fig. 1. Instances layouts for object types *Lokacja*, *Punkt*, *Okrag*

3.2. OBIEKTY - METODY WIRTUALNE

3.2.1. METODY WIRTUALNE ORAZ POLIMORFIZM

Wykorzystywane w środowisku *Object Pascal* metody wirtualne opierają się na zaimplementowanym bardzo silnym mechanizmie zwanym polimorfizmem, za pomocą którego możliwe jest dokonywanie pewnych uogólnień. Polimorfizm z języka greckiego to „wiele

postaci”, i według [**3**] jest to sposób na przyporządkowanie pewnej akcji jednej nazwie, która jest współdzielona w górę i dół w hierarchii obiektów, tzn. występuje jako nazwa metody we wszystkich obiektach, począwszy od obiektu macierzystego poprzez wszystkie obiekty potomne, gdzie w każdym z tych obiektów w/w akcję można zaimplementować w inny sposób, uzyskując przez to inne własności obiektu.

3.3. TABLICE METOD WIRTUALNYCH VMT

Każdy typ obiektowy zawierający lub dziedziczący metody wirtualne, konstruktory lub destruktory posiada stowarzyszoną z nim tablicę metod wirtualnych *VMT*, składowaną w inicjującej części segmentu danych programu. Segment danych, który adresowany przez rejestr *DS* zawiera wszystkie typy stałych z następującymi po nich wszystkimi zmiennymi globalnymi. Rejestr *DS* nigdy nie jest zmieniany podczas wykonywania programu, natomiast rozmiar segmentu danych nie może przekroczyć 64 Kb.
Dla każdego typu obiektowego, za pomocą właściwego konstruktora, automatycznie tworzona jest tylko jedna tablica *VMT*; jednocześnie dwa różne typy obiektowe nigdy nie współdzielą tej samej tablicy *VMT*. W podobny sposób, tzn. za pomocą konstruktorów obiektów, automatycznie tworzone są wskaźniki do ich tablic wirtualnych i zapamiętywane są w deklarowanych w programie wystąpieniach (instancjach) tych typów. Zarówno tablice wirtualne *VMT*, jak również wskaźniki do nich, przez program nie są obsługiwane.

3.3.1. BUDOWA TABLICY METOD WIRTUALNYCH VMT

Tablica metod wirtualnych *VMT* dla pewnego typu obiektowego składa się z szeregu następujących po sobie elementów. W pierwszym elemencie tej tablicy, o długości jednego słowa, zapamiętana jest liczba określająca rozmiar wystąpienia (instancji) stowarzyszonej z właściwym dla niej typem obiektowym. Jest to informacja, która wykorzystywana jest przez konstruktora lub destruktora wystąpienia tego obiektu, w celu określenia, ile bajtów pamięci powinny zaalokować lub zwolnić używając rozszerzonej składni standardowych procedur *New* oraz *Dispose*.

Drugi element tej tablicy, również o długości jednego słowa, zawiera ujemną liczbę określająca rozmiar wystąpienia stowarzyszonej z właściwym dla niej typem obiektowym. Ta informacja wykorzystywana jest przez metody wirtualne wywołujące mechanizm sprawdzający dane *VM* (V*alidation Mechanism*), w celu wykrycia niezainicjowanych obiektów, tj. wystąpień obiektów dla których nie wywołano konstruktora. Kiedy za pomocą dyrektywy kompilatora {$R+} dokona się włączenia mechanizmu *VM*, wówczas kompilator generuje kod odpowiedzialny za wywołanie procedury sprawdzającej, której adres znajduje się w jednym z dalszych pól *VMT*, przed każdym wywołaniem metody wirtualnej. W końcu, poczynając od *offset*’u 4 tablicy *VMT*, rozpoczyna się lista 32-bitowych wskaźników, tj. adresów o długości dwóch słów, określających początki metod wirtualnych, tj. jeden wskaźnik na jedną metodę wirtualną w danym typie obiektowym w porządku ich deklaracji. Poniżej, w Tab.1, przedstawiono dalszy ciąg przykładu z paragrafu 3.1 w części dotyczącej zawartości tablic *VMT* dla zdefiniowanych tam obiektów *Lokacja*, *Punkt* oraz *Okrag*.

Tablica VMT obiektu *Punkt*
$0008
$FFF8
@Punkt.Done
@Punkt.Show
@Punkt.Hide
@Punkt.MoveTo

Tablica VMT obiektu *Okrag*
$000A
$FFF6
@Punkt.Done
@Okrag.Show
@Okrag.Hide
@Punkt.MoveTo
@Okrag.Fill

Tab.1 . Zawartość tablic *VMT* dla typów obiektowych *Lokacja*, *Punkt* oraz *Okrag*
Tab. 2. Contents of the *VMT* tables for object types *Lokacja*, *Punkt* and *Okrag*

3.3.2. WYWOŁYWANIE METOD WIRTUALNYCH

W celu wywołania jakiejś metody wirtualnej pewnego obiektu, kompilator generuje kod, który odczytuje zawartość 16-bitowego pola *VMT* tego obiektu przechowującego *offset* tablicy *VMT* zapisanej w segmencie danych programu, następnie zaś na podstawie adresu zapisanego w tablicy *VMT* tego obiektu, wywołuje właściwą metodę wirtualną. Przykładowo, mając wskaźnik *PP* wystąpienia obiektu typu *PPunkt* zdefiniowanego w par.3.1, wówczas wywołanie metody wirtualnej *PP^.Show* tego obiektu generuje kod, pokazany na Wydr. 12, w języku assembler.

```
LES    DI, PP                ;Załadowanie wskaźnika PP do rejestru ES:DI, przekazanego jako parametr Self.
PUSH  ES                     ;
PUSH  DI                     ;
MOV   DI, ES:[DI+6] ;Odczytanie offsetu tablicy VMT z pola VMT obiektu PPunkt (patrz rys. 1)
CALL  DWORD PTR [DI+8]       ;Odczytanie początku adresu metody Show (patrz Rys. 2) oraz jej wykonanie
```

Wydr. 12. Fragment kodu metody wirtualnej *PP^.Show* napisany w języku assembler
List.12. Snippet of virtual method code *PP^. Show* written in assembler language

Analiza tego krótkiego programu pozwala stwierdzić, że wykonana zostanie metoda *Punkt.Show*, ponieważ zawartość tablicy *VMT* pod *offset*'em 8 wskazuje na jej początek.
W przypadku metod zarówno wirtualnych, jak i statycznych, przekazywana jest niejawnie (*implicite*) do ciała metody, zawarta w liście parametrów, zawsze jako ostatnia, zmienna o identyfikatorze *Self*, która zawsze przybiera postać 32-bitowego wskaźnika do instancji, przez którą ta metoda jest wywoływana. Inaczej mówiąc, reprezentuje wystąpienie (instancję) obiektu. Przy powrocie z metody do programu głównego, metoda musi usunąć parametr *Self* ze stosu w taki sam sposób, w jaki usuwa wszystkie inne normalne parametry.

3.3.3. ZADANIA KONSTRUKTORÓW I DESTRUKTORÓW OBIEKTÓW

3.3.3.1 ZADANIA KONSTRUKTORÓW OBIEKTÓW

Każdy obiekt, posiadający metody wirtualne, musi posiadać swojego konstruktora. Konstruktor jest specjalnym typem metody, który wykonuje pewne czynności inicjalizujące, których efekty wykorzystywane są później przez metody wirtualne. Zanim metody wirtualne będą mogły być wywołane, najpierw musi być wywołany konstruktor obiektu.
Jakie są zatem zadania konstruktora obiektu? Każdy typ obiektowy posiada tablicę metod wirtualnych *VMT* w segmencie *DS* danych programu, która zawiera rozmiar typu obiektu oraz wskaźniki do początków obszarów pamięci *RAM*, zawierających kody poszczególnych metod wirtualnych. Zadaniem konstruktora obiektu jest ustanowienie połączenia (*link*) pomiędzy wystąpieniem (instancją) wywołującą konstruktora pewnego typu obiektowego, a jego tablicą metod wirtualnych *VMT*.

3.3.3.2 ZADANIA DESTRUKTORÓW OBIEKTÓW

W celu zwolnienia pamięci zaalokowanej dla instancji pewnego obiektu, używa się destruktorów. Destruktor jest specjalnym typem metody, którego celem jest czyszczenie i zwolnienie dynamicznie zaalokowanych w pamięci obiektów. Łączy w sobie zadania polegające na zwolnieniu pamięci na stosie z innymi czynnościami, które trzeba wykonać dla danego typu obiektowego. Jak każda metoda, również i destruktor może być metodą wirtualną. Podczas czyszczenia dynamicznie zaalokowanych obiektów destruktor gwarantuje, że zawsze zwalniana jest właściwa liczba bajtów pamięci stosu.

Ponieważ rozmiary różnych typów obiektowych są różne, skąd zatem destruktor ‘wie’ - kiedy nadejdzie czas zwolnienia obiektów zaalokowanych na stosie - jaką liczbę bajtów zaalokowanej pamięci zwolnić? Destruktor rozwiązuje ten problem przez odczytanie potrzebnych informacji z miejsca, gdzie były zapamiętane, tj. z tablicy *VMT* wystąpienia (instancji) obiektu, który ma być zwolniony. Tablica *VMT* dla każdego typu obiektowego dostępna jest przez niewidoczny parametr *Self*, który przekazywany jest do destruktora podczas jego wywołania.

3.3.4. WYWOŁYWANIE KONSTRUKTORÓW I DESTRUKTORÓW OBIEKTÓW

Przy wywoływaniu konstruktorów i destruktorów używa się tej samej konwencji jak przy wywoływaniu normalnych metod. Wyjątek stanowi dodatkowy parametr przekazywany pośrednio do ciała konstruktora (*implicite*) zwany parametrem *VMT* o rozmiarze jednego słowa, który przekazywany jest na stos przed parametrem *Self*.

3.3.4.1 WYWOŁYWANIE KONSTRUKTORÓW OBIEKTÓW

W przypadku wywoływania konstruktorów, parametr *VMT* zawierający *offset* do tablicy *VMT* zapamiętywany jest w elemencie tablicy *VMT*, w polu *Self*, w celu inicjacji parametru *Self*. Ponadto, w sytuacji kiedy konstruktor w celu dokonania alokacji obiektu dynamicznego, który wywoływany jest za pomocą rozszerzonej składni standardowej procedury *New*, wówczas w parametrze *Self* przekazywany jest wskaźnik o wartości *nil*.

Powoduje to, że konstruktor po zaalokowaniu nowego obiektu dynamicznego, podczas powrotu do procedury wywołującej zwraca w rejestrze *DX:AX* adres tego obiektu. Jeśli konstruktor nie mógł zaalokować nowego obiektu, wówczas w tym rejestrze zwracany jest wskaźnik o wartości *nil*.

W końcu, kiedy konstruktor wywoływany jest za pomocą identyfikatora typu obiektu, po którym następuje kropka i identyfikator metody, wówczas wartość zero jest przekazywana w parametrze *VMT*. Wskazuje to konstruktorowi, że nie powinien inicjować pola *Self* w elemencie tablicy *VMT*.

3.3.4.2 WYWOŁYWANIE DESTRUKTORÓW OBIEKTÓW

W przypadku wywoływania destruktorów, przekazane zero w parametrze *VMT* wskazuje normalne wywołanie, natomiast wartość niezerowa wskazuje, że destruktor był wywoływany za pomocą rozszerzonej składni standardowej procedury *Dispose*. Powoduje to, że destruktor zwalnia pamięć zajmowaną przez *Self* przed powrotem do procedury wywołującej. Rozmiar *Self* , a tym samym wielkość pamięci do zwolnienia, odczytywana jest z pierwszego elementu tablicy *VMT*.

3.4. PROCES ZAPISU I ODCZYTU OBIEKTÓW

Po wstępnym przeanalizowaniu strumieni, jako procesu, za pomocą którego możliwe jest odczytywanie i zapisywanie obiektów w strumieniach, powstaje pytanie, co tak naprawdę środowisko *Object Pascal* robi z obiektami, kiedy są odczytywane (metoda *Get*) i zapisywane (metoda *Put*) do strumieni.

3.4.1. PROCES ZAPISU I ODCZYTU OBIEKTÓW

W sytuacji, gdy do strumienia - za pomocą metody *TStream.Put* - wstawiany jest jakiś obiekt, wówczas strumień odczytuje wskaźnik do jego tablicy *VMT* z *offset*'u zerowego tego obiektu a następnie przegląda listę zarejestrowanych w systemie strumieni typów obiektów. Gdy wśród zarejestrowanych obiektów, tzn. ich rekordów rejestracyjnych typu *TStreamRec*, strumień odszuka pozycję którego pole *VmtLink* zawiera taki sam offset, wówczas strumień odczytuje unikalny numer rejestracyjny *ID* tego obiektu i zapisuje go do miejsca przeznaczenia, np. do pliku na dysku. W końcu, strumień na podstawie adresu metody *Store*, zapisanego w polu *Store* rekordu rejestracyjnego, wywołuje ją w celu ostatecznego zapisania wszystkich pozostałych informacji z obiektu. Implementację metody *TStream.**Put**,* dokonaną w języku assembler, przez f-mę *Borland* , pokazano na Wydr. 13.

W sytuacji odwrotnej, gdy za pomocą metody *TStream.Get* odczytywany jest jakiś obiekt, wówczas strumień najpierw odczytuje jego unikalny numer identyfikacyjny ID, a następnie wśród zarejestrowanych w systemie strumieni obiektów, tzn. ich rekordów rejestracyjnych typu *TStreamRec*, odszukuje pozycję, która posiada taki sam numer. W końcu, na podstawie rekordu rejestracyjnego, strumień odczytuje adres metody *Load* zapisanej w polu *Load* rekordu rejestracyjnego oraz tablicę metod wirtualnych *VMT* - na podstawie wskaźnika zapisanego w polu *VmtLink*. Po wywołaniu metody, odczytywana jest pozostała właściwa ilość danych, na podstawie informacji zawartej w pierwszym polu tablicy *VMT* przechowującej wielkość odczytywanego obiektu. Implementacja metody *TStream.Get* dokonana przez f-mę *Borland* pokazano na Wydr. 14.

```
procedure TStream.Put(P: PObject); assembler;
asm
LES     DI,P     ;Załadowanie wskaźnika P do rejestru ES:DI
        MOV     CX,ES
        OR      CX,DI
        JE      @@4
        MOV     AX,ES:[DI]
        MOV     BX,StreamTypes
        JMP     @@2
@@1:    CMP     AX,[BX].TStreamRec.VmtLink
        JE      @@3
        MOV     BX,[BX].TStreamRec.Next
@@2:    OR      BX,BX
        JNE     @@1
        LES     DI,Self
        MOV     DX,stPutError
        CALL    DoStreamError
        JMP     @@5
@@3:    MOV     CX,[BX].TStreamRec.ObjType
@@4:    PUSH    BX
        PUSH    CX
        MOV     AX,SP
        PUSH    SS
        PUSH    AX
        MOV     AX,2
        PUSH    AX
        LES     DI,Self
        PUSH    ES
        PUSH    DI
        MOV     DI,ES:[DI]
        CALL    DWORD PTR [DI].TStream_Write
        POP     CX
        POP     BX
        JCXZ    @@5
        LES     DI,Self
        PUSH    ES
        PUSH    DI
        PUSH    P.Word[2]
        PUSH    P.Word[0]
        CALL    [BX].TStreamRec.Store
@@5:
end;
```

Wydr. 13. Metoda wirtualna *TStream.Put* napisana w języku assembler

List.13. Virtual method *TStream.Put* written in assembler language

```
function TStream.Get: PObject; assembler;
asm
        PUSH    AX
        MOV     AX,SP
        PUSH    SS
        PUSH    AX
        MOV     AX,2
        PUSH    AX
        LES     DI,Self
        PUSH    ES
        PUSH    DI
        MOV     DI,ES:[DI]
        CALL    DWORD PTR [DI].TStream_Read
        POP     AX
        OR      AX,AX
        JE      @@3
        MOV     BX,StreamTypes
        JMP     @@2
@@1:    CMP     AX,[BX].TStreamRec.ObjType
        JE      @@4
        MOV     BX,[BX].TStreamRec.Next
@@2:    OR      BX,BX
        JNE     @@1
        LES     DI,Self
        MOV     DX,stGetError
        CALL    DoStreamError
@@3:    XOR     AX,AX
        MOV     DX,AX
        JMP     @@5
@@4:    LES     DI,Self
        PUSH    ES
        PUSH    DI
        PUSH    [BX].TStreamRec.VmtLink
        XOR     AX,AX
        PUSH    AX
        PUSH    AX
        CALL    [BX].TStreamRec.Load
@@5:
end;
```

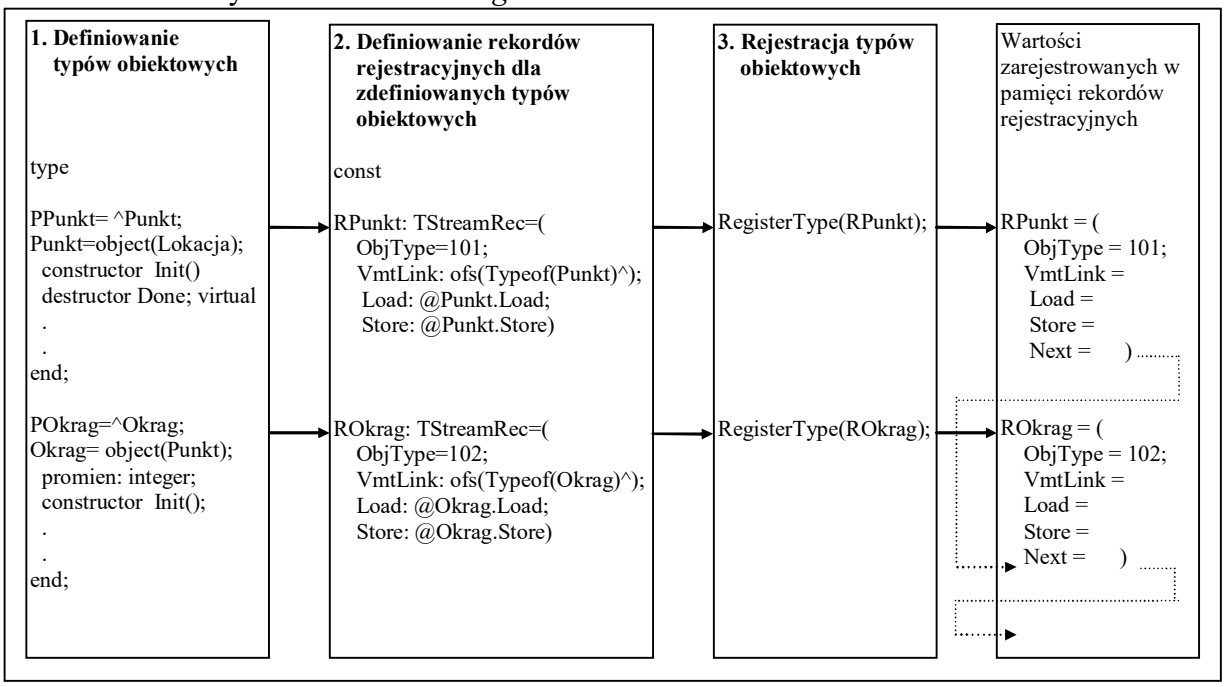
Wydr. 14. Metoda wirtualna *Stream.Get* napisana w języku assembler

List.14. Virtual method *TStream.Put* written in assembler language

3.5. PROCESY REJESTRACJI, ZAPISU I ODCZYTU OBIEKTÓW.

W celu ilustracji wcześniej omawianych procesów rejestracji, zapisu i odczytu obiektów, poniżej przedstawiono je w graficznej formie biorąc do rozważań zdefiniowane w paragrafie 3.1 przykładowe deklaracje typów obiektowych *Punkt* oraz *Okrag*.

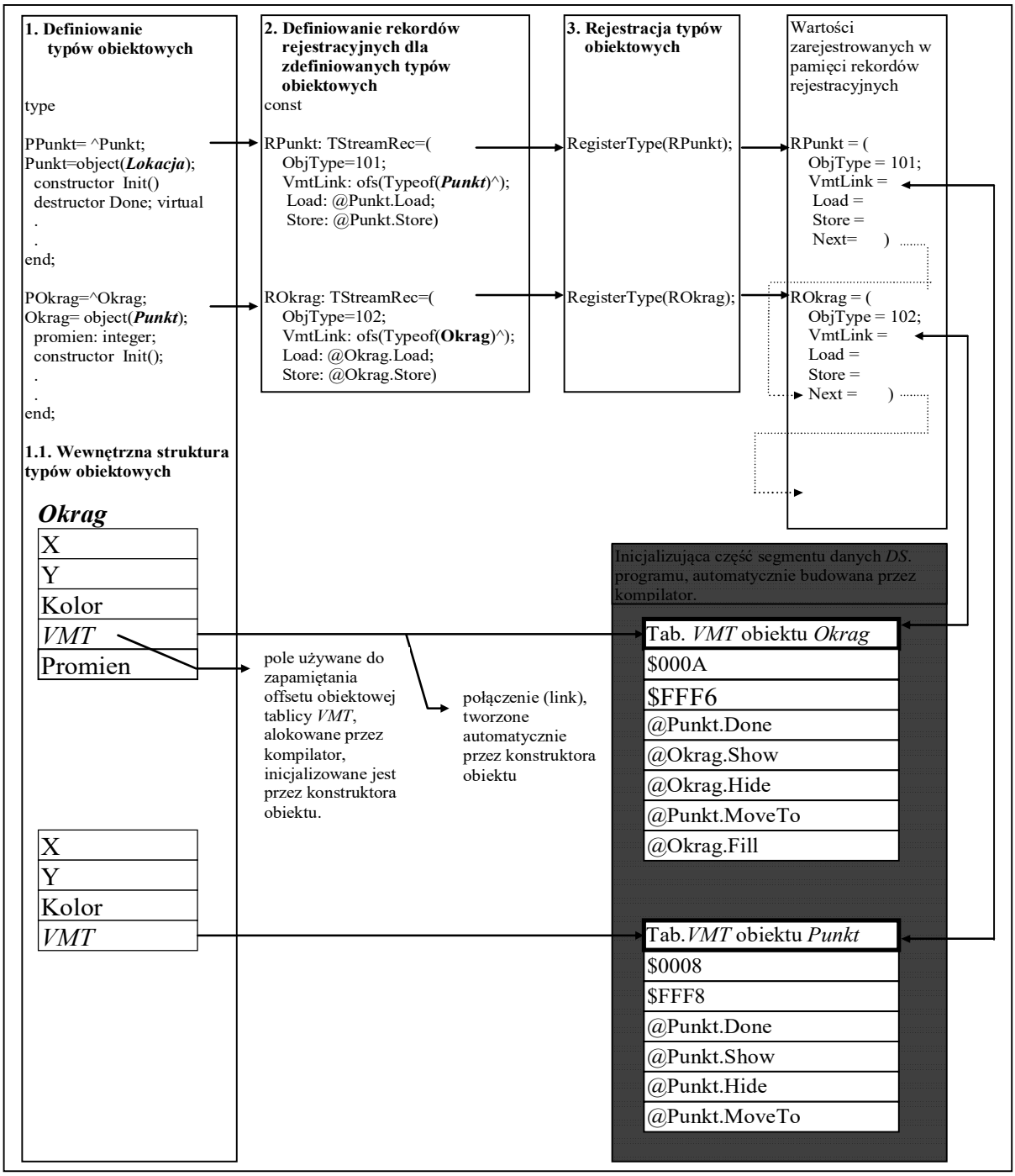
3.5.1. PROCES REJESTRACJI OBIEKTÓW - dla przykładowych deklaracji typów obiektowych *Punkt* oraz *Okrag*.



Rys. 2. Zarys procesu rejestracji typów obiektów
Fig. 2. Registration layout process for object types

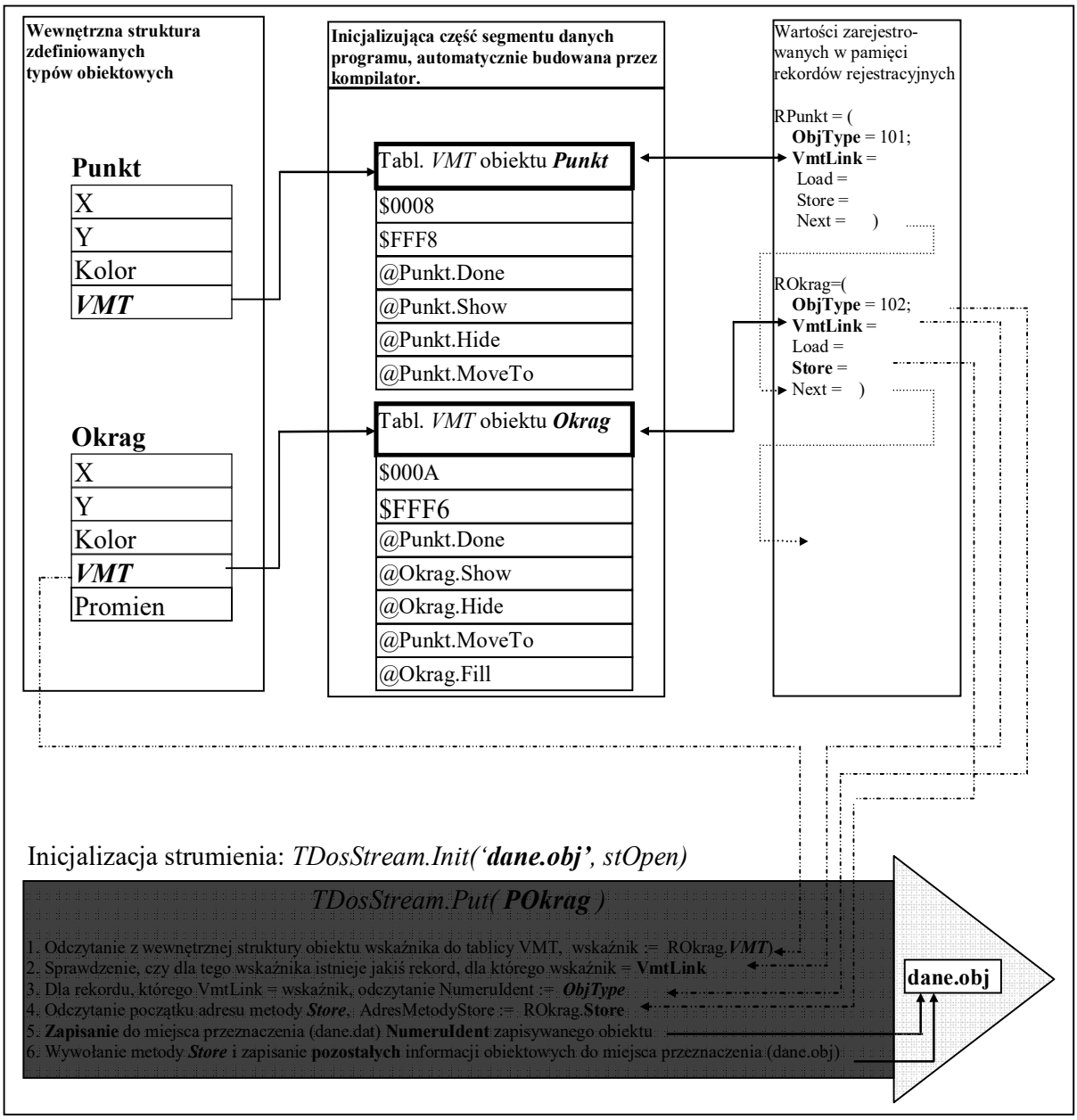
3.5.2. WZAJEMNE RELACJE POMIĘDZY OBIEKTAMI

Wzajemne relacje pomiędzy obiektami, ich tablicami *VMT* oraz rekordami rejestracyjnymi dla przykładowych deklaracji typów obiektowych *Punkt* oraz *Okrag*, pokazano na Rys. 3



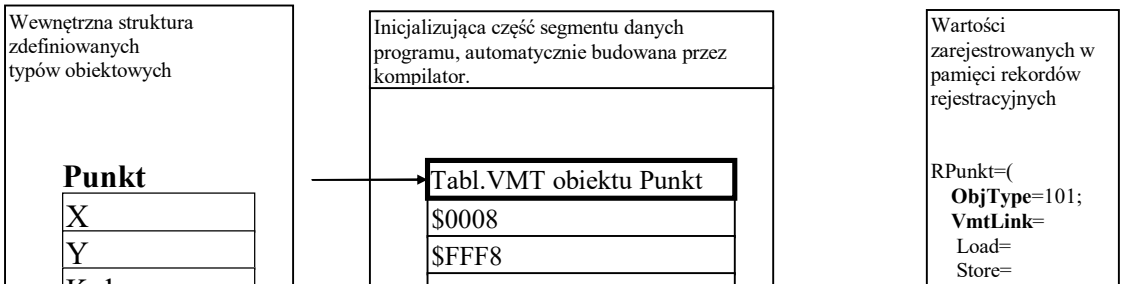
Rys. 3. Relacje pomiędzy obiektami, ich tablicami *VMT* oraz rekordami rejestracyjnymi
Fig. 3. Correlation relationships between objects, their *VMT* tables and registration records.

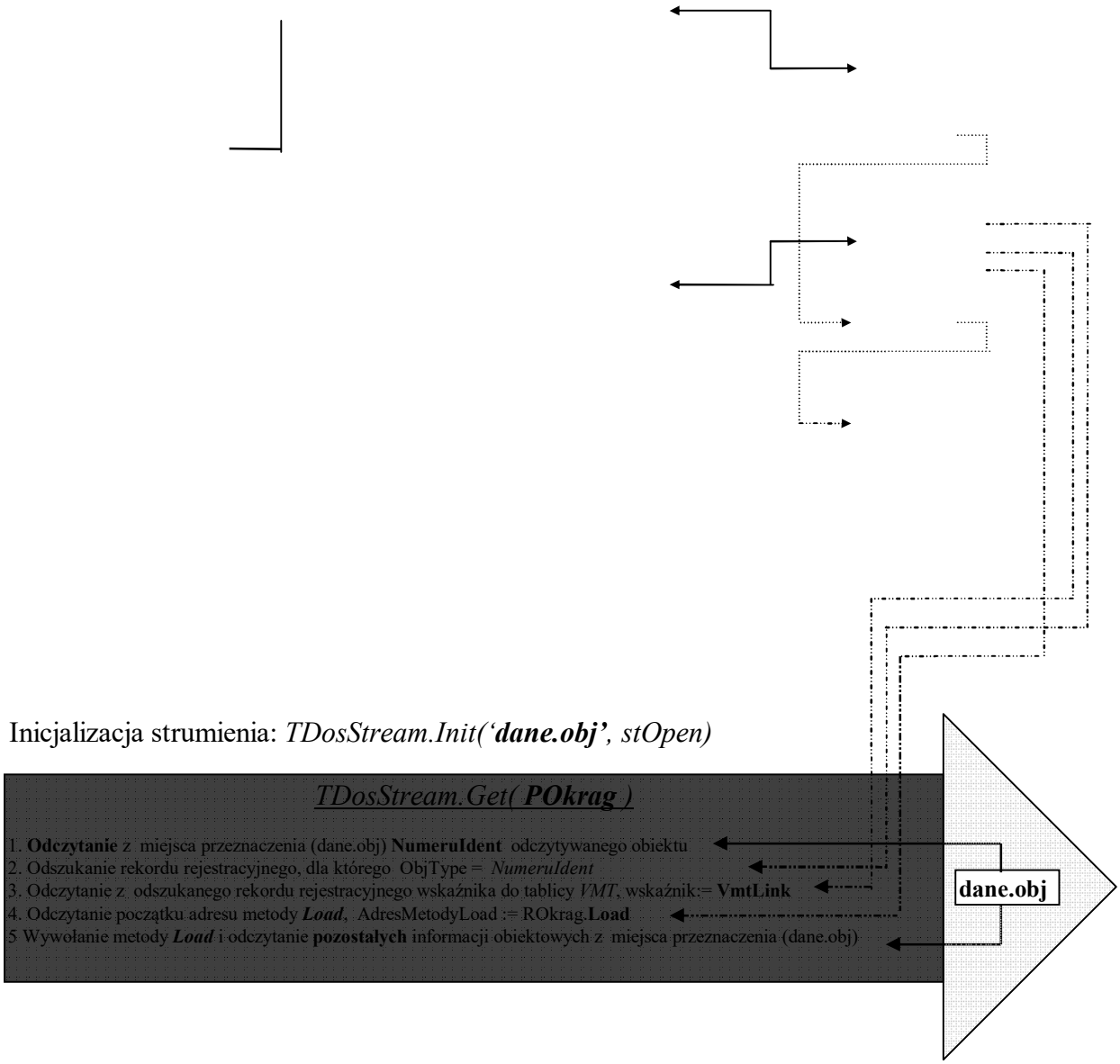
3.5.3. PROCES ZAPISU OBIEKTÓW - dla przykładowych deklaracji typów obiektowych *Punkt* oraz *Okrag*. Na Rys. 5 przedstawiono ilustrację procesu zapisu obiektu *Okrag* za pomocą metody *PUT* do strumienia (do miejsca przeznaczenia - plik o nazwie dane.obj).



Rys. 5. Proces zapisu obiektów do strumienia
Fig. 5. Write object process to the stream.

3.5.4. PROCESU ODCZYTU OBIEKTÓW - dla przykładowych deklaracji typów obiektowych *Punkt* oraz *Okrag*. Na Rys. 6 zilustrowano proces odczytu obiektu Okrag za pomocą metody *GET* ze strumienia (ze źródła danych - plik o nazwie dane.obj).





Rys. 6. Proces odczytu obiektów ze strumienia
Fig. 6. Read object process from the stream.

4. WYMIANA DANYCH POMIĘDZY APLIKACJĄ ZORIENTOWANĄ OBIEKTOWO A OBIEKTOWĄ BAZĄ DANYCH

4.1. ZARZĄDZANIE DANYMI - KOLEKCJE

Środowisko *Turbo Vision*, *Object Windows* oraz *Delphi* oferuje elastyczne i o dużych możliwościach narzędzie, w postaci typu obiektowego, zwanego kolekcją (*TCollection)* służące do zarządzania danymi. Kolekcja podobna jest w swej istocie do rozszerzalnej tablicy wskaźników, mogących wskazywać na pewien rodzaj danych, tj. na obiekty lub rekordy.

Za jej pomocą, w aplikacji zorientowanej obiektowo, można dokonywać następujących operacji:

- odczytać obiekty ze strumienia (z bazy danych),
- wyświetlić dane z poszczególnych obiektów,
- nawigacja pomiędzy obiektami,
- dodawanie i kasowanie obiektów,
- zapisywanie obiektów do strumienia (do bazy danych).

Istotnym elementem koniecznym do efektywnego wykonania w/w operacji jest interfejs użytkownika, w formie okna dialogowego o postaci formularza, z pewną ilością pól informacyjnych odpowiadających, ściśle według kolejności wyświetlania, polom pewnego rekordu transferowego, pełniącego rolę swoistego bufora pomiędzy interfejsem użytkownika a kolekcją.

4.2. WSPÓŁPRACA KOLEKCJI ZE STRUMIENIAMI

W celu efektywnego użycia kolekcji w systemie strumieni, niezbędnym elementem jest utworzenie pewnego obiektu pośredniczącego (rekord transferowy), pochodzącego od obiektu *TObject*, tworzącego pewnego rodzaju powłokę wokół danych obiektu, który musi posiadać następujące elementy:

- pole lub pola zawierające dane,
- zdefiniowaną metodę *Store* do zapisywania danych obiektu w strumieniu,
- zdefiniowaną metodę *Load* do odczytywania danych obiektu ze strumienia,
- rekord rejestracyjny.

Poniżej pokazano przykładową deklarację obiektu pośredniczącego *TKadryObj* wraz z rekordem transferowym.

```
type

PKadryObj = ^TKadryObj;
TKadryObj = object(TObject)
    RekordTransferowy: TRekordKadrowy;
    constructor Load (var S: TStream);
    procedure Store (var S: TSstream);
end;

constructor TKadryObj.Load (var S: Tstream);
begin
  inherited Load;                                          { odczytanie danych przodka }
  S.Read(RekordTransferowy, SizeOf(RekordTransferowy));    {odczytywanie danych ze strumienia}
end;

procedure TKadryObj.Store (var S: Tstream);
begin
  inherited Store;                                         {zachowanie danych przodka}
  S.Write(RekordTransferowy, SizeOf(RekordTransferowy));   {zachowanie dodatkowych danych do strumienia}
end;

RKadryObj: TStreamRek = ( ObjType: 2000;
                          VmtLink: Ofs(TypeOf(TKadryObj)^);
                          Load: @TKadryObj.Load;
                          Store: @TKadryObj.Store );
```

4.2.1. CZYTANIE KOLEKCJI ZE STRUMIENIA

Jedną z własności kolekcji jest możliwość współpracy z dowolnymi wskaźnikami do danych, lecz w sytuacji kiedy, kolekcja odczytuje się lub zapisuje do strumienia, wówczas zakłada, że każda jej pozycja (*item*) jest pewnym zarejestrowanym typem obiektowym wyprowadzonym z obiektu podstawowego *TObiect*.

Poniżej przedstawiono przykład, za pomocą którego można dokonać odczytu do kolekcji, korzystając z mechanizmu strumieni, przykładowych danych kadrowych, z pliku o nazwie „*KADRY.OBJ*" , będących w postaci obiektów.

```
procedure LoadKadry;
var
  StrumienPlikuKadrowego: TBufStream;
  KadrowaKolekcjaDanych: PCollection;
begin
   StrumienPlikuKadrowego.Init('KADRY.OBJ', stOpenRead, 1024);
   KadrowaKolekcjaDanych := StrumienPlikuKadrowego.Get;
   StrumienPlikuKadrowego.Done;
end;
```

Powyższy przykład ilustruje korzyści, jakie płyną z faktu gromadzenia danych w postaci obiektów w kolekcji. Pojedynczą instrukcją odczytuje się wszystkie obiekty, bez konieczności odczytywania poszczególnych obiektów.

4.2.2. MANIPULOWANIE ELEMENTAMI KOLEKCJI

W sytuacji, kiedy kolekcja obiektów została załadowana do pamięci operacyjnej, wówczas za pomocą obiektu pośredniczącego, *TKadryObj* omawianego w punkcie 4.2, możliwa jest współpraca z interfejsem użytkownika, który powinien umożliwić nawigację po wszystkich obiektach kolekcji, rozszerzanie, modyfikację, zmniejszanie oraz zapisywanie ich do strumienia. Poniższy przykład pokazuje, w jaki sposób można odczytać informacje kadrowe, za pomocą rekordu transferowego, z pierwszego obiektu kolekcji:

```
constructor TAppKadry.Init;
begin
  .
  .
  LoadKadry;
  BieżącyNrPracownika := 0;
  Informacje_o_Pracowniku := PKadryObj(KadrowaKolekcjaDanych^.At(BieżącyNrPracownika). RekordTransferowy);
end;
```

W podobny sposób można zapisać informacje kadrowe, za pomocą rekordu transferowego, z pewnego obiektu kolekcji:

```
procedure TAppKadry.ZachowajDaneKadrowe;
begin
  PKadryObj(KadrowaKolekcjaDanych^.At(BieżącyNrPracownika). RekordTransferowy) := Informacje_o_Pracowniku;
  SaveKadry;
end;

procedure SaveKadry;
var
  StrumienPlikuKadrowego: TBufStream;
begin
   StrumienPlikuKadrowego.Init('KADRY.OBJ', stOpenWrite, 1024);
```

```
  StrumienPlikuKadrowego.Put(KadrowaKolekcjaDanych);
  StrumienPlikuKadrowego.Done;
end;
```

5. PODSUMOWANIE

Zaimplementowany w środowisku *Turbo Vision*, *Object Windows* oraz *Delphi* mechanizm strumieni, dodatkowo wzbogacony o składnię i możliwości do zarządzania odczytem i zapamiętywaniem obiektów w obiektowej bazie danych, jest bardzo efektywnym i wygodnym narzędziem programisty dla tworzenia i obsługi obiektowych baz danych.

Mechanizm strumieni posiada szereg zalet, wśród nich najważniejsze to:

- prostota w użyciu,
- łatwość w modyfikowaniu własności predefiniowanych typów strumieni,
- zdolny, dzięki polimorfizmowi, do zapisywania w tym samym pliku różnych typów obiektowych,
- obsługuje operacje we/wy nie na poziomie danych, lecz na poziomie obiektów,
- duża szybkość w działaniu - dzięki implementacji najbardziej podstawowych metod w języku asemblera,
- zapewnia trwałe obiekty (*persistent objects*) - obiekt zapisuje się wraz z jego wewnętrznym identyfikatorem.

Do wad natomiast zaliczyć można:

- konieczność nadzorowania przydzielania unikalnych numerów *ID* nowo definiowanym typom obiektowym, w przeciwieństwie innych języków np. O_2C, gdzie unikalność zapewniona jest przez system [8],
- brak współdzielonego dostępu do pojedynczego obiektu z serwera obiektów w systemach wielodostępnych,
- brak zaimplementowanego mechanizmu przetwarzania transakcyjnego, jak również ich współbieżnego wykonywania, koniecznego do zachowania integralności obiektowej bazy danych.

Ponadto, w przypadku wymiany danych pomiędzy bazami i aplikacjami zorientowanymi obiektowo, zdefiniowany interfejs użytkownika w formie okna dialogowego, w postaci formularza z pewną ilością pól informacyjnych, musi odpowiadać, ściśle według kolejności wyświetlania, polom rekordu transferowego.

L I T E R A T U R A

1. BORLAND INTERNATIONAL *„Turbo Pascal 7.0”*
2. BORLAND INTERNATIONAL *„Programming Guide”*
3. BORLAND INTERNATIONAL *„Users Guide”*
4. PACHECO X.& TEIXEIRA S. „*Delphi2 - Developher’s Guide”,* SAMS Publishing, printed in the USA, 1996, ISBN:0-672-30914-9
5. MISIUREWICZ P. *„Układy mikroprocesorowe”,* WNT 1983, ISBN 83-204-0541-6

6. BIELECKI J. *„Turbo Pascal", WKŁ 1989,* ISBN 83-206-0903-8
7. COAD P., YOURDON E. *„Object-Oriented Design",* PRENTICE Hall, Inc., 1991, ISBN 83-85769-18-8
8. AUGUSTYN D., STĄPOR K., „*Obiektowo Zorientowany System Zarządzania Bazą Danych* O_2", Zeszyty Naukowe Politechniki Śląskiej, INFORMATYKA z.31, Gliwice, 1996

Abstract

In this article an analysis of the streams mechanism model implemented in the Turbo Pascal 7.0 environment in the process of the object data exchange between object-oriented databases and applications has been presented. In particular, an characteristic to this implementation object types registration mechanisms with streams and their backup and restore mechanisms have been presented. This article also presents an internal data format of objects especially their virtual method tables (VMT). Virtual method calls mechanism based on the object's VMT is described too.
Finally, an object data exchange model in this environment between an object-oriented application and database has been explained.