STUDIA INFORMATICA 2018
Volume 24 Number (48)

Zygmunt BOK

MECHANIZM STRUMIENI ŚRODOWISKA OBJECT PASCAL W BUDOWIE OBIEKTOWYCH BAZ DANYCH

Streszczenie. W artykule przedstawiono metodę wykorzystania systemu strumieni, zaimplementowanego w środowisku *Object Pascal* i współpracującego z kolekcją obiektów, do budowy obiektowej bazy danych. W szczególności przedstawiono charakterystyczne dla tej implementacji mechanizmy rejestracji typów obiektowych w systemie strumieni, jak również ich składowania i odczytywania. Zaproponowano metodę pozwalającą na zdefiniowanie podstawowych elementów do budowy interfejsu użytkownika, umożliwiające wykonanie elementarnych operacji na wszystkich obiektach kolekcji, tj. rozszerzanie, modyfikację, zmniejszanie oraz zapisywanie ich, poprzez system strumieni, do obiektowej bazy danych w postaci obiektów trwałych.

USING STREAMS IN THE OBJECT ENVIRONMENT PASCAL FOR CONSTRUCTION OF THE OBJECT DATABASE

Summary. In the article a method of the application system of streams was presented, implemented in the Object Pascal environment and cooperating with the collection of objects, for construction object database. In particular the mechanisms characteristic of this implementation of the registration of object types were presented in the system of streams, as well as storing them and reading. An allowing method for defining basic components for construction of the user interface was proposed, which enabling performance of basic operations on all objects of the collection, i.e. widening, an alteration, reducing and enrolling them, through the system of streams, in the object database in the form of long-lasting objects were proposed.

1. Wprowadzenie

Technika programowania zorientowanego obiektowo [**1**], zaimplementowana w środowisku *Object Pascal* i obecnym w takich produktach jak: *Turbo Vision, Object Windows, Delphi, Lazarus*, daje silne mechanizmy do hermetyzacji (*encapsulation*) oraz dziedziczenia [**2**] kodu i danych, jak również do budowy wzajemnie powiązanych struktur obiektów. *Object Pascal*, to obiektowe rozszerzenie języka *Pascal*. Różne implementacje języka *Object Pascal* użyte są w środowiskach programistycznych *Borland Delphi*, *CodeGear Delphi*, *Kylix* oraz w *developer*'skim *Delphi*, wykorzystujące bibliotekę *VCL* (*Visual Component Library*), stworzoną w języku *Object Pascal* przez firmę *Borland* na potrzeby tego środowiska, później zaadaptowaną też do środowiska *C++ Builder*. *VCL* jest biblioteką wspomagającą programowanie w środowisku *Windows*, zwłaszcza tworzenie interfejsu użytkownika. Była konkurencją dla biblioteki firmy *Microsoft – MFC*, dołączaną do środowiska *Visual Studio*.

Borland Delphi - jest *developer*'skim środowiskiem *Object Pascal*, bazującym na wcześniejszych produktach, w którym dodano *GUI IDE*, które ukierunkowało go w stronę środowisk *RAD* (*Rapid Application Development*). Pierwsza wersja *Delphi 1* została udostępniona w 1995 roku i przeznaczona była do pracy w 16-bitowym *Microsoft Windows 3.1*. Kolejne wersje udostępniane były w latach 1996-2003. *Delphi 7 Enterprise* – pracuje na szybkim i efektywnym kompilatorze 32/64 bit. Jest środowiskiem *developer*'skim typu *RAD* dla konsoli, desktopów, aplikacji *web*'owych i mobilnych, współpracujących z systemami operacyjnymi *Windows* (32-bit/64-bit), *Mac OS X, iOS* oraz *Android*. W wersji *Delphi 8* wprowadzono środowisko .NET. W latach 2005-2010 wprowadzane były wersje *Delphi* 2005-2010. W latach 2010-2015 wprowadzano wersje *Delphi XE1-XE8,* zaś w latach *2015-2017* wersje *Delphi 10.1-10.2.*

Lazarus - Rapid applications development tool and libraries for FPC (*Free Pascal Compiler*) - to zintegrowane środowisko *developer*'skie *IDE*, bazujące na kompilatorze *Free Pascal Open Source Compiler for Pascal and Object Pascal*. Jest wzorowane na wizualnym środowisku *developer*'skim *Borland Delphi* opartym na bibliotece *LCL* (*Lazarus Component Library*), która jest odpowiednikiem biblioteki *VCL*. Program napisany w środowisku *Lazarus* można bez żadnych zmian skompilować dla dowolnego obsługiwanego procesora, systemu operacyjnego i interfejsu okienek. *Lazarus*, w większości przypadków, jest zgodny z *Borland Delphi*. Jest brakującą częścią układanki, która pozwala na rozwijanie programów, podobnie jak w *Delphi*, na wszystkich platformach obsługiwanych przez *FPC*. Jest 32/64-bitowym kompilatorem języka *Pascal*, dostępnym dla wielu różnych platform sprzętowych i systemów

operacyjnych. W odróżnieniu od *Javy*, która stara się, aby raz napisana aplikacja działała wszędzie (*write once run anywhere*), *Lazarus* oraz *Free Pascal* starają się, aby raz napisana aplikacja kompilowała się wszędzie (*write once compile anywhere*). Ponieważ dostępny jest dokładnie taki sam kompilator, w większości przypadków nie trzeba wprowadzać żadnych zmian, aby otrzymać taki sam produkt dla różnych platform. Obecna wersja środowiska *Lazarus* 1.8 dostępna jest na platformy *Windows (*32/64 bit*)*, *Linux, Mac OS X*.

Jedną z najważniejszych zasad w programowaniu zorientowanym obiektowo jest zasada hermetyzacji. Na jej gruncie, programista podczas pisania programu powinien myśleć o kodzie oraz o danych jednocześnie, bowiem dane kierują przepływem wykonywanego kodu, natomiast kod manipuluje postacią i wartościami danych. Według [**3**], 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.

W przypadku konieczności wykonania prostej czynności, jak np. zapamiętania kilka obiektów na dysku, stanowiącej jedną z podstawowych elementów systemu zarządzania obiektowymi bazami danych typu *OO-DBMS* (*Object-Oriented Database Managament System*) pojawia się problem, jak tę czynność wykonać. Można oczywiście odseparować dane od obiektów i zapisać je w pliku na dysku, lecz byłby to krok wstecz. Rozwiązaniem problemu w tym przypadku jest mechanizm strumieni (*streams*). Strumieniem w środowisku *Object Pascal* stanowi kolekcję 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. 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 istnieje 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. Ponadto, nie wiadomo, co zrobić w przypadku, kiedy zaistnieje potrzeba zachowania na dysku obiektów różnych typów.

- *Problem 2*. Brak możliwości użycia pliku do bezpośredniego zapisywania obiektów. *Object Pascal* nie zezwala na utworzenie pliku typu '*object'*, ponieważ zapisywane obiekty mogą zawierać metody wirtualne, których adresy określane są dopiero w czasie wykonania programu (*at Run-time*) na podstawie specjalnej tablicy *VMT* (*Virtual Method Table*) [**4**]. Zachowywanie zatem informacji o adresach tych metod poza programem jest bezcelowe.

Według opracowania [**5**], istnieją trzy zasadnicze podejścia stosowane do zarządzania danymi: (*i*) zarządzanie danymi za pomocą zwykłego pliku, zapewniającego podstawową obsługę danych, (*ii*) zarządzanie danymi w relacyjnych bazach danych, opierające się na teorii relacyjnej [**8**], (*iii*) w podejściu najnowszym, stanowiącym najnowszą implementację technologii *DBMS* dla systemów zarządzania obiektowymi bazami danych, mamy do czynienia z systemami zarządzania danymi w obiektowych bazach danych typu *OO-DBMS*.

Produkty handlowe *OO-DBMS* pojawiły się po raz pierwszy w 1986 roku. Producenci przyjmują jedno z dwóch głównych rozwiązań – (*i*) rozszerzone relacyjne oraz (*ii*) rozszerzony obiektowo zorientowany język programowania typu *OOPL* (*Object Oriented Programming Language*). W pierwszym rozwiązaniu, dostępne na rynku rozszerzone systemy relacyjne wzbogacają system zarządzania relacyjnymi bazami danych, dodając do nich abstrakcyjne typy danych oraz dziedziczenie, które według [**9**] jest mechanizmem do wyrażania podobieństwa pomiędzy obiektami lub inaczej - współdzielenia [**10**] atrybutów oraz metod, jak również kilka usług ogólnego przeznaczenia do tworzenia obiektów i manipulowania nimi. W drugim rozwiązaniu, rozszerzony obiektowo *OOPL* (*Object-Oriented Programming Language*) 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. W przeciwieństwie do tego rozwiązania, rozszerzony *OOPL OO-DBMS* może zapewnić trwałe obiekty (*persistent objects*) [**11**]. Zapisuje on w obiektowej bazie danych dokładnie ten sam obiekt wraz z jego wewnętrznym identyfikatorem, który jest jego adresem w bazie danych, 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.

Mechanizm strumieni posiada szereg zalet, wśród nich najważniejsze to: (1) prostota w użyciu, (2) łatwość w modyfikowaniu własności predefiniowanych typów strumieni, (3) zdolność, dzięki polimorfizmowi, do zapisywania w tym samym pliku różnych typów obiektowych, (4) obsługa operacji we/wy nie na poziomie danych, lecz na poziomie obiektów, (5) duża szybkość w działaniu - dzięki implementacji najbardziej podstawowych metod w języku asemblera, (6) zapewnienie, że obiekt zapisuje się wraz z jego wewnętrznym identyfikatorem, tworząc w ten sposób trwałe obiekty.

Do wad strumieni zaliczamy: (1) konieczność nadzorowania przydzielania unikalnych numerów *ID* nowo definiowanym typom obiektowym, w przeciwieństwie do innych języków np. O_2C, gdzie unikalność zapewniona jest przez system [**9**], (2) brak współdzielonego dostępu do pojedynczego obiektu z serwera obiektów w systemach wielodostępnych, (3) brak zaimplementowanego mechanizmu przetwarzania transakcyjnego, jak również ich współbieżnego wykonywania, koniecznego do zachowania integralności obiektowej bazy danych. Ponadto, f-ma *Borland* nie stworzyła interfejsu użytkownika, pozwalającego na nawigację po wszystkich obiektach kolekcji, w tym rozszerzanie, modyfikację, zmniejszanie oraz zapisywanie ich do strumienia.

W niniejszej pracy zaproponowano metodę, wykorzystującą mechanizm strumieni współpracujący z kolekcją obiektów, umożliwiającą zdefiniowanie podstawowych elementów niezbędnych do budowy interfejsu użytkownika, pozwalających na nawigację po wszystkich obiektach kolekcji oraz wykonanie operacji rozszerzania, modyfikacji, zmniejszania oraz zapisywania ich poprzez system strumieni, jako obiekty trwałe, do obiektowej bazy danych, definiowanej jako kolekcję obiektów [**12**].

2. PODSTAWY MECHANIZMU STRUMIENI

2.1. POLIMORFIZM STRUMIENI

Na podstawowym poziomie o strumieniach można myśleć w podobny sposób, jak o plikach *Pascal*'owych, które są prostymi sekwencyjnymi urządzeniami we/wy. Strumienie natomiast są polimorficznymi sekwencyjnymi urządzeniami we/wy, co oznacza, że zachowują się podobnie do plików sekwencyjnych, z możliwością czytania i zapisywania obiektów różnych typów, których typ nie musi być określony w czasie kompilacji.

2.1.1. Polimorfizm strumieni w środowisku Object Pascal

Tak długo, jak 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 typu *TObject* w środowisku *Object Pacscal*, pokazano 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 *TObject*
List. 1. *TObject* 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 *TObject*
List. 2. Basic methods of the *TObject*

Mechanizm strumieni oparty jest o obiekt typu *TStream*, którego definicję przedstawiono na Wydr. 3. Jak widać, *TStream* jest obiektem potomnym, wywodzącym się z obiektu nadrzędnego *TObject*.

```
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

2.1.2. Polimorfizm strumieni w środowisku Borland Delphi

W przypadku środowiska *Borland Delphi*, zaimplementowano klasę *TPersistent* [**13**], przedstawioną na Rys. 4, z której wyprowadzono wszystkie inne klasy. Jest ona klasą potomną do klasy *TObject*. Obsługuje obiekty trwałe. Zachowuje wartości obiektu do pliku, który ma być później użyty do ponownego utworzenia obiektu w tym samym stanie i z tymi samymi danymi.

W środowisku *Delphi*, w czasie projektowania (*at design time*), obiekty zachowywane są w plikach typu *DFM (Delphi ForM, Delphi Form File)*, a następnie odtwarzane w czasie wykonywania (*at run time*), kiedy tworzony jest specyficzny komponent kontenerowy. Plik *DFM* jest plikiem, którego format opracowała firma *Embarcadero Technologies*. Zawiera binarną reprezentację form utworzonych za pomocą *Form Designer*'a. Służy do przechowywania obiektów i ich atrybutów w nich zawartych, jak również współrzędne kontrolek.

```
{$M+}
TPersistent = class(TObject)
private
  procedure AssignError(Source: TPersistent);
protected
  procedure AssignTo(Dest: TPersistent); virtual;
  procedure DefineProperties(Filer: TFiler); virtual;
  function  GetOwner: TPersistent; dynamic;
public
  destructor Destroy; override;
  procedure Assign(Source: TPersistent); virtual;
  function  GetNamePath: string; dynamic;
end;
```

Wydr. 4. Definicja klasy *TPersistent*
List. 4. *TPersistent* class definition

Wsparcie mechanizmu strumieni nie jest zawarte w klasie *TPersistent* wprost, lecz w klasach potomnych. Obiekty potomne tej klasy mogą zapisywać do strumienia i czytać z niego swoje własności w chwili, kiedy zostały utworzone. Jednym z powodów leżących u podstaw takiego zachowania klasy *TPersistent* jest fakt, że klasa ta została skompilowana z włączoną flagą {$M+}. Ta flaga aktywuje generację rozszerzonej informacji typu *RTTI* (*Run Time Type Information*) dla publicznego fragmentu klasy *TPersistent*. *RTTI* – zawierająca informację o typie klasy w trakcie wykonywania programu - jest techniką stosowaną w nowoczesnych obiektowych językach programowania. Technikę *RTTI* stosuje się w celu umożliwienia tworzenia obiektów, podczas wykonywania programu, na podstawie nazwy klasy tworzonego obiektu. Umożliwia też sprawdzania typu obiektu (przynależności do wybranej klasy), a także zmiany własności obiektów, poprzez modyfikację dostępnych (także poprzez nazwę) własności (*property*). Technika ta polega na dołączeniu do kodu programu informacji o typach klas, czasami też ich własnościach i dostępnych metodach.

Zachowywanie danych obiektów typu *in-memory* w systemie strumieni *Borland Delphi* następuje poprzez listowanie wartości wszystkich własności w publicznej części klasy. Kiedy pewna własność odwołuje się do innego obiektu, *Delphi* zachowuje nazwę tego obiektu lub cały obiekt. Kopiowanie wartości obiektu następuje za pomocą publicznej procedury *Assign*.

2.1.3. Polimorfizm strumieni w środowisku Lazarus

W środowisku *Lazarus*, klasa *TStream* [**14**], pokazana na Rys. 5, jest klasą potomną klasy *TObject*. Jest klasą bazową dla wszystkich klas strumieniowych. Definiuje metody dla czytania i zapisu do strumienia oraz funkcje określające rozmiar strumienia, jak również bieżącego położenia w strumieniu. Potomne klasy, dziedziczące metody z klasy *THandleStream*=class(*TStream*) pokazanej na Rys. 6, takie jak *TMemoryStream* oraz *TFileStream* [**15**], pokazaną na Rys. 7, zmieniają te metody, umożliwiając zapis strumieni do pamięci lub pliku.

type **TStream** = class(TObject)	
protected	
procedure InvalidSeek; virtual;	
procedure Discard();	
procedure DiscardLarge();	
procedure FakeSeekForward();	
function GetPosition; virtual;	
procedure SetPosition(); virtual;	
function GetSize; virtual;	
procedure SetSize64(); virtual;	
procedure SetSize();	Sets the size of the stream
procedure ReadNotImplemented;	
procedure WriteNotImplemented;	
public	
function Read(); virtual;	
function Write(); virtual;	Reads data from the stream to a buffer and returns the number of bytes read.
function Seek();	Writes data from a buffer to the stream and returns the number of bytes written.
procedure ReadBuffer();	Sets the current position in the stream
procedure WriteBuffer();	Reads data from the stream to a buffer
function CopyFrom();	Writes data from the stream to the buffer
function ReadComponent();	Copy data from one stream to another
function ReadComponentRes();	Reads component data from a stream
procedure WriteComponent();	Reads component data and resource header from a stream
procedure WriteComponentRes();	Write component data to the stream
procedure WriteDescendent();	Write resource header and component data to a stream
procedure WriteDescendentRes();	Write component data to a stream, relative to an ancestor
procedure WriteResourceHeader();	Write resource header and component data to a stream, relative to an ancestor
procedure FixupResourceHeader();	Write resource header to the stream
procedure ReadResHeader;	Not implemented in FPC
function ReadByte;	Read a resource header from the stream.
function ReadWord;	Read a byte from the stream and return its value.
function ReadDWord;	Read a word from the stream and return its value.
function ReadQWord;	Read a DWord from the stream and return its value.
function ReadAnsiString;	Read a QWord value from the stream and return its value
procedure WriteByte();	Read an ansistring from the stream and return its value.
procedure WriteWord();	Write a byte to the stream.
procedure WriteDWord();	Write a word to the stream.
procedure WriteQWord();	Write a DWord to the stream.
procedure WriteAnsiString();	Write a QWord value to the stream
Position: Int64;	Write an ansistring to the stream.
Size: Int64;	The current position in the stream.
end;	The current size of the stream.

Wydr. 5. Definicja klasy *TStream* środowisku *Lazarus*
List. 5. *TStream* class definition in *Lazarus* environment

type **THandleStream** = class(TStream)
protected
  procedure SetSize();
public
  constructor Create();
  function Read(); override;
  function Write(); override;
  function Seek(); override;
  Handle: THandle;
end;
	
	Create a handlestream from an OS Handle.
	Overrides standard read method.
	Overrides standard write method.
	Overrides the Seek method.
	The OS handle of the stream.

Wydr. 6. Definicja klasy *THandleStream* środowisku *Lazarus*
List. 6. *THandleStream* class definition in *Lazarus* environment

type **TFileStream** = class(THandleStream)
public
  constructor Create();
  destructor Destroy; override;
  FileName: ;
end;
	
	Creates a file stream.
	Destroys the file stream.
	The filename of the stream.

Wydr. 7. Definicja klasy *TFileStream* środowisku *Lazarus*
List. 7. *TFileStream* class definition in *Lazarus* environment

2.2. REJESTRACJA OBIEKTÓW

Mając na uwadze polimorfizm strumieni, nasuwają się następujące pytania:

1. 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?
2. 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 obiektów. 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.

2.3. INICJALIZACJA STRUMIENIA

W celu efektywnego wykorzystania strumienia w procesie zapisu i odczytu obiektów z pliku, należy dokonać jego inicjalizacji. *TStream* jest obiektem abstrakcyjnym wraz ze swoimi

najbardziej podstawowymi metodami tj. *Get*, *Put* oraz *Error*. Do inicjacji strumienia należy zatem użyć obiektu potomnego. Dla programisty dostępne są następujące potomne strumienie typu: *TMemoryStream*, *TEmsStream* oraz *TDosStream*, z którego wyprowadzono *TBufStream*. Metody *Get*, *Put* z typu obiektowego *TBufStream*, 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. 8.

```
var
  BufDosStream: TBufStream;

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

Wydr. 8. Inicjalizacja buforowanego strumienia *TBufStream*
List. 8. *TBufStream* buffered stream initialization

2.4. ZAPISYWANIE I CZYTANIE OBIEKTÓW ZE 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 instancji (wystąpień), zawierających lub dziedziczących metody wirtualne, tzn. jedna tablica *VMT* na jeden typ obiektu. Zdefiniujmy przykładowy, przedstawiony na Wydr. 9., typ obiektowy *TMojObject*, pochodzący od podstawowego typu obiektowego *TObject*, o którym zakłada się, że był wcześniej zarejestrowany, natomiast *BufDosStream* jest zainicjowanym strumieniem (obiektem) pochodzącym od typu obiektowego *TStream*.

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

Wydr. 9. Definicja obiektu *TMoj Object*
List. 9. *TMojObject* object definition

Przyjrzyjmy się bliżej metodzie *Put*, ze strumienia *BufDosStream*, której składnia jest następująca: *BufDosStream.Put(PMojObiekt)*. Strumień *BufDosStream*, bazując na informacji odczytanej z tablicy *VMT* skojarzonej z obiektem *PMojObiekt*, jest w stanie określić: (1) jakiego typu jest to obiekt, (2) jaki jest jego numer identyfikacyjny (*ID*), (3) wielkość obiektu w bajtach (dane oraz kod), (4) 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 [**3,8**], 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*), wszystkie obiekty podrzędne zostaną zapisane do pliku automatycznie. Kiedy program dokona odczytu takiego złożonego obiektu 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*).

2.5. TWORZENIE OBIEKTÓW WSPÓŁPRACUJĄCYCH ZE STRUMIENIAMI

Wszystkie standardowe obiekty zdefiniowane w środowisku *Object Pascal*, 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 trzeba 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 zapisywać i odczytywać ze strumienia dodatkowo dodane pola. Pozostałe pola, odziedziczone po swoim przodku, obsługiwane są automatycznie dzięki odziedziczonym metodom.

Przykładowo, załóżmy, że wyprowadzono nowy obiekt *TMyWindow*, przedstawiony na Wydr. 10., z obiektu *TWindow*, do którego dodano nowe pole, np. *field1*. Na Wydr. 11, 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*.

```
Type

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

Wydr. 10. Definicja obiektu *TMyWindow w* z dodatkowym polem

List. 10. *TMyWindow* object definition with additional field

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

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

Wydr. 11. Procedura zapisywania i odczytywania obiektu z dodatkowym polem

List. 11. The object save and read procedures with an additional field

W końcu, należy poinformować system strumieni o nowym obiekcie poprzez: (*i*) zdefiniowanie rekordu rejestracyjnego strumieni typu *TStreamRec* dla tego obiektu, który pokazano na Wydr. 12., (*ii*) przekazanie go do procedury *RegisterType*, którą pokazano na Wydr. 13. W omawianym przykładzie można, np. zdefiniować rekord rejestracyjny *RMyWindow* oraz wykonać procedurę *RegisterType*. Od tej pory, po wykonaniu powyższych czynności, instancje nowo zdefiniowanego obiektu *TMyWindow* można zapisywać i odczytywać za pomocą standardowych strumieni.

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

Wydr. 12. Rekord rejestracyjny strumieni typu *TStreamRec*

List. 12. Registration record of streams of the type *TStreamRec*

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

RegisterType( RMyWindow );
```

Wydr. 13. Rekord rejestracyjny *RMyWindow* oraz procedura *RegisterType*

List. 13. 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.

2.5.1. REJESTRACJA REKORDÓW REJESTRACYJNYCH W STRUMIENIACH

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. 14 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. 14. Implementacja procedury *RegistrerType* w języku assembler
List. 14. Implementation of the *RegistrerType* procedure in the assembler language

2.6. MIEJSCE STRUMIENI NA TLE OBIEKTOWEGO SYSTEMU ZARZĄDZANIA DANYCH

W artykule [**9**] 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 obiektowego systemu zarządzania danych. W stanie obecnym, ze względu na brak chociażby takich cech jak: (1) języka manipulowania danymi w bazie danych, (2) mechanizmów transakcji, jak również ich współbieżnego wykonywania, (3) języka zapytań, mechanizm strumieni może być wykorzystany przez programistę do tworzenia prostych obiektowych baz danych oraz podstawowej ich obsługi.

3. STRUMIENIE – 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, zwane polem tablicy metod wirtualnych *VMTF* (*Virtual Method Table Field*), które używane jest do zapamiętania *offset*'u obiektowej tablicy *VMT* w segmencie danych *DS* (*Data Segment*) 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* 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. 15, a na Rys. 1 zarys ich 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*.

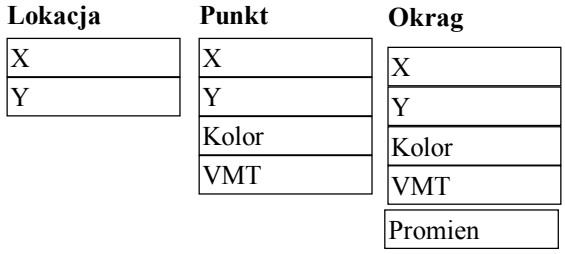
```
type

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.15. Deklaracje typów obiektowych *Lokacja*, *Punkt*, *Okrag*
List. 15. 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 polimorfizmu, za pomocą którego możliwe jest dokonywanie pewnych uogólnień. Polimorfizm z języka greckiego to „wiele postaci”, i według [**4**] 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* programu, zawiera wszystkie typy stałych z następującymi po nich wszystkimi zmiennymi globalnymi. Rejestr *DS* programu nigdy nie jest zmieniany podczas wykonywania programu.

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 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 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 do tego celu rozszerzonej składni standardowych procedur *New* oraz *Dispose*.

Drugi element tej tablicy, również o długości jednego słowa, zawiera ujemną liczbę określająca rozmiar instancji stowarzyszonej z właściwym dla niej typem obiektowym. Ta informacja wykorzystywana jest przez metody wirtualne wywołujące mechanizm sprawdzający dane *VM* (*Validation Mechanism*), w celu wykrycia niezainicjowanych obiektów, tj. instancji 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 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. 1. 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 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* instancji obiektu typu *PPunkt* zdefiniowanego w par.3.1, wówczas wywołanie metody wirtualnej *PP^.Show* tego obiektu generuje kod w języku assembler, pokazany na Wydr. 16.

```
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. 16. Fragment kodu metody wirtualnej *PP^.Show* napisany w języku assembler
List. 16. 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, który w 32-bit maszynach jest jego lokacją w głównej pamięci [**12**], przez którą ta metoda jest wywoływana. Inaczej mówiąc, reprezentuje instancję obiektu. Przy powrocie z ciała tej 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 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* instancji, która ma być zwolniona. 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 I DESTRUKTORÓW OBIEKTÓW

W przypadku wywoływania konstruktorów, parametr zawierający offset do tablicy *VMT* zapamiętywany jest w elemencie tej tablicy, tj. w polu *Self*. Ponadto, w sytuacji kiedy kon-

struktor 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*. 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 firmę *Borland*, pokazano na Wydr. 14. 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 *Borland* pokazano na Wydr. 15.

```
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. 17. Metoda wirtualna *TStream.Put* napisana w języku assembler

List.17. Virtual method *TStream.Put* written in assembler language

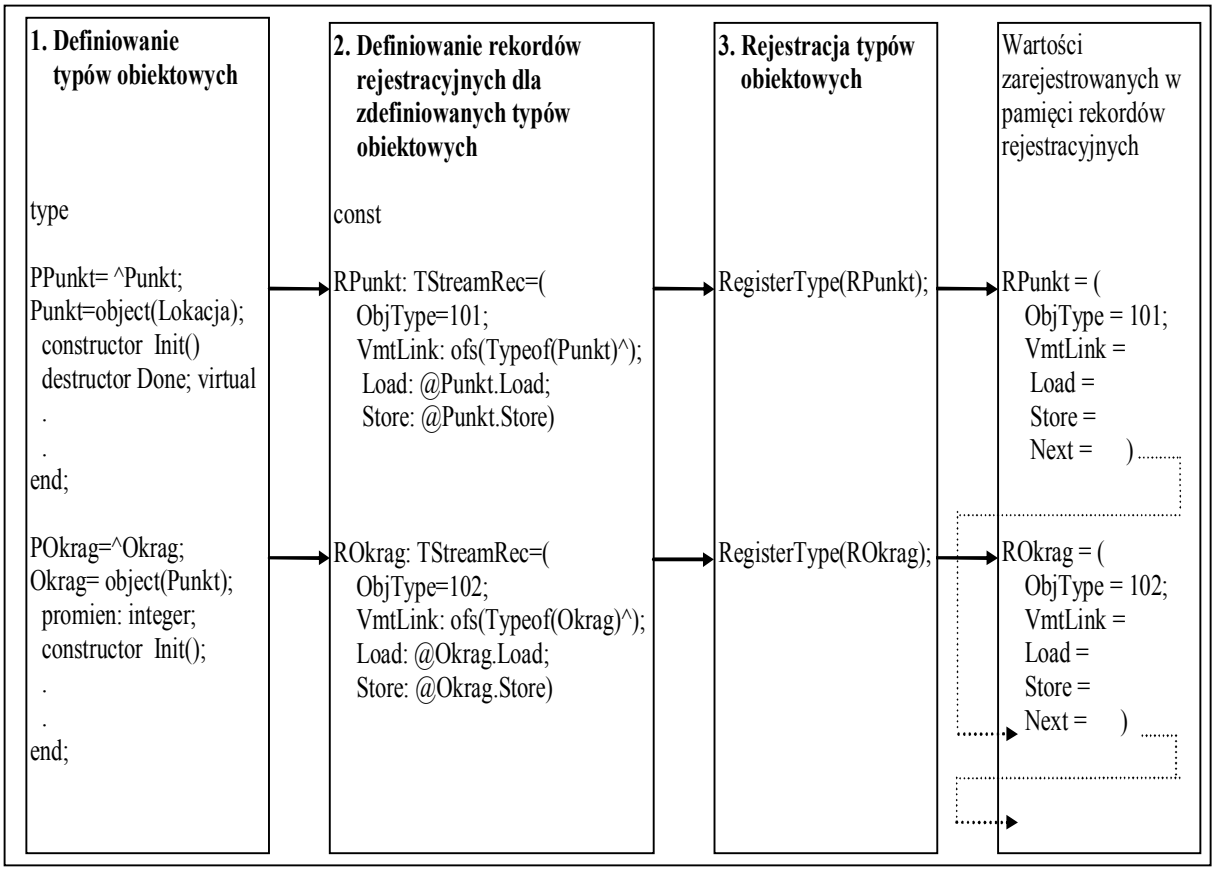
```
function TStream.Get: PObject; assembler;
asm
        PUSH    AX
        MOV     AX,SP
        PUSH    SS
        PUSH    AX
        MOV     AX,2
        PUSH    AX
        LES     DI,Self
        PUSH    ES
        PUSH    DI
        MOV     DI,ES:[DI]
        CALL    DWORD PTR [DI].TStream_Read
        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. 18. Metoda wirtualna *Stream.Get* napisana w języku assembler

List.18. Virtual method *TStream.Put* writen in assembler language

3.5. PROCES ZAPISU I ODCZYTU OBIEKTÓW

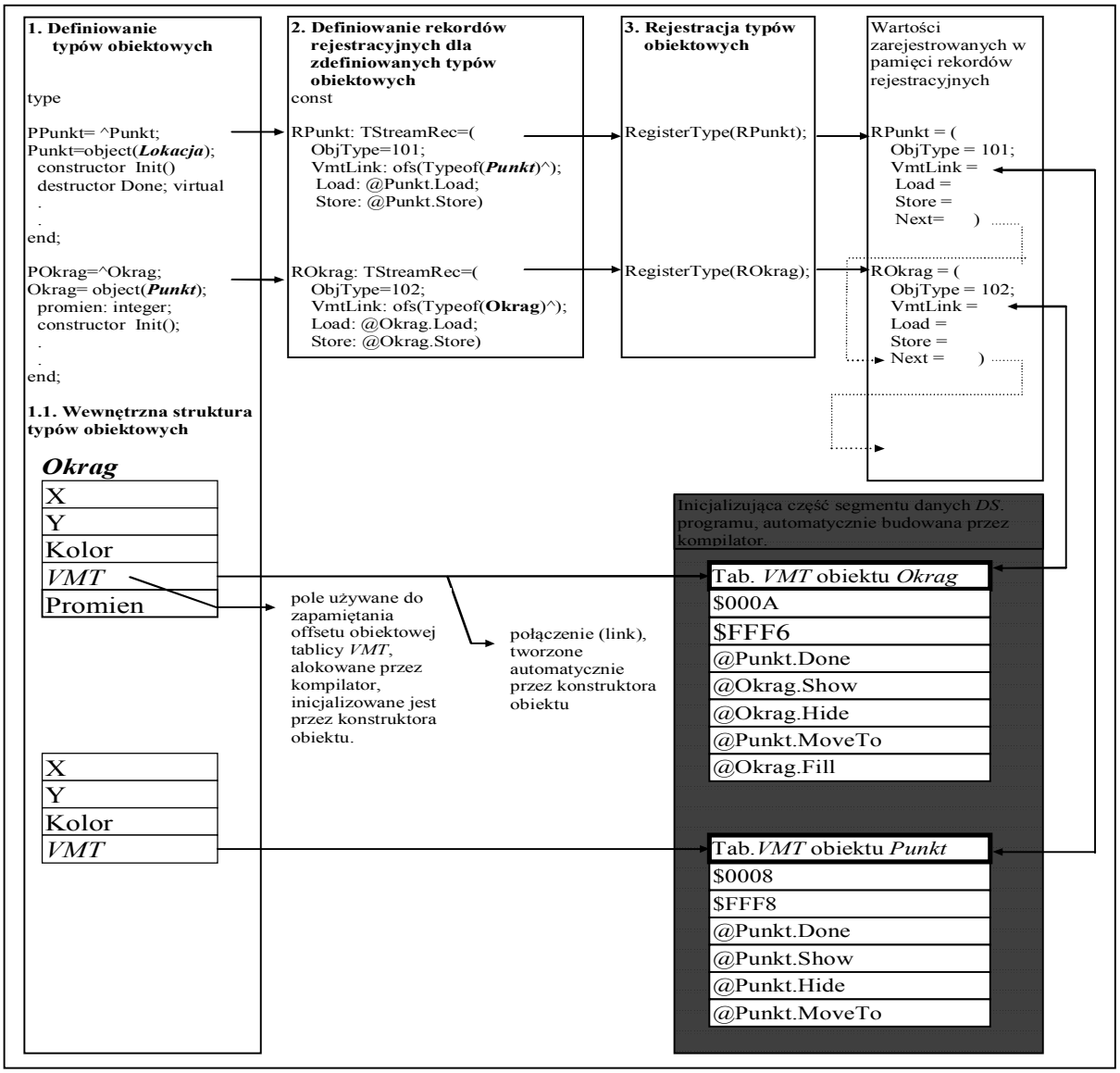
W celu ilustracji wcześniej omawianych procesów rejestracji, zapisu i odczytu obiektów, poniżej na Rys. 2 przedstawiono je w graficznej formie, biorąc do rozważań zdefiniowane w paragrafie 3.1 przykładowe deklaracje typów obiektowych *Punkt* oraz *Okrag*.



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

3.5.2. WZAJEMNE RELACJE POMIĘDZY OBIEKTAMI

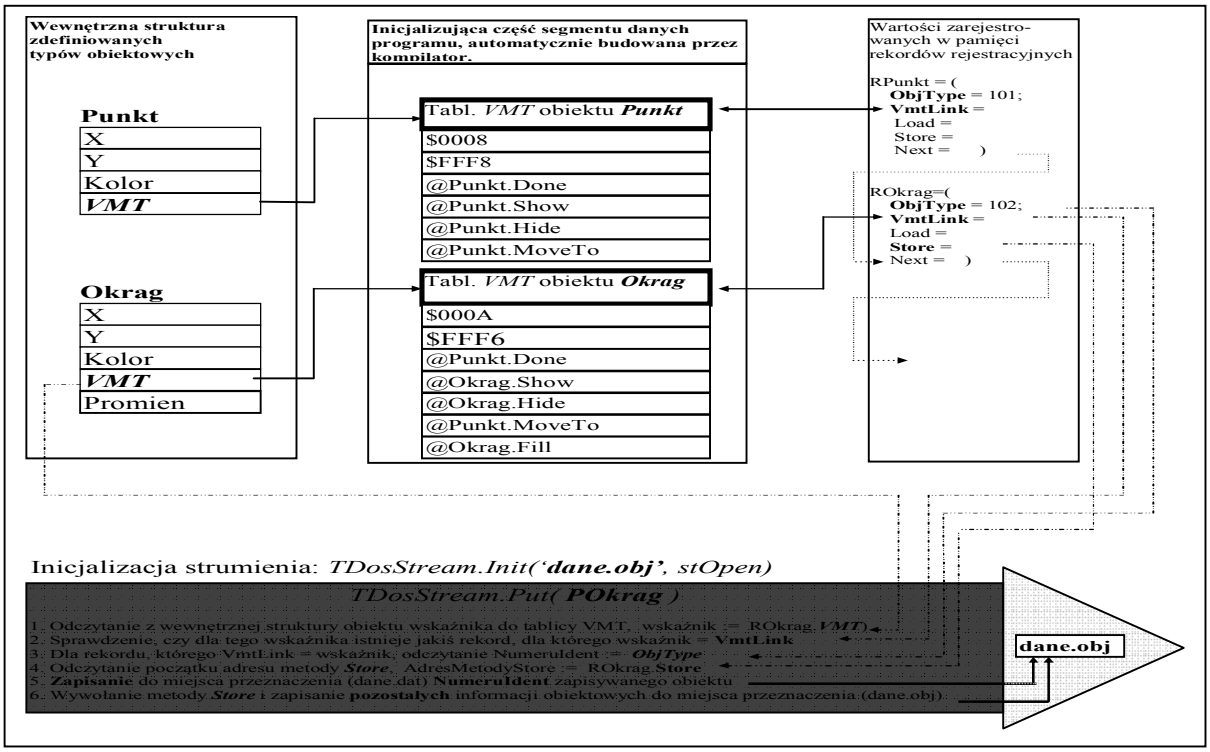
Wzajemne relacje pomiędzy obiektami, ich tablicami *VMT* oraz rekordami rejestracyjnymi dla przykładowych deklaracji typów obiektowych *Punkt* oraz *Okrag*, pokazano na Rys. 3.



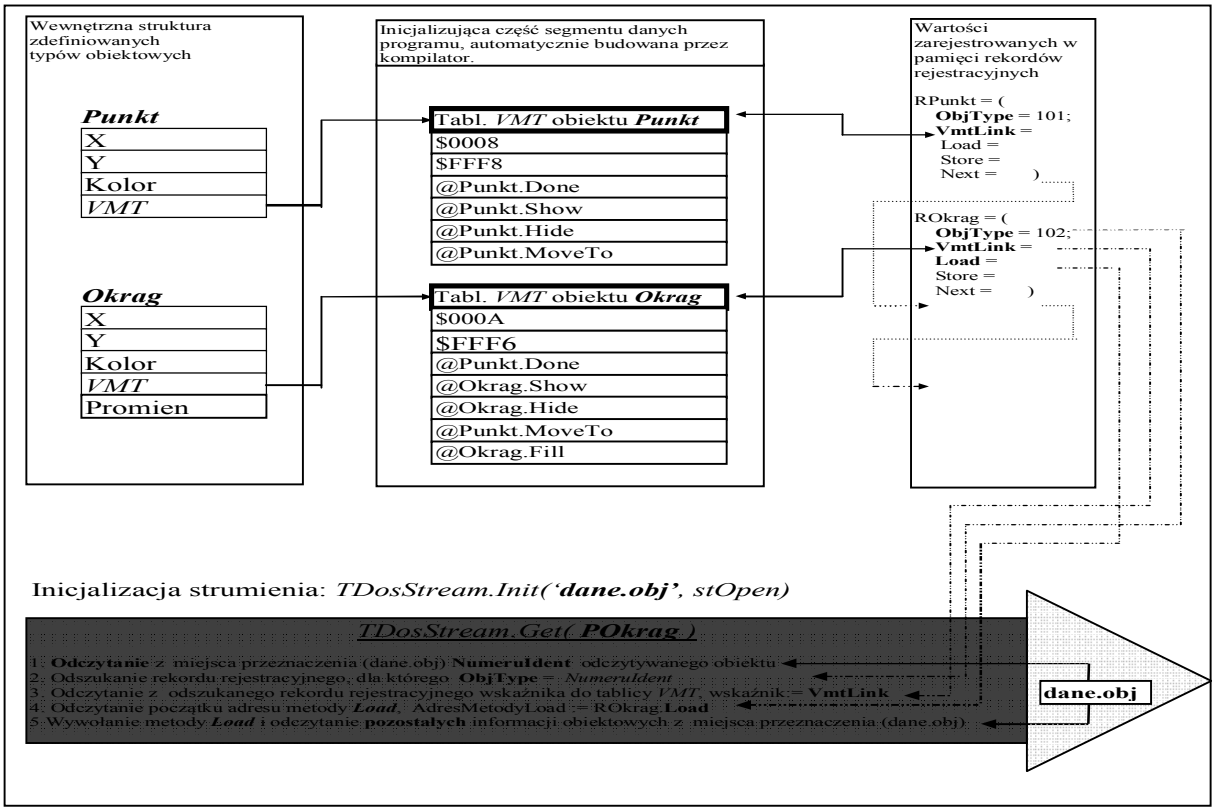
Rys. 3. Relacje pomiędzy obiektami, ich tablicami *VMT* oraz rekordami rejestracyjnymi
Fig. 3. Correlation relationships between objects, their *VMT* tables and registration records

3.5.3. PROCES ZAPISU I ODCZYTU OBIEKTÓW

Poniżej, dla przykładowych deklaracji typów obiektowych *Punkt* oraz *Okrag*, na Rys. 4 przedstawiono ilustrację procesu zapisu obiektu *Okrag* za pomocą metody *PUT* do strumienia (do miejsca przeznaczenia - plik o nazwie dane.obj). Na Rys. 5 zilustrowano proces odczytu obiektu *Okrag* za pomocą metody *GET* ze strumienia (z pliku o nazwie dane.obj).



Rys. 4. Proces zapisu obiektów do strumienia
Fig. 4. Write object process to the stream



Rys. 5. Proces odczytu obiektów ze strumienia
Fig. 5. Read object process from the stream.

4. WYMIANA DANYCH ZA POMOCĄ STRUMIENI POMIĘDZY OBIEKTOWĄ APLIKACJĄ I BAZĄ OBIEKTOWĄ

4.1. ZARZĄDZANIE DANYMI - KOLEKCJE

Środowisko *Object Pascal* oferuje do zarządzania danymi elastyczne i o dużych możliwościach narzędzie, w postaci typu obiektowego *TCollection*, zwanego kolekcją. Kolekcja podobna jest w swej istocie do rozszerzalnej tablicy wskaźników, wskazujących na obiekty.

Za pomocą kolekcji, w aplikacji zorientowanej obiektowo, można dokonywać następujących operacji: (1) odczytywanie obiektów ze strumienia, (2) wyświetlanie danych z poszczególnych obiektów, (3) prowadzenie nawigacji pomiędzy obiektami, (4) dodawanie i kasowanie obiektów, (5) zapisywanie obiektów do strumienia.

4.2. ZARZĄDZANIE DANYMI W BAZIE OBIEKTOWEJ

Jak wspomniano już we wprowadzeniu, w niniejszej pracy zaproponowano metodę, wykorzystującą mechanizm strumieni współpracujący z kolekcją obiektów, umożliwiającą zdefiniowanie podstawowych elementów niezbędnych do budowy interfejsu użytkownika, pozwalające na nawigację po wszystkich obiektach kolekcji oraz wykonanie operacji bazodanowych polegających na rozszerzaniu, modyfikacji, zmniejszaniu oraz zapisywaniu obiektów kolekcji, poprzez system strumieni, jako obiekty trwałe, do obiektowej bazy danych, definiowanej jako kolekcję obiektów.

Istotą zaproponowanej metody jest zdefiniowanie obiektu pośredniczącego, pochodzącego od typu obiektowego *TObject*, umożliwiający efektywnego użycia kolekcji w systemie strumieni. Obiekt pośredniczący musi posiadać następujące elementy: (1) pole lub pola zawierające dane, (2) zdefiniowaną metodę *Store* do zapisywania danych obiektu w strumieniu, (3) zdefiniowaną metodę *Load* do odczytywania danych obiektu ze strumienia, (4) rekord rejestracyjny.

Wymieniony powyżej interfejs użytkownika, w postaci okna dialogowego, jest istotnym elementem pozwalającym na efektywne wykonywanie w/w operacji. Okno dialogowe tego interfejsu powinno zawierać formularz z pewną ilością pól informacyjnych, odpowiadających - ściśle według kolejności wyświetlania - polom rekordu transferowego, pełniącego rolę bufora pomiędzy interfejsem użytkownika, a kolekcją obiektów.

Ilustrację powyższych wymagań pokazano na Wydr. 19, z deklaracją obiektu pośredniczącego typu *TTransferObject*, wyprowadzonego z typu obiektowego *TObject*, wraz z rekordem transferowym. Na jego podstawie wyprowadzono przykładowy typ obiektowy

TKadryObj, który użyto do dalszych rozważań zmierzających do zachowywania przykładowych obiektów kadrowych. W tym przykładzie, rekord transferowy jest typem pewnego typu rekordu kadrowego zdefiniowanego w aplikacji użytkowej, służącego do wymiany informacji pomiędzy interfejsem użytkownika a kolekcją obiektów zapisywanych lub odczytywanych ze strumienia.

```
Type
PTransferObject = ^TransferObject;
TransferObject = object(TObject)
     RekordTransferowy: TRekordKadrowy;
     constructor Init;
     constructor Load (var S: TStream);
     procedure Store (var S: TSstream);
end;

PKadryObj = ^TKadryObj;
TKadryObj = object(TTransferObject)
constructor Init;
constructor Load (var S: TStream);
procedure Store (var S: TSstream);
end;

constructor TKadryObj.Init;
begin
  inherited Init;
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 );
```

Wydr. 19. Deklaracja obiektu pośredniczącego *TransferObject* z rekordem transferowym
List.19. Declaration of the intermediary object *TransferObject* with transfer record

4.2.1. CZYTANIE KOLEKCJI ZE STRUMIENIA

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

Poniżej na Wydr. 20, przedstawiono przykład, za pomocą którego można dokonać zapisu do kolekcji, przykładowych danych osobowych, zawartych w obiektach, do pliku o nazwie „*KADRY.OBJ*". Na Wydr. 21. pokazano zaś przykład odczytu do kolekcji obiektów zawierających dane osobowe.

```
procedure SaveKadry;
var
  StrumienPlikuKadrowego: TBufStream;
  KadrowaKolekcjaObiektow: PCollection;
begin
 StrumienPlikuKadrowego.Init('KADRY.OBJ',stOpenWrite,1024);
 StrumienPlikuKadrowego.Put(KadrowaKolekcjaObiektow);
 StrumienPlikuKadrowego.Done;
end;
```

Wydr. 20. Zapis do kolekcji obiektowych danych kadrowych
List.20. Write to collection personnel object's data

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

Wydr. 21. Odczyt do kolekcji obiektowych danych kadrowych
List. 21. Read to collection personnel object's data

Przykłady te ilustrują korzyści, jakie płyną z faktu gromadzenia danych w postaci obiektów w kolekcji. Pojedynczą instrukcją zapisuje się lub odczytuje wszystkie obiekty z kolekcji.

4.2.2. MANIPULOWANIE ELEMENTAMI KOLEKCJI

W sytuacji, kiedy kolekcja obiektów została załadowana do pamięci operacyjnej, wówczas za pomocą przykładowego obiektu pośredniczącego typu *TKadryObj*, omawianego w punkcie 4.2, możliwa jest współpraca z interfejsem użytkownika, zdefiniowanego w postaci przykładowego obiektu *TInterfejsUzytkownika*, wyprowadzonego z obiektu *TWindow*, umożliwiającego: rozszerzanie, modyfikacja, zmniejszanie oraz zapisywanie ich do strumienia, a w konsekwencji do bazy obiektowej.

```
Type
PInterfejsUzytkownika = ^TInterfejsUzytkownika;
TInterfejsUzytkownika = object(TWindow);
  constructor Init(AParent: PWindowsObject; ATitle: PChar);
  procedure OdczytajDaneObiektowe(var IndeksObiektu: integer);
  procedure ZachowajDaneObiektowe (var IndeksObiektu: integer);
  procedure DodajObiekt(var IndeksObiektu: integer);
  procedure KasujObiekt (var IndeksObiektu: integer);
  procedure SzukajObiekt(var IndeksObiektu: integer);
end;
```

Wydr. 22. Przykład definicja interfejsu użytkownika
List. 22. Example definition of the user interface

4.2.2.1. Zapisywanie i odczytywanie danych z kolekcji

Poniższy przykład, przedstawiony na Wydr. 23, pokazuje, w jaki sposób zapisać przykładowe informacje osobowe zawarte w obiekcie pośredniczącym, za pomocą rekordu transferowego, a następnie zapis do pliku.

```
procedure TInterfejsUzytkownika.ZachowajDaneObiektowe(var IndeksObiektu: integer);
begin
   PKadryObj(KadrowaKolekcjaObiektow ^.At(IndeksObiektu).RekordTransferowy) := Dane_o_Pracowniku;
  SaveKadry;
end:
```

Wydr. 23. Zapisanie danych, poprzez rekord transferowy, do obiektu kolekcji
List. 23. Writing data, with the transfer record, to the object of the collection

W podobny sposób, za pomocą rekordu transferowego, można odczytać informacje osobowe, przedstawione na Wydr. 24, z pierwszego obiektu kolekcji:

```
constructor TInterfejsUzytkownika.OdczytajDaneObiektowe(var IndeksObiektu: integer);
begin
   Dane_o_Pracowniku := PKadryObj(KadrowaKolekcjaObiektow ^.At(IndeksObiektu).RekordTransferowy);
end;
```

Wydr. 24. Odczytanie danych, za pomocą rekordu transferowego, z obiektu kolekcji
List. 24. Reading data, with the transfer record from the object of the collection

4.2.2.2. Dodawanie i kasowanie obiektów z kolekcji

Dodanie nowego obiektu kadrowego do obiektowej bazy danych pokazano na Wydr. 25, natomiast kasowanie obiektu z bazy danych na Wydr. 26.

```
procedure TInterfejsUzytkownika.DodajObiekt(var IndeksObiektu: integer);;
begin
  TKadryObj^.RekordTransferowy := Dane_o_Pracowniku;
  KadrowaKolekcjaObiektow ^.AtInsert(IndeksObiektu, ^TKadryObj);
  SaveKadry;
end;
```

Wydr. 25. Dodanie nowego obiektu do obiektowej bazy danych
List. 25. Adding the new object to database

```
procedure TInterfejsUzytkownika.SkasujObiekt(var IndeksObiektu: integer);
begin
  KadrowaKolekcjaObiektow ^.AtInsert(IndeksObiektu, ^TKadryObj);
  SaveKadry;
end:
```

Wydr. 26. Kasowanie obiektu w obiektowej bazie danych
List. 26. Adding the new object to database

5. PODSUMOWANIE

W pracy przedstawioną dogłębną analizę, zaimplementowanego w środowisku *Object Pascal*, mechanizmu strumieni. Mechanizm strumieni, dodatkowo wzbogacony o składnię i możliwości do zarządzania obiektami, pozwala na wykonanie operacji bazodanowych polegających na rozszerzaniu, modyfikacji, zmniejszaniu oraz zapisywaniu obiektów kolekcji, jako obiekty trwałe, do obiektowej bazy danych definiowanej jako kolekcję obiektów. Jest bardzo efektywnym narzędziem dla tworzenia i obsługi obiektów zachowanych w obiektowej bazie danych.

W pracy zaproponowano metodę, wykorzystującą mechanizm strumieni współpracujący z kolekcją obiektów, umożliwiającą zdefiniowanie podstawowych elementów niezbędnych do budowy interfejsu użytkownika. Istotą zaproponowanej metody jest zdefiniowanie obiektu

pośredniczącego, pochodzącego od typu obiektowego *TObject*, umożliwiający efektywne użycia kolekcji w systemie strumieni. Dzięki temu, że obiekt pośredniczący posiada: (1) pole lub pola zawierające dane, (2) zdefiniowaną metodę *Store* do zapisywania danych obiektu w strumieniu, (3) zdefiniowaną metodę *Load* do odczytywania danych obiektu ze strumienia, (4) rekord rejestracyjny, możliwe jest zarządzanie danymi w obiektowej bazie danych.

W zaproponowanej metodzie oparto się na mechanizmie strumieni posiadający szereg zalet. Wśród nich najważniejsze to: (1) prostota w użyciu, (2) łatwość w modyfikowaniu własności predefiniowanych typów strumieni, (3) zdolność, dzięki polimorfizmowi, do zapisywania w tym samym pliku różnych typów obiektowych, (4) obsługa operacji we/wy nie na poziomie danych, lecz na poziomie obiektów, (5) duża szybkość w działaniu - dzięki implementacji najbardziej podstawowych metod w języku asemblera, (6) zapewnienie, że obiekt zapisuje się wraz z jego wewnętrznym identyfikatorem, tworząc w ten sposób trwałe obiekty.

Niestety, po stronie utrudnień dla programisty, 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.

W pracy, na osnowie tej metody, zaproponowano kilka podstawowych przykładowych elementów niezbędnych do budowy interfejsu użytkownika, pozwalające na wykonanie podstawowych operacji bazodanowych polegających na rozszerzaniu, modyfikacji, zmniejszaniu oraz zapisywaniu obiektów kolekcji, poprzez system strumieni, jako obiekty trwałe, do obiektowej bazy danych, definiowanej jako kolekcję obiektów.

LITERATURA

1. Dumnicki R., Kasprzyk A., Kozłowski M.:Analiza i projektowanie obiektowe, ISBN: 83-7197-052-8, Helion, 1998.
2. Rumbaugh J., Blacha M., Premerlani W. Eddy F., Lorensen W.: Object-Oriented Modelling and Design, Prentice-Hall, Inc. ISBN: 0-13-630054-5, 1991.
3. Borland International „Users Guide”.
4. Borland International „Programming Guide”.

5. Pacheco X.& Teixeira S. „Delphi2 - Developher's Guide", SAMS, ISBN:0-672-30914-9, Publishing, printed in the USA, 1996.
6. Misiurewicz P. „Układy mikroprocesorowe", ISBN 83-204-0541-6, WNT 1983.
7. Bielecki J. „Turbo Pascal", ISBN 83-206-0903-8, WKŁ 1989.
8. Date C. J.: Database Design and Relational Theory, 2012.
9. Coad P., Yourdon E. „Object-Oriented Design", ISBN 83-85769-18-8, PRENTICE Hall, Inc., 1991.
10. Augustyn D., Stąpor K., „Obiektowo Zorientowany System Zarządzania Bazą Danych O2", Zeszyty Naukowe Politechniki Śląskiej, INFORMATYKA z.31, Gliwice, 1996.
11. Mary E.S. Loomis, Ph.D.: Object Databases: The Essentials, Corporate & Professional Publishing Group, ISBN: 0-201-56341-X, 1995.
12. Eaglestone B., Ridley M.: Object Databases: An Introduction, McGraw-Hill Book Company, 1998.
13. The TPersistent Class, Copyright eTutorials.org 2008-2017, http://etutorials.org/Programming/mastering+delphi+7/Part+I+Foundations/Chapter+4+Core+Library+Classes/The+TPersistent+Class/.
14. Tstream, http://lazarus-ccr.sourceforge.net/docs/rtl/classes/tstream.html.
15. TFileStream, http://lazarus-ccr.sourceforge.net/docs/rtl/classes/tfilestream.html.

Recenzent:

Wpłynęło do Redakcji 30 lutego 2018 r.

Abstract

In this article a method of the application was presented implement in the Object Pascal environment system of streams, and cooperating with the collection of objects, for construction of the object database. In particular the mechanisms characteristic of this implementation of the registration of object types were presented in the system of streams, as well as storing them and of reading out. Elements of the user interface, enabling the navigation after all objects of the collection, widening, alteration, reducing and writing them were proposed, through the stream system to the object database in the form of persistent objects. Implemented in the Object Pascal environment mechanism of streams, additionally made rich for syntax and possibilities for managing the reading and with remembering objects in the

object database, is a very effective and comfortable tool of the programmer for creating and the service of object databases.

The mechanism of streams has a number of virtues, among them most important it:

- simplicity in the use,
- easiness of modifying the property of predefined types of streams,
- gifted, thanks to the polymorphism, for saving in the same file different object types,
- is supporting operations you not on the level of data, but on the level of objects,
- high speed in action - thanks to the implementation the most of basic methods in the assembly language,
- is ensuring persistent objects - the object is saved along with his internal identifier.

However to rank among defects it is possible:

- need to supervise allotting unique numbers of Id's to newly defined object types, in the opposite of other language, e.g. O_2C, where the uniqueness is ensured by the system,
- is lacking the shared access to the single object from the server of objects in multi access systems,
- lack implement of mechanism of transaction processing, as well as their concurrent performance, necessary to keep the integrity of the object database.

At the work, on the basis of this method, a few basic model essential elements were suggested for construction of the user interface, letting the collection for performing basic operations of database objects relying on widening, alteration, reducing and writing, through the system of streams, as long-lasting objects, to the object database, defined as the collection of objects.