Programovací jazyk Oberon a jeho objektově orientované vlastnosti

Andrea Němcová, Jan M. Honzík

Motto: "Když jsem se narodil víly mi určily budoucnost. Navždy budu skřítkem, budu nejhezčím tvorem na zemi, budu umět číst v lidských srdcích i myslich, všichni ptáci a zvířata mně budou poslouchat a budu se moci přemístit na jakékoliv místo na Zemi." řekl Oberon

Legenda o Oberonovi

1. Úvod

Příspěvek je přehledem objektově orientovaných vlastností jazyka Oberon. Oberon je nejen programovací jazyk, ale i operační systém. Byl navržen N. Wirthem a J. Gutknechtem (1991). Je poslední na Wirthově vývojové linii jazyků Pascal a Modula-2.

Hlavními rysy jazyka jsou moduly (které jsou samostatně kompilovány), silná typová kontrola (přes hranice modulů) a typová rozšiřitelnost.

V Oberonu je objektová orientace dosažena pomocí použití typové rozšiřitelnosti a procedurálním proměnným. Klíčovými pojmy objektové orientace v jazyku Oberon jsou:
- generický modul,
- heterogenní datové struktury,
- objekt a jeho reprezentace,
- dynamická vazba procedur,
- struktura modulů objektově orientovaného programu.

V příspěvku budou vysvětleny jednotlivé výše uvedené pojmy a na příkladech názorně předvedeny jednotlivé vlastnosti.

2. Objektové vlastnosti Oberonu

2.1 Generický modul

Je první v hierarchii modulů, které se definují při výstavbě programu. Měl by obsahovat základní (bázové) typy a procedury, které s ním pracují. Jeho návrh by měl být udělán tak, aby se nemusel znovu kompilovat při malých změnách v programu.

Výstavbu generického modulu si ukážeme na příkladu fronty FIFO. Tato fronta nás bude provázet částí článku a na ní budou prezentovány vlastnosti jazyka Oberon.
Definice fronty může být následující:

```
MODULE FIFOs;
TYPE
  Node*=POINTER TO NodeDesc;
```

```
  NodeDesc*=RECORD next:Node END;
  FIFO*=RECORD first,last:Node END;

 PROCEDURE Open*(VAR q:FIFO);
 BEGIN ... END;
 PROCEDURE Enqueue*(VAR q:FIFO; n:Node);
 BEGIN ... END;
 PROCEDURE DequeueNode*(VAR q:FIFO):Node;
 BEGIN ... END;
 END FIFOs;
```

Možná implementace některé procedury je taková:

```
 PROCEDURE Open*(VAR q:FIFO);
 BEGIN q.first := NIL;
 END Open;
```

Typová rozšiřitelnost umožňuje, aby modul FIFOs byl použitelný v mnoha dalších aplikacích, které potřebují FIFO frontu. Aplikačně specifická data jsou přidávána až v modulu **klienta** (modul, který importuje modul FIFOs). Při této definici modulu FIFOs je takový modul nazýván **generickým modulem**.

Klientský modul může být definován takto:

```
(* Je to simulační program, kde zákazníci jsou ve frontě *)

 MODULE Sim;
 IMPORT FIFOs, ...;
 TYPE
  Customer*=POINTER TO CustomerDesc;
  CustomerDesc*=RECORD(FIFOs.NodeDesc)
   priority:INTEGER;
  END;
 ...
 VAR c:Customer; q:FIFOs.FIFO; temp:FIFOs.Node;
 ...
 END Sim;
```

Nový zákazník c je vytvořen a vložen do fronty pomocí následujících příkazů:

```
 NEW(c); c.priority:=0 ; FIFOs.Enqueue(q,c); ...
```

Je nutné, aby c byla proměnná typu **Customer**, protože pak tato proměnná může být uchována ve frontě. Následující příkazová sekvence získá zákazníka c čekajícího ve frontě q:

```
 temp:=FIFOs.DequeueNode(q);
 IF (temp#NIL)&(temp IS Customer)
  c:=temp(Customer); ...
 END;
```

Pro potřebu práce s procedurou FIFOs.DequeueNode musí být proměnná **temp** typu FIFOs.Node. Pro přiřazení do proměnné c však musí být výsledný typ procedury přetypován na typ **Customer**. Tato operace se provádí pomocí kulatých závorek, jak je vidět na předcházejícím příkladě.

Generické moduly mohou být využívány mnoha klienty, které obvykle použijí typovou rozšiřitelnost na definované bázové typy.

2.2 Heterogenní datové struktury

Dynamické datové struktury se většinou skládají z elementů a odkazy na ně jsou pomocí ukazatelů. Ukazatel odkazuje na uzlový typ, a proto se celá datová struktura skládá z uzlů stejného typu. V mnoha případech je toto neakceptovatelné omezení.

Typová rozšiřitelnost je prostředkem pro stavbu heterogenních dynamických struktur. Tyto struktury se skládají z různých (ale svým způsobem kompatibilních) uzlových typů. Klíčovou myšlenkou je deklarace základního společného typu, který obsahuje společná data všech nadstavbových uzlů. Privátní data jsou obsažena v jednotlivých rozšiřujících typech.

Předvedeme to na modulu **Sim**, který mimo jiné může obsahovat i následující deklarace. Například:

```
Customer1=POINTER TO C1D;        Customer2=POINTER TO C2D;
C1D=                             C2D=
RECORD (FIFOs.NodeDesc)          RECORD (FIFOs.NodeDesc)
  priority:INTEGER                 timeStamp:REAL;
END;                               workDemand:REAL
                                 END;
```

Oba dva typy **Customer1** a **Customer2** jsou rozšířením typu **FIFOs.Node**, proto mohou být vloženy do stejné FIFO fronty. Je nutné, aby dynamický typ uzlu nebyl ztracen a typový test odhalil dynamický typ uzlu, což pak umožňuje použít typově specifické vlastnosti jednotlivých zákazníků.

V dalších příkladech se budeme zabývat seznamem grafických objektů užívaných v grafických editorech. Editor užívá lineární seznam pro uchování objektů zobrazovaných na ploše. Typickými položkami takového seznamu jsou čáry, obdélníky, kruhy, elipsy a texty. Popis obrazců založíme na typu **Figure**, který obsahuje jen strukturovanou informaci:

```
MODULE Graphics;
TYPE
  Figure*=POINTER TO FigureDesc;
  FigureDesc*=RECORD
    next:Figure
  END;
...
END Graphics;
```

Každá specifická vlastnost je reprezentována pomocí instance tohoto typu, který rozšiřuje základní typ **Figure**. Z mnoha možností si vybereme dva příklady. Jsou to typy **Line** a **Rect**, které definují odpovídající obrazce:

```
MODULE Lines;
IMPORT Graphics, ...;
TYPE
 Line*=POINTER TO LineDesc;
 LineDesc*=RECORD(Graphics.FigureDesc)
  x1,y1,x2,y2:INTEGER
 END;
PROCEDURE DrawLine(Li:Line);
...
END Lines;

MODULE Rectangles;
IMPORT Graphics, ...;
TYPE
 Rect*=POINTER TO RectDesc;
 RectDesc*=RECORD(Graphics.FigureDesc)
  x,y,w,h:INTEGER
 END;
PROCEDURE DrawRect(r:Rect);
...
END Rectangles;
```

Po této definici může seznam zobrazovaných objektů obsahovat několik čar a obdélníků. V následující části popíši dvě typické akce: vytvoření nového objektu a vykreslení celého seznamu na plochu. Instance typu **Line** je vytvořena a vložena do seznamu takto:

```
PROCEDURE NewLine (list:Graphics.Figure;
          x1,y1,x2,y2:INTEGER):Graphics.Figure;
VAR Li:Line;
BEGIN
 NEW(Li); Li.x1:=x1; Li.y1:=y1; Li.x2:=x2; Li.y2:=y2;
 InsertLast(list,Li);
   (* implementace procedury InsertLast je známá *)
 RETURN Li;
END NewLine;
```

Formální parametr **list** je odkaz na seznam objektů tvořících graf. Podobná inicializační procedura se vyžaduje pro každý objekt (např. **NewRect**, **NewCircle**, ... atd.).

Jiná akce typicky vykonávaná v grafických editorech je vykreslení všech objektů, které jsou obsaženy v seznamu. Touto procedurou je **DrawAll**, která projde celý seznam objektů a jednotlivé objekty vykreslí.

```
PROCEDURE DrawAll(list:Figure);
VAR f:Figure;
BEGIN
```

```
f:=list;
WHILE f#NIL DO
 IF f IS Line THEN DrawLine(f(Line))
 ELSIF f IS Rect THEN DrawRect(f(Rect))
 ELSIF ...
 END;
 f:=f.next;
END
END DrawAll;
```

kde **DrawLine** a **DrawRect** jsou procedury vykreslující linku a obdélník na plochu.

Procedura pro zpracování heterogenního seznamu při výběru je založena na dynamických typech jednotlivých objektů v seznamu.

2.3 Objekt a dynamická vazba procedur

Pojem objektu (v objektově orientovaném přístupu) je v Oberonu svázán s pojmem záznamu. Záznam reprezentuje objekt a jeho vlastnosti (vlastnosti pomocí procedurálních proměnných).

Předpokládejme, že nový objekt (elipsa) se přidává do grafického editoru. Jaké změny budou v textu programu? První je definice nového typu. Podobně jako u **Line** a **Rectangles** ho nazveme **Ellipse**, který je rozšířením základního typu **Figure**:

```
MODULE Ellipses;
IMPORT Graphics,...;
TYPE
 Ellipse*=POINTER TO EllipseDesc;
 EllipseDesc*=RECORD(Graphics.FigureDesc)
  x,y,a,b:INTEGER
 END;
...
END Ellipses;
```

Musí se vytvořit procedura **NewEllipse**, která vytváří instanci typu **Ellipse** a vkládá ji do seznamu. Podobně se vytvoří i procedura **DrawEllipse**. A konečně se musí udělat úpravy v proceduře **DrawAll**. Jde o zařazení části, která zpracovává vykreslení elipsy. Tato část je následující:

```
...
ELSIF f IS Ellipse THEN DrawEllipse(f(Ellipse))
...
```

Jak je na příkladu vidět, modifikace není jednoduchá a ve složitých programech se může stát, že programátor na něco zapomene. Tuto složitou modifikaci lze odstranit použitím procedurálních proměnných. V našem příkladu by to bylo takto:

```
FigureDesc*=RECORD
 draw:PROCEDURE (f:Figure);
 next:Figure
END;
```

Přidala se proměnná **draw** (případně další např. clear, move, ... atd.). Při užití předefinovaného typu **FigureDesc**, musíme přepsat proceduru **DrawAll** takto:

```
PROCEDURE DrawAll(list:Figure);
VAR f:Figure;
BEGIN
  f:=list;
  WHILE f#NIL DO
    f.draw(f);
    f:=f.next
  END
END DrawAll;
```

Pokud se podíváme na předcházející tvar procedury **DrawAll**, je jasné, že tento popis je efektivnější a lépe zabezpečený proti chybám, které mohou vzniknout při přidání nového objektu do grafického editoru. Popis vykreslovacích procedur je abstraktní. Je platný pro všechny objekty v editoru (nynější i budoucí).

Pro obecné schéma práce se typově specifická procedura musí přiřadit do pole **draw**, když je objekt vytvářen. Každý typový objekt musí mít následující inicializační proceduru:

```
      PROCEDURE  NewEllipse (list:Graphics.Figure;
x,y,a,b:INTEGER):Graphics.Figure;
  VAR e:Ellipse;
  BEGIN
    NEW(e);e.x:=x;e.y:=y;... e.draw:=DrawEllipse;
    InsertLast(list,e);
    RETURN e;
  END NewEllipse;
```

Odlišnost od předchozích inicializací je přiřazení procedury **DrawEllipse** do procedurální proměnné.

Použití procedurálních proměnných nám umožňuje tzv. **dynamickou (pozdní) vazbu**, která je specifikována až při běhu programu. Tato vazba je opakem **statické (brzké) vazby**, která je známá už při kompilaci.

Při použití dynamické vazby by pak procedura pro vykreslení jednotlivých objektů měla obsahovat typovou ochranu v celém rozsahu procedury. Ukážeme si to schematicky na příkladu procedury **DrawEllipse**:

```
PROCEDURE DrawEllipse(f:Graphics.Figure);
BEGIN
  WITH f:Ellipse DO
  ...
  END
END DrawEllipse;
```

Všimněte si, že formální parametr procedury je základním (bázovým) typem celé hierarchie. Je to proto, že při definici proměnné **draw** musíme uvést typ, který je znám.

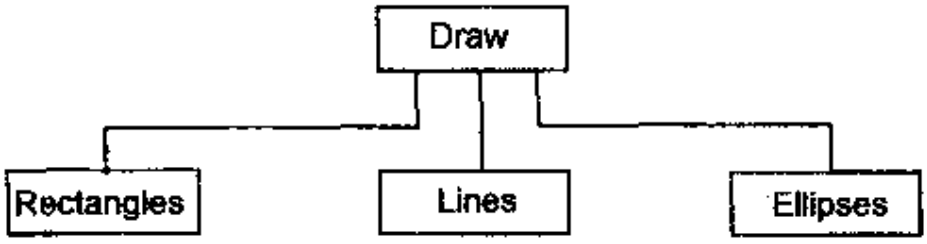
Pokud použijeme procedurální proměnné tímto způsobem, pak přidání dalších objektů do grafického editoru se skládá z následujících kroků:

- definování typu (rozšíření základního typu),
- definování typově závislých procedur,
- definování procedury pro vytvoření instancí nového objektu.

Je samozřejmé, že počet a struktura procedurálních parametrů je na vůli programátora.

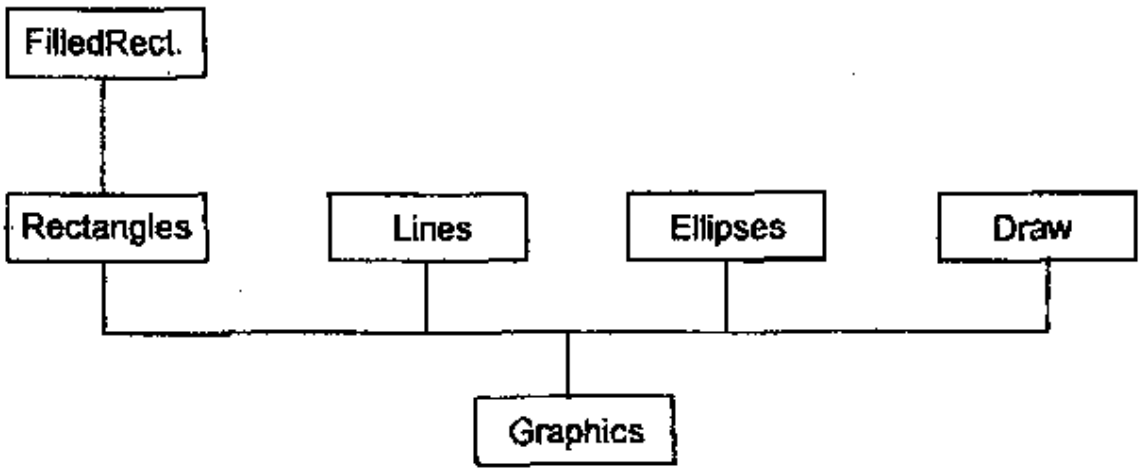
2.4 Struktura modulů objektově orientovaného programu

Jak jsme si ukázali na příkladech, je struktura modulů programu velice významnou otázkou. Při vytváření nového programu by se programátor měl nejprve zabývat navrhovanou hierarchií modulů. Na našem příkladu grafického editoru to můžeme znázornit tímto způsobem:



Obr.2 Normální (většinou navrhovaná) hierarchie modulů

Z obrázku je vidět, že při každém přidání nového objektu (modulu **Ellipses**) se musí rekompilovat modul **Draw**. Naproti tomu na dalším obrázku je hierarchie modulů provedena tak, že se při přidání dalšího objektu kompiloval jen modul s tímto objektem.



Obr.1 Rozšiřitelná hierarchie modulů

Pokud se v myšlenkách vrátíme na začátek článku, pak je nám jasné, že modul **Graphics** je generickým modulem grafického editoru. V něm jsou definovány společné vlastnosti zobrazovaných objektů a další možné obecné procedury. Rozšiřující moduly (např. **Lines, Rectangles**, ... atd.) mají za úkol především:

- rozšířit základní typ a doplnit specifická data,

- definovat procedury, které se přiřazují do procedurálních proměnných, a případně přidávat další,
- vytvářet instance základních objektů.

Struktura viditelná na druhém obrázku je už popsaná v předchozích příkladech. Pokud chceme definovat další objekt např. **Filled Rectangles** můžeme využít již definovaného modulu **Rectangles**. Z rozboru této úlohy je zřejmé, že se musí předefinovat jen typ **Rect** a procedury **DrawRect** a **NewRect** (i v předefinovaných procedurách se mohou použít již definované procedury pro **Rectangles**), u ostatních procedurálních proměnných se použijí procedury z modulu **Rectangles**.

Tímto způsobem můžeme využít již definovaných modulů resp. objektů při definici nových.

V Oberonu je objektová orientace výsledkem hierarchie modulů, které je obrácené "zdola-nahoru" a použití tzv. "up-call" (tento pojem je vysvětlen v [1]). Užitkem z objektové orientace je izolace různých specializací do různých modulů. Dvě významné vlastnosti jsou, že moduly mohou být přidávány bez rekompilace jiných částí programu a znovu použitelnost již napsaného kódu.

3. Závěr

V závěru shrneme všechny poznatky, tohoto příspěvku. Jako první jsme si uvedli definici **generického modulu**. Jde o modul, který stojí v hierarchii modulů nejníže (na začátku). Obsahuje základní (společné) typy a procedury, které s nimi pracují. Měl by se navrhnout tak, aby se nemusel rekompilovat při přidávání dalších modulů nebo malé úpravě textu programu. Teprve **klientský modul** (modul, který generický modul importuje), dodává pomocí typové rozšiřitelnosti aplikačně závislá pole do záznamů a procedury.

Heterogenní datové struktury (záznam) jsou v Oberonu jedny ze dvou vlastností umožňující objektovou orientovanost. Aplikačně závislé typy (definované obvykle v jiných modulech) jsou vytvářeny jako instance rozšiřující základní typ. Záznam může obsahovat nejen základní typy (např. INTEGER, REAL,... atd.), ale i tzv. **procedurální proměnné**, které jsou v Oberonu druhou vlastností umožňující objektovou orientovanost. Při použití procedurálních proměnných jsou jako případné formální parametry procedur předávány objekty základního (bázového) typu. Aplikačně závislé procedury pak musí obsahovat ve svém těle typovou ochranu. Při inicializaci objektu se musí do procedurálních proměnných přiřadit procedury, které pracují s tímto specifickým objektem nebo procedury, které jsou zděděny od předků. Objekt si při přiřazení do hierarchicky nižšího typu (typ některého předka) proměnné pamatuje svůj dynamický typ. Této vlastnosti se dá využít při zjišťování dynamického typu a případné typové ochrany na aplikaci procedur vázaných na tento vyšší typ.

Struktura je jednou z významných otázek při návrhu programu. Měla by se navrhnout tak, aby jednotlivé komponenty byly v samostatných module a hierarchie nenutila rekompilovat již použité moduly při přidání dalšího nebo malé změně. Modifikace by měla být textově lokalizována.

Nakonec bychom se stručně zmínili o jazyku OBERON-2. Tento jazyk je rozšířením jazyka OBERON v několika směrech (např. zavedení cyklu FOR, read-only export, atd.). Toto

rozšíření se také dotklo objektových vlastností jazyka. Ale protože to nebylo předpokládanou náplní tohoto článku další podrobnosti najdete v [1].

Literatura

[1] Reiser M., Wirth N.: *Programming in OBERON*. Step beyond Pascal and Modula, ADDISON-WESLEY, 1992

Ing. Andrea Němcová
nemcova@dcse.fee.vutbr.cz

Doc. Ing. Jan M. Honzik, CSc
honzik@dcse.fee.vutbr.cz

Adresa do zaměstnání:
ÚIVT FEI VUT Brno
Božetěchova 2
612 66 BRNO

Tel: 05-7275242 (Doc. Honzik)
05-7275218 (Ing. Němcová)