Unity a Objekty

Programování 2
(NMIN102)

RNDr. Michal Žemlička, Ph.D.

Větší programy

Časté problémy:

- Ve více programech by se nám hodilo využít stejné řešení nějakého podproblému
  dalo by se vyřešit překopírováním příslušné části zdrojového kódu, ale:
  - jak by se opravovaly chyby (pro každou kopii zvlášť)?
  - to bychom si nechali narůst zdroják nad to, co jsme schopni vnímat?

- co když to, co potřebujeme z jednoho místa, používá stejné identifikátory, jako to, co potřebujeme odjinud?

  $\Rightarrow$ kopírováním bychom si spíše ublížili

- Hodilo by se nám hotové a odladěné části nějak „zabalit“

Moduly (unity)

- Je možné uchovávat logické celky odděleně – po modulech
- Moduly dovolují přistupovat k dané problematice jako k černé skříňce – stačí znát jen rozhraní (+ případná omezení)
- Tyto části je možné samostatně překládat
- Mají oddělené rozhraní a implementaci

Moduly – Syntaxe (PASCAL)

UNIT název; jméno modulu a jméno souboru by si měly odpovídat (UNIT vektory; $\rightarrow$ vektory.pas)

INTERFACE – uvozuje část popisující rozhraní modulu (to, co by měli vidět ti, co s tím budou chtít pracovat)

IMPLEMENTATION – uvozuje část popisující implementaci modulu (aneb jak to funguje – to zas nebývá třeba číst)

Inicializace modulu – kód mezi BEGIN a END se provádí pro každé USES daného modulu v jiném modulu nebo v hlavním programu

Moduly (unity)

- Modul může být používán dlouho a z mnoha programů
- Rozsáhlejší program může být (měl by a bývá) dekomponován do více modulů
- Členění na moduly by mělo pokrývat skupiny problémů nebo uzavřené implmentační celky
- Vystavené podprogramy by měly testovat validitu parametrů a měly by mít ošetřené chyby

Moduly – příklady

Do samostatných modulů můžeme dát například:

- implementaci nějaké složitější datové struktury (či skupiny struktur);
- logicky uzavřenou část aplikace (rozhraní, výpočet, ...)
- *forward:* implementaci tříd či skupin tříd, co k sobě patří
- podporu specifických funkcí (něco jako ovladače) pro určitou skupinu zařízení

Moduly – doporučení

- Názvy (alespoň identifikátory rozhraní) by měly svým názvem jasně říkat, co reprezentují
- Co nezvládneme názvem, doplníme komentářem (je to součást dokumentace)
- Modul je logicky uzavřená část – je dobré jej zdokumentovat (třeba jej bude třeba někdy opravovat či upravovat)
- Je bezpečnější globální proměnné modulu nevystavovat uživateli (má-li mít možnost je měnit, tak raději

nějakými podprogramy, aybchom zajistili, že je nastaví „rozumně“)

Objekty

Objekty

Do samostatných modulů můžeme dát například:

- propojení dat a podprogramů s nimi manipulujících do těsnějších celků
- umožňují obecný náhled na skupinu příbuzných řešení
- umožňují doplňování či modifikování stávajícího kódu o/pro nové chování
- jeden z výrazných směrů vývoje softwaru

Objekty – syntaxe (zjednodušeně)

```
jmeno_typu = OBJECT
             promenna : typ;
             CONSTRUCTOR jmeno;
             DESTRUCTOR jmeno;
             PROCEDURE jmeno(parametry);
             FUNCTION jmeno(parametry):typ;
             END;
```

Objekty – části

atributy – lokální proměnné objektu (něco jako RECORD část)

metody – specifické podprogramy pracující s daným objektem

Objekty – povinné části

konstruktor – metoda, co se provádí při vytváření objektu

destruktor – metoda, co se provádí při zániku objektu

Pomocné struktury je třeba inicializovat a „uklízet“

Objekty – nepovinné části

atributy – lokální proměnné objektu – uchováváme si v nich stav objektu a potřebné údaje

metody – podprogramy pro manipulaci s objektem – není dobré nechávat uživatele (tedy ani nás samotné), aby se „hrabal“ v atributech; mohl by napáchat dost škody.

Objekty – implementace

- každou metodu (až na později řečené vyjímky) je třeba implementovat (hlavička, kde jménem podporgramu je jmeno_tridy.jmeno_metody) + samozřejmě vhodné „tělo“

Objekty – reprezentace množiny

- množiny můžeme v počítači reprezentovat mnoha různými způsoby; reprezentace se liší mj. tím:
  - jaké operace dokáží efektivně podporovat
  - jak velké domény mohou podporovat
  - jak náročné jsou jednotlivé operace
- rádi bychom, abychom mohli používat množiny a nemuseli řešit, jak konkrétně jsou realizovány (např. když se zjistí, že by jiná implementace byla vhodnější, taky abychom mohli změnu provést co nejsnáze)

Objekty – reprezentace množiny (2)

Začneme s jednoduchou obecnou reprezantací množiny (aby se to vešlo na slajd, tak si ukážeme jen podporu pár operací).

Následně si ukážeme dvě různé implementace množiny, které z této základní reprezentace vycházejí.

Objekty – reprezentace množiny (3)

```
TYPE
  prvek = Integer;
  mnozina = OBJECT
    CONSTRUCTOR PrazdnaMnozina;
    DESTRUCTOR Konec;
    FUNCTION jePrazdna:Boolean; virtual;
    PROCEDURE pridejPrvek(x:prvek); virtual;
    FUNCTION jePrvkem(x:prvek):Boolean; virtual;
    FUNCTION jeStejnaJako(m:mnozina):Boolean;
    FUNCTION jePodmnozinou(m:mnozina):Boolean;
    END;
```

Objekty – reprezentace množiny (4)

Popsali jsme si, jak má vypadat typ, který bude reprezentovat objekty typu množna (třidu množina)

Ke každé metodě (včetně konstruktorů a destruktoru) je třeba ukázat, jak má být implementována.

Objekty – reprezentace množiny (5)

```
CONSTRUCTOR mnozina.PrazdnaMnozina;
  BEGIN
  END;

DESTRUCTOR mnozina.Konec;
  BEGIN
  END;

FUNCTION mnozina.jePrazdna:Boolean;
  BEGIN
    jePrazdna:=false;
  END;
```

Objekty – reprezentace množiny (6)

Množinu můžeme reprezentovat například seznamem či binárním stromem

Objekty – reprezentace množiny (7)

Zavedeme si pomocné typy ...

```
seznam = ^uzelSeznamu;
uzelSeznamu = RECORD
  next : seznam;
  data : prvek;
  END;
```

Objekty – reprezentace množiny (8)

... i si odvodíme množinu reprezentovanou seznamem z množiny

```
mnozinaSeznamem = OBJECT(mnozina)
  zacatek : seznam;
  CONSTRUCTOR PrazdnaMnozina;
  DESTRUCTOR Konec;
  FUNCTION jePrazdna:Boolean; virtual;
  PROCEDURE pridejPrvek(x:prvek); virtual;
  FUNCTION jePrvkem(x:prvek):Boolean; virtual;
  FUNCTION jeStejnaJako(m:mnozina):Boolean; virtual;
  FUNCTION jePodmnozinou(m:mnozina):Boolean; virtual;
  END;
```

Objekty – poznámky

- konstruktory a destruktory je třeba implementovat vždy, i když se jmenují stejně, jako u třídy, od níž dědíme
- ostatní metody, jejichž implmentace se od implementace přímého předka neliší, nemusíme řešit znovu
- je možné se v implementaci metody odkázat na implementaci jiné své metody či metody předka