Principy objektově orientovaného programování

Jan M.Honzík

Příspěvek se snaží zpřístupnit široké programátorské obci základní myšlenky, principy a postupy, označované v poslední době souhrnným názvem "objektově orientované programování". Stať má přehledový charakter; detailními vlastnostmi některých objektově orientovaných jazyků, by se měly zabývat jiné příspěvky tohoto sborníku. Příspěvek lze zařadit do oblasti technologie programování, nebo do oblasti anglicky označované jako "software engineering". Vychází především z monografie Bertranda Meyera [1], tvůrce jednoho ze známých objektově orientovaných jazyků "Eiffel" a z pramenů a jejich zpracování uvedených v [2].

0. Ú v o d

'Objektově orientovaný" (ve zkratce tvořící často předponu:"OO"), je jedním z posledních sloganů ve světě tvorby programového vybavení, doplňující nebo snad již nahrazující pojem "strukturovaný" ve smyslu "dokonalý", "moderní" nebo "nejúčinnější". Protože se stává neodmyslitelným atributem pohybu programové tvorby vpřed, používá tento pojem mnoho OO-infikovaných informačních zdrojů a následně i jejich čtenářů. Různé zdroje však používají tento pojem v různých souvislostech a často i s různými významy. Velmi typická je "třídkroková" reakce mnohých programátorů, kteří se začínají seznamovat s filosofií objektově orientovaného programování:

a. "... ale to je přece triviální!"
b. " To přece nemůže pracovat!"
c. " Nic nového! Tak jsem to dělal už dávno...!"

Sekvence může být seřazena i jinak. Ve skutečnosti lze však říci, že ve všech ohledech platí pravý opak. Nedá se říci, že by objektově orientované programování (OOP) bylo něco jednoduchého. Prokazatelně lze říci, že tato technologie je plně funkční a pracuje. Zcela určitě je přístupy OOP v mnohém zcela jiný, než jsou přístupy, s nimiž pracuje většina výkonných programátorů a v řadě aspektů je s nimi nekompatibilní.

Přijměme náhled na OOP jako na soubor principů, metod a nástrojů, které mohou vést k tvorbě aplikačního programového díla, jehož kvality budou mnohem vyšší, než jsou kvality u současné produkce.

Objektově orientovaný návrh je ve své nejjednodušší podobě založen na poměrně jednoduché myšlence: výpočetní systém provádí určité akce nad určitými objekty. Aby výsledný systém byl flexibilnější a jeho části šly snadno

použít znova," je lepší, aby struktura programového systému byla založena na objektech, než na akcích.

Co je to ale přesně "objekt"? Jak se takové objekty určují a jak se definují? Jak s nimi program manipuluje? Jaké jsou možné vazby mezi objekty? Jak lze najít společné rysy mezi různými druhy objektů? Jak se všechny tyto nové přístupy vztahují k osvědčeným vlastnostem tradičního programování jako je správnost programu, snadnost jeho použití a jeho účinnost?

Odpověď lze nalézt v široké škále programovacích technik pro efektivní tvorbu znovupoužitelných, rozšiřitelných a modifikovatelných, spolehlivých a robustních programů, mezi které lze zařadit :

- dědičnost ve své lineární i mnohanásobné podobě
- dynamickou vazbu
- nový náhled na typy a jejich kontrolu
- generičnost
- ukrývání informace
- používání asercí
- "smluvní" programování (programming by contract)
- bezpečné zpracování výjimek

Pro všechny nástroje byly vyvinuty účinné implementační techniky, které umožní jejich praktické využití.

1. K v a l i t a p r o g r a m u a j e j í s l o ž k y .

Stejně jako i u produktů jiného druhu, soustřeďuje se pozornost produkce programů na jejich kvalitu. Objektově orientované programování si klade za cíl výrazné zvýšení kvality programů. Co vlastně tvoří kvalitu programu? Je to soubor vlastností, které podle postavení osoby vůči programu můžeme rozdělit na vlastnosti vnější - ověřitelné uživatelem či zákazníkem a vlastnosti vnitřní - ověřitelné profesionálním programátorem, který program tvoří, rozšiřuje či udržuje. Mezi vnější faktory patří především rychlost, spolehlivost, robustnost,uživatelský komfort, ale také univerzálnost a možnost spolupráce s jinými programy. Mezi vnitřní vlastnosti, můžeme zařadit takové vlastnosti jako modulárnost, modifikovatelnost, možnost nového použití programu či jeho částí a srozumitelnost.

1.1 Správnost programu

Správnost je schopnost programového produktu provádět svou činnost přesně podle požadavků a z nich vyvozených specifikací. Je jednou z významných složek kvality programu.

1.2 Robustnost programu

Robustnost je schopnost programu uchovat si svou funkci i za podmínek, které nejsou běžné či normální. Robustnost odpovídá na otázku, co se stane za podmínek, které nestanovuje zadání. Tím se liší od správnosti, která odpovídá za chování systému v explicitně stanovených podmínkách. Na rozdíl od správnosti je robustnost mnohem neurčitější než přesnost. Zahrnuje chování v situacích, které sice nejsou normální, ale jsou předvídané. Množina takových situací závisí na tvůrci programu. Měla by zajistit, aby nekorektní podnět či situace nevedl ke katastrofickému výsledku, ale spíše ke zdvořilé degradaci programu či jeho čistému ukončení. Někdy se používá pojem spolehlivost jako synonymum pro robustnost; bylo by ale správnější používat pojem spolehlivost pro vlastnost danou současně jak správností tak robustností.

1.3 Modifikovatelnost programu

Mírou modifikovatelnosti programu je snadnost, s jakou může být programový produkt změněn s ohledem na změny ve specifikacích. Změny v malých programech nebývají obtížné. Čím je však program rozsáhlejší, tím obtížnější se v něm provádějí změny. Na zvýšení modifikovatelnosti se podílejí především dva principy:

a. jednoduchost návrhu
b. decentralizace a autonomizace programových částí - modulů

1.4 Znovupoužití

Znovupoužití je vlastnost programového produktu umožňující jeho nové použití pro novou aplikaci, ať už jako celku, nebo jeho části. K nutnosti nového použití vede řada důvodů. Existuje mnoho typických částí programových systémů, které se často opakují. Je bytostně ekonomické soustředit se na jejich vysokou kvalitu s cílem vícenásobného použití.

1.5 Kompatibilita

Mírou kompatibility je snadnost, se kterou lze daný programový produkt kombinovat s jinými produkty. Kompatibilita je velmi důležitou vlastností. Přesto se až příliš často stává, že dochází k potížím při snaze o interakci s jinými programy proto, že náš program staví na podmínkách, které jsou v konfliktu s okolním světem.

Cesta ke kompatibilitě vede přes homogenitu návrhu a přes přijetí standardních konvencí pro meziprogramovou komunikaci. To nejčastěji zahrnuje:

a. Standardní formát souborů (jako např v Unixu, kde každý soubor je chápán jako posloupnost znaků)

b. Standardní formát dat (jako např. v Lispu, kde každá struktura dat i programů je representovaná binárním stromem)

c. Standardní uživatelské rozhraní (jako např. ve Smalltalku, kde každá jednotka vede jednotný způsob styku s uživatelem, založený na oknech, ikonách, grafice ap.)

1.6 Další složky kvality programu

Uvedme pro úplnost stručné definice ještě některých dalších významných složek kvality programu:

Účinnost – je dobré využití technických vlastností výpočetního systému. I když snahy v tomto směru mohou vést i k přehnané snaze o snížení úrovně programování na úroveň stroje, dobré využití technických výpočetních zdrojů v zůstává základním požadavkem kvality programu.

Přenositelnost programu. Mírou přenositelnosti je snadnost, se kterou může být programový produkt přenesen do jiných technických i programových výpočetních prostředí.

Verifikovatelnost. Mírou verifikovatelnosti je snadnost se kterou lze vytvořit testovací procedury a testovací data za účelem detekce a sledování chyb v procesu ověřování programového produktu.

Integrita programu je jeho schopnost zabránit neautorizovanému přístupu a změnám v jeho vnitřní výstavbě.

Snadnost použití, někdy také nazývaná **uživatelský komfort** je vlastnost, umožňující snadné naučení se používat daný program, připravovat pro něj vstupní data a interpretovat jeho výsledky.

Je zřejmé, že mezi všemi až dosud uvedenými složkami kvality programu je řada složek, které se vzájemně vylučují nebo potlačují. Snaha o co nejúčinnější program může vést k úzké vazbě na technické vlastnosti výpočetního prostředku, což může potlačit možnosti portability. Při tvorbě programového produktu s vysokou kvalitou je nutné hledat a stanovit kompromis mezi řadou požadovaných vlastností a cílů. Pro uživatele a zákazníka jsou důležité vnější složky a vlastnosti kvality. K jejich dosažení je třeba použít vnitřních složek kvality programu, které dovede rozeznat pouze profesionální programátor.

1.7 Údržba programů

Hovoří-li se o nákladech na programátorskou práci, hovoří se obvykle o nákladech na vývoj a tvorbu programu. Dlouholeté zkušenosti vyspělých pracovišť však ukazují, že až 70 procent nákladů se vynaloží na údržbu programů. Co vše se rozumí pojmem údržba? Program prostě přestal plnit v plném rozsahu podmínku znovupoužitelnosti a "údržbou" se má této schopnost dosáhnout stejně, jako např u automobilu. Podle [3], lze náklady na údržbu rozdělit zhruba podle velikosti takto:

41.8%	změny v požadavcích uživatele
17.4%	změny ve formátech dat
12.4%	prostředky pro ochranu před závažnými chybami a stavy
9.0%	ladění procedur a funkcí
6.2%	změny v technickém vybavení
5.5%	dokumentace
4.0	zlepšení účinnosti programu
3.4	ostatní

Z přehledu je vidět, že více než dvě pětiny nákladů, jsou důsledky změn a rozšíření programů na základě požadavků uživatelů. Znamená to, že programy jsou "tvrdší", než říká jejich anglicky název (software), a zatím nemají dostatečné předpoklady pro rozšiřitelnost a modifikovatelnost. To vše vede k úvahám o nových vlastnostech programového vybavení, k objektově orientovanému pojetí.

2 M o d u l a r i t a

Pojem "modularita" patří opět k velmi frekventovaným pojmům současných trendů v návrhu programů. Je třeba jej však blíže vysvětlit. Tradiční pojetí jej chápalo jako konstrukce programového celku vytvořeného z malých úseků, nejčastěji podprogramů či subrutin. Tento přístup přinese jen málo užitku v oblasti znovupoužití a rozšiřitelnosti, nebude-li se zabývat otázkou, zda malé úseky – moduly – jsou autonomní, koherentní a organizované v robustní struktuře. Problém modularity se tedy přenáší do oblasti návrhu modulů.

2.1 Pět kriterií hodnocení návrhu modulů

Pro hodnocení návrhu modulů lze použít pěti kriterií:

a. schopnost modulární dekompozice
b. schopnost modulární kompozice
c. srozumitelnost funkce modulu
d. modulární spojitost
e. modulární ochrana

Metoda návrhu modulů splňuje **kriterium modulární dekompozice**, jestli podporuje dekompozici daného problému na několik podproblémů, které mohou být řešeny samostatně. Obecně může být tento proces rekurzivní. Jedním z typických postupů dekompozice je postup shora-dolů.

Metoda návrhu modulů splňuje **kriterium modulární kompozice**, když podporuje vytváření programových elementů, které mohou být volně spojovány do nových programových systémů a to i v jiném programovém prostředí, než ve kterém byly vyvinuty. Zatím co dekompozice probíhá odvozováním programových elementů na základě specifikací, kompozice má obrácený směr. Zajišťuje, že existující elementy mohou být slučovány do nových systémů. Kompozice je úzce spojena se znovuužitím. Cílem je navrhnout dobře definované programové úseky, které by byly široce použitelné v různých situacích.

Příkladem kompozice je spojovací konvence Unixu, která zápisem A|B reprezentuje programový celek, který zajišťuje vstup modulu A a jeho výstup vloží na vstup modulu B.

Kompozice je nezávislá na dekompozici a často je s ní ve svých důsledcích v přímém rozporu. Dekompozice shora-dolů vede k tvorbě modulů, které pro specifické požadavky řeší zvláštní podproblém a nejsou obvykle znovupoužitelné.

Metoda návrhu modulů splňuje **kriterium srozumitelnosti**, jestli podporuje tvorbu modulů, jejichž funkce je srozumitelná člověku z modulu samého, bez potřeby studovat funkci ostatních modulů systému. Jako příklad naprosté nesrozumitelnosti lze uvést systém sestavený z modulů, v němž skutečná funkce modulu závisí na pořadí aktivace ostatních modulů, takže funkci žádného izolovaného modulu nelze určit.

Metoda návrhu modulů splňuje **kriterium spojitosti**, jestliže malá změna ve specifikacích problému má za následek malou změnu v jednom, nebo jen v malém počtu modulů. Tyto změny by neměly postihnout architekturu systému - tedy vzájemné vazby mezi moduly. Pojem "spojitost" je převzata z matematické definice spojité funkce (pro kterou platí, že malá změna argumentu funkce vede k malé změně hodnoty funkce). Této vlastnosti se také říká modifikovatelnost. Příkladem vlastnosti, která podporuje modulární spojitost, je užití symbolických konstant. Změna konstanty se provede pouze na jenom místě a její vliv se rozšíří na všechna místa výskytu této konstanty.

Metoda návrhu modulů splňuje **kriterium modulární ochrany**, jestliže jejím výsledkem je architektura, v níž

vyskyt nenormální situace či chyby se omezí na jeden modul, nebo v nejhorším případě na několik sousedních modulů. Nejde zde o ochranu proti vzniku chyb či o zotavení se z chyby, ale pouze o zabránění jejího šíření.

2.2 Pět principů modularity

Dobrý modulární systém by měl vycházet z pěti dalších principů:

a. princip jazykové modulární jednoty
b. princip malého počtu rozhraní
c. princip slabého rozhraní (slabého spřažení)
d. princip explicitního rozhraní
e. princip ukrývání informace

Princip jazykové modulární jednoty vyjadřuje nutnost podpory modularity jazykovými prostředky programovacího jazyka. Tento princip splňuje modul, který koresponduje se syntaktickou jednotkou použitého jazyka. Tento princip vychází z požadavku kompozice, dekompozice i ochrany proti chybám, které byly uvedeny v předcházejícím odstavci. Uplatnění tohoto principu snižuje naději, že by dobrý modulární návrh mohl být implementován bez prostředku podporujícího modularitu.

Princip malého počtu rozhraní říká, že modul by měl komunikovat s co nejmenším počtem ostatních modulů. Je-li celkový počet modulů N, pak by se celkový počet vzájemných komunikačních rozhraní měl co nejvíce blížit počtu (N-1) (jak je to v hvězdicově centrálním propojení).

Princip slabého rozhraní (slabého spřažení) žádá, aby si dva moduly, které vzájemně komunikují, vyměňovaly co nejméně informace. Typickým příkladem opaku slabého rozhraní je Fortranský blok COMMON.

Princip explicitního rozhraní vyžaduje, aby všude, kde dva moduly vzájemně komunikují, to bylo patrné z textového tvaru těchto modulů. Za tímto principem stojí kriterium kompozice, dekompozice, spojitosti i srozumitelnosti.

Princip ukrývání informace žádá, aby veškerá informace o modulu byla dostupná pouze uvnitř modulu s výjimkou té, která byla prohlášena za informaci veřejnou.

2.3 Otevřený a uzavřený modul

Dobrá technika modulární dekompozice umožňuje vytváření modulů, které mohou být otevřené nebo uzavřené.

Modulu se říká, že je otevřený, jestliže může být dále rozšiřován. Lze například rozšířit jeho datové struktury o nové položky, nebo soubor jeho funkcí o některé nové funkce.

Modul je uzavřený, jestliže je již plně k dispozici k použití jinými moduly, je dobře a úplně definován. Je-li modulem programovacího jazyka, může být přeložen a uchováván v knihovně modulů.

3 Různé pohledy na znovupoužití

Opakované použití jako princip se ve světě programování vyskytuje v mnoha podobách. Již sám cyklus je triviálním případem znovupoužití sekvence příkazů. Velmi často se znovu používají zdrojové podoby úspěšných algoritmů, např. v oblasti vyhledávání a řazení. Absolutní znovupoužití programů a jejich úseků má i své psychologické bariery; vzato ad absurdum, vedlo by ke ztrátě zaměstnání pro programátory.

3.1 Struktura modulu s ohledem na znovupoužití

Proveďme analyzu typického algoritmu pro vyhledávání v tabulce. Pro účely zápisu použijme konvencí vycházejících z Pascalu:

```
function VyhledejElement
            (Element:TypElement; Tab:TypTabulky):Boolean;
var Pozice:TypPozice;
begin
   Pozice: PočátečníPozice(Element,Tab);
   while notTabulkaVyčerpaná(Pozice,Tab) and
            not PrvekNalezen(Pozice,Element,Tab)
   do Pozice:=DalšíPozice(Pozice,Element,Tabulka);
   Search:=not TabulkaVyčerpaná(Pozice,Tab)
end
```

Aby modul, jehož jádrem bude vyhledávání, splňoval požadavky na znovupoužití, měl by mít tyto vlastnosti:

a. možnost změny typu
b. možnost změny datové struktury a jí odpovídajících algoritmů
c. existence samostatných sdružených operací
d. použití operací bez potřeby znalostí jejich implementace
e. koncepční jednota a integrita návrhu a implementace k sobě vzájemně náležejících operací

Možnost **změny typu** se na ukázce projeví možností použití různých typů TypElement.

Možnost změny datové struktury a jí odpovídajících algoritmů se projeví v možnosti, aby implementace tabulky mohla být změněna na sekvenční vyhledávání, stromové vyhledávání nebo na vyhledávání v tabulce s rozptýlenými položkami.

Vyhledávání není jedinou operací nad tabulkou. S vyhledáváním souvisí i ostatní sdružené operace jako inicializace, vkládání, rušení a jiné operace.

Nezávislost na implementaci lze chápat nejen v souvislosti s dobou života programu – tedy s ohledem na znovupoužití – ale i s ohledem na změny v době běhu programu – tedy s ohledem na rozšiřitelnost. Připusťme možnost, že operace search(Element,Tab) může pracovat s různými typy tabulek. Pak by modul musel obsahovat mechanismus, který by aplikoval algoritmus vyhledávání s ohledem na použitý typ tabulky.

```
např. if Tab je typu A then
          "použij vyhledávací algoritmus A"
      elsif Tab je typu B then
          "použij vyhledávací algoritmus B"
      elsif ...
```

Statická implementace takového rozhodovacího mechanismu ať už zabudovaná do služebního modulu, nebo požadovaná od uživatele, vede ke zvýšení složitosti a nepřehlednosti programu. Řešením tohoto závažného problému je dynamická (nebo také pozdní) vazba (dynamic binding, late binding), která spřáhne volání operace s právě požadovanou verzí implementace až v době běhu programu. Tato vlastnost je úzce spojena s dědičností, což je charakteristický rys objektově orientované technologie.

Cílem jednoty a integrity společných operací je nalézt a udržet konzistenci implementace. Na uvedeném příkladě to lze demonstrovat v pro několik implementací typu sekvenčního vyhledávání. Algoritmus vyhledávání bude velmi podobný ať už půjde o vyhledávání v poli, seznamu nebo souboru. Bude se lišit jen implementací některých základních operací jako "InicializaceVyhledávání" (nastavující první index pole, ukazatel začátku seznamu, nebo provádějící Reset pro soubor), "DalšíPozice" (zajišťující přechod na další prvek sekvenční struktury) a "TabulkaVyčerpaná" (určující, zda se již prošlo celou sekvencí).

3.2 Podprogramy

Klasické východisko potřeby znovupoužitelnosti je vytváření knihoven podprogramů. Knihovny podprogramů,

procedur, funkcí jsou velmi úspěšné v řadě oblastí. Účinné využití takových knihoven je spojeno se splněním těchto podmínek:

- Každý problém má jednoduchou specifikaci a jednotlivé instance problému lze rozlišit malým počtem parametrů.

- Jednotlivé problémy mohou být od sebe snadno a jasně rozlišitelné.

- Problémy by neměly být vázány na složité datové struktury; tyto datové struktury by pronikly do mnoha podprogramů a ztratila by se jejich vzájemná nezávislost a autonomie.

Všude tam, kde je problém vázán na složitější datové struktury, které mohou navíc měnit a rozšiřovat svou podstatu, přestávají být prosté podprogramy a jejich seskupení do knihoven účinnými.

3.3 Balíky (packages) a moduly

Balíky Ady nebo moduly jazyka Modula-2 jsou krokem vpřed ve směru k lepší znovupoužitelnosti a rozšiřitelnosti. Modul (nebo balík) může obsahovat více než jeden podprogram a obsahuje také deklarace typů, konstant a proměnných, nad kterými pracují podprogramy. V uvedeném příkladu by tedy mohl existovat modul "VyhledávacíTabulka", v němž by byly obsaženy podprogramy - operace, pro práci s tabulkou jako:

```
InicializaceTabulky(Var Tab:TypTab)
VložElement(Element:TypElement;var Tab:TypTab)
ZrušElement(Element:TypElement;var Tab:TypTab)
VyhledejElement(Element:TypElement;var Tab:TypTab):Boolean
```

Současně s operacemi jsou v modulu lokalizovány datové struktury, které danou tabulku implementují. Tento přístup je v souladu s principem jazykové modulární jednoty (2.2) a přináší výrazné zlepšení vlastností s ohledem na uzavírací a data skrývající mechanismy.

Výhoda pro tvůrce implementace spočívá v tom, že všechny části, které patří principiálně k sobě jsou umístěny na jedném místě a jsou kompilovány společně. Tato skutečnost výrazně podporuje vývoj a údržbu programu. Jsou-li jednotlivé operace odděleny, vždy se vyskytuje nebezpečí, že při určité aktualizaci a modifikaci přehlédneme nebo zapomeneme na některou operaci.

Pro uživatele modulu je výhodou, že všechny operace, které patří k danému problému, nachází na jednom místě.

Balíky a moduly jsou vyspělejším nástrojem než podprogramy, neřeší však ostatní požadavky znovupoužití, a to zejména možnost změny datové struktury a jejích algoritmů (b), nezávislost uživatele na modulů, nazávislost uživatele na implementaci (d) a koncepční jednota a integrita návrhu a implementace k sobě náležejících operací.

3.4. Polymorfismus a generičnost.

Významnou vlastností vyššího stupně technologie tvorby programování je **polymorfismus** (v [1] se používá pojem přeplnění - overloading). Je to schopnost spojit s jedním jménem (identifikátorem) v programu více významů. Typickým kandidátem pro aplikaci polymorfismu jsou jména operací (procedur a funkcí). Představme si případ, kdy v jednom programu vytváříme několik různých implementací vyhledávací tabulky, každá se svou množinou operací nad tabulkou. Tradiční přístup by nás nutil použít pro operace každé implementace odlišná jména. Polymorfismus umožňuje, aby příkaz ve tvaru

```
b:=VyhledejElement(Element,Tab)
```

mohl být použit bez ohledu na to, který z několika možných typů tabulky je jako parametr Tab použit.

Další významnou vlastností na této úrovni je generičnost, tak jak ji nalézáme v jazycích ADA a CLU, a které v jiných jazycích dosahujeme za cenu méně přímých obratů. Generičnost je schopnost definovat parametrizované moduly. (Pozn. Parametrické programy, které jsou pojmem velmi blízkým, byly ostatně předmětem zájmu řady ročníků semináře Programování v Ostravě.) Generický modul není ve své původní podobě přímo použitelný. Slouží jako vzor, forma, v níž jsou na místě typů použity **formální generické parametry**. Skutečné moduly, které jsou instancemi generického modulu, se získají dosazením **skutečných generických parametrů** za parametry formální.

Zkusme nyní zvážit všechna pro a proti polymorfismu a generičnosti, ve světle požadavků na znovupoužitelnost (3.1.). Polymorfismus i generičnost poskytují symetrické možnosti:

- Polymorfismus umožňuje uživateli modulů, aby vytvářel týž uživatelský program a přitom používal různých implementací datových struktur, poskytovaných různými moduly.

- Generičnost umožňuje tvůrci modulů, aby vytvářel týž program modulu, popisující všechny instance téže

implementace datové struktury aplikované na různé datové typy.

Z hlediska znovupoužití řeší generičnost požadavek (a) z odstavce 3.1 (možnost změny typu). Polymorfismus uspokojuje požadavek (b) (změna datových typů a odpovídajících algoritmů) a (d) (použití operací bez potřeby znalosti jejich implementace). Při bližším posouzení to zase tak velký úspěch není. Nebyl vůbec uspokojen požadavek (e), který by měl umožnit zachytit společná jádra skupin implementací nad obecně stejnou datovou strukturou. Polymorfismus v tomto bodu nic neřeší. Ani generičnost nepomáhá, protože poskytuje pouze dva typy modulů:

- Generické moduly, které jsou parametrizované a tudíž otevřeny změnám, ale nejsou přímo použitelné.
- Instance modulů, které jsou přímo použitelné, ale neumožňují další úpravy a změny.

Ani jeden mechanismus neumožňuje složitější hierarchii reprezentací s různou úrovní parametrizace. Oba mechanismy trpí i jinými významnými omezeními: Neumožňují uživateli používat různé implementace datové struktury, aniž by uživatel musel vědět, která implementace je v dané instanci použita.

- Generické moduly nejsou uživatelem použitelné přímo, ale pouze prostřednictvím jejich instancí. Tyto instance již ztratily svou modifikovatelnost, protože vznikly dosazením skutečných generických parametrů na místo formálních generických parametrů.

- Polymorfismus není v podstatě nic jiného, než mechanismus, který zbavuje uživatele povinnosti vymýšlet různá jména pro různé implementace týchž operací a z toho pohledu je to tedy jen zátěž pro kompilátor. Každé volání polymorfní operace se odkazuje právě na jednu verzi operace; přitom uživatel v daném okamžiku přesně ví, o kterou implementaci jde

Všimněme si, že uživatel nemusí znát, jak jsou jednotlivé verze implementované, protože většina jazyků pracujících s generickými moduly (ADA), umožňuje použití modulů prostřednictvím rozhraní, které popisuje každý modul nezávisle na jeho vnitřní implementaci. Uživatel ale musí rozhodnout a vědět, jaký druh implementace použije pro např. vyhledávací tabulku , tedy zda použije strom, sekvenční strukturu či tabulku s rozptýlenými položkami.

Skutečná nezávislost na reprezentaci znamená, že můžeme zapsat operaci VyhledejElement(Element,Tab) se sémantikou:

Zjisti zda v tabulce Tab existuje prvek Element a to bez ohledu na to, jaky typ tabulky se v okamžiku vyvolání použije. Dosažení tohoto stupně modifikovatelnosti, podstatné pro dosažení znovupoužitelných programových úseků, je možné prostředky objektově orientovaného návrhu.

4 Cesta k objektově orientované technologii programování

4.1 Funkce versus data

Jednou z klíčových otázek je, zda je pro podporu požadavku snadné modifikovatelnosti lepší vytvářet strukturu programu na základě funkce nebo na základě dat.

Každý úspěšný programový komplex projde v průběhu svého života řadou změn a nejčastěji se požaduje, aby se rozšiřovaly jeho funkce. Např. program pro výpočet mezd, se postupně rozšiřuje o pořizování statistiky, o evidenci o úsporách zaměstnanců, o informace o daňovém odvodu atd., nebo překladač, který původně produkoval jen cílový kód, je postupně rozšiřován o syntaktický analyzátor, statický analyzu, optimalizaci atd. Změny jsou postupné, inkrementální, systém jako takový však má stále tyž charakter: je to mzdový program resp. překladač. Je-li architektura programu postavena na funkcích, lze jen těžko očekávat, že malé změně v zadání bude odpovídat malá změna v programu a že změny v programu budou tak spojité, jako změny v požadavcích. Je-li program strukturován na základě údajů, lze očekávat, že i při změnách bude program pracovat se stejnými nebo z jistého abstraktního pohledu podobnými daty, ať již to jsou denní výkazy či daňové tabulky u mzdového programu, nebo zdrojový text, syntaktický strom, cílový kód u překladače. Zdá se, že při typických změnách, které se očekávají u programu, přetrvávají data programu mnohem výrazněji, než jeho funkce.

Podrobíme-li z tohoto zorného úhlu velmi populární postup vytváření programu na základě funkční dekompozice shora-dolů, s překvapením shledáme, že tato metoda je velmi vhodná pro výuku základům algoritmizace a pro tvorbu malých nebo speciálních programů, o nichž se nebude předpokládat další vývoj nebo údržba. Není však vhodný pro rozsáhlé programy s dlouhou životností a s předpokladem mnoha modifikací. Ne, že by se tímto způsobem program nedal program dobře a rychle vyvinout. Musíme si ale uvědomit, že v tomto případě obětujeme snadnost provádět změny a modifikace, které mohou mít dlouhodobý charakter, rychlému

vývoji programu, který má krátkodobý charakter. Na dřívějším názoru na účinnost postupu shora-dolů se tedy vlastně nic nemění. Z širšího ekonomického pohledu na náklady spojené s dlouhým životem programu však není funkční dekompozice účelná. S ohledem na specializovanost funkcí vzniklých dekompozicí je jejich znovupoužitelnost velmi problematická. Protože při funkčně orientované architektuře programu, jsou použité datové struktury sekundární, je i kompatibilita programů , která závisí především na datových strukturách, záležitostí sekundární.

Je-li program postaven na základě dat, lze pozorovat, že nedostatky, které se vyčítají funkčnímu přístupu jsou přednostmi přístupu datového, a to jak kompatibilita, možnost znovupoužití programu nebo jeho částí a spojitost, jako míra modifikovatelnosti.

4.2 Objektově orientovaný návrh

Jako pracovní definici můžeme přijmout, že:

> **Objektově orientovaný návrh je metoda, která vede k architektuře programového celku, založené na objektech, s nimiž manipuluje každý systém a podsystém.**

S objektově orientovaným přístupem k návrhu se traduje moudrost, která říká, že upírání pozornosti na to, co má programový systém dělat je vhodné co nejvíce oddalovat. Návrh a tvorba systému má charakter analýzy vztahů, která vede ke klasifikaci objektů a jejich tříd. Návrh systému je postupným zlepšováním a zpřesňováním založeným na lepším porozumění objektových tříd. Specifikaci nejvyšší, konečné a nejabstraktnější funkce, kterou má programový celek plnit, je nutné odložit na co nepozdější etapu vývoje. Takový přístup k návrhu je zcela nový, a pro mnoho programátorů zcela šokující. Tradičním zvykem je, že ke konečné funkci programu upírá návrhář pozornost od samotného začátku vývoje. Teze, že je nutno upřít pozornost napřed na data, která se budou vyskytovat v systému a nesoustředit se na bezprostřední funkci systému, je klíčem ke znovupoužití a k modifikovatelnosti.

S objektově orientovaným návrhem souvisejí čtyři základní otázky:

- Jak lze najít objekty
- Jak popisovat objekty
- Jak popisovat vzájemné vztahy a společné vlastnosti objektů

- Jak použít objektů ke strukturalizaci programů

4.2 Nalezení objektů

Prvním problémem, zejména pro programátora nezkušeného v oblasti OOP je vymezení objektů v daném systému. Abychom pochopili oč vlastně jde, podívejme se na to, čemu vlastně slouží programy. Programy dávají určité odpovědi na stav vnějšího světa (v případech programů řešících určité problémy), provádějí interakci se světem (v případě řídicích systémů) nebo vytvářejí nové entity světa (jako např. při použití textového editor nebo kompilátoru). V každém případě musí být program založen na popisu aspektů vnějšího světa, týkajících se řešené aplikace, např. fyzikálních zákonů v případě vědeckého výpočtu, mzdového systému v případě mzdového programu, nebo syntaxe a sémantiky jazyka v případě kompilátoru. Na dobře strukturovaný program pak nahlížíme jako na **operační model** určitého aspektu vnějšího světa. Komponenty našeho programu jsou vytvářeny na základě obrazu komponent reálného problému. Je skutečností, že celý tento přístup k programování je nápadně podobný, ne-li identický s přístupem k tvorbě **simulačních modelů**. Není náhodou, že programovací jazyk Simula 67 byl jedním z východisek pro hledání cest OOP. Tento pohled nám usnadní určování toho, co v našem vytvářeném programu bude vytvářet objekty.

4.3 Abstraktní typ dat jako popis třídy objektů

Pojmem **třída** se označuje popis množiny objektů se shodnými vlastnostmi. Pojem třída se často chápe jako synonymum pro **typ objektu**. Ačkoliv ne ve všech objektově orientovaných literárních zdrojích je smysl těchto pojmů chápán jednotně, lze přijmout výklad, že pojem **třída** je širší a obecnější pojem a že **typ objektu**, je jednou z možností, jak specifikovat třídu (viz Turbo Pascal 5.5 [2]). Oba pojmy představují úhrnné označení jednotlivých instancí se společnými vlastnostmi. Jak bylo podrobněji uvedeno v [2], je typ charakteristický typovou kontrolou, jako významnou vlastností, která sice zvýrazňuje čistotu a bezpečnost konstrukce, ale může omezovat tvůrčí invenci programátora. O třídě se v této souvislosti hovoří jako o "šabloně", pomocí které lze vytvářet nové množiny objektů - nové třídy.

V souvislosti s typovou specifikací třídy objektů se hovoří o specifikaci pomocí **abstraktního typu dat**. Abstraktní typ dat (ATD) je definován množinou hodnot, kterých může nabýt a množinou operací, které jsou nad ním definovány. Specifikace ATD však nic neříká o tom, jak jsou vlastní data a operace nad nimi implementovány. Proto

skutečné abstraktní typy dat plně uskutečňují to, čemu říkáme ukrývání vnitřní informace. Specifikace ATD sestává ze specifikace jejich syntaxe a sémantiky. Zatímco ke specifikaci syntaxe se často používá algebraických zápisů v textové nebo grafické podobě (viz. např. diagram signatury), sémantiku lze formálně popisovat soustavou axiomů. Specifikace ATD zásobník může mít např. následující tvar:

Typy
    TSTACK[TE]
Operace
    empty : TSTACK[TE] -> BOOLEAN
    StackInit : -> TSTACK[TE]
    push : TE x TSTACK[TE] -> TSTACK[TE]
    pop : TSTACK[TE] -> TSTACK[TE]
    top : TSTACK[TE] +> TSTACK[TE]
Předběžné podmínky
    pre top(STACK:TSTACK[TE])= (not empty(STACK))

Axiomy
    Pro všechny E:TE, STACK:TSTACK musí být pravdivé axiomy:

    empty(new(STACK))
    not empty(push(E,STACK))
    top(push(E,STACK))=E
    pop(push(E,STACK))=STACK

Parametr TE je typ elementu zásobníku, a jeho uvedení v hranaté závorce znamená, že daný abstraktní typ dat je generický.

Symbol +> v algebraické specifikaci operace top znamená, že jde o parciální operaci, která nemusí být definována nad každým objektem typu TSTACK. Jak víme, operace top není definována pro prázdný zásobník.

Stvořitelská operace, která nemá na levé straně šipky žádný operátor se říká také generátor, nebo konstruktor. Protože tato operace inicializuje práci s objektem, bývá její jméno často odvozeno od pojmu "inicializace".

Na závěr odstavce můžeme shrnout, že ATD mohou být generické a lze je definovat pomocí operací, předběžných podmínek a axiomů. Předběžné podmínky a axiomy vyjadřují sémantiku typu a jsou významné pro jeho úplný a jednoznačný popis. Specifikace ATD má charakter formálního matematického popisu, bez vedlejších jevů. Třída je reprezentována určitou implementací abstraktního typu dat, nebo skupinou takových implementací.

4.4 Upřesnění definice objektově orientovaného návrhu

Upřesněme definici z 4.2 na tvar:

> **Objektově orientovaný návrh je konstrukce programového vybavení jako strukturovaného souhrnu implementací abstraktních typů dat**

V objektově orientované architektuře je každý modul postaven na datové abstrakci, tedy na množině datových struktur popsaných operacemi, které definuje jejich rozhraní a vlastnostmi těchto operací.

Stavebním kamenům objektově orientovaných systémů říkáme třídy. Jak vyplývá z výše uvedené definice, třídou rozumíme implementaci abstraktního typu dat, nikoli typ sám o sobě.

Definice používá pojmu **souhrn**, aby **zdůraznila** nezávislost tříd na systému a jejich samostatný význam jako takových. Takové třídy mohou být znovupoužity i v jiných systémech. Výstavbu systému pojímáme spíše jako zdola-nahoru vytvářený souhrn existujících tříd, než proces shora-dolů, který začíná na prázdném papíře.

Souhrn je **strukturován** což odráží skutečnost, že mezi jednotlivými třídami existují významné vztahy. Významnými vztahy jsou vztahy označované jako vztah **klienta** a vztah potomka.

Mezi dvěma třídami je vztah klienta, jestliže jedna třída (klientská), využívá služeb jiné třídy (služební). Je-li např. třídou "zásobník", který má být implementován polem, může být třída zásobník klientem třídy "pole".

Třída je **potomkem** jedné, nebo i více tříd, je-li navržena jako rozšíření, nebo naopak specializace těchto tříd. Mechanismu, který takový návrh umožňuje se nazývá **dědičnost**, a je jedním z nejvýznamnějších vlastností OOP.

4.5 Sedm kroků na cestě k objektově orientovanému návrhu

Objektově orientovaný návrh je termín, který se používá pro různé programovací technologie používající různé programovací jazyky. Na každý postup či použitý jazyk, splňuje plnohodnotně představy o skutečné objektové

orientaci. Proto uveďme postupné kroky, které vedou ke konečnému cíli. Lze říci, že až po splnění posledního kroku, lze hovořit o skutečně objektově orientovaném programování.

> 1.krok: Modulární struktura založená na objektech:
> Systémy jsou modulární na základě jejich datových struktur

> 2.krok: Datové abstrakce:
> Objekty jsou popsány jako implementace abstraktních typů dat

Tento požadavek splňují některé jazyky jako ADA, Modula-2 nebo i Fortran a naopak nesplňují ho ANSI Pascal, Cobol nebo Basic.

> 3.krok: Automatické ovládání dynamického přidělování paměti:
> Nepoužívané objekty musí být odstraněny a musí uvolnit paměť automatickým mechanismem, bez přímého zásahu programátora

Tento požadavek splňuje jen málo jazyků. Mezi ně patří i LISP, který se právě proto často používá pro implementaci objektově orientovaných jazyků.

Další krok nutně oddělí objektově orientované jazyky (OOJ) od ostatních. Mohlo by se chápat, že jazyk, který má prostředky pro vytváření ATD jako ADA nebo Modula, že jsou vhodné pro OOP. V těchto jazycích je však modul čistě syntaktickou konstrukcí, umožňující shrnout prvky programu, které patří k sobě ale modul sám nemá v jazyce samostatnou vlastní sémantiku jako typ, proměnná či procedura. OOJ spojují pojem modul s pojmem typ. Spojení těchto dvou různých pojmů dává OOP tu zvláštní příchuť, která budí pochybnosti v tradičně orientovaném programátoru.

> 4.krok: Třídy :
> Každý typ, který není jednoduchý je modul a každý modul vyšší úrovně je typ

Na základní typy (např. integer) se samozřejmě nenahlíží jako na moduly: obrat "modul vyšší úrovně" dovoluje aby

programovou strukturovací jednotkou byla třeba procedura, která není typem. Třídou se tedy rozumí konstrukce, dovolující kombinovat aspekty modulu a typu.

Další krok je přirozeným důsledkem předcházejícího kroku: jsou-li typy spojeny s moduly, mohla by se možnost nového použití spojit s těmito dvěma principy:

- možnost, aby modul mohl přímo využívat entity definované v jiném modulu (vztah "klient")
- princip podtypu, který je definován přidáním nových vlastností k již existujícímu typu (novou vlastností může být i omezení či zrušení původních vlastností)

> 5.krok: Dědičnost :
> Třída může být definována jako rozšíření nebo omezení vlastností jiné třídy

V takovém případě říkáme, že nová třída je dědicem, nebo potomkem jiné třídy.

Dědičnost otevírá možnost **polymorfismu**, který dovoluje, aby se daná programová entita odkazovala v době výpočtu na jistou službu různých tříd a také možnost **pozdní (dynamické) vazby**, která zajišťuje, že systém automaticky vybírá tu verzi služby (službu té třídy), která odpovídá dané instanci.

> 6.krok: Polymorfismus a pozdní (dynamická) vazba:
> Entity programu se mohou odkazovat na objekty více než jedné třídy a operacím je dovoleno, aby měly různé realizace v různých třídách

Poslední krok rozšiřuje pojem dědičnosti.

> 7.krok: Násobná a opakovaná dědičnost:
> Je možné deklarovat třídu, která je dědicem více než jedné třídy a která je dědicem téže třídy více než jednou.

Povolíme-li násobnou dědičnost, pak dříve nebo později dojdeme k situaci, že chceme definovat třídu-dědice dvou

různých tříd, které jsou obě odvozeny dědictvím od téže třídy třetí.

5 Závěr

Objektově orientované programování a jeho technologie nepřišlo na svět proto, aby potřelo staré a osvědčené zkušenosti, ale jen proto, aby je naplnilo. Mnoho z nás starších, se možná nedožije nadšení z výsledku vlastního produktivního programování objektově orientovaným přístupem s využitím dokonalého objektově orientovaného jazyka. Nemůžeme však nevidět, že generaci nastupujících programátorů očekávají problémy, jejichž obzor je mnohým z nás ještě skryt. Pro mnohé z nich, bude prezentovaná technologie základní metodou řešení.

Bylo by zajímavé probrat, jak by se filosofie OOP dala implantovat do stávajících jazyků, nebo do jazyků, které se objektově orientovanému přístupu více či méně blíží. To by však už byl úplně jiný příspěvek.

6 Literatura

[1] Meyer B.: Object-oriented software construction
Prentice Hall, New York,London,Toronto,Sydney,Tokyo
1988

[2] Honzík J.M.: Turbo Pascal 5.5 - technologie objektově
Orientovaného programování v programovacím jazyku Turbo
Pascal verze 5.5, JZD Agrokombinát Slušovice, 1989

[3] Lientz,B.P.,Swanson E.B.: Software Maintenance:
A User/Management Tug of War,
Data Management,pp.26-30, April 1979

[4] Honzík,J.M. a kolektiv: Programovací techniky
skriptum FE VUT v Brně, SNTL Praha 1987

Doc.Ing.Jan M.Honzík,CSc.
katedra počítačů FE VUT v Brně
Božetěchova 2, 612 66 Brno 12