OOP z pohledu programování

Pavel Drbal

Objektově orientované programování je věc vysloveně módní a moderní, hodně se o tom mluví, stále se ale mluví o tom, jak objektově programovat, méně již proč. Místo vysvětlení předností a nedostatků se používají metafory. Je to dvojsečný nástroj. Objekty se přirovnávají k panelům. Z panelů postavíme dům rychleji než z cihel (cihla = příkaz programovacího jazyka). Nejsem však úplně přesvědčen, že panelové domy jsou lepší než cihlové.

Objektově orientované programování má výhody, velmi výrazné výhody, ty ale neleží na povrchu. Musíme se nad problémy programování trochu zamyslet, abychom určili, jaké použití objektů přinese výhody a co je jen ornamentem na umění programovat. Ty dvě nebo tři výhody jsou popsány v následujícím textu.

Abstraktní data a abstraktní datové typy

Již dlouhou dobu kolovala mezi programátory myšlenka, že program by měl být nezávislý na reprezentaci dat.

Hlavní myšlenka je tato: Datům jsou přiřazeny funkce, které umožňují manipulaci s nimi. Program nesmí manipulovat přímo s daty (nesmí například použít přiřazovací příkaz); manipulace s daty je možná pouze prostřednictvím zmíněných funkcí. Všechny požadavky na data je možné realizovat pouze jako parametry funkcí. Abstraktní data se tento přístup jmenuje proto, že program nepracuje s konkrétními daty, ale s jejich abstrakcemi, definovanými pouze příslušnými operacemi (dovolenými funkcemi).

Připadá Vám to nápadně podobné pojmu zapouzdření v OOP? Podobnost není náhodná. OOP je praktickou realizací několika dosud spíše teoretických principů. Jedním z nich je i princip abstraktních dat.

Jaká je výhoda OOP oproti abstraktním datům? Vidím dvě takové výhody, obě významné.

Myšlenka použití abstraktních dat byla pouze doporučením. Říkala: programátor „má udělat", programátor „musí", ale tím, že to bylo pouze doporučení, připouštělo i opak.

Přímo modelový případ znám ze svého okolí. Programový systém byl rozdělen na několik komponent. Centrální komponentu vytvářela skupina lidí, která používala běžné

metodiky (vrstevnatost, nezaměnitelnost služeb různých úrovní, abstraktní data ap.). Ostatním komponentám nabízela služby ve formě volání procedur. Autorem jiné, klíčové komponenty, byl člověk, který všechny metodiky pokládal za hloupost, věřil jen tomu, co sám objevil (říkejme mu František). Tento František nepoužíval služeb procedur a informace získával přímo z bajtů a bitů. Jak léta ubíhala, programový systém se rozvíjel a měnil a asi dvakrát ročně bylo vidět rozčíleného Františka, jak nadává na celý svět, protože programy, které vůbec neměnil, najednou přestaly fungovat. Vysvětlení je prosté. Při rozvoji systému se měnila i centrální komponenta, ovšem tak, že zveřejněné procedury (služby) zůstaly zachovány. Služby se rozšiřovaly a přidávaly a úměrně tomu se měnila fyzická reprezentace dat. Měnila se třeba v tom smyslu, že některé úseky paměti, které dosud byly nulové (rezervní), se začaly používat – a občas byly nenulové. Za těchto podmínek při ilegálním používání fyzické reprezentace dat je pramen Františkových potíží jasný.

V OOP k takovým potížím nemůže dojít. Uživatel objektu má přístup pouze k zveřejněným službám, ať již procedurám (metodám) nebo datům (slotům). Tou výraznou užitečnou vlastností OOP je to, že je znemožněn přístup k soukromým složkám. Přesněji řečeno, OOP je podmíněno příslušným aparátem, který umožňuje používání objektů a znemožňuje jejich zneužití. Objektové programování bez tohoto aparátu nelze považovat za plnohodnotné. Objektově orientované programování pouze jako doporučení ztrácí svou významnou výhodu. I když máme sebelepší předsevzetí, že tato doporučení dodržíme, po čase z lenosti, z roztržitosti nebo zapomětlivosti pravidla porušíme. Vznikne zdroj chyb, navíc skutečný program nebude odpovídat našim představám o něm (nebudou dodrženy pravidla OOP, i když si myslíme, že je dodržujeme). Dle mého názoru se OOP úzce váže na aparát, který jej umožňuje a kontroluje.

Druhá výhoda OOP je psychologického charakteru, je hůře postižitelná. Vzhledem k módnosti OOP, bez ohledu na zkušenosti, programátor chce pracovat s objekty. Musí je tedy vyhledat, musí nejprve určit, co to jsou objekty a jaké služby od nich lze vyžadovat. Program vzniká jako výsledek střetnutí dvou koncepcí:

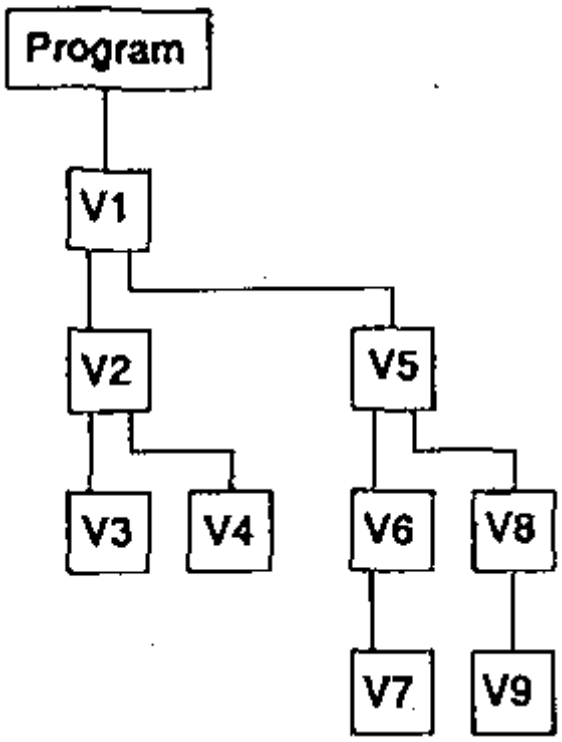
- jaká je funkce programu a jaké služby jsou vyžadovány k zajištění této funkce;
- jaké jsou objekty a jaké služby mohou poskytovat.

To je velká výhoda OOP, nepreferuje ani funkční stránku, ani datovou, preferuje kompromis mezi nimi.

Skrývání informací a odklad rozhodnutí

I když říkáme „děláme program", ve skutečnosti děláme sadu verzí programu. Jednotlivé verze programu se od sebe liší z různých důvodů. Například některá verze nepracuje tak, jak by měla, musíme ji opravit, tj. musíme vytvořit další verzi s lepší funkcí. Také

mohou být jednotlivé verze určeny do různého prostředí (pro různé počítače) nebo nastane změna požadavků na činnost programu. Tak jak se vyvíjí prostředí, v němž program pracuje, tak se má i měnit program. Tvorbu programu si lze představit jako strom složený z jednotlivých verzí. Graficky to lze vyjádřit takto:



Čísla určují pořadí tvorby verzí, spojnice představují práci vynaloženou na tvorbu verze. Verze V3, V4, V7 a V9 jsou konečné (fungující verze), verze V1, V2, V5, V6 a V8 jsou pomocné (rozpracované) verze.

Podívejme se na to z hlediska pracnosti. Spojnice představuje tvorbu nové verze; tj. představuje pracnost. Tvorba první pracující verze programu (verze V3) stála tři činnosti. Tvorba verze V4 z verze V3 stále jednu činnost, kdežto tvorba V7 z V4 stála tři činnosti; tvorba V9 z V7 dvě činnosti.

Snížení pracnosti tvorby programu se dosáhne tím, že nové verze budou vytvářeny co nejníže, co nejblíže pracovním verzím. O čem se vlastně jedná? Při tvorbě programu musíme činit různá rozhodnutí. Tvorba nové verze sestává z toho, že učiníme určitá rozhodnutí a tato realizujeme.

Příklady takových rozhodnutí:

- pracovní data mohou mít formu pole nebo spojového seznamu;
- nejdříve určím akci se záznamem (úprava, přidání, výmaz) a pak příslušný záznam vyhledám; nebo jestli nejdříve záznam vyhledám a pak určím akci.

Nejefektivnější je realizovat co nejpozději (nejníže) **ta rozhodnutí, která budou později měněna.**

Taková formulace není použitelná. Jak mohu při tvorbě programu vědět, co budu potřebovat změnit za rok nebo dva.

Rozhodnutí při tvorbě programu lze zhruba dělit na koncepční (obecná) a na konkrétní. (Příklad: koncepční rozhodnutí je to, že pracovní data tvoří posloupnost; konkrétní rozhodnutí je, že tato data jsou realizována polem, seznamem nebo souborem.)

Je dost zřejmé, že u konkrétního rozhodnutí je větší pravděpodobnost změny. Můžeme tedy formulovat přijatelnější zásadu:

Odkládejme konkrétní rozhodnutí na pozdější etapy návrhu.

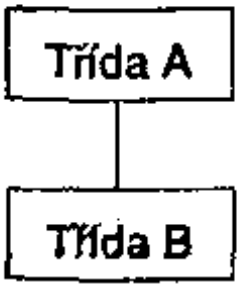
Tuto zásadu zavedl Parnas, i v této formulaci je velmi neurčitá a málo použitelná.

Změnu přineslo OOP. Dědění a pozdní spojování (inheritation, late binding) jsou nástroji, které výborně slouží k odkladu rozhodnutí. Princip postupu je jednoduchý. Ukážeme si to na ilustrativním příkladě hledání v sekvenci prvků.

Zavedeme si třídu objektů A, u ní metodu HLEDEJ, ve které budeme realizovat algoritmus hledání. Pro naše ilustrativní účely zvolíme ten nejprostší algoritmus. Jeho schéma je toto:

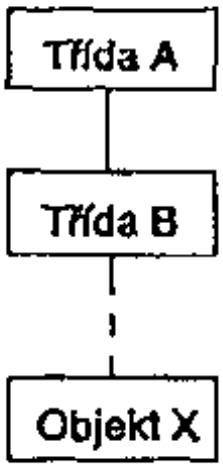
```
ZAČNI_HLEDÁNÍ
while not KONEC_DAT and not NALEZEN do
        POSUN_POSICE;
if not KONEC_DAT then return POSICE
        else CHYBA
```

Na úrovni třídy objektů A realizujeme organizační příkazy, ale skutečné operace s daty odložíme do procedur (metod), které v této fázi nerealizujeme. Ve třídě objektů A určíme pouze organizační příkazy, tj. určíme pořadí provádění funkcí, jak jsou opakovány a podobně. Odložené metody sice musíme také deklarovat, ale nemusíme je podrobně rozepisovat, můžeme je nechat prázdné.



Určíme-li, že třída objektů B je dědicem (potomkem), tak třída B zdědí i všechny metody třídy A, jinými slovy třída B má tytéž metody jako třída A.

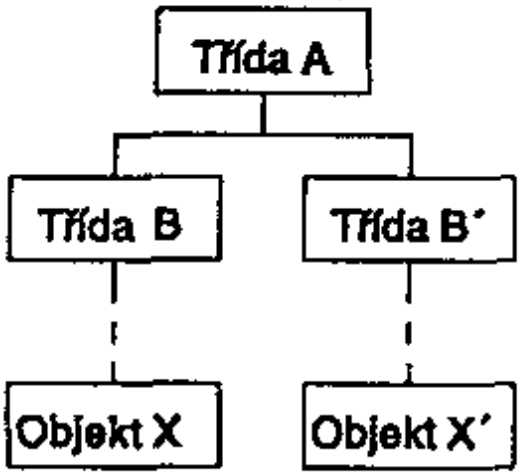
V třídě B však můžeme některé metody přepsat (overriding). V našem případě přepíšeme například metody ZAČNI_HLEDÁNÍ, NALEZEN a podobně. Tyto metody můžeme konkretizovat pro daný typ dat, ve kterých se hledá, například pro hledání v tabulce. Pak například ZAČNI_HLEDÁNÍ pro hledání v tabulce má tělo i=1, tj. dosadí počáteční hodnotu indexu.



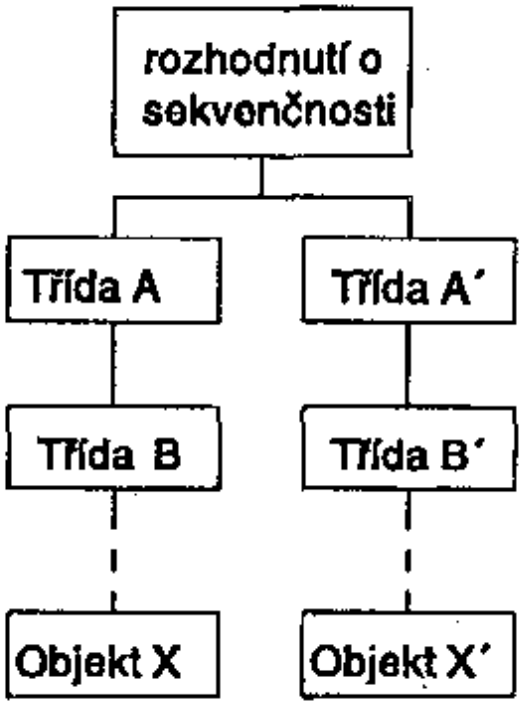
Od třídy objektů B můžeme vytvořit instanci, tj. objekt (nazveme jej X), který zajistí hledání v tabulce.

Vztah dědění vlastností a metod je vyjádřen plnou spojnicí, vztah instance, tj. odvození konkrétního objektu určité třídy je znázorněn tečkovanou čárou.

Dojde-li se po čase k rozhodnutí, že data nemohou být uložena v tabulce, ale musí být uložena v souboru, lze přepsat konkrétní metody třídy B (např. nové ZAČNI_HLEDÁNÍ bude obsahovat funkce **open** a **read**). Podstatné je, že jsme nemuseli měnit metody třídy A. Schematicky to lze vyjádřit takto:



Dědění v OOP má i další příjemnou vlastnost, mohu změnit rozhodnutí na abstraktní úrovni, aniž bych měnil konkrétní rozhodnutí (přesněji řečeno, aniž bych jej příliš měnil). V našem ilustrativním příkladě si to můžeme představit tak, že algoritmus sekvenčního hledání v třídě A změníme na algoritmus hledání půlením. Změní se realizace metody třídy A, málo se změní (přibudou) realizované metody třídy B, nemění se interface objektu X se zbytkem programu (objekt X má stejný interface jako objekt X'. Schematicky to lze vyjádřit takto:



Porovnání skrývání informací a abstraktních dat

Skrývání informací je vyšším stupněm abstrakce. Zatím co v ideji abstraktních dat byl program isolován od fyzické reprezentace dat (resp. byl oddělen algoritmus programu od fyzické reprezentace dat), ve skrývání informací jsou na vyšších úrovních programu (abstraktnějších) skryty algoritmy nižších úrovní. Realizace „abstraktních dat" si lze představit bez zvláštních technických prostředků (bez OOP), „skrývání informací" vyžaduje aparát, který jej umožňuje – a tímto aparátem je OOP.

Dva světy objektů

Všeobecně známé zásady „objektově orientovaného přístupu", tj.

- zapouzdření,
- veřejné a neveřejné služby,
- polymorfismus,
- dědění

jsou dostatečně obecné. Tyto zásady lze použít i jinak, než pouze v rámci programování. Kromě OOP (objektově orientované programování) existuje i OOA (objektově orientovaná analýza), OOD (objektově orientovaný design), ODBMS (objektová databáze). To

jsou již běžné, zavedené zkratky, které označují některé metodiky kolem informačních systémů.

Na celou tuto problematiku se můžeme podívat globálně, ne pouze jako na jednotlivé techniky. Informační systémy jsou určité modely skutečnosti (výseku skutečnosti), které nám umožňují získávat informace, které bychom přímo ze skutečnosti získávali obtížně.

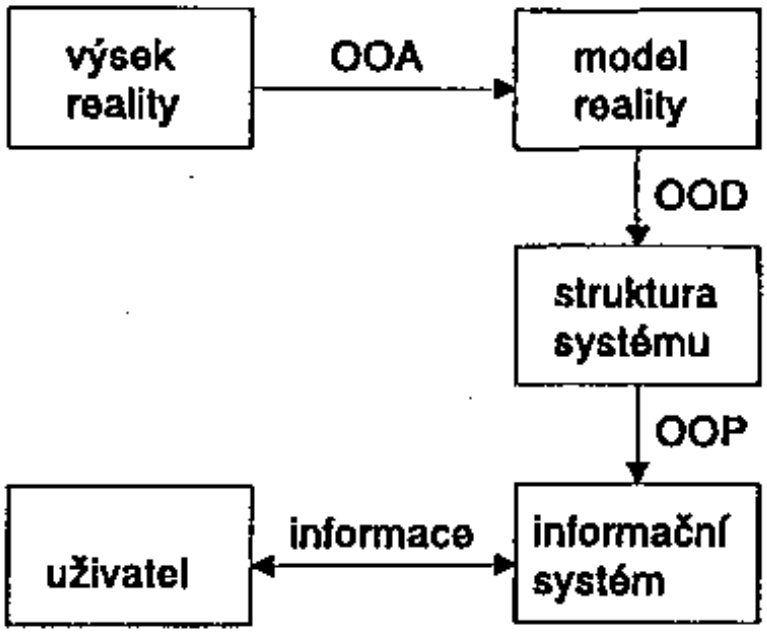
Například je zřetelně jednodušší nechat si vypsat roční produkci z informačního systému než stát celý rok před vrátnicí a počítat expedované výrobky.

Informační systém je modelem skutečnosti. Může to být datový model (v zásadě statický), nebo funkční model (dynamický), nebo, v poslední době, model objektový, který se snaží soustředit výhody obou předchozích. V průběhu jeho vzniku lze použít objektově orientované metodiky asi takto:

OOA – objektově orientovaná analýza.
Lze charakterizovat jako tvorba modelu reality pomocí „objektového" aparátu. Výsledkem je model (esenciální model) reality.

OOD – objektově orientovaný design.
Navrhuje strukturu vytvářeného informačního systému, tj. jeho členění, etapizace ap. V podstatě je výsledkem návod na realizaci systému.

OOP – objektově orientované programování.
Prostředek pro tvorbu programů, tj. pro implementaci informačního systému.



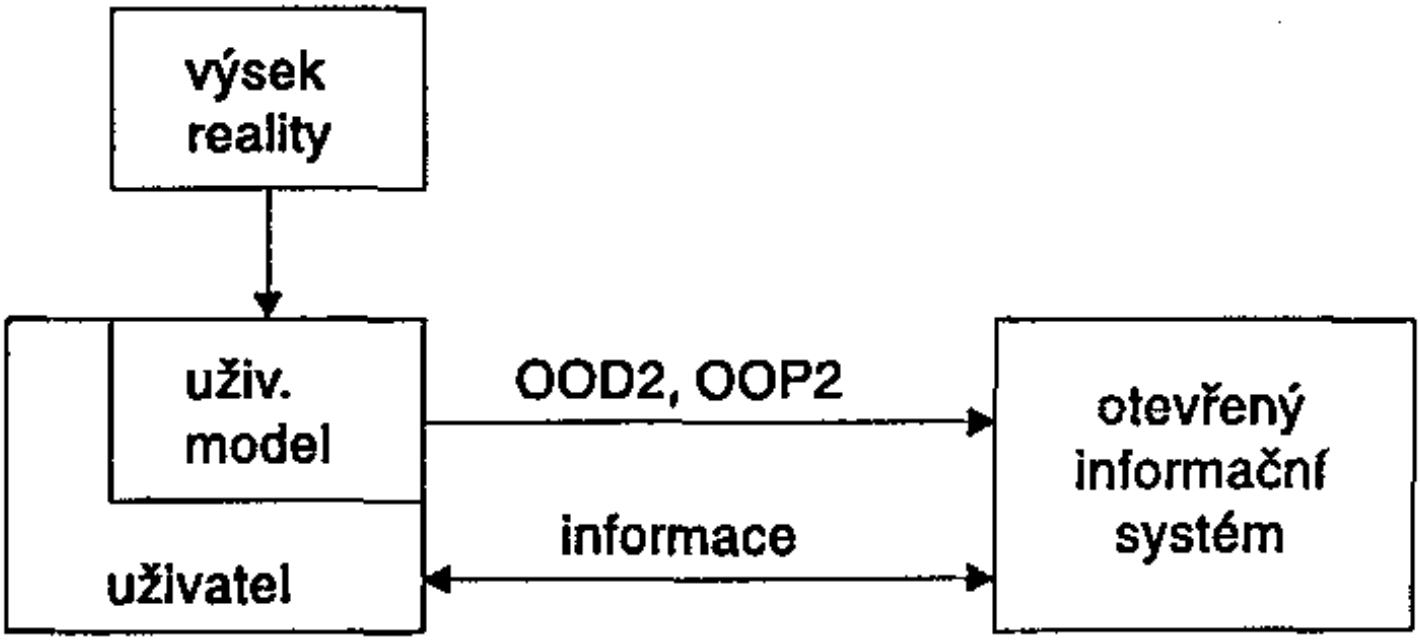
Historicky nejdříve vzniklo OOP, vymysleli jej programátoři pro usnadnění své práce. Záhy se ukázalo, že to je výborný prostředek pro tvorbu programátorských nástrojů. Pomocí OOP jsou vytvořeny významné pomůcky, které programování velmi usnadňují, například Windows, Turbo Vision, Iostream; brzy se však rozvoj zastavil. Ukázal se problém. Není snadné určit, co má být třídou. Ve známém prostředí programovacího jazyka se známými předměty provede programátor toto rozhodnutí intuitivně, proto jsou jako objekty zpracovány soubory, menu, seznamy. V předmětné oblasti (o fakturách, stájích, strojích, pozemcích) nemá programátor dostatek informací a zkušeností. Proto vznikají metodiky OOA a OOD, které mají rozlišit objekty, určit jejich vlastnosti a metody, srovnat to do tříd a tak připravit implementaci systému.

Je nutno poznamenat, že programy vytvořené OOP se ve svých vnějších projevech neliší od programů jinak vytvořených. Z výše uvedeného hlediska jsou metody OOA, OOD a OOP pouze způsobem, jak zefektivnit tvorbu klasických informačních systémů.

Klasické informační systémy mají v sobě zabudovány struktury dat, se kterými pracují. Jsou určeny pro jednorázové použití. Je-li vyžadována změna systému, musí znovu nastoupit specializované týmy na OOA, OOD a OOP a tyto změny realizovat.

Samozřejmým dalším krokem je tvorba otevřených informačních systémů, tj. systémů, které by byly shodné pro řešení jakýchkoliv úloh. Dobré metodické prostředí k tomu poskytují „objekty", tj. mechanismus tříd a metod.

Situace vypadá takto:



Předpokladem takového schématu je otevřený informační systém, který umožňuje uživateli manipulaci s třídami (definice třídy, odvození třídy pomocí dědičnosti, definice metod, ustavení objektů). Uživatel si pak sám vytvoří svůj uživatelský model, sám si jej navrhne a realizuje.

Takové otevřené systémy se vydávají za systémy „bez programování". Mají z velké části pravdu. Tvorba a odvozování tříd jsou spíše logické operace, určování vlastností (tj. atributů či slotů) také. Určitá potíž je s vytvářením metod bez programování. To se řeší výběrem velmi mohutných funkcí, které obhospodařují většinu pravděpodobných situací. Řazení těchto funkcí za sebe (a zajištění výhybek a opakování) se říká jinak než programování (například konstrukce datového toku ap.). Ovšem slušné otevřené systémy umožňují napsat metodu v některém běžném programovacím jazyce, např. v jazyce C.

Podstatou otevřeného informačního systému je to, že **interpretuje** uživatelovy objekty (třídy, metody). Vůbec není podstatné, jak je sám systém naprogramován, jestli objektově nebo ne.

Podíváme-li se na otevřený informační systém s odstupem, vidíme, že to je vlastně interpret nějakého objektově orientovaného jazyka.

Takto lze rozlišit dva světy objektů:

- generované objekty, kdy příslušné pojmy (třídy, metody) existují pouze ve zdrojovém textu programovacího jazyka. Z tohoto zdrojového textu se vygeneruje program, který už pojem „objekt" nezná.
- interpretované objekty, kdy příslušné pojmy (třídy, metody) žijí před očima uživatele. Jejich „život" je zajišťován interpretací příslušných datových struktur.

Zdá se, že interpretované objekty (otevřené informační systémy) mají před sebou budoucnost, vzhledem k malým zkušenostem s jejich použitím je na nějaké hodnocení ještě brzy.

Autor: RNDr. Pavel Drbal, CSc.
VŠE, katedra IT,
nám. W.Churchilla 4,
13000 Pha 3
tel: 2125437