Jak objektově programovat

Jiří Polák

Úvod

Objektově orientované programování je dnes velmi populární, alespoň v časopisech o výpočetní technice. Patří ke koloritu dnešní doby i mezi výrobci programových produktů; poněkud horší to je se znalostí toho, jak objektově orientované programování používat. Což je také způsobeno tím, že se moc neví, co to objektově orientované progranování vůbec je. A proto než si ukážeme na příkladu, jak objektově programovat, musíme si vysvětllit základy objektově orientovaného programování.

Objektově orientované programování je jedním z nejprogresívnějších směrů v oblasti programování, návrhu a realizace programových produktů. Objektově orientovaný přístup se ale netýká jen vlastního procesu programování, ale i např. reprezentace světa v počítači, nebo organizace paměti.

V oblasti programování se jedná o styl, přesněji řečeno o metodiku a přístup, který je slučitelný jak s imperativním (procedurálním), tak i funkcionálním či logickým vzorem/modelem (angl. paradigma) programování.

Objektové architektury počítačů nemění architekturu stroje zásadním způsobem, mění jen jednu část architektury, totiž **paměť**, která se ze zřejmých důvodů vyskytuje ve všech architekturách stroje. A proto lze hovořit o objektově orientovaných imperativních modelech programovacích jazycích, atp.), o objektově orientovaných data-flow architekturách, atp. - nelze tedy objektově orientovanou architekturu/paradigma/ přístup/styl /model klást na stejnou úroveň jako non von Neumannovské architektury stroje, nejedná se totiž o tak zásadní změnu.

Objektově orientovaný přístup se snaží řešit snad hlavní problém von Neumannovy architektury, to je **problém primitivnosti paměti,** v níž nerozeznáme, co paměťová buňka obsahuje a můžeme to libovolným způsobem interpretovat. Objektově orientované modely paměti se vyznačují sebeidentifikací dat v paměti s lokálními účinky změn. Hovoří se také o **sémanticky strukturované paměti.** Objektová paměť tedy obsahuje entity - říkáme jim objekty, které si nesou i identifikaci svého obsahu a které lokalizují do svého nitra změny v paměti.

Základem objektových systémů je tedy **OBJEKT.** Objektová paměť tedy obsahuje objekty, objektově orientované programování pracuje s objekty. K podrobnějšímu popisu objektu se dostaneme později, zde si jen uveďme, že objekt je **neděli-**

telná entita obsahující data, jejich identifikaci a možnosti svého použití (přípustných operací).

Dosud jsme hovořili o objektově orientovaném přístupu na abstraktní úrovni vzdálené od konkrétních progranovacích prostředků. Při aplikaci objektově orientovaných modelů do úrovně přímo ovlivňující programování se lze setkat se dvěma možnostmi, se dvěma hlavními modely objektů:

1. model objektů jako chráněných (pasivních) dat (Modula, CLU, Alphard, Ada, Hydra, StarOS, iAPX 432) - **statický model pasivních objektů**
2. model objektů jako aktivních komponent výpočtu (Simula, Smalltalk, Loops, Actor) - **dynamický model aktivních objektů**

V historii softwarového inženýrství (tedy toho, jak pořádně programovat) sehrála velkou roli problematika datových typů. Vývojem se dospělo k tzv. abstraktnímu datovému typu (ADT), jehož příklad si uvedeme. O ADT hovoříme proto, že objektové programování v něm má své kořeny, a proto že objektově orientované programování ADT používá.

Abstraktní datový typ (ADT) je definován množinou hodnot, kterých může nabýt, a množinou operací, které jsou nad ním definovány. Specifikace ADT však nic neříká o tom, jak jsou vlastní data a operace nad nimi implementovány. Proto abstraktní typy dat plně uskutečňují to, čemu říkáme ukrývání vnitřní informace. Specifikace abstraktních datových typů se sestává ze specifikace jejich syntaxe a sémantiky. Ke specifikaci syntaxe se často používá algebraických zápisů v textové nebo grafické podobě, sémantiku lze formálně popisovat soustavou axiomů. Specifikace abstraktního datového typu 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) + > TE

Předběžné podmínky

top (STACK: TSTACK(TE)) = (not empty (STACK)

Axiomy

Pro všechny E : TE, STACK : TSTACK musí platit:

empty(new(STACK))

not empty (push(E,STACK))

top(push(E,STACK)) = E

pop(push(E,STACK)) = STACK
. konec specifikace

Parametr TE je typ elementů zásobníku a jeho uvedení v hranaté závorce znamená, že daný abstraktní typ dat je na něm nezávislý, může být zásobníkem položek libovolného typu. Tato vlastnost datového typu bývá označována jako generičnost. Symbol "+>" v algebraické specifikaci operace top znamená, že jde o parciální operaci, která nemusí být definována nad každou instancí typu TSTACK. Jak víme, operace top není definována pro prázdný zásobník.

Stvořitelské operaci, která nemá na levé straně šipky žádný operátor, se také říká generátor nebo konstruktor. Protože tato operace inicializuje práci s instancí datového typu, bývá její jméno často odvozeno od slova "inicializace".

Pro pojem **abstraktně definovaný datový typ** se v programovacích jazycích používají různé názvy (angl. module, cluster, package, class). My budeme používat zejména pojem **třída**. Objekt je zde definován jako exemplář/instance třídy a lze jej považovat za vnitřně strukturovanou hodnotu, jejíž vlastnosti jsou definovány třídou. Pojmy třída a objekt lze považovat za zobecnění běžně známých pojmů typ a hodnota. Programovací jazyk, jehož sémantika je založena na pojmech třída a objekt označíme jako objektově orientovaný vyšší programovací jazyk a programování v něm si lze představit jako vytváření nebo obohacování programového prostředí tvořeného jednotlivými uživatelsky definovanými třídami, ve kterém běží jejich vlastní výpočet. Objekty mohou být v takovém světě buď aktivní nebo pasivní, což vede ke dvěma různým (výše zmíněným) modelům, o nichž dále podrobněji pohovoříme.

Objektový model s dynamickou sémantikou objektů vyžaduje, aby v době výpočtu byly v paměti reprezentovány všechny abstraktní informace o třídách a objektech. Dynamickou sémantiku objektového modelu pak musí zajistit vrstva programového vybavení ve spolupráci s architekturou stroje. Dynamický objektový model má zhruba **tři hlavní přednosti:**

- adaptibilita a pružnost: je možné měnit programové prostředí za běhu. Statistický model vyžaduje rekompilaci celého prostředí, změní-li se definice jedné třídy, nové sestavení programu apod. Zde stačí rekompilovat pouze změněnou (nebo zkompilovat nově přidanou) definici třídy a zbytek prostředí zůstává nezměněn (dokonce v něm může zatím probíhat výpočet).
- polymorfismus: dané jméno objektu lze po dobu platnosti jeho deklarace vázat na různé instance různých tříd. Kromě toho lze zprávu téhož jména posílat různým objektům. "Typová kontrola" se posune do doby výpočtu. To umožňuje kontrolovat všechny sémantické chyby. Navíc lze psát univerzálnější (generické) programy, neboť program není fixován na jednu reprezentaci objektu.

- inkrementální vývoj programového vybavení (prostředí): programový celek lze ladit po částech, používat vždy pouze určitou část prostředí, kdy zbytek nemusí být ještě odladěn nebo dokonce naprogramován. To je zvlášť výhodné pro začátečníky a při tvorbě rozsáhlých programů.

Dynamický objektový model má však i svoje nevýhody:

- nečitelnost: význam daného jména (ať objektu či operace) lze určit až v době výpočtu a může se dynamicky měnit. To zhoršuje čitelnost a srozumitelnost programů. "Typovou chybu" je možno odhalit až v době výpočtu. Tyto problémy lze zmírnit používáním výmluvných jmen a komentářů a jsou činěny pokusy doplnit objektový model mechanismem, který dokáže vydedukovat pro každé deklarované jméno množinu všech tříd, na jejichž instance může být při výpočtu vázáno.
- neúplnost: protože stav prostředí lze dynamicky měnit, je obtížnější zabezpečit, aby prostředí bylo úplné.
- nízká efektivita výpočtu: většina sémantických akcí je odložena na dobu výpočtu a to vede při implementaci na klasické von Neumannově architektuře, kde je nutné model implementovat programově, k poklesu efektivity.

ÚMLUVA: Dohodněme se, ve shodě s převažujícím chápáním pojmů objektový a objektově orientovaný ve světě, že nadále budeme za objektový či objektově orientovaný považovat jen takový systém/model/ přístup, který je v souladu s dynamickým modelem objektů.

1. Objektově orientované programování

Objektové programování či objektová technologie tvorby programů modernizuje a mění způsob návrhu, vývoje a udržování programových produktů. Základní myšlenka objektového programování spočívá v tom, že data a procedury jsou společně reprezentovány ve struktuře nazývané objekt. Příslušná data jsou přístupná pouze pomocí procedur z odpovídajícího objektu. Síla objektového přístupu je v mnohonásobné použitelnosti jednou vytvořeného kódu a ve snažší udržovatelnosti programových systémů. Objektové programování navazuje na myšlenky strukturovaného a modulárního programování a dále je rozvíjí.

Objektový přístup k technologii programování se vyznačuje následujícími rysy:

- **modifikovatelnost**: měnit existující programový systém je možné bez toho, že by se znovu celý nový systém kompiloval a sestavoval, mění se/přidává se jen ta část kódu, která je pro modifikaci nutná, často je možno systém měnit/doplňovat i za jeho běhu,

- **násobná použitelnost**: jako existují knihovny procedur a funkcí např. pro Fortran, tak se vytvářejí katalogy tříd, které jsou však obecnější, neboť se nejedná jen o jednotlivé operace, ale o celé datové typy se všemi operacemi,
- **integrovanost**: objektové programovací prostředky tvoří integrovaný systém dovolující používat objekty (jeden a týž objekt) ve více úlohách, celý integrovaný systém je uživateli přístupný a dovoluje mu uniformním způsobem pracovat i s operačním systémem.

Objektový přístup vyžaduje nové způsoby myšlení o tvorbě programových systémů, proto si tyto odlišnosti, a hlavně nové pojmy, nejprve vysvětlíme.

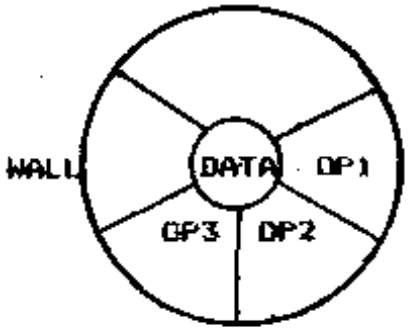
1.2 Základní pojmy objektově orientovaného programování

Objekt, zapouzdření a posílání zpráv

V procedurálně založených systémech (Algol, Fortran, Cobol, Pascal) se pracuje s globálními daty, s globálními datovými strukturami. Procedury a funkce pak pracují s těmito globálními daty, což přináší velké nebezpečí neplánované či nežádané změny globálních a tedy nechráněných dat. Snadno může být např. ve Fortranu celé číslo interpretováno jako hodnota jiného datového typu.Takovým změnám/záměnám se předchází zaváděním typů a následnou typovou kontrolou. V těchto jazycích (např. Pascal) je třeba při překonávání kontrol typů postupovat rafinovaněji, ale téměř vždy je možné je nějak obejít.

Při objektovém přístupu se systém vytváří z jednotlivých objektů. **Objekt** obsahuje lokální datové struktury a lokální procedury, které jediné zabezpečují přístup k datům. Říkáme, že data jsou **zapouzdřena** (angl. encapsulated). Ochrana dat je plně zabezpečena, protože bez "souhlasu" objektu nejsou jeho data zpřístupněna. Objekt je zde aktivní komponentou výpočtu.

Na obr. 1.1 vidíme objekt jako spojení dat a zdi (wall), která tato data chrání.



Obr.1.1 Objekt

Ochranná zeď nejenže chrání data, ale také je identifikuje. To znamená, že nám říká, o jaký údaj se jedná.

Samozřejmě záleží na tom, jak bezpečně jsou objekty vytvořeny, jak bezpečný je program. I zde platí, že objektové programovací jazyky svými prostředky podporují a umožňují správné psaní programů zachovávajících ochranu dat, ale je nutné prostředků využívat a ne je obcházet.

Zapouzdření dat tedy slouží i k jejich identifikaci, objekt ví, jaká data obsahuje - zda to je např. číslo, pravdivostní hodnota, či množina hodnot. Identifikaci objektů je možné v programu používat a při běhu programu ji kdykoliv zjišťovat. Výhody zapouzdření jsou v tom, že:

- programy mohou být testovány po menších částech, častá chyba při procedurálním programování je skryta v přístupu ke sdíleným datům, čemuž zapouzdření metodicky předchází,
- interní struktura dat může být měněna bez nutnosti změn okolí objektů,
- mohou být vytvářeny katalogy či knihovny objektů, vlastně tedy abstraktních datových typů, které je možné v jiných aplikacích volně použít,
- je zabezpečena ochrana a identifikace dat, se kterými program pracuje.

Objektově orientované programovací systémy se od sebe odlišují také v tom, jak velké entity jsou považovány za objekty. Hovoří se v této souvislosti o **granularitě objektů**. Pokud se za objekty považují jen určité datové struktury, např. záznamy, pole, pak je označujeme jako systémy s velkou granularitou (angl. large-grain object systems); jestliže jsou v systému i znaky a čísla považována za objekty, pak je označujeme jako systémy s jemnou granularitou (angl. fine-grain object systems).

Objekty komunikují s jinými objekty pomocí **posílání zpráv** (angl. sending messages). Přípustnými zprávami pro objekt je definováno jeho **rozhraní**. Množina zpráv, na které objekt reaguje, se někdy nazývá **protokol**. Budeme také hovořit o vlastnostech objektu jako o operacích, které je možné zprávami vyvolat.

Mezi objekty se posílají **zprávy**, které se typicky skládají z adresáta neboli příjemce zprávy, tedy určitého objektu, a tzv. selektoru zprávy (angl. message selector) s případnými argumenty. **Selektor zprávy** odpovídá jménu procedury/funkce v klasickém programování. **Metoda** (angl. method) je popis jak provést jednu z operací objektu, tedy vlastně část programu, podprogram. Hlavička matody se nazývá **vzor metody** (angl. method pattern) a odpovídá hlavičce procedury/funkce obsahující jméno a formální parametry.

"Tiskárna tisk: co jak: kurzíva" je příkladem poslání zprávy "tisk: co jak: kurzíva" objektu "Tiskárna", "tisk: jak:" je selektor zprávy a "co, kurzíva" jsou argumenty. Vzor metody je "tisk: kusTextu jak: popisStylu".

Data, která jsou v objektu zapouzdřena, je možné změnit pouze tak, že objektu pošleme zprávu, které rozumí. Zprávy, kterým objekt rozumí, jsou ve formě kódu metod obsaženy ve zdi, která data chrání.

V modelu posílání zpráv se skrývá fundamentální odlišnost od klasického imperativního, procedurálního programování: kód operace mající provést vyžadovanou akci je klasicky určen hned při jejím vyvolání (např. je tedy znám v době překladu programu), zatímco v objektovém přístupu záleží výběr akce - metody - na příjemci zprávy, tedy na určitém objektu (je tedy známa až při běhu programu). Nikdy není žádná metoda přímo vyvolána, vždy probíhá hledání příslušné metody, což může být díky dědění relativně složitý proces. Z toho ovšem vyplývá, že tatáž zpráva zaslaná různým objektům má různý efekt, ale to lépe popisuje vlastnost nazývaná polymorfismus.

Polymorfismus

Polymorfismus je obecně vlastnost, díky níž totéž jméno (např. proměnných, funkcí, procedur) může mít více významů. **Polymorfismus** v objektovém programování znamená, že ta samá zpráva může být poslána rozličným objektům bez toho, že by nás při jejím poslání zajímala implementace odpovídajících metod a datových struktur objektu (příjemce zprávy), každý objekt může reagovat na poslanou zprávu po svém.

Ta samá zpráva může tedy u různých objektů vyvolat různé reakce. Je-li metoda ukažSe obsažena v protokolu objektů **Funkce, Text, Místnost** a **Postel**, pak poslání zprávy **ukažSe** uvedeným objektům způsobí např. postupně podle příjemce zprávy zobrazení grafu funkce, vypsání textu, vykreslení půdorysu místnosti či uvedení jména/jmen aktuálních uživatelů postele. Není tedy třeba dbát na různost selektorů/vzorů metod pro významově podobné akce jen proto, že jsou prováděny s různými daty, jak je tomu se jmény procedur/funkcí v klasických procedurálních jazycích.

V souvislosti s posíláním zpráv se setkáváme ještě s tzv. **pozdní**, resp. **brzkou vazbou kódu** (angl. late resp. early binding). Chápe se tím doba, kdy je znám kód operace, kterou se provede činnost způsobená správným posláním zprávy (činnost způsobená voláním procedury/funkce). V dynamických objektových systémech se setkáváme s pozdní vazbou - kód operace je určen až za běhu programu, v okamžiku, kdy objekt začne provádět vyžádanou činnost. Při statickém chápání je naopak kód operace znám již v době překladu programu.

Další význačnou vlastností objektových systémů je dědičnost. Odlišuje objektově orientované programování od modulárního programování, jak je třeba používáno v jazycích Modula a Ada.

Dědičnost

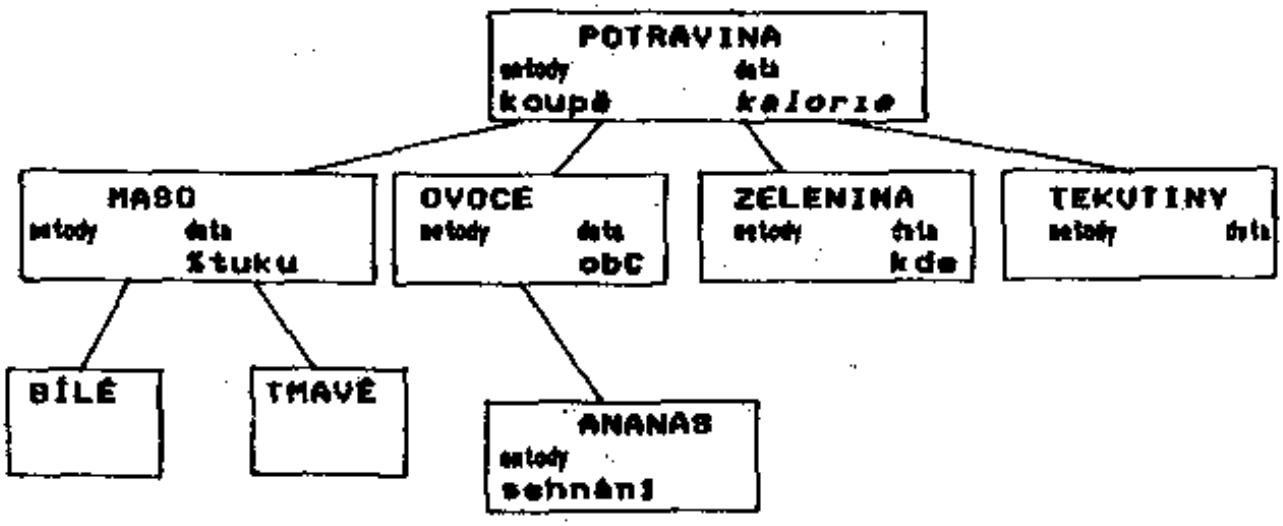
Objekty mohou **dědit** (angl. inherit) vlastnosti, tj. strukturu lokálních dat a možných operací s nimi, od jim nadřazených objektů. Vzniká tak hierarchie objektů,

přičemž nejobecnější objekty mají jen málo společných vlastností a ty, které jsou v hierarchii níže, mohou být velmi jemně (postupně) odlišovány od obecných, týká se to zejména přípustných operací s daty.

Příkladem jsou potraviny a jejich hierarchie (viz obr. 1.2), nejvýše je nejobecnější objekt - **potravina**. Ten má pro jednoduchost jen jediný lokální údaj obsahující kalorickou hodnotu potraviny a jedinou operaci **koupě** oné potraviny. Níže jsou v hierarchii: **maso, ovoce, zelenina, tekutiny**, máso se dělí na **bílé** a **tmavé**. Pro masa má význam udávání množství tuku, pro ovoce obsah vitamínu C a pro zeleninu místo vypěstění. Všechny potraviny, ale v této hierarchii dědí operaci **koupě** a udávání kalorické hodnoty (lépe energie v kJ).

Objektové systémy obvykle umožňují, aby se dědění některých vlastností dalo pro nižší vrstvy hierarchie odstínit, např. pro **ananasy** se v našich podmínkách operace **koupě** mohla dříve odstínit operací **sehnání**.

V objektových systémech se lze také setkat s násobnou dědičností, kdy objekt smí dědit vlastnosti od více různých objektů najednou, či s prostou dědičností, kdy dědí jen od jednoho nadřazeného objektu - samozřejmě i zde není omezen počet "synovských /dceřiných" objektů.



Obrázek 1.2 *Dědění*

Výhody dědičnosti jsou:

- struktura dědění a hierarchie objektů dobře odpovídá mnoha skutečným pojmoslovným a přírodním či společenským strukturám, které je nutné často reprezentovat v počítači, což se využívá zejména v oblasti reprezentace znalostí,
- při programování není třeba opakovat mnoho kódu, neboť odlišení datových struktur je velmi jemné.

Třída

Pro čistě **objektové programování** bychom vystačili s pojmy objekt a posílání zpráv. Je to minimum, které musí každý objektový systém obsahovat. V takovémto modelu objektového programování je ale skryta velká neefektivnost, zejména velké paměťové nároky. Vždyť bychom s každými daty museli uchovávat všechny informace o nich a o operacích pro ně přípustné, resp. o zprávách, které jim můžeme poslat a kterým budou rozumět. Každý jednotlivý objekt má v této podobě velké paměťové nároky. Jistým kompromisem může být čistě objektový systém s velkou granularitou objektů.

Většina objektových modelů však používá třídu jako prostředek pro tvoření a společný popis objektů identických vlastností (tj. strukturu lokálních dat a možných oprerací s nimi). Uspokojivá definice třídy byla již uvedena v části věnované modelu abstraktních typů. **Třída** (angl. class) tak šetří mnoho paměťových nároků, neboť operace, lépe metody, jsou uloženy a zpřístupňovány v jednom místě společně pro více objektů. Např. v situaci, kdy potřebujeme pracovat s místnostmi a dveřmi, je výhodné místo stovek objektů dveří s udáním jejich rozměrů, s udáním odkud kam vedou a s operací jejich zavření/otevření pracovat s jednou třídou všech dveří. Říkáme pak, že jednotlivé dveře jsou **instancemi** třídy dveře.

Třída také v sobě obsahuje informace o lokálních proměnných instancí a o proměnných globálních pro všechny objekty, pro všechny instance té které třídy.
Popis/definice třídy obsahuje:

- jméno třídy s uvedením nadřazené třídy v hierarchii,
- deklaraci jmen lokálních proměnných instancí,
- deklaraci jmen proměnných globálních neboli sdílených všemi instancemi dané třídy,
- metody použitelné instancemi při reagování na zaslané zprávy.

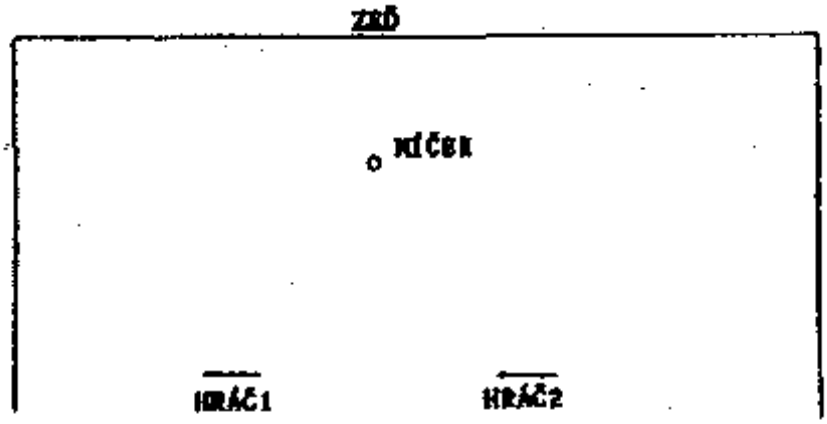
Třídy lze pak chápat také jako jisté továrny na objekty, neboť na vyžádání dovedou vytvořit nový objekt, svoji instanci. Třída dveří tedy umí vytvořit nový objekt určitých dveří. Hierarchie tříd je vzhledem k jejich menšímu počtu přehlednější než hierarchie všech objektů, nicméně i zde je třeba hierarchii vytvářet velmi plánovitě.

Častou chybou při tvorbě prvního programu v objektově orientovaném programovacím systému je hledání jedné, všeobsahující hierarchie tříd spojeným mezi sebou vztahy dědičnosti. Mezi třídami neexistuje pouze vztah dědičnosti, ale také vztah **Pán-Sluha**, někdy se také označuje odběratel - dodavatel či kupec-prodejce. Dědění je totiž někdy nepříjemné, protože nás nutí zdědit i něco, co se nehodí, co je vhodné spíše potlačit. Zde je výhodnější použít jen něco z toho, co třída nabízí. Prakticky se to realizuje tak, že instance **Sluhy** se použije pro určité činnosti **Pána**.

Konkrétní příklad dědičnosti a vztahu Pán-Sluha si ukážeme v následujícím příkladu.

2. Příklad návrhu objektově orientovaného programu

Rozdíly mezi tradičním (tj. procedurálním, modulárním) a objektově orientovaným programováním si ukážeme na příkladu videohry dle obr. 2.1. Vidíme dva hráče, jeden míček a odrazovou zeď. Úkolem je vytvořit program pro hru, přičemž hráči se pohybují jen vlevo a vpravo a mají se střídat při úderech do míčku, který se odráží od zdí, pokud míček propadne "dolů", pak bod získává ten hráč, který jako poslední do míče udeřil.



Obrázek 2.1 *Hra na obrazovce.*

Zvolme nejprve tradiční postup popisu chování programu (tj. jeho funkcí) a metodu postupného zjemňování shora dolů. Na nejvyšší úrovni abstrakce (1. krok) se jedná o jediný příkaz/modul:

1. krok - Video hra.

Další zjemňování vede na cyklus:

2. krok - Dokud chce někdo hrát, tak opakuj
- inicializace hry
- řízení jedné hry.

Ve třetím kroku zjemňování shora dolů se dostáváme k dalším detailům

3. krok - inicializace hry
- kresba zdí
- pevně zvolená výchozí místa hráčů
- připrav polohu, směr a rychlost míčku, ale nespusť a neukazuj ho

* řízení jedné hry
  - nastav skore 0:0
  - pokud není konec hry opakuj:
    -- hoď míček
    -- hrej míček
    -- dokud není míček mimo opakuj
      --- jednomu z hráčů přičti bod
      --- zaznamenej výsledek

Další krok již bude obsahovat řadu zjemnění jednotlivých úloh uvedených v předchozím kroku. Ukažme si jen jeden malý detail pro hru míčku.

hrej míček

-jaká pozice míčku	-na zdi	- proveď odraz
	-u hráče 1	- proveď odraz a zapamatuj Hráče 1
	-u hráče 2	- proveď odraz a zapamatuj Hráče 2
	-na boční stěně	- proveď odraz
	-v prostoru	- pokračuj v pohybu míčku

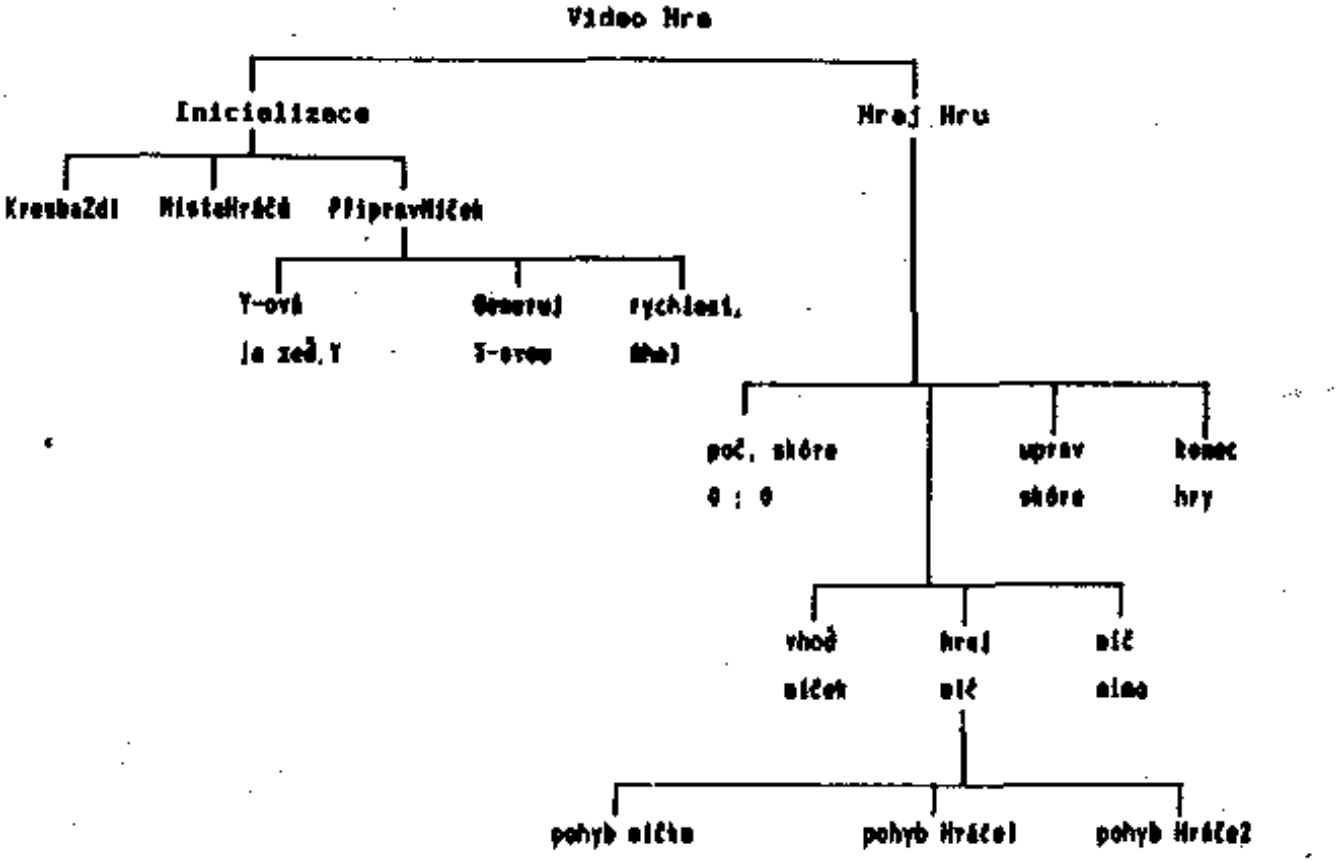
pohyb Hráče 1

- odečti údaje o vyžadovaném pohybu (vlevo, vpravo, rychlost)
- u zdi a směr zeď - stát
- vedle Hráče2 a směr na místo Hráče2 -Hráč2 chce za Hráče1 - výměna
  -Hráč2 stojí - za něj.

Abychom se ještě více přiblížili konkrétnímu programovacímu jazyku, ukažme si jinak zapsaný popis pohybu Hráče1.

```
procedure Pohyb_Hráče (Hráč1 : hráč)
   case Hráč1.souřadnice of
      u_levé_zdi:  if chce_vlevo              then stát
                                              else pohyb,
      u_pravé_zdi: if chce_vpravo             then stát
                                              else pohyb,
      vedle_Hráče2: if chce_na_Hráč2          then
                                              if Hráč2_stojí    then za_něj
                                                                else výměna,
      otherwise:pohyb,
end;
```

Je tedy možné znázornit si celou hru/program vytvářený tradičním způsobem pomocí následujícího schématu (obr. 2.2), který popisuje především funkce/chování jednotlivých částí programu.



Obrázek 2.2 Funkční schéma programu.

Je zřejmé jak postupujeme, když klasickým způsobem popisujeme funkce programu, u popisu programu pohyb_Hráče1 jsme se dostali téměř na úroveň příkazů programovacího jazyka. Objektově orientovaný přístup volí jiný postup. Již jsme si uvedli jeho základní pojmy, a tak nás nepřekvapí, že začneme od hledání tříd, jejich vzájemných vztahů (dědění, pán-sluha) a nesmíme zapomenout na popis jejich metod.

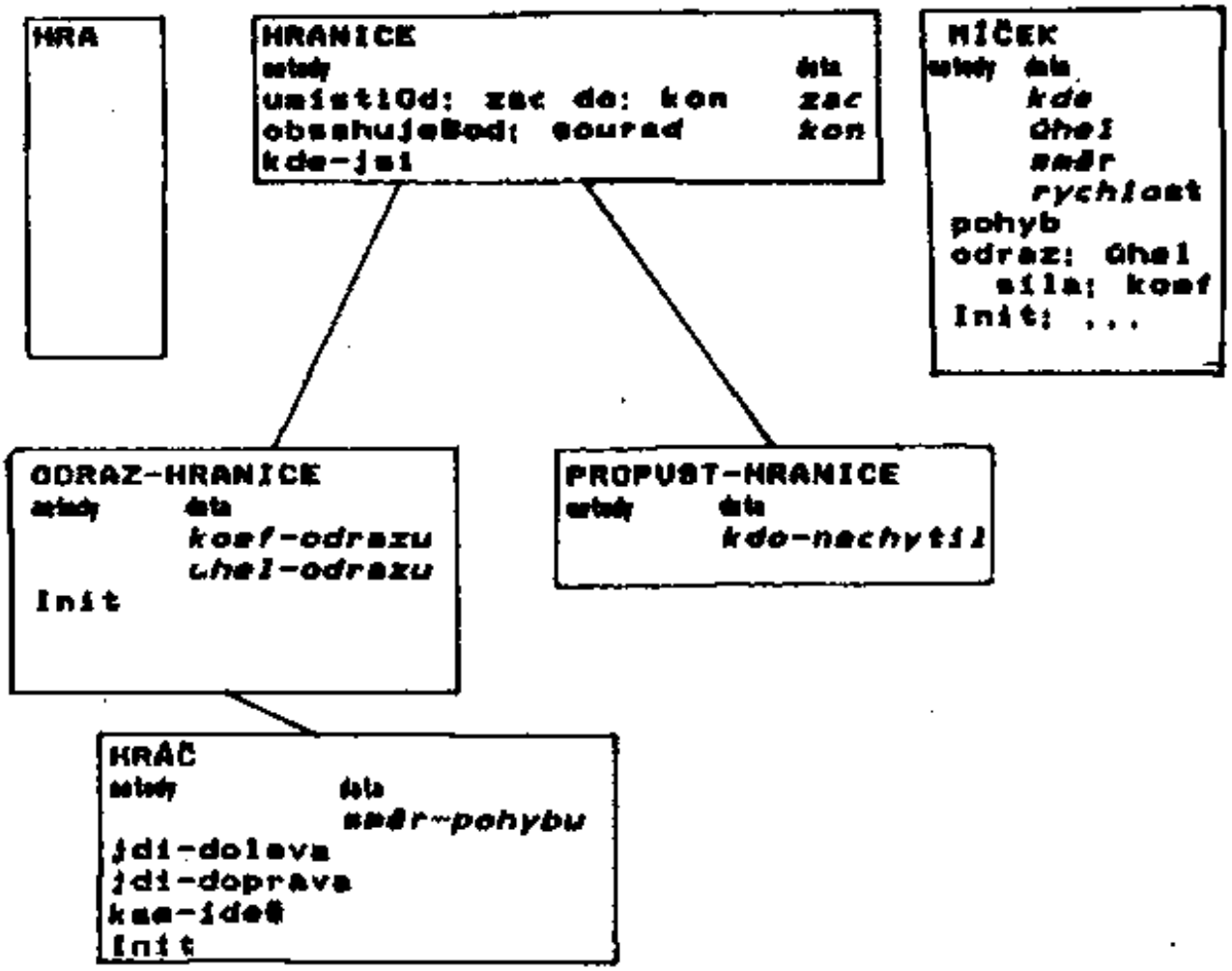
Celá hra by mohla být jedna třída, která by jako své sluhy používala třídy **Hráč**, **Míč**, **Stěny**, **Prostor** a **Mimo**. Mezi některými z nich budou vládnout vztahy dědění, u jiných (např. Hráč) bude celá hra používat i několika instancí zároveň.

Na obr. 2.3 jsou schématicky znázorněny použité třídy, lokální data jejich instalací a metody, kterým instance rozumí.

HRA je třída, která obsahuje instance všech ostatních tříd a zodpovídá za řízení celé hry. Bude tedy obsahovat instanci třídy MÍČEK, která obsahuje informace o

svém umístění, charakteristikách pohybu a informaci o hráči, který se poslední míče dotknul. Kromě toho jsou zde instance ohraničení hracího pole a povšimněme si, že i samotní hráči jsou jen instance podtřídy třídy HRANICE. Bylo by možné pro hráče vytvořit jejich vlastní (nezávislou) třídu, ale protože dědí tolik společných vlastností od odrazových stěn, považujeme za vhodné takové dědění zvýraznit. Spíše z metodických důvodů jsou hranice, od kterých se míček odráží, rozděleny na dvě třídy - chceme tím ukázat význam globálních údajů pro instance, zde koeficientu síly odrazu. Hloubavému čtenáři zde nabízíme zajímavý úkol: Pozorujte, jak by se změnil pro-gram, kdybychom chtěli hrát zároveň s více míčky v obou přístupech, pokuste se o určité porovnání.

Zůstali jsme, u objektově orientovaného přístupu, něco dlužni naznačení řízení celého programu. Tedy chybí uvedení lokálních dat instancí třídy HRA a jejich metod. Data jsme slovně popsali v předchozích odstavcích, ale neuvedli jsme údaje týkající se skore hry.



Obrázek 2.5 Schéma tříd programu.

Nyní je snad zřejmé, že při programování s objektově orientovaných programovacích jazyků je třeba postupovat odlišně od funkčního popisu úlohy. I oblíbená metoda shora dolů vyžaduje určité modifikace. Hlavním úkolem je nalézt třídy (jejich data a metody) a zejména pak dobře zvolit vztahy mezi jednotlivými třídami. První úkol, tj. hledání tříd, lze podpořit radou: za třídu zvolte to, co je datovým typem, co je konkrétní či abstraktní věcí v popisu úlohy. Pro hledání vztahů mezi třídami může pomoci následující rada: Personifikujte si do počítače ty věci (entity, objekty, údaje, ...), které se vyskytují ve vašem programu (tedy třídy). Pracuje-li váš program s potravinami, pak potraviny musí v našem/vašem programu ožít a mít všechny ty vlastnosti a schopnosti, jaké mají ve skutečném světě. A podle vztahů v reálném či myšleném světě volme i vztahy v programu. (nebudeme zde zabíhat do detailů, který svět je skutečný a který jen myšlený.)

3. Závěr

Jednoduchou odpovědí na otázku obsaženou v názvu našeho příspěvku může být:

J i n a k !

Taková odpověď však příliš nepomáhá programátorovi. A proto si shrňme, že zatímco klasicky se popisují, zjemňují funkce programu, tak objektově orientovaný přístup hledá a zjemňuje třídy a jejich vlastnosti (lokální data, možné operace). Neméně důležité je nalezení správných vztahů mezi třídami. A k tomu je nutné vědět, co to je třída a co to je objekt.

Literatura

[1] Polák J.: Objektově orientované programování, učební text, Ediční středisko ČVUT Praha, 1991, 156 stran, ISBN 80-01-00544-2.

[2] Polák J.: Turbo Pascal v.6.0 a objektově orientované programování, GComp, Praha, 1991.

Autor: Ing. Jiří P o l á k
Kat. počítačů, FEL ČVUT,
121 35 Praha 2
tel. 02 - 297841 kl.311