ZÁKLADNÍ PRINCIPY OBJEKTOVĚ ORIENTOVANÉHO PARADIGMATU

Jiří Polák

Motto: *Mluvte čirosceně*

L.Rosten, Pan Kaplan má třídu rád

ÚVOD

Naším cílem je podat trošku obecnější nadhled nad celou problematikou objektově orientovaného přístupu. Věříme, že naše informace budou užitečné po delší dobu a nestanou se zbytečnými hned, jak se na trhu objeví nový produkt. Proto se také budeme jen v nejnutnější míře zmiňovat o dnes používaných produktech s cílem vysvětlit jejich používání, protože takové informace zastarávají nejrychleji.

JAK SE DÍVÁME NA OOP

Objektově orientované programování je jedním z nejprogresivnějších směrů v oblasti programování, návrhu a realizace programových produktů. **Objektově orientovaný přístup** (angl. object–oriented paradigm) se netýká jen vlastního procesu programování, ale i např. reprezentace světa v počítači, metodiky řešení celého problému (tj. analýzy problému, návrhu řešení úlohy a implementace úlohy), 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 programováním.

K výkladu, učení či ilustraci principů objektově orientovaného programování lze dnes použít několika programovacích systémů/jazyků, které o sobě prohlašují, že jsou objektově orientované. Ale stejně jako je lepší vysvětlovat principy a používání strukturovaného programování na příkladech v jazyce Pascal či Modula, které svými konstrukcemi strukturované programování podporují – a ne např. v jazyce Basic (i když i to je možné), kde takové konstrukce nejsou – je podle nás lepší vysvětlovat principy a techniky objektově orientovaného programování s použitím jazyka, který svými konstrukcemi plně podporuje objektově orientované programování. Proto jsme se rozhodli pro jazyk blízký jazyku Smalltalk80, který uvedený požadavek splňuje, což nelze říci např. o implementaci jazyka

Pascal šířené pod označením Turbo Pascal, verze 5.5, 6.0 či 7.0. Jak i v nich správně uplatnit jejich prostředky podporující objektově orientované programování, je hlavním cílem předkládaného textu, protože a přestože se jedná o text na obecnější úrovni.

Myšlenka objektového programování (a přístupu) je důležitá i pro rozvoj nových operačních systému, systémů pro reprezentaci znalostí a také ve spojení s databázemi. Uplatňuje se při vývoji velmi rozsáhlých systémů; výsledný produkt je díky flexibilnosti objektově orientovaných programů snázeji udržovatelný a lépe umožňuje reagovat na změny v zadání.

Objektově orientované programování a jeho technologie nepřišlo na svět proto, aby popřelo staré a osvědčené zkušenosti, ale jen proto, aby je lépe naplnilo. Jak uvádíme ještě několikrát v textu, objektově orientované programování se sebou přináší nové názvosloví. Dnes není ani ve světě názvosloví ustáleno a s novým produktem té které firmy přicházejí i nová jména či nové významy starých jmen. Snažíme se nekopírovat slepě různá názvosloví, snažíme se spíše zavést určitý pořádek do názvosloví a uvádět české překlady pojmů, i když se předem omlouváme za nedostatky, kterých jsme se v této oblasti dopustili. Předem také upozorňujeme, že nechceme řešit dlouze problematiku, zda je správné říkat objektové programování či objektově orientované programování; používáme proto obojího.

PROČ OOP

Počítače se rozvíjejí velmi rychle a mohutně. Za posledních čtyřicet let se jejich parametry výrazně zlepšily a zejména několikanásobně vzrostl jejich výkon. Pokroky pozorujeme hlavně v technickém vybavení počítačů, v tzv. hardware (větší paměť, rychlejší procesor, celkově menší rozměry, atp.). Mnohem menší změny nastaly v uplynulých letech ve vývoji programového vybavení. Je vhodné dodat, že zde se změny měří mnohem a mnohem hůře než technické parametry počítačů. Věnujme se problematice software trochu podrobněji a pokusme se poněkud zpochybnit tezi o dosavadním vývoji software. Proto si nejprve uveďme krutou pravdu o tvorbě programu.

Statistiky z USA ukazují, že v osmdesátých letech pouze zhruba 2% programu byly použity tak, jak byly vytvořeny. Další 2%–3% se mohly používat po mírném přepracování, které nezasahuje víc jak 10–15% programového textu. Velká část, 20%, je přepracovávána zásadním způsobem. Tzn., že 20% dodaných programů programátorskými firmami je vráceno a přepracováváno většinou pomocí nových kontraktů. Další velká část (téměř 50% programů) byla dodána, ale nikdy ji uživatelé nepoužívali a zbytek byl dodán v takovém stavu, že byl nepoužitelný. Tato čísla naznačují, že s programovými produkty to skutečně není tak jednoduché nebo tak krásné, jak se často ukazuje nebo předvádí. Je třeba si uvědomit, že prakticky se používá jen zlomek ze všech vytvořených programů –

a to těch, které se osvědčily. Nejde nám však pouze o toto konstatování, ale i o hledání příčin selhávání programů, které je mnohem častější než selhání hardware.

Při tvorbě software se stále většinou jedná o metodu pokusu a omylu, a výsledný produkt je oproti spolehlivosti a flexibilitě používání hardware neuspokojující. Jak jsme již uvedli, nové technologie stále počítače zmenšují, zrychlují a zvětšují jejich paměť. Proto bývá zvykem hledat příčiny v oblasti tvorby programů. A proto se také hovořívá, i když ne moc nahlas, aby se nepostrašili zákazníci kupující si programy, o nespolehlivosti software a problémech s jeho tvorbou, což někdy vede i k zajímavé argumentaci o ceně tvorby programu.

Nechceme v žádném případě tvrdit, že tvorba programu je bez problémů, ale pokusíme se hledat příčiny i jinde. Nejprve se budeme snažit odhalit podstatu vývojových změn technického a programového vybavení počítačů, tj. podstatu změn hardware a software. A proto se pokusíme podívat na celý výpočetní systém jako na jeden celek (tj. spojení software a hardware), a budeme hovořit o jeho architektuře, což je pojem dostatečně abstraktní, abychom ho mohli použít pro popis takového výpočetního systému.

Když trochu předběhneme výklad, řekneme si, že všechny výše zmíněné problémy s programováním mají základ v původní staré von Neumannovské architektuře počítače, která neodpovídá abstraktním vyjadřovacím schopnostem moderních programovacích jazyků a požadavkům uživatelů na abstraktní komunikaci s počítačem. Věnujme se proto problematice architektury podrobněji.

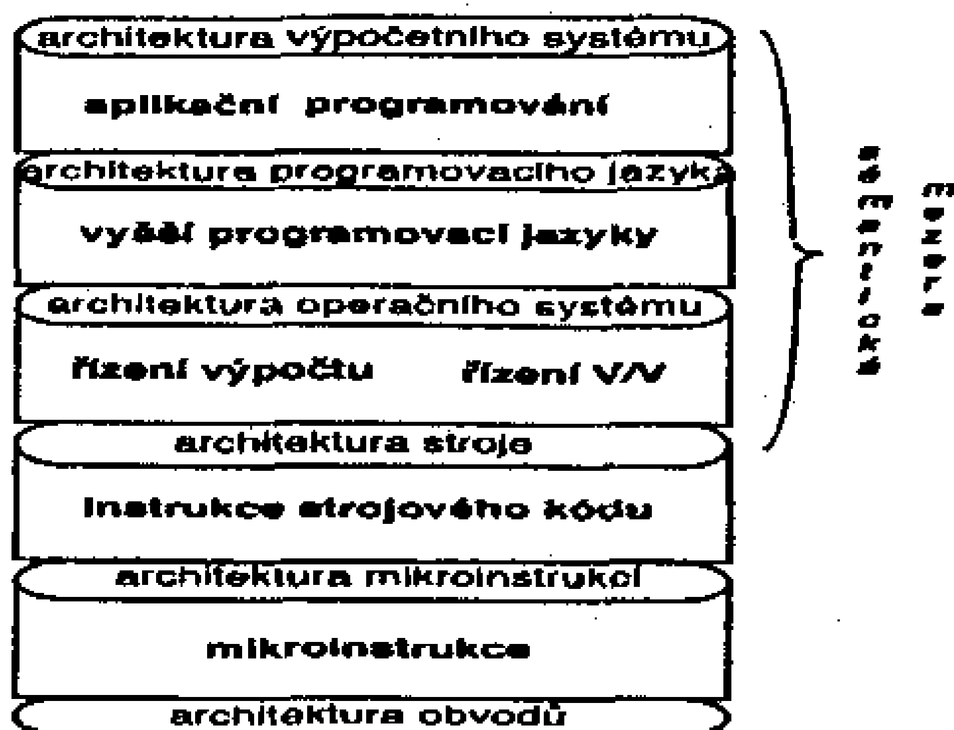
Architektura počítačů

Zde se omezíme jen na nejnutnější poznámky tak, abychom mohli identifikovat problémy jejího používání a posléze specifikovat objektově založené architektury počítačů, jako jedno z možných budoucích řešení.

Architekturou (na dané úrovni) budeme rozumět popis možností (tj. nástrojů) dané úrovně výpočetního systému (obr. 1). Tak např. architektura stroje je určena možnostmi použití počítače na jeho strojové úrovni, tj. jeho strojový jazyk spolu s organizací paměti a adresováním, s postupem provádění instrukcí, atd.; *architektura stroje* tvoří rozhraní mezi hardware a software.

Architektura výpočetního systému jako celku je pak hierarchickým systémem, který je horizontálně členěn podle úrovní abstrakce. Abstraktně vyšší úroveň je pak implementována pomocí prostředků úrovně bezprostředně nižší, např. architektura stroje je implementována pomocí nástrojů architektury mikroinstrukcí. Architektura dané úrovně se však nezajímá o problémy spojené s její implementací nástroji nižší úrovně. Implementace tak vytváří na vyšší úrovni zdání dokonalejšího počítače, který bývá někdy označován jako virtuální počítač. Architektura vyšší úrovně plně odstiňuje detaily architektury úrovně

nižší, např. programově lze na von Neumannově architektuře implementovat třeba i redukční počítač.



Obr. 1 Architektura výpočetního systému

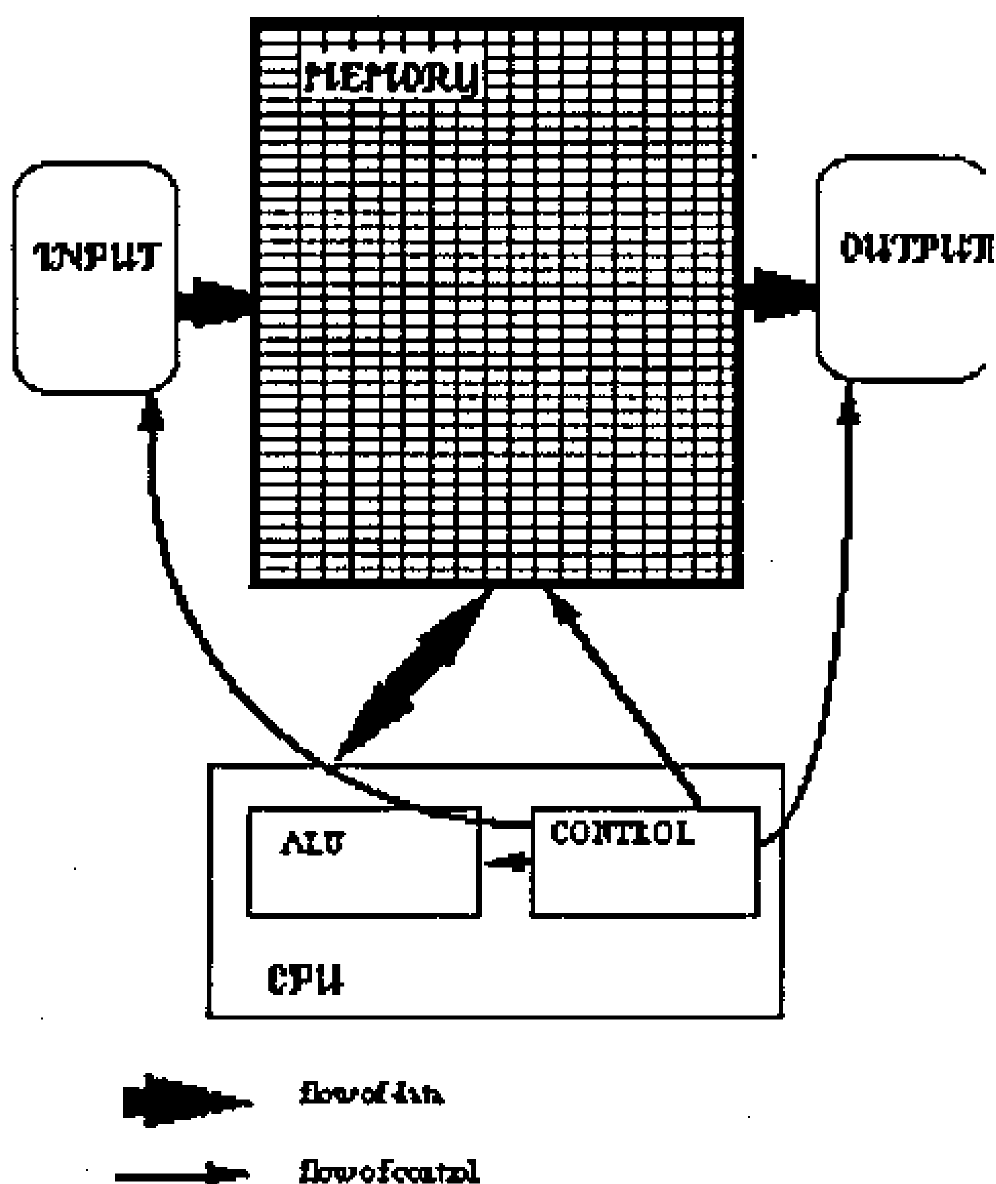
Pro naše úvahy je podstatné, že architektura stroje se dosud principiálně nezměnila. Prošla nicméně dlouhým a bohatým vývojem, který byl odrazem změn v její implementaci. Než si budeme stručně charakterizovat zmíněný vývoj architektury stroje, popišme si základní principy von Neumannova počítače, tj. nejrozšířenější architektury stroje (s tím jsou spojeny i určité implementační detaily architektury stroje, kterým se nemůžeme vyhnout).

Počítač je na úrovni architektury obvodu tvořen aritmetickologickou jednotkou, řadičem, pamětí a vstupními/výstupními jednotkami (obr. 2.).

Činnost počítače je řízena pomocí řadiče programem uloženým v paměti. Řadič provádí jednotlivé instrukce z programu (v jazyce stroje) v předem daném pořadí, které mohou určité instrukce měnit (příkaz skoku). Program sám je tvořen instrukcemi, které se postupně (sekvenčně) provádějí; programátor si při psaní programu musí umět představit dynamickou posloupnost provádění instrukcí programu. Zpracovávaná data jsou uložena v téže paměti jako program. Paměť je lineární a je tvořena buňkami stejné délky, které obsahují sekvence bitů reprezentující čísla, znaky, instrukce a adresy představující operandy instrukcí.

Po léta probíhá zdokonalování právě popsané architektury. Jednalo se o zdokonalování implementace architektury stroje (tj. zdokonalování nižších úrovní architektury jako např.

pásové zpracování instrukcí – angl. pipeling, vyrovnávací paměti, koprocesory, atp.). Nebo se jednalo o zdokonalování instrukčního souboru strojových jazyku, které nejprve vedlo k jeho rozšiřování a později k jeho omezování.



Obr. 2 von Neumannův model počítače

Jestliže nelze pozorovat zásadní změny architektury stroje (technického vybavení, hardware), pak vývoj programovacích prostředků (architektury programovacích jazyků, software), ač nedosahující zářných výsledků (jak jsme již dříve ukázali), obsahuje mnoho principiálních změn směrem k vyšší abstrakci nabízených prostředků. Máme zde na mysli např. rozdíl mezi jazykem Fortran z roku 1950 a jazyky jako jsou Prolog, Occam, Miranda, Eiffel, Lucid či ML; podstatný je rozdíl jejich vyjadřovacích schopností. Nebo si jen povšimněme, jak vzdálené strojovému jazyku je ovládání počítače při návrhu účesu či návrhu karosérií aut. Implementovat takové velmi abstraktní systémy vyžaduje programově

překrýt uvedený rozdíl ve vyjadřovacích schopnostech mezi architekturou stroje a architekturou výpočetního systému pojmenovávaný sémantická mezera (angl. semantic gap).

Snaha o architekturu vyšší abstraktní úrovně není samozřejmě absolutní, příkladem velmi úspěšné architektury programovacího jazyka je C. Jeho obliba je naopak dána tím, že je velmi blízký architektuře stroje, a proto ji dovoluje v programu plně využívat pro výrazně efektivnější kód.

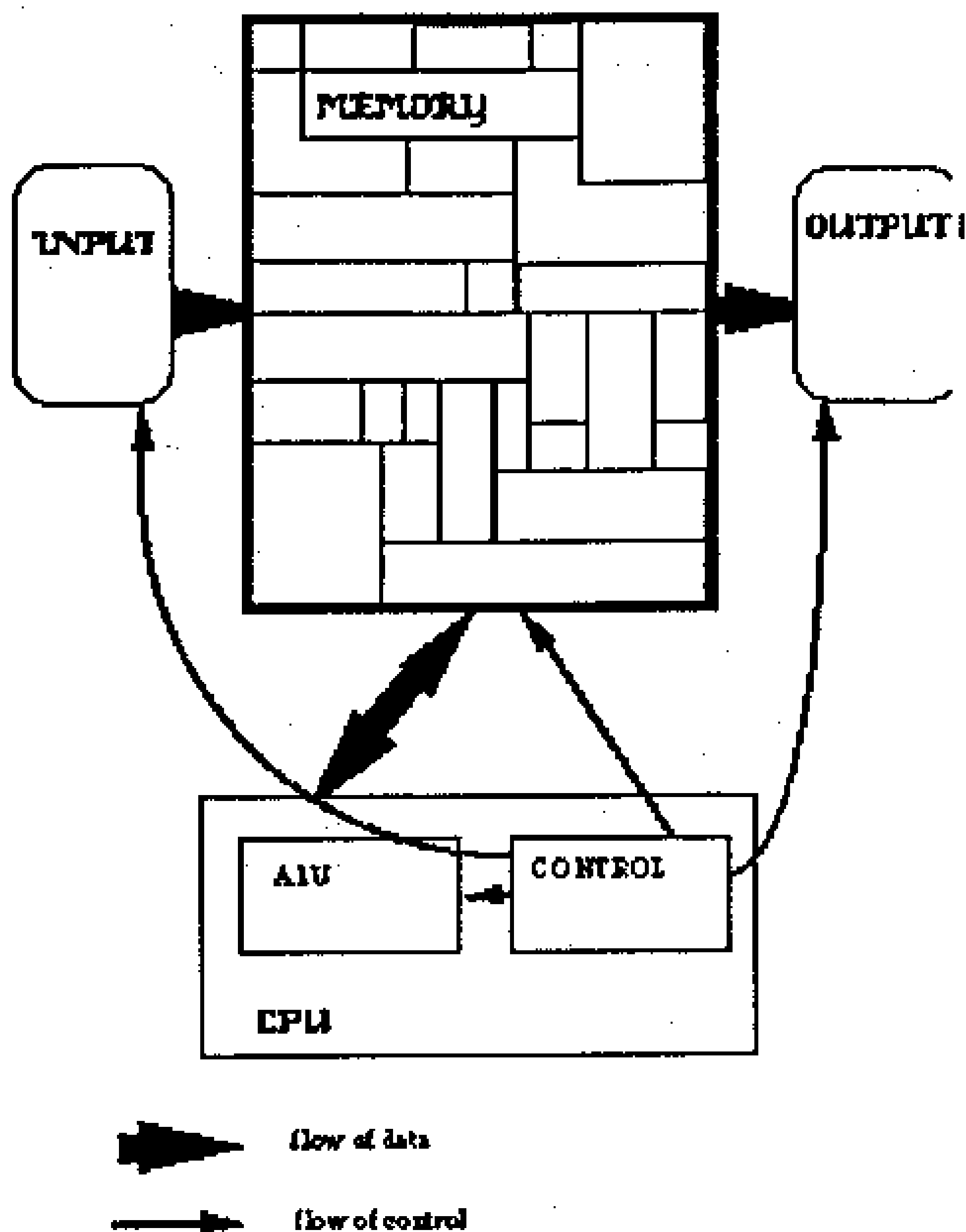
PROČ OOP – ZNOVU

Bezchybná implementace velmi abstraktních výpočetních systémů na klasické von Neumannově architektuře je vzhledem ke své složitosti téměř neřešitelný problém (pokud jsou nějaké pokusy v tomto směru použitelné, pak jsou řádově pomalejší, jsou méně efektivní). Proto se možné *řešení* nehledá ve zlepšování von Neumannovy architektury, ale v *zásadní změně architektury stroje, v jejím přiblížení abstraktní úrovni vyšších programovacích jazyků*. Právě uvedené zjištění není ničím novým, objevným. Připomeňme si jen, že podobná úvaha zřejmě vedla k založení japonského projektu počítačů páté generace a všech jeho následovatelů. V Japonsku se vrhli na vývoj počítačů nebývalých schopností, což je zajímavé pro laickou veřejnost, my si spíše povšimněme, že za strojový jazyk zvolili jazyk velmi blízký jazyku Prolog. Jiným z nabízených řešení jsou právě objektově založené architektury.

V literatuře jsou také popsány další možné *neimperativní* (non von Neumannovské) *architektury stroje*: data–flow, redukční, deduktivní. Kromě toho dnes známe neuronové počítače a objevují se i další nová paradigmata výpočtu (tak se dnes obvykle označuje architektura vyššího programovacího jazyka; v našem příspěvku jsou pojmy architektura a paradigma často zaměnitelné, což někdy zdůrazňujeme jejich společným uvedením). Všem jim je společná zásadní změna architektury stroje. Uvedené neimperativní architektury bývají také označovány jako neimperativní *modely výpočtu*.

JAK OOP

Objektové architektury počítačů nemění architekturu stroje tak 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 (nebo paradigma, nebo přístup, nebo styl) klást na stejnou úroveň jako dříve zmíněné ne von Neumannovské architektury stroje; nejedná se totiž o tak zásadní změnu architektury. Objektově orientovaný přístup znamená *změnu práce s pamětí* a ne o změnu modelu výpočtu.



Obr. 3 Objektový von Neumannův model počítače

Např. uplatnění objektově orientovaného přístupu v rámci von Neumannovy architektury stroje se projeví změnou pouze části organizace architektury stroje, tj. paměti. Na obr. 3. vidíme model objektově orientovaného von Neumannovského počítače, kdy v paměti nejsou uloženy jednotlivé bity a bajty, ale paměť obsahuje přímo objekty, to jest entity, které obsahují sebeidentifikaci (tj. můžeme se jich zeptat, co jsou zač, jako se můžeme ptát i entit Homo Sapiens). Objekty, jak vidíme, jsou různě veliké a druhou jejich základní vlastností je ochrana přístupu k jejich vnitřku, tj. lokálním údajům, které obsahují.

Objektově orientovaný přístup se tak snaží řešit snad hlavní problém von Neumannovy architektury, to je *problém primitivnosti paměti*, v níž nerozeznáme, co paměťová buňka

obsahuje (bez znalosti kontextu a bez systémové pomoci) a můžeme její obsah libovolným způsobem interpretovat. Objektově orientovaný přístup řeší také problém ***globálnosti paměti***, tedy toho, že i ta nejmenší změna v paměti mění také celkový stav paměti. Taková změna je pak těžko lokalizovatelná. Obojí se komplikuje ještě uložením dat a programu do stejné paměti (harwardská architektura proto paměť rozděluje na paměť programu a paměť dat). Řešením jsou objektově orientované modely paměti vyznačující se 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 (z globálního pohledu zůstává paměť nezměněná – obsahuje tytéž entity).

Z předchozího výkladu vyplývá nutnost řešit problémy dnešních počítačů odstraňováním jejich základní příčiny: sémantické mezery. Nestačí přerozdělovat rozložení funkcí architektury do různých úrovní při zachování primitivnosti von Neumannovy architektury stroje, vynucené primitivní organizací paměti, neexistencí informací o významu (sémantice) toho, co je v paměti uloženo. Při uložení čehokoliv do von Neumannovy paměti jsou odtrženy abstraktní informace od svého významu. Do paměti se uloží pouze bitově kódovaná hodnota, kterou je možné později interpretovat libovolným způsobem, nezávisle na původním významu, čehož lze koneckonců používat k různým programátorským trikům. Někteří (opravdoví) programátoři jsou Mistry takových triků.

Abstraktní sémantika obsahu paměti je součástí programu ve vyšších programovacích jazycích a při transformacích na programy s nižší úrovní abstrakce se sémantika zpracovává (ztrácí se pro nižší úroveň). Její poslední zbytky (na úrovni velmi jednoduchých datových typů a jejich operací) jsou obsaženy v instrukcích von Neumannovy architektury stroje. Překladač vyššího programovacího jazyka musí zajistit správnost (když zanedbáme efektivnost) *této transformace, tj.* aby výsledek sémanticky slepého výpočtu na úrovni architektury stroje byl při abstraktní interpretaci správný. Není divu, že (izolovaná) vysoká úroveň abstrakce programovacích prostředků vede za těchto podmínek do slepé uličky. Vede totiž ke zvětšování sémantické mezery (některé vysoce abstraktní vyšší programovací jazyky mají velmi jednoduchou sémantiku – jejich implementace může být tudíž relativně jednoduchá, ale v tomto případě je velmi výrazně neefektivní – snaha o efektivnější implementaci vede opět k problematické rozsáhlosti překladače mající vliv na jeho korektní chování).

Jediné možné řešení problému sémantické mezery spočívá v přenesení větší zodpovědnosti nebo její části na funkční a paměťové obvody, např. v organizaci paměti na vyšší úrovni abstrakce. Za nejperspektivnější modely architektury počítačů s abstraktně (objektově) definovanou pamětí se považují objektové modely.

Základem objektových systému je tedy OBJEKT. Objektová paměť tedy obsahuje objekty, objektově orientované programování pracuje s objekty, atp. K podrobnějšímu popisu objektu se dostaneme později, zde si jen uveďme, že objekt je nedělitelná entita obsahující data, jejich identifikaci a možnosti svého použití (přípustných operací).

Takto je objekt chápán na abstraktní úrovni architektur; připomeňme, že i celá řada praktických implementací ho chápe obdobně. Nesmí nás ale překvapit, když v určité situaci bude představa objektu modifikována.

JAK SE DÍVÁME NA OBJEKT

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 např. může být ve Fortranu celé číslo interpretováno jako hodnota jiného datového typu. Takovým změnám nebo záměnám se předchází zaváděním typů a následnou typovou kontrolu. V těchto jazycích (např. Pascal) je třeba při překonávání kontrol typu 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.

Na obr. 4. vidíme objekt jako spojení dat a zdi (wall), která tato data chrání.

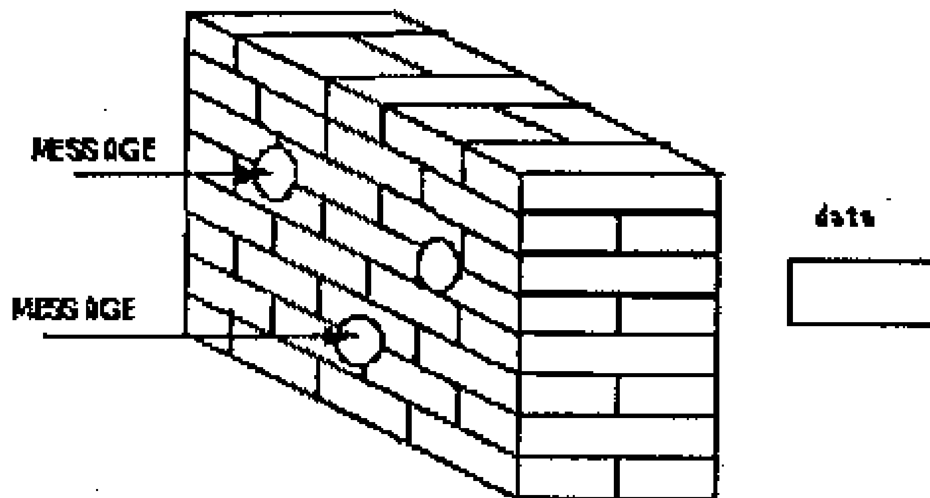
Ochranná zeď nejenže chrání data, ale také je identifikuje. To znamená, že nám říká, o jaký údaj se jedná.

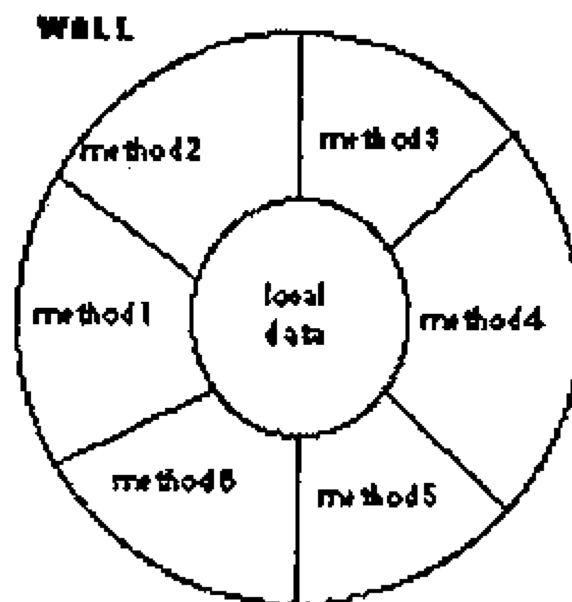
Na obr. 4 je kromě zdi z cihel uvedeno grafické vyjádření objektu ve tvaru kruhu, kde jsou data uvedena ve vnitřním kruhu; mezikruží pak obsahuje tzv. metody, které popisují jedině možné operace s objekty. Ve zdi z cihel jsou tyto znázorněny dírami, kterými je možné získat přístup k lokálním údajům objektu. Na objekty se obracíme zprávami (angl messages). K pojmům metoda, zpráva a posílání zpráv se ještě vrátíme.

Konkrétní objekt vidíme na obr. 5; jedná se o objekt množina s metodami *union* a *isElement*. Upozorněme, že dále budeme ke znázornění objektu používat již jen kruhový tvar, kde je zeď ochraňující data znázorněna mezikružím.

Zapouzdření dat tedy slouží i k jejich identifikaci; objekt ví co je a tudíž i jaká data obsahuje – zda to je např. číslo, pravdivostní hodnota, či množina hodnot. Identifikaci objektu je možné v programu používat a při běhu programu ji kdykoliv zjišťovat.

WALL OF CODE AROUND EACH PIECE OF DATA





Obr. 4 Objekt

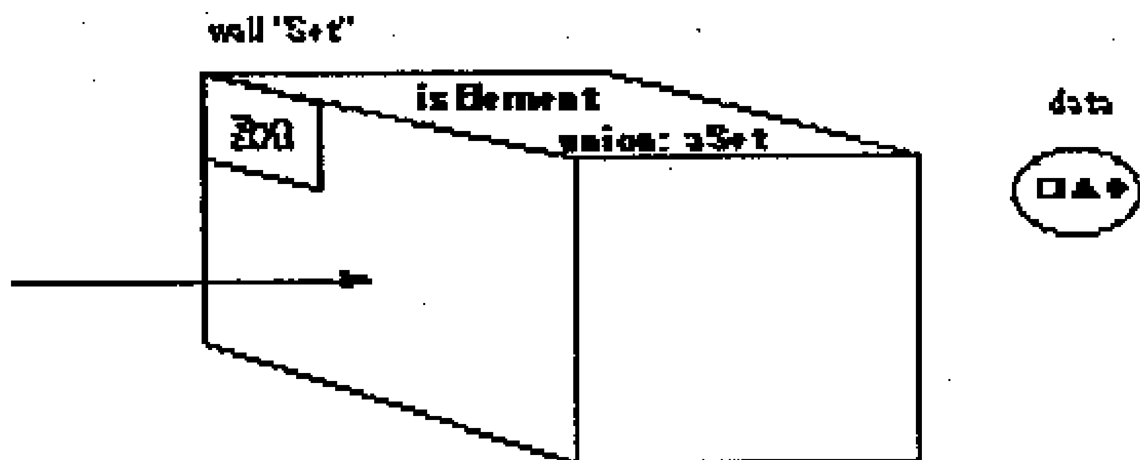
Výhody zapouzdření jsou v tom, že:

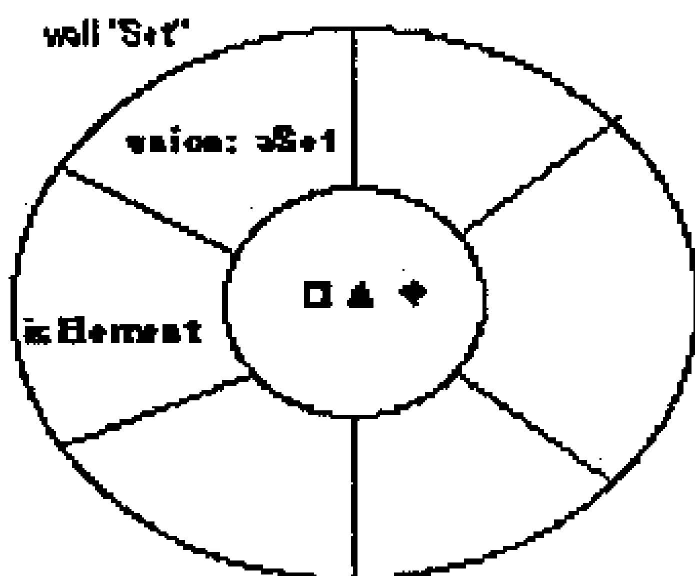
– programy mohou být testovány po menších částech; častá chyba při procedurálním programování je skryta v přístupu ke sdíleným údajům, čemuž zapouzdření metodicky předchází;

– interní struktura dat může být měněna bez nutnosti změn okolí objektu;

– zapouzdření vlastně ukrývá lokální data před okolím objektu – hovoříme o tzv. externím skrývání;

– mohou být vytvářeny katalogy či knihovny objektů, vlastně tedy abstraktních datových typů, které je možné v jiných aplikacích volně použít;





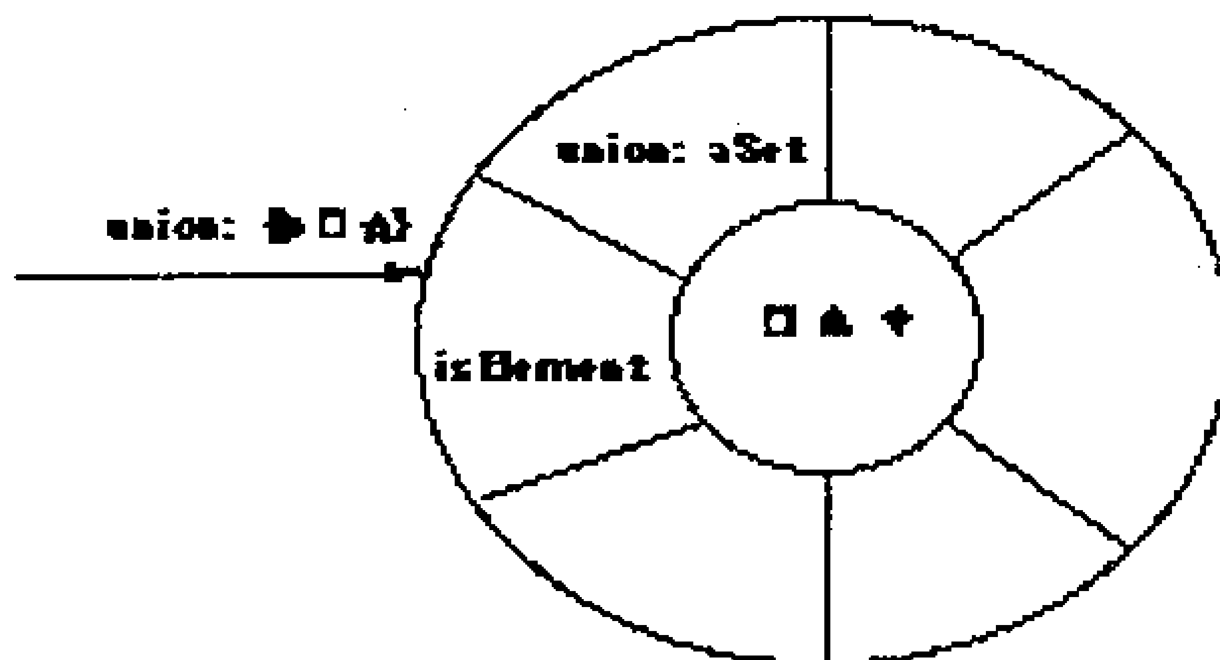
Obr. 5 Objekt množina

– 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 objektu.** 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).

O ZPRÁVÁCH

Objekty komunikují s jinými objekty pomocí **posílání zpráv** (angl. sending messages); již na obr. 4 vidíme poslání zprávy. Přípustnými zprávami pro objekt je definováno jeho



Obr. 6 Poslání zprávy objektu množina

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.

Objektům 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 (nebo funkce) v klasickém programování. Metoda (angl. method) je popis jak provést jednu z operací objektu. Metoda je tedy vlastně část programu, tj. podprogram. Provedení operace vyvoláme posláním zprávy s odpovídajícím selektorem, který identifikuje metodu. Hlavička metody se nazývá **vzor metody** (angl. method pattern) a odpovídá hlavičce procedury (nebo funkce) obsahující jméno a formální parametry.

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

Na obr. 6 vidíme poslání zprávy **union:** {✱ ◻ ☆} objektu množina. V zápisu si povšimněme, že zpráva požadující sjednocení množin má zapsaný pouze jeden argument – tím druhým („chybějícím") je totiž příjemce zprávy.

Data, která jsou v objektu zapouzdřena, je možné změnit pouze tak, že objektu pošleme zprávu, které rozumí. Nikdy není možné si na vnitřní data objektu přímo „sáhnout". Ve shodě s obr. 5 a 6 si představme, že kódy metod jsou obsaženy ve zdi, která lokální data objektu chrání. Takto jsou uloženy přímo u objektu zprávy, kterým on rozumí.

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 např. v jazyce FORTRAN či Pascal), 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 při běhu programu, což může být díky dědění (viz dále) relativně složitý proces.

Obecně se model posílání zpráv popisuje pomocí okamžiku určení kódu, kterým se provede určitá operace. Rozlišujeme tedy tzv. **pozdní a brzkou vazbou kódu** (angl. late resp. early binding). Chápe se tím doba, kdy je znám kód metody, kterou se provede činnost způsobená správným posláním zprávy (tj. obecně činnost způsobená voláním podprogramu). V 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 volání podprogramu je naopak kód operace znám již v době překladu programu – jedná se o brzkou vazbu.

Z toho ovšem vyplývá, že v objektových systémech může mít tatáž zpráva zaslaná různým objektům různý efekt, což lépe popisuje vlastnost nazývaná 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 plánku místnosti či uvedení jména (nebo jmen) aktuálních uživatelů postele.

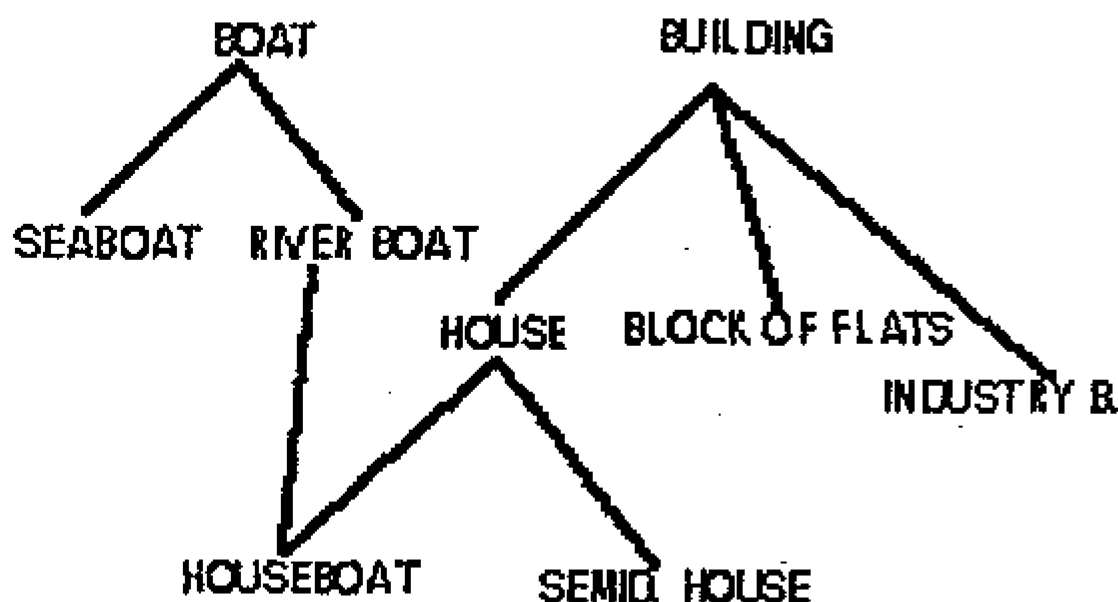
Není tedy třeba dbát na různost selektorů metod pro významově podobné akce jen proto, že jsou prováděny s různými daty, jak je tomu se jmény procedur (nebo) funkcí v klasických procedurálních jazycích – pro objekty naopak vidíme maximální snahu používat minimum různých selektorů metod.

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 relativně málo společných vlastností – a ty, které jsou v hierarchii níže, mohou být velmi jemně a postupně odlišovány od obecných; týká se to zejména přípustných operací s daty. K popisu vztahů v hierachii se také používá pojmenování předchůdce – následník, nebo rodič – dítě.

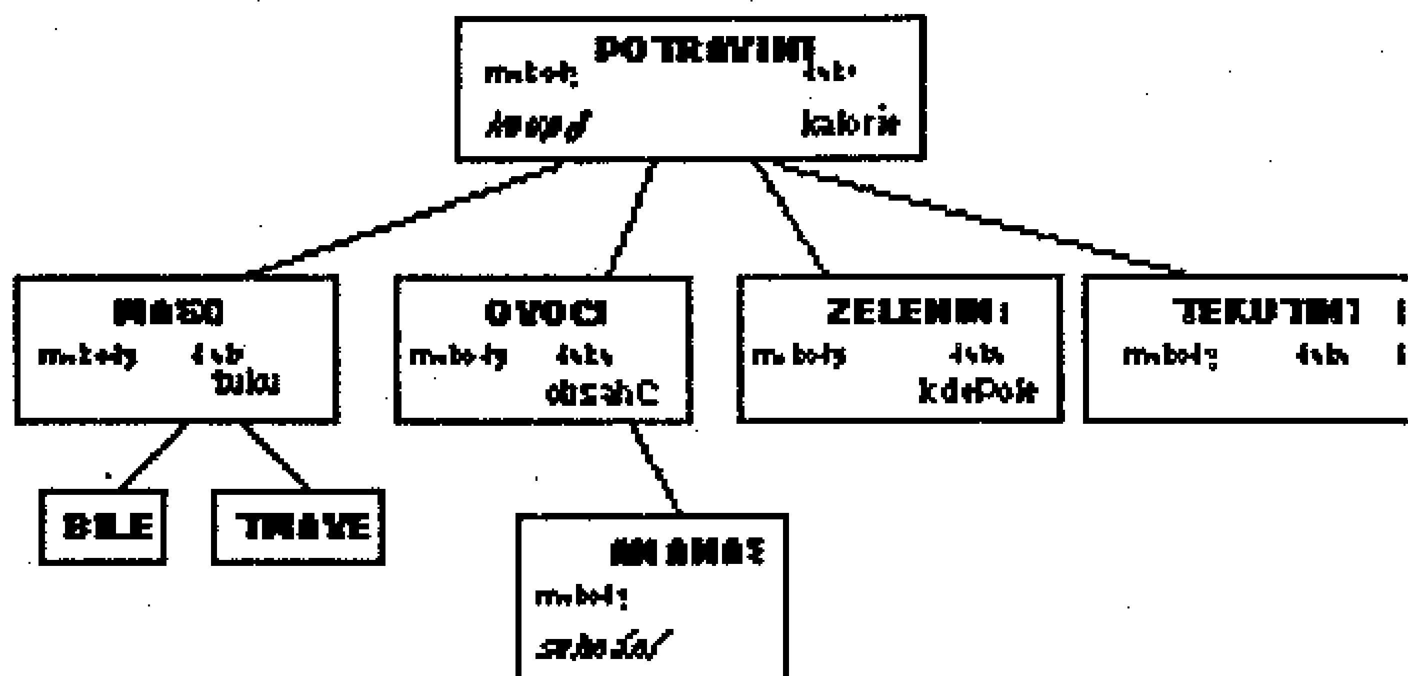
Na obr. 7 vidíme příklad **hierarchie dědění** mezi objekty. *Člun* (angl. boat) je předchůdcem *člunu na moře* a *člunu na řeku*; ty od něj dědí společné vlastnosti člunu a navíc si každý přidává něco svého, něco specifického. Podobně je zde *budova* (angl. building) předchůdce *vilky*, *paneláku* a *průmyslové budovy*. V uvedených případech se jednalo o jednoduchou (neboli prostou) dědičnost, kdy pro každý objekt v hierarchii platilo, že má nejvýše jednoho předchůdce. *Hausboat* je příkladem objektu, který dědí od dvou rodičů; v tomto případě hovoříme o **násobné dědičnosti** (angl. multiple inheritance).



Obr. 7 Hierarchie dědění mezi objekty

Podobné hierarchie známe i z některých přírodních věd, kde se vytvářejí taxonomie rostlin nebo živočichů.

Jiným příkladem jsou potraviny a jejich hierarchie; 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 (následníci): *maso, ovoce, zelenina, tekutiny*; *maso* se dělí na *bílé* a *tmavé*. Pro masa má význam udávání množství tuku, pro ovoce obsah vitamínu C a pro zeleninu místo vypěstění. Všechny



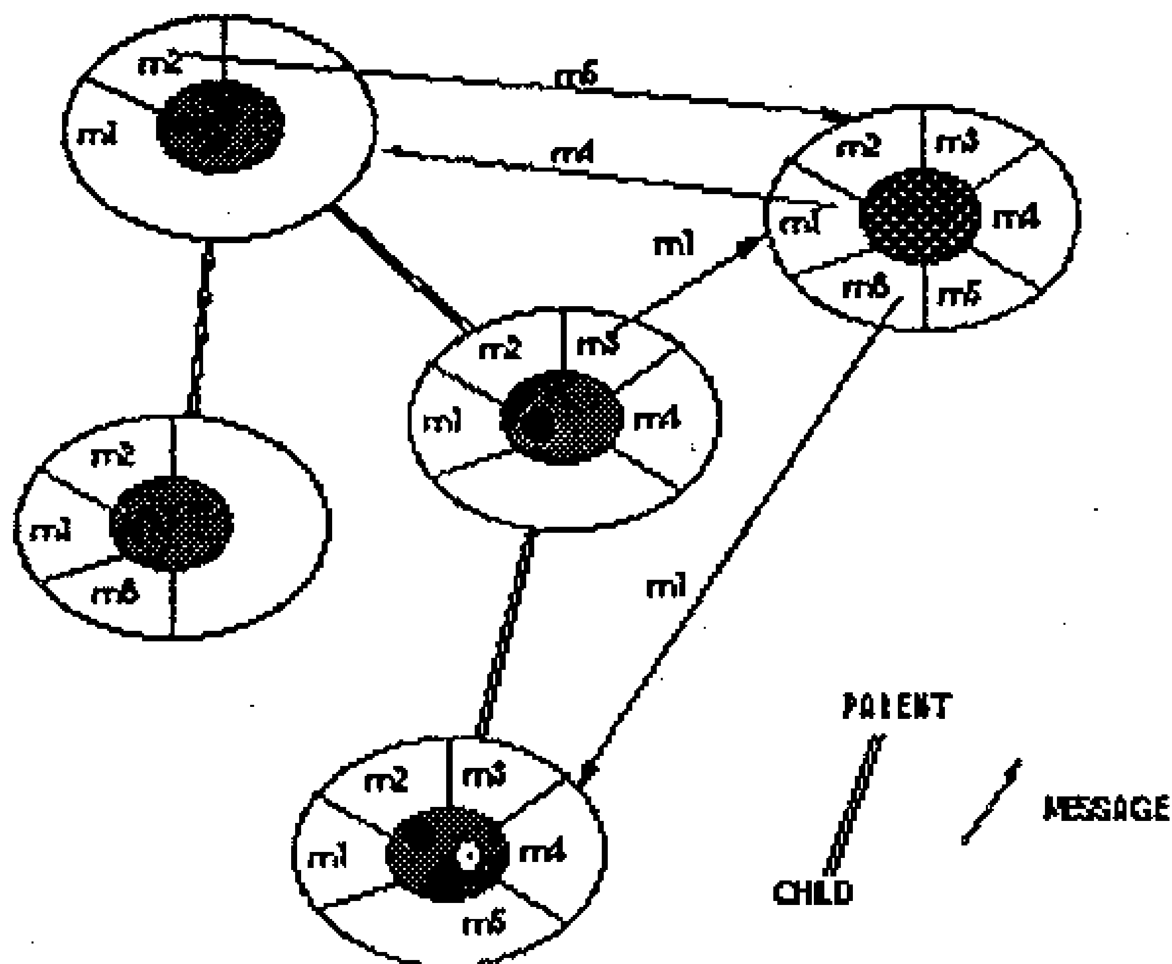
Obr. 8 Hierarchie potravin

potraviny v této hierachii dědí operaci *koupě* a udávání kalorické hodnoty (lépe energie v kJ) – od předchůdce.

Složitější a abstraktnější příklad hierarchie dědění vidíme na obr. 9. Zde je vlevo nahoře objekt, který má definovány metody *m1* a *m2*, jejich kód je uveden v rozhraní tohoto objektu. Tyto metody od něj dědí všechny jeho děti (což je zdůrazněno uvedením jejich jmen v rozhraní všech dětí – angl. child) a dále i jejich následníci (praděti a prapraděti, atd.). Metody v následnících levého horního objektu jsou na obr. 9 obsaženy v jejich zdech, které chrání lokální data těchto následníků. Pokud následník metodu dědí (a nemění ji), pak je uvedení jejího jména ve zdi kolem lokálních dat vlastně nadbytečné, protože zde kód metody není. V následnících je kód stejnojmenných metod uveden jen tehdy, pokud si následník definuje metodu nově. Metody *m1* a *m2* pravého horního objektu, který v našem příkladu nemá ani předchůdce ani následníky, nemají – kromě jména – nic společného s metodami *m1* a *m2* z levé hierarchie objektů.

Na obr. 9 také vidíme několik poslání zpráv. Uvědomme si, že v systémech, kde se vyskytují jen objekty, může být poslání zprávy pouze součástí kódu některé jiné metody. Ne všechna poslání zpráv jsou vždy správná – příkladem chybného poslání zprávy je zpráva *m4*.

Vraťme se ještě k dědění na obr. 9. Objekt ve středu obrázku má v lokálních datech, které zdědil od svého rodiče (angl. parent) navíc svůj specifický lokální údaj – černou tečku. Ta je zděděna jeho následníkem, který si ještě do lokálních dat přidal svoji bílou



Obr. 9 Hierarchie objektů s posíláním zpráv

tečku. POZOR! Následníci (tj. děti) nedědí lokální data včetně jejich hodnot, zděděna je pouze struktura lokálních dat. Tedy: Dva objekty na obr. 9 mají ve svých lokálních datech údaj popsaný černou tečkou, hodnota černé tečky je však různá v obou objektech. Podobně můžeme v hierarchii dědění vysledovat i dědění a přidávání metod. Uvedený příklad nemůže postihnout všechny možnosti vztahu předchůdce – následník, jak se s nimi setkáváme v nejrůznějších programovacích systémech.

Objektové systémy obvykle také 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 – viz obr. 8.

VZTAH PŘEDCHŮDCE–NÁSLEDNÍK, angl. „IS A“

Uveďme si nyní přehled možností ve vztahu mezi následníkem a předchůdcem.

Následník může:

– přidat metodu či metody;

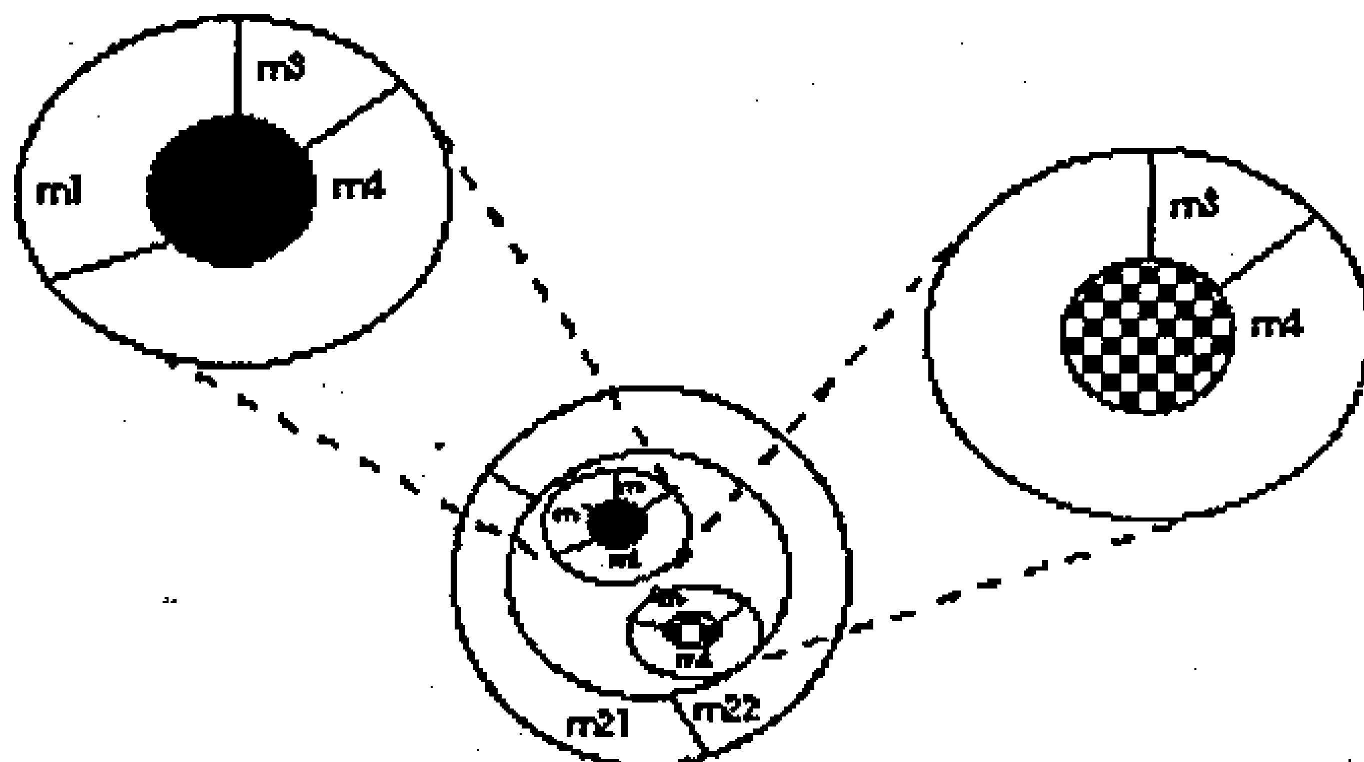
- přidat lokální údaj či data;
- předefinovat metodu či metody – chování následníka je pak odlišné od chování předchůdce i pro tutéž zprávu;
- mít skryta určitá data svého předchůdce, což se předepisuje v definici předchůdce – viz jazyk C++; jedná se o interní skrývání dat v hierachii dědění
- mít skryty určité metody svého předchůdce, což se předepisuje v definici předchůdce – viz jazyk C++; jedná se o interní skrývání metod v hierachii dědění

Výhody dědičnosti jsou:

- struktura dědění a hierarchie objektu 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.

VZTAH SKLÁDÁNÍ OBJEKTŮ, PÁN – SLUHA, angl. „HAS A"

Kromě vztahu předchůdce–následník existuje mezi objekty druhý základní vztah, totiž **skládání objektů**, které si ukážeme na příkladu skládání dvou objektů v jeden.



Obr. 10 Skládání objektů

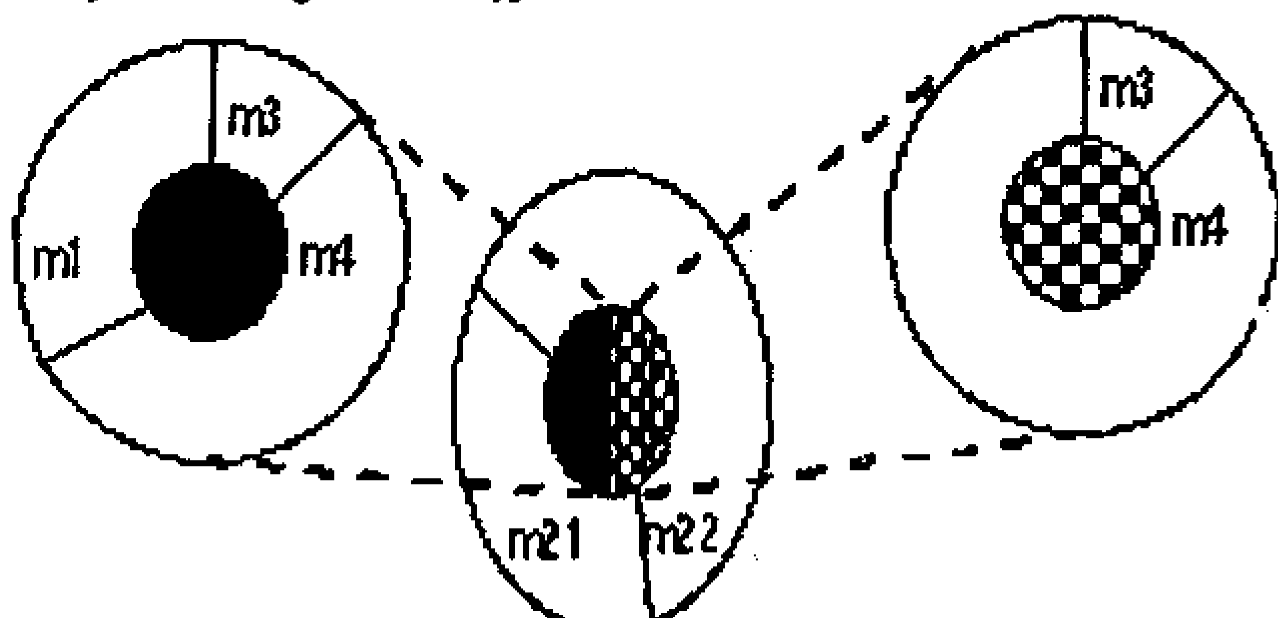
Na obr. 10 vidíme jak se dva objekty složily v jeden. Lokální data prostředního objektu jsou tvořena oběma objekty, jak dobře vidíme. Má to ale zajímavý důsledek, který nemusí být na první pohled zřejmý – díky zapouzdření není možné prostřednímu – složenému – objektu poslat zprávy *m1, m3, m4*. Ty nejsou obsaženy v rozhraní prostředního objektu!

Složený objekt tedy staví novou zeď, která nově určuje, které z metod vnitřních objektů (a jak) budou použitelné. Zde je ukryto zdůvodnění, proč se tomuto vztahu objektů také někdy říká vztah PÁN SLUHA; PÁN si vybírá od svých objektů (nebo jen jednoho vnitřního objektu), co z jeho vlastností použije. Objekt, který tvoří jeho lokální data, je v roli SLUHY, který poskytuje to, co PÁN vyžaduje.

Často lze slyšet otázku, který z uvedených základních vztahů mezi objekty je více „objektově orientovaný". Simplifikující přístupy upřednostňují dědění (protože je to něco typického pro objektový přístup), ale my vidíme, že skládání objektů je neméně důležité, neboť umožňuje to, co je pro objekt snad ještě důležitější než dědění, umožňuje ochranu lokálních dat – zapouzdření.

Naše odpověď na výše uvedenou otázku tedy zní: ani jeden přístup nelze upřednostňovat mechanicky, pro danou úlohu a její řešení je vždy důležité dobře zvážit, který z uvedených vztahů je ten vhodnější. Ve schopnosti správného vyřešení tohoto problému leží velká část umění objektově orientovaného řešení úloh. Ukazováním správných a chybných řešení se budeme později věnovat, až budeme řešit konkrétní příklady.

Na obr. 11 vidíme jak budeme dále znázorňovat skládání objektů. Oproti obr. 10 se jedná o zjednodušení grafického vyjádření.



Obr. 11 Skládání objektů – zjednodušené znázornění

JAK JSME SE DÍVALI NA OBJEKTOVÉ SYSTÉMY

Objektové systémy, jak jsme o nich dosud hovořili, byly založeny na objektech, posílání zpráv, dědičnosti mezi objekty a na skládání objektů. Existují i takové programovací systémy, které s touto představou objektového přístupu vystačí; jedním z nich je systém, který bývá označován jako actorový model. Nyní se ale soustředíme na pojem, který je často považován za nejdůležitější z celého objektově–orientovaného přístupu, totiž na pojem ***třída***. Nejprve si uveďme některé motivace, které nás vedou k zavedení a používání tříd.

MOTIVACE

Zdroje, z nichž objektově orientovaný přístup vyrůstá, lze hledat v mnoha oblastech počítačového výzkumu. Zde se zmíníme pouze o dvou takových liniích. První z nich jsou ***abstraktní datové typy***, které mají mnohé vlastnosti tříd, jak dále uvidíme. Totéž platí i o ***teorii rámců*** a umělé inteligence a tedy o reprezentaci znalostí.

Připomeňme si jen zcela odlišnou motivaci obou zmíněných přístupů. Abstraktní datové typy patří do teorie programování a vedou ke zlepšování kvality tvorby programů; reprezentace znalostí má za cíl co nejpřímočařejším způsobem převést svět kolem nás do světa počítače a mít schopnost s ním manipulovat.

U tříd se v tomto pohledu uplatňuje základní princip používání vědeckých metod, totiž ***abstrakce***. Kdy pro určité účely (např. programu pro navrhování účesů) abstrahujeme od některých skutečností pro uvedený záměr nepodstatných (např. vyživování pokožky, hormonální vlivy).

Podobným způsobem vlastně vznikají i pojmy v našem jazyce, kdy od představy všech konkrétních stromů abstrahujeme k tomu, abychom si vytvořili jeden pojem stromu. K podobným účelům nám bude sloužit i třída, kdy bude představovat třídu všech stromů a její jednotlivé instance budou stromy. Hierarchie tříd budou postupně vést k taxonomii. Taxonomii je možné vytvářet i opačným způsobem, tj. dedukcí, kdy od obecných pojmů v hierarchii přecházíme k více specializovaným a jejich vlastnosti odvozujeme od těch více abstraktních.

Hierarchie tříd (vlastně pojmů) tak vznikají několikerým způsobem, první z nich je živelný vývoj v přírodě (náhodné mutace), nebo cíleně jako např. klasifikace zvířat, což je jen popis existujících hierarchií v přírodě, nebo zcela umělé jako hierarchie datových typů v programovacích jazycích.

S datovými typy v programovacích jazycích také souvisí abstraktní datové typy. Ty vyjadřují obecný koncept datových typů popisovaných nejčastěji formálním algebraickým aparátem. Abstraktní datové typy se vyznačují možnou parametrizací,

obohacováním a skrývání implementace. Jednoznačně definují rozhraní – použitelné operace.

Motivací pro zavedení abstraktních datových typů je také ukrývání informací (a vedou tak i k modulárnímu programování s moduly s jasným rozhraním). Abstraktní datové typy popisují celou množinu možných instancí. Instancemi abstraktních datových typů jsou dnes nejčastěji statické proměnné v různých programovacích jazycích, které abstraktních datových typů používají.

Snahou hierarchií je zavést určitou klasifikaci (to již od Platóna), která je duální k taxonomii. Při taxonomii popisujeme shora dolů rozdíly (u Aristotela to byly dichotomie), při abstrakci naopak zdola nahoru popisujeme a dáváme dohromady společné vlastnosti.

O motivacích zavedení hierarchií a tudíž často i tříd jsme již hovořili dostatečně, ale ještě musíme přidat jeden důležitý důvod, proč jsou třídy do programovacích jazyků přidávány, kterým je implementace takových jazyků a zejména její efektivita.

Když jsme hovořili o objektech a jejich hierarchiích, tak jsme zdůrazňovali, že každý objekt si de facto s sebou nese mnoho informací ve svém rozhraní, přičemž hierarchií objektů přeci jen dovedou množství informace v každém objektu díky dědění snížit. Zásadnějším řešením je zavedení tříd, které v sobě přináší i nemalou úsporu paměti, které se jinak nedostává.

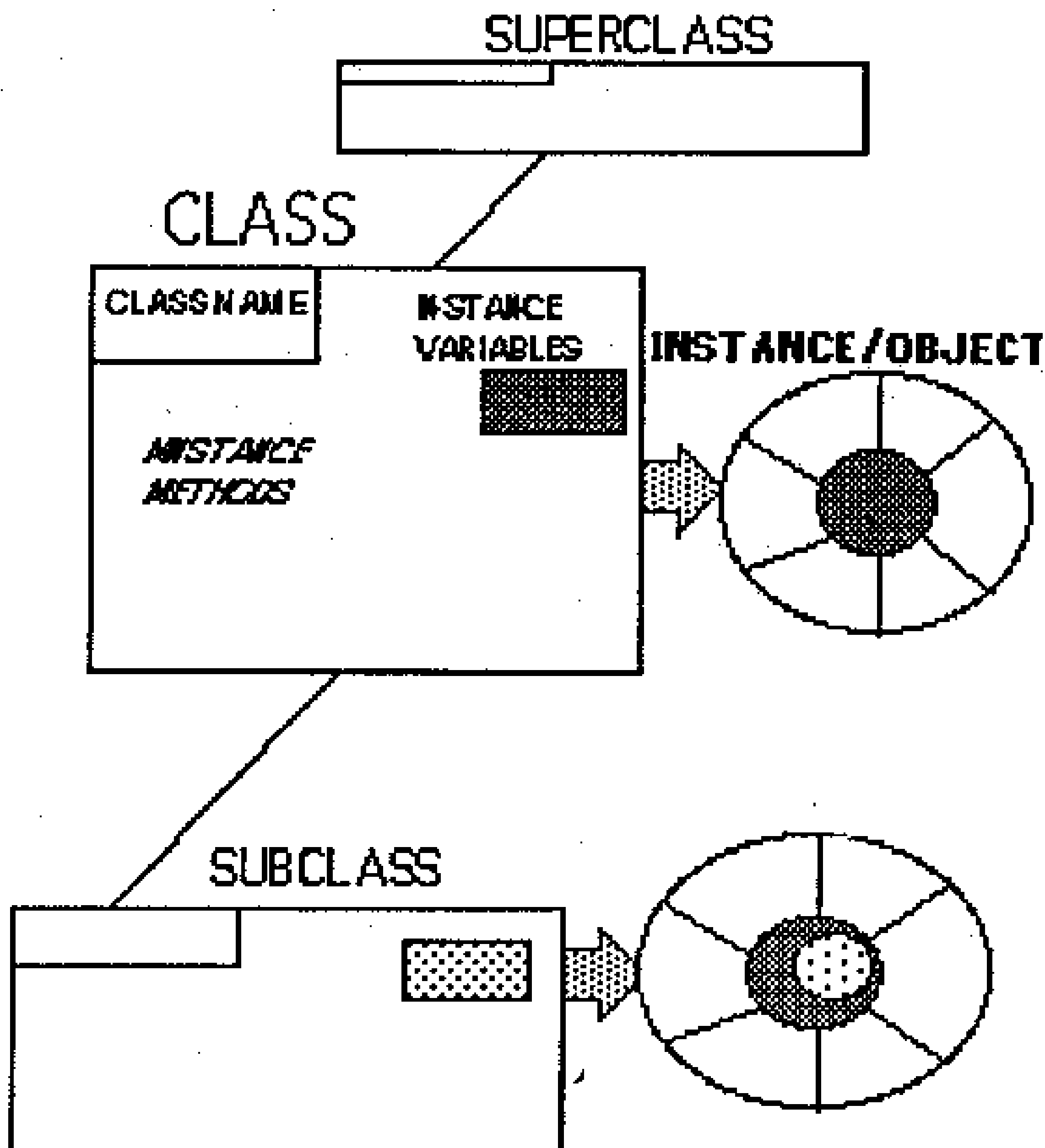
TŘÍDA

Jak jsme již naznačili je **třída** místem, kde jsou uvedeny *vlastnosti instancí* (tj. objektů). Tedy popisují se zde *lokální proměnné instancí* a *metody*, kterým instance rozumí (angl. postupně: instance variables, instance methods).

Třída je také do určité míry zodpovědná za tvorbu svých instancí, říká se, že je *továrnou na objekty*.

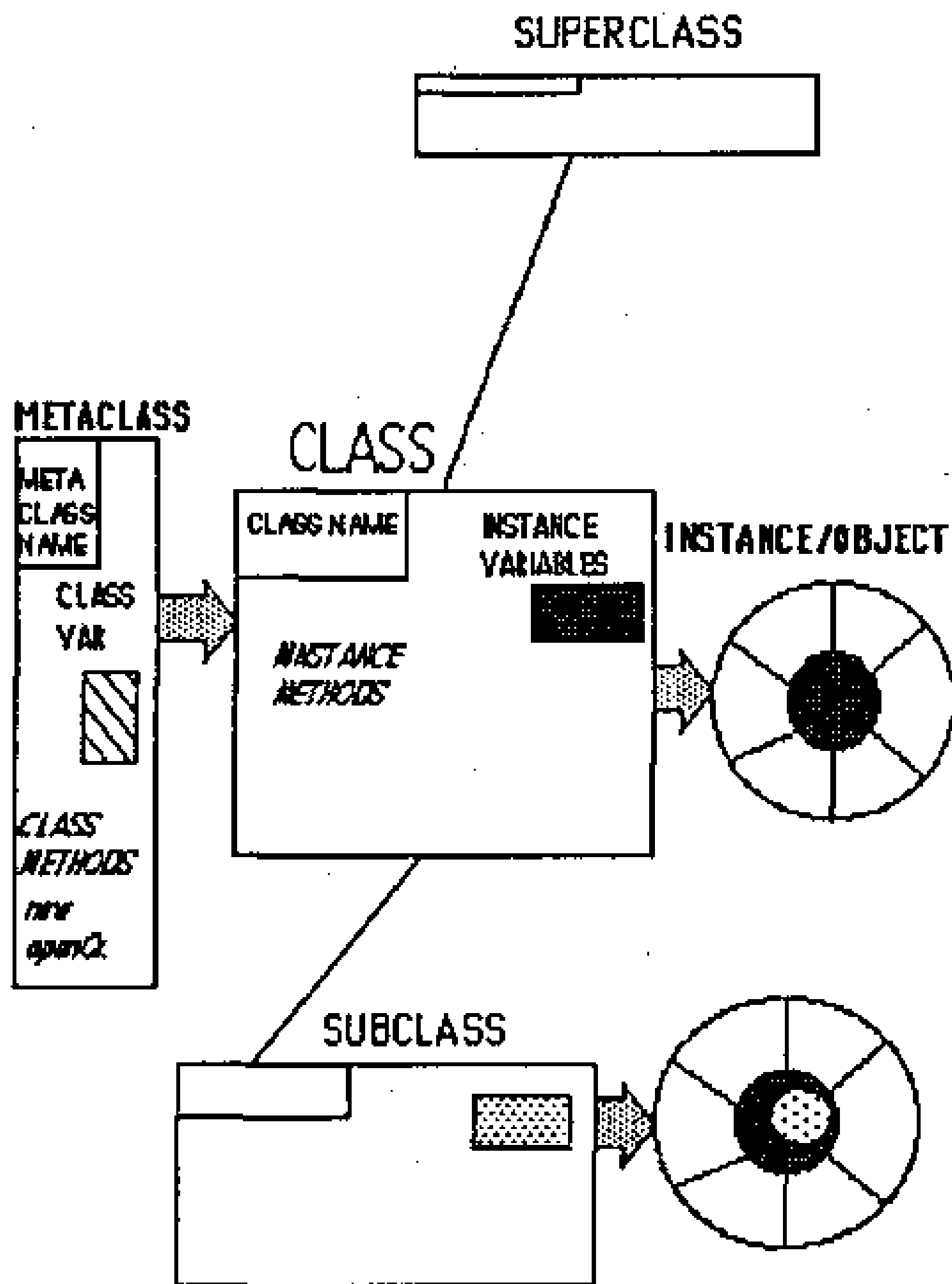
Máme-li tedy např. třídu *Židle*, pak obsahuje informace o tom, jaké hodnoty se u jednotlivých židlí budou ukládat – *barva, tvar, výrobce, ...* – a metody jaké je možné pro tyto židle použít – *natřít, opravit, změnit potah, ...* . Instance třídy jsou pak jednotlivé konkrétní židle mající svoji barvu, svého výrobce, atd. a je možné jim posílat zprávy, jako např. natřít na modro.

Vytváření instancí se v každém případě děje v souhlase s popisem, který je obsažen ve třídě. Může se tak dít staticky nebo i dynamicky za běhu programu. Naše chápání objektových systémů se vyznačuje tím, že po nich požadujeme možnost dynamického vytváření objektů. Jednoduchou situaci s děděním mezi třídami a zavádějící naši další



Obr. 12 Třída, nadtřída, podtřída

grafickou reprezentaci tříd a objektů, kdy vidíme třídu (angl. class), její podtřídu (angl. subclass) a její nadtřídu (angl. superclass), máme znázorněnu na obr. 12. Zde je i znázorněno, že ve třídě se uvádí popis lokálních dat objektů: (angl. instance variables) a metod, které určují možné chování objektů; (angl. instance methods). O vztazích mezi třídami budeme podrobněji hovořit později, zde jen vidíme, že v podtřídě jsou popsána nová lokální data jejích instancí. Tedy situace, kdy instance podtřídy obsahuje lokální data jak zděděná tak popsána na své úrovni. Na obr. 13 vidíme ještě navíc tzv. ***metatřídu*** (angl. metaclass), která vytváří třídu jako svoji instanci. Třída, aby mohla přijímat zprávy (jako např. „vytvoř svoji novou instanci"), musí být také objektem – tj. instancí jiné třídy. Touto třídou je tedy metatřída. Jednotlivé programovací systémy se navzájem dost odlišují podle vlastností metatříd.



Obr. 13 Metatřída

Zapamatujme si nyní jen, že v metatřídě se popisují vlastnosti tříd (lokální data – angl. class variables – a metody pro chování tříd – angl. class methods), ve třídách se popisují vlastnosti objektů, které nejsou třídy.

VZTAHY TŘÍD & OBJEKTŮ

Podobně jako jsme mohli dříve vidět různé vztahy mezi objekty tak tyto plně existují i v systémech s objekty a třídami. V jisté paralele existují i mezi třídami, ale jen do určité míry.

Co je zcela obdobné, je *vztah dědění (IS A)* > mezi třídami. Následník dědí od svého předchůdce veškeré vlastnosti a podobně může i podtřída dědit popis instancí. Dědění mezi třídami se tedy fakticky projevuje tím, že veškeré vlastnosti instance některé třídy jsou určeny celou hierarchií předchůdců této třídy. K dědění nedochází mezi objekty, ale mezi třídami.

Vztah pán–sluha (HAS A), což druhý ze základních vztahů mezi objekty, již nemá tak přímočarou obdobu v systémech se třídami. Samozřejmě, že i zde je možné skládání objektů. Popis skládání je však uveden ve třídě. Není to nic, co by nás mělo překvapit, protože se jedná o důsledek toho, že instance se plně popisují ve třídě. A proto i popis skládání objektů je uveden ve třídě té instance, která je složena z více objektů. Samozřejmě, že i tato vlastnost instancí se díky možné hierarchii dědění projeví i u instancí následníků.

Jak jsme právě uvedli, je v systémech se třídami *dědění* záležitostí dědění mezi třídami. Díky dědění mezi třídami pak instance tříd – objekty – dědí všechny vlastnosti o všech nadtříd své třídy. Pro vztah předchůdce následník mezi třídami platí obdobné možnosti, které jsme uváděli minule pro tento vztah u objektů.

POUŽÍVÁNÍ TŘÍD

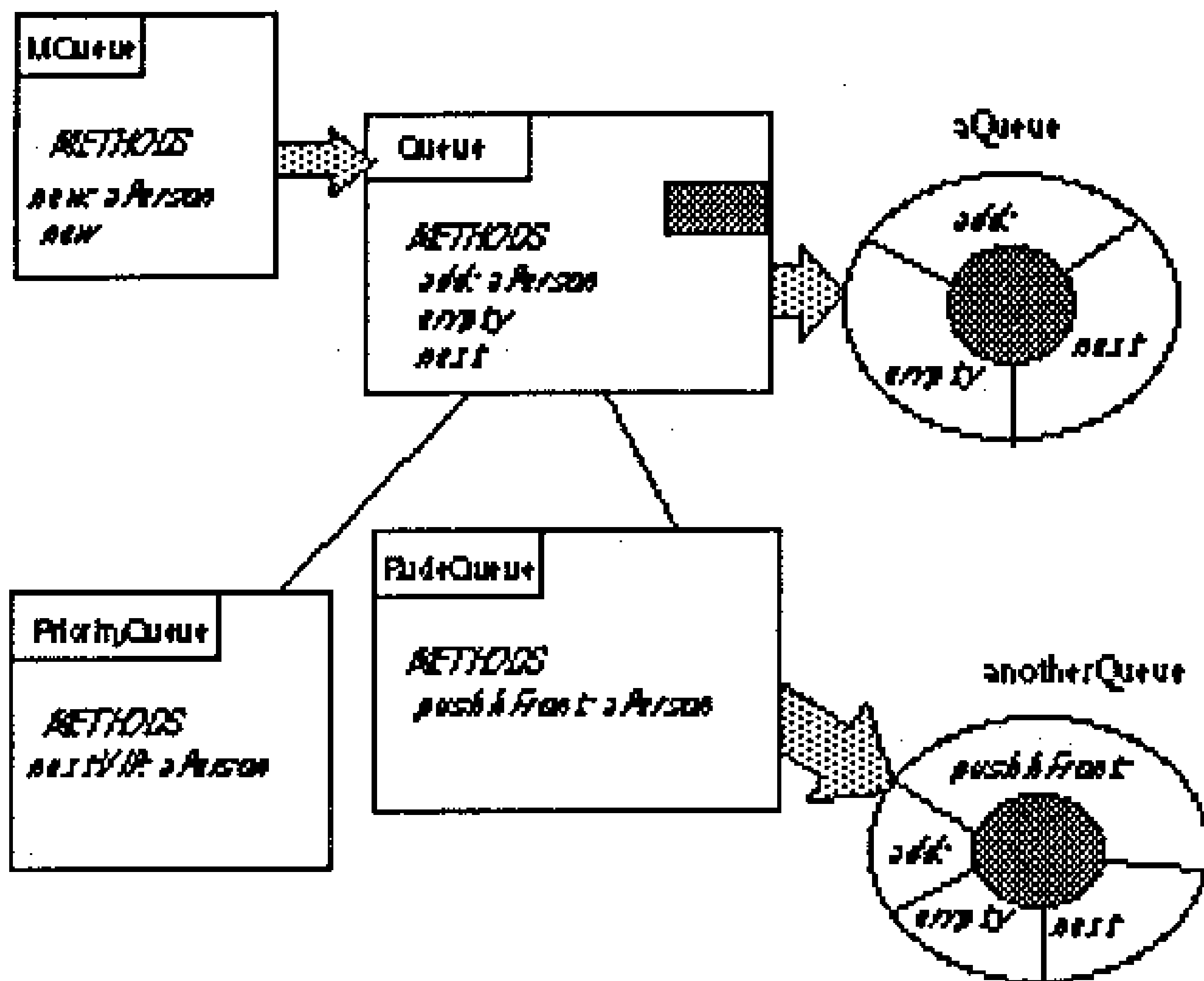
Pokusme se vyřešit jednoduchý příklad, totiž implementaci fronty v prostředí se třídami.

Na obr. 14 vidíme ideální situaci. Máme zde třídu *Queue*, jejímiž instancemi jsou konkrétní fronty. S frontou lze provádět operace přidej někoho (*add:*), vyber dalšího na řadě (*next*) a je možné zjišťovat, zda je daná fronta prázdná (*empty*). Třída *Queue*, která dovoluje jen korektní chování, má dva následníky, totiž třídu fronta s prioritami – *PriorityQueue*, a třídu dovolující neslušné chování – *RudeQueue*. Tyto třídy popisují reálnější situace ze života.

V třídě s prioritami je možné, aby byl přímo z fronty vytažen někdo – kdo má přednost, což je popsáno pomocí metody *nextVIP*. Ve frontě s neslušným chováním je možné předeběhnout na začátek fronty díky metodě *pushInFront*..

Při řešení podobných úloh je dobré se podívat po knihovně již existujících tříd a vybrat tu, která nám dovolí splnit úkol a přitom vynaložit co nejméně programátorského úsilí. Takovou třídou bývá v objektových knihovnách třída popisující uspořádané sady (angl. *OrderedCollection*). My si ukážeme dvě řešení s použitím třídy *OrderedCollection*. První z nich nebude dobře – protože, co se povede na poprvé? Teprve druhé řešení je správně.

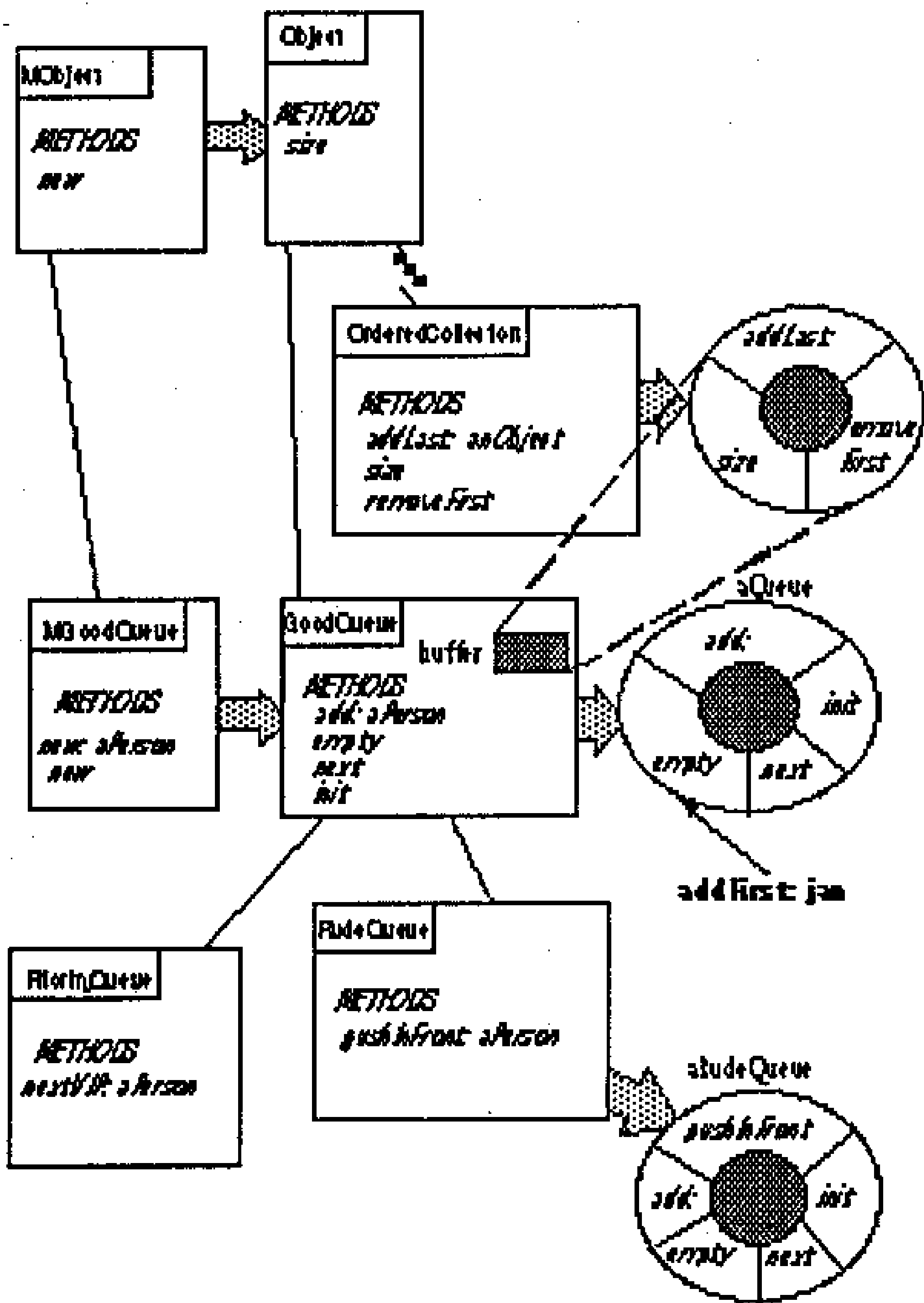
Na obr. 15 vidíme první řešení, které není dobře, protože, jak je na obr. naznačeno rozumí instance třídy *BadQueue* také zprávě *addFirst: jan*, což by nemělo být přípustné – ve frontě s korektním chováním přeci není možné předbíhat (ani pro nějakého Honzu).



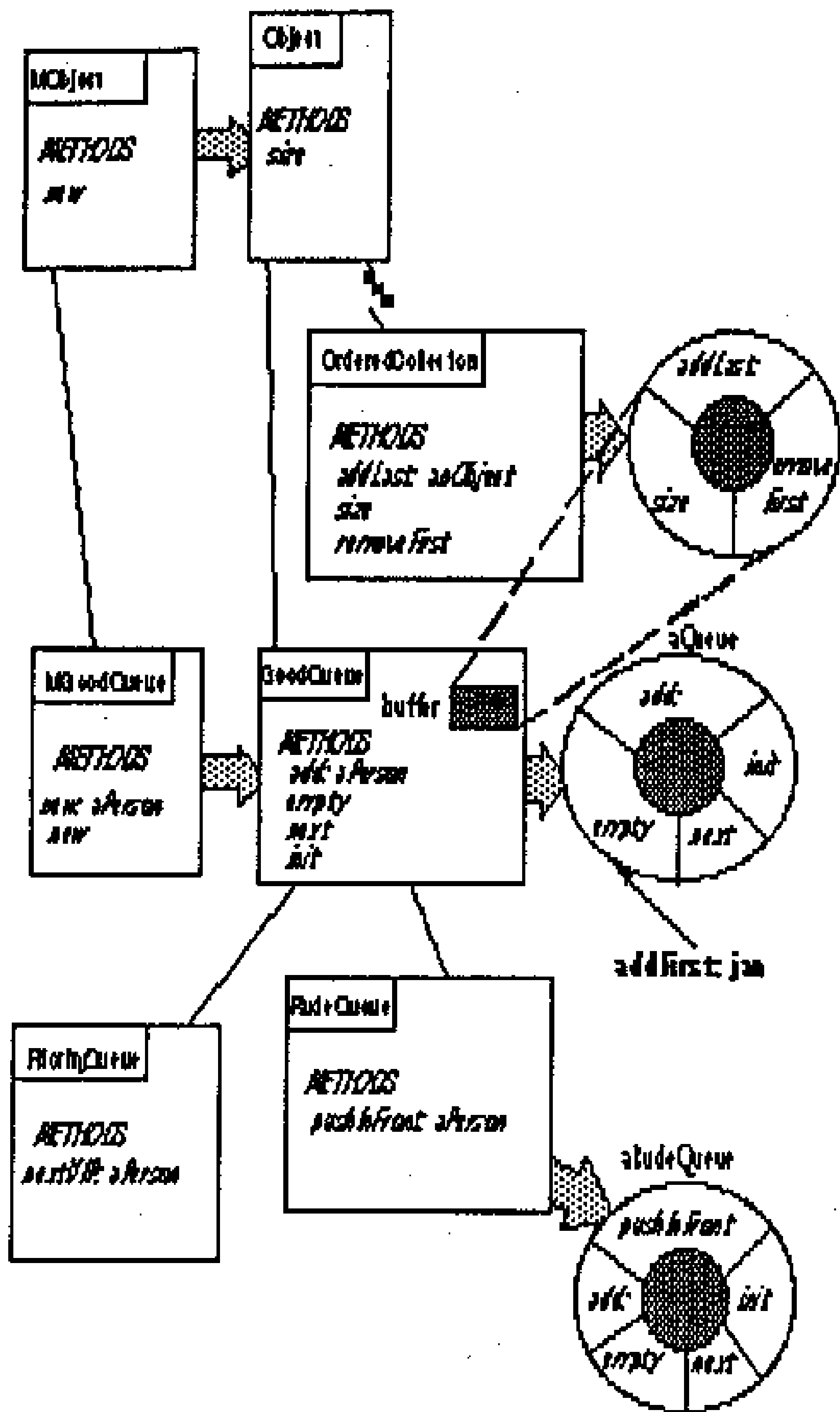
Obr. 14 Fronta

Podívejme se nyní na důvod, proč k tomu dochází. Jak vidíme na obr. 15, tak třída *BadQueue* je následník třídy *OrderedCollection*, tedy instance *aQueue* dědí všechny metody i od třídy *OrderedCollection*; např. tedy i metodu *addFirst:*. Dědění je zde použito nesprávným způsobem! Dědění tedy není vždy správný vztah mezi třídami!

Obr 16 pak udává řešení správné, kdy se jedná o skládání objektů – o vztah pán (tj. „třída" *GoodQeue*) – sluha (tj. instance třídy *OrderedColection*). Lokální proměnná instancí pojmenovaná *buffer* v sobě ukrývá instanci třídy *OrderedCollection* a rozhraní objektu pak zabezpečuje, že jen ty metody popsané ve třídě *GoodQueue* je možné použít při přístupu ke konkrétním frontám – např. k frontě *aQueue* z obr. 16. Zde poslání zprávy *addFirst: Jan* neuspěje!



Obr. 15 Špatná fronta



Obr. 16 Fronta dobře

ZÁVĚR

Povšimněme si na závěr nejprve té skutečnosti, že třídy chápeme jako přirozené rozšíření objektových systémů, jako vhodný programátorský nástroj. Výše uvedený příklad implementace fronty jasně ukazuje na to, že i v systémech, kde jsou nabízeny prostředky pro objektově orientované programování, se lze dopustit výrazné chyby.

Shrňme si, že příspěvek se snažil dát základní orientaci v pojmech používaných pro objektově orientované paradigma. Zabývali jsme se pouze těmi nejzákladnějšími pojmy. Nicméně jejich pochopení je důležité pro správnou aplikaci objektově orientovaného přístupu.

Autor: Ing. Jiří Polák, CSc
Katedra počítačů
FEL ČVUT
Karlovo n. 13
121 35 Praha 2
tel (02) 29 34 85
fax (02) 29 80 98