Zkušenosti s objektově orientovaným přístupem při tvorbě aplikací

Vojtěch Merunka, Tomáš Vaňák

Katedra počítačů FEL ČVUT v Praze, Katedra informatiky PEF ČZU v Praze

V současné době již objektově orientovaný přístup postupně ztrácí svůj nádech exotičnosti a dostává se mu v oblasti tvorby software maximální pozornosti. Nesmí se však zapomínat, že OOP je především výsledkem snahy řešit již delší dobu trvající softwarovou krizi, která se projevuje rostoucí složitostí programů překonávajících rostoucí sémantickou mezeru, nebezpečně nízkou spolehlivostí programového vybavení a neúměrnými náklady na pracovní sílu pro programování a údržbu provozovaného softwaru, což je něco poněkud jiného než skvělá budoucnost informatiky projevující se v nabídce stále „barevnějších" a lepších (a větších a dražších!) verzí programů sloužících k uspokojení potřeb zákazníka, jak tvrdí prodejci všeho druhu. Autoři se v článku snaží naznačit **příčiny daného stavu, základní principy OOP** a zkušenosti s méně rozšířenými, ale od základu objektovými, prostředky jako je použití **objektových databází** a systémů založených na jazyku **Smalltalk**.

Úvod - příčina krize

Schopnost řídit a využívat informace se rychle stalo kritickým dělítkem mezi úspěšnými a mezi poraženými na poli ekonomiky. Až donedávna (samozřejmě na vzácné výjimky) mnoho počítačových odborníků předpokládalo, že starý výpočetní model monolitické softwarové aplikace bude neustále a relativně jednoduše schopen se přizpůsobovat novým požadavkům.

Ještě poměrně nedávno se o zpracování účtů společnosti starali úředníci, kteří dovedli ručně zpracovat milióny „transakcí". Dnes jsou nahrazeni počítači, ale přibyly nám místnosti plné programátorů, kteří generují milióny instrukcí aplikačního kódu. Z určitého pohledu lze tedy prohlásit, že armádu úředníků vystřídala armáda programátorů. Studie za studií ukazuje, že většina organizací zabývajících se informačními systémy vkládá do udržování již existujících systémů 80 % času a námahy a pouze zbylých 20 % věnuje vývoji nových systémů. Před érou výpočetní techniky byl hardware relativně malý, modulární a hlavně levný. K dispozici bylo množství jednoduchých třídících, počítacích, psacích a tisknoucích mašinek, který se relativně jednoduše zapojovaly a konfigurovaly. Jejich uživatelé v malých i velkých organizacích se k nim chovali tak, jako se dnes mnozí chovají ke svým PC.

S nástupem mainframů se vše změnilo. Mainframy byly zpočátku vyvíjeny ve výzkumných laboratořích vládních agentur a univerzit, hlavně pro vojenské použití. Jejich výkon umožňoval provádět operace tisíckrát rychleji než jakékoliv dřívější zařízení. Tyto počítače umožňovaly spouštění větších a komplexnějších programů a kombinovat operace, které byly

dříve rozděleny mezi několik jednodušších strojů, do jednoho komplexního běžícího programu. V tomto bodě počítačové historie bylo samozřejmostí, že **větší je lepší**. Tento přechod měl jeden nepříjemných dopad, který pociťujeme dodnes, a to **obrovské náklady.** Výrobci hardwaru i softwaru investovali velké sumy do vývoje a podpory **nových verzí systémů**, které pak museli prodávat za astronomické částky už jen pro to, aby přežili. Nové systémy však postupně musely být stále více a více **kompatibilní** s jejich předchozími verzemi. Jejich zákazníci se snažili zlevnit provoz pomocí současného spouštění co největšího možného počtu běžících úloh.

Stejně tomu bylo i na poli softwaru. Nastala exploze vývoje zákaznického softwaru, která postupně vytvářela pro trh stále „lepší" nástroje (vyšší programovací jazyky, generátory aplikací, report writery, CASE prostředky). Zároveň vznikaly nové operační systémy. Vše nakonec vedlo k **zakonzervování** zastaralé von Neumannovy architektury počítače, ke **krizi programování** a k pokusům ji řešit právě objektově orientovanými architekturami.

Slepá ulička

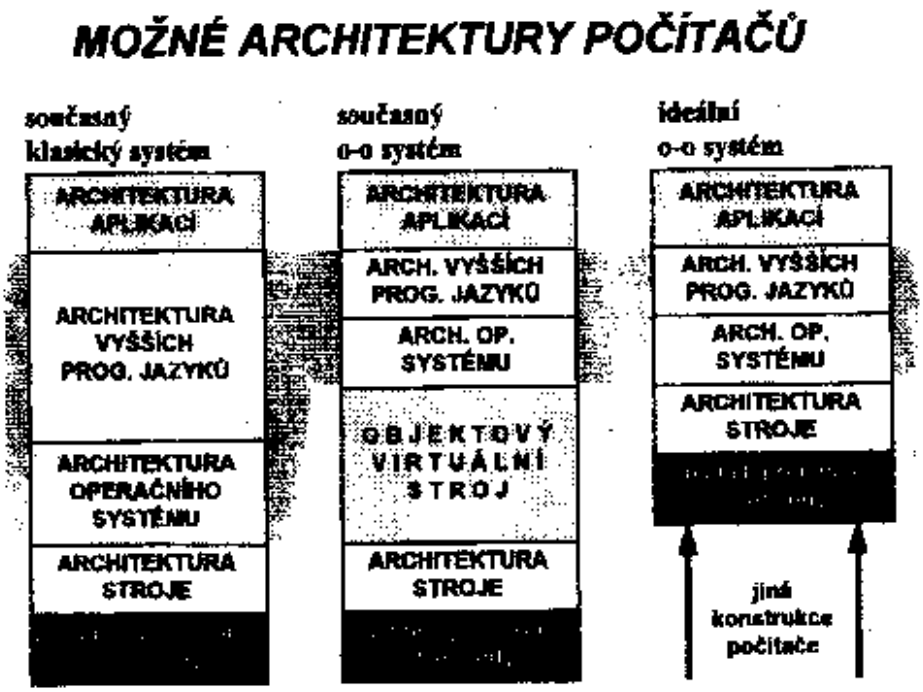
S nástupem osobních počítačů a pracovních stanic, které jsou dnes stejně výkonné jako dřívější (samozřejmě ne dnešní) mainframy, ale mají nepoměrně nižší pořizovací a udržovací náklady, dochází k **odklonu od mainframů** jako jediného možného návrhu výpočetního systému. Mnoho nových softwarových aplikací se dnes může provozovat na PC. Kromě toho, že pro PC existuje velké množství softwarových balíků, které se dají víceméně jednoduše přizpůsobovat pro koncového uživatele, mají PC i další „výhodu": aplikace mohou obsahovat grafický interface a to nejen okna a tlačítka, ale také nejrůznější multimediální efekty (obrázky, animace, zvuk, obchodní grafika).

Se zavedením PC se však hlavní problém nevyřešil, v podstatě se pouze zmodernizovaly a převzaly monolitické aplikace z mainframů na PC. Na mainframech jsme měli jen jeden centrální počítač, na kterém odděleně běželo současně více monolitických aplikací. Nyní máme více PC, kde na každém z nich běží jedna monolitická aplikace. Pokud se budeme tohoto vývoje držet dostaneme se k super PC, na kterém poběží více aplikací podobným způsobem jako na mainframech. A udržování aplikací stojí a bude stát peníze. Jakmile více aplikací bude sdílet více společných dat, bude se muset postavit interface, který bude zprostředkovávat předávání dat z aplikace do aplikace, což je však další software, jehož provoz spolkne další prostředky. Skutečný problém totiž nespočívá ani v hardwaru ani v softwaru. Skutečný problém spočívá v **architektuře celého informačního systému**. Je třeba docílit toho, aby všechny prvky systému spolu inteligentně komunikovaly a byly schopny se **levně a rychle přizpůsobovat měnícím se požadavkům.** V dnešní době se stále více ukazuje, že zatím jediným možným způsobem, který je schopen uspokojivým způsobem výše naznačené problémy vyřešit, je objektově orientovaný přístup.

Řešení - objektově orientovaný přístup

Všechny výše zmíněné problémy s počítači mají základ v původní **von Neumannovské architektuře počítače**, která dnes neodpovídá abstraktním vyjadřovacím schopnostem moderních programovacích jazyků a požadavkům současných uživatelů na abstraktní komunikaci s počítačem. Bezchybná implementace současných výpočetních systému na klasické von Neumannově architektuře (vNA) je vzhledem ke své programové složitosti téměř neřešitelný problém (na obrázku naznačeno šedým pozadím). Objektově orientovaný přístup, je právě zatím jedinou úspěšnou variantou vyřešení tohoto problému. Jeho

úspěšnost spočívá v tom, že objektový model výpočtu předpokládá **jinou architekturu stroje** než je klasická vNA. Dnes, v polovině 90 let, se výhody OOP již neprojevují jen na speciálně zkonstruovaných počítačích (několik jich bylo skutečně za posledních 20 let vyrobeno), ale začínají se výrazně prosazovat i na klasické architektuře stroje, pokud je softwarově „vylepšena" tzv. objektovým virtuálním strojem.



Model výpočtu

V OOP se na rozdíl od klasického pojetí operuje pouze s **objekty**, které popisují jak datovou, tak i procesní stránku modelované problematiky. Objekt je určitá jednotka, která modeluje nějakou část reálného světa a z funkčního pohledu víc odpovídá malému kompaktnímu programu jako automatu (majícímu svoji aktivitu definovanou svými stavy, přechody a interakcemi s ostatními objekty), než jedné proměnné příslušného datového typu, i když z programátorského pohledu takovou proměnnou je. Objektový systém je potom souborem takovýchto vzájemně na sebe působících malých programových celků.

Datová povaha objektu je dána tím, že objekty se skládají z příslušných vnitřních dat - **složek**, což jsou v rozumném případě opět nějaké jiné objekty. Funkční povaha každého objektu je dána tím, že každý objekt má jakoby okolo svých vnitřních dat obal či zeď, která je tvořena množinou malých samostatných částí kódů, jenž jsou nazývány **metodami**.

Metody slouží k tomu, aby popisovaly, co daný objekt **dokáže dělat** se svými složkami. Se složkami nějakého objektu lze manipulovat (číst, nastavovat, měnit) pouze pomocí kódu nějaké metody tohoto objektu. Každý objekt dovoluje provádět jen ty operace, které **povoluje** jeho množina metod. Proto se hovoří o **zapouzdření dat** uvnitř objektů.

Množina povolených operací s objektem se nazývá **protokol objektu**, což je také z hlediska vnějšího systému jeho jediný a plně postačující popis (charakteristika). Popis vnitřní

struktury objektu (data) je vzhledem ke svému zapouzdření a závislosti na metodách z hlediska vnějšího systému nedůležitý.

V objektovém modelu výpočtu pracujeme pouze se dvěma možnými operacemi s objekty. První z nich je **pojmenování** nějakého objektu. Druhou je tzv. **poslání zprávy**. Zpráva představuje žádost o provedení operace - metody nějakého objektu. Součástí zprávy mohou být tzv. **parametry zprávy**, což jsou vlastně **data** - objekty, které představují **dopředný datový tok** (ve směru šíření zprávy) směrem k objektu přijímajícímu danou zprávu.

Poslání zprávy má za následek provedení kódu jedné z metod objektu, který zprávu přijal, tak tento zmíněný kód také většinou dává nějaký **výsledek** v podobě nějakých dat - objektů, které představují zpětný datový tok ve směru od objektu - příjemce zprávy k objektu - vysílací zprávy (tj. v opačném směru k šíření zprávy). Vzhledem k možnostem kódů metod se výsledky po poslaných zprávách neomezují pouze na hodnoty jednotlivých složek objektů, protože jsou dány libovolně složitým výrazem příslušné metody nad množinou všech složek objektu sjednocenou s množinou parametrů zprávy.

Běžící objektově orientovaný program je tvořen soustavou mezi sebou navzájem **komunikujících objektů**, který je řízení především sledem vnějších událostí z rozhraní programu. Objektová aplikace nepotřebuje mít hlavní program, který běží od svého „begin" ke svému „end". Pokud použijeme nějaký progresivní programovací nástroj, tak můžeme programy měnit, doplňovat a ladit za jejich chodu.

Obecně se model posílání zpráv popisuje pomocí okamžiku určení kódu, kterým se provede určitá operace. Rozlišujeme tedy tzv. **pozdní a brzkou vazbu** kódu. Chápe se tím doba, kdy je znám kód metody, kterou se provede činnost způsobená posláním zprávy. V objektových systémech se setkáváme především 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.

Je úplnou samozřejmostí, že objektově-orientované principy neslouží jen k lepšímu návrhu grafického uživatelského rozhraní. Dobrý program se rozezná právě podle toho, že objekty používá i **„uvnitř výpočtu"** a ne jenom pro ovládání tlačítek, oken a menu.

Polymorfismus

Koncept **poslání zprávy a vykonání metody** nahrazuje koncept **volání funkce** (podprogramu) v klasických výpočetních modelech. Na rozdíl od volání funkce je tu však od sebe v případě použití pozdní vazby odlišen požadavek (tj. poslání zprávy) a jeho provedení (vykonání metody) objektem přijímajícím zprávu, což dovoluje posílat stejnou zprávu různým objektům s různým účinkem. Takovéto objekty jsou potom z pohledu těchto zpráv navzájem zaměnitelné.

Polymorfismus tedy 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 svojí metodou. Programátor není nucen brát ohled u významově podobných operací na to, že jsou požadovány od různých objektů.

Skládání objektů

Skládání je **nejdůležitější**, ale také nejvíce **opomíjenou** hierarchií mezi objekty. Skládání objektů znamená, že vnitřní data objektů jsou opět nějaké jiné objekty. Složíme-li například nějaké dva objekty do objektu nového, potom tento nový objekt může v kódech svých metod využívat funkčnost v něm složených objektů, přičemž funkčnost v něm složených objektů je vně skládajícího objektu vlivem zapouzdření ukrytá.

Můžeme také vytvářet objekty s jediným vloženým objektem uvnitř. Tato zdánlivě nadbytečná struktura má svůj smysl v tom, že zapouzdřením jediného objektu dovnitř nového sice neziskáme nová data, ale dodáme k těmto již existujícím datům novou interpretaci - a to právě pomocí nových metod obklopujícího objektu, což má v praktický význam. Pomocí takovéhoto skládání objektů potom můžeme na úrovni programovacích prostředků vyjádřit rozdíl (tj. chování = metody) mezi jedním prvkem samostatně a mezi jednou množinou s tímto jedním prvkem, což lze využít například v databázově orientovaných programových aplikacích a nebo dodávat stejným datům v různých kontextech různou interpretaci. Složený objekt staví novou zeď, která nově určuje, které z metod vnitřních objektů a hlavně jak budou použitelné. Zde je ukryto zdůvodění, 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.

Skládání objektů proto neslouží jen k prostému ukládání dat, ale i k **modelování funkčnosti** objektů, kdy si nadřazený objekt (tzv. pán) vybírá z funkčnosti jemu podřízených objektů (tzv. sluhové).

Třídy objektů

Velmi důležitým pojmem v oblasti OOP je pojem **třídy objektů.** Máme-li v systému velké množství objektů, které mají sice různá data, ale shodnou vnitřní strukturu a shodné metody, potom je výhodné pro tyto objekty zavést jeden speciální objekt, který je nazýván **třída**, a který pro všechny tyto objekty se shodnou strukturou shromažďuje popis jejich vnitřní struktury - tzv. **šablonku** a jejich **metody,** což přináší kromě úspory paměti i možnost sdružování objektů podle typů. Takto popsané objekty jedné třídy jsou nazývány **instancemi** této třídy.

Rozdělení objektů na třídy a instance však mění model výpočtu, protože instance obsahují jen data a metody jsou uloženy mimo ně v jejich třídě. Je-li tedy instanci poslána zpráva, tak instance musí požádat svoji třídu o vydání příslušné metody. Kód metody je poté pouze dočasně poskytnut instanci k provedení

V některých systémech (např. Object-Pascal nebo C++) jsou třídy implementovány pouze jako **abstraktní datové typy** a ne jako objekty (jako např. ve Smalltalku nebo CLOSu a na obrázku), což kromě jiných omezení znamená, že nemohou být vytvářeny či modifikovány za chodu programu.

Dědění mezi objekty

Další vlastností je **dědění mezi objekty**, které nám umožňuje definovat vnitřní strukturu a metody nových objektů pomocí definic jiných již existujících objektů. To, že se při popisu a

programování metod nových objektů odkazujeme na již existující objekty, nám umožňuje tyto nové objekty definovat pouze tím, jak se od stávajících objektů liší, což přináší kromě úspor při psaní programů také možnost vytvářet „nadtypy" a „podtypy" mezi objekty, vyčleňovat společné vlastnosti různých objektů atp.

Umožňuje-li systém přímo (ne zprostředkovaně přes jiné objekty) dědit od více než jednoho objektu, potom podporuje tzv. **vícenásobné dědění** (multiple inheritance), jinak se jedná o tzv. **jednoduché dědění** (single inheritance, tj. od nejvýše jednoho objektu).

Dědění versus skládání

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, a to ochranu lokálních dat - zapouzdření. Na úrovni implementační architektury programového systému lze prohlásit, že skládání je základní hierarchií a dědění je hierarchií odvozenou, jak dokazuje např. architektura systémů CLOS a Smalltalk. Lze si dokonce za určitých zjednodušujících podmínek, kdy nebudeme brát v úvahu např. výkonnost a rozlehlost systému, představit objektově orientovaný systém bez dědění, který se bude **vůči svému uživateli** navenek tvářit úplně stejně jako jiný systém řešící stejnou úlohu za pomoci dědění. Bez skládání objektů bychom však ztratili objekty samotné a hypotetický systém by nebylo možné sestavit.

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 závisejících především na **zvládnutí technik objektově orientované analýzy a návrhu,** což bohužel přesahuje kapacitu tohoto článku. Proto se spokojme alespoň s výčtem několika faktů, které obecně pro dědění platí,

- **uživatele dědění nezajímá**. Pokud budete vysvětlovat základy OOP začátečníkům, uživatelům objektových programů nebo manažerům, nepleťte jim hned od počátku hlavu s děděním, třídami apod. Základními pojmy jsou objekt, zpráva, metoda, polymorfismus a skládání objektů.
- **dědění není podmínkou polymorfismu.** Na to, aby dva objekty reagovaly na stejné zprávy, není třeba tyto objekty spojovat do hierarchie dědění - pro polymorfismus stačí, aby u obou objektů byly příslušné metody. Pokud nás nesvazuje nějaký konkrétní programovací jazyk (např. Object-Pascal či C++), tak dědění je jen jednou z možností, jak polymorfismus objektů zajistit.
- **dědění není jediným implementačním nástrojem** pro zajištění vzájemných souvislostí mezi objekty. Uveďme si klasický manuálový příklad: Udělejme objekty třídy Bod (s hodnotami souřadnic x,y) Potom můžeme realizovat např. Úsečku pomocí dědění z Bodu, kde doplníme ještě souřadnice toho druhého bodu (např. x2, y2). Tak a je to. Dědění přineslo úsporu kódu - ať žije OOP. Nebylo by však lepší vše implementovat jako objekt složený z dvou bodů? Zkusme se nad tím zamyslet hlavně pro případ, že se budou měnit vlastnosti (metody, či systém souřadnic,...).

Dědění mezi třídami

Vztah dědění a vztah třída-instance je na sobě navzájem nezávislý. Existují dokonce i objektové systémy bez tříd a s děděním nebo naopak. Je však pravdou, že naprostá většina objektových systémů je založena na kombinaci dědění a tříd.

V případě použití tříd se dědění odehrává pouze mezi třídami. Kromě dědění metod se tu navíc objevuje i dědičnost mezi šablonkami instancí tříd. Je-li nějaké instanci poslána zpráva, tak tato instance zachová podle třídně-instančního modelu požádá svoji třídu o poskytnutí příslušné metody k vykonání. Pokud třída metodu nemá, nastupuje dědění mezi třídami. Když je metoda nalezena, je předána instanci k provedení.

Delegování, alternativní aktorový model

Hierarchie dědičnosti i vazby třída-instance slouží z pohledu modelu výpočtu k podpoře sdílení společných vlastností pro více objektů, což se projevuje především sdílením kódů metod. na mechanismus sdílení je také možné nahlížet jako na **doplňování do funkčnosti** objektů potřebné přídavné chování z jiných objektů.

Tzv. **aktorový model** jako alternativní model objektového výpočtu je založen na myšlence, že objekty (používaný název v tomto kontextu je „aktory") v systému nemusejí implementovat všechny pro ně požadované funkce, přičemž ty funkce, které přesahují okruh identity jednoho objektu (a které by se jinak do objektu sdílely děděním), mohou být dynamicky převáděny ke zpracovávání jiným objektům. Namísto sdílení kódů od předků objektů či od tříd v aktorovém modelu dochází k přenosům částí výpočtu mezi aktory navzájem, což znamená, že v případě potřeby využití nějakého společného či obecného kódu tento kód není děděn ani jinak do provádějícího objektu sdílen, neboť výpočet probíhající na jednom aktoru je dynamicky přenášen - **delegován** na jiný (vesměs za tímto účelem nově vytvořený) nezávislý aktor. **Delegování (delegation) stejně jako dědičnost a nebo třídně-instanční vazba zabraňuje duplikacím stejných kódů u více objektů.**

Operace **delegování** je formálně totožná s operací poslání zprávy. Pro zpracovávání aktorových zpráv je však charakteristický **asynchronní paralelismus,** který se projevuje tím, že objekt posílající zprávu **nečeká na dokončení jejího zpracování a přijetí návratové hodnoty,** ale hned pokračuje v provádění následujícího kódu (tj. v posílání následujících zpráv). Takovéto zprávy jsou označovány jako zprávy typu **fork** na rozdíl od klasických zpráv, které jsou označovány jako typ **wait,** protože způsobují čekání na provedení jimi vyvolané metody.

Vzhledem k tomu, že aktorové zprávy nemohou z důvodu zmíněného paralelismu přímo realizovat zpětné datové toky, jsou v případě potřeby tyto **zpětné datové toky** ve směru od příjemce zprávy k vysílači zprávy realizovány jako **dopředné datové toky** jiných zpráv posílaných (po dokončení zpracování metody vázané na primární zprávu) od příjemce první zprávy k vysílači první zprávy.

Pro bližší objasnění aktorového modelu si představme následující příklad: Váš šéf (objekt) vám nařídí (zpráva), aby jste (zprávu přijímající objekt) vyplnili na psacím stroji formulář o vaší služební cestě. Protože psací stroj nemáte, tak si ho půjčíte u kolegy, zeptáte se ho, jak se zapíná, a jak se do něj vkládá papír, on vás to naučí,...., formulář sami na stroji vyplníte, potom stroj vrátíte a vyplněný formulář dáte šéfovi. To byl klasický objektový model

výpočtu. Lze to ale i jinak. Když zjistíte, že nemáte psací stroj, tak zajdete za sekretářkou a požádáte ji (delegujete na ni vaši přijatou zprávu), aby vám sama formulář na svém stroji vyplnila. Protože víte, že ho bez vaší asistence vašemu šéfovi po vyplnění vrátí (přenos výsledku), tak se o věc již nestaráte a děláte si svoje další věci (asynchronní paralelismus).

I když aktorové modely mají původ v experimentálních systémech pro modelování znalostí, tak je i u třídně-instančních systémů s děděním však výhodné, pokud to **dovoluje implementační prostředí a použitá metodologie analýzy a designu** a podporuje zadání problému, **využití** vlastnosti delegování a asynchronního paralelismu nejen v implementaci, ale již ve fázi **analýzy a návrhu.**

Závislost mezi objekty

Vztah závislosti mezi objekty, který také považujeme za důležitou hierarchii objektů, bývá úspěšně využíván při modelování grafických uživatelských rozhraní a při tvorbě nejrůznějších simulačních modelů. Z nejpoužívanějších objektových programovacích jazyků jej však přímo podporuje pouze Smalltalk. V C++ či Object Pascalu se musí implementovat pomocí hierarchie skládání.

Ve vztazích mezi objekty rozeznáváme dva typy objektů: řídící objekty - tzv. **klienty** a řízené objekty - tzv. **servery.** Žádá-li nějaký klient provedení nějaké operace od serverů, tak posílá zprávu, neboť zpráva je i zde jedinou možností, jak spustit nějakou operaci, na rozdíl od standardního poslání zprávy, kdy je třeba znát příjemce zprávy (a udržovat na něj např. referenci), v případě závislých objektů klient nepotřebuje znát svoje servery, protože oni sami jsou povinni svého klienta **sledovat** a zprávy od něj **zachycovat.** Systém podporující závislost tedy dovoluje jednodušší mechanismus **šíření zpráv** mezi objekty, jejichž počet a vlastnosti se za chodu aplikace mohou průběžně měnit.

POROVNÁNÍ NEJPOUŽÍVANĚJŠÍCH P.J.

	poly-morfis-mus	pozdní vazba	třídy jsou objekty	dědič-nost	závis-lost	para-lelis-mus	non OOP
Smalltalk	ano	ano	ano	jedno-duchá	ano	ano	ne
CLOS	ano	ano	ano	více-násobná	ano	ne	ano
Objective C	ano	ano	ano	jedno-duchá	ne	ne	ano
C++	jen s děděním	jen pomocí VMT	ne	více-násobná	ne	ne	ano
O-Pascal	jen s děděním	jen pomocí VMT	ne	jedno-duchá	ne	ne	ano

Nutnost uchovávání objektů - databáze

Vzhledem k potřebě komunikace jednotlivých aplikací a potřebě jejich opakovaného spouštění, je třeba zajistit nějakým způsobem **uchovávání objektů.** Z tohoto důvodu existují služby pro ukládání a vybírání objektů. V současné době má zelenou zajištění perzistence dat pomocí různých variant **architektury klient - server.** Jejím základním smyslem je dynamicky ukládat a vybírat objekty tak, jak jsou využívány jednotlivými běžícími aplikacemi.

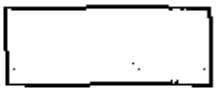
Zde se **objektové a relační databáze velmi liší** ve filosofii práce s daty vzhledem k uživatelům databáze. Relační databáze svým uživatelům poskytuje prostředky, jak lze pomocí programů běžících v operační paměti pracovat s daty na discích, takže na logické úrovni se všechna data databáze presentují jako disková (tablespaces, files,...). Objektové databáze na rozdíl od relačních vytvářejí svým uživatelům na svých rozhraních zdání toho, že všechna data jsou uložena v operační paměti podobným způsobem jako běžné proměnné, jak je známe z vyšších programovacích jazyků a o práci s diskem se interně stará systém, který neustále odlehčuje operační paměti odkládáním dat na disk a naopak načítá z disku do paměti (tzv. „swizzling"), čímž se zajišťuje i aktualizace a synchronizace stavu báze dat.

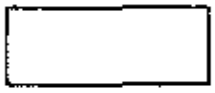
Současné **velké objektové databáze**, jejímž představitelem je Gemstone, jsou téměř vždy přirovnávány a testovány s relačními databázemi ORACLE, Ingres, Informix, Progress a nebo Sybase, protože patří do stejné velikostní třídy (Gemstone podporuje až 3200 uživatelů a až 4 mld. objektů v jedné databázi) a v porovnávacích testech s databází ORACLE rozhodně nezůstává pozadu (čím je báze dat strukturovanější, tím je objektové řešení např. v rychlostech dotazování výhodnější).

V poslední době se objevují nové verze relačních databázových systémů, které do relační struktury dovolují ukládat i jiné než „klasické" datové typy (např. OLE objekty, grafiku,...) a dovolují definovat do určité míry i funkční chování uložených dat (např. triggery). O těchto produktech jejich výrobci s oblibou prohlašují, že se jedná o objektové databáze. jedna věc je ale **datový model** a druhá je **druh dat,** které lze do příslušného datového modelu uložit. Pokud je nosičem databáze relační tabulka se všemi svými relačními vlastnostmi (normalizace, primární a sekundární klíče, operace relační algebry,...), potom je databázový systém z pohledu své funkčnosti nutně relační a ne objektový. Tak jako existují relační databáze s „objekty" (obrázky, zvuky, vnořené OLE,...), tak i existují skutečně objektové databáze ve kterých jsou objekty pouze textové nebo číselné povahy (tj. bez obrázků a zvuků).

MOŽNOSTI ŘEŠENÍ DB APLIKACÍ

strana klienta	strana serveru
vlastní DB generátory aplikací (ORACLE forms, reports, ...)	**relační DB server** (ORACLE)
Emb. SQL v klas. prog. jazyce (Cobol, Fortran, C, Pascal)	**relační DB server** (ORACLE)
Spec. generátory aplikací (PowerBuilder, Gupta)	**relační DB server** (ORACLE)
OOPL klient (Smalltalk, Objective-C, ev. C++)	**relační DB server** (ORACLE)
OOPL klient (Smalltalk, C++)	**objektový DB server** (ObjectStore, Gamstone)


objektový datový model


relační datový model

Technologie RDBMS byla velmi úspěšná, protože podstatně zjednodušila a zkvalitnila práci v oblasti podpory jednoduchých předefinovaných datových typů spolu s dnešního pohledu jednoduchými operacemi (např. selekce, projekce, spojení). V relačních databázích vše pracuje dobře, pokud se veškerá data dají jednoduše převést do hodnot v tabulkách a používají se pouze výše naznačené jednoduché operace. Pokud ovšem aplikace obsahuje **data komplexní**, grafy, obrázky, audio, video atd., jsou možnosti implementace takových dat v relační databázi **velice omezené**. Informace je tu dána totiž nejen samotnými daty (objekty), ale také jejich **vzájemnými vazbami**, jako například skládání, závislost a podobně. V objektových databázích jsou tyto závislosti modelovány přímo a jsou součástí rozhraní databázového stroje serveru. Tento rozdíl ilustruje následující příklad: Předpokládejme, že modelujeme proces ukládání auta na noc do garáže. V **ODBMS** je auto objekt, garáž je objekt a celou činnost zabezpečíme jednoduchou operací. V **RDBMS** se data celého auta musejí rozdělit do tabulek podle stejného typu, takže auto musíme rozdělit na součástky - kola zvlášť, písty zvlášť, ... a vše pečlivě označit klíčem. Ráno pak musíme auto zase složit dohromady. Použití RDBMS jako serveru pro aplikace moderního typu tedy přináší dva problémy: a) zvýšené nároky na programování, což se sice dá obejít použitím různých nadstaveb nad relačním strojem, b) zpomalení systému.

Vhodný programovací jazyk - SMALLTALK

Smalltalk byl vyvíjen v Kalifornii v **Palo Alto Research Center** (PARC) kolektivem vědců vedených dr. Alanem Kayem a dr. Adelou Goldbergovou v letech 1970-1980. Předmětem celého výzkumu, který byl financován v největší míře firmou Xerox, byl projekt **„Dynabook"** pro vývoj osobního počítače budoucnosti. Počítač Dynabook se měl skládat z **grafického displeje** formátu listu papíru o velikosti přibližně A4 s jemnou **bitovou grafikou**, klávesnici, v té době novou periférií - **perem** (stejném jako u dnešních „pen-počítačů"), později nahrazeným **myší**, a jeho součástí měl být i **síťový interface**. Pro vzhled systému byla poprvé na světě použita překryvná **okna**, vynořovací **menu** a později i **ikony**. V průběhu 70. let bylo dokonce vyrobeno několik prototypů takových počítačů, které

obsahovaly jednotné softwarové současně plnící úlohu operačního systému i programovacího jazyka s vývojovým prostředím. Právě tento software dostal název Smalltalk.

Ve Smalltalku, který byl jako projekt dokončen na začátku 80. let, se nejvíce odrazily prvky z neprocedurálního jazyka **LISP** a z prvního objektově orientovaného jazyka **Simula**. Část týmu v PARC zůstala a založila pod vedením A. Goldbergové firmu **ParcPlace Systems**, která rozvíjí Smalltalk dodnes, jiní spolu s A. Kayem odešli do firmy **Apple Computer**, kde poté uvedli na trh první dostupný komerční osobní počítač s grafickým uživ. rozhraním (GUI). Smalltalk a jeho GUI byl v průběhu 80. let využíván zpočátku pouze na výkonných pracovních stanicích té doby, z nichž nejznámější byl **Tektronix 4404** z roku 1982. V USA vzniklo několik firem (např. Knowledge Systems), které již okolo roku 1985 používali Smalltalk pro náročné aplikace z oblasti expertních systémů, řízení výroby, řízení projektů apod. (např. program Analyst vytvořený na zakázku Texas Instruments).Smalltalk byl používán také ve výzkumu na vysokých školách. Myšlenka GUI Smalltalku dala během 80. let vznik systémům Macintosh OS, MS Windows, X-Windows apod. Jazyk Smalltalku přímo ovlivnil vznik programovacích jazyků Objective-C, Actor, CLOS, Object Pascal, C++, Oberon aj.

Syntaxe i sémantika jazyka Smalltalk je jednoduchá, ale nezvyklá a vyžaduje od začátečníka znajícího jiný programovací jazyk bohužel větší úsilí než je obvyklé např. při přechodu z Pascalu do C. Smalltalk elegantně využívá výhod třídně-instančního objektově orientovaného modelu, tj. skládání objektů, dědění, závislosti mezi objekty, polymorfnosti a vícenásobné použitelnosti kódu. **Jazyk je integrován s programovacím prostředím** (lze jej odstranit v hotové aplikaci), které je napsané taktéž v jazyce Smalltalk. Vše je přístupné včetně zdrojových kódů. Navenek se systém chová jako **jediný rozsáhlý program**, který je programátorem měněn (doplňován) za svého chodu. I když Smalltalk podléhá vývoji v oblasti OOP, kde stále udržuje náskok před jinými systémy, tak v tomto článku popsané vlastnosti systému jsou jeho součástí již **od roku 1976.**

Systém je natolik otevřený, že umožňuje nejen tvorbu vlastních programovacích a ladících nástrojů, ale i **změny v samotném systému** (syntaktický analyzátor, překladač, mechanismus výpočtu, debugger), což dovoluje pod Smalltalkem např. implementovat jiné programovací jazyky, doplňovat systém o vícenásobnou dědičnost, backtracking,... Mezi systémovými a doplněnými vlastnostmi není formálního ani funkčního rozdílu.

Smalltalk podporuje koncepci **metatříd, paralelní programování** (procesy, semafory), z dalších vlastností např. **ošetřování výjimek** v programu, **sledování verzí** programového kódu během programování s možností návratu do libovolného z předchozích stavů aj. Programátor může např. do proměnných přiřazovat **nejen data, ale i kód**. Kód může být v proměnné přeložen, ale nevyhodnocen a celek se chová jako objekt, který na poslání zprávy vrací příslušné hodnoty. Získaná hodnota samozřejmě záleží na stavu systému v době spuštění a ne na stavu v době vytvoření.

Čisté objektově orientované prostředí jazyka Smalltalk **neobsahuje příkaz skoku, podprogramu ani funkce**. V aplikaci dokonce nemusí být ani tzv. hlavní program, jak jsme zvyklí z procedurálních jazyků. **Aplikaci tvoří množina objektů**, kteří prováděním svých metod určují, jak mají reagovat na došlé zprávy. Běh aplikace začíná poslaním nějaké zprávy zvenčí (klávesnice, myš...), která způsobí posílání zpráv dalším objektům aplikace. **Program se za chodu chová jako simulační systém řízený sledem vnějších událostí.**

Smalltalk implementuje výhradně dynamický model, ale nenutí programátora používat ukazatele do operační paměti. Správu paměti včetně „garbage collection" řídí **systémové paralelní procesy**, které aplikační programátor nepotřebuje znát.

Smalltalk maximálně podporuje kreativní inkrementální programování. Ve Smalltalku je možné experimentovat s libovolně velkými částmi kódu vytvářeného programu, psát či překládat a odladťovat programy po částech a **měnit je za jejich chodu**. Každá nově naprogramovaná metoda je po svém napsání okamžitě (v reálném čase chodu GUI) překládána a zapojována do systému.

Program ani po své finalizaci neztrácí pružnost, neboť je možné ukládat na disk jeho tzv. „snímek". Tato technika umožňuje programátoru i uživateli pokračovat v systému (či hotovém programu) přesně od toho bodu, ve kterém program opustil. Kromě stavu pracovních souborů se uchová i obsah a vzhled jednotlivých oken, menu i běžících paralelních procesů.

Samostatnou kapitolou je architektura a výkonnost systému. Smalltalk totiž nevyužívá přímo strojový kód počítače, na kterém běží. Namísto toho překládá programy do tzv. byte kódu, který je za chodu programu překládán (a uchováván v cache paměti) do hostitelského strojového kódu. Účinnost tohoto řešení dosahuje podle typu úlohy hodnot přibližně od 0.5 do 0.95. Ve Smalltalku-80 je byte kód jednotný pro všechny podporované počítačové platformy od Apple a PC k mnoha typům pracovních stanic, což znamená že programy (hotové i rozpracované) mohou být okamžitě přenositelné z platformy na platformu. K tomuto přispívá i na OS nezávislý model grafických objektů a diskových souborů v systémové knihovně.

Přehled nejrozšířenějších systémů SMALLTALK

1. **Smalltalk-80** firmy ParcPlace - Digitalk. Je prodáván pod obchodním názvem **VisualWorks** (dříve ObjectWorks) a je koncipován pro širokou platformu počítačů od PC (Win, OS/2) přes počítače Macintosh až k řadě mnoha typů pracovních stanic Sun/SPARC, HP, DEC, IBM atd. VisualWorks obsahuje vizuální programování GUI aplikací, CASE tool pro tvorbu klient-server aplikací pracujících i s relačními databázemi a tzv. Chameleon View, které umožňuje za chodu přepínat GUI vzhled programů (Win, OS/2, Motif, Apple, OpenLook). Smalltalk-80 má velkou podporu jak v možnostech rozšiřování základního systému (knihovny objektů), tak i např. v napojení na databázové systémy.
2. **Visual Smalltalk** je také produktem firmy ParcPlace - Digitalk. Pracuje pouze pod operačními systémy MS Windows, OS/2 (existují i jeho starší verze označené Smalltalk/V pro samotný MS DOS) a pro operační systém počítačů Macintosh. Programovací jazyk je až na některá zjednodušení totožný s programovacím jazykem Smalltalk-80. Největší odlišnosti jsou v grafických knihovnách, z tohoto důvodu není zdrojový kód jednoduše přenositelný do Smalltalku-80 a naopak. Vývojové prostředí je také poněkud jednodušší než ve Smalltalku-80. Finalizované programy jsou na PC v podobě *.EXE souborů, ale nejsou přenositelné mezi platformami.
3. **IBM Smalltalk** je zatím ještě mladý systém odvozený ze Smalltalku/V a je velmi podobný Visual Smalltalku. Je součástí produktu **VisualAge** (CASE pro tvorbu GUI klient-server aplikací). Podporuje práci s relačními databázemi DB2, DB2/VSE, VM (SQL/DS), DB2/400, Microsoft SQL Server, ORACLE a SYBASE SQL Server. IBM

Smalltalk patří v zahraničí mezi nejprodávanější programovací nástroje v MS Windows a OS/2 a má podobně jako Smalltalk-80 mnoho možností k rozšiřování.

4. **Smalltalk/X** je produktem firmy Tomcat v Mnichově. Je do značné míry kompatibilní se Smalltalkem-80 a je zajímavý svým originálním způsobem vytvořenou vazbou na jazyk C s možnostmi překladu do strojového kódu. Je dostupný na jakémkoliv počítači s operačním systémem UNIX.
5. **Smalltalk DB** je DDL a DML jazykem objektově orientovaného databázového systému Gemstone (výkonovými a kapacitními parametry srovnatelný s relační databází Oracle).
6. **GNU Smalltalk** je produkt v rámci projektu GNU. V základní verzi neobsahuje ani GUI podporu. S tímto Smalltalkem není zatím možné vytvářet seriózní aplikace.
7. Enfin je produktem společnosti Easel Corporation z Burlingtonu v Massachusetts. Pracuje pod operačními systémy Windows (3.1 a výše), OS/2 a Unix. Použitá verze jazyka Smalltalk není kompatibilní s verzí Smalltalk-80. Obsahuje CASE pro tvorbu objektového modelu aplikace a pro generování uživatelských obrazovek. Součástí systému je i tzv. Mapping tool. Mapping tool umožňuje objektově orientovaným aplikacím pracovat s objekty, jejichž data jsou uložena v relační databázi - vytvoří vrstvu, přes kterou se vyvíjené aplikaci zdá, že je napojena na objektovou databázi. Vrstva obsahuje např. Object Query Editor, Object Request Broker atd. Tvůrce aplikace pak může svou pozornost věnovat objektům nemusí se starat o problémy vzniklé použitím relační databáze v objektovém prostředí.