

# **Objektově orientovaná tvorba softwaru**

Vojtěch Merunka

Robert Pergl

Marek Pícka



# Obsah

<b>OBSAH.....</b>	<b>3</b>
<b>1 OBJEKTOVĚ ORIENTOVANÉ PARADIGMA .....</b>	<b>8</b>
1.1 Softwarová krize .....	8
1.1.1 Současný stav .....	8
1.1.2 Historické souvislosti .....	8
1.2 Objektově orientovaná architektura výpočetního systému .....	9
1.2.1 Architektura počítače .....	9
1.2.2 von Neumannova architektura .....	10
1.2.3 Sémantická mezera.....	11
1.2.4 Metody překonávání sémantické mezery .....	13
1.2.4.1 Jiná konstrukce počítače .....	13
1.2.4.2 Zdokonalení v oblasti softwaru.....	14
1.2.5 Sémanticky strukturovaná paměť .....	14
1.3 Současné trendy rozvoje softwaru .....	15
1.3.1 Vizualní přístup.....	15
1.3.2 Objektově orientované operační systémy, počítačové agenti.....	16
1.3.2.1 institucionální počítač .....	17
1.3.2.2 osobní počítač.....	18
1.3.2.3 intimní počítač.....	18
1.4 Objektový software .....	18
1.4.1 Objekt jako abstraktní datový typ .....	19
1.4.2 Struktura objektově orientovaného programu.....	20
1.4.3 Možnosti implementace objektově orientovaného jazyka .....	20
1.5 Objektově orientované programovací jazyky .....	21
1.5.1 Vhodný programovací jazyk - Smalltalk .....	22
1.5.1.1 historie Smalltalku .....	22
1.5.1.2 Smalltalk dnes .....	23
1.5.1.3 současné systémy Smalltalk.....	24
1.6 Počítačové databáze .....	25
1.6.1 Současný stav .....	25
1.6.2 Vztah relační a objektové technologie .....	26
1.6.3 Současné trendy v databázové technologii .....	27
1.7 Programovací styly.....	28
1.7.1 Naivní technologie .....	29
1.7.2 Klasické - imperativní technologie .....	29
1.7.3 Strukturovaná technologie .....	29
1.7.4 Procedurální technologie.....	29
1.7.5 Modulární technologie .....	30
1.7.6 Neimperativní technologie .....	30
1.7.7 Funkcionální programování .....	30
1.7.8 Logické programování .....	30
1.7.9 Objektová technologie .....	31
1.7.10 Komponentové programování.....	31
1.8 Základní principy objektově orientovaného přístupu. ....	31
1.8.1 Model výpočtu .....	32
1.8.2 Polymorfismus .....	33
1.8.3 Skládání objektů.....	34
1.8.4 Třídy objektů.....	34
1.8.5 Dědění mezi objekty .....	35
1.8.6 Dědění versus skládání.....	35
1.8.7 Dědění mezi třídami.....	36
1.8.8 Delegování, alternativní aktorový model.....	37
1.8.9 Závislost mezi objekty .....	38

<b>2 ÚVOD DO SMALLTALKU .....</b>	<b>40</b>
2.1 První seznámení s Smalltalkem .....	40
2.1.1 Instalace Systému VisualWorks .....	40
2.1.1.1 Microsoft Windows .....	41
2.1.1.2 Unix/Linux.....	41
2.1.2 Práce se systémem .....	41
2.2 Objekty a zprávy .....	43
2.3 Uživatelské prostředí .....	45
2.4 Jazyk Smalltalku .....	46
2.4.1 Jména objektů .....	47
2.4.2 Konstanty .....	47
2.4.2.1 Číselné konstanty .....	48
2.4.2.2 Znakové konstanty .....	48
2.4.2.3 Řetězcové konstanty .....	48
2.4.2.4 Konstanty symbol .....	48
2.4.2.5 Bajtová pole .....	49
2.4.2.6 Pole jiných konstant.....	49
2.4.3 Proměnné .....	49
2.4.4 Výrazy.....	50
2.4.4.1 Zprávy .....	50
2.4.4.2 Sekvence výrazů .....	51
2.4.4.3 Kaskáda zpráv .....	52
2.4.4.4 Návrátový výraz.....	52
2.4.4.5 Použití pomocných proměnných .....	52
2.4.4.6 Bloky výrazů.....	53
2.4.5 Jednoduchý program.....	54
2.5 Instance, třídy, metatřídy a metody .....	55
2.5.1 Instanční proměnné.....	57
2.5.2 Instanční proměnné tříd .....	58
2.5.3 NameSpaces a ostatní proměnné ve Smalltalku .....	58
2.5.4 Architektura třídního systému .....	60
2.5.5 Polymorfismus a generický kód .....	62
2.5.6 Programování nových tříd a metod.....	63
2.6 Systém tříd ve Smalltalku .....	67
2.6.1 Systém tříd Collection .....	67
2.6.1.1 Příklady práce s instancemi systému Collection.....	70
2.6.2 Systém tříd Magnitude.....	71
2.6.3 Systém tříd Stream.....	73
2.6.3.1 Příklady použití.....	74
2.7 Řízení výpočtu ve Smalltalku .....	75
2.7.1 Větvění.....	75
2.7.2 Iterace.....	76
2.7.3 Operace nad sadami (Collection).....	76
2.7.3.1 Příklad použití metod z knihovny Collection - asSet .....	77
2.7.3.2 Příklad použití metod z knihovny Collection - asBag, occurrencesOf: .....	78
2.7.3.3 Příklad použití metod z knihovny Collection - select:, includes: .....	78
2.7.3.4 Příklad použití metod z knihovny Collection - inject:into:.....	78
2.7.4 Úloha polymorfismu v objektové algoritmizaci .....	79
2.7.4.1 Využití při návrhu nových objektových komponent – vzor double dispatching .....	79
2.7.4.2 Úloha datového modelování a polymorfismu při návrhu objektových algoritmů.....	81
2.8 Ošetřování chybových stavů.....	83
2.8.1 Mechanismus výjimek .....	83
2.8.1.1 Zachytávání výjimek.....	83
2.8.1.2 Vytváření vlastních výjimek a vyvolávání výjimek .....	84
2.8.2 Závislost objektů.....	86
2.8.2.1 Malé příklady využití závislosti objektů.....	88
2.8.3 Architektura MVC .....	90

2.8.4	Komponenty grafického uživatelského rozhraní (GUI).....	92
2.8.4.1	Dialogy.....	92
2.9	Paralelismus.....	93
2.9.1	Koordinace paralelních procesů.....	94
2.9.2	Malé příklady paralelních procesů.....	94
<b>3</b>	<b>POKROČILÉ TECHNIKY TVORBY SOFTWARE.....</b>	<b>96</b>
3.1	Vztah mezi informačním a řídicím systémem uvnitř organizace.....	96
3.2	Modelování požadavků na informační systémy.....	98
3.2.1	Myšlenka konvergenčního inženýrství.....	98
3.3	Životní cyklus vývoje informačního systému.....	99
3.3.1	Objektová analýza a návrh.....	99
3.3.2	Současné objektové metodologie.....	101
3.4	Řízení vývojového týmu.....	102
3.4.1	Softwarové profese.....	103
3.4.2	Organizace pracovních týmů.....	103
3.4.3	Algoritmizace rozpočtu.....	104
3.5	Alternativní metody řízení projektů.....	105
3.5.1	Vývoj pohledu na softwarové projekty.....	106
3.5.2	Tradiční přístup k řízení.....	106
3.5.2.1	Omezení tradičního přístupu.....	107
3.5.2.2	Rozsah použití.....	107
3.5.2.3	Požadavky na vývojové nástroje.....	107
3.5.3	Alternativní metody řízení.....	107
3.5.3.1	Řízení projektů metodou extrémního programování.....	108
3.5.3.2	Omezení metody XP.....	109
3.5.3.3	Rozsah použití.....	109
3.6	Návrhové vzory.....	109
3.6.1	Co to je návrhový vzor.....	110
3.6.2	Jak se návrhový vzor popisuje.....	110
3.6.3	Příklad návrhového vzoru Composite.....	111
3.6.3.1	Účel.....	111
3.6.3.2	Použití.....	111
3.6.3.3	Struktura.....	112
3.6.3.4	Součásti.....	112
3.6.3.5	Spolupráce.....	112
3.6.3.6	Důsledky.....	112
3.6.4	Známá použití.....	113
3.6.5	Příbuzné vzory.....	113
3.7	Softwarové metriky a metoda funkčních bodů.....	113
3.8	Techniky ladění programů.....	115
3.8.1	Úvod.....	115
3.8.2	Hledání místa vzniku chyby.....	115
3.8.2.1	Vizuální kontrola zdrojového kódu.....	115
3.8.2.2	Kontrolní výpisy.....	116
3.8.2.3	Prohlížení objektů za běhu – Inspector.....	116
3.8.2.4	Umělé zastavení programu.....	117
3.8.3	Debugger.....	117
3.8.3.1	Analýza výjimek z uživatelského rozhraní.....	118
3.8.4	Předcházení chybám.....	118
3.9	Úvod do refaktoringu.....	119
3.9.1	Základní refaktorizační úkony.....	119
3.9.2	Refaktorigace v prostředí VisualWorks.....	120
3.10	Problémy objektového návrhu.....	120
3.10.1	Dědění, hierarchie typů a taxonomie nejsou vždy totéž.....	120
3.10.2	Úspora za každou cenu.....	122
3.10.3	Vicenasobná dědičnost?.....	123

3.10.4 Třídy versus množiny objektů .....	123
3.10.5 Podtřídy nebo instance? .....	124
<b>4 OBJEKTIVĚ ORIENTOVANÉ DATABÁZE – GEMSTONE .....</b>	<b>125</b>
4.1 Úvod.....	125
4.2 Objektivě orientované a objektivě relační databáze .....	125
4.3 Objektivě orientovaný datový model.....	127
4.4 Jak vytvořit objektivou databázovou aplikaci.....	128
4.5 Gemstone .....	129
4.5.1 Historie Gemstone .....	129
4.5.2 Vlastnosti Gemstone .....	129
4.6 Programovací jazyk Smalltalk DB .....	130
4.6.1 Architektura programů ve Smalltalku.....	131
4.6.2 Rozdíly mezi Smalltalkem-80 a Smalltalkem DB.....	131
4.7 Příklad objektivě orientované databáze .....	132
4.7.1 Popis úlohy .....	132
4.7.2 Implementace úlohy.....	132
4.7.3 Program v jazyce Smalltalk DB databázového systému Gemstone .....	133
4.8 Příklady dotazů .....	136
4.9 Shrnutí.....	137
<b>5 OBJEKTIVĚ RELAČNÍ MAPOVÁNÍ – OBJECTLENS .....</b>	<b>138</b>
5.1 Transformace relačního datového modelu do objektivě orientovaného .....	138
5.1.1 Tvorba tříd objektů na základě entitních množin (tabulek).....	139
5.1.2 Úprava relačního schématu.....	139
5.1.3 Vytvoření vazeb skládání mezi objekty na základě klíčů v tabulkách .....	140
5.1.4 Návrh pohledů nad objektivě orientovanou strukturou databáze a konstrukce rozhraní.....	141
5.2 Zhodnocení vlastností objektivě relačního přístupu ObjectLens.....	142
<b>6 DISTRIBUOVANÉ OBJEKTY.....</b>	<b>144</b>
6.1 Object Request Broker (ORB).....	144
6.2 Existující standardy distribuovaných objektů.....	144
6.3 CORBA.....	145
6.3.1 Základní struktura .....	145
6.3.2 Interface Definition Language .....	145
6.3.3 Rozhraní CORBA .....	146
6.3.4 Object Services .....	147
6.4 Použití distribuovaných objektů ve Smalltalku .....	147
<b>7 JAZYK UML.....</b>	<b>149</b>
7.1 Úvod.....	149
7.2 Vznik UML.....	149
7.3 Diagramy UML.....	149
7.3.1 Použití diagramů UML .....	150
7.4 Vybrané diagramy UML.....	150
7.4.1 Diagram tříd.....	150
7.4.2 Diagram aktivit .....	151
7.4.3 Sekvenční diagram.....	152
7.4.4 Diagram spolupráce .....	152
7.4.5 Diagram případů užití .....	153
7.4.6 Stavový diagram .....	153
<b>8 SLOVNÍČEK POJMŮ .....</b>	<b>155</b>
<b>9 POUŽITÁ A DOPORUČENÁ LITERATURA.....</b>	<b>165</b>



# 1 Objektově orientované paradigma

## 1.1 Softwarová krize

### 1.1.1 Současný stav

Počítače za posledních čtyřicet let výrazně změny svoje parametry (cena, oblast použití, dostupnost, ...) a zejména několikanásobně vzrostl jejich výkon. Pokroky pozorujeme hlavně v hardware (větší paměť, rychlejší procesor, celkově menší rozměry, atp.). Mnohem menší změny nastaly v uplynulých letech ve způsobu činnosti samotného počítače (strojový kód, paměť, ...). Je vhodné dodat, že zde se změny měří mnohem hůře než prosté technické parametry počítačů. Věnujme se proto problematice software podrobněji a pokusme se zpochybnit populární tezi o dosavadním vývoji software. Zaměříme se tedy na proces tvorby software.

Na rozdíl od klasických disciplin, jakými je třeba stavitelství nebo automobilový průmysl, je softwarové inženýrství stále velmi mladou (což by vadit nemělo) a nevyspělou (což už vadí) disciplínou. Dnes téměř nikoho nenapadne postavit most nebo automobil "na koleně" bez pomoci standardů, metodologických technik, znalostí atp. Software je však stále ve velké míře vytvářen tímto řemeslným způsobem. Statistiky z USA nám ukazují velmi varující důsledky takového počínání. Jen méně než 50% programů je sestaveno podle dodržení projektového plánu a skutečně se používá. 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. Právě proto je třeba hledat příčiny selhávání programů, které je mnohem častější než selhání hardware.

Při tvorbě software se většinou jedná o metodu pokusu a omylu. Výsledný produkt je oproti spolehlivosti a flexibilitě používání hardware neuspokojivý. Úkolem softwarového inženýrství je od 60. let právě hledání teoretických a praktických prostředků vedoucích k významnému zvýšení spolehlivosti a efektivity programátorské práce v kombinaci s novými architekturami výpočetních systémů.

Celý problém je též umocněn obecným trendem růstu složitosti informačních systémů, který souvisí se stále masovějším nasazováním výpočetní techniky. Je jisté, že samotný proces tvorby software není bezproblémový, ale příčiny tohoto stavu je třeba hledat jinde. **Důležité je odhalit podstatu vývojových změn technického a programového vybavení počítačů, tj. podstatu změn hardware a software.** Proto je třeba se zaměřit na celý výpočetní systém jako na jeden celek (tj. spojení software a hardware), a hovořit o jeho architektuře, což je pojem dostatečně abstraktní, aby mohl být použit pro popis výpočetního systému.

### 1.1.2 Historické souvislosti

Před érou současné výpočetní techniky bylo technické vybavení kanceláří relativně malé, modulární a hlavně levné. K dispozici bylo množství jednoduchých třídících, počítacích a tisknoucích přístrojů, které 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 počítačových mainframů (sálových počítačů) se vše změnilo. Počítače byly zpočátku vyvíjeny ve výzkumných laboratořích vládních agentur a univerzit především 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 nepříjemný dopad, který pociťujeme dodnes, a to **obrovské náklady do zavádění informačních technologií**. Výrobci

hardwaru i softwaru investovali velké sumy do vývoje a podpory **nových verzí systémů**, což ovlivňovalo jejich cenu na trhu. Nové systémy však postupně musely být stále více a více **kompatibilní** s jejich předchozími verzemi. Jejich zákazníci se snažili zlevnit provoz pomocí současného spuštění co největšího počtu běžících úloh a budováním centrálních výpočetních středisek.

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, generátory sestav, CASE prostředky). Zároveň vznikaly nové operační systémy. Vše nakonec vedlo k **zakonzervování** zastaralé von Neumannovy architektury, ke **krizi programování** a k pokusům ji řešit právě objektově orientovanými architekturami.

S nástupem osobních počítačů a pracovních stanic, které jsou dnes minimálně stejně výkonné jako dřívější mainframy, ale mají nižší pořizovací a udržovací náklady, došlo 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 jsou PC používány velmi podobným způsobem. Jakmile více aplikací musí sdílet společná data, je nutné použít interface, které zprostředkovávají předávání dat z aplikace do aplikace, což je však další software, jejichž provoz vyžaduje další prostředky. Skutečný problém totiž nespočívá ani v hardwaru ani v softwaru. Skutečný problém spočívá v **architektuře celého informačního systému**. Je třeba opustit snahu budovat rozsáhlé monolitické aplikace s množstvím funkcí a docílit toho, aby všechny prvky systému spolu inteligentně komunikovaly a byly schopny se **levně, rychle a nezávisle na sobě přizpůsobovat měnícím se požadavkům**. V dnešní době se stále více ukazuje, že zatím jediným možným způsobem, který je v praxi schopen uspokojitým způsobem výše naznačené problémy vyřešit, je OOP.

Všechny výše zmíněné problémy s počítači mají příčinu v rostoucích požadavcích současných uživatelů na **abstraktní komunikaci** s počítačem, tj komunikaci, která více odráží realitu světa a umožňuje oprostít se od technických detailů počítače. Bezchybná implementace tak komplexních výpočetních systémů na klasické von Neumannově (vN) architektuře je vzhledem ke své programové složitosti téměř neřešitelný problém. OOP, je právě zatím jedinou úspěšnou variantou řešení tohoto problému. Jeho úspěšnost spočívá v tom, že objektový model výpočtu předpokládá **jinou architekturu stroje** než je klasická vN. 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 objektovým virtuálním strojem (viz. následující kapitola).

## 1.2 Objektově orientovaná architektura výpočetního systému

### 1.2.1 Architektura počítače

Pojem architektura počítače je velmi užívaným pojmem. Jeho interpretace se liší podle toho, jaký je konkrétní vztah každého člověka k počítači. Jinak chápe architekturu konstruktér, jinak programátor a jinak uživatel. Pro úplný popis je proto třeba definovat **různé úrovně architektury počítače**, vždy podle jednotlivých úrovní abstrakce. Každou úroveň (tj. konkrétní architekturu) lze potom definovat jako **funkční rozhraní** mezi touto a podřízenou nižší úrovní. Lze ji také definovat jako funkční popis hypotetického počítače pod daným rozhraním. Takovému popisu potom říkáme **virtuální počítač**. Virtuální počítač může být považován za dále nedělitelný.

Z celé této hierarchie je pro nás nejdůležitější existence **vzájemných rozporů** mezi jednotlivými úrovněmi. Tyto rozpory jsou současně hnací silou neustálého vývoje počítačů a také jim vděčíme za dnešní složitost počítačů. V souladu s výše uvedenými fakty lze v klasickém počítači rozlišit šest různých úrovní architektury počítače:

1. **Architektura výpočetního systému.** Je popis rozhraní mezi vnějším světem a počítačem. Toto rozhraní musí zvládnout každý uživatel počítače. V praxi je realizována pomocí hotových aplikačních programů.
2. **Architektura vyšších programovacích jazyků (VPJ).** Je to rozhraní mezi aplikačními programy a programovacími jazyky. Vyšší programovací jazyky jsou popsány množinou sémantických konstrukcí a výrazových prostředků. Kromě prostředí samotných programovacích jazyků sem náleží i další konfigurovatelné a programovatelné softwarové aplikace. Toto rozhraní musejí zvládnout nejen programátoři, ale i znalí uživatelé.
3. **Architektura operačního systému.** Je tvořena různými funkcemi a službami, kterými je počítač vybaven pro zabezpečení a pomoc pro běh programů vytvořených pomocí VPJ.
4. **Architektura stroje - rozhraní mezi hardwarem a softwarem.** Je funkční specifikace činnosti vlastního fyzického počítače. Zde je přítomen soubor prostředků pro implementaci operačního systému a VPJ. Tato architektura zahrnuje i **syntaxi a sémantiku strojového kódu a model výpočtu.**
5. **Architektura mikroinstrukcí a obvodů.** Nejnižší úroveň zabývající se přímou konstrukcí počítače jako stroje.

Každou úroveň lze ještě horizontálně členit podle jednotlivých funkcí, čímž se však v tomto textu není třeba hlouběji zabývat. Důležitý je fakt, že lze vytvářet nad holým počítačem (podle architektury stroje) "slupkovitě" **virtuální počítače**, které zastíňují nepotřebné detaily nižších úrovní a zároveň zastíňují jejich vnitřní vlastnosti a nedokonalosti. Virtuální počítač potom svému uživateli poskytuje abstraktnější a vyspělejší funkce, čímž **vytváří iluzi práce s dokonalejším počítačem.**

### 1.2.2 von Neumannova architektura

Počítače za půl století své existence doznaly nebývalého rozvoje. Dnešní stav je charakterizován exponenciálním rozvojem počítačových systémů, množstvím programovacích jazyků, množstvím aplikačního software, čímž počítače pronikají do všech sfér lidské činnosti. Architektura drtivé většiny dnešních počítačů vychází z modelu formulovaného ve 40. letech kolektivem vědců okolo matematika **Johna von Neumanna**. Model byl navrhován jako co nejjednodušší implementace počítače a vychází z matematické teorie Turingových strojů. Již sám von Neumann tvrdil, že po překonání technologických problémů bude třeba přistoupit k dokonalejší architektuře stroje, protože již v tehdejší době byly známe její nedostatky. Charakteristika von Neumannovy architektury stroje je následující:

1. **Počítač se skládá z paměti, procesoru (řadič + ALU) a vstupně/výstupních jednotek.** Tento rys architektury je pozitivní z hlediska relativně snadné konstrukce počítače. Jedná se o známé blokové funkční schéma počítače.
2. **Struktura počítače je neměnná, jeho chování se mění obsahem jeho paměti.** Toto je nejrevolučnější myšlenka vN modelu. Jejím důsledkem je universalita počítačů.
3. **V jedné paměti jsou instrukce i data.**
4. **Paměť je posloupnost buněk stejné délky (bytů) adresovaná jejich pořadovými čísly.** Je to velmi slabý rys architektury. V této souvislosti hovoříme o primitivnosti

paměti vN počítače. Pro funkci počítače by bylo mnohem výhodnější používat buňky proměnlivé velikosti podle ukládané hodnoty. K paměťovým buňkám je také výhodnější se dostávat skrze jejich hodnoty (asociativní výběr) a ne přes jejich pořadová čísla (adresní výběr).

5. **Program je posloupnost příkazů v paměti, které se sekvenčně provádějí.** Toto je další slabá stránka architektury - problém sekvenčního výpočtu. V praxi je většina úloh řešitelná paralelním způsobem.
6. **Instrukce jsou trojího typu.**
  - a. operace nad daty
  - b. přesuny dat procesor - paměť
  - c. změny toku řízení (cykly, podprogramy atd.)

Jediný potřebný typ pro popisy úloh jsou operace nad daty. Další dva typy instrukcí jsou pomocné, protože vN stroj nemá vlastní inteligenci a jeho programátor musí kromě požadovaných datových operací ještě popisovat detailní postup řešení problému, protože stroj sám není schopen nic provádět. To, že v programech musejí být navíc i tyto příkazové instrukce, je poslední slabou stránkou vN modelu počítače a v této souvislosti se hovoří o problému imperativního modelu výpočtu. (Všimněme si, že u většiny dnešních aplikačních programů uživatel zadává jen požadované operace nad daty. Zbylé dva typy operací však musejí být v systému také přítomny.)

7. **Obsah paměti je zakódován ve dvojkové soustavě.** Je to pozitivní vlastnost architektury z hlediska snadné konstrukce pomocí elektronických obvodů.

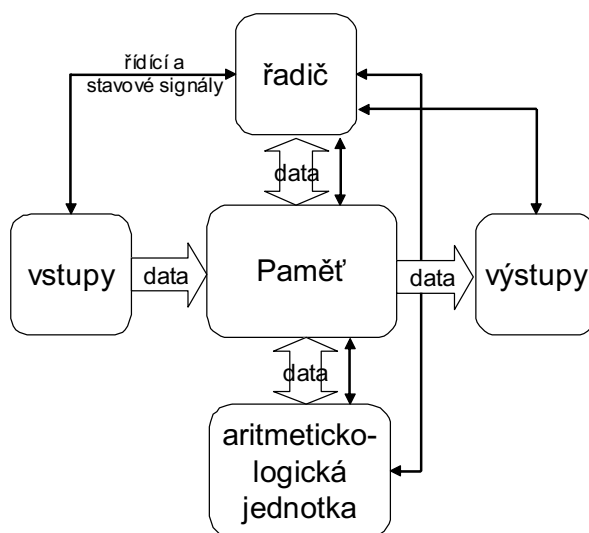


Schéma Von-Neumannovy architektury

### 1.2.3 Sémantická mezera

Od 60. let se architektura stroje kvalitativně nevyvíjí. Pouze se osvojily a staly samozřejmostí objevy z 50. let, které určitým způsobem zdokonalovaly vN architekturu. Jedná se především o přerušovací systém, vícerozměrné adresování paměti, proudové zpracování instrukcí, koprocesory atd. Vzhledem k pokroku technologií však **architektura počítače stagnuje**, protože se vyrábějí pouze rychlejší a menší stroje s větší kapacitou paměti.

Přibližně od 60. let proto tvůrce software trápí stále rostoucí problém - **sémantická mezera** (angl. *semantic gap*). Problém spočívá v primitivnosti paměti vN počítače a v imperativním modelu výpočtu

na jedné straně a složitými výpočetními systémy založenými na neimperativních modelech na druhé straně.

Bezchybná implementace velmi abstraktních výpočetních (a tím i komplexních) systému na klasické von Neumannově architektuře je totiž vzhledem ke své složitosti téměř neřešitelný problém (pokud jsou nějaké pokusy v tomto směru použitelné, pak jsou řádově pomalejší a méně efektivní). Proto se možné řešení nehledá ve zlepšování von Neumannovy architektury, ale ve **změně architektury stroje**, v jejím přiblížení abstraktní úrovni vyšších programovacích jazyků.

Software, který ovládá vN počítač, se totiž skládá z množství jednoduchých operací, které mění stav počítače, především stav jeho paměti. Příkazy určují kromě požadovaných operací nad daty i to, jak tyto datové operace provádět. Je třeba řídit přesouvání dat v paměti, řídit tok algoritmu. Úkolem programátora je potom bezchybně a beze zbytku všechno popsat včetně pořadí provádění. Takový styl uvažování však není člověku vlastní. Člověk uvažuje na mnohem vyšší úrovni a není mnohdy ani schopen ani ochoten psát úlohy na úrovni strojových instrukcí vN stroje (i když i v této oblasti lze získat zručnost).

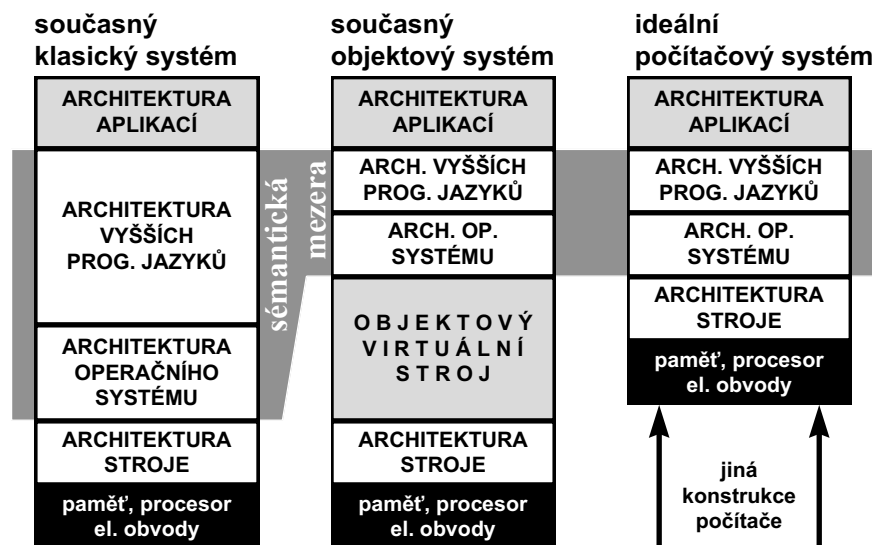
Výsledkem předchozích úvah je vznik vyšších programovacích jazyků a balíků aplikačních programů (databáze, tabulkové procesory atd...). V těchto systémech se jejich uživatel může vyjadřovat na mnohem vyšší úrovni. Dochází však k **rostoucímu rozdílu** mezi schopnostmi takových produktů (architektury výp. systému a VPJ) a mezi schopnostmi vN architektury stroje:

- Vyšší úrovně architektury používají abstraktní datové typy, lokální proměnné, složené datové typy, složité operace nad daty, paralelní procesy a jejich synchronizace a další složité vlastnosti.
- vN architektura ale nemá lokální proměnné, má pouze jednoduché datové typy (bajty nebo jejich násobky), nemá možnost provádět a synchronizovat paralelní procesy. Nezná jiný model výpočtu než ten nejprimitivnější - imperativní.

Jedním z největších nedostatků vN architektury je její primitivnost paměti. vN paměť kromě konstantní velikosti buněk a nevhodnému přístupu pomocí adres ještě nemá sebeidentifikaci hodnot v paměti. Z adresy buňky ani z hodnoty v buňce nelze poznat typ uschované hodnoty. Z toho plyne fakt, že v případě použití nějakého systému vyšší úrovně musíme mít softwarovými prostředky zabezpečenu identifikaci dat a převod našich složitých operací na primitivní strojové vN operace.

Kdyby šlo vše překonat softwarem, tak bychom vlastně nad primitivním vN počítačem vytvořili "obal" inteligentnějšího **virtuálního počítače**, který by vyřešil uvedené problémy. Naneštěstí jsou rozpory v sémantické mezeře natolik závažné, že je třeba zvolit jiné řešení. Důsledky vN architektury dokonce nutí i ve vyšších programovacích jazycích používat primitivní takzvané "vN operace". Jedná se především o přiřazovací příkazy umožňující přesuny dat a skoky pro změny toku řízení (příkazy cyklu). Fakt, že se sémantickou mezerou jsou opravdu velké potíže, se promítá i do různých vnějších podob operačních systémů a různých aplikačních programů.

## MOŽNÉ ARCHITEKTURY POČÍTAČŮ



Sémantická mezera v důsledku stagnace vN architektury způsobuje **krizi programování**. Jejím projevem je zřejmě i nechuť k výuce programování a následná víra ve všemocnost aplikačních programů na naší fakultě. *(Na zahraničních pracovištích však věnují výuce programování velikou pozornost u všech studijních oborů, protože si jsou dobře vědomi toho, že nejen budoucí tvůrci softwaru, ale všichni absolventi specializovaných oborů technicky a matematicky zaměřených vysokých škol potřebují znát základy algoritmizace a analýzy a návrhu informačních systémů.)* Krize programování přináší mnoho problémů, které se musejí řešit. Mezi nejzávažnější patří:

- **Složitě vyjadřování algoritmů a velké množství VPJ různých typů a určení.**
- **Nedoladitelné rozsáhlé programy.** Primitivnost architektury umožňuje výskyt sémantických chyb, které mohou způsobit závadu až v předem neodhadnutelně velké části následujícího kódu. Chyby navíc bývají i časově neodhadnutelné.
- **Problematická údržba a modifikace hotových programů** vlivem složitých VPJ.
- **Za chodu věnuje procesor více než 50% času systémovým programům.** Protože se sémantická mezera softwarově překonává různými překladači, interprety a službami operačního systému, spotřebovává se značné množství strojového času.

### 1.2.4 Metody překonávání sémantické mezery

#### 1.2.4.1 Jiná konstrukce počítače

Nejlépeším způsobem, jak se vypořádat se sémantickou mezerou, je **jiná konstrukce počítače**. vN architektura v takovéto nové architektuře bude zřejmě nějakým způsobem obsažena. (Přibližně stejným způsobem jsou v buňkách lidského mozku zachovány základní funkce, které mají i buňky primitivních jednobuněčných živočichů.) Nová architektura stroje ale musí převzít většinu funkcí, které nyní musí provádět software překonávající sémantickou mezeru. Jde o **zvýšení inteligence hardwaru**. Programy by se potom mohly spoléhat na jeho inteligenci. Nejznámějším projektem v této oblasti byl nezdařený ambiciózní japonský projekt páté generace počítačů.

### 1.2.4.2 Zdokonalení v oblasti softwaru

Problémy spojené se sémantickou mezerou lze překonávat i **různými možnostmi konstrukce VPJ na stávajícím hardwaru**. Přístupů je několik a lze je kombinovat mezi sebou. Zmíníme se o čtyřech nejdůležitějších:

1. Imperativní programovací jazyky
  - a. **Jazyky se silnou typovou kontrolou** (např. Pascal). Překladač kontroluje, jestli typy přiřazovaných hodnot odpovídají přesně deklaraci.
  - b. **Jazyky se slabou typovou kontrolou** (např. C). Typová kontrola stále existuje, ale je volnější. Díky slabé typové kontrole je možné provádět „rafinované“ konstrukce, je zde však větší nebezpečí chyb z nepozornosti programátora.
2. **Jazyky bez typové kontroly** (např. Smalltalk). Proměnné nemají předem určený typ a programátor je tedy sám zodpovědný za interpretaci hodnot v paměti a správný chod programu. Výhodou je větší vyjadřovací schopnost jazyka, na druhou stranu ale k odhalení některých chyb dojde až při běhu programu.
3. **Neprocedurální (neimperativní) jazyky**. Tyto jazyky jsou založeny na abstraktnějším vyjadřování. Dělí se podle matematického aparátu, na kterém jsou založeny na:
  - a. funkcionální (např. Lisp, Haskell): jejich základem je tzv. lambda kalkul a výpočet je z velké části založen na rekurzi.
  - b. logické (např. Prolog): programuje se pomocí výrokového počtu, tj. faktů a pravidel. Logické programování je ze všech jazyků nejabstraktnější způsob vyjadřování, kdy specifikujeme pouze co a ne jak chceme vypočítat.

Objektová orientace je v principu možná u kteréhokoliv přístupu (např. CLOS je objektový Lisp). Jazyk Smalltalk by se dal charakterizovat jako čistě objektově orientovaný imperativní jazyk (příkazy = posílání zprávy) s funkcionálními prvky (vlastní průběh výpočtu je řízen interními vazbami objektů).<sup>1</sup>

### 1.2.5 Sémanticky strukturovaná paměť

Architekturu se sémanticky strukturovanou paměti je možno považovat za nástupce vN architektury v nejbližší budoucnosti protože pouhé vylepšování vN architektury nemůže rozhodujícím způsobem vyřešit dosavadní problémy. Projekt "objektově orientovaného počítače", jak se tato architektura také někdy nazývá, pro první fázi vývoje počítá se **zachováním imperativního modelu výpočtu**. Pro dostatečné překonání sémantické mezery a tím následné zjednodušení a zvýšení výkonnosti softwaru totiž stačí hardwarově vytvořit **sémanticky strukturovanou paměť**. Taková paměť musí mít schopnost sebeidentifikace uložených hodnot.

Pro rychlou funkci počítače je nezbytné využívat nových vlastností architektury stroje. Je tedy nezbytné sestrojít paměť, která nám umožní většinu operací (doposud prováděných systémovým softwarem) implementovat přímo **pomocí obvodů** počítače. Nejdůležitějším problémem je otázka konstrukce mechanismu pro práci s takovouto pamětí. Tento mechanismus se označuje „adresování založené na způsobilosti“ (angl. *capability - based addressing*).

Taková paměť má svoji vnitřní strukturu jako **strukturu množin objektů schopných sebeidentifikace**. Paměť sama se musí na hardwarové úrovni starat o jejich správu (např. problémy fragmentace paměti, uvolňování nepotřebných bloků paměti apod.) Výpočetní systém se potom na vlastnosti paměti spoléhá

---

<sup>1</sup> Verze jazyka Smalltalk-76 byla i ve svém zápisu velmi podobná jazyku Lisp.

a pouze **posílá instrukce**, které mění stav jednotlivých objektů, pracuje s nimi atd. Způsobnost přístupu k objektu je potom soubor vlastností, kterými se daný objekt prezentuje svému uživateli.

Každý objekt tedy obsahuje kromě své zakódované vnitřní hodnoty i zakódovaný popis svých vlastností. Každý uživatel objektu (nějaký výpočetní proces) má také svoji identifikaci. Je-li volána nějaká operace, systém sám zavolá potřebnou operaci, která vrátí požadovaný výsledek.

Je zřejmé, že kromě zjednodušení uživatelského softwaru tato architektura zásadně řeší **problém bezpečnosti** uložené informace v počítači. Stroj "sám rozhoduje", kdo a jak smí s objekty manipulovat. Pro konstrukci tohoto typu paměti je třeba mít na paměti, že v ní uložená informace již **není homogenní a lineární**, jako ve vN paměti. V současnosti existují dva možné způsoby řešení:

1. **Paměť se sebeidentifikací.** (angl. *tagged architecture*) Sebeidentifikace tvoří s hodnotou integrovaný celek v paměti. Toto řešení je "čistší" z hlediska teorie, ale přináší konstrukční problémy spojenými s buňkami různé délky. Příkladem je existující experimentální velmi výkonný počítač SWARD firmy IBM.
2. **Paměť s deskriptory.** (angl. *descriptor architecture*) Identifikace je od hodnot oddělená a tvoří zvláštní popisnou část nazývanou deskriptor. Tato koncepce je mnohem propracovanější a dovoluje nám použít standardně vyráběné paměťové součástky. Celková organizace paměti je pak částečně podobná organizaci souborů na disku: Informace v deskriptorech (které analogicky odpovídají alokačním blokům na disku) obsahují identifikaci a ukazatel do bloků hodnot paměti (které odpovídají datovým sektorům souborů na disku). Příkladem je komerčně bohužel neúspěšný procesor Intel APX432.

Objektové architektury počítačů nemění architekturu stroje tak zásadním způsobem jako např. data flow; mění jen jednu část architektury - 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. Objektově orientovaný přístup především znamená **změnu práce s pamětí**.

### 1.3 Současné trendy rozvoje softwaru

V předstihu před předpokládanými změnami stroje postupují změny ve strukturách vyšších architektur, jako jsou architektury operačních systémů, vyšších programovacích jazyků a aplikačních programů. Tyto změny jednak reagují na existující sémantickou mezeru a také připravují podmínky pro dosud nevyvinutý hardware. Již v dnešní době jsou některé výsledky tohoto výzkumu využívány v praxi, a proto je třeba se o nich zmínit. Kromě nástupu **počítačových sítí a multimédií** se v tomto kontextu jedná o: **Vizuální přístup**

*"Ukaž na něco z toho, co vidíš". Nikoliv "Napiš něco z toho, co si pamatuješ".*

Principem vizuálního přístupu je snaha maximálně usnadnit netvůrčí uživatelskou a programátorskou práci. Vizuální vlastnosti softwarových aplikací bývají velmi často mylně (i záměrně z komerčních účelů) ztotožňovány s objektovými.

Při vizuálním programování se nejčastěji jedná o tvorbu uživatelského interfacu. Jsou však i systémy podporující tímto způsobem tvorbu celých programů. Všechny funkce potřebné pro práci se systémem jsou na obrazovce počítače reprezentovány ikonami nebo jinými grafickými symboly. Uživatel si potom nemusí přesně pamatovat jednotlivé příkazy, protože jejich grafická reprezentace na obrazovce je zároveň jejich náповědou.

### 1.3.2 Objektově orientované operační systémy, počítačové agenty

Průkopníky v oblasti objektově orientovaných operačních systémů jsou od poloviny 80. let firmy **Parc Place Systems** se produkty postavenými nad systémem **Smalltalk-80**, **Apple Computer** s operačním systémem **MacOS**, **Sun Microsystems** s operačním systémem **Solaris** a **Hewlett Packard** se svým produktem **New Wave** a **Distributed Smalltalk**. Objektově orientované operační systémy přinášejí následující vlastnosti:

1. **Stírají se rozdíly mezi klasickými funkcemi** operačních systémů (práce se soubory a spouštění programů) **a mezi ovládním a dokonce i tvorbou programů**. Vše se odehrává v jednotném objektově orientovaném prostředí.
2. **Jednotná objektová interpretace dat umožňuje vzájemná propojení aplikací**. Aplikace si mohou velmi jednoduše předávat data, vzájemně se volat apod.
3. **Operační systém je možné používat neimperativním způsobem**. Operační systém například sám volá příslušný aplikační program, protože uživateli stačí aktivovat potřebná data podle zásad **datově orientovaného přístupu** v kontrastu s klasickým **příkazově orientovaným přístupem**.

V souvislosti s objektově orientovanými operačními systémy se hovoří o tzv. **agentech** (terminologie dr. Alana Kaye). Počítačovým agentem se rozumí softwarové doplnění operačního systému o prvky umělého intelektu a představuje jeden ze směrů rozvoje **expertních systémů** a výzkumu v oblasti **umělé inteligence**. Mezi konkrétní schopnosti počítačového agenta by měly patřit ty rutinní činnosti, které dnes musí provádět lidská obsluha osobního počítače. Patří mezi ně například:

- **Podpora vlastní práce s počítačem**. Agent by měl být schopen poradit v kterékoli situaci při práci s osobním počítačem. Měl by být schopen vést uživatele při jeho práci (názorné nápovědy, reakce na chyby atd.).
- **Multiprocessing**. Agent by neměl při plnění svých úkolů blokovat operační systém. Na počítači by měl mít možnost pracovat současně uživatel a několik uživatelem dříve vyvolaných agentů na různých úlohách.
- **Pomoc při práci s bázemi dat**. Rozsáhlé báze dat (například multimedia apod.) totiž neumožňují jednoduchý přístup k informacím. K uloženým informacím se uživatelé musejí složitě dostávat. Agent by měl plnit funkci "knihovníka", kterému uživatel sdělí svoje požadavky např. formou dotazu.
- **Sledování bází dat**. Agent by měl pravidelně svého uživatele informovat a stavech různých databází, vybírat například z denního tisku články týkající se určitého tématu, sledovat některé důležité ukazatele v podnikové databázi a reagovat na ně.
- **Správa vlastních dat**. Agent by měl sledovat tvorbu a koordinaci souvisejících dat různého typu. Vychází se z předpokladu, že běžný dokument se skládá z heterogenních dat pořizovaných různými programy, nejčastěji textovým editorem, tabulkovým procesorem, databází a kreslicím programem.
- **Mít na starosti komunikaci s jinými počítači**. Extrémním případem je vzájemná komunikace agentů bez nutnosti přímé účasti uživatelů v určitých typech úloh.
- **Programovatelnost**. Uživatel musí mít k dispozici prostředky, pomocí nichž si bude moci implementovat nové nebo upravovat stávající schopnosti agentů.

Softwarové agenty lze dělit podle způsobu použití na

- **Softbots** (*Software Robots*), kteří poskytují konzistentní rozhraní mezi pracujícím uživatelem a softwarovým prostředím, například propojení WWW a Unixového shellu pro

jednotnou interakci s různými Internetovými aplikacemi a nebo rozšíření např. databázových systémů o vlastnosti známé z klasických expertních systémů.

- **Scheduling Agents** pomáhající uživateli organizovat a plánovat kalendář událostí.
- **News Filtering Agents** pomáhající uživateli organizovat a zpracovávat data v podobě nejruznějších článků a dokumentů dostupných v síti Internet a Usenet.
- **Email Filtering Agents** pomáhající uživateli organizovat a zpracovávat elektronickou poštu.

Podle modelu výpočtu lze agenty dělit na

- **deliberativní**, jejich architektura je založena na explicitní reprezentaci a představují symbolický model příslušné části reálného světa, přičemž činnost agenta je založena na symbolických manipulacích a porovnávání vzorů uvnitř tohoto modelu,
- **reaktivní**, jejich architektura je založena na interakci systému s vnějším prostředím pomocí definovaných pravidel bez explicitního vnitřního symbolického modelu příslušné části reálného světa a
- **hybridní** jako kombinace obou předchozích typů.

Softwaroví agenti jsou z pohledu jejich programové architektury založeni na **objektově orientovaném modelu výpočtu** se zaměřením na **paralelismus** a možnost definování chování objektů **pomocí faktů a pravidel** závislých na stavu daných objektů. Tento model, jenž představuje specializaci objektově orientovaného modelu výpočtu směrem k inteligentnímu chování objektů je v literatuře označován jako **agentově-orientovaný** (*agent-oriented*) model výpočtu používající tzv. **aktivní objekty** (*active objects*).

Ideální agent na počítači 5. generace by měl být svému uživateli společníkem, měl by hlídat a koordinovat činnost svého "pána". Hostitelský počítač by měl být napojen na síťové informační systémy včetně dnes běžných médií. Hardwarové nároky na takový počítač odpovídají současným výkonným notebookům. Bohužel z důvodu sémantické mezery má hardware dnešních počítačů dost práce s klasickými aplikacemi a obsluhou operačního systému a na další funkčnost již nezbyvá kapacita. Spolu s počítačovými agenty, objektově orientovanými operačními systémy a pátou generací počítačů se dr. Alan Kay zmiňuje o tzv. **třetí počítačové revoluci**. Jednotlivé revoluce z pohledu vztahu počítač - uživatel jsou následující:

### 1.3.2.1 institucionální počítač

*"Dojdi si na příslušné místo s žádostí o zpracování dat. Po přijetí tvého požadavku si několik dní (týdnů) počkej, než ti budou poslány výsledky"*

typický představitel: klasický sálový počítač IBM 370 nebo EC 1027.

- Dávkové zpracování dat.
- Alfnumerické vstupy a výstupy. Převážná vstupní a výstupní informace je v podobě textu nebo tabulky čísel.
- Příkazově orientovaný přístup.
- Jeden počítač náleží mnoha uživatelům.
- Isolovanost počítače.

### 1.3.2.2 osobní počítač

a) *"Dojdi si za svým podržickým, který umí ovládat osobní počítač a sděl mu, co potřebuješ. On ti to udělá během několika minut (hodin)."*

b) *"Sedni si ke svému osobnímu počítači a udělej si co potřebuješ. Po několika minutách (hodinách) sedění u počítače to máš hotové."*

typický představitel: Apple Macintosh. (Koncepte počítačů IBM PC je v některých rysech poplatná institucionálním počítačům, které se snaží stírat operační systém Windows)

- Interaktivní zpracování dat.
- Grafické vstupy a výstupy dat.
- Datově orientovaný přístup.
- Jeden počítač náleží jen několika uživatelům.
- Možnost připojení počítače na informační síť.

### 1.3.2.3 intimní počítač

*"Vyvolej si na svém počítači agenta a sděl mu co potřebuješ. On ti to udělá během několika sekund (minut, hodin)"*

V současné době ve výzkumu, který se přesouvá do oblasti aplikačního software.

- Vizuální přístup.
- Multimedia.
- Počítač je připojen na informační síť.
- Počítač náleží jednomu uživateli, stává se jeho osobní pomůckou. Má malé rozměry.
- Operační systém má umělý intelekt. Počítač pomocí agentů sleduje cíle svého uživatele a vykonává je nezávisle na něm.

Příkladem tohoto přístupu mohou být i různí průvodci (wizardi), kteří jsou dnes velmi často začleňováni do různých programových produktů.

## 1.4 Objektový software

V předchozí části textu jsme diskutovali vznik objektové orientace jako reakce na existující sémantickou mezeru. Dotkli jsme se této problematiky z hlediska architektury počítače. Následující kapitoly se již zabývají objektovou orientací z hlediska **softwaru a tvorby softwaru**.

Ještě před požadavkem sémanticky strukturované architektury byly vytvořeny základy pro tvorbu objektově orientovaného softwaru v jazyce **Simula**. Kromě hardwarových problémů lze totiž objektovou orientací elegantně řešit i problémy spojené s:

1. **modely operačních systémů**
2. **modely datových abstrakcí**
3. **modely výpočetních procesů**
4. **modely reprezentací znalostí v umělé inteligenci**

Pro programátorskou obec je příznačné, že si pod pojmem objektově orientovaného programování představují především jeho implementaci v konkrétním programovacím jazyce, nejčastěji v Object Pascalu nebo C++. Jazyků, které poskytují na různé úrovni možnost využití objektové orientace, je však od 70. let vyvinuto značné množství. Připomeňme jazyky Simula, Smalltalk, Actor, Eiffel, Objective C, Flavors, CLOS, Dragoon, Mainsail, ESP, Perl, Java, Beta, ABCL, Actalk, Plasma aj.

Pro naivní uživatele výpočetní techniky je příznačné, že pojem „objektově orientovaný“ ztotožňují s výhodami, které přinášejí grafická uživatelská rozhraní současných softwarových systémů.

### 1.4.1 Objekt jako abstraktní datový typ

Literatura se vesměs shoduje na několika kritériích, která musí systém splňovat, aby ho bylo možné považovat za objektově orientovaný, což lze považovat za vymezení objektu jako abstraktního datového typu (ADT):

- Údaje a jejich funkčnost, čímž rozumíme množinu operací, které lze s danou skupinou údajů provádět, jsou spojeny do jediné logické entity nazývané **objekt**. Operace jsou nazývány **metodami**. Hodnoty údajů i kódy metod jsou v objektu uzavřené (tzv. **zapouzdření dat** - *encapsulation*) a jsou přístupné jen tvůrci objektu. Uživatel objektu zná jen specifikaci metod jako tzv. **rozhraní objektu**, které je jediným prostředkem pro manipulaci s vnitřními hodnotami objektu (tzv. **zed' z metod**).
- Objekty mají schopnost **dědit své vlastnosti** od jiných objektů. Objekty mající různé údaje stejného typu a stejnou množinu operací nad nimi, jsou **instance** téže třídy. Třídy objektů jsou v systému uspořádány do orientovaného grafu podle dědičnosti. Je-li možné dědit vlastnosti jen z jediné třídy, hovoříme o **jednoduchém dědění**. **Dědění** nám umožňuje navrhovat nové objekty pouze tím, jak se liší od těch, jejichž vlastnosti dědí. Stejně charakteristiky objektů se nemusejí znovu vytvářet.
- Různé objekty jsou schopné různě reagovat v závislosti na svém konkrétním obsahu na stejnou **zprávu** (volání operace nad údajem objektu), což označujeme jako **polymorfismus**. Program využívající tuto vlastnost je označován jako **generický program**.
- Objekty mají kromě dědění také vzájemné vazby **skládání, závislosti a delegování**.
- **Metody** jsou programy obsahující operace nad údajem objektu. Objekty mezi sebou komunikují pomocí **posílání zpráv**.
- Identita objektů je nezávislá na jejich datovém obsahu.

Skutečnost, že je uživateli utajena implementace vnitřních hodnot objektu a jeho metod, je pro něj pozitivní v tom, že ji **nemusí znát**. Tím je zaručeno, že uživatel bude s objektem nakládat pouze tak, jak mu předepsal jeho tvůrce. Tato vlastnost je charakteristická pro tvorbu bezpečných programů při použití moderních programovacích technologií.

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 prvků grafického uživatelského rozhraní nebo pro tvorbu vstupních formulářů či výstupních sestav.

V rámci implementace objektově orientovaných systémů rozeznáváme **statický** a **dynamický** model. Pro plné využití výhod, které objektově orientované programování poskytuje, je nutný dynamický model. Pro statický model je typické, že z důvodů vynucených vN architekturou počítače, se systém v době běhu programu chová jako klasicky přeložený program. Objekty jsou použity pouze pro výhodnější zápis zdrojového textu programu.

## 1.4.2 Struktura objektově orientovaného programu

Čistý objektově orientovaný program se skládá z množiny objektů a má následující vlastnosti:

- U každého objektu je definováno jeho **chování** (metody) a jeho **vnitřní struktura**. Pro implementaci objektů je využito dědění, polymorfismus atd.
- Objekty mezi sebou komunikují pomocí **posílání zpráv**, což je také jediný druh příkazů v celém systému. Posíláním zpráv je možné také **předávat data mezi objekty**. Instrukce posílající zprávy jsou ukryty v kódech metod jednotlivých objektů. Je-li nějakému objektu poslána nějaká zpráva, začne se provádět ta metoda, pomocí které příslušný objekt na poslanou zprávu reaguje. V kódu této metody se může pracovat s vnitřními hodnotami objektu (také objekty se svým vlastním chováním).
- **Čistý OO systém nepotřebuje hlavní program ani dekompozici na podprogramy**. Jeho běh začíná vnější událostí z jeho rozhraní, která se interpretuje jako zpráva poslaná objektům, které mají na starosti vstup a výstup. Příslušné objekty zareagují tím, že začnou provádět odpovídající metody, jež posílají další zprávy. Během chodu systému se mohou vytvářet nové nebo mohou zanikat staré objekty.
- **V systému nemusí být uveden přesný algoritmus provádění operací od startu k cíli**. Programování se omezuje na implementaci příslušných objektů a na vhodné definování jejich vzájemných vazeb. Systém není řízen pevně daným algoritmem. Výpočet je řízen sledem posílaných zpráv (neboli sledem vnějších událostí) mezi objekty. Na čistý OO systém je možné se dívat jako na **asynchronní diskrétní simulační model**, který implementuje programátorem popisovanou podmnožinu reálného světa.
- Čistý OO systém podporuje **paralelní způsob výpočtu**.

## 1.4.3 Možnosti implementace objektově orientovaného jazyka.

Je třeba si uvědomit rozdíl mezi jazykem a jeho implementací. Jazyk je definován svojí gramatikou (tj. zápisem konstrukcí) a sémantikou (tj. významem konstrukcí). Jeho implementace je konkrétní program (sada programů) na počítači, které nám umožňují vytvořit a spustit kód.

Rozlišujeme tyto možné přístupy v implementaci:

1. **Kompilované versus interpretované provádění**. Kompilované provádění je typické například pro jazyky C a Pascal. Zdrojový text programu je před prováděním přeložen do strojového kódu počítače. V případě interpretovaného provádění je vytvořen pouze tzv. byte-kód, což je jazyk virtuálního počítače, který je v době provádění interpretován virtuálním strojem. Interpretované provádění je méně efektivní z hlediska rychlosti výpočtu, umožňuje však velmi mocné věci, jako např. změnu programu za běhu systému.
2. **Typová kontrola v době výpočtu**. Kromě typové kontroly při překladu (pouze u jazyků se silnou a slabou typovou kontrolou) je možné ještě provádět typovou kontrolu při výpočtu. Zpravidla jazyky bez typové kontroly provádí tuto kontrolu a jazyky s typovou kontrolou při běhu již typy nekontrolují.
3. **Včasná a pozdní vazba (early and late binding)**. Určení, která metoda bude vyvolána zprávou je možné učinit buď v době překladu nebo až při provádění. První způsob je výpočetně efektivnější, neumožňuje však polymorfismus. Jazyk C++ umožňuje určit typ vazby (klíčové slovo *virtual*), Smalltalk používá vždy pozdní vazbu<sup>2</sup>.

---

<sup>2</sup> Java implicitně používá pozdní vazbu, ale u tříd určených klíčovým slovem *final*, používá včasnou, neboť takové třídy již nesmějí mít potomky a polymorfismus tedy není třeba.

Nejčastěji používaná implementace jazyka Smalltalk-80, prostředí VisualWorks, je z tohoto pohledu

- **interpretované** : programujeme vlastně systém za běhu
- **s typovou kontrolou za běhu**: připomeňme si, že typ objektu je v plně objektovém prostředí určen zprávami, na které objekt umí reagovat
- **s plnou pozdní vazbou**

V dnešní době u většiny úloh zdržení způsobené interpretovaným prováděním, typovou kontrolou za běhu a pozdní vazbou nevádí a je bohatě vyváženo jejich výhodami.

## 1.5 Objektově orientované programovací jazyky

Pojem objektově orientovaného programování (OOP) je u nás spojován především s jeho využitím v tzv. **hybridních objektově orientovaných jazycích** (diskutováno dále). Typickými představiteli této skupiny programovacích jazyků jsou jazyky Object Pascal a C++. Tyto jazyky byly navrženy jako rozšíření klasických programovacích jazyků Pascal a C. Jejich kompilátory jsou proto schopny přeložit "neobjektově orientované" programy, tj. klasické programy v jazyce Pascal či C. Právě hybridním programovacím jazykům a především jejich přímé návaznosti na klasické programovací jazyky dnes vděčíme za stále rostoucí široký zájem o využívání objektově orientovaných systémů.

**Vznik objektově orientovaného přístupu je však spojen s takzvanými "ryze objektově orientovanými" programovacími jazyky.** Tyto programovací jazyky jsou nazývány jako jazyky založené na čistých objektově orientovaných prostředích (*EPOL - environment-based pure object languages*). Nejznámějšími jazyky této kategorie jsou **Smalltalk**, CLOS a Eiffel. Mezi jejich společné vlastnosti patří skutečnost, že se nejedná pouze o kompilátory příslušných jazyků, které by pracovaly pod nějakým klasickým operačním systémem.

**EPOL obsahují kromě implementace jazyků také vlastní kompletní vývojová prostředí pro tvorbu programů a vlastní, vesměs grafické, nadstavby operačních systémů pro podporu běhu programů.**

Tyto grafické nadstavby mohou spolupracovat s klasickými grafickými operačními systémy či nadstavbami, jako jsou například MS Windows, různá Linuxová/Unixová grafická uživatelská rozhraní nebo Mac OS. **EPOL je možné považovat za objektově orientované operační systémy s možností nejen spouštět, ale i vytvářet programy.**

Druhou zvláštností jsou jejich vlastní jazyky. EPOL byly a jsou od začátku navrhovány jako výhradně objektově orientované. Kromě objektů žádné jiné datové typy neobsahují. Jejich čistá syntaxe i model výpočtu jsou proto zpočátku pro klasicky založeného programátora zvláštní. Z určitého pohledu jsou při osvojování si nějakého EPOL jazyka zvýhodnění úplní začátečníci oproti ostříleným programátorům například v jazycích FORTRAN či COBOL.

V EPOL jazycích chybí některé procedurální konstrukce, jako jsou například podprogramy, příkazy skoku a podobně. Naopak však tyto jazyky dovolují elegantně využít všech výhod objektově orientovaného přístupu ve srovnání s hybridními jazyky, v nichž lze (nebo je programátor nucen) objektově orientovanou technologii různými způsoby šidit.

**Uvedené odlišnosti EPOL jazyků od hybridních objektově orientovaných (object-oriented) jazyků vedou některé autory k definici EPOL jazyků jako jazyků přímo "objektově založených" (object-based).**

Mezi čisté dynamické obj. orientované systémy patří jazyky vhodné především pro rozsáhlé úlohy z oblasti umělé inteligence, simulace, počítačové grafiky, databází i z jiných oborů. Jsou to jazyky:

**Simula, Smalltalk, CLOS, Flavors, Dragoon, Eiffel, Beta, Mainsail, ESP a Object-Prolog.** Protože byly navrhovány od samého počátku jako obj. orientované, tak je jejich výhodou mj. i čistá syntaxe a úplně obj. orientované prostředí se všemi vlastnostmi.

Další velikou skupinou programovacích jazyků jsou **hybridní jazyky**, které vznikly obohacením klasických jazyků o určité statické i dynamické obj. orientované rysy. I když tyto jazyky umožňují využívat většinu OO rysů, tak je pro ně charakteristická ta vlastnost, že v nich lze OO zásady "šdit" používáním klasických prostředků. (Stejně snadno jako je možné ve Fortranu nebo Basicu obcházet zásady strukturovaného programování). Mezi tyto jazyky patří především **Object-Pascal, Object ADA, Objective C, C++, Java a Visual Basic.**

### 1.5.1 Vhodný programovací jazyk - Smalltalk

Smalltalk je typickým představitelem EPOL jazyků a pro svoji syntaktickou čistotu a elegantní podporu všech důležitých objektových vlastností je již přes 15 let často používán jako implementační nástroj v odborné literatuře a výzkumu. V posledních letech se však také stává rozšířeným nástrojem pro tvorbu aplikačního softwaru, kde se v roce 1995 dokonce stal nejžádanějším programovacím jazykem v USA.

#### 1.5.1.1 historie Smalltalku

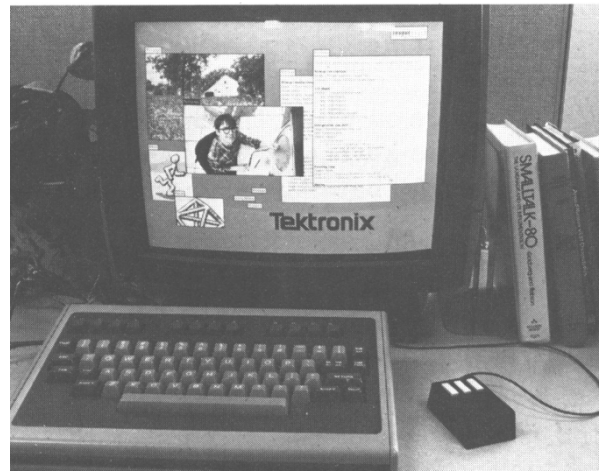
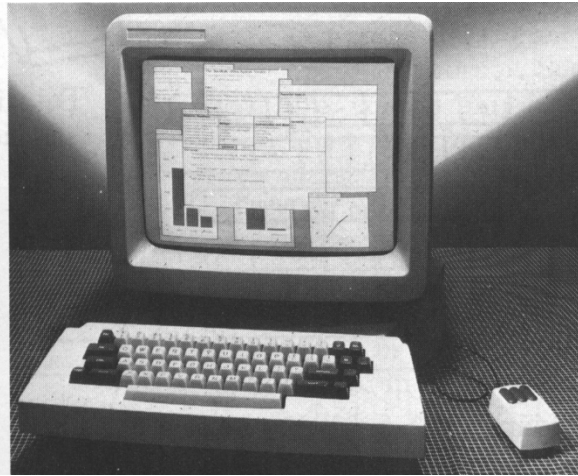
Smalltalk byl vyvíjen v Kalifornii v **Palo Alto Research Center (PARC)** kolektivem vědců vedených dr. Alanem Kayem (tým *Learning Research Group*) a dr. Adelou Goldbergovou (tým *System Concepts Laboratory*) v letech 1970-1980. Předmětem celého výzkumu, který byl financován v největší míře firmou Xerox, 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**, klávesnicí, 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**, vnořovací **menu** a později i **ikony**. V průběhu 70. let bylo dokonce vyrobeno několik prototypů takových počítačů. Předpokládalo se, že počítač bude obsahovat jednotné softwarové prostředí, které bude současně plnit úlohu operačního systému i programovacího jazyka s vývojovým prostředím. Právě tento software dostal název Smalltalk.

Ve Smalltalku, který byl jako projekt dokončen v roce 1980, 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 **Lisa** s grafickým uživ. rozhraním (dále jen 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 používaly 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. Smalltalk svojí filosofií přímo ovlivnil vznik programovacích jazyků Objective-C, Actor, CLOS, Object Pascal, C++, Oberon, Java a C#.

---

\* Jeden exemplář tohoto počítače je v osobním vlastnictví autora.



Xerox Dorado - 1976, Tektronics 4404 - 1982

### 1.5.1.2 Smalltalk dnes

Smalltalk je nenahraditelným pomocníkem ve výzkumu na univerzitních pracovištích, kde je často využíván i jako první vyučovaný programovací jazyk. Je používán především všude tam, kde je třeba v krátkém čase vytvářet náročné aplikace.

Od počátku 90. let se však objevuje nová oblast využití v oblasti tvorby rozsáhlých programů šitých na míru konkrétnímu zákazníkovi (*in-house software*) z ekonomické oblasti, řízení výroby apod. Smalltalk je jedním z mála podporovaných jazyků v moderních objektově orientovaných databázích (Gemstone, Versant, Ontos, Orion, ArtBase,...). Je patrný posun využití Smalltalku směrem od tvorby prototypů ke tvorbě "středních" a "vyšších" programových aplikací, kde se jeví jako vhodnější než například jazyk C++, jehož oblast využití se dnes posunuje směrem k "nižším" aplikacím (systémový software, univerzální software apod.). V roce 1994 si Smalltalk zvolila firma IBM za jeden z podporovaných aplikačních programovacích jazyků (viz. produkt „VisualAge“).

Syntaxe jazyka Smalltalk je jednoduchá, ale jeho sémantika je natolik originální, že 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, polymorfismu a vícenásobné použitelnosti kódu. **Jazyk je integrován s vývojovým 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 zde 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, semafore), z dalších vlastností např. **ošetřování výjimek** v programu, **sledování verzí** programového kódu během programování s možností návratu do libovolného z přechozích 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 posláním nějaké zprávy zvenčí (klávesnice, myš...), která způsobí posílání zpráv dalším objektům aplikace. **Program se za chodu chová jako simulační systém řízený sledem vnějších událostí.**

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 odladit 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.

Smalltalk totiž nevyužívá přímo strojový kód počítače, na kterém běží. Namísto toho překládá programy do tzv. **byte kódu**, který je za běhu 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.

### 1.5.1.3 současné systémy Smalltalk

1. **Smalltalk-80** firmy Cincom (dříve Parc Place Systems a Digtalk). 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, 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, moduly obchodní grafiky, ...), tak i v napojení na relační a objektové databázové systémy. Existují i univerzitní projekty s volně dostupnými knihovnami na Internetu.
2. **Visual Smalltalk** Je také produktem firmy Cincom. Pracuje pouze pod operačními systémy MS Windows, OS/2 (existují i jeho starší verze označené Smalltalk/V pro samotný MSDOS) 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. Na internetu je pro akademické instituce bezplatně dostupná verze označená **Smalltalk Express**.
3. **IBM Smalltalk** je 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 alias VisualAge 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 svou originálním způsobem vytvořenou vazbou na jazyk C s možnostmi překladač do strojového kódu. Je dostupný na jakémkoliv počítači s operačním systémem UNIX.

5. **Smalltalk DB** je jazykem objektivě orientovaného databázového systému Gemstone.
6. **Enfin** je produktem společnosti Easel Corporation z Burlingtonu v Massachusetts. Pracuje pod operačními systémy Windows, 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. Podobně jako VisualWorks umožňuje objektivě orientovaným aplikacím pracovat s objekty, jejichž data jsou uložena v relační databázi. Tvůrce aplikace pak může svou pozornost věnovat objektům a nemusí se starat o problémy vzniklé použitím relační databáze v objektovém prostředí.
7. **GNU Smalltalk** je produkt v rámci unixového projektu GNU. Jeho praktickou použitelnost však v současné době omezuje neexistence integrovaného grafického vývojového prostředí.
8. **Little Smalltalk** je také produkt GNU. Jedná se o nejmenší smalltalkový systém (velikost image 50kB), který je vhodný např. pro programování dávek operačního systému a jiné úsporné programy.
9. **Squeak** vzniká v Apple Computers pod vedením Allana Kaye především jako platforma pro revoluční přístupy k výuce dětí a testování nových přístupů práce s počítačem, hlavně s moderními multimédii. Squeak disponuje 3D grafikou, zvukem, MIDI, syntézou řeči, atd. V konceptech vizuálního programování je ze všech implementací Smalltalku nejdále. Příjemné také je, že je k dispozici zdarma ke stažení na internetu.
10. **Smalltalk MT** je projekt firmy Object Connect pro podporu programování ve Windows NT a 95, zatím ve stadiu betaverze dostupné na internetu.
11. **Dolphin Smalltalk** je internetový projekt firmy Dolphin se stejným zaměřením jako MT.
12. **Pocket Smalltalk** je odlehčená varianta Dolphin Smalltalku pro kapesní počítače s operačním systémem Palm OS.

## 1.6 Počítačové databáze

### 1.6.1 Současný stav

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). Největší časovou spotřebu mají v relačním modelu dotazy využívající operaci spojení tabulek s vazbami typu 1:N a M:N. Pokud se podaří stejné zadání transformovat podle zásad objektové analýzy a designu na hierarchickou strukturu vzájemně se skládajících objektů, tak lze na srovnatelné objektové databázi registrovat 2 až 3 řádové časové úspory při zpracovávání dotazů.

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. triggerů). 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, primárních a sekundárních klíčů, operacemi relační algebry a relačního kalkulu, potom je databázový systém z pohledu své funkčnosti nutně relační a neobjektový. 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 SOUČASNÝCH DB APLIKACÍ Z POHLEDU VYUŽÍVÁNÍ OBJEKTIVÝCH TECHNOLOGIÍ

strana klienta	strana serveru
<b>vlastní DB generátory aplikací</b> (ORACLE forms, reports, ...)	<b>relační DB server</b> (ORACLE)
<b>SQL v klasickém progr. jazyce</b> (Cobol, Fortran, C, Pascal)	<b>relační DB server</b> (ORACLE)
<b>Spec. generátory aplikací</b> (PowerBuilder, Gupta, Delphi)	<b>relační DB server</b> (ORACLE)
<b>klient z obj.orient.progr. jazyka</b> (Smalltalk, Objective-C, ev. C++)	<b>relační DB server</b> (ORACLE)
<b>klient z obj.orient.progr. jazyka</b> (Smalltalk)	<b>objektový DB server</b> (Gemstone)

*míra podpory objektového modelu výpočtu*

### 1.6.2 Vztah relační a objektové technologie

Technologie RDBMS byla velmi úspěšná, protože podstatně zjednodušila a zkvalitnila práci v oblasti podpory jednoduchých předdefinovaných 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.

Na současných relačních databázích je patrný trend směrem k postupnému **rozšiřování relačního datového modelu**. První z nich jsou tzv. **BLOBs** (binary-large objects), v různých systémech označované i např. jako „RAW“ nebo i „MEMO“. Druhou z nich je možnost přidávání procedur k záznamům v relačních tabulkách - tzv. **triggers**. Toto vede mnohé prodejce relačních databází k reklamním tvrzením, že jejich produkty jsou objektově orientované, přičemž tato tvrzení jsou dovedně dokládána možnostmi ukládání např. obrázků a využíváním výše zmíněných triggerů v příkladech předváděných na výstavách, konferencích či v literatuře.

Realitou současných objektových databází jsou možnosti více různých rozhraní k objektově orientovaným programovacím jazykům, a to nejčastěji Smalltalku, C++ a nebo Lispu. Otázka jazyka SQL není v objektových databázích tak jednoduchá, jak by se mohlo zdát, protože vlastnostem (možnosti dotazování, funkčnost aplikace, ...) objektového datového modelu mnohem lépe vyhovují dnes již standardizované obj. orientované programovací jazyky (např. Smalltalk), které navíc umožňují napsat na rozdíl od SQL celou aplikaci včetně např. uživatelského rozhraní a tím vyřešit **impedanční problémy**. Jazyk SQL je tedy bez nadstandardních rozšíření (např. SQL3, O<sub>2</sub>SQL, Object SQL) pro objektové databáze v podstatě nepoužitelný.

Moderní databáze nabízejí dvě možnosti **ovládání transakcí**. První z nich - tzv. pesimistické řízení - je shodná s klasickým mechanismem známým z relačních databází. Protože však možnosti uzamykání a přístupu na data nejsou v objektových databázích omezeny na celé záznamy (objekty), ale uzamykání

Lze provádět mnohem jemnějším způsobem (po částech objektů, na volání zpráv, ...), je v objektových databázích k dispozici volitelně i mechanismus tzv. **optimistického řízení**.

Od roku 1993 probíhá **standardizační proces v objektové technologii** a to jak na aplikační úrovni (jazyky), tak i na fyzické úrovni (komunikační protokoly, formáty v paměti). Jsou již první výsledky ve formě zpráv ODMG-93 a specifikací CORBA 1.0 (Common Object Request Broker Architecture) a CORBA 2.0. V rámci norem ANSI existují a již první výsledky podávají výbory ANS X3J20 pro Smalltalk, ANS X3J16 pro C++ a pro objektová rozšíření jazyka SQL výbor ANS X3H2.

Mezinárodní sdružení firem zabývajících se objektovou technologií OMG (Object Management Group) vydává dokument „Request for Technology for Object Query Services“ obsahující specifikaci jazyka OQL. Každým rokem ve světě probíhá několik konferencí, kde je diskutována jak praxe, tak i příspěvky k matematické teorii objektivě orientovaného datového modelu.

Objektové a relační databáze se však 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 prezentují 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 nebo aplikačních programů 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.

### 1.6.3 Současné trendy v databázové technologii

V databázové technologii v souvislosti s objektovým datovým modelem můžeme v současnosti vysledovat dva trendy. Jedná se o trend evoluční a revoluční.

1. **evoluční trend** spočívá v postupných změnách a vylepšeníh klasického relačního datového modelu směrem k objektům. Jedná se o využívání triggerů, možnost tvorby nových datových typů, BLOBů, objektivě orientovaných aplikačních programovacích jazyků (např. známé databáze Delphi, Gupta, aj. pro MS Windows), které postupně vede k rozšiřování datového modelu po teoretické stránce (např. odklon od požadavku na první normální formu v databázi ADABAS). Zajímavé řešení podporuje firma ORACLE od verze Oracle-8, který podporuje několik objektových rysů, ale interně používá relační databázový stroj. Tento přístup volí firmy, které mají silnou pozici v oblasti relačních technologií, které jsou rozšířené a pro stále ještě dost případů (skladové evidence, personalistika, účetnictví, ...) vesměs vyhovující. Přejechod k objektům je u nich brzděn požadavky zpětné kompatibility a uspokojivými zisky z relačních systémů a na druhé straně vynucen konkurenčním bojem a postupným ztrácením pozic v oblasti speciálních databázových aplikací (GIS, CAD/CAM, Expertní systémy, Systémy pro podporu rozhodování, ...), odkud je postupně vytlačují objektivě orientované systémy.
2. **revoluční trend** je plně objektivě orientovaná databáze. Tento přístup představuje hlavní hnací sílu rozvoje této nové technologie a volí ho takové firmy, které nejsou svázány žádným výše naznačeným způsobem s relačními databázemi. Dnes jsou na světovém trhu již desítky takovýchto objektivě orientovaných databází nejrůznější kvality. Je však již nejméně 5 skutečně kvalitních systémů plně srovnatelných např. se systémy typu Oracle, se kterými se lze setkat především v prostředí Unixu a pracovních stanic. Kromě experimentálních systémů, které mají svůj původ většinou na nějaké renomované univerzitě, lze komerčně dostupné systémy rozdělit do dvou kategorií.
  - a. databáze založené na systému Smalltalk. Tyto systémy jsou vlastně nejrůznější smalltalková prostředí doplněná o potřebné databázové vlastnosti (persistence objektů, transakce, dotazování, ...).
  - **GemStone** firmy Gemstone Systems, jehož databázový stroj je vybudován na základě speciálně vytvořené verze Smalltalku. Databáze obsahuje rozhraní na většinu verzí jazyka Smalltalk a také na jazyky C, Pascal a SQL.

- **Versant** firmy Versant Object Corporation. Jádro systému je implementováno v jazyce C (pozor - ne C++). Systém podporuje podobně jako GemStone různé verze jazyků Smalltalk.
- **ArtBase** firmy ArtInApples. Tato databáze je zajímavá hned z dvou pohledů. Za prvé i když se jedná o výkonnou víceuživatelskou unixovou (ale i PC a Novell) databázi, tak je realizována jako „čistý“ program ve Smalltalku-80 bez využití jakýchkoliv modulů psaných v strojově orientovaných jazycích. ArtBase je tedy možné snadno kombinovat s každým jiným programem ve Smalltalku-80. Druhou zajímavostí je fakt, že firma ArtInApples sídlí v Bratislavě, což znamená, že bývalé Československo bylo první a zatím jedinou zemí východního bloku, kde byly podmínky pro vznik a rozvoj domácích progresivních softwarových technologií. ArtInApples, která úzce spolupracuje s firmou Parc Place Systems je dodnes jedinou „východní“ firmou, která je členem prestižního sdružení OMG - Object Management Group. (ArtInApples také pořádala mezinárodní konference EastEurOOP'91 a '93). Artbase je součástí finského metamodelovacího CASE nástroje Metaedit firmy Metacase Ltd.

b. ostatní objektové databáze

- **O<sub>2</sub>** firmy O<sub>2</sub> Technology má jádro vybudované v jazycích Lisp a C. Podporuje jazyk C++ a vlastní jazyky O<sub>2</sub>SQL a O<sub>2</sub>C.
- **Iris** firmy Hewlett-Packard je implementován v jazyce C. Podporuje jazyky Lisp, C a OSQL.
- **Orion** firmy Microelectronics and Computer Corporation je založen na systému Common Lisp. Podporuje jazyk Orion SQL.
- **Vbase** alias **Ontos** firmy Ontologic je implementován v jazyce C. Podporuje jazyky COP (C Object Processor) a TDL (Type Definition Language).
- **ObjectStore** firmy ObjectStore je implementován v jazyce C. Podporuje jazyky SQL, C++ a Smalltalk.

Pro optimální využití vlastností objektových databází není možné databázovou strukturu navrhovat stejným způsobem jako relační či dokonce přímo převzít relační schéma. Velmi důležitou roli zde mají **techniky objektově orientované analýzy a designu**.

## 1.7 Programovací styly

Vzhledem k požadavkům praxe na software je patrná snaha učinit z programování deterministickou a automatizovatelnou činnost. Kvalitní software je však nesnadnou záležitostí. V žádném případě dnes neexistuje a asi nikdy existovat nebude nějaký jednoznačný postup, pomocí kterého by šlo bezchybně, deterministicky a snadno vytvářet software. *(Možná to vyplývá i z filosofického pohledu na tvorbu programů: Tvorba a používání programů je modelování nebo doplňování reálného světa okolo nás. S poznatelností procesu vlastního programování to tedy musí být podobné jako s poznatelností světa vůbec)*

Zkreslený laický pohled poslední doby se omezuje pouze na návrh grafických uživatelských rozhraní, což je nesprávné a nebezpečně ohrožuje samotný proces tvorby softwaru.

Vysokoškolsky vzdělaný uživatel počítače by neměl otázku technologií tvorby softwaru podceňovat, protože tyto znalosti mu umožňují znásobovat produktivitu práce i mimo oblast samotného programování. Nejde jen o to, že takový uživatel by nutně musel být také tvůrcem softwaru. Dostatečná softwarová gramotnost totiž především velmi usnadňuje komunikaci mezi uživatelem, zadavatelem a tvůrcem programového díla.

### 1.7.1 Naivní technologie

*"Sestav jakýmkoliv způsobem za pomoci programovacího systému program, který řeší daný problém."*

Naivní technologii označujeme styl bez použití jakékoliv jiné technologie. Zpravidla se jedná o takový postup, který se opírá pouze o použitý systém a znalosti programátora o něm.

Největším nedostatkem naivní technologie je naprostá vyjadřovací svoboda, která je závislá jen na míře zvládnutí systému. Taková "volnost" ale neposkytuje žádný návod pro pochopení a ověření činnosti aplikace. Forma popisu, protokol i vzhled a funkce jsou zcela záležitostí řešitele. Nekontrolované používání libovolných konstrukcí značně znehodnocuje především adaptabilitu, odladitelnost a další podstatné vlastnosti. Na neštěstí řada uživatelů ovládajících softwarové balíky a řídicí jazyky v aplikačních programech dostává pocit, že dosáhli v oblasti výpočetní techniky vrcholu a další věci jsou podle nich již jen univerzitní výmysly, kterými se nemá cenu pro praxi zabývat. Mnozí se též po povrchním seznámení s počítači výše uvedeným způsobem domnívají, že tvorba softwaru je velmi snadná mechanická záležitost a povýšeně nahlíží na jeho problematiku.

### 1.7.2 Klasické - imperativní technologie

Podstata strukturované, ale i dalších technologii spočívá mj. ve výběru relativně malé, ale postačující sady instrukcí a dalších prostředků pro tvorbu programů. Každá taková technologie potom dovoluje přehledný zápis. Jednotlivé efekty jsou snadno formálně popsitelné.

Ve všech třech případech se jedná o čistě imperativní programovací technologie vycházející z vN modelu činnosti počítače. Tyto technologie vývojem vyústily v 80. letech do objektově orientované technologie.

### 1.7.3 Strukturovaná technologie

*"Navrhni strukturu vstupních a výstupních dat a struktury, v nichž bude informace udržována během zpracování. Dále sestav výhradně z vyhrazených konstrukcí postup, který vede k řešení problému."*

Strukturovaná technologie je nejstarší technologií programování. Její první výsledky v oblasti tvorby programů vyústily ke vzniku prvního vyššího programovacího jazyka Fortran. Do povědomí se ale dostala poprvé v 60. letech v souvislosti s jazykem Algol. V 80. letech, kdy došlo k prudkému rozvoji osobních počítačů, se strukturovaná technologie programování masově rozšířila především s rozšířením programovacího jazyka Pascal. Mezi přípustné základní konstrukce patří **zřetězení** (*sequence*), **cyklus** (*iteration*) a **větvení** (*selection*).

Strukturovaná technologie doznala značných rozšíření a vyústila v **procedurální a modulární technologie**. V současné době se někdy nepřesně pod pojmem strukturované technologie myslí zároveň technologie procedurální a modulární.

### 1.7.4 Procedurální technologie

*"Rozlož řešený problém na několik podproblémů. Navrhni řešení podproblémů jako samostatné procedury. Při vyjádření vlastního algoritmu řešení problému využij navržených procedur"*

Na rozdíl od strukturovaného přístupu, který klade důraz na datové struktury, procedurální přístup je zaměřen především na **algoritmické vyjádření**. U jednotlivých procedur se předpokládá minimum tzv. "vedlejších efektů". Odtud plyne i dobře známá zásada jednoho vstupu a jednoho výstupu.

### 1.7.5 Modulární technologie

*"Rozlož řešený problém na několik podproblémů. Při rozkladu odděl řízení zpracování od údržby informace v datových strukturách, pro kterou navrhni potřebné moduly. Nakonec s využitím akcí modulů vyjádři vlastní algoritmus řešení."*

Jedná se o další rozpracování strukturované a procedurální technologie. Tato technologie přináší další zlepšení především v tom, že na hlavní úrovni řešení problému není třeba uvádět deklarace datových struktur. Datové struktury jsou skryty v modulech a jsou dostupné přes akce modulů. Modul vždy tvoří **samostatnou jednotku**, kterou lze samostatně a opakovaně používat. Nedaří-li se vyčlenit procedury pracující pod společnými daty nebo není-li problém příliš složitý, postačí použít procedurální technologie. Procedurální technologii lze též použít při realizaci jednotlivých modulů.

Z hlediska vytváření větších programových celků přináší modulární technologie pokrok v tom, že při dekompozici problému na menší části řešitel ponechává návrh a obhospodařování dílčích datových struktur na řešitelích modulů. Tím je značně usnadněna komunikace mezi řešiteli, protože při klasické strukturované nebo procedurální technologii je nutno zahnovat i dohodu o všech používaných datových strukturách.

### 1.7.6 Neimperativní technologie

Logická i funkcionální technologie jsou charakteristické tím, že nejsou postaveny na imperativním modelu výpočtu. V souvislosti s těmito technologiemi se hovoří o programování v oblasti umělé inteligence a systémů pro podporu rozhodování. Od poloviny 80. let jsou obě technologie úspěšně doplňovány o objektivě orientovaný přístup. I když jsou založeny na **odlišných modelech výpočtu**, tak jsou známy prostředky pro jejich vzájemnou transformaci.

### 1.7.7 Funkcionální programování

*"Rozlož řešený problém na podproblémy. Takto postupuj (případně rekurzivně) tak dlouho, pokud k řešení problému není dostupná známá (případně primitivní) funkce. Z takto získaných řešení postupně zkombinuj funkci, která řeší daný problém"*

Funkcionálnímu modelu nejlépe vyhovuje výpočetní stroj založený na data-flow modelu výpočtu. Řešení se neskládá z detailního popisu, jakým způsobem dojít k cíli. Ve funkcionálním návrhu je řešením množina funkcí, které jsou potřebné k vyřešení daného problému. Systém sám během chodu využívá potřebné funkce a řídí jejich provádění. Nejznámějším funkcionálním programovacím jazykem je **Lisp**. Funkcionální filosofie Lispu byla využita i v návrhu objektivě orientovaného jazyka Smalltalk. Teoretickými aspekty možného sjednocení funkcionálního a objektivě orientovaného programování se zabývá výzkum.

### 1.7.8 Logické programování

*"Rozlož řešený problém na podproblémy. Takto postupuj (případně rekurzivně) tak dlouho, pokud k řešení problému není dostupná známá (případně primitivní) relace. Z takto získaných řešení postupně zkombinuj relaci, která reprezentuje daný problém"*

Logické metodologii programování se v anglosaské literatuře říká **COP** (*Constraint Oriented Programming*). Logickému modelu odpovídá inferenční stroj, jehož činnost je založena na **predikátové logice**. V logickém návrhu je řešením množina logických faktů a pravidel, které jsou potřebné k vyřešení daného problému. Systém během chodu programu využívá potřebná fakta i pravidla a řídí jejich provádění. Typickým nástrojem je programovací jazyk Prolog.

## 1.7.9 Objektová technologie

*"Najdi (popř. navrhni) potřebné objekty a realizuj jejich strukturu a operace podle zadání. Řešení problému sestav pomocí vzájemných vazeb a komunikace objektů v systému."*

Při použití objektově orientované technologie rozdělujeme problém na jednotlivé objekty jako celky, ve kterých jsou pohromadě jak datové hodnoty, tak i jejich chování. Zkušený objektový návrhář si vytváří knihovny objektů, o nichž předpokládá, že je v budoucnu použije. Každý problém se snaží řešit obecně. Poměrně rychle tak lze získat množinu objektů, které popisují "problémový prostor" určité třídy úloh. Nové objekty je žádoucí vytvářet z objektů již existujících s využitím vazeb skládání, dědičnosti, závislosti a delegování a nově implementovat jen ty charakteristiky, jimiž se liší od již stávajících objektů.

Objektově orientovaná technologie překonává zatím nejlépe ze všech jiných technologií nebezpečí, že **tvůrce vyčerpá svoji energii na jednotlivostech** vynucených obtížnou implementací detailů projektu na konkrétním operačním systému a programovacím jazyce.

U dobře navrhnuté objektově orientované aplikace lze skrýt velké množství primitivních vN operací, které jsou při tvorbě rozsáhlejší aplikace na obtíž (skoky, cykly, jednoduché algoritmy pro změny hodnot, řazení, třídění a operace mající přímou vazbu na hardware počítače nebo jeho periférie). Jsou totiž zapouzdřeny do kódů metod objektů, a takové objekty potom spolu komunikují na vyšší úrovni. Mnoho objektů lze využívat jako **hotové díly** v knihovně systému. Celý systém může být navrhován, testován a laděn po částech - jednotlivých objektech. Podobně jako v neimperativních technologiích není třeba v systému explicitně popisovat celý algoritmus výpočtu od začátku do konce, neboť řízení provádí za chodu systém pomocí informací obsažených v jednotlivých objektech (data, metody, vazby).

## 1.7.10 Komponentové programování

*"Rozlož řešený problém na objekty. Podívej se, jestli nějakou část systému neřeší již vytvořené komponenty. Opatři si (kup) vhodné komponenty. Uprav chování komponent k obrazu svému. Objektově doprogramuj zbylé části systému. Slož komponenty a doprogramované část dohromady v systém. Zamysli se, jestli jsi nevytvořil nové komponenty. Nové komponenty dej do repozitáře komponent."*

Komponentové programování má vztah k objektovému podobný, jako modulární k strukturovanému. Hlavním principem je rozdělit systém podle problémů a pokusit se získat (typicky koupit, nebo v horším případě doprogramovat) komponenty, řešící části těchto problémů. Komponenty mohou být i rozsáhlé – například, když budu programovat informační systém, kde potřebuji řešit vkládání a úpravy textu, tak mohu použít komponentu řešící textový editor. Komponenta typicky obsahuje vevnitř mnoho objektů k nimž je přístup pouze přes rozhraní komponenty – komponenta je zapouzdřena. V současné době je nejrozšířenějším standardem zajišťující komunikaci mezi komponentami DCOM (Distributed Common Object Model) firmy Microsoft, na kterém je postaven operační systém Windows NT.

## 1.8 Základní principy objektově orientovaného přístupu.

V dnešní době se zdá, že popis základních principů OOP je zbytečným nošením dříví do lesa. Dnes již stěží najdeme informatika, který by nikdy neslyšel slovo objekt a nikdy neslyšel například o jazycích Java nebo C++. Došlo však k tomu, že slovo „objekt“ a „objektově orientovaný“ se stalo natolik módním slovem - v angličtině existuje trefné označení „buzzword“, že se používá velmi často pro prosté označení čehosi progresivního, co má něco společného s počítači. Jedna věc je však použití slova a druhá věc je pochopení, co doopravdy znamená. (Možná se brzy dočkáme objektově orientovaného notebooku nebo alespoň klávesnice).

Právě proto je třeba se pokusit o velmi stručnou rekapitulaci základních principů OOP a to záměrně z poněkud jiného pohledu, než je dnes vesměs prezentován.

## 1.8.1 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, než jedné proměnné příslušného datového typu, i když z programátorského pohledu takovou proměnnou je. Objektový systém je potom souborem takovýchto vzájemně interagujících 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 kódu, 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.

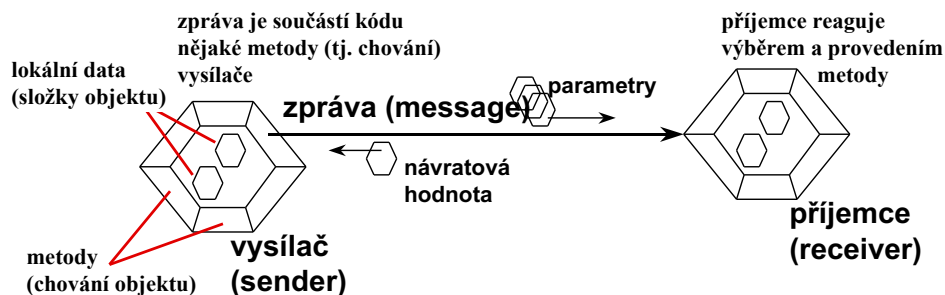
V objektovém modelu výpočtu pracujeme pouze se dvěma možnými operaci s objekty. První z nich je **pojmenování** nějakého objektu. Druhou je tzv. **poslání zprávy**. Zpráva představuje **žádost o provedení** operace - metody nějakého objektu. Součástí zprávy mohou být tzv. **parametry zprávy**, což jsou 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 - vysílači 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 řízen 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 (dále bude vysvětleno). 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.

# ZAPOUZDŘENÍ, METODY, POLYMORFISMUS



**typ zpráv**  
**"WAIT" nebo "FORK"**

**PROTOKOL**

= seznam, jaké zprávy je možné objektu poslat  
 = jediná prezentace objektu navenek

**včasná nebo pozdní vazba**

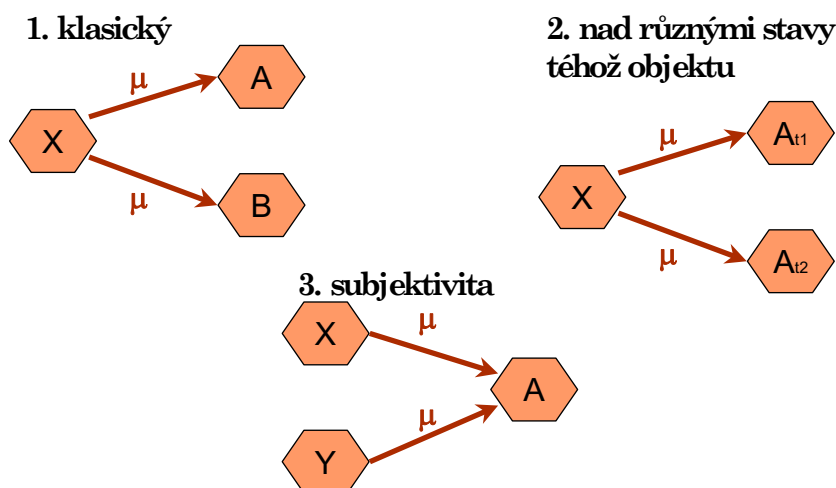
Je úplnou samozřejmostí, že objektově-orientované principy neslouží jen k lepšímu návrhu grafického uživatelského rozhraní. Dobrý program se rozezná právě podle toho, že objekty používá i „uvnitř výpočtu“ a ne jenom pro ovládání tlačítek, oken a menu.

## 1.8.2 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.

Pro příklad si uveďme zprávu „dej svoji velikost“. Pošleme-li ji objektu, který reprezentuje nějakou osobu, tak sdělí její výšku. Pošleme-li ji objektu představujícímu množinu nějakých číselných hodnot, sdělí jejich počet, pošleme-li ji objektu představujícímu nějaký obrázek, sdělí jeho rozměr, atd. Je tomu tak proto, že všechny uvedené objekty jsou polymorfní a zpráva „dej svoji velikost“ může být poslána kterémukoliv z nich. Důležitá je tu ta skutečnost, že o výběru odpovídající metody na poslanou zprávu rozhoduje přijímající objekt, což znamená, že vysílající objekt (nebo část programu) se o detaily zpracování nemusí starat.

## 3 TYPY POLYMORFISMU OBJEKTŮ



*Polymorfismus je hlavní příčinou výhod OOP*

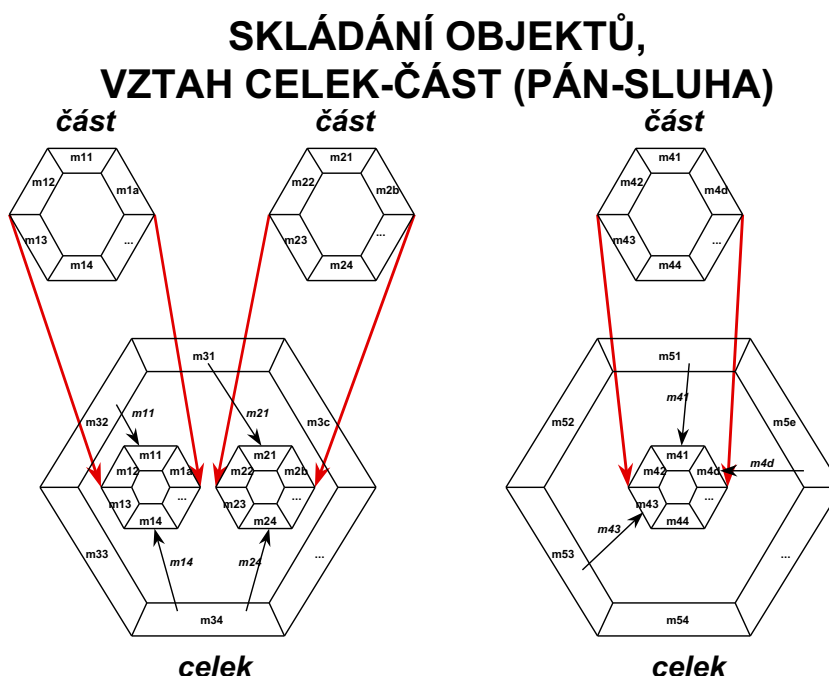
Polymorfismus tedy v objektovém programování znamená, že ta samá zpráva může být poslána rozličným objektům bez toho, že by nás při jejím poslání zajímala implementace odpovídajících metod a datových struktur objektu (příjemce zprávy); každý objekt může reagovat na poslanou zprávu po svém svojí metodou. Programátor není nucen brát ohled u významově podobných operací na to, že jsou požadovány od různých objektů.

### 1.8.3 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 nezískáme 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 takového skládání objektů potom můžeme 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 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.

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).



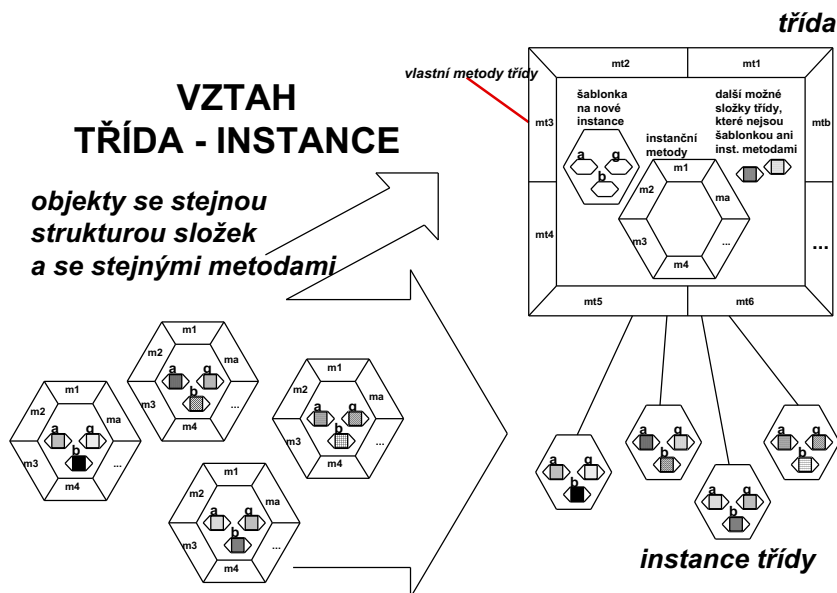
### 1.8.4 Třídy objektů

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 spoji 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.



### 1.8.5 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 **liši**, 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).

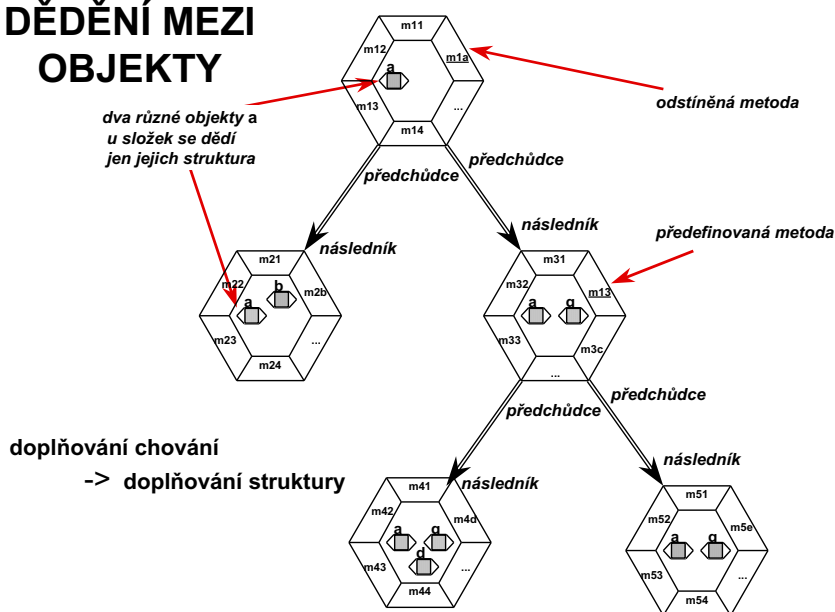
### 1.8.6 Dědění versus skládání

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. systém jazyků 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 závisující především na zvládnutí technik objektově orientované analýzy a návrhu, což přesahuje možnosti tohoto článku. Proto se spokojíme alespoň s výčtem několika faktů, které obecně pro dědění platí.

- **uživatel dědění nezajímá.** Pokud budete vysvětlovat základy OOP začátečníkům nebo uživatelům objektových programů, 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 nespojuje 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. Uvedme si klasický manuálový příklad: Udělejme objekty třídy Bod (s hodnotami souřadnic x,y). Potom můžeme realizovat např. Obdélník pomocí dědění z Bodu, kde doplníme ještě souřadnice toho druhého bodu úhlopříčky (např. x2,y2). Tak a je to. Dědění přineslo úsporu kódu - ať žije OOP. Nebylo by však lepší implementovat obdélník jako objekt složený z dvou bodů? Zkusme se nad tím zamyslet hlavně pro případ, že se budou měnit vlastnosti (metody, či systém souřadnic, ...) bodů. Výběr skládání-dědičnost může být obecně složitým rozhodováním a v dobré literatuře je tomuto problému věnována velká pozornost. Nejjednodušším prvním vodítkem při rozhodování je např. položit si otázku, jestli objekt JE nebo OBSAHUJE druhý objekt. (Např. obdélník *obsahuje* body, ale obdélník *je* grafický objekt).

## DĚDĚNÍ MEZI OBJEKTY

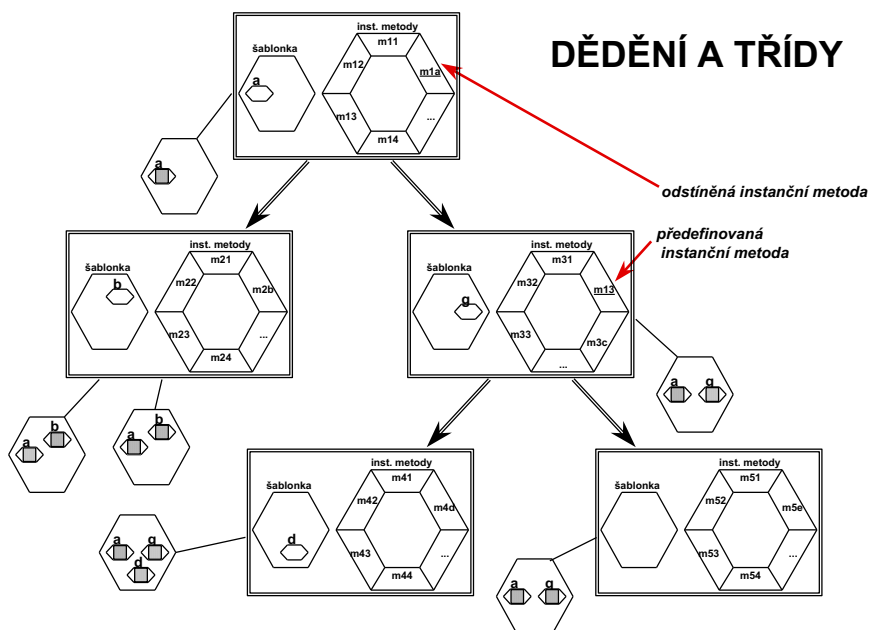


### 1.8.7 Dědění mezi třídami

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-istančního 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.



### 1.8.8 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.

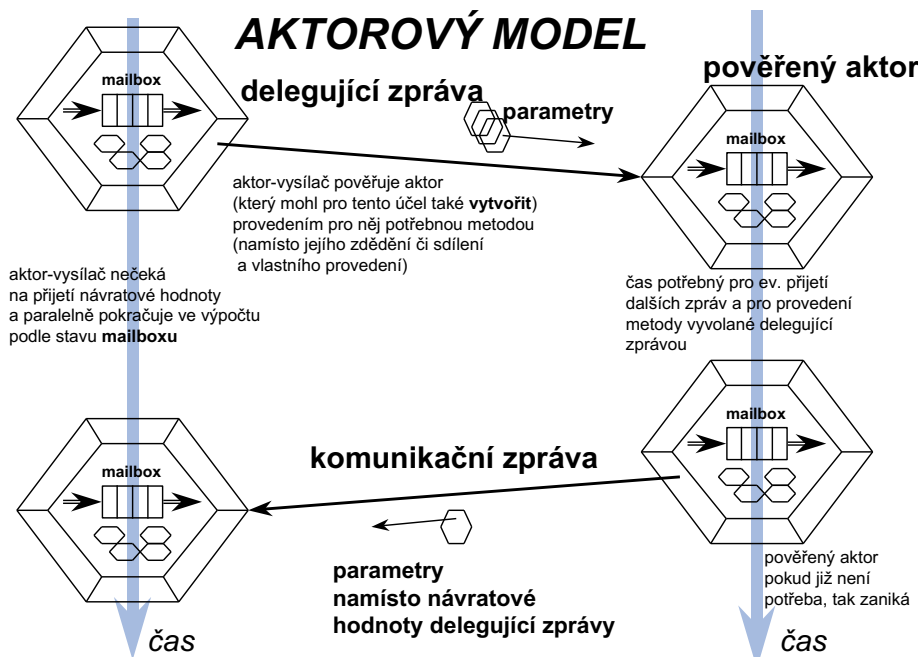
Tzv. **aktorový model** jako alternativní model objektového výpočtu je založen na myšlence, že objekty (používaný název v tomto kontextu je "aktory") v systému nemusejí implementovat všechny pro ně požadované funkce, přičemž ty funkce, které přesahují okruh identity jednoho objektu (a které by se jinak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

V praxi používané aktorové objektové systémy (jakými je například Act, Plasma nebo Omega) tedy hierarchii dědičnosti a vazby třída-instance mezi objekty **nepodporují** a využívají výhradně delegování. Pozor: programovací jazyk Actor, který je znám především z verze pro MS Windows je však založen na dědičnosti a třídně-istanční hierarchii.)

Operace **delegování** je formálně totožná s operací poslání zprávy. Pro zpracování 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 zpětného 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 usnadnění tohoto přenosu výsledku zpět do původního aktoru, který tuto část výpočtu delegoval, původní aktor vystupuje v nových aktorech, které sám při delegování použil, jako jedna z jejich složek.



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, kde se použije kód třídy nebo dědění. 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).

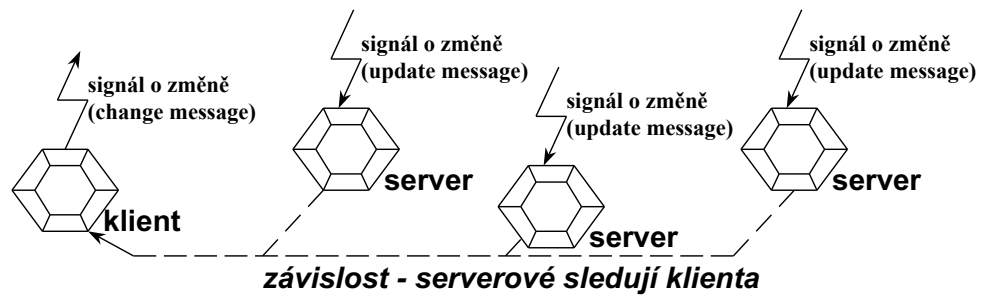
Aktorové modely výpočtu jsou zatím pouze využívány v experimentálních systémech pro modelování znalostí. I u třídně-instančních systémů s děděním je však výhodné, pokud to dovoluje implementační prostředí (C++ a Object Pascal nedovolují, Smalltalk dovoluje) a podporuje zadání problému, **využití** vlastností delegování a asynchronního paralelismu v analýze, návrhu i implementaci modelovaného systému.

### 1.8.9 Závislost mezi objekty

Ve vztazích mezi objekty rozeznáváme dva typy objektů: řídicí 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, v případě závislých objektů klient nepotřebuje znát svoje servery, protože oni sami jsou povinni svého klienta **sledovat** a zprávy od něj **zachycovat**. Zprávu, která je signálem pro servery, objekt klient posílá bez udání příjemce a systém ji automaticky **rozšiřuje** na příslušné servery, jejichž počet a vlastnosti se mohou v systému průběžně měnit.

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

## ZÁVISLOST OBJEKTŮ, VZTAH KLIENT-SERVER



## 2 Úvod do Smalltalku

Programovací jazyk a systém Smalltalk patří v programátorské praxi mezi poměrně nové systémy, i když jeho první verze vznikaly již v 70. letech (viz. historie Smalltalku), ale stále zřetelněji se ukazuje, že ho nepostihne osud jiných jazyků a systémů, které na čas zazářily na softwarovém nebi a poté upadly v zapomenutí. Smalltalk se postupně stal nástrojem především profesionálních programátorů a počítačových výzkumníků. Do blízké budoucnosti se očekává podstatný nárůst jeho vlivu v přímé souvislosti s masovým nasazením OOP, kde je Smalltalk jedním z průkopnických systémů, a s očekávanými výkonnými počítači nových architektur. Kromě profesionálního využití je Smalltalk využíván pro svoji eleganci i amatérskými a aplikačními programátory, kde se prosazuje jako výkonná alternativa k různým "prostředkům pro tvorbu programů bez znalosti programování".

Smalltalk je určen zejména pro výzkumné účely a pro rychlý návrh programů využívajících grafické uživatelské rozhraní ovládané myší s okny ikonami a menu. Je mimořádně vhodný pro tvorbu aplikací z oblasti umělé inteligence a z oblasti objektově orientovaných databázových systémů.

Smalltalk není jen programovací jazyk, neboť kromě překladače vlastního jazyka je jeho nedílnou součástí jeho vlastní grafické prostředí pro podporu tvorby a běhu programů, které tvoří vlastní kompaktní objektově orientovaný operační systém využívající bitovou grafiku, okna, menu a ovládání myší. Systém se v počátku 80. let stal vzorem pro grafické uživatelsky orientované nadstavby ke klasickým operačním systémům (MultiFinder, GEM, Motif, MS Windows, XWindow etc.).

Jazyk Smalltalku je svojí systaxí velmi čistý, jednoduchý a přirozeně objektově orientovaný. Na rozdíl od procedurálních programovacích jazyků (Fortran, Algol, C, Pascal) v něm nenajdeme syntaktické výjimky, na rozdíl od hybridních objektově orientovaných jazyků (Object Pascal, C++) v něm chybí pro OOP nepotřebné procedurální konstrukce, jako například podprogramy, funkce, příkazy skoku apod. Celý systém je přímo založen na třídě instancním objektovém modelu. Na rozdíl od hybridních programovacích systémů se ve Smalltalku předpokládá využívání OOP a OOD nejen pro návrh a realizaci uživatelského interface, ale i pro vlastní model výpočtu ve vytvářeném programu.

Masovému rozšíření odpovídajícímu jeho kvalitám však dosud brání konzervatismus IT manažerů ruku v ruce s dlouholetou navyklostí programátorské veřejnosti na systémy založené na jazyku C (C, C++, Java). Tento stav má kořeny v dobách, kdy osobní počítače nedosahovaly výkonů potřebných k provozování komplexních vývojových prostředí založených na jazyce Smalltalk<sup>3</sup>. Podobnost s jazykem C je například důvodem úspěchu Javy, která i přes své nesporné schopnosti nikdy nemůže dosáhnout možností plně objektového Smalltalku.

### 2.1 První seznámení s Smalltalkem

Popíšeme si instalaci systému a ukážeme si několik jednoduchých příkladů. Systematickému popisu se budeme věnovat v následujících kapitolách. Čtenář také najde příklady na adrese <http://kii.pef.czu.cz/predmety/OMP>

Budeme používat implementaci Smalltalku-80 od firmy Cincom, která se jmenuje Visual Works. Příklady a postupy zde uváděné odpovídají nejnovější verzi v době psaní tohoto textu (7.2), nicméně měly by platit i pro verze budoucí.

#### 2.1.1 Instalace Systému VisualWorks

Systém VisualWorks se jako typický Smalltalkový systém skládá především z *image* a *virtuálního stroje*. Image jsou uloženy v adresáři *image* (dvojice *.im* a *.cha*), virtuální stroje pro různé platformy v adresáři *bin*. Za zmínku stojí také adresář *doc* obsahující kompletní dokumentaci systému.

---

<sup>3</sup> V kapitole „Možnosti implementace objektově orientovaného jazyka“ jsme si vysvětlili příčiny vyšší výpočetní náročnosti.

Z instalačního CD přepokopírujeme celý systém na disk. Ze souborů v adresáři `image` odstraníme `read-only` atribut. V případě Unixu/Linuxu musíme nastavit příslušná práva. Další postup zprovoznění se liší podle operačního systému:

### 2.1.1.1 Microsoft Windows

Poklepeme na soubor `image`, který chceme spustit (přípona `.im`). Otevře se dialog pro asociaci přípony k programu. Tlačítkem `Browse...` vybereme virtuální stroj `visual.exe` pro Windows z příslušného podadresáře `bin`. Příští spuštění systému poklepnáním na soubor `.im` již proběhne automaticky<sup>4</sup>.

Po spuštění systému je třeba nastavit domovský adresář (tj. kořenový adresář systému) v menu `File/Set VisualWorks home...` (Někdy Windows samy chybně nastaví domovský adresář na hodnotu – např. `C:\vw7.2nc\image`. Potom se musí ručně odebrat poslední adresář `image` a výsledná hodnota potom bude `C:\vw7.2nc`). Nastavení domovského adresáře se zapamatuje do registrů systému.

### 2.1.1.2 Unix/Linux

Nejlépe způsob spuštění je vytvořit si v textovém editoru dávkový soubor a učinit ho spustitelným příkazem `chmod +x <jmeno_souboru>`.

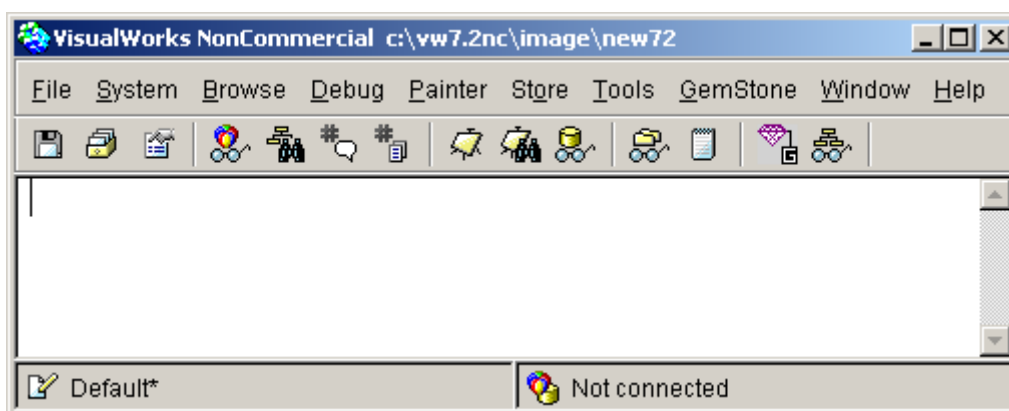
```
#!/bin/bash
VISUALWORKS="<smalltalk_image_dir>"; export VISUALWORKS
cd <smalltalk_image_dir>/image
../bin/linux86/visual <image_name>.im &
```

Poslední řádek se může lišit podle verze systému. Oproti Windows se zde kořenový adresář nastavuje proměnnou systému `VISUALWORKS` před spuštěním.

## 2.1.2 Práce se systémem

Spuštěný systém nejprve ohlásí číslo svého procesu (v UNIXu), protože byl spuštěn paralelně (znak `&` v druhém příkazu) a poté se na obrazovce objeví několik nových oken<sup>a</sup>, která již náleží systému Smalltalku.

Prvním oknem je `VisualLauncher`. Jeho úkolem je řízení celého systému a spuštění jednotlivých nástrojů a operací systému Smalltalk.

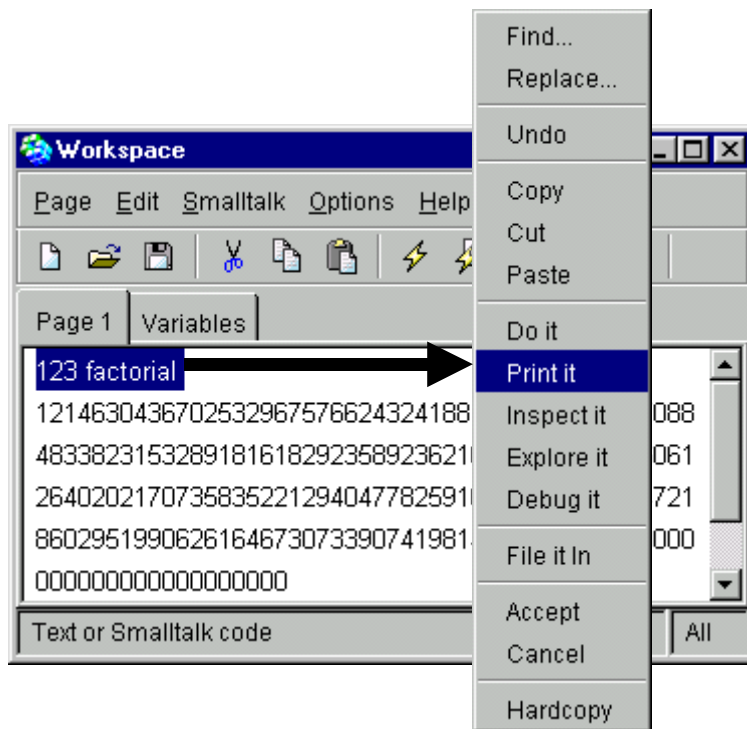



<sup>4</sup> Je tu ještě jedna možnost: pokud se soubor s virtuálním strojem a soubor `image` jmenuje stejně a liší se jen příponou (`.EXE` a `.IM`) a leží ve stejném adresáři, potom stačí Smalltalku startovat spuštěním souboru `.EXE`, který si soubor `image` otevře sám.

<sup>a</sup> Ukázky oken v našich příkladech byly pořízeny ze Smalltalku-80 běžícího pod systémem MS Windows. V případě jiného operačního systému (Apple OS, OS2, UNIX&OpenLook nebo UNIX&Motif) se budou okna lišit pouze svými okraji a záhlavím. Vnitřní komponenty oken (texty, obrázky, tlačítka, šoupátka, ...) jsou ve všech operačních systémech totožná.

Součástí VisualLauncheru je textová část pojmenovaná v jazyce Smalltalku jako objekt Transcript. Tato bílá část okna pod pásem menu obsahuje textový editor a plní také úlohu výstupní konzole. Zde se za chodu systému objevují různé hlášky o stavu systému. Okno bývá také využíváno jednoduchými programy pro textový výstup.

Dalším oknem je workspace. Je používán pro interaktivní psaní a spouštění menších smalltalkových příkazů a programů. Okna workspace a transcript v operačním systému Smalltalku plní obdobnou úlohu, jako console nebo X-term v klasickém systému UNIX.



Pomocí menu v VisualLauncheru nebo stisknutím tlačítka  si lze otevřít nové okno Workspace a v něm vyhodnocovat různé výrazy. Pro spouštění příkazů nám slouží menu, které se otvírá podržením druhého tlačítka myši kdekoli v ploše příslušného okna. Prozatím se spokojíme s volbou print it. Okna Workspace můžeme je použít k počítání aritmetických výrazů (známe-li jazyk Smalltalk) nebo k jiným operacím mající vztah k operačnímu systému (spouštění nových programů apod.). Příklady ukazují několik takových příkazů:

```
123 factorial
0.89 sin
78 + 50
Time now
Date today
Transcript show: 'Hello world!'
```

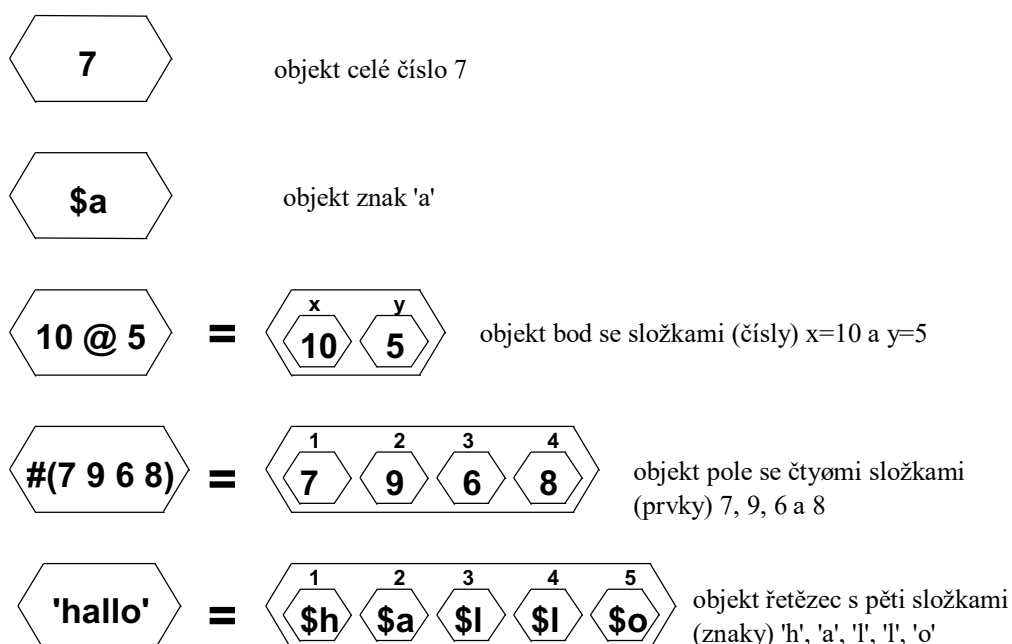
Svoje experimentování se Smalltalkem ukončíme volbou File-Exit -> VisualWorks-quit z menu VisualLauncher. Při opuštění systému je možné uložit si jeho stav do souboru na disk (quit and save), což je výhodné při další práci se systémem, protože je nám umožněno pokračovat od toho bodu, kde jsme svoji práci ukončili. Tato možnost je také výhodná pro začátečníka, který tímto způsobem může uchovat svůj narušený systém do příchodu experta.

## 2.2 Objekty a zprávy

Jak již bylo vysvětleno v kapitole „Objektově orientované programovací jazyky“, Smalltalk patří na rozdíl od tzv. hybridních objektově orientovaných jazyků, jakými jsou např. Object Pascal nebo C++, do rodiny tzv. EPOL systémů (Environment-based Pure Object Language). Ve Smalltalku je vše založeno na objektech. Na příklad čísla jsou objekty, řetězce znaků jsou objekty, okna jsou objekty. Celý jazyk je založen pouze na práci s objekty. Program ve Smalltaku neobsahuje kromě posílání zpráv objektům a přiřazování jmen objektům žádné jiné příkazy, ani volání podprogramů, funkcí nebo skoků.

Smalltalkový objekt je entita obsahující v sobě nějakou zapouzdřenou informaci, která není zvenčí přístupná. Objekt vystupuje navenek jako celek obsahující nějaké lokální údaje, jenž má vlastnost sebeidentifikace a reaguje na vnější podněty.

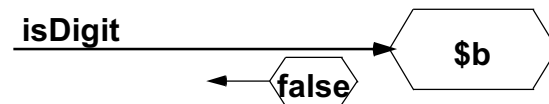
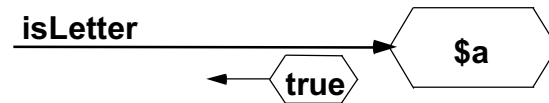
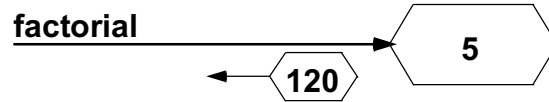
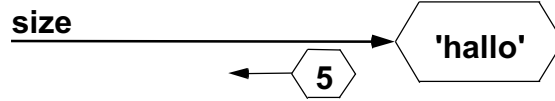
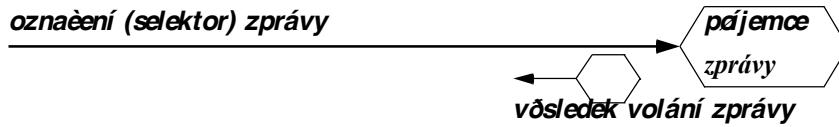
### OBJEKTY VE SMALLTALKU



Jedinou možností, jak s informací uvnitř objektu pracovat je poslat příslušnému objektu **zprávu**. Poslaná zpráva může obsahovat kromě svého jména i nějaké jiné objekty jako **parametry**. Zpráva budeme v textu označovat jako **selector**. Protože objekty mají schopnost sebeidentifikace, tak objekty na poslanou zprávu reagují vyvoláním příslušné **metody**, tj určité části kódu, která pracuje s informací uvnitř objektu. Metody tvoří u každého objektu jakýsi obal (tzv. **zed' z metod**) okolo informací uvnitř objektů. Výsledkem volání metody je vždy nějaký objekt, který může být jinde použit jako výsledek poslané zprávy. Kód metody může být využit i ke změně stavu jiných objektů (obsahuje totiž posílání zpráv) než pouze příjemce zprávy.

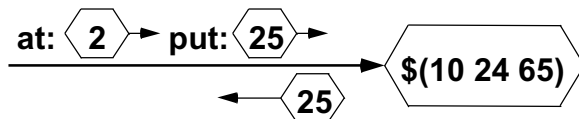
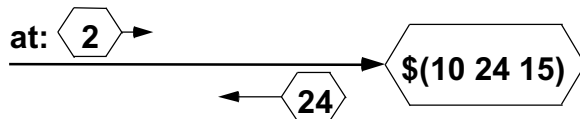
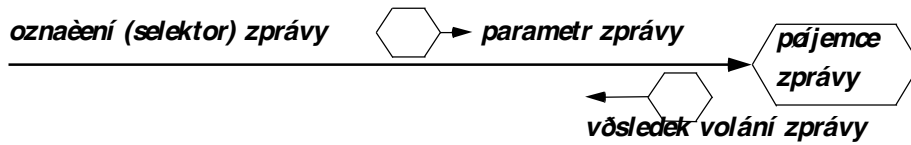
## OBJEKTY A ZPRÁVY

označení (sektor) zprávy

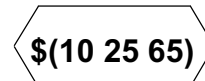


## OBJEKTY A ZPRÁVY S PARAMETRY 1. část

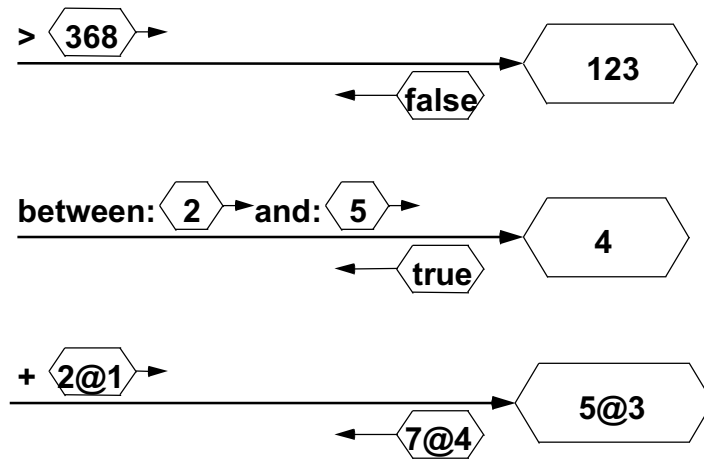
označení (sektor) zprávy



následkem této zprávy  
pøijemce zmìnil obsah na:



## OBJEKTY A ZPRÁVY S PARAMETRY 2. část



V syntaxi jazyka Smalltalk se výše uvedené příklady píší následujícím způsobem (podrobněji je syntaxe vysvětlena dále):

```
'hallo' size.           5 factorial.
$a isLetter.           $b isDigit.
#(10 24 65) at: 2.     5 + 10.
#(10 24 65) at: 2 put: 25. 123 > 368.
4 between: 2 and: 5.     (5 @ 3) + (2 @ 1).
```

Jednotlivé výrazy jsou ve Smalltalku od sebe odděleny tečkou. Je-li jednomu objektu posíláno několik zpráv v jednom výrazu, lze zprávy řadit do tzv. kaskády zpráv, ve které se jednotlivé zprávy oddělují středníkem.

```
Transcript show: 'Good morning'.
Transcript cr.
Transcript show: 'everybody'.
```

Lze napsat pomocí kaskády jako jeden výraz:

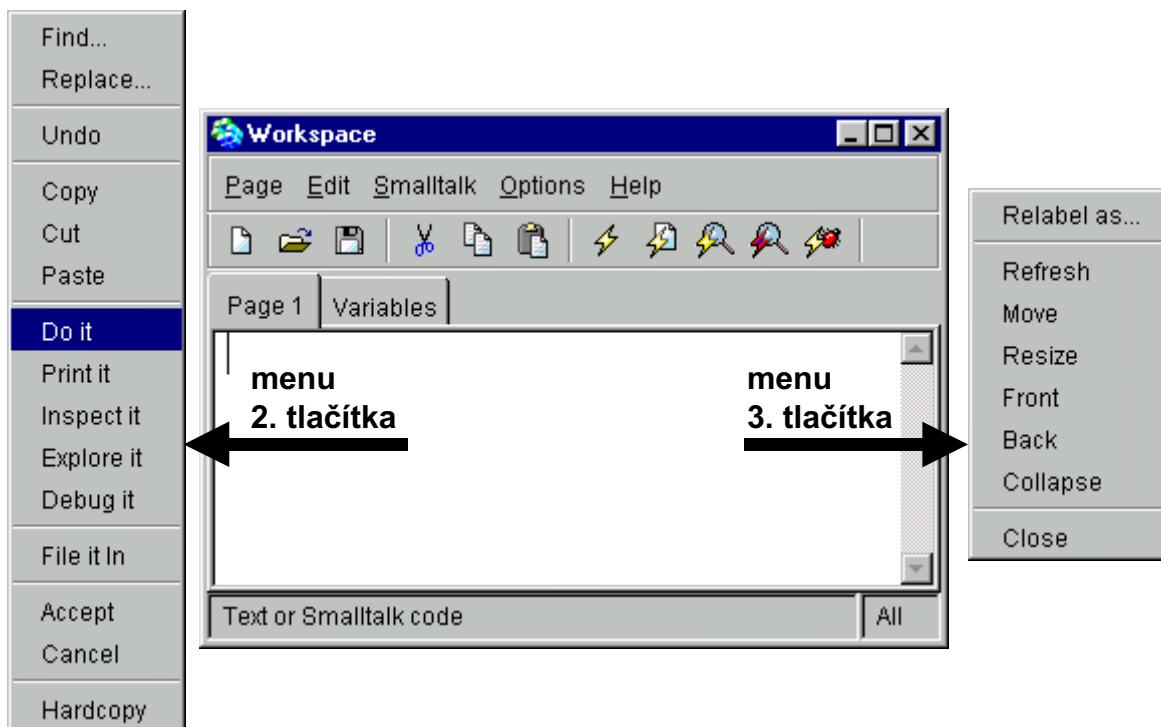
```
Transcript
  show: 'Good morning';
  cr;
  show: 'everybody'.
```

### 2.3 Uživatelské prostředí

Jak již bylo řečeno, Smalltalk je od počátku (první verze systému je z roku 1972) systém využívající okna, menu a ovládání myši. Pro ovládání systému, který byl navržen dlouho před dnes používanými grafickými nadstavbami operačních systémů, se standardně používá třílačítková myš. Označení a funkce jednotlivých tlačítek myši ještě z doby výzkumu v 70. letech v laboratořích PARC je následující:

1. **žluté tlačítko** (první - levé), které slouží k označování textu nebo k aktivaci jiných grafických objektů (spínače, šoupátka apod.).
2. **červené tlačítko** (druhé - prostřední), které slouží k aktivaci a výběru položky z vnořovacího menu majícího souvislost s obsahem příslušné oblasti okna.
3. **modré tlačítko** (třetí - pravé), které slouží k aktivaci a výběru položky z vnořovacího menu, jenž nabízí operace týkající se celého okna (přejmenování záhlaví okna, přesun okna, změna velikosti, přesun okna do pozadí či do popředí apod.). Tyto funkce jsou částečně dostupné i pomocí ikon v záhlaví jednotlivých oken.

U myši s méně než třemi tlačítky lze použít k vyvolání funkce třetího tlačítka kombinaci druhého tlačítka s klávesou Ctrl.



Systém Smalltalku není omezen pouze na jedno okno s jednou aplikací. Ve Smalltalku se běžně pracuje s několika okny, přičemž v jednom se nachází např. vytvářená aplikace, ve druhém debugger, ve třetím pořadač zdrojového kódu (angl. browser) etc. Základem prostředí jsou však vždy dva typy oken, se jménem Transcript a Workspace.

Objekt Transcript slouží jako textový výstup pro nejrůznější systémová hlášení. Okno Workspace slouží k psaní a interaktivnímu vyhodnocování Smalltalkových výrazů. (V systému může být otevřeno několik oken Workspace.) Výraz je vyhodnocen, když je nasvícen žlutým tlačítkem myši, a když je vybrána červeným tlačítkem z menu volba do it (vyhodnot') nebo print it (vyhodnot' a zobraz výsledek).

V případě, že v systému není některé z uvedených oken otevřeno, je třeba ho otevřít. V našem Smalltalku k tomu slouží **spouštěč** (VisualLauncher), jenž vlastně plní úlohu permanentně připraveného systémového menu.

## 2.4 Jazyk Smalltalku

Celý systém Smalltalku není tvořen jen překladačem tohoto programovacího jazyka, ale obsahuje i grafické vývojové a operační prostředí na úrovni samostatného operačního systému pro podporu tvorby

a běhu Smalltalkových programů se sadou nejrůznějších softwarových nástrojů. V přehledu syntaxe se budou vyskytovat pojmy, které budou průběžně objasňovány až v dalších kapitolách.

Syntaxe moderního programovacího prostředí musí splňovat tři hlavní předpoklady:

1. musí být jednoduchá a snadno zvládnutelná, extrémním příkladem je programovací jazyk LISP,
2. musí mít dostatečně bohaté výrazové prostředky, příkladem zde může být například jazyk PL/1
3. musí být člověku srozumitelná, což prakticky znamená, že jazyk by neměl obsahovat pouze zkratkovité a symbolické struktury (např. jazyk APL), ale pokud možno jednoduché vhodně stylizované věty, jako například jazyky SQL nebo COBOL.

Výše uvedené požadavky jsou však většinou protichůdné, a proto je syntaxe programovacího jazyka vždy kompromisní záležitostí. O syntaxi jazyka Smalltalk lze říci, že v oblasti OOP se jedná o mimořádně vydařenou rovnováhu uvedených požadavků na moderní programovací prostředek.

### 2.4.1 Jména objektů

Každý objekt ve Smalltalku může mít svoje jméno. Jménem rozumíme alfanumerické označení, které musí začínat písmenem. Ve Smalltalku jsou důsledně rozlišována velká a malá písmena, na což si musejí zpočátku dávat veliký pozor programátoři zvyklí pracovat s programovacími jazyky, v jejichž syntaxi na velkých a malých písmenech nezáleží (např. Pascal) a nebo je v jejich syntaxi psaní velkých a malých písmen pouze záležitost konvence (např. C).

Přiřazení jména objektu se provádí operátorem `:=`, přičemž jednomu objektu může náležet i více různých jmen.

Následující tabulka obsahuje typy objektů podle jejich pojmenování:

druh objektů	počáteční písmeno jména
Kategorie tříd	velké
Třídy a NameSpaces	velké
Třídní proměnné a proměnné z NameSpaces	velké
Instanční proměnné	malé
Pomocné proměnné	malé
Kategorie zpráv	malé
Zprávy	malé

### 2.4.2 Konstanty

Ve Smalltalku existuje šest typů objektů, které je možné v kódu programů psát v podobě konstant (angl. literal constants). Jedná se o:

1. čísla (angl. numbers)
2. znaky (angl. characters)
3. řetězce (angl. strings)
4. symboly (angl. symbols)
5. bajtová pole (angl. byte array)
6. pole jiných konstant (i polí).

K tomu navíc přistupují tři speciální konstanty, kterými jsou `true`, `false` (logické hodnoty) a `nil` (prázdný, nedefinovaný objekt).

### 2.4.2.1 Číselné konstanty

Čísla jsou zapisována obvyklou formou, kterou nejlépe následující ukázka:

```
123                -456.0                328.598
1.5897e21          3.71023e-4          -349875321
```

Zápis bez desetinné tečky nebo exponentu označuje celá čísla (angl. integers), jinak se jedná o čísla s pohyblivou řádovou čárkou (angl. float). Rozsah celých čísel je neomezený (viz. například factorial z minulé lekce), rozsah čísel s pohyblivou řádovou čárkou je v intervalu přibližně od  $10^{38}$  do  $10^{-38}$  s přesností 8 až 9 platných cifer, přičemž může záviset na hardwarové implementaci.

Celá čísla je možné zapisovat i v libovolné jiné číselné soustavě než desítkové. V tomto případě je třeba uvést číselný základ dané soustavy pomocí tzv. radix předpony:

```
16r89A0           2r11001011           8r776432
5r41003           9r21008                3r1220102
```

Kromě běžných čísel s pohyblivou řádovou čárkou je možné použít i čísla s dvojitou přesností (angl. double) s přesností 14 až 15 platných cifer a rozsahem od  $10^{307}$  do  $10^{-307}$ . Jejich zápis se od obyčejných čísel liší tím, že místo znaku `e` je v exponentu použit znak `d`:

```
1.57d0            -3.16283d12           7.906530087123d156
```

### 2.4.2.2 Znakové konstanty

Znakové konstanty se zapisují pomocí daného znaku uvozeného znakem `§`:

```
§A      znak 'A'
§       znak ' ' (mezera)
§k      znak 'k'
§+      znak '+'
§4      znak '4'
§§      znak '§'
```

### 2.4.2.3 Řetězcové konstanty

Řetězcové konstanty jsou znaky uzavřené v apostrofech (ne v uvozovkách!). Je-li apostrof také součástí daného řetězce, musí být zdvojen:

```
'take it easy'      'it''s fine'        'maybe'
'This is string.'  'say: "well"'      '123.89'
```

### 2.4.2.4 Konstanty symbol

Symbole jsou speciální řetězce, které neobsahují znak mezera. Bývají používány pro identifikaci objektů. Na rozdíl od řetězců nejsou uzavřeny apostrofy, ale jsou uvozeny znakem `#`, který není součástí symbolu a může být vynechán, jestliže je symbol elementem konstanty pole.

Příklady:

```
#sin                #Program1          #factorial
#- #<=             #exit
```

### 2.4.2.5 Bajtová pole

Představují pole bytů a zapisují se do hranatých závorek uvozených znakem # . Jednotlivé elementy v poli (čísla v intervalu 0 až 255) jsou od sebe odděleny mezerami. Bajtová pole používá systém k reprezentaci strojových dat (bajtů), např. bitmap obrázků či ikon, při běžném programování nemají příliš velký význam.

```
#[ 255 0 1 ]                #[ 3 100 51 0 21 7 128 ]
```

### 2.4.2.6 Pole jiných konstant

Jsou pole, jejichž elementy mohou být libovolné jiné konstanty (i jiná pole). Není podmínkou, aby všechny elementy byly stejného typu. Pole se zapisují do kulatých závorek se znakem # na začátku:

```
 #( 1.23 $a $b $c )                #( 'big' 'small' )
 #( 'January' 28 1993 )            #( #sin #cos 890.57 )
 #( $x $y ( 'it is' 'we go' ) (23 78 43 ( 2.8e9 16r8f ) ) 2 )
```

### 2.4.3 Proměnné

Ve Smalltalku máme čtyři typy proměnných<sup>a</sup>, kterými jsou:

1. pomocné proměnné (angl. temporary)
2. instanční proměnné
3. sdílené proměnné (angl. shared variables)
4. speciální proměnné

Celý Smalltalk je dynamický systém objektů. Každý objekt (tedy i proměnná) má svůj zrod (vytvoření), trvání a zánik (uvolnění z paměti, které probíhá automaticky). O vytvoření objektu - proměnné rozhoduje programátor ve svém programu. Pro deklaraci proměnné stačí vždy uvést pouze její jméno bez nutnosti udávat typ budoucího objektu.

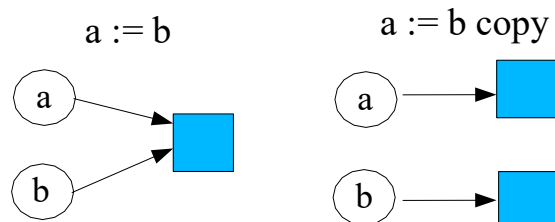
Na rozdíl od klasických programovacích jazyků ve Smalltalku pod proměnnou rozumíme označení nějakého objektu a ne paměťové místo<sup>b</sup>. Proto také operátor := ve Smalltalku nemá význam přiřazení hodnoty, ale **přiřazení jména**. Například příkaz `a := b` způsobí, že objekt, který je přístupný skrze jméno `b`, byl nyní pojmenován i jako `a`. Označení `a`, `b` jsou tedy dvě jména jednoho objektu.

Chceme-li pod označením `a` mít uložený objekt, který není s objektem `b` totožný<sup>c</sup>, ale má stejný obsah, je třeba ho nejprve z objektu `b` vytvořit a potom této kopii jméno `a` přiřadit následujícím způsobem: `a := b copy`. Tuto situaci ilustruje následující obrázek.

<sup>a</sup> O proměnných se důkladně hovoří v kapitole nazvané Instance, třídy, metatřídy a metody.

<sup>b</sup> Z tohoto důvodu se ve Smalltalku nepracuje s ukazateli v klasickém slova smyslu, neboť všechny smalltalkové proměnné jsou ve skutečnosti abstrahovanými ukazateli do paměti.

<sup>c</sup> Proto ve Smalltalku máme dvě možnosti porovnávání objektů. Výraz `a = b` testuje shodný obsah dvou objektů, které mohou být různé a výraz `a == b` testuje, zda se jedná o totožné objekty.



Rozsah platnosti každého nově vytvořeného objektu je vždy určen nějakým jiným jemu nadřazeným objektem. Jestliže zaniká nadřazený objekt, automaticky zaniká i daný objekt:

- Pomocné proměnné jsou vázány svým rozsahem platnosti na prováděnou část kódu, ve které jsou použité, a proto zanikají s provedením tohoto kódu.
- Instanční proměnné jsou vázány na instance, ve kterých představují jejich vnitřní strukturu.
- Třídní proměnné jsou vázány na příslušné třídy.
- Sdílené a globální proměnné jsou vázány na celý systém (protože ve Smalltalku je vše objektem, tak i celý systém je jedním objektem se jménem `Smalltalk`).

Mezi speciální proměnné patří `self`, odkazující se na příjemce zprávy (tedy „já“) a `super` odkazující na rodiče příjemce zprávy. `super` umožňuje zavolat původní kód metody, který přepisujeme v potomkovi, např.

```
initialize
    super initialize.
    ... přepsaný vlastní kód..
```

K těmto proměnným ještě patří `thisContext`, což je proměnná ve které je uložen objekt, ze kterého byla zpráva poslána.

## 2.4.4 Výrazy

V jazyce Smalltalku existuje kromě pojmenování objektu pouze jediný typ výrazu, kterým je poslání zprávy. Jedná se o volání jedné nebo několika zpráv, přičemž příjemce i parametry daných zpráv mohou být takové objekty, které jsou nejen konstanty či proměnné, ale i výsledky jiných zpráv – zprávy lze tedy řetězit a skládat..

### 2.4.4.1 Zprávy

Zprávy se dělí na:

1. **unární** (bez parametrů, ang.. unary messages)
2. **binární** (s jedním parametrem, angl. binary m.)
3. **slovní** (s jedním či více parametry, angl. keyword m.)

Unární zprávy jsou alfanumerické symbolové konstanty a zapisují se za příjemce zprávy:

<i>příjemce</i>	<i>selektor</i>	<i>zápis</i>
<code>zprávy</code>	<code>zprávy</code>	<code>zprávy</code>
<code>132</code>	<code>#factorial</code>	<code>132 factorial</code>
<code>3.96</code>	<code>#sin</code>	<code>3.96 sin</code>
<code>anImage</code>	<code>#copy</code>	<code>anImage copy</code>
<code>še</code>		<code>#isDigit</code> <code>še isDigit</code>

```
#( 4 7 ) #size
```

```
#( 4 7 ) size
```

Binární zprávy jsou jedno nebo dvouznakové nepísmenné konstanty a zapisují se za příjemce zprávy před parametr zprávy:

<i>příjemce</i>	<i>selektor</i>	<i>parametr</i>	<i>zápis</i>
<i>zprávy</i>	<i>zprávy</i>	<i>zprávy</i>	<i>zprávy</i>
1	#+	3	1 + 3
5.68	#>=	1.7e2	56.8 >= 1.7e2
4	#**	3	4 ** 3
a	#==	b	a == b

Slovní zprávy jsou alfanumerické symbolové konstanty, které se při zápisu píší za příjemce zprávy a jsou rozděleny na tolik částí, kolik mají parametrů (nejméně však jeden). Každá část musí být zakončena dvojtečkou:

<i>příjemce</i>	<i>selektor</i>	<i>parametry</i>	<i>zápis</i>
<i>zprávy</i>	<i>zprávy</i>	<i>zprávy</i>	<i>zprávy</i>
Transcript	#show:	'hello'	Transcript show: 'hello'
45	#min:	78	45 min: 78
9	#between:and:	8, 12	9 between: 8 and: 12
1	#to:by:	20, 2	1 to: 20 by: 2

Při zápisu zpráv do výrazu je možné používat závorky. Výsledkem volání jakékoliv zprávy jakémukoliv objektu je vždy nějaký výsledek, který je samozřejmě opět objektem, který může být použit v jiné zprávě jako příjemce nebo jako parametr.

Pořadí vyhodnocování zpráv (neurčují-li závorky jinak) se děje vždy zleva doprava a podle priority, kde největší prioritu mají unární zprávy, po nich následuje vyhodnocování binárních zpráv a nakonec slovních zpráv. Unární a binární zprávy je možné řadit vedle sebe. Je třeba mít na pozoru, že všechny binární zprávy mají stejnou prioritu, a proto jsou vyhodnocovány zleva doprava a ne podle matematické priority operátorů. Pořadí vyhodnocování objasní následující příklady:

<i>zápis</i>	<i>pořadí vyhodnocení</i>
36 sin + 1.56	(36 sin) + 1.56
a + 23 * b	(a + 23) * b
a + (23 * b)	a + (23 * b)
data add: 38 - x sqrt	data add: (38 - (x sqrt))
89 sin squared	(89 sin) squared

#### 2.4.4.2 Sekvence výrazů

Jednotlivé výrazy jsou v sekvenci mezi sebou odděleny znakem tečka. Protože tečka není součástí výrazu, ale slouží jen jako oddělovač, tak se za posledním výrazem v sekvenci nemusí psát. Řádkování nemá podobně jako v jazycích Pascal a C na syntaxi žádný vliv.

Příklad nějaké sekvence výrazů:

```
#( 23 8 'bye' ) at: 2 put: 18.  
a := b - 10.  
x := a sqrt
```

Protože přiřazením jména k nějakému objektu (kterým může být i objekt, jenž vznikl jako výsledek nějakého výrazu) ve Smalltalku zůstává zachována jeho hodnota, tak jej lze použít dále v jiných výrazech. Výše uvedenou sekvenci výrazů je tedy možné upravit beze změny významu na:

```
#( 23 8 'bye' ) at: 2 put: 18.  
x := (a := b - 10) sqrt
```

### 2.4.4.3 Kaskáda zpráv

S kaskádou zpráv jsme se již setkali. Používáme ji tehdy, když za sebou následuje v jedné sekvenci několik výrazů, které jsou složeny z různých zpráv posílaných témuž objektu. Při použití kaskády je příjemce napsán pouze poprvé a jednotlivé zprávy kaskády jsou mezi sebou odděleny středníkem.

příklad:

```
x at: 1 put: 1.56e2.  
x at: 2 put: 2.78.  
x at: 3 put: 'hello'
```

pomocí kaskády:

```
x at: 1 put: 1.56e2; at: 2 put: 2.78; at: 3 put: 'hello'
```

nebo přehledněji:

```
x at: 1 put: 1.56e2;  
  at: 2 put: 2.78;  
  at: 3 put: 'hello'
```

nebo jiný příklad

```
Transcript cr.  
Transcript show: 'ready'.  
Transcript cr
```

pomocí kaskády:

```
Transcript cr;  
  show: 'ready';  
  cr
```

### 2.4.4.4 Návratový výraz

Návratový výraz (angl. return expression) bývá v sekvenci výrazů používán pro označení takového výrazu, který obsahuje výsledek a zároveň ukončení výpočtu sekvence výrazů. Není-li v sekvenci návratový výraz uveden, tak je za výsledek považován výsledek posledního výrazu v sekvenci a výpočet je řádně ukončen až po provedení všech výrazů sekvence. Návratový výraz je označen na začátku znakem ^ (angl. caret).

### 2.4.4.5 Použití pomocných proměnných

Pro výpočet jsou většinou nezbytné pomocné proměnné (angl. *temporaries*). Používají se při práci ve `Workspacu`, uvnitř metod objekt objektů a v blocích. Ve `Smalltalku` stačí na začátku sekvence výrazů, ve kterých budou pomocné proměnné používány, uvést seznam jejich jmen uzavřených ve znacích `|` bez ohledu na jakou budou odkazovat hodnotu. Pomocná proměnná má vždy na počátku hodnotu `nil`, tj. prázdný, nedefinovaný objekt, a je možné ve výpočtu jejím jménem pojmenovat jakýkoliv jiný objekt. Po ukončení výpočtu sekvence pomocné proměnné zanikají. Příklad přináší ukázkou sekvence výrazů s použitím pomocných proměnných a s použitím návratového výrazu:

```
| anArray x y |  
anArray := #( 1 3 2 4 ).  
x := anArray at: 2.  
y := anArray at: 1.  
^ x + y
```

#### 2.4.4.6 Bloky výrazů

Bloky výrazů představují vyčleněné sekvence výrazů. Blok výrazů je samozřejmě objektem, může být pojmenován, mohou mu být posílány příslušné zprávy, a může být použit v jiných výrazech (zprávách) jako příjemce nebo parametr (viz příklad programu na konci lekce). Výrazy, které patří do jednoho bloku se píší (včetně deklarace jejich pomocných proměnných) do hranatých závorek. Na ukázce přinášíme blok výrazů z předchozí ukázky:

```
A1 := [ | anArray x y |
      anArray := #( 1 3 2 4 ).
      x := anArray at: 2.
      y := anArray at: 1.
      ^ x + y ]
```

Nyní máme tedy objekt-blok výrazů, který je pojmenovaný A1. Protože se jedná o objekt, můžeme mu kdykoliv poslat příslušnou zprávu, kterou bude požadavek na vyhodnocení v něm uschovaných výrazů. Tato zpráva se jmenuje #value, a tedy po poslání této zprávy objektu A1 jako A1 value dostáváme výsledek, tj. číslo 4.

Je třeba mít na paměti, že výrazy v blocích se vyhodnocují vždy až při příslušném požadavku na jejich vyhodnocení, a ne při vytvoření bloku. Tentýž blok proto může v různých situacích vracet různé výsledky.

Protože bloky slouží k výpočtům v jiných výrazech, obsahuje Smalltalk mechanismus, kterým je možné posílat data dovnitř bloků pomocí deklarace speciálních pomocných proměnných, které slouží jako jejich vstupní parametry. Za výstupní údaj z bloku je považován objekt-výsledek, který je dán výslednou hodnotou výrazů v bloku. Pro vstup hodnot slouží slovní zprávy s parametry #value:, #value:value:, #value:value:value: atd.

Pomocné proměnné sloužící pro vstup dat se deklarují pomocí formálního jména s dvojtečkou na začátku a píší se hned za levou hranatou závorku. Deklarace těchto proměnných musí být od ostatního kódu (případná deklarace vnitřních pomocných proměnných a následné výrazy) oddělena znakem | .

Celá syntaxe pro blok výrazů je tedy následující:

```
[ :arg1 :arg2 ... argn |
  | temp1 temp2 ... tempn |
  statement1.
  statement2.
  ...
  statementN ]
```

Příklad přináší jednoduchou ukázkou bloku se dvěma vstupními parametry, které jsou v bloku označené jako a, b s jednou pomocnou proměnnou c.

```
Pyth1 := [:a :b | | c | c := (a ** 2) + (b ** 2). ^- c sqrt ].
```

bez použití pomocné proměnné c by blok dávající shodné výsledky vypadal:

```
Pyth2 := [:a :b | ^((a ** 2) + (b ** 2)) sqrt ].
```

Takto vytvořené objekty Pyth1 i Pyth2 je možné používat, jak je z kódu v bloku na první pohled zřejmé, pro výpočet přepony v pravoúhlém trojúhelníku pomocí dvou odvěsen. Tedy výraz Pyth1 value: 3 value: 4 vrátí hodnotu 5.0, výraz Pyth1 value: 1 value: 1 vrátí hodnotu 1.414213 atd.

## 2.4.5 Jednoduchý program

Výše uvedený přehled obsahuje veškerou syntaxi jazyka Smalltalk. Práce ve Smalltalku je záležitostí kombinace těchto prostředků a vlastností systému (dynamičnost, dědičnost, polymorfismus, ...). Smalltalk nepotřebuje klasické procedurální prostředky, jakými jsou ukazatele, skoky, cykly, podprogramy, procedury a další prostředky, bez kterých si klasické procedurální programování nedovedeme představit.

Následující ukázka obsahuje jednoduchý program, který je možné napsat v pracovním okně (Workspace) a vyhodnotit (`print it`). Za povšimnutí stojí použití bloků jako parametru ve slovní zprávě `#to:do:` (podobných zpráv je více a budeme se jimi zabývat v následující lekci), kde tato zpráva a její parametry plní úlohu procedurálního cyklu. Program zjistí ze vstupního řetězce znak s největším kódem. Pro porovnání uvádíme i jeho procedurální variantu ve fiktivní verzi jazyka C:

### *jazyk C*

```
void main
{
    char s[ ];
    int i;
    char c, temp;
    printf("enter text: ");
    readLine(s);
    c = ' ';
    for (i = 1; i <= length(s); i ++)
        {
            temp = s[i];
            if (temp > c) c = temp;
        }
    printChar(c);
}
```

### *jazyk Smalltalk*

```
| s c |
s := Dialog request: 'enter text:'.
c := $ .
1 to: s size do:
[:i |
    |temp|
    temp := s at: i.
    temp > c ifTrue: [c := temp]
].
^c.
```

Oba programy byly záměrně napsány tak, aby si byly velmi podobné. Není pochyb o tom, že lze v obou jazycích stejný algoritmus napsat úsporněji. Při dobré znalosti standardních (rozuměj v systému již obsažených) tříd objektů a zpráv ve Smalltalku je však možné výše uvedený program velmi zjednodušit<sup>a</sup>:

```
^(Dialog request: 'enter text:') asSortedCollection last.
```

a pro zajímavost kdybychom si představili nějaký fiktivní český Smalltalk, tak:

```
^(Dialog pozaduj: 'zadej text:') setridene posledni.
```

---

<sup>a</sup> Tento příklad byl také záměrně vybrán proto, aby ukázal výhody OOP i v oblasti základní algoritmizace, která je někdy považována za doménu klasického procedurálního programování. OOP totiž není jen nástrojem pro počítačovou grafiku a programování uživatelských rozhraní.

## 2.5 Instance, třídy, metatřídy a metody

Shrňme si nyní poznatky z kapitoly „Objekt jako abstraktní datový typ“. Každý objekt obsahuje nějakou lokální, navenek ukrytou informaci. Jedinou možností, jak se dostat k vnitřní struktuře objektu zvenčí, je poslat danému objektu zprávu. Tato zpráva, kterou můžeme považovat za **žádost o operaci**, se dostane k obalu daného objektu. Obal objektu je složen z relativně malých částí kódů nazývaných **metody**, které mají za úkol reagovat na příslušné zprávy. Jestliže tedy pošleme objektu zprávu, na kterou objekt umí reagovat, je na základě poslané zprávy spuštěn kód příslušné metody. Není-li zprávě odpovídající metoda nalezena, objekt hlásí systému chybu. Zprávy kromě označení žádosti mohou obsahovat i **parametry**, jež jsou vlastně vstupními daty pro kódy metod. Po vykonání kódu metody je vyslán vždy právě jeden objekt, jež může být považován za výsledek volání zprávy. S hodnotami uvnitř objektu mohou manipulovat pouze kódy jeho vlastních metod.

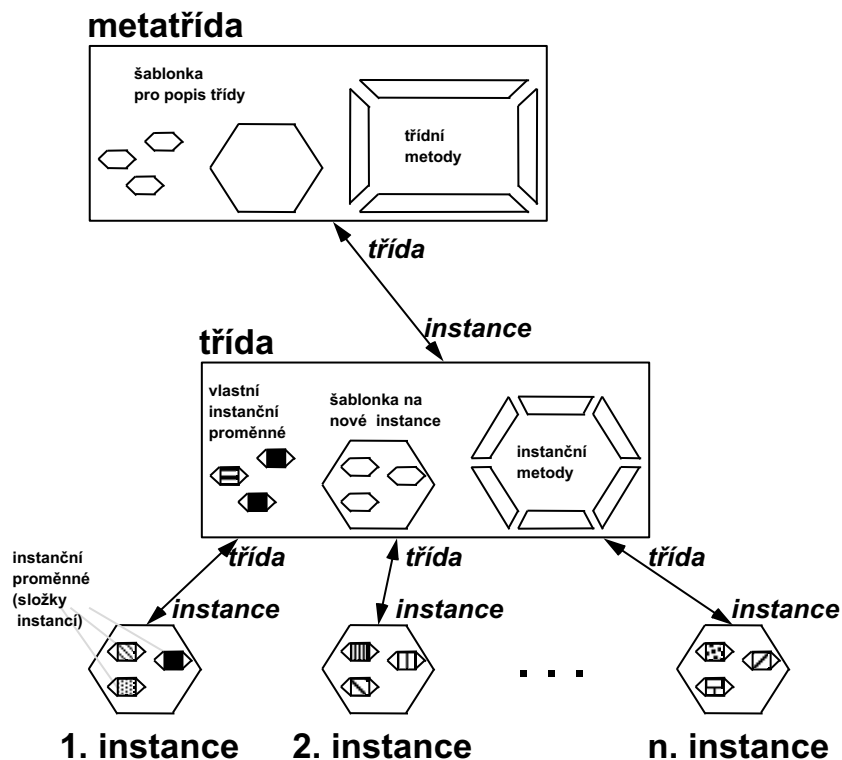
V systému jsou veliká množství objektů, které se sice navzájem liší svými vnitřními hodnotami, ale které reagují na přicházející zprávy stejnými metodami. Jedná se například o množinu všech celých čísel, o množinu všech řetězců, o množinu všech obrázků, o množinu všech grafických oken atd. Bylo by neúčelné této vlastnosti nevyužít, neboť kromě ztráty jiných výhod, které budou popsány později, by docházelo k velkému plýtvání pamětí. Kódy metod proto nejsou uloženy u každého objektu zvlášť, a jsou uloženy jen v některých objektech zvaných **třídy**. Třídy jsou tedy takové objekty, které obsahují kódy metod pro jiné objekty, jež jsou nazývány **instance**. Tyto metody jsou proto nazývány **instanční metody** (angl. **instance methods**). Každá instance si musí pamatovat svoji třídu, protože ji potřebuje pro vyhodnocení svých přijatých zpráv. Můžeme říci, že všechny instance náležející jedné třídě mají ve své třídě jednotně uschován svůj obal z kódů metod.

Kromě toho, že jsou ve třídách uschovány (jako jejich vnitřní data) kódy metod pro potřeby jejich instancí, tak třídy slouží i ke **tvorbě jejich nových instancí**. Třídy jsou tedy také **generátory nových objektů**. Tvorba nové třídy či instance, přidání kódu nové metody, změna kódu existující metody nebo odstranění kódu nějaké metody ve třídě se děje samozřejmě pomocí jediného možného mechanismu, kterým je posílání příslušné zprávy.

Podívejme se nyní podrobněji na třídy jako na objekty. Třídám, jako každým jiným objektům, jsou posílány zprávy. Tyto zprávy se formálně neliší od zpráv, které jsou posílány instancím, a proto na ně třídy musejí také reagovat spuštěním příslušných kódů odpovídajících metod. Nemohou to však být kódy, které si v podobě vnitřních dat třídy uchovávají pro svoje instance. Tyto kódy totiž musejí být součástí obalů daných tříd, aby chránily jejich vnitřní údaje navenek a umožnily reagovat na přijaté zprávy. Kromě toho je ve Smalltalku také možné třídy vytvářet a je také možné vytvářet, měnit a rušit kódy metod, kterými tyto třídy reagují na posílání zpráv. **Se třídami se ve Smalltalku pracuje stejně jako s jejich instancemi, protože i třídy mají všechny vlastnosti objektů.**

V systému proto musejí být přítomny další objekty, které umožňují třídy vytvářet a pracovat s nimi. Tyto objekty, které jsou nazývány **metatřídy**, jsou ve stejném vztahu ke třídám, jako jsou třídy ke svým instancím. Každá třída má svoji metatřidu, kterou si musí pamatovat, neboť ji potřebuje pro vyhodnocení svých přijatých zpráv. V metatřídách jsou jako vnitřní data uloženy kódy metod potřebné pro vyhodnocování zpráv posílaných třídám. Tyto metody jsou nazývány **třídní metody** (angl. **class methods**). Každá metatřída má vždy právě jednu instanci, kterou je k ní příslušná třída.

Metatřídy jsou však také objekty, které reagují na zprávy, a proto jsou instancemi nějaké další "super metatřídy". Tento objekt opravdu existuje, jmenuje se `MetaClass`, a je pro všechny metatřídy v systému společný. Všechny metatřídy jsou jeho instancí. Tato "super metatřída" má také svoji třídu, ve které je její jedinou instancí. Tato metatřída se jmenuje `MetaClass class`. Aby se ve Smalltalku neřetězily na sebe třídy a instance donekonečna, tak platí, že `MetaClass` je instancí třídy `MetaClass class` a zároveň `MetaClass class` je instancí třídy `MetaClass`.



Síla Smalltalku spočívá v tom, že se všemi uvedenými objekty je možné programátorsky pracovat. Smalltalk totiž obsahuje nástroje, kterými je možné vytvářet, rušit či měnit třídy i metatřídy, a měnit tím celé systémové prostředí, tj. jeho vzhled i chod systému. Kromě toho jsou v systému přístupné všechny zdrojové kódy (i kódy pro překladač, pro popis metatřídního chování a pro mechanismus vyhodnocování zpráv).

Práce s instancemi, třídami a metatřídami je pro programování v systému nezbytná. Práce s objekty `MetaClass` a `MetaClass class` je sice také možná, ale málokdy vede ke zdokonalení systému. (V případě experimentů s těmito objekty vřele doporučujeme si pořídit záložní kopii Smalltalku)

Pro vyzkoušení uvedených vlastností systému předkládáme čtenáři několik příkladů. Ve Smalltalku jsou třídy pojmenovány vždy symbolem s velkým písmenem na začátku. Metatřídy jsou označeny jménem k nim příslušných tříd s příponou `class`. Jestliže je libovolnému objektu poslána unární zpráva `#class`, tak objekt vrátí jako výsledek volání této zprávy svoji třídu.

Napište v okně `Workspace` a vyhodnocujte pomocí volby `print it` v menu následující výrazy:

výraz	výsledek
<code>#(1 3 2 4) class</code>	<code>Array</code>
<code>12.34 class</code>	<code>Float</code>
<code>123 class</code>	<code>SmallInteger</code>
<code>(#(1 3 2 4) at: 3) class</code>	<code>SmallInteger</code>
<code>-1.7823e10 class</code>	<code>Float</code>
<code>#(red green brown) class</code>	<code>Array</code>
<code>'hello world!' class</code>	<code>String</code>
<code>('hello' at: 2) class</code>	<code>Character</code>
<code>'take it easy' class</code>	<code>String</code>
<code>\$x class</code>	<code>Character</code>
<code>Array class</code>	<code>Array class</code>
<code>78.5 class</code>	<code>Float class</code>
<code>Float class</code>	<code>Float class</code>
<code>78.5 class class</code>	<code>MetaClass</code>
<code>78.5 class class class</code>	<code>MetaClass class</code>
<code>78.5 class class class class</code>	<code>MetaClass</code>

Mezi instance patří z našeho příkladu **objekty**  `#(1 3 2 4)`,  `12.34`,  `123`,  `2`,  `-1.7823e10`,  `#(red green brown)`,  `'hello world!'`,  `$e`,  `'take it easy'`,  `$x a 78.5`.

Třídami těchto instancí jsou objekty  `Array`,  `Float`,  `SmallInteger`,  `String` a  `Character`. Tyto objekty jsou zároveň instancemi svých metatříd, kterými jsou objekty<sup>5</sup>  `Array class`,  `Float class`,  `SmallInteger class`,  `String class` a  `Character class`. Metatřídy jsou instancemi objektu  `Metaclass`, který je instancí objektu  `Metaclass class` (a naopak).

Pro pozorného čtenáře si ještě uvedme dvě zprávy  `isKindOf:` a  `respondsTo:`:

```
78.9 isKindOf: Number                true
```

protože  `78.9` je instancí třídy  `Float`, která je potomkem třídy  `Number`.

```
78.9 respondsTo: #sin                true
```

protože objekt  `78.9` dokáže zpracovat zprávu  `#sin`.

Na tomto místě je vhodné se zmínit o systému objektů v jazycích  `Object Pascal` a  `C++`. Jejich uživatelé vědí, že některá data v těchto jazycích nejsou reprezentována v podobě objektů. Ale kromě toho také v obou jazycích, i když jsou jejich objektová rozšíření také založena na třídě-instančním modelu, třídy jako objekty neexistují. Pojem třída zde totiž splývá s pojmem popis datového typu ve zdrojovém kódu programu, což prakticky znamená, že přeložený program pracuje pouze s instancemi a za jeho chodu není možné s třídou pracovat, protože existuje jen jako zdrojový kód. Všechny instance tříd potom musejí být známy (deklarovány) již v době překladače zdrojového kódu, nebo je třeba se uchýlovat k pomocným konstrukcím pro tvorbu nových instancí (např. tzv. konstruktory). Vzhledem k tomu, že se třídy nechovají jako objekty, tak v těchto jazycích chybí metatřídy.

## 2.5.1 Instanční proměnné

Již víme, že každý objekt má nějakou vnitřní strukturu. Prakticky to vždy znamená, že objekty se skládají z jiných objektů, které jsou nazývány **instanční proměnné objektu** (angl. *instance variables*). S těmito proměnnými, jenž tvoří vnitřní data uschovaná v objektu, pracují příslušné kódy metod. Proměnné objektu proto musejí být pro tyto kódy přístupné, což je ve  `Smalltalku` umožněno dvěma způsoby.

Jednou z možností je **pojmenování proměnných**. Proměnné objektu jsou pojmenovávány symboly s malým písmenem na začátku. Jako příklad nám mohou posloužit instance třídy  `Point` (česky bod), kde každá instance obsahuje číselnou hodnotu vodorovné ( `x`) a svislé ( `y`) souřadnice. Každá instance třídy  `Point` tedy obsahuje proměnné, které jsou pojmenované  `x` a  `y`. Příklad kódu instanční metody, která je uschovaná ve třídě  `Point`, a která je součástí rozhraní každé instance této třídy, je například metoda, která počítá polární souřadnici  `r`, jako vzdálenost bodu ( `x`,  `y`) od bodu ( `0`,  `0`). Tato instanční metoda reaguje na zprávu  `#radius`, a její kód může vypadat následovně:

```
radius
    ^((x * x) + (y * y)) sqrt
```

Posleme-li tedy nějaké instanci třídy  `Point` zprávu  `#radius`, a víme-li, že tato instance představuje např. bod ( `3`,  `4`) tj. hodnota její proměnné  `x` je rovna  `3` a hodnota její proměnné  `y` je rovna  `4`, tak jako výsledek dostáváme hodnotu  `5`.

---

<sup>5</sup> Ve skutečnosti metatřídy ve  `Smalltalku` jména nemají. Název skládající se z jména třídy a slova “class” je ve skutečnosti poslání unární zprávy “class”.

Jiným příkladem může být myšlená třída objektů, které jsou součástí nějaké databáze. Třída se potom může jmenovat např. `Zakaznici` a proměnnými instancí této třídy mohou být objekty `jmeno`, `prijmeni`, `adresa`, `konto` atp.

Ve Smalltalku však mohou být i objekty, jejichž vnitřní struktura je co do počtu složek (v ní uschovaných dalších objektů) proměnlivá. Typickým příkladem jsou objekty tříd, které již známe, a to `Array`, `ByteArray` a `String`. U těchto objektů je proto použit druhý z možných způsobů označování proměnných, kterým je **indexace proměnných**. Jednotlivé proměnné objektu potom nemají svá jména, ale mají svoje číselné označení. Jak se s těmito objekty a s jejich proměnnými pracuje, již známe. Jedná se totiž o zprávy `#at:` a `#at:put:`, jak nám ukazuje i následující příklad:

```
'hallo' at: 3
```

tato zpráva vrátí třetí složku (proměnnou) objektu `'hallo'`, kterým je znak `$l`.

```
'hallo' at: 2 put: $e
```

tato zpráva nastaví druhou složku (proměnnou) na znak `$e`. Objekt `'hallo'` tedy změnil svůj obsah na `'hello'`.

Instance s indexovanými proměnnými mohou být navzájem různě velké, a přesto mohou náležet stejné třídě (např. řetězce). Třídy, kterým náležejí takovéto instance, se nazývají **variabilní třídy** (angl. `variable class`).

## 2.5.2 Instanční proměnné tříd

Právě uvedené proměnné objektů jsou tzv. **instanční proměnné**. Jako instanční jsou označeny proto, že tvoří vnitřní strukturu instancí.

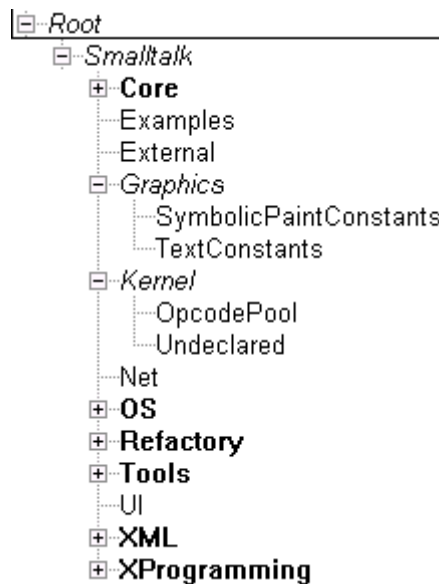
Víme, že pro vytváření nových instancí a pro uschování kódů metod instancí nám slouží třídy. Ve třídách proto musí být uchována také informace o tom, jaké instanční proměnné (ne hodnotou, ale strukturou) její instance obsahují. V každé třídě tedy musí být kromě kódu metod instancí také uchována tzv. **šablonka** (angl. `template`) **instančních proměnných** pro instance této třídy. Tato šablonka se ve Smalltalku skládá se seznamu jmen instančních proměnných a z informace, jestli instance také obsahují indexované proměnné či nikoliv (tj. jestli je třída variabilní či nikoliv).

Protože třídy jsou také objekty a samy se chovají jako instance metatříd, tak mohou také obsahovat (kromě šablonek instančních proměnných pro své instance) svoje **vlastní instanční proměnné**. Tyto proměnné jsou dostupné v kódech metod, kterými reagují třídy na přicházející zprávy. Protože kódy těchto metod jsou uschovány v metatřídách, tak i šablony pro tyto instanční proměnné tříd jsou uschovány v metatřídách. Tyto instanční proměnné tříd nejsou ale dostupné v kódech metod pro instance tříd. Právě tak nejsou dostupné instanční proměnné tříd v kódech metod pro třídy, protože zde platí princip zapouzdření (= proměnné uvnitř nějakého objektu jsou přístupné pouze pro jeho vlastní metody).

## 2.5.3 NameSpaces a ostatní proměnné ve Smalltalku

V systému Smalltalk-80 existuje kromě instančních proměnných ještě jeden typ proměnných, které se na rozdíl od instančních proměnných se označují symboly s velkým písmenem a nepředstavují složky ani instancí ani třídy samotné, protože jsou implementovány jako prvky zvláštních oblastí paměti. Tyto oblasti paměti se označují `NameSpaces` a mají hierarchickou strukturu podobně jako adresáře v souborovém systému na disku.

Na obrázku je vidět tato hierarchie pro základní verzi VisualWorks 7.2:



Znamená to tedy, že uvnitř každého jednoho NameSpace mohou být nějaké pojmenované objekty jako jeho prvky neboli proměnné. Jejich jména by měla být vždy s velkým písmenem na začátku a mohou to být:

1. Jednotlivé objekty-proměnné, které se označují jako „NameSpace variables“.
2. Třídy, což je nejobvyklejší druh pro NameSpace proměnnou.
3. Proměnnou uvnitř NameSpace může být další NameSpace.

Vzhledem k hierarchické struktuře a možnosti uvádět stejná jména pro různé objekty v různých NameSpaces mají objekty ve Smalltalku také tzv. plná neboli absolutní jména anglicky označovaná jako „full qualifiers“ (mj. dostupná pomocí unární zprávy `fullName`). Například třída `FileViewer` má plné jméno `Tools.FileTools.FileViewer`. Koncepte NameSpaces také dovoluje definovat objekty, které jsou mimo vlastní NameSpace nedostupné pomocí atributu „private“.

Třídy jsou vždy členem nějakého namespace. Účel je v tom, ve dvou různých namespace může být třída se stejným názvem. Může tak existovat třída `Tools.Parser` i `XML.Parser`, ale nikdy nemohou být dvě třídy se stejným názvem v jednom namespace.

Kromě uvedených hierarchicky uspořádaných NameSpaces je i podobná struktura uvnitř každé třídy. Lze si to představit tak, že každá třída má pro sebe a pro všechny svoje instance svůj vlastní privátní NameSpace, uvnitř kterého je také možné definovat proměnné, které se označují jako sdílené proměnné (shared variables).

Následující tabulka ukazuje, jak jsou různé druhy proměnných dostupné pro různé druhy metod.

<i>druh proměnných</i>	<i>dostupné v kódech instančních metod jedné třídy</i>	<i>dostupné v kódech třídních metod jedné třídy</i>	<i>dostupné v kódech metod objektů jiných tříd stejného NameSpace</i>
instanční v instancích	<b>ano</b>	<b>ne</b>	<b>ne</b>
instanční ve třídách	<b>ne</b>	<b>ano</b>	<b>ne</b>
Sdílené proměnné v třídě	<b>ano</b>	<b>ano</b>	<b>ne</b>
proměnné v NameSpaces	<b>ano</b>	<b>ano</b>	<b>ano</b>

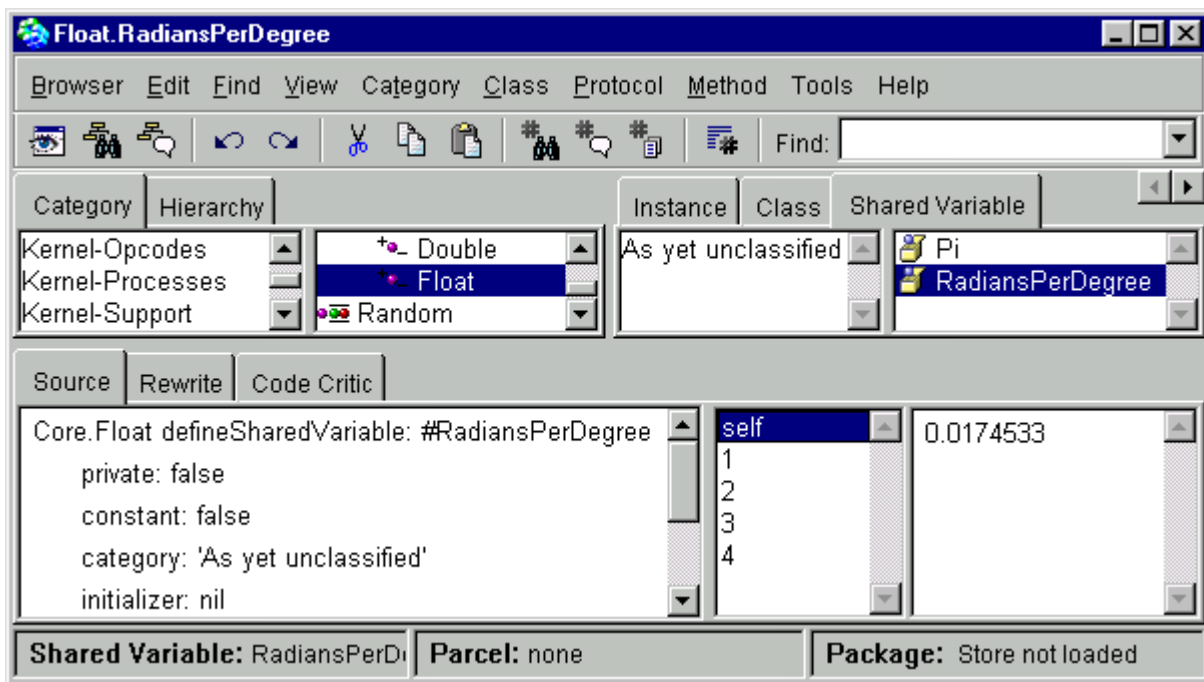
Význam různých druhů proměnných je v tom, že si vždy můžeme zvolit takovou strukturu, kde hodnota uvnitř proměnné je dostupná pouze pro objekty, kde je potřeba, a pro ostatní objekty, kde se používat nebude, tato hodnota může zůstat skrytá. Právě ve tvorbě velkých systémů v praxi je velmi důležité

stavět kód, který není zatížen nežádoucími vazbami mezi objekty, které by vedly k jeho horší udržovatelnosti a spolehlivosti.

Příklad:

```
Smalltalk.Core defineClass: #Point
  superclass: #{Core.ArithmeticValue} odkud dědíme
  indexedType: #none
  private: false dosazitelnost mimo NameSpace
  instanceVariableNames: 'x y ' instanční prom. v instancích
  classInstanceVariableNames: '' instanční proměnné v třídě
  imports: ''
  category: 'Graphics-Geometry'
```

a ještě jeden příklad na sdílené proměnné:



### 2.5.4 Architektura třídního systému

Třídy objektů jsou v systému Smalltalk hierarchicky uspořádány do stromové struktury podle vlastností, které jsou definovány kódy metod instancí a šabloněk instančních proměnných.. Na vrcholu stromu stojí třída pojmenovaná Object, která implementuje všechny základní metody, jenž jsou pro všechny objekty společné. Mezi tyto základní metody všech objektů patří např. schopnost porovnání se s jiným objektem, schopnost vytvořit svoji kopii, schopnost identifikace své třídy apod.

Od třídy Object je odvozeno velké množství dalších tříd, jejichž instance mají kromě vlastností popsaných ve třídě Object další specifické vlastnosti. Tyto třídy jsou nazývány **podtřídami** (angl. subclass) třídy Object, přičemž třída Object je jejich společnou **nadtřídou** (angl. superclass). Mezi tyto podtřídy patří například třída Magnitude, pro jejíž instance lze definovat uspořádání podle velikosti, třída Collection, jejíž instance mají schopnost obsahovat množiny jiných objektů, třída Stream, jejíž instance mají schopnost čtení a zápisu objektů atd.

Od podtříd třídy Object mohou být dále odvozeny další třídy objektů a od nich další. V systému Smalltalk je u určitých tříd běžné několik takovýchto úrovní. Například třída SmallInteger, jejíž instance jsou čísla v intervalu -536870912 až 536870911 (29 bitů ve čtyřech bajtech v paměti), je od třídy Object odvozena následujícím způsobem:

Object

v této třídě jsou kódy metod společné pro všechny objekty.

Magnitude

podtřída třídy Object. Jsou zde kódy metod, pomocí kterých je možné objekty uspořádat mezi sebou podle velikosti.

ArithmeticValue

podtřída třídy Magnitude. Jsou zde kódy metod pro základní aritmetické operace.

Number

podtřída třídy ArithmeticValue. Jsou zde kódy metod pro operace se všemi čísly (např. ln, exp, sin, cos,...).

Integer

podtřída třídy Number. Jsou zde kódy metod pro operace typické pro celá čísla (např. factorial).

SmallInteger

podtřída třídy Integer. Jsou v ní kódy metod pro celočíselné operace, které jsou charakteristické pro celá čísla, jejichž paměťová reprezentace vystačí se čtyřmi bajty.

Pro Smalltalk platí, že každá třída může mít libovolné množství podtříd a přímo nejvýše jednu nadtřidu, přičemž třída Object nadtřidu nemá a ostatní třídy mají právě jednu.

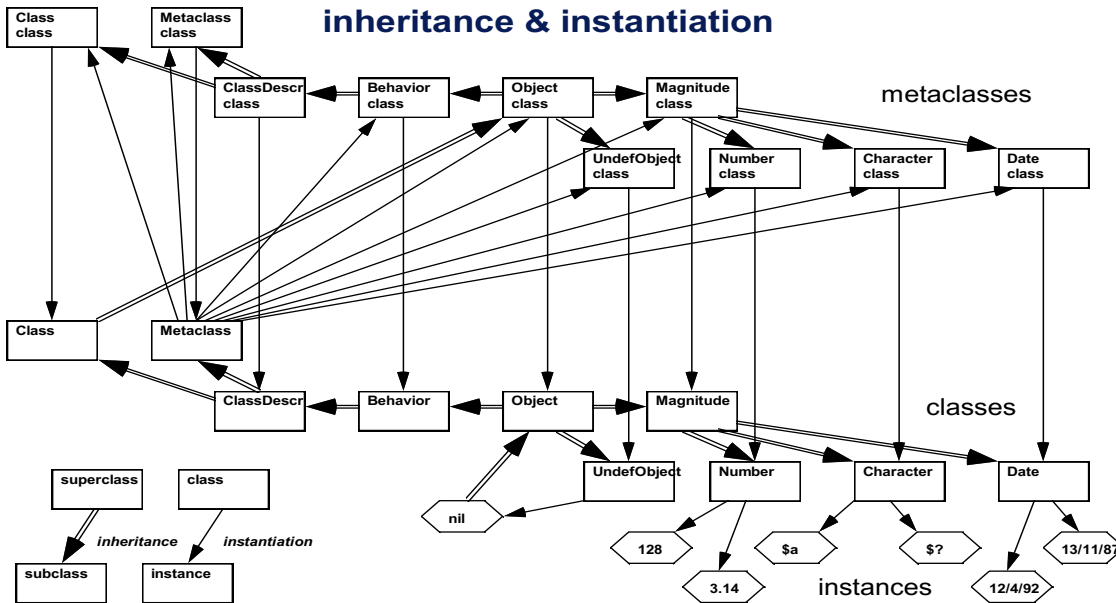
Hierarchie tříd je výhodná v tom, že kódy metod, které jsou stejné pro více jinak navzájem různých tříd, jsou uloženy v jejich společné nadtřídě, tedy na jen jednom místě v paměti. O třídách, které daný kód využívají, potom říkáme, že tento kód **dědí** se své společné nadtřídě.

Aby mohly třídy mezi sebou dědit, tak ve Smalltalku musí být přítomen **vyhledávací mechanismus** pro kódy metod příslušných tříd, který musí pracovat v době běhu programu. Jestliže totiž není nalezena k přijaté zprávě odpovídající metoda v množině kódů metod třídy, je třeba kód metody hledat v nadtřídě. Není-li kód metody nalezen ani tam, je třeba postoupit o další úroveň vzhůru na nadtřidu nadtřídě atd. Mechanismus končí buď nalezením příslušného kódu nebo chybovým hlášením o nemožnosti nalezení kódu.

Každá třída kromě kódů metod pro své instance ze své nadtřídě také **dědí** ještě **šablony** pro instanční proměnné svých instancí. Struktura instancí každé třídy je tedy dána sjednocením informací ze šablony třídy a ze šablon všech příslušných nadtříd. Jinými slovy to znamená, že instance každé třídy má kromě svých specifických instančních proměnných stejnou strukturu (tj. stejné instanční proměnné), jako instance její nadtřídě.

Ve Smalltalku je možné do mechanismu dědění kódů metod zasahovat. Nejčastějším způsobem je opětovná definice kódu metody. Prakticky to znamená, že kód metody z nadtřídě, který by mohl být do třídy děděn, je nově nadefinovanou stejnojmennou metodou **odstíněn** a nedědí se.

## MetaClasses, Classes, instances inheritance & instantiation



Mechanismus dědičnosti platí jak pro třídy, tak i pro metatřídy. Znamená to, že jestliže je nadtrídou třídy `Integer` třída `Number`, potom je také nadtrídou metatřídy `Integer class` metatřída `Number class`. Oba dva systémy (trídni a metatřídni) jsou však od sebe **oddělené**<sup>a</sup>. Kódy uložené ve třídách reagují pouze na zprávy posílané instancím. Kódy uložené v metatřídách reagují pouze na zprávy posílané třídám. Jestliže je vyhledáván kód instancní metody, je vyhledáván pouze v kódech instancních metod tříd. Jestliže je vyhledáván kód třídni metody, je vyhledáván pouze v kódech třídni metod uložených v metatřídách.

### 2.5.5 Polymorfismus a generický kód

**Polymorfismus** (angl. polymorphism) je vlastnost, při které různé objekty reagují různým způsobem na stejnou poslanou zprávu. V kódu, který danou zprávu posílá potom není třeba rozlišovat mezi různými typy objektů, kterým může být zpráva posílána, protože o výběru ke zprávě vhodné metody rozhodují objekty, kterým je zpráva posílána, a ne programátor.

Pro příklad nám může posloužit zpráva `#displayOn:at:`, která způsobí provedení takové metody, jenž zobrazí příjemce zprávy. Prvním parametrem této zprávy je objekt třídy `GraphicsContext`, tj. objekt, ve kterém se naše zobrazení uskuteční, a druhým parametrem je relativní souřadnice, která určuje polohu levého horního rohu požadovaného zobrazení. Prakticky to znamená, že jestliže budeme chtít zobrazit řetězec znaků `'hello world!'` na část obrazovky, které přísluší objekt `subWindowA`, do levého horního rohu, je třeba k tomu použít výraz

```
'hello world!' displayOn: subWindowA at: Point zero.
```

Výhoda polymorfismu zde spočívá v tom, že zobrazovaný objekt může být uschován v nějaké proměnné, např. `obj1`. Potom bude náš výraz vypadat

```
obj1 displayOn: subWindowA at: Point zero.
```


<sup>a</sup> Mezi systémem tříd a metatříd, které jsou si navzájem symetrické (což je využito i v brouserch `Smalltalku`), však v implementaci systému existuje určitá spojitost; za prvé platí, že každá metatřída je instancí třídy `Metaclass` a za druhé platí, že třída `Class` je nadtrídou třídy `Object class`. Toto propojení dědičnosti mezi systémem metatříd a systémem tříd způsobuje, že 1) **všechny objekty** (instance, třídy i metatřídy) v systému `Smalltalku` dědí vlastnosti definované pro instance třídy `Object`, 2) **všechny třídy** dědí vlastnosti definované pro instance třídy `Class`, 3) **všechny metatřídy** mají vlastnosti definované pro instance třídy `Metaclass` a 4) **všechny společné vlastnosti tříd a metatříd** jsou děděny jako vlastnosti instancí třídy `ClassDescription`.

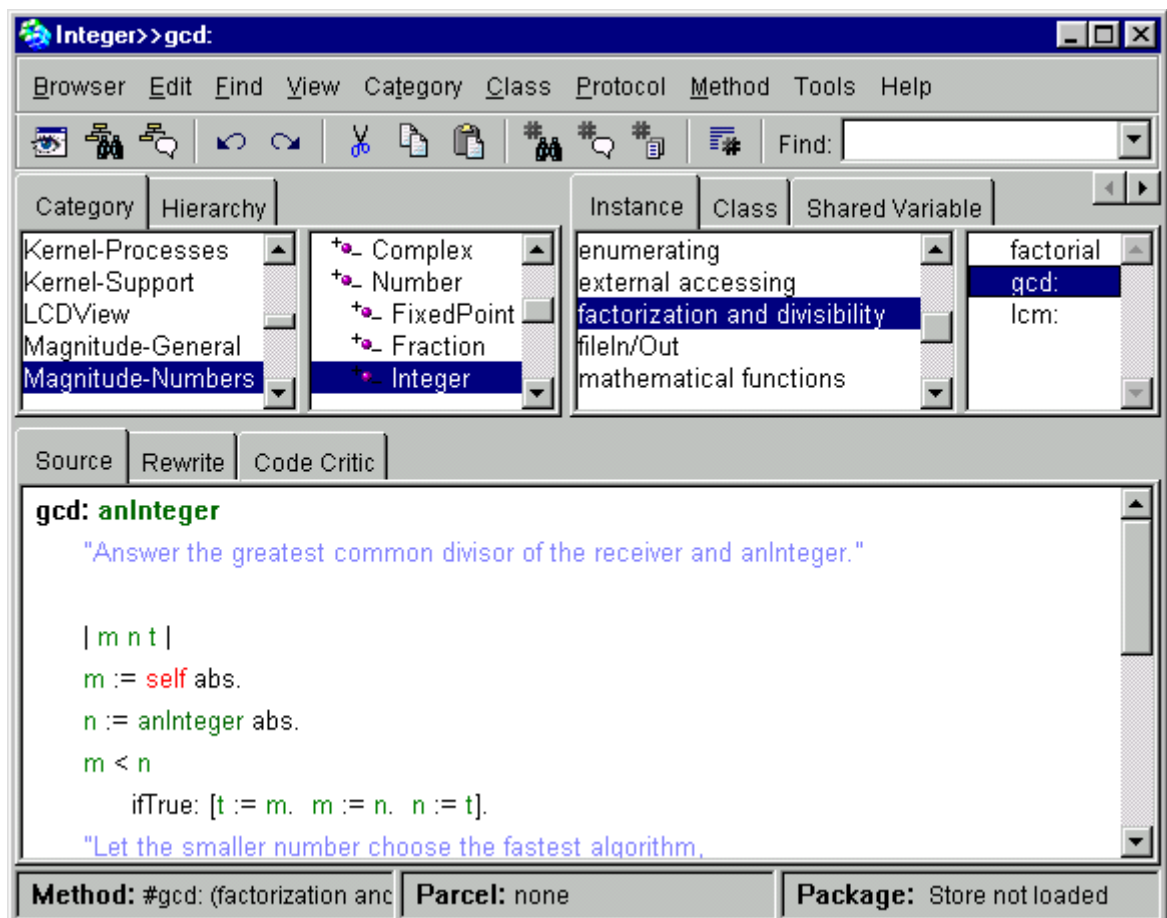
Takto námi napsaný kód bude neměnný, i když někde v předcházející části bude proměnné `obj1` přiřazen objekt jiné třídy než je `String`. Může to být objekt třídy `Image` nebo `Number` nebo kterýkoliv jiný, pokud je v jeho třídě (nebo v některé nadtřídě) implementován kód metody, která reaguje na zprávu `#displayOn:at:`. Konkrétní podoba tohoto kódu metody, která je velmi závislá na typu objektu, není ale důležitá pro program, který ji používá. Při využití polymorfního kódu dochází k velmi výhodnému utajení nepotřebných vnitřních detailů implementace, což vede k programování na vyšší úrovni abstrakce ve srovnání s konvenčním procedurálním stylem.

Dalším neméně důležitým a také příjemným důsledkem polymorfismu jsou **generické** instrukce nebo i celé programy. K vysvětlení pojmu použijeme stejný příklad. Představme si nyní, že máme již hotový program (nebo část programu), který využívá polymorfní zprávu `#displayOn:at:` pro zobrazování příslušných objektů. Po odladění nebo až během užívání tohoto kódu však dojde k tomu, že je vlivem nějakých okolností vytvořena nová třída objektů, se kterými by náš již hotový program také měl pracovat a tedy je i zobrazovat. Pro procedurálního programátora tato situace vždy znamená spoustu nepříjemností spojených se změnami v celém programu. Ve Smalltalku však stačí u nově vzniklé třídy objektů implementovat metodu, která reaguje na zprávu `#displayOn:at:`, a novou třídu s touto metodou k již existujícímu programu přidat. Generický kód je tedy takový kód, který bez nutnosti změn (tj. přepsání a znovupřeložení) za pomoci polymorfismu nabývá různých vlastností podle nových vnějších souvislostí.

## 2.5.6 Programování nových tříd a metod

Pro programování ve Smalltalku existuje několik nástrojů, z nichž jedna má hlavní postavení, neboť nám zpřístupňuje všechny zdrojové kódy celého systému. Tato utilita se jmenuje `System Browser`, a

spouští se pomocí volby `Browse -> System` z menu `VisualLauncher` nebo tlačítkem :



System Browser je rozdělen na čtyři horní části a jednu dolní. V první části nahoře zleva se objevují **kategorie tříd**, ve druhé části zleva **třídy** nebo **metatřídy** (podle nastavení přepínače `instance/class`), ve třetí části zleva (také podle stejného přepínače) kategorie třídních nebo instančních metod - tzv. **protokoly** a v poslední části nahoře **názvy metod**. V dolní části okna se objevují **zdrojové kódy**. Kategorie tříd a metod slouží pouze k lepšímu uspořádání zdrojového kódu. Za běhu programu nezáleží na tom, do které kategorie třída nebo metoda patří, a výpočet je závislý pouze na dědičnosti mezi třídami a na tom, jestli hledaná metoda je ve třídě implementovaná či nikoliv. Některé jednodušší verze Smalltalku, jako například Smalltalk/V, proto nepodporují členění zdrojového kódu do kategorií.

Naším první úkolem bude doplnění metody do existující třídy. Naprogramujeme si novou instanční metodu `#fact` do třídy `Integer`, která bude počítat faktoriál celého čísla.

Nejprve je třeba v systému nalézt třídu `Integer`. K tomu nám poslouží menu, které náleží kategoriím tříd. Dále je třeba nastavit záložky `instance/class/...` na volbu `instance`, protože budeme doplňovat instanční metodu.

Nyní je již třeba jen vybrat si vhodný protokol a doplnit do něj kód metody. Po zvolení si protokolu se nám v dolní části objeví předloha pro kód metody, která nám ukazuje, že nejprve je třeba napsat tzv. hlavičku metody, která obsahuje informaci o tom, na jakou zprávu bude daná metoda reagovat, po ní následuje komentář v uvozovkách, poté deklarace pomocných proměnných a nakonec vlastní kód metody:

```
message selector and argument names
  "comment stating purpose of message"

  | temporary variable names |
  statements
```

Napsaný kód se překládá a začleňuje do systému volbou `accept` v menu pravého tlačítka myši. Ačkoli je `accept` téměř okamžitý, při psaní programů ve Smalltalku je zdrojový kód po částech ihned překládán, a je možné části programů průběžně ladit ihned po jejich napsání.

Při psaní kódu je rovněž možné využít automatické zarovnávání (volba `format` v menu).

Systém Smalltalku standardně obsahuje množství operací, mezi nimiž je také i `#factorial`. Ze cvičných důvodů však tuto metodu naprogramujeme znovu pod jiným jménem `#fact`, protože je velmi obtížné najít jiný jednoduchý příklad, který by ve Smalltalku nebyl hotov. Naše nová metoda může být součástí jiného protokolu (např. `arithmetic`). Příslušný kód, ve kterém bylo využito rekurze, je následující.

```
fact
  "výpočet faktoriálu"

  self = 0 ifTrue: [^1]
         ifFalse: [^ self * (self - 1)fact]
```

V kódu je použita speciální proměnná `self`. Připomeňme si znovu, že proměnná `self` spolu s další speciální proměnnou `super` označuje ten objekt, součástí jehož rozhraní je kód, ve kterém je tato proměnná použita. Tyto speciální proměnné tedy v kódech metod označují **příjemce zpráv**. V našem případě jestliže vyhodnotíme výraz `23 fact`, tak proměnná `self` v kódu metody `#fact` bude mít hodnotu `23`.

Proměnné `self` a `super`, i když označují tentýž objekt, se liší v tom, jak reagují na poslané zprávy. Proměnná `self` na poslanou zprávu reaguje hledáním příslušné metody ze svého rozhraní uschovaného ve své třídě, tedy tak, jak to známe. Proměnná `super` však hned začíná odpovídající metodu hledat ve své nadtřídě, takže se přeskakuje možnost najít metodu ve vlastní třídě. Proměnná `super` se tedy

používá tehdy, když potřebujeme využít kód metody z nadtřídy, který je ostíněný kódem stejnojmenné metody v naší třídě.

Druhým příkladem bude naprogramování nové třídy objektů, která implementuje datový typ fronta (Queue). Třída `OrderedCollection` je standardní součástí systému. Objekty této třídy obsahují ve svých metodách dostatečné množství operací potřebných pro vytvoření naší fronty, a proto je použijeme pro implementaci naší fronty. Zprávy posílané objektům třídy `Queue` budou zpracovávány pouze metodami z rozhraní celého objektu (pána), přičemž v jejich kódech bude využito vlastností již v systému již obsaženého rozhraní vnitřního objektu (sluhy). Konkrétně budou využity metody `#isEmpty` (test, zda je sada prázdná), `#size` (zjištění velikosti sady), `#removeFirst` (vyjmutí prvního objektu ze začátku sady) a `#addLast`: (zařazení dalšího objektu na konec sady).

Pro naprogramování lze použít opět `system browser`. (přehledem ostatních nástrojů se budeme zabývat v některé z pozdějších lekcí). Pro tvorbu nové třídy je také vhodné do systému zařadit novou kategorii, kterou jsme pojmenovali `Examples`. Po vytvoření této nové kategorie se v dolní části `system browseru` objeví šablona-předloha kódu<sup>6</sup> pro deklaraci třídy:

```
Core defineClass: #NameOfClass
  superclass: #{NameOfSuperclass}
  indexedType: #none
  private: false
  instanceVariableNames: 'instVarName1 instVarName2'
  classInstanceVariableNames: ''
  imports: ''
  category: 'Examples'
```

a vlastní kód nové třídy je následující:

*Queue (deklarace třídy a instančních metod)*<sup>a</sup>

```
Core defineClass: #Queue
  superclass: #{OrderedCollection}
  indexedType: #none
  private: false
  instanceVariableNames: ''
  classInstanceVariableNames: ''
  imports: ''
  category: 'Examples'

Queue methodsFor: 'removing' (protokol 'removing')
next
  "odebere a vrati objekt z fronty"
  ^self removeFirst

Queue methodsFor: 'adding' (protokol 'adding')
add: anObject
  "prida objekt do fronty"
  self addLast: anObject
```

Metody `#size` a `#isEmpty` jsou do instančních metod třídy `Queue` děděny. Jak je vidět, tak implementace nové třídy objektů je jednoduchá a elegantní. Má však jednu značnou nevýhodu. Instance třídy `Queue` totiž dědí od třídy `OrderedCollection` kromě metod i všechny ostatní metody třídy `OrderedCollection`, tedy i například `#addFirst`, `#removeLast`, což by mohlo způsobit například při týmové tvorbě programů nedovolené zacházení s nově naprogramovaným objektem.

<sup>6</sup> Za povšimnutí stojí upozornit na to, že tato předloha je vlastně posláním osmiparametrové zprávy objektu `NameSpace`. Třidu lze samozřejmě vytvořit volbou z menu druhého tlačítka.

<sup>a</sup> Všimněte si, že nová třída ve `Smalltalku` vzniká posláním zprávy `#subclass:instanceVariableNames:category:` nějaké jiné třídě, která potom bude nově třídě nadtřídou. Přidávání metod do systému je také implementováno posláním příslušných zpráv.

Jednou z možností je nežádoucí metody také implementovat, aby se zabránilo jejich dědění, podle následujícího příkladu:

```
Queue methodsFor: 'error handling'  
addFirst:  
    "zakaz predbihani ve fronte"  
    ^self error: 'unappropriate operation for queue'
```

Nyní použijeme jinou variantu, která spočívá v tom, že každá instance naší nové třídy bude v sobě obsahovat (vztah celek - část) jeden objekt se jménem `buffer` třídy `OrderedCollection` (uspořádaná sada). Tato implementace je bezpečná v tom, že všechny ostatní nepoužité metody z rozhraní vnitřního objektu jsou navenek **zakryté**, protože o práci s celým objektem rozhoduje pouze jeho vlastní rozhraní bez ohledu na vlastnosti rozhraní v něm uschovaných objektů (částí). Máme tedy zaručeno, že s objekty třídy `Queue` bude zacházeno pouze předepsaným způsobem, a že s těmito objekty nebude možné provádět pro frontu nepřípustné operace (např. přímý vstup doprostřed fronty, předbíhání ve frontě apod.), i když tyto operace jsou v možnostech objektů třídy `OrderedCollection`, kterou jsme v programu použili.

*Queue (deklarace třídy a instančních metod)*

```
Core defineClass: #Queue  
    superclass: #{Object}  
    indexedType: #none  
    private: false  
    instanceVariableNames: 'buffer'  
    classInstanceVariableNames: ''  
    imports: ''  
    category: 'Examples'  
  
Queue methodsFor: 'testing' ( tj. protokol 'testing')  
isEmpty  
    "vrati true nebo false podle obsahu fronty"  
    ^buffer isEmpty  
  
size  
    "vrati pocet objektu ve fronte"  
    ^buffer size  
  
Queue methodsFor: 'removing' (protokol 'removing')  
next  
    "vysledkem je odebrany objekt z fronty"  
    ^buffer removeFirst  
  
Queue methodsFor: 'adding' (protokol 'adding')  
add: anObject  
    "prida objekt do fronty"  
    buffer addLast: anObject  
  
Queue methodsFor: 'instance initialization' (prot. 'instance  
    initialization')  
initialize  
    "pripravi instanci k pouzivani"  
    buffer := OrderedCollection new  
  
Queue class methodsFor: 'instance creation' (protokol 'instance  
    creation')  
new  
    "tvorba nove instance"  
    ^super new initialize
```

V kódu je použita speciální proměnná `super`, které je v kódu metody `#new` posílána zpráva `#new`. Tento trik umožňuje v kódu metody využít kód stejnojmenné metody z nadtřídy. Implementace nové metody `#new` potom spočívá v **doplnění původního** (děděného, ale odstíněného) kódu metody `#new` o posláni

zprávy #initialize, která nám vytvořenou instanci nastaví do počátečního stavu. Uživatel (i jiný programátor) našeho programu nemusí znát detaily naší implementace a pro tvorbu nových instancí naší třídy Queue může používat standardní zprávu #new.

Není-li kód metody ukončen návratovým výrazem, výsledkem se stává celý objekt příjemce. (což je stejné, jako kdyby byl na konci kódu navíc návratový výraz ^self) Pro lepší pochopení si ještě uveďme snáze pochopitelnou verzi této metody:

```
Queue class methodsFor: 'instance creation' (protokol 'instance
  creation')
new
  "tvorba nove instance"
  | newObject |
  newObject := super new. ... zde využíváme děděnou metodu
  newObject initialize.
  ^newObject
```

Program si vyzkoušíme v okně Workspace. Nejprve si vytvoříme jednu instanci třídy Queue, například se jménem Frontal pomocí výrazu Frontal := Queue new. Protože jméno začíná velkým písmenem, tak se jedná o globální proměnnou. S takto vytvořeným objektem můžeme potom experimentovat například následujícím způsobem:

výraz	výsledek
Frontal add: 'Jan'	Frontal
Frontal add: 'Pavel'	Frontal
Frontal size	2
Frontal add: 'Petr'	Frontal
Frontal size	3
Frontal next	'Jan'
Frontal size	2
Frontal next	'Pavel'
Frontal isEmpty	false
Frontal next	'Petr'
Frontal isEmpty	true

## 2.6 Systém tříd ve Smalltalku

Bez alespoň základních znalostí systému tříd programátor nemůže používat jednu z největších výhod, které přináší OOP, a kterou je znovupoužitelnost kódu a programování ve stylu "by exception", kdy se v programech píše jen ten kód, který implementuje pouze nové vlastnosti nebo odlišnosti nových objektů od již existujících objektů. V případě dobrého zvládnutí systému tříd Smalltalku je možné dosahovat až 70% znovupoužitelnosti (tj. využitelnosti existujícího) kódu. U komerčně orientovaných aplikací pro zákazníky z oblasti např. obchodu a financí pokročilí programátoři ve Smalltalku dokonce dosahují i více než 90% znovupoužitelnosti, což jim umožňuje v některých případech i tvorbu softwarových prototypů ve velmi krátkém čase.

Při studiu následujících kapitol doporučujeme seznámit se s probíranými třídami také přímo v systémovém pořadači a orientačně si prohlédnout jejich další možnosti, na které se v textu z důvodu místa již nedostane.

### 2.6.1 Systém tříd Collection

Tento systém obsahuje implementace objektů, které bychom mohli česky pojmenovat jako sady, kolekce, sbírky nebo množiny dat. Jedná se totiž o nejrůznější strukturované datové typy (pole, množiny, fronty ...) sloužící k uchovávání jiných objektů a k jejich manipulaci. Pojem množina, i když je někdy používán, je v tomto kontextu nepřesný, neboť jen jeden datový typ (třída Set) z toho systému přesně odpovídá matematickému pojetí množiny.

Celkový počet takových tříd se v základní sadě Smalltalku pohybuje okolo 120 (což si lze ověřit výpočtem `Collection allSubclasses size`).

Kolekce jsou jedním z pilířů programování ve Smalltalku. Jakmile se programátor naučí využívat jejich možností, pocítí obrovský nárůst produktivity<sup>7</sup>.

### **Přehled vybraných tříd systému Collection. V závorkách je uvedena struktura instancí jednotlivých tříd (instanční proměnné).**

```
Collection
  SequenceableCollection
    OrderedCollection(firstIndex lastIndex)
    FontDescriptionBundle
    SortedCollection(sortBlock)
      SortedCollectionWithPolicy(sortPolicy)
    LinkedOrderedCollection(backup)
  LinkedList(firstLink lastLink)
  Semaphore(excessSignals)
  HandlerList
  ArrayedCollection
    RunArray(runs values cacheRun cacheRunStart)
    List(dependents collection limit collectionSize)
      DependentList
      TreeModel(root childrenBlock)
  WeakArray(dependents)
  Array
    ScannerTable(value0 defaultValue letterValue digitValue separatorValue)
    DependentsCollection
    SegmentedCollection(compressed)
      LargeWordArray
      CharacterTable
      LargeArray
    ActionSequence
    BOSSReaderMap(baseIndex storage)
  CharacterArray
  Text(string runs)
  String
    ByteEncodedString
    OS2String
    ISO8859L1String
    MacString
    ByteString
    MSCP1252String
    TwoByteString
  Symbol
    TwoByteSymbol
    ByteSymbol
    FourByteSymbol
  GapString(string gapStart gapSize)
  FourByteString
  IntegerArray
  WordArray
  ByteArray
    BinaryStorageBytes
    BOSSBytes
  DwordArray
  TableAdaptor(dependents baseCollection adaptors columnSize transposed)
  TwoDList(dependents collection rows columns transposed)
  Interval(start stop step)
```

---

<sup>7</sup>Viz “Příklad použití metod z knihovny `Collection - asSet`”

```

KeyedCollection
  Palette
    CoveragePalette(maxPixelValue)
    ColorPalette
      FixedPalette(redShift redMask greenShift greenMask blueShift blueMask)
      MappedPalette(hasColor palette inverseMap mapResolution size)
      MonoMappedPalette(whitePixel blackPixel)
    ColorPreferencesCollection
      ColorPreferencesDictionary(preferences constantCodeArray)
      ChainedColorPreferences(base node)
      LookPreferences(foregroundColor backgroundColor selectionForegroundColor
                      selectionBackgroundColor borderColor hiliteColor shadowColor)
    GeneralNameSpace(organization bindings specificImports generalImports)
      NameSpaceOfClass(owner)
      NameSpace(parent name)
      WorkspaceVariablePool(workspace)
    MethodDictionary
Bag(contents)
Set(tally)
Dictionary
  IdentityDictionary(valueArray)
  WeakDictionary(executors accessLock)
  HandleRegistry
  ExternalRegistry
  WeakAssociationDictionary
  LinkedWeakAssociationDictionary(backup)
  ExternalDictionary
  PropertyListDictionary(basicSize)
CEnvironment
IdentitySet
  ObjectRegistry
  ExceptionSet
  SignalCollection
NameSpaceBindings
WeakNameSpaceBindings(slots)
OrderedSet(values comparisonBlock)

```

Vzhledem k tomu, že počet tříd je značný, tak vybíráme jen několik pro základní práci se Smalltalkem nejdůležitějších tříd. Následující tabulka tedy přináší podrobnější informaci o některých třídách objektů tohoto systému.

<i>název třídy</i>	<i>tvorba instance</i>	<i>typ elementů</i>	<i>uspořádání</i>	<i>přístup</i>
Bag	new	Object	není	hodnotou
Array	new: aNumber	Object	1 .. size	indexem
String	new: aNumber	Character	1 .. size	indexem
Text	aString asText	Character	1 .. size	indexem
ByteArray	new: aNumber	SmallInteger	1 .. size	indexem
Interval	aValue to:by:	ArithmeticValue	1 .. size	pořadím
OrderedCollection	new	Object	first..last	pořadím
SortedCollection	new	Magnitude	pravidlem	pořadím
Set	new	Object	není	hodnotou
Dictionary	new	Object	není	klíčem

Instance tříd tohoto systému mají implementováno veliké množství metod, které není možné (a ani není cílem) uvést v této lekci. Všechny použitelné metody jsou včetně svých zdrojových kódů dosažitelné v systémovém pořadači (browseru) a v referenčním manuále Smalltalku. V systémovém pořadači je také možné u každé třídy si zobrazit komentář (volba `comment`) a nebo si nechat příslušnou metodu vysvětlit (volba `explain`). Na tomto místě pouze sdělujeme, že všechny instance tohoto systému tříd regují na zprávu `size`, která vrací počet elementů, a na různé konvertující zprávy (`asSet`, `asBag`, `asOrderedCollection`, `asArray`, `asSortedCollection`, ...).

Pro tvorbu instancí je u všech těchto uvedených tříd možné mj. použít zprávy `with:`, `with:with:`, `with:with:with:` a `with:with:with:with:`, které jsou posílány příslušným třídám.

Příklad:

<i>výraz</i>	<i>výsledek</i>
<code>ByteArray with: 12 with: 34 with: 100</code>	<code>#[12 34 100]</code>
<code>String with: \$a with: \$b with: \$c with: \$d</code>	<code>'abcd'</code>

Instance tříd `Bag` (sáček, pytlík) a `Set` (množina) obsahují svoje elementy bez vnitřního uspořádání. V instancích třídy `Bag` se na rozdíl od instancí třídy `Set` mohou jednotlivé elementy opakovat (je možný vícenásobný výskyt stejného objektu). Protože neexistuje u obou typů možnost vnitřního uspořádání elementů, tak s jejichmi elementy je možné pracovat pouze pomocí jejich hodnoty. Pro přidávání objektů se proto používá zpráva `add: anObject` a pro odebrání zpráva `remove: anObject`.

Instance tříd `Array`, `String`, `Text` a `ByteArray` patří mezi tzv. indexované objekty. Pro přístup k jejím elementům slouží již v předchozích pokračováních vysvětlené zprávy `at:` a `at:put:`. Instance třídy `Text` se od obyčejných řetězců liší tím, že každý znak může mít kromě svojí hodnoty i atributy (barva, silné, podtržené, šikmé apod. písmo). Pro tvorbu těchto instancí o předem požadované velikosti může sloužit i zpráva `new: anInteger`, která je posílána příslušné třídě.

Instance třídy `Interval` se nejčastěji vytvářejí pomocí posílání zpráv `to: aValue` nebo `to: aValue by: aStep` instancím třídy `Number`. Přístup na jednotlivé elementy umožňují například zprávy `first` a nebo `last`.

Třída `OrderedCollection` (uspořádaná sada), se kterou jsme se již seznámili, implementuje uspořádanou sadu elementů. Jednotlivé elementy se řadí za sebe v tom pořadí (na začátek nebo na konec řady), v jakém byly do sady přidávány. V minulé lekci jsme tuto třídu použili pro implementaci datového typu `Fronta` (`Queue`). Elementy je možné přidávat pomocí zpráv `add: anObject`, `addLast: anObject` a nebo `addFirst: anObject`. Ze sady lze odebírat elementy kromě tradičního `remove:` i pomocí zpráv `removeLast` nebo `removeFirst`.

Třída `SortedCollection` (setříděná sada) implementuje podobné vlastnosti jako třída `OrderedCollection`. Pořadí jednotlivých elementů však není určeno pořadím při přidávání do sady, ale vnitřním pravidlem (tzv. `sortBlock`). Není-li pravidlo uvedeno jinak, tak systém doplní pravidlo (při výrazu `SortedCollection new`) ve tvaru `[ :a :b | a < b ]`.

Příklad:

Výraz `SortedCollection sortBlock: [:x :y | x abs > y abs]` vytvoří instanci třídy `SortedCollection`, která uspořádává jednotlivé elementy sestupně podle jejich absolutních hodnot.

Poslední z uvedených tříd je třída `Dictionary`, která implementuje sady prvků, jenž jsou podobné instancím třídy `Set`. Na rozdíl od instancí třídy `Set` jsou však elementy přístupné také podle tzv. klíče (`key`). Každý element má přiřazený svůj klíč, kterým může být libovolný objekt ve `Smalltalku` (nejčastěji však řetězec nebo symbol). Přístup na jednotlivé elementy je potom umožněn právě pomocí klíčů (`at: aKey`, `at: aKey put: aValue`, `removeKey: aKey`).

### 2.6.1.1 Příklady práce s instancemi systému `Collection`

Pro procvičení vyložených vlastností tříd a metod systému `Collection` přinášíme několik příkladů, které je možné si procvičovat v okně `Workspace`.

```
X := OrderedCollection with: 12 with: 34 with: 15.
```

Je vytvořena (v globální proměnné) uspořádaná sada s prvky (12 34 15) .

```
X addLast: 56. X addFirst: 17.
```

Do této sady je na začátek přidán objekt 17 a na konec objekt 56.

```
X removeFirst.
```

Ze sady `OrderedCollection(17 12 34 15 56)` je odebrán první prvek 17.

```
 #(1 2 3 5 7) reverse.  
 'hello world' reverse.
```

Obrací pořadí prvků.

```
 |y|  
 y := #(1 2 6 5 3) asSortedCollection.  
 y add: 4.  
 ^y
```

Je vytvořena setříděná sada `y`, do které je vložen objekt 4, který není zařazen na konec sady, ale na příslušné místo podle třídícího pravidla.

```
 |y|  
 y := #(1 2 6 5 3) asSortedCollection: [:a :b | a > b].  
 y add: 4.  
 ^y
```

Je vytvořena setříděná sada, do které je vložen objekt 4. Všimněte si odlišného třídícího pravidla.

```
 |d|  
 d := Dictionary new.  
 d at: 'Jan' put: '54 23 30';  
   at: 'Honza' put: '35 44 40';  
   at: 'Milan' put: '338 2274'.  
 ^ d at: 'Honza'
```

Je vytvořena sada telefonních čísel. Každé telefonní číslo má svůj klíč, v našem případě jméno nějaké osoby. Pro nahlédnutí do struktury této sady doporučujeme poslední výraz v tomto příkladu (`^ d at: 'Honza'`) nahradit výrazem `d inspect`.

## 2.6.2 Systém tříd Magnitude

Některé třídy systému `Magnitude` již známe z kapitoly, která se celá věnovala výkladu syntaxe jazyka `Smalltalk`.

**Přehled vybraných tříd systému `Magnitude`. V závorkách je uvedena struktura instancí jednotlivých tříd (instanční proměnné).**

```
Magnitude  
  Character  
  Time(hours minutes seconds)  
  LookupKey(key)  
    Association(value)  
      AssociationTree(children)  
      WeakKeyAssociation  
      Ephemeron(manager)  
  VariableBinding(value usage category)  
    InitializedVariableBinding(method)
```

```

ArithmeticValue
  Point(x y)
  Number
    FixedPoint(numerator denominator scale)
    Fraction(numerator denominator)
    Integer
      SmallInteger
      LargeInteger
      LargeNegativeInteger
      LargePositiveInteger
    LimitedPrecisionReal
      Float
      Double
    RBNumberWithSource(number source)
    MetaNumeric
      SomeNumber(value)
      Infinitesimal(negative)
      Infinity(negative)
      NotANumber
    Complex(real imaginary)
  Date(day year)
  GeneralMethodDefinition
    ContextDefinition(context)
    MethodDefinition(inheritingClass implementingClass selector extraText)
      OverriddenMethodDefinition
      MethodInstallationRecord(method protocol instVarNames)
    InitializerDefinition(nameSpace key)
    MessageTallyDefinition(tally)
  DeferredBinding(key resolved resolvedValue method)
    ResolvedDeferredBinding
    ResolvedDeferredConstant
  NodeTag(namespace type qualifier)
  ObjectSegment(firstIndex lastIndex firstPart lastPart)
  AbsentClassImport(name isMeta format instanceVariables dummyBehavior dummyClassPool)
  Timestamp(year month day hour minute second millisecond)
  ObjectNameWrapper(fullName simpleName environmentName)
  MessageTally(class method name tally samples parent receivers)

```

Abychom doplnili základní znalost o tomto systému tříd, tak si popíšeme vlastnosti tříd `Fraction`, `Point`, `Date` a `Time`. Instance subsystému `ArithmeticValue` reagují na zprávy, z nichž mnohé představují aritmetické operace známé i z jiných programovacích jazyků. Stačí tedy jen upozornit na to, že binární zpráva `** aValue` slouží k umocňování, binární zpráva `// aValue` k celočíselnému dělení a binární zpráva `\ \ aValue` k získání zbytku po celočíselném dělení.

Instance třídy `Fraction` (zlomek) vznikají při dělení celých čísel. Pro jejich převod na desetinná čísla slouží zpráva `asFloat`. Zlomková aritmetika může v některých případech být přesnější a také rychlejší než desetinná.

Instance třídy `Point` (bod) jsou objekty, které reprezentují body v dvojrozměrné kartézské soustavě souřadnic. Nejobvyklejším způsobem pro tvorbu instance třídy `Point` je poslání binární zprávy `@ aValue` nějaké číselné hodnotě. (Například `4 @ 5` je bod o souřadnicích  $x = 4$  a  $y = 5$ ). Body, které patří do systému `ArithmeticValue`, reprezentují také vektory, a je tedy možné je mezi sebou vzájemně například porovnávat, sčítat a nebo odčítat, což je velmi využíváno při zobrazování v grafickém rozhraní, které `Smalltalk` používá. (Na čtenáři necháme rozmyšlení o tom, jak asi vypadá, a jak je ve `Smalltalku` složitá implementace komplexních čísel.)

`Date` a `Time` jsou třídy, které implementují datum a čas. Protože tyto objekty také patří do systému `Magnitude`, tak mají například implementovány metody pro vzájemné porovnávání, pro přičítání a odečítání. Instanci reprezentující aktuální systémové datum získáme posláním zprávy `today` třídě `Date`. Instanci reprezentující aktuální systémový čas získáme posláním zprávy `now` třídě `Time`.

Následný příklad práce s instancemi třídy `Date` obsahuje výpočet poměru délky vašeho manželství (nejste-li svobodní či rozvedeni) k délce vašeho života.

```
|birthDate wedDate|

birthDate :=
    Date readFrom: (ReadStream on:
        (DialogView request: 'Enter your birthdate...')).
wedDate :=
    Date readFrom: (ReadStream on:
        (DialogView request: 'Enter your wedding date...')).
^ (Date today subtractDate: wedDate) / (Date today subtractDate: birthDate)
    roundTo: 0.01
```

### 2.6.3 Systém tříd Stream

Posledním systémem tříd, který patří do základu programování ve Smalltalku, je systém `Stream`. Pod pojmem `Stream` si můžeme představit zařízení, které zprostředkovává sekvenční přístup pro čtení nebo psaní do nějaké sady, kterou může být kromě běžné sady, jak jsme je popsali v této lekci (`Collection`), i soubory na disku nebo obsah textových oken apod.

#### **Přehled vybraných tříd systému Stream. V závorkách je uvedena struktura instancí jednotlivých tříd (instanční proměnné).**

```
Stream
  PeekableStream
  PositionableStream(collection position readLimit writeLimit policy)
  ExternalStream
    BufferedExternalStream(lineEndCharacter binary lineEndConvention bufferSize ioBuffer ioConnection)
    ExternalReadStream
      ExternalReadAppendStream(writeStream)
      ExternalReadWriteStream
      CodeReaderStream(swap isBigEndianPlatform scratchBuffer format)
    ExternalWriteStream
  InternalStream
    WriteStream
      TextStream(lengths emphases currentEmphasis runStartPosition)
      ReadWriteStream
        ByteCodeReadWriteStream(noPeekPosition)
        InternalCodeWriterStream(scratchBuffer)
    ReadStream
      InternalCodeReaderStream(swap isBigEndianPlatform scratchBuffer format)
    EncodedStream(binary stream encoder policy lineEndConvention lineEndCharacter skipSize)
  Random(seed)
    MinimumStandardRandom(a m q r)
    FastRandom(increment modulus fmodulus multiplier)
  RBScanner(stream buffer tokenStart currentCharacter characterType classificationTable
    numberType separatorsInLiterals extendedLiterals comments errorBlock nameSpaceCharacter)
  RBPatternScanner
```

Implementovaných metod v tomto systému je také veliké množství, a proto z nich vybereme jen některé nejdůležitější, neboť předpokládáme, že si programátoři zbytek najdou v referenčním manuále nebo v systémovém pořadači.

Pro všechny instance tohoto systému jsou implementovány metody pro čtení:

```
next                                přečte jeden objekt
next: aNumber                       přečte aNumber objektů
```

Pro zápis:

<code>nextPut: anObject</code>	zapiše jeden objekt
<code>nextPutAll: aCollection</code>	zapiše několik objektů v sadě <code>aCollection</code> .

Pro testování a řízení:

<code>atEnd</code>	test konce
<code>skip: anInteger</code>	skočí dopředu či dozadu o <code>anInteger</code> pozic
<code>contents</code>	vrátí obsah

Pro některé speciální typy, jakým je například `TextStream`, jsou implementovány další metody, z nich si uveďme například `nextLine`, `nextLinePut:`, `skipSeparators`, `nextChunk`, `nextChunkPut:` apod. (tzv. chunk je blok textu, který může být několikařádkový, oddělený oddělovačem `$!`. A nebo to je jeden oddíl v XML. Tento formát je využíván například pro uchování zdrojových kódů metod).

### 2.6.3.1 Příklady použití

První příklad představuje jednoduchý automat, který transformuje řetězec textu. Transformace spočívá v tom, že všechny znaky `$@` jsou nahrazeny řetězcem s dnešním datem.

```
|in out char|
in := ReadStream on: 'Today is @. Today must be @!'.
out := WriteStream on: String new.

[in atEnd]
whileFalse:
    [(char := in next) = @$@
     ifTrue: [out nextPutAll: Date today printString]
     ifFalse: [out nextPut: char]
    ].
^out contents
```

Druhý příklad ukazuje rozkódování pole bytů, které představuje kompresovanou informaci. Algoritmus komprese (a dekomprese) spočívá v tom, že opakující se řada stejných bytů je uložena v podobě dvou bajtů, kde první bajt značí počet opakování zvětšený o 200 a druhý bajt číslo, které se opakuje. Číslo větší než 200, i když se neopakují, tak jsou uložena ve dvou bajtech (aby nedošlo k záměně s kódem pro opakování), kde první bajt má hodnotu 201 (jakoby jedno opakování) a druhý bajt hodnotu příslušného čísla. Tento algoritmus komprese je téměř shodný s kompresí obrázků ve formátu PCX.

```
|compr decompr a b |
compr := ReadStream on: #[161 207 37 201 220 78].
decompr := WriteStream on: ByteArray new.

[compr atEnd]
whileFalse:
    [(a := compr next) > 200
     ifTrue: [b := compr next.
              (a - 200) timesRepeat: [decompr nextPut: b]]
     ifFalse: [decompr nextPut: a]
    ].
^decompr contents
```

## 2.7 Řízení výpočtu ve Smalltalku

V jazyce Smalltalku neexistují speciální konstrukty, které by byly určeny k vyjádření řízení výpočtu. Struktury pro řízení výpočtu jsou ve Smalltalku realizovány také pomocí posílání zpráv různým objektům. Z tohoto důvodu jsme již některé z nich používali dříve, aniž bychom je nějak zvlášť popisovali. Na tomto místě se budeme jimi podrobněji zabývat.

### 2.7.1 Větvení

Ve Smalltalku neexistuje příkaz skoku. Větvení výpočtu v závislosti na nějaké logické podmínce (rozuměj v závislosti na booleovském objektu o hodnotě `true` nebo `false`) je realizováno pomocí zpráv, které jsou posílány těmto booleovským objektům. Jsou to čtyři zprávy:

```
ifTrue: aTrueBlock
ifFalse: aFalseBlock
ifTrue: aTrueBlock ifFalse: aFalseBlock
ifFalse: aFalseBlock ifTrue: aTrueBlock
```

Parametry `aTrueBlock` a `aFalseBlock` jsou bloky výrazů, které jsou vyhodnocovány v závislosti na logické hodnotě příjemce zprávy (`true` či `false`).

Příklad:

```
(a < b) ifTrue: [c := 100 + a] ifFalse: [c := 200 + b]
```

protože však ve Smalltalku je výsledkem volání zprávy vždy nějaký objekt, tak výše uvedený příklad je také možné napsat úsporněji jako:

```
c := (a < b) ifTrue: [100 + a] ifFalse: [200 + b]
```

*Poznámka:*

Logický součin je implementován v metodách instancí booleovských objektů `& aBooleanObject` nebo `and: aBooleanBlock`.

Logický součet je implementován v metodách `| aBooleanObject` nebo `or: aBooleanBlock`.

Jejich rozdíl spočívá v tom, že u metod `& aBooleanObject` a `| aBooleanObject` v případě, že parametr `aBooleanObject` je logický výraz, tak dochází vždy k jeho vyhodnocování bez ohledu na hodnotu příjemce zprávy. U metod `and: aBooleanBlock` a `or: aBooleanBlock` nedochází k vyhodnocování parametru `aBooleanBlock` v případě, že již samotná logická hodnota příjemce jednoznačně rozhoduje o celém výsledku.

Příklad:

- a) `(aFileStream atEnd) or: [aFileStream next = $*]`  
tento výraz v případě, že platí první podmínka (`aFileStream atEnd`), tak již **nevyhodnocuje kód bloku** druhé podmínky a vrací hodnotu `true`.
- b) `(aFileStream atEnd) | (aFileStream next = $*)`  
Tento výraz před tím, než bude volat zprávu `|` prvnímu výrazu, tak **musí oba dva výrazy vyhodnotit**. Náš příklad záměrně ukazuje, že v tomto příkladu není možné beztrestně nahradit zprávu `or: zprávaou |`, protože v případě, že první podmínka je splněna, tak není možné vyhodnotit druhou podmínku, neboť z daného souboru nelze číst, protože je již na konci. Příklad b) tedy může vést k chybě za běhu programu.

## 2.7.2 Iterace

Pro **jednoduchou iteraci** slouží zpráva `timesRepeat: aRepeatBlock`, která se posílá celým číslem. Příklad:

```
10 timesRepeat: [Transcript show: 'hello world!'; cr]
```

Pro **iteraci**, při které jsou zadány meze a eventuelně i krok slouží zprávy `to: aNumber do: aBlock` nebo `to: aNumber by: aStep do: aBlock`. Příklad:

```
1 to: 10 by: 0.5 do: [:i | Transcript show: i printString; cr]
```

**Podmíněný cyklus** je ve Smalltalku nahrazen zprávami `whileTrue`, `whileFalse`, `whileTrue: aTrueBlock` a `whileFalse: aFalseBlock`, které jsou posílány instancím třídy `BlockClosure` (blok). Jejich použití je názorné z následných ukázek:

### 1. cykly typu "while-do"

```
[myFile atEnd] whileFalse: [Transcript show: myFile next].  
[i > 0] whileTrue: [x := x + i. i := i - 1].
```

### 2. cyklus typu "repeat-until" `x := x + i. (i := i - 1) > 0] whileTrue`

Bloku je též možné zaslat unární zprávu `repeat`, která způsobí nekonečné opakování. Blok potom musí obsahovat vlastní přerušení běhu.

## 2.7.3 Operace nad sadami (Collection)

Objekty, které jsou instancemi systému tříd `Collection`, reagují i na takové zprávy, které reprezentují ve srovnání s konvenčními programovacími jazyky (Pascal, C) velmi komplexní operace. Tyto operace v těchto programovacích jazycích si musejí programátoři implementovat ve svých programech vesměs pomocí nejrůznějších podmíněných cyklů, a proto je uvádíme spolu s ostatními zprávami využívanými ve Smalltalku k řízení výpočtu. Jsou to především následující zprávy:

Zpráva `do: aBlock` provede daný blok pro všechny prvky sady.

Příklad:

```
 #(12 34 56 78) do: [:i | Transcript show: i printString; cr].  
 ((1 to: 10), #(12 15)) do: [:n | Transcript show: n printString; cr].
```

Zpráva `select: aBlock` vrací novou sadu, ve které jsou jen ty prvky z příjemce, které splňují podmínku danou v bloku `aBlock`.

Příklad:

```
 #(1 5 2 4 3) select: [:x | x > 2]   vrací jako výsledek #(5 4 3).  
 'hello world' select: [:ch | ch isVowel]   vrací jako výsledek 'eoo' .
```

Zpráva `reject: aBlock` filtruje příjemce zprávy opačným způsobem než zpráva předchozí. Výsledkem je sada, ze které jsou odebrány ty prvky z příjemce, které splňují podmínku v bloku `aBlock`.

Příklad:

```
'hello world' reject: [:ch | ch isVowel]   vrací jako výsledek 'hll wrld'.
```

Zpráva `collect: aBlock` vrací novou sadu, ve které všechny prvky prošly transformací bloku `aBlock`.

Příklad:

```
#(1 5 2 4 3) collect: [:x | x + 10] vrací jako výsledek #(11 15 12 14 13).
```

Zpráva `detect: aBlock` vybírá podobně jako zpráva `select:` prvky vyhovující dané podmínce v bloku. Na rozdíl od zprávy `select:` však nevrací podmnožinu všech prvků, které prošly testem, ale pouze první prvek, který splňuje podmínku testu.

Příklad:

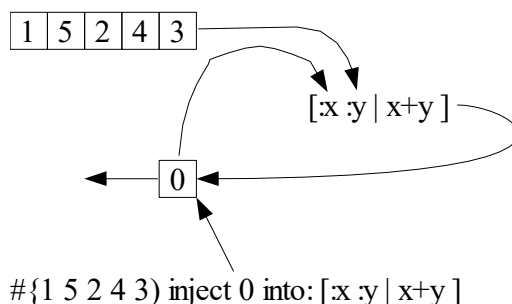
```
#(1 5 2 4 3) detect: [:x | x > 2] vrací jako výsledek 5.
```

Poslední zprávou, kterou jsme vybrali je zpráva `inject: aValue into: aBlock`. Tato zpráva umožňuje provést se všemi prvky sady nějakou kumulativní operaci. Hodnota `aValue` slouží k inicializaci příslušné operace.

Příklad:

```
#(1 5 2 4 3) inject: 0 into: [:x :y | x + y] vrací jako výsledek 15 (součet).  
#(1 5 2 4 3) inject: 1 into: [:x :y | x * y] vrací jako výsledek 120 (násobek)."
```

Způsob provádění ilustruje následující obrázek.



### 2.7.3.1 Příklad použití metod z knihovny Collection - asSet

Následující úloha je velmi zajímavým příkladem nekonvenčního použití metod systému `Collection`. Představme si, že máme za úkol spočítat, kolik různých znaků obsahuje daný řetězec textu. Pouze s povrchní znalostí vlastností systému `Collection` se jedná o poměrně komplikovaný algoritmus, který může vypadat například následovně:

```
| aString n tempArray |  
  
String := Dialog request: 'Enter your text:'.  
tempArray := Array new: aString size withAll: 1.  
1 to: aString size - 1 do:  
  [:i | i + 1 to: aString size do:  
    [:j | (aString at: i)=(aString at: j) ifTrue: [tempArray at: j put: 0]].  
  n := 0. 1 to: aString size do: [:i | n := n + (tempArray at: i)].  
  ^n
```

S využitím vlastností třídy `Set` lze však tento úkol vyřešit mnohem elegantnějším způsobem:

```
^(Dialog request: 'Enter your text:') asSet size
```

### 2.7.3.2 Příklad použití metod z knihovny Collection - asBag, occurrencesOf:

Tento příklad přímo navazuje na předchozí. Tentokrát je zadání ještě složitější v tom, že potřebujeme znát kolikrát se každý znak ve vstupním řetězci vyskytuje. „Klasické“ řešení pro jeho obtížnost nebudeme raději ani ukazovat. Řešení využívající vlastnosti systému `Collection` je následující:

```
| aString result |
aString := Dialog request: 'Enter your text:'.
result := Dictionary new.
aString asSet do:
    [:char | result at: char put: (aString asBag occurrencesOf: char)].
^result
```

Výraz `aString asBag` je však součástí bloku ve zprávě `do:`, což u zpracování rozsáhlejších datových struktur může způsobit i vážná zpomalení výpočtu a profesionální programátor by se měl takovýchto chyb vyvarovat a vždy pro hodnoty, které se vícekrát používají, použít pomocnou proměnnou:

```
| aString result aBagString |
aString := Dialog request: 'Enter your text:'.
result := Dictionary new.
aBagString := aString asBag.
aString asSet do:
    [:char | result at: char put: (aBagString occurrencesOf: char)].
^result
```

### 2.7.3.3 Příklad použití metod z knihovny Collection - select:, includes:

Tento příklad přináší ukázkou jednoduché implementace průniku dvou sad A a B:

```
^ A select: [:x | B includes: x]
```

### 2.7.3.4 Příklad použití metod z knihovny Collection - inject:into:

Jednou ze zajímavých možností, jak využít metodu `inject:into:` ukazuje následující příklad, který lze považovat za návrhový vzor<sup>8</sup>:

V programech občas potřebujeme například najít maximum z nějaké sady prvků. Pokud si představíme, že diskutovaná sada obsahuje pouze nezáporná čísla a je pojmenovaná například `myNumbers`, tak kód, který vyhledává maximální prvek může vypadat následovně:

```
| result x |
result := 0.
1 to: myNumbers size do:
    [:i | x := myNumbers at: i.
    x > result ifTrue: [result := x] ].
^result
```

S využitím metody `do:` a metody `max:` lze tento kód zjednodušit na následující tvar:

```
| result |
result := 0.
```

---

<sup>8</sup> Tyto tzv. **návrhové vzory** mají v případě profesionální tvorby softwaru velmi důležitou úlohu a míra jejich znalostí a používání je jedním z měřítek porovnávání kvality softwaru i samotných programátorů. Návrhovými vzory se zabývá kapitola “Návrhové vzory”.

```
myNumbers do: [:x | result := result max: x].
^result
```

Tento tvar je nejen srozumitelnější, ale je také výhodnější v tom, že na rozdíl od předchozího tento kód dokáže pracovat i se sadami bez vnitřního uspořádání, tedy například `Set` nebo `Bag`, na jejichž prvky nelze přistupovat pomocí indexu.

Ale ani tento tvar není konečný, protože celou úlohu lze vyjádřit posláním jediné zprávy následujícím způsobem:

```
^ myNumbers inject: 0 into: [:a :b | a max: b]
```

Tento tvar ve srovnání s prvním uvedeným názorně ukazuje, jak velmi je důležitá znalost knihovny s hotovými objekty a metodami, které lze uplatnit při tvorbě objektových programů. Proto by knihovna hotových objektů měla být co největší a také snadno rozšiřitelná.

## 2.7.4 Úloha polymorfismu v objektové algoritmizaci

Polymorfismus hraje ve tvorbě objektového softwaru velmi důležitou úlohu. Právě polymorfismus je příčinou toho, že objektový model výpočtu má svoje vlastní techniky a nástroje pro konstrukci algoritmů, které se mnohdy velmi podstatným způsobem liší od algoritmů v klasickém modelu výpočtu. V klasickém modelu výpočtu se totiž pracuje pouze se strukturami jednotlivých příkazů (iterace, větvení, skoky, ...). Objektový model výpočtu ale navíc dovoluje do algoritmů zahrnovat i datové hierarchie a využívat jejich vlastnosti. Tato skutečnost potom může velmi významným způsobem odlehčit vlastnímu kódu.

### 2.7.4.1 Využití při návrhu nových objektových komponent – vzor *double dispatching*

Představme si, že máme za úkol implementovat nějaký subsystém, kde se bude používat  $N$  různých tříd objektů v  $M$  různých situacích, tedy  $N \times M$  různým způsobem.

Dobrým příkladem mohou být například třídy grafických objektů `Line`, `Circle` a `Rectangle`, které budeme potřebovat jak pro zobrazení na tiskárnu (`Printer`), tak samozřejmě i na obrazovku (`Screen`), tedy v našem případě je  $N=3$  a  $M=2$ . Dále máme za úkol zajistit, aby všechny objekty bylo možno na všech zařízeních vytisknout. Musíme tedy napsat 6 vzájemně odlišných kódů.

Jednou z možností je napsat pro každou třídu jednu jedinou metodu `displayOn:` následujícím způsobem:

Line instance methods

**displayOn: aDevice**

```
aDevice class = Printer ifTrue: [... kód jak tisknou čáru na tiskárně...].
aDevice class = Screen ifTrue: [... kód jak zobrazit čáru na obrazovce
...]
```

Circle instance methods

**displayOn: aDevice**

```
aDevice class = Printer ifTrue: [... kód jak tisknout kruh na tiskárně...].
aDevice class = Screen ifTrue: [... kód jak zobrazit kruh na obrazovce
...]
```

Rectangle instance methods

**displayOn: aDevice**

```
aDevice class = Printer ifTrue: [... kód jak tisknout obdélník na
tiskárně...].
aDevice class = Screen ifTrue: [... kód jak zobrazit obdélník na obrazovce
...]
```

Máme tedy diskutovaných 6 kódů schovaných ve třech metodách. Toto řešení má výhodu v tom, že z pohledu vnějšího uživatele takto navrženého systému jsou jednotlivé detaily skryty do kódů metod a uživatel může bez omezení využívat takto vytvořeného polymorfismu například následujícím způsobem:

```
anyShape displayOn: anyDevice
```

Vzhledem k polymorfismu zprávy `displayOn:` může proměnná `anyShape` odkazovat na objekt libovolné třídy ze tříd `Line`, `Circle` a `Rectangle` a proměnná `anyDevice` na objekt libovolné třídy ze tříd `Printer` a `Screen`.

Uvedené řešení však trpí jedním podstatným nedostatkem. Kromě toho ani testovací výrazy v uvedených metodách nepatří mezi příliš elegantní způsoby algoritmizace. Problém spočívá v tom, jak by asi byla složitá implementace nějaké další nové třídy do množiny  $M$  tříd, jako například nové výstupní zařízení – třeba `Plotter`. V případě uvedeného řešení by to znamenalo přepsat všechny metody `displayOn:` ve všech  $N$  třídách a doplnit je o další kód pro `plotter`. Takže přidání nové třídy do systému by neznamenal jen přidání této třídy sama o sobě, ale i změny kódu okolních tříd, což je potenciálním zdrojem mnoha chyb a proto se takovéto zásahy u profesionálního software považují za nepřijatelné.

Jak tedy jiným způsobem rozložit diskutovaných 6 kódů, které potřebujeme naprogramovat? Musíme docílit toho, aby každý kód měl svoji samostatnou metodu, aby se každý kód pokud možno staral jen o vlastní třídu a zároveň aby nedošlo k porušení polymorfismu. Toto vše je možné velmi elegantně vyřešit následujícím způsobem:

Line instance methods

```
displayOn: aDevice
```

```
  aDevice displayLine: self
```

Circle instance methods

```
displayOn: aDevice
```

```
  aDevice displayCircle: self
```

Rectangle instance methods

```
displayOn: aDevice
```

```
  aDevice displayRectangle: self
```

a potom potřebné zobrazovací kódy implementovat jako

Printer instance methods

```
displayLine: aLine
```

```
  ... kód, jak tisknout čáru na tiskárně...
```

```
displayCircle: aCircle
```

```
  ... kód, jak tisknout kruh na tiskárně...
```

```
displayRectangle: aRectangle
```

```
  ... kód, jak tisknout obdélník na tiskárně...
```

Screen instance methods

```
displayLine: aLine
```

```
  ... kód, jak zobrazit čáru na obrazovce...
```

```
displayCircle: aCircle
```

```
  ... kód, jak zobrazit kruh na obrazovce...
```

```
displayRectangle: aRectangle
```

```
  ... kód, jak zobrazit obdélník na obrazovce...
```

Tento návrhový vzor se nazývá double dispatching a je v systému Smalltalk používán na mnoha místech, mezi nejznámější patří aritmetika mezi čísly různých datových typů, práce se sadami a streamy a samozřejmě také grafika. Tento návrhový vzor vlastně vychází z jednoho ze základních pravidel objektově orientovaného přístupu, a to zásady „každý at' dělá co je v jeho kompetenci, nic víc, nic méně“. Je tedy logické, že to, jak vytisknout určitý grafický prvek by mělo být starostí tiskárny a ne tohoto prvku. Uvedený příklad byl ale jen velkým zjednodušením původního řešení, ve kterém je několik desítek různých objektů. Pro zájemce doporučujeme také seznámit se s podporou PostScriptu™ v tomto subsystému. Na tomto principu je také implementována knihovna všech číselných datových typů.

#### 2.7.4.2 Úloha datového modelování a polymorfismu při návrhu objektových algoritmů

Představme si následující úkol: Máme napsat program, který bude napodobovat chování prodejního automatu na jízdenky na vlak. Program musí rozlišovat mezi I. a II. třídou a mezi osobním vlakem a rychlíkem intercity (IC) a bude počítat cenu jízdenky v závislosti na druhu a počtu kilometrů. Ihned se nabízí následující jednoduché řešení, jehož podstatnou částí by mohl být následující kód:

```
. . .
druhJizdenky = 'osobni II. trida'
  ifTrue: [cena := km * cenaZaKM].
druhJizdenky = 'osobni I. trida'
  ifTrue: [cena := km * cenaZaKM + prirazkaZa1Tridu].
druhJizdenky = 'IC II. trida'
  ifTrue: [cena := km * (cenaZaKM + ic)].
druhJizdenky = 'IC I. trida'
  ifTrue: [cena := km * (cenaZaKM + ic) + prirazkaZa1Tridu].
. . .
```

Jak je vidět, tak uvedený kód obsahuje přímo vlnobití rozhodovacích výrazů `ifTrue: ... ifTrue:..`. V některých imperativních programovacích jazycích pro tento případ má programátor dokonce k dispozici zvláštní syntaktickou strukturu, kterou lze toto větvení napsat úsporněji. (Příkaz `case of` v jazyce Pascal a nebo `switch` v jazyce C) Je však třeba připomenout že tato syntaktická zkratka nijak podstatně nemění průběh výpočtu samotným počítačem.

Podívejme se však na uvedený příklad z pohledu objektového modelu výpočtu. Na úlohu je možné také nahlížet jako na objekt, který musí vypočítat cenu jízdného na základě druhu jízdenky a požadovaného počtu kilometrů. Nic nám nebrání v tom, abychom tento diskutovaný objekt ztotožnili se samotnou jízdenkou (nebo druhem jízdného) a navrhli čtyři různé navzájem polymorfní třídy jízdenek, které budou společně modelovat různé varianty - například takto:

```
Object Smalltalk defineClass: superClass: #{}#ObycejneJizdne
ObycejneJizdne instance methods
```

```
cenaZaKM: aNumber
  ^ aNumber * cenaZaKM
```

```
Object Smalltalk defineClass: superClass: #{}#ObycejneJizdne1Trida
ObycejneJizdne1Trida instance methods
```

```
cenaZaKM: aNumber
  ^ aNumber * cenaZaKM + prirazkaZa1Tridu
```

```
Object Smalltalk defineClass: superClass: #{}#ICJizdne
ICJizdne instance methods
```

```
cenaZaKM: aNumber
  ^ aNumber * (cenaZaKM + ic)
```

```
Object Smalltalk defineClass: superClass: #{}#ICJizdne1Trida
```

ICJizdne1Trida instance methods

```
cenaZaKM: aNumber
  ^ aNumber * (cenaZaKM + ic) + prirazkaZa1Tridu
```

Pokud máme takto připravené polymorfnní třídy objektů, tak se nám samotný program zjednoduší na posílání jedné jediné zprávy:

```
. . .
naseJizdne cenaZaKM: km
. . .
```

Vidíme tedy, že větvení, které v prvním případě bylo nutnou součástí programu, nyní z tohoto programu zcela zmizelo, protože jeho úlohu nyní dělají objekty samy pomocí svého polymorfismu. Je totiž třeba mít vždy na paměti, že při tvorbě objektových algoritmů hrají důležitou roli i hierarchie samotných objektů, které mohou, jak vidíme, mít podstatný vliv na konečnou podobu kódu jednotlivých programů. Pozornému čtenáři jistě neunikla ještě možnost využití dědičnosti pro podporu vzájemného sdílení kódu metod vypočítávajících cenu jízdného, jako například:

```
Object Smalltalk defineClass: superClass: #{}#ObycejneJizdne
ObycejneJizdne instance methods
```

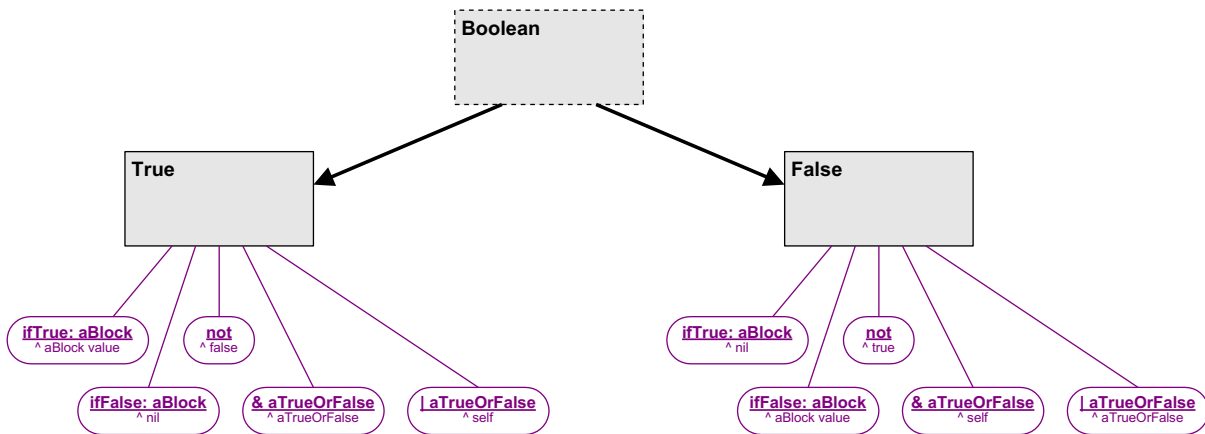
```
cenaZaKM: aNumber
  ^ aNumber * cenaZaKM
```

```
ObycejneJizdne Smalltalk defineClass: superClass: #{}#ObycejneJizdne1Trida
ObycejneJizdne1Trida instance methods
```

```
cenaZaKM: aNumber
  ^ (super cenaZaKM: aNumber) + prirazkaZa1Tridu
```

To, že při návrhu objektových programů hrají podstatnou úlohu i hierarchie tříd objektů a polymorfismus metod, svědčí i mnohé konstrukce v samotné systémové knihovně Smalltalku. Uvedme si proto jeden reprezentativní příklad, kterým je výběr z implementace tříd pro podporu boolské algebry a výrokové logiky.

Boolská algebra je implementována ve třídách True a False, které dědí ze společné abstraktní nadtřídy Boolean. Třída True má jako svoji instanci konstantu true a třída False má jako svoji instanci konstantu false. Podívejme se nyní na implementaci metod & (logický součin), | (logický součet), not, ifTrue: a ifFalse:...



Ze schématu je zřejmé, že v návrhu kódů jednotlivých metod je dobře uplatněna znalost teoretického základu, což se projevuje v elegantní symetrii kódů jednotlivých metod. Další velmi pozoruhodnou skutečností je fakt, že kódy uvedených metod již nepotřebují pro výpočet svých výsledků provádět žádné

operace porovnávání, skoky nebo podmíněné výrazy, neboť veškerou potřebnou logiku v tomto případě zastane samotný objektový model výpočtu opírající se o sebeidentifikaci objektů a polymorfismus. Tato vlastnost objektových programů, kterou jsme tímto příkladem částečně ukázali, má ve svém důsledku mnoho pozitivních dopadů na celý proces tvorby i provozu objektového software.

Například v prostředí Smalltalku totiž způsob implementace tříd `True` a `False` způsobuje, že na systémové úrovni běžící jakýkoliv výpočet nedělá žádné skoky a průběh výpočtu je bez větvení, neboť je tvořen pouze lineární posloupností posílaných zpráv objektům! Ve Smalltalku je variabilita výpočtu zajišťována výhradně pomocí pozdní vazby a polymorfismu samotných navzájem různých objektů během výpočtu. Toto řešení je v souladu s myšlenkami „čistého“ objektového programování, a opírá se o něj i realizace objektového debuggeru a dovoluje také poměrně snadnou implementaci ošetření výjimek a různých softwarových konstrukcí souvisejících s paralelním programováním a nebo umělou inteligencí jako například podpora zpětného šíření (angl. *backtracking*) výpočtu atp.

## 2.8 Ošetřování chybových stavů

Při běhu programu se mohou vyskytnout různé chybové situace. Ty mohou vzniknout jak uvnitř systému (např. dělení nulou), tak zvnějšku – například uživatel zadá neplatná data. Na tyto situace je třeba určitým způsobem reagovat. Jedním z nejdůležitějších hledisek kvality programu je právě schopnost adekvátně ošetřit tyto situace. V případě, že je možné se z chyby zotavit, program by tak měl učinit (např. vyzvat uživatele k novému zadání hodnoty) a ne se zhroutit. I v případě chyby, kdy již dále není možné pokračovat ve výpočtu, kvalitní program ohlásí vysvětlujícím způsobem, k jaké situaci došlo. Obecná hláška typu „došlo k chybě“ uživateli ani správci programu příliš v nápravě nepomůže.

Ošetřování chybových stavů je pro programátory velmi otravnou a nudnou záležitostí, neboť to znamená přestat se soustředit na jádro programovaného problému a řešit okrajovou situaci. Často je proto ošetřování chybových stavů zanedbáváno. Navíc ošetřování výjimečných stavů znehledňuje kód a jeho čitelnost, protože zanáší operace, které nesouvisí přímo s problémem.

Nejpohodlnějším a nejmocnějším mechanismem je v současnosti mechanismus výjimek. Ačkoliv není striktně vázán na objektovou orientaci (např. jazyk Ada též výjimky používá), setkáme se s ním většinou v objektově orientovaných systémech. Mechanismus tříd a dědění totiž poskytuje velkou sílu pro výjimky.

Základní myšlenkou výjimek je oddělení výkonné části kódu od části kódu, kde zpracováváme chybové stavy, což významně zvyšuje přehlednost.

### 2.8.1 Mechanismus výjimek

Smalltalk implementuje standardizovaný ANSI mechanismus výjimek (až na drobná rozšíření). Podobně se pracuje s výjimkami i v ostatních jazycích s podporou objektové orientace (C++, Java).

Výjimky jsou instance podtřídy třídy `Exception`, které vzniknou v okamžiku, kdy dojde k chybovému stavu při běhu programu<sup>9</sup>. Ošetření výjimky znamená, že ji tzv. *zachytíme* a nějak na ni zareagujeme. Pokud toto není učiněno, výjimka se šíří dál do nadřazené metody, tedy metody ze které byla volána metoda, ve které k výjimce došlo.

#### 2.8.1.1 Zachytávání výjimek

Obecné schéma chráněného kódu vypadá takto:

[chráněný blok, ve kterém může dojít k výjimce]

---

<sup>9</sup> jakým způsobem se tak děje, viz sekce “Vytváření vlastních výjimek a vyvolávání výjimek“

```

on: Třída_výjimky, Třída_výjimky2
do: [:ex | ošetření_výjimky]

```

Pokud tedy při provádění chráněného bloku vznikne výjimka, která je instancí třídy *Třída\_výjimky* (tj. zpráva `isKindOf: Třída_výjimky` vrací `true`), je řízení předáno do bloku *ošetření\_výjimky*, kde na výjimku můžeme zareagovat. Jak vidíme, odchyťovaných tříd může být i více (*Třída\_výjimky2*). Proměnná `ex` bude obsahovat vzniklou výjimku a můžeme jí tedy posílat zprávy. Z nich nejdůležitější jsou zprávy:

<code>isResumable</code>	Odpovídá, jestli je možné pokračovat ve výpočtu.
<code>resume</code>	Pokračuje ve výpočtu za zprávou, která výjimku vyvolala.
<code>return</code>	Ukončí provádění chráněného bloku.
<code>retry</code>	Znovu vykoná chráněný blok.
<code>retryUsing:</code>	Místo původního bloku vykoná blok uvedený jako parametr.
<code>exit</code>	Provede <code>resume</code> nebo <code>return</code> podle charakteru výjimky ( <code>resumable</code> )
<code>description</code>	Obsahuje zprávu, kterou se výjimka hlásí. Vhodné např. pro informování uživatele typu  Dialog warn: 'Vznikla chyba ', ex description, ', informujte správce'.

Příklady:

```

| x y |

x := 7.
y := 0.
[x / y]
  on: ZeroDivide
  do: [ :ex | Transcript show: 'Doslo k deleni nulou.'; cr.]
[^ x / y]
  on: ZeroDivide
  do:
    [:exception|
      "raději dělitel hodně malý, ale > 0"
      y := 0.00000001.
      exception retry]

```

To, co činí objektový mechanismus výjimek mocným (na rozdíl např. od mechanismu výjimek jazyka Ada) je právě začlenění mezi standardní objektové paradigma a tedy existence třídní hierarchie. Pokud budeme například odchyťovat výjimky třídy `ArithmeticError`, zachytí se jak `ZeroDivide`, tak `RangeError`.

### 2.8.1.2 Vytváření vlastních výjimek a vyvolávání výjimek

Programátor má možnost si vytvářet vlastní hierarchii výjimek pro svoji aplikaci. Ve správně postaveném programu by každý druh chybového stavu měl odpovídat některé výjimce. Svoje výjimky budeme většinou vytvářet jako podtřídu třídy `Error`. Nové výjimce obvykle není třeba implementovat žádnou další speciální funkčnost, neboť mechanismus výjimek ji jen používá k signalizaci a stačí tedy, že „je“.

A nyní se dostáváme k otázce, jak vlastně výjimky vznikají. Výjimky jsou obvykle vytvářeny posláním zprávy `raiseSignal:` příslušné třídě výjimky. Parametrem je `String`, obvykle s vysvětlením příčiny. Pokud tedy vytvoříme třídu `InvalidTelephoneException`, můžeme potom napsat metodu:

```

getTelephoneFromUser
|telephone|
telephone := Dialog request: 'Zadej telefonni cislo'
.....
.....telephone size ~= 9
ifTrue: [InvalidTelephoneException raiseSignal: 'Chybne
telefonni cislo: ', telephone]
^telephone

```

A použít ji s ošetřením výjimky např.:

```

[Telephone dial: self getTelephoneFromUser]
on: InvalidTelephoneException10
do:
[:ex |
Dialog warn: 'Spatne telefonni cislo!'.
ex retry]

```

Jelikož text výjimky `InvalidTelephoneException` bude zřejmě vždy začínat `'Chybne telefonni cislo'`, mohla by se o tento výpis starat výjimka sama (dle hesla "Každý by měl dělat co mu přísluší, nic méně, nic víc"). Předefinujeme tedy její třídní metodu `raiseSignal`:

```

raiseSignal: zprava
^super raiseSignal: 'Chybne telefonni cislo: '

```

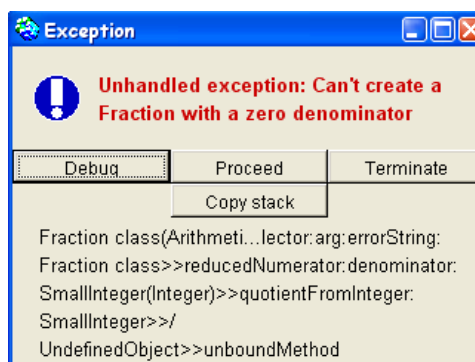
a bude pak stačit psát jen:

```

InvalidTelephoneException raiseSignal: telephone

```

Pozorný čtenář si možná všiml, že text `'Chybne telefonni cislo: '` se vlastně nikde nezobrazí ani nijak nevyužije, jelikož uživatele informujeme sami v obsluze výjimky. To je pravda, ale tento slovní popis by se hodil v případě, že bychom někde zapoměli výjimku odchyťovat a zrovna by k ní došlo. Pokud totiž výjimka není na žádné úrovni programu odchyčena, „probublá“ až na úroveň systému, což se projeví otevřením okna se zprávou. Např. při pokusu o vytvoření zlomku s nulovým dělitelem dostaneme:



Tlačítko *Terminate* ukončí provádění programu, tlačítko *Debug* otevře okno Debuggeru (viz kapitola „Techniky ladění programů“) a tlačítko *Proceed* se pokusí zopakovat neúspěšnou činnost.

<sup>10</sup> V praxi bychom navíc ještě zřejmě odchyťovali nějakou výjimku, která by mohla vzniknout při vytáčení čísla, např. `LineBusyException`.

## 2.8.2 Závislost objektů

Ve Smalltalku naprogramovaná aplikace nemá tzv. hlavní program, který je základním pilířem procedurálního programování. Program se ve Smalltalku skládá z množiny objektů, které mezi sebou komunikují pouze pomocí navzájem si posílaných zpráv. Některé objekty mají na starosti správu uživatelského rozhraní, což znamená, že přijímají nebo vydávají zprávy uživateli. Tyto zprávy jsou nazývány jako *vnější události*. Na systému je důležitý fakt, že v něm neexistuje žádný souvislý blok kódu (např. nějaká metoda), jehož provádění by probíhalo celým životním cyklem spuštěné aplikace od startu až po její ukončení. V procedurálním programování je však tato "páteř" programu naprosto nezbytná (i v Object Pascalu a C++) a odvíjí se od ní také mnoho dobře známých procedurálních programovacích technik (např. JSD, Structure Chart diagramy apod.). Důsledkem této koncepce je nemožnost měnit za chodu funkčnost programu, neboť po překladu jsou všechny kódy "zalité do betonu" z přeloženého hlavního programu.

Objektově orientovaný program ve Smalltalku se však skládá jenom z navzájem komunikujících objektů. Hlavní algoritmus nemusí být nikde explicitně popsán, neboť program se zjednodušeně řečeno řídí sám podle vnějších souvislostí a podle dynamicky utvářeného sledu zpráv z jednotlivých prováděných kódů metod zúčastněných objektů. Namísto bloku hlavního programu zde výpočet řídí, zahajuje a ukončuje uživatelské rozhraní.

V naznačené architektuře aplikačního programu je výpočet řízen kódy metod u jednotlivých objektů. Pro spouštění těchto operací nám slouží technika posílání zpráv objektům. Jednotlivé zprávy zde vystupují jako žádosti o operace. Kromě jednoduchého posílání zpráv však ve Smalltalku existuje i další mnohem rafinovanější řízení provádění operací. Tato technika využívá tzv. závislosti (dependency) objektů mezi sebou.

Cílem této techniky je stejně jako při přímém posílání zpráv provedení nějaké operace s nějakým objektem. Při obyčejném posílání zprávy však žadatel (vysílač zprávy) vždy musí znát objekt (příjemce zprávy), se kterým má být příslušná operace provedena, protože kód volání zprávy se vždy skládá z příjemce zprávy, selektoru zprávy a případných parametrů. Při použití analogie z reálného světa se v tomto případě jedná o situaci, kdy se přímo obracíme k nějakému subjektu s žádostí (rozkazem) o vykonání nějaké činnosti.

V reálném světě však také dochází k tomu, že si přejeme provedení nějaké činnosti, aniž bychom znali (nebo potřebovali znát), kdo danou operaci bude provádět. V tomto případě tedy dáváme žádost (povel) k vykonání, ale neobracíme se přímo k subjektům, kteří povel vykonají, protože spoléháme na to, že existuje někdo, kdo je na nás závislý, na náš povel zareaguje, a provede ho. Tato myšlenka je základem tzv. *klient-server* vztahu objektů ve Smalltalku využívajícího vzájemné závislosti objektů na sobě.

Objekty typu *server* jsou závislé na objektem typu *klient*. Jestliže objekt typu *klient* vyšle příslušný signál, tak všechny jemu závislé objekty typu *server* provedou předepsanou operaci (metodu).

Do této závislosti se může dostat jakýkoliv objekt. Proto systém Smalltalku tyto vlastnosti implementuje v mnoha instančních metodách třídy `Object`. Z nich vybíráme následující metody:

```
addDependent: aServerObject
    příjemci zprávy se přidá závislý objekt aServerObject
```

```
dependsOn: aClientObject
    příjemce se stane závislým na objektu aClientObject
```

```
dependents
    výsledkem volání je množina příjemci závislých objektů
```

Objekty typu *klient* se při posílání žádostí o operaci neobracejí ke svým *serverům*, protože je ani nemusejí znát. Žádosti o operace jsou však také realizovány pomocí zpráv, které si *klienti* posílají jakoby "bez příjemce sami sobě" (jsou jejich příjemci):

changed

na poslání této zprávy příslušní *serverové* zareagují provedením metody `update`

changed: aParameter

na poslání této zprávy příslušní *serverové* zareagují provedením metody `update`: aParameter

changed: aParameter1 with: aParameter2

na poslání této zprávy příslušní *serverové* zareagují provedením metody `update`: aParameter1  
with: aParameter2

Při využití techniky závislých objektů je tedy třeba nejprve vytvořit vazbu mezi objekty (`addDependent:`, `dependsOn:`) a poté u objektů *server* implementovat kód metody `update` (`update:`, `update:with:`), který bude obsahovat požadované operace.

Jinou možností práce se závislými objekty je případ, kdy objekty typu *server* mají provádět více různých operací a rozlišení operace pomocí parametrů ve zprávě `changed`: není z nějakých důvodů vhodné. V tomto případě žádosti objektů typu *klient* obsahují i název (selektor) metody, kterou budou objekty typu *server* provádět. Tyto žádosti stejně jako předchozí si *klienti* posílají "sami sobě":

broadcast: aSelector

na poslání této zprávy *klientu* je příslušným *serverům* poslána zpráva s názvem aSelector

broadcast: aSelector with: aParameter

na poslání této zprávy *klientu* je příslušným *serverům* poslána zpráva s názvem aSelector a s parametrem aParameter

broadcast: aSelector with: aParameter1 with: aParameter2

na poslání této zprávy *klientu* je příslušným *serverům* poslána zpráva s názvem aSelector a se dvěma parametry aParameter1 a aParameter2

Poznámka:

I když obsah této poznámky přímo nesouvisí se závislostí objektů, tak považujeme za vhodné ukázat, že ve Smalltalku existuje i možnost, že každý objekt umí poslat sám sobě žádost o operaci (jakoby byl sám na sobě závislým):

perform: aSelector

na poslání této zprávy je příjemci poslána další zpráva s názvem aSelector

perform: aSelector with: aParameter

na poslání této zprávy je příjemci poslána další zpráva s názvem aSelector a s parametrem aParameter

perform: aSelector with: aParameter1 with: aParameter2

na poslání této zprávy je příjemci poslána další zpráva s názvem aSelector a s parametry aParameter1 a aParameter2

Jednoduchý příklad nám vypočítá hodnotu faktoriálu čísla 20:

```
|oper|
oper := #factorial.
20 perform: oper
```

Kromě bloků tedy máme ve využití zpráv `perform`: další možnost, jak lze ve Smalltalku realizovat parametrizované operace. Praktické použití může být například rozhodování mezi různými činnostmi na základě nějakého vstupu:

```
|d command msg|
d := Dictionary new.
d at: 'tisknout' put: #doPrint;
  at: 'zobrazit' put: #doShow;
  at: 'smazat' put: #doDelete.
command := Dialog request: 'Prikaz'.
msg := d at: command ifAbsent: [^Dialog warn: 'Neznamy prikaz'].
self perform: msg.
```

Tento příklad je také další ilustrací toho, že v čistě objektově orientovaném prostředí jsme velmi často schopni se elegantně vyhnout krkolomným konstrukcím if-then-else.

### 2.8.2.1 Malé příklady využití závislosti objektů

Příkladem může být část myšlené aplikace, která bude simulovat závody v běhu. Budeme potřebovat rozhodčího (*klienta*), který bude závod startovat a závodníky, kteří v závodě poběží (*servery* rozhodčího). Rozhodčí při odstartování závodu nepotřebuje znát, kolik a kteří závodníci v závodě poběží. Kód programu může být následující:

*kód rozhodčího*

```
Smalltalk defineClass: #Starter superClass: #{Object}
  instanceVariableNames: ''
  category: 'athletics'
```

```
Starter methodsFor: 'activity'
```

**doStart**

```
"zde rozhodčí vyšle signál ke všem závodníkům"
self changed
```

*kód závodníka*

```
Smalltalk defineClass: #Athlete superClass: #{Object}
  instanceVariableNames: 'name'
  category: 'athletics'
```

```
Athlete class methodsFor: 'instance creation'
```

**named: aString**

```
"vytvoří nového pojmenovaného závodníka"
^ (super new) name: aString
```

```
Athlete methodsFor: 'instance initialization'
```

**name: aString**

```
"pojmenuje závodníka"
name := aString
```

```
Athlete methodsFor: 'activity'
```

**update**

```
"závodník se rozeběhne"
Transcript show: name , ' runs' ; cr
```

Takto napsaný kód si lze vyzkoušet ve Workspace. Mějme například tři závodníky a jednoho rozhodčího:

```
|s|
```

```
"inicializace subsystému"
```

```

s := Starter new.
s addDependent: (Athlete named: 'Martin').
s addDependent: (Athlete named: 'Charles').
s addDependent: (Athlete named: 'Bou Melhem').

"startovací signál rozhodčího"

s doStart

```

Nyní si představme, že rozhodčí startuje více závodů. například na 100 a na 200 metrů. Někteří závodníci mají běžet jeden závod, jiní druhý závod:

#### *kód rozhodčího*

```

Smalltalk defineClass: #Starter superClass: #{Object}
  instanceVariableNames: ''
  category: 'athletics'

Starter methodsFor: 'activity'

doStart: aType
  "zde rozhodčí vyšle signál ke všem závodníkům, ale v parametru je druh
závodu"
  self changed: aType

```

#### *kód závodníka*

```

Smalltalk defineClass: #Athlete superClass: #{Object}
  instanceVariableNames: 'name type'
  category: 'athletics'

Athlete class methodsFor: 'instance creation'

named: aString type: aNumber
  "vytvoří nového závodníka pro určitý závod"
  ^ (super new) name: aString ; type: aNumber

Athlete methodsFor: 'instance initialization'

name: aString
  "pojmenuje závodníka"
  name := aString

type: aNumber
  "nastaví disciplínu závodníka"
  type := aNumber

Athlete methodsFor: 'activity'

update: aType
  "jestliže souhlasí disciplína, závodník se rozeběhne"
  aType = type
  ifTrue: [Transcript show: name , ' runs' ; cr]

```

Napsaný kód si opět vyzkoušíme. Na rozdíl od předešlého příkazu by se však měli rozeběhnout pouze dva závodníci:

```

|s|

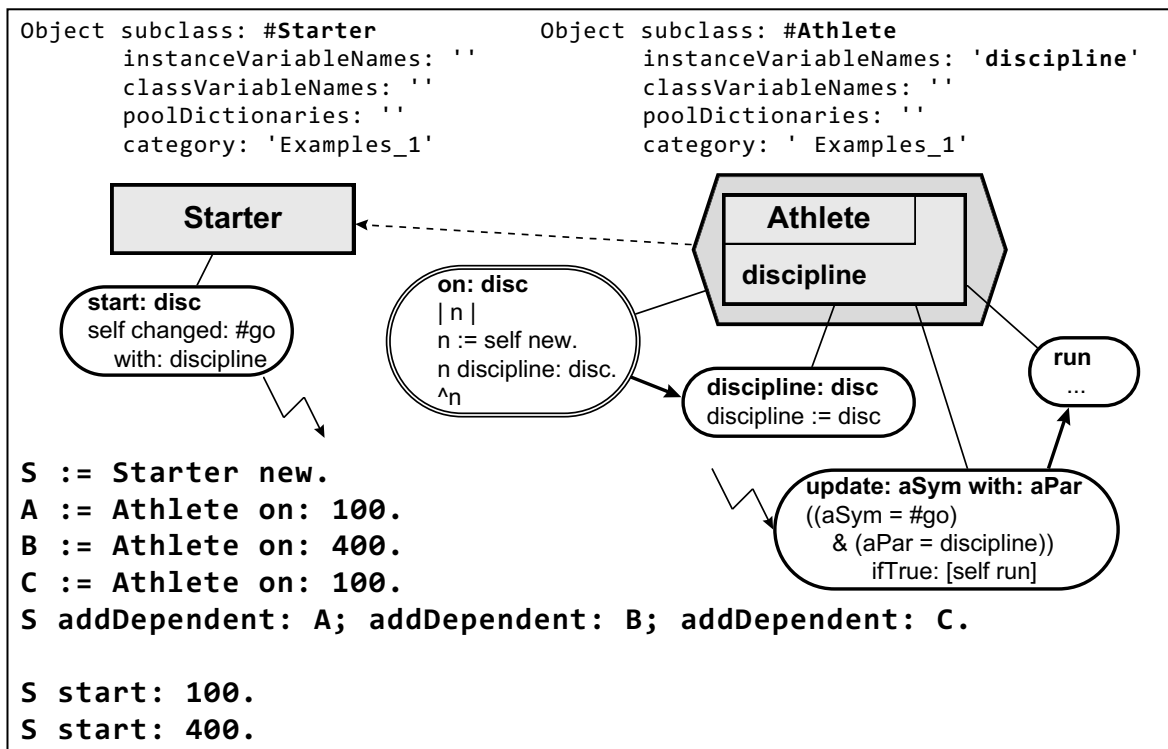
"inicializace subsystému"
s := Starter new.
s addDependent: (Athlete named: 'Martin' type: 100).

```

```
s addDependent: (Athlete named: 'Charles' type: 200).
s addDependent: (Athlete named: 'Bou Melhem' type: 100).

"startovací signál rozhodčího"

s doStart: 100
```



### 2.8.3 Architektura MVC

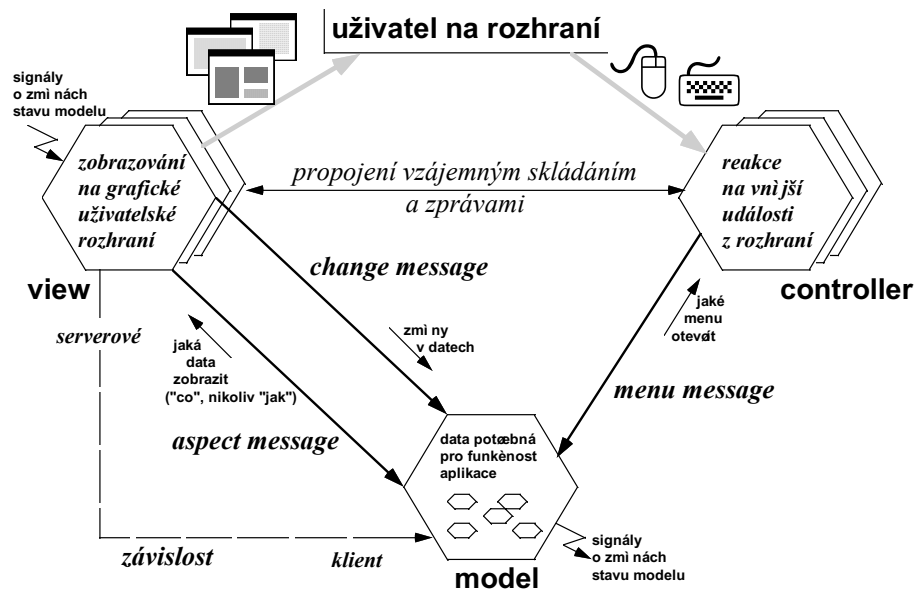
Ve Smalltalku je kromě uživatelského rozhraní objektivě orientovaný celý výpočet. S tímto faktem se však někteří programátoři z naší vlastní zkušenosti obtížně vyrovnávají. V prostředí jiných systémů (např. C++) je totiž vždy možné vyvíjet klasický procedurální program a objekty si vypůjčovat pouze pro zobrazování v uživatelském rozhraní. Podle některých zahraničních průzkumů až 90% všech programů napsaných v prostředích Borland či Microsoft C++ není čistě objektivě orientovaných.

Jak jsme již dříve řekli, ve Smalltalku neexistuje hlavní program. Aplikace se tedy skládá z jednotlivých objektů. Tyto objekty lze rozdělit do tří hlavních skupin:

- objekty typu *Model*, které zabezpečují vnitřní datovou a funkční reprezentaci programované aplikace,
- objekty typu *View*, které zabezpečují zobrazování do uživatelského rozhraní během výpočtu,
- objekty typu *Controller*, které zabezpečují vstup vnějších událostí od uživatele přes uživatelské rozhraní do systému (např. menu).

Při tvorbě aplikace ve Smalltalku s využitím techniky MVC programátor vytváří vazby mezi těmito typy objektů. Jsou to především následující vazby (viz. obrázek):

## MODEL-VIEW-CONTROLLER ARCHITEKTURA



1. Jakými zprávami se budou objekty *view* dotazovat svých *modelů* na data, která mají zobrazit (tzv. *aspectMessages*),
2. jaké zprávy a s jakými parametry budou od objektů *controller* a *view* posílány objektům *model* při změnách zobrazených dat (*changeMessages*) a
3. jakými zprávami budou objekty *controller* dotazovat svých *modelů* na menu, která mají zobrazit (*menuMessages*).

K překreslování objektů *view* při změnách stavu *modelu* je využívána *klient-server* vazba závislosti *view* na *modelech*. Jeden objekt typu *model* běžně bývá *klientem* několika různých objektů typu *view* a *controller*, protože aplikace ve Smalltalku se málokdy skládá z jediného okna s jedinou oblastí v něm.

Mezi objekty *view* a *controller* je **bez nutnosti programování** (v systému jsou již obsažené příslušné metody) oboustranně vytvářena vazba skládání. Stejně tak i objekty *model* jsou za chodu programu přítomny v MVC systému jako automaticky vytvořené složky objektů *view*.

Pro Smalltalk je důležité, že při programování klasických aplikací jsou ze systému znovuvyužívány všechny potřebné objekty (a automaticky vytvářena většina jejich vzájemných vazeb) typu *view* a *controller* (různá tlačítka, editory, prohlížeče apod.), což drasticky snižuje náklady na tvorbu programu.

Další výhodou architektury MVC a neexistence hlavního programu je možnost ladit a měnit programy za jejich chodu, což je také velmi často ve Smalltalku záměrně využíváno: Programátor napíše kostru aplikace, spustí ji, a za jejího chodu postupně doplňuje, experimentuje a mění metody (funkčnost) objektů *model*, *view* i *controller*.

Při bližším se zamyšlení nad architekturou systému Smalltalk vychází najevo zajímavá skutečnost. Celý systém se totiž chová jako jeden kompaktní fungující objektový program a všechny aplikace v něm jsou vlastně jeho stále za chodu doplňované, po částech překládané a modifikované vlastnosti a data. Proto se o Smalltalku někdy hovoří jako o objektově orientovaném operačním systému a ne jen o programovacím jazyce. Tyto vlastnosti činí Smalltalk ve srovnání s jinými systémy velmi flexibilním. Možnost programování za chodu spolu s využitelným a doplňovatelným systémem všech objektů a s možností ukládat systém při ukončení práce na disk (funkce `save image`) včetně všech běžících procesů a opětovné 100% obnovení systému při následném spuštění jsou hlavními příčinami vysoké produktivity a kreativity programátorské práce ve Smalltalku ve srovnání s klasickými systémy.

## 2.8.4 Komponenty grafického uživatelského rozhraní (GUI)

Základem uživatelského rozhraní každé aplikace je okno. Ve Smalltalku jsou okna nejčastěji instancemi třídy `ScheduledWindow`. Pro práci s okny slouží (jak jinak) velké množství metod, z nichž vybíráme několik v následujícím příkladě (pro `Workspace`):



```
|win|
win := ScheduledWindow new.
win label: 'Hello World!';
  minimumSize: 150 @ 100;
  insideColor: ColorValue yellow;
  open
```

Každé okno může obsahovat nějaký zobrazitelný objekt (text, obrázek). Následující příklad ukazuje okno, ve kterém je zvětšeně zobrazen obrázek sejmутý z obrazovky:



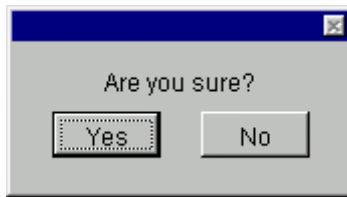
```
|win|
win := ScheduledWindow new.
win label: 'my image';
  minimumSize: 150 @ 100;
  component: (Image fromUser magnifiedBy: 2@2);
  open
```

### 2.8.4.1 Dialogy

Poslední součástí uživatelského rozhraní zde představenou jsou dialogy. Jedná se o zvláštní okna, která se v GUI objevují v případě potřeby interaktivního vstupu nebo výstupu dat. V největší míře se jedná o instance tříd `View` a `Menu`. Následující příklady jejich užití objasňují pomocí malých kódů, které je možné samostatně spouštět v okně `Workspace`:



Dialog request: 'Enter name...' slouží ke vložení řetězce,



Dialog confirm: 'Are you sure?' vrací true nebo false,



```
(Menu labelList: #( ('alpha' 'beta') ('gamma'))
  values: #(1 2 3))startUp
```

vrací hodnotu za sady v druhém parametru, tj. jeden prvek z pole #(1 2 3) .

## 2.9 Paralelismus

S tvorbou, laděním a chodem výše popsaných aplikací přímo souvisí problematika podpory paralelního průběhu výpočtu. Ve Smalltalku je možné libovolně velký blok kódu prohlásit za paralelní. V jednom časovém intervalu tedy může probíhat (a vždy také probíhá) několik logicky nezávislých bloků kódu současně. V systému je k tomu implementováno veliké množství metod, a to především u tříd `Process`, `ProcessorScheduler` a `Semaphore`.

Instance třídy `Process` představují jednotlivé paralelní procesy. Jednotlivé procesy mohou mít osm různých úrovní priority. Nejjednodušším způsobem, jak vytvořit paralelní proces, oddělit ho od hlavního proudu instrukcí a spustit ho, je poslat zprávu `fork` nějakému bloku. Vytvořený proces si ponechává stejnou úroveň priority, jakou měl kontext, ze kterého byl vytvořen. Jestliže má mít vytvářený proces jinou úroveň priority, je možné použít zprávu `forkAt: aPriorityLevel`.

úroveň	označení	použití
80	timing	procesy závislé na reálném čase
70	highIO	kritické I/O, např. správa sítě
60	lowIO	běžné I/O, např. klávesnice
50	userInterrupt	řízení chodu GUI
40	userScheduling	normální interakce s uživatelem
30	userBackground	uživatelské procesy na pozadí
20	systemBackground	systémové procesy na pozadí
10	-	rezervovaná nejnižší úroveň

Každý proces lze možné pozastavit (`suspend`), znovu spustit (`resume`) nebo ukončit (`terminate`). O automatické řízení procesů se v systému stará globální objekt `Processor`, který je instancí třídy `ProcessorScheduler`. Z jeho možností vybíráme metodu `yield`, která v systému dočasně potlačuje procesy běžící na nejvyšších úrovních priority, pokud jejich běh způsoboval stárnutí nebo uváznutí jiných procesů z nižších úrovní.

## 2.9.1 Koordinace paralelních procesů

Pro koordinaci procesů ve Smalltalku slouží instance třídy `Semaphore`. Každý semafor je objekt, který je schopen pozastavovat nebo znovu spouštět paralelní procesy. Pro pozastavení jednoho procesu slouží metoda `wait`, pro spuštění jednoho procesu slouží metoda `signal`. Jestliže jeden semafor zadržuje více procesů, tak se řadí přednostně podle své priority a poté podle pořadí. Například šest procesů A1 A2 B1 B2 C1 C2 (číslo značí úroveň priority), které byly jedním semaforem pozastaveny v pořadí (zleva doprava) A1 B1 A2 B2 C1 C2, opustí semafor v pořadí A2 B2 C2 A1 B1 C1.

Pro pozastavení procesu na nějaký daný časový úsek slouží instance třídy `Delay`, které se vytvářejí pomocí třídních metod `forSeconds: aNumber` nebo `forMilliseconds: aNumber`.

Pro předávání dat mezi paralelně běžícími procesy, což není vždy zcela jednoduchá záležitost, slouží instance třídy `SharedQueue`. Přístup do této vyrovnávací fronty je umožněn známými metodami pro objekty typu `Stream` ( například `next: a nextPut:`). Použití této vyrovnávací fronty umožňuje jednomu procesu zapsat data dříve, než přijímající proces je schopen data číst. Jestliže přijímající proces se pokouší číst data, která ještě nebyla jiným procesem napsána, je do této doby vyrovnávací frontou automaticky pozastaven.

## 2.9.2 Malé příklady paralelních procesů

Prvním příkladem budou digitální hodinky. Nejprve si vytvoříme okno, ve kterém se bude zobrazování času odehrávat (první řádek zařazuje do systému globální proměnnou pojmenovanou `Win`):

```
Smalltalk at: #Win put: ScheduledWindow new.  
Win label: 'time' ; open.
```

Nyní naprogramujeme zobrazovací cyklus. Algoritmus, který využívá možnost sekundového porovnávání instancí třídy `Time`, každou sekundu vytiskne do okna novou hodnotu času:

```
|tOld tNew gc|  
gc := Win graphicsContext.  
[true] whileTrue:  
  [(tNew := Time now) = tOld ifFalse:  
    [gc clear;  
     displayString: (tOld := tNew) printString  
     at: 20 @ 20]  
  ]
```

Takto naprogramovaný cyklus však neprobíhá paralelně, a proto nám "zablokuje" prostředí Smalltalku, ve kterém kromě běžících hodin vše ostatní stojí (zastavíme pomocí `Ctrl-Break`). Následující úprava kódu nám však zajistí, že hodiny poběží jako paralelní proces:

```
[ |tOld tNew gc|  
  gc := Win graphicsContext.  
  Smalltalk at: #B put: true.  
  [B] whileTrue:  
    [(tNew := Time now) = tOld ifFalse:  
      [gc clear;  
       displayString: (tOld := tNew) printString  
       at: 20 @ 20]  
    ]  
] forkAt: 20
```

Hodiny již tedy běží v okně paralelně. Paralelní provádění kódu však s sebou přináší důsledky, které jsou pro programátory zvyklé výhradně na sekvenční zpracování výpočtu nepříjemné. Je to především otázka opravy chyb a ladění, neboť všechny paralelně běžící výpočty nejdou jednoduše jako ty sekvenční zastavit stiskem `Ctrl-Break`. Nezkušeného programátora potom může v opakujících se

intervalech například strašit periodicky se objevující chybové hlášení způsobené nějakým paralelním výpočtem, do kterého však z uživatelského rozhraní nemůže zasáhnout. Přesně z tohoto důvodu jsme do našeho kódu začlenili globální proměnnou `B`, protože změnou její hodnoty na `false` lze snadno zastavit jinak obtížně dosažitelný paralelně probíhající cyklus.

## 3 Pokročilé techniky tvorby softwaru

### 3.1 Vztah mezi informačním a řídicím systémem uvnitř organizace

Podívejme se ještě podrobněji na postavení procesního modelování v kontextu modelu celé architektury organizace. Zjednodušené přístupy k problematice tvorby informačních systémů, které jsou dnes tak oblíbené, předstírají, že informační systém a řídicí systém je totéž (a v nejextrémnější podobě se ještě od některých prodejců softwaru můžeme dovědět, že informační systém je jejich softwarový produkt, který nám právě nabízejí). Taková zjednodušení a nepochopení potom velmi často vedou k velkým problémům které lze popsat následujícím scénářem:

1. Očekává se, že problémy řídicí a organizační povahy, které organizace má, musejí být řešeny vybudováním či nákupem „nového informačního systému“ (nebo jeho nové komponenty). = *Příčina problémů v řídicím systému se nesprávně chápe jen jako nedostatečné vybavení informačními technologiemi.*
2. Očekává se, že nově zavedené technologie „vylepší“ stávající řídicí a organizační struktury. = *Organizace se vědomě či nevědomě vyhýbá reorganizaci stávajících řídicích struktur a snaží se je zachovat beze změny a jen doplnit o nové technologie.*
3. Nový software neslouží podle očekávání – může se dokonce stát, že se celý **řídicí systém** natolik zkomplikuje, že je dokonce méně efektivní, než byl dříve. Tato skutečnost se samozřejmě tají a navenek se předvádí, jak je nový software moderní a funkční a jak dokonalý **informační systém** organizace má.

Při zavádění informačních technologií by si měl řídicí pracovník uvědomit, že vztah mezi informačními technologiemi, informačním systémem a řídicím systémem je složitější než výše uvedená nesprávná představa. Tento vztah lze popsat následovně:

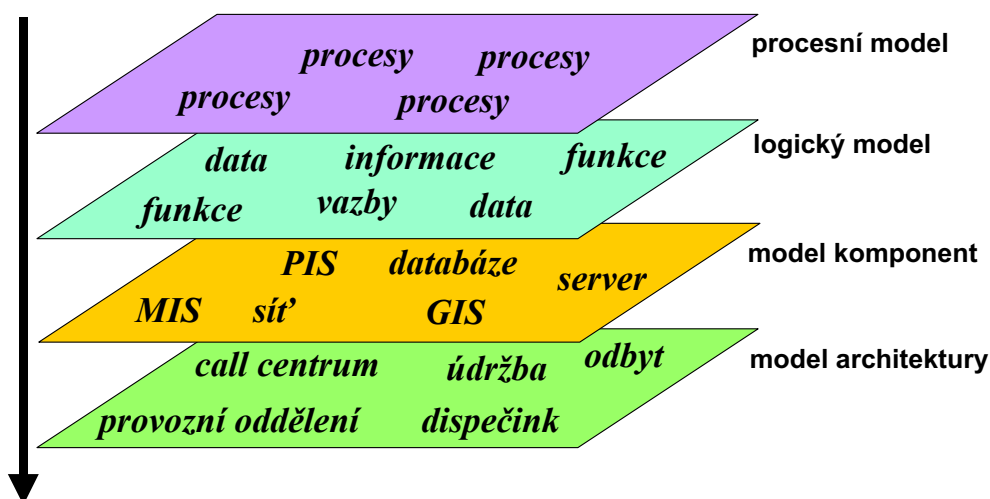
#### informační systém

- = **informační technologie** (programy, počítače, síť, ...)
- + zabezpečení (správci systémů, údržba systémů, zajištění bezpečnosti informačních systémů, zajištění kvality, ...)
- + uživatelé (jaké služby jim systém poskytuje a také co od nich vyžaduje)

#### řídicí systém

- = organizační struktura
- + pravidla, řídicí funkce, kompetence, ...
- + **informační systém.**

Informační systém je jen jednou součástí řídicího systému organizace. Analýza a návrh řídicího systému organizace je složitou záležitostí. Podle většiny objektově orientovaných metodik a také v souladu s konvergenčním přístupem se doporučuje na jeho model nahlížet čtyřmi různými způsoby (viz. obrázek).



1. Prvním možným úhlem pohledu je úroveň procesů. Pod **procesním modelem** organizace si můžeme představit množinu vzájemně souvisejících modelů procesů, které dohromady popisují vše, co se v organizaci děje.
2. Jinou možností, jak úplně popsat organizaci je **logický model**. Logický model popisuje data, funkce a pravidla. Při použití objektového přístupu lze k tomuto popisu použít konceptuální objektové diagramy – například diagramy objektů a tříd, stavové diagramy a nebo diagramy objektových komunikací.
3. Dalším způsobem je sestavení **modelu komponent**. Jedná se sestavení modelu, jehož prvky jsou například konkrétní moduly informačních nebo jiných subsystémů systémů jako například personální informační systém, evidence zásob, GIS, DWH, podnikový intranet apod. Vztahy v tomto modelu jsou vzájemné souvislosti a závislosti vyjmenovaných prvků na sobě.
4. Posledním způsobem, jak sestavit model podniku je **model architektury**. Tento model sleduje skutečnou geografickou lokaci a organizační strukturu. Prvky tohoto modelu jsou například provozní oddělení, podnikové výpočetní středisko, zákaznické centrum apod.

Všechny čtyři přístupy k sestavení modelu řídicího systému organizace mohou vést k jeho úplnému popisu, ale pokaždé jiným způsobem. Jejich prvky jsou samozřejmě vzájemně provázány, ale rozhodně tu neplatí nějaké jednoduché vzájemně jednoznačné zobrazení napříč úrovněmi. Například jednomu elementu z modelu komponent odpovídá v modelu architektury více prvků, jeden prvek z logického modelu má vztah k více procesům atp.

Význam komponentového pohledu na model řídicího systému organizace je v tom, že tu platí silná vertikální závislost mezi dílčími modely přičemž vše primárně závisí na procesech, což bylo potvrzeno zkušenostmi s velkými projekty. Změna v procesech má vliv na změnu v logické architektuře, ta zase na komponenty a ty nakonec mohou mít vliv na architekturu. Tuto závislost je třeba respektovat.

Dochází-li ke změně na jakékoliv úrovni, tak je vždy třeba provést rozbor dopadů této změny na úrovně vyšší postupně až k procesům. Například rozhodnutí vedení firmy „postavíme si nové výpočetní středisko“ nebo „v budově XY uděláme zákaznické oddělení“, které se týká úrovně architektury, má smysl pouze tehdy, víme-li jaké subsystémy z nadřazené úrovně tento zásah vyžadují, jaké logické důsledky z další nadřazené úrovně to přinese a kterých procesů z nejvyšší úrovně se bude změna týkat, tedy zda procesy změníme, vylepšíme, zrušíme nebo nastavíme nové. Podobné nebezpečí na úrovni komponent v sobě skrývá například záměr „začneme používat GIS“, zavedeme „webový portál na internetu“ atd.

Jestliže se totiž rozhodnutí provádějí pouze na úrovni, kam prvek patří, tak reálně hrozí, že jediné, co změna přinese, jsou zbytečně vyhozené peníze.

## 3.2 Modelování požadavků na informační systémy

Při práci na velkých projektech se analytici informačních systémů setkávají s problémem, kdy při startu projektu nejsou známy všechny požadavky na systém a od zákazníka se očekává, že jejich nalezení a upřesnění bude až součástí projektu. Celá záležitost je ještě o to složitější, protože funkčnost budovaných rozsáhlých systémů má vliv i na vlastní organizační a řídicí strukturu podniku nebo organizace, kam se systém zavádí – jsou to například nové či pozměněné pracovní funkce, změna řízení, nová oddělení atp. Proto je žádoucí se při práci na informačních systémech zabývat i změnou těchto souvisejících struktur. A právě procesy a procesní modely jsou ověřenou a v praxi používanou metodou pro analýzu, návrh a implementaci organizačních změn za aktivní spoluúčasti zadavatelů (interview, workshopy, ...).

Z objektově orientovaného procesního modelu lze dobře s aktivní pomocí zadavatelů najít a) funkce, b) strukturu, c) rozsah požadovaného systému a d) také role budoucích uživatelů vytvářeného systému. Získání této informace z interview také neklade na zadavatele extrémní nároky na detailní znalosti technik softwarového inženýrství.

Běžně používané metody tvorby softwaru, ať už jsou či nejsou objektově orientované, se však bohužel touto problematikou příliš nezabývají (např. SA/SD, OMT, UML) a spoléhají na to, že hranice systému, jeho požadovaná funkčnost a role jeho uživatelů jsou známy na počátku projektu a že se v průběhu projektu nebudou měnit.

### 3.2.1 Myšlenka konvergenčního inženýrství

Objektová technologie může naštěstí poměrně jednoduše modelovat jak softwarové systémy, tak i systémy podnikové či organizační, které můžeme souhrnně podle některých autorů nazvat jako systémy sociotechnické. Právě proto, že jedna technologie slouží k modelování obojího, tak není nemožná myšlenka modelovat podnikový a informační systém ne jako systémy dva, ale jako systém jeden a změny a vlastnosti procesů přímo promítat do změn a vlastností softwaru. Tento přístup je podrobně popsán například v publikacích Davida Taylora.

Klasický přístup, kdy je ostrá hranice mezi podnikovým systémem a informačním systémem, vede při změnách a reorganizacích v podniku ke komplikovaným zásahům do konstrukce softwaru, což v mnohých případech je natolik nepružné a nákladné, že může vedoucí pracovníky od procesu změny odradit. Výsledná kombinace struktur je potom naneštěstí extrémně odolná ke změnám a požadavek praxe potom nesprávně preferuje konzervativní a neadaptovatelné informační systémy.

Fakt je, že dříve byly přístupy k budování informačních systémů a k budování podnikových či organizačních systémů chápány jako dvě zcela odlišné činnosti a pokud vůbec měly nějakou souvislost, tak se jednalo o totální vztah podřízenosti informačního systému na organizačním systému. V dnešní době to vede k velkým problémům s nesouladem v návrzích podniku/organizace a rozvojem informačních technologií.

Konvergenční inženýrství, které využívá myšlenky objektového přístupu, přináší následující velké výhody:

1. Zjednodušuje celý proces analýzy a návrhu a snižuje celkovou spotřebu práce, neboť se buduje jen jeden systém namísto dvou.
2. Řeší strukturální nesoulad mezi podnikovými procesy a jejich podpůrnými komponentami informačního systému.
3. Souvislosti s řízením a organizací podniku a strukturou informačního systému jsou srozumitelné, informační systém je lépe realizovatelný.
4. Usnadňuje problémy a náklady spojené s návrhem prováděním změn, což vede k adaptivnější organizaci.

Objektový přístup je sice předpokladem, ale samotná technika, ani drahý CASE nástroj zde nestačí. Pokud při modelování neexistuje vedoucí pracovník s vizí, který je schopen a ochoten změny prosadit, tak není možné projekt úspěšně realizovat a zahájení projektu je ztrátou času a peněz. Této zodpovědnosti se vedoucí pracovníci nemohou jednoduše zbavit tím, že objednájí drahé konzultační služby a jmenují do funkce vedoucího projektu podřízeného pracovníka, u kterého není v souladu zodpovědnost a povinnost s potřebnou mírou autority a pravomocí.

### 3.3 Životní cyklus vývoje informačního systému

V klasických metodách návrhu se rozeznávají dva základní modely pro popis životního cyklu programového díla od zadání, přes analýzu, návrh, k implementaci, ladění, testování, provozu a údržbě. Jedná se o tzv. "vodopádový model" nebo o tzv. "spirální model" publikovaný poprvé v 80. letech Boehmem. Vodopádový model předpokládá sérii navazujících činností (zadání, analýza, návrh, implementace, dodání, údržba) které jdou jedna po druhé od začátku do konce. Vzhledem k vlastnostem OOP, je pro tvorbu objektových aplikací vhodnější spirální model, ve kterém se všechny činnosti cyklicky opakují, vždy jakoby na vyšší úrovni spirály. Počet cyklů nepřímo závisí na tom, do jaké míry se jedná o problém, který je variantou známého již vyřešeného zadání.

Jeden cyklus spirálního modelu v OOP má fáze zadání, analýzy, návrhu, implementace, testování a provozu. Fáze zadání a analýzy se dohromady označují jako **stadium expanze**, protože při nich dochází ke hromadění informací potřebných pro vytvoření aplikace. Stadium expanze končí s dokončením analytického konceptuálního modelu, který na logické úrovni reprezentuje požadované zadání a formálně popisuje řešený problém.

Zbývající fáze od návrhu přes implementaci k testování a provozu se označují jako **stadium konsolidace**. Je tomu tak proto, že v těchto etapách se model, který je produktem předchozí expanze, postupně stává fungujícím programem, což znamená, že na nějakou myšlenkovou "expanzi" zadání zde již není prostor ani čas. (V tomto stadiu se také počítá s tím, že od některých idejí z expanzního stadia bude třeba upustit vzhledem k časovým, kapacitním, implementačním nebo i intelektuálním schopnostem - odtud tedy název tohoto stadia.)

Na životním cyklu v OOP je také zajímavý samotný programový produkt, který je v OOP vždy považován za jakýsi prototyp, neboť může kdykoliv posloužit jako součást nového zadání při nastartování dalšího vývojového cyklu. V OOP tedy neznáme prototypy v klasickém pojetí, protože každý produkt je úměrně k zadání, ze kterého vyšel, funkční a může být proto použit (v počítačové angličtině se označuje jako "deliverable"). Stejně však také může sloužit pro tvorbu nové verze produktu - a to nejen skrze zkušenosti s ním, ale přímo i svým kódem jako kostra či výchozí model další expanze. Vzhledem ke vztahu předchozí a následné verze nebývá velkým problémem jejich přechod za plného provozu systému, pokud je programován v čistě objektově orientovaném prostředí.

I když objektový spirální model přináší více volnosti do životního cyklu aplikace než klasické modely, přesto i zde platí některá omezení. Nejdůležitější z nich je skutečnost, že není možné jednu část cyklu řešit pomocí klasické techniky a nástrojů a jinou pomocí objektové techniky a nástrojů. Střídání klasických a objektových technik podle našich zkušeností přináší více škody než užitku.

#### 3.3.1 Objektová analýza a návrh

Objektově orientovaný přístup v metodologiích analýzy a návrhu informačních systémů je založen na teorii objektového datového a výpočetního modelu. Modelujeme-li pomocí nějaké objektové techniky, tak se vždy snažíme najít množinu - systém objektů, které popisují jednotlivé části z modelované problémové oblasti. Tyto objekty spolu komunikují pomocí **posílání zpráv**. Mezi jednotlivými objekty jsou modelovány hierarchie **skládání, dědění, závislosti a delegování**.

V objektově orientované analýze nedochází při tvorbě modelu ke vzájemnému odtržení datového a funkčního pohledu na modelovaný systém. V ideálním případě je dokonce možné vystačit s jedním

nástrojem (jeden typ diagramu) a s jednou základní technikou. V případech, kdy to možné není, si však stále jednotlivé modely, i když jsou zaměřeny na různé části problému, zachovávají mnohem větší stupeň vzájemných souvislostí, než je tomu u klasické analýzy.

Lze říci, že analýza v OOP končí tehdy, je-li řešený problém popsán mj. v příslušném konceptuálním modelu, který úplně a formálním způsobem popisuje zadání daného problému.

Návrh je v OOP přímým pokračováním analýzy bez nějakého extrémně výrazného přechodu. Na rozdíl od klasického návrhu se zde v maximální míře používají stejné nástroje, jaké sloužily i v analýze. Konceptuální modely, které ve fázi analýzy popisovaly řešený problém (tj. zadání), se ve fázi návrhu rozpracovávají do podoby použitelné pro jejich počítačovou implementaci.

Vzhledem k povaze kódu objektového programu OOP nezná ani ostrý přechod mezi fází návrhu a fází implementace. V případě použití vyspělých objektových programovacích nástrojů (vývojových prostředí) a programovacích jazyků lze považovat psaní kódu programu za ukončení návrhu na nejnižší úrovni granularity. Takto viděný objektový program je potom vlastně stále model, který je však rozpracován do takové podrobnosti, že jeho jednotlivé elementy jsou totožné s výrazovými prostředky použitého programovacího prostředí. V klasickém pojetí je mezi návrhem a mezi kódováním programu velmi ostrý přechod, který je dán na jedné straně počtem a vlastnostmi jednotlivých používaných modelů a na straně druhé používanými programovými prostředky a požadavky na podobu, kódování a ladění klasického programu.

Pro ulehčení procesu vytváření jednotlivých objektových modelů byl navržen tzv. „**multi layer - multi component model**“, poprvé publikovaný v pracech Coada a Yourdona, jehož podobu lze v nejrůznějších formách najít i v jiných metodologiích. Vícevrstevný (*multilayer*) model je založen na tvorbě objektového modelu, metodou postupného doplňování od vrstvy **subjektů**, přes vrstvu **objektů**, vrstvu **struktury** až k vrstvě **služeb**, které se vzájemně prolínají a doplňují. Cílový objektový model je tvořen souhrnem struktur z jednotlivých vrstev. Komponenty, které představují druhý rozměr členění systému, jsou relativně samostatné objektové modely (vytvořené metodou více vrstev). Každá z komponent popisuje řešený problém z jiné stránky. Zpravidla i v té nejjednodušší aplikaci můžeme vždy rozeznat komponentu **vlastního problému** (problem domain) a komponentu **uživatelského rozhraní** (human interaction). Právě důslednost v rozlišení komponent problému je téměř vždy přímo úměrná celkové kvalitě výsledného programu. Kromě uvedených komponent je možné podle typu řešené úlohy ještě rozeznat komponentu **zpracování dat** (data management) a **řízení úloh** (task management).

V OOAD se doporučuje na rozdíl od klasické data-flow analýzy problém nestrukturovat pouze vertikálně (jako jsou např. jednotlivé vrstvy DFD), ale využívat také možnosti horizontálního členění. Při horizontálním členění je systém členěn na tzv. **funkční cesty** neboli **procesy**, které jsou postupně **odděleně rozpracovávány**. Metoda funkčních cest aplikovaná na objektovou analýzu řeší nedostatky klasické analýzy, jako například odtržení funkcí od datové struktury a ztrátu kontextu při jednotlivých rozkladech abstraktních procesů.

I když je tvorba objektového modelu usnadněna jeho rozdělením do oddělených vrstev a komponent, tak se stále jedná o velmi obtížný a přitom klíčový problém. Zatím není k dispozici jednoznačný návod, který by posloužil k efektivní a optimální transformaci zadání problému do formální podoby modelu. Velkou roli zde hraje zkušenost návrháře a jeho schopnost komunikovat se zadavateli. Jedním z největších problémů je **rozpoznání procesů a jejich objektů včetně jejich požadovaných vlastností**. K řešení těchto potíží byly a jsou vyvíjeny různé různé složité techniky. Tou nejjednodušší a také nejméně účinnou je návod, vycházející z jazykové analýzy zadání (podstatná jména - objekty, slovesa - metody). Mezi sofistikovanější metody patří například:

- **Object Behavioral Analysis**, která slouží k získávání prvotní představy o esenciálním objektovém modelu jako soustavy vzájemně komunikujících objektů v různých procesech. Touto technikou může začínat celý životní cyklus systému.
- **CRC karty** (Class-Responsibility-Colaborators), pomocí kartiček se jmény třídy, její odpovědností (tj co dělá) a spolupracujícími třídami, hledáme v rámci interakce s uživateli nové třídy.

Podcení-li se tyto metody a je-li návrh ovlivněn neobjektovými technikami, dochází při stanovování struktury objektového modelu například k následujícím chybám:

- **Podcenění možností skládání objektů** a i jiných hierarchií na úkor **přeceňování dědičnosti**. Je to způsobeno tím, že dědění je pojmem novým, a proto se na něj v literatuře zabývající se výkladem OOP klade větší důraz, než na skládání, které již dříve bylo určitým způsobem využíváno.
- **Podcenění možností metod**, které vede k jejich omezení na pouhou manipulaci se složkami objektů (metody pouze typu "ukaž" a "nastav"). Vzhledem k provázanosti datové a funkční stránky v OOP je velmi vhodné s některými metodami počítat přímo v návrhu datové struktury a tím ušetřit na objektových vazbách a datových atributech objektů.

### 3.3.2 Současné objektové metodologie

Přibližně od roku 1990 bylo doposud vyvinuto několik vzájemně odlišných metodologií pro objektově orientovanou analýzu a návrh. Mezi dnes nejpoužívanější metodologie patří:

- **OMT - Object Modelling Technique**, jejímž autory jsou J. Rumbaugh, M. Bláha, W. Premerlani, F. Eddy a W. Lorensen. Metoda byla poprvé publikována nakladatelstvím Prentice Hall v knize "Object-oriented Modelling and Design" v roce 1991. Technika je zajímavá tím, že je v podstatě hybridní technikou, zahrnující jak objektové, tak i klasické nástroje. Nejčastěji se používá pro návrh databázově orientovaných aplikací s objektovým klientem a relačním serverem.
- **Coad-Yourdon Method**, jejímiž autory jsou P. Coad a E. Yourdon (autor již zmiňované klasické strukturované metody). Metoda byla poprvé publikována v časopise American Programmer v roce 1989 a posléze v knihách obou autorů nakladatelství Yourdon Press. Od počátku je k dispozici jako stejnojmenný CASE prostředek podporující jazyky Smalltalk a C++. Z uvedených technik je nejsnazší co do počtu používaných pojmů.
- **OOSD - Object-Oriented Software Development**, jejímž autorem je G. Booch. Metoda byla poprvé publikována v roce 1991 v knize nakladatelství Benjamin/Cummings "Object-Oriented Design with Applications". Tato poměrně velmi komplexní metodologie je určena pro týmový vývoj rozsáhlých aplikací v jazyce C++ a Smalltalk, a ze všech vyjmenovaných metod pokrývá nejvíce objektových vlastností.
- **OOSE - Object-Oriented Software Engineering** autora Ivara Jacobsona. Metoda je velmi kvalitně popsána ve stejnojmenné knize. (Alternativní název metody je „**Objectory**“). Vzhledem k tomu, že metoda má původ ve skandinávské škole, je velmi zajímavá pro aplikace z oblasti simulace a řízení. Jacobsonova metoda se jako první začala zabývat problematikou získávání a modelování informací v prvních fázích životního cyklu ještě před budováním konceptuálního diagramu. Úvodní technika této metody - „**use case**“ byla adoptována i do ostatních objektových metodologií.
- **Unifikovaný proces (UP)** (někdy označovaný také jako USDP – Unified Software Development Process) vznikl jako preferovaná metodologie pro vývoj informačních systémů pomocí UML (aby ne, vždyť je od jeho autorů). UP vznikl zejména na základě metodologií Erickson a Objectory a prací autorů I. Jacobsona, G. Boocha aj. UP je zejména sbírkou postupů jaké lze použít při objektově orientované analýze a návrhu – pro každý typ projektu se musí vybrat patřičné nástroje. UP je obecným otevřeným standardem a na jeho základě jsou vytvářeny komerční varianty. Nejznámější variantou je RUP (Rational Unified Process) firmy Rational (v ní jsou zaměstnáni tvůrci UML a UP).
- Metoda **Martin-Odell** autorů J. Martina (spolu s E. Yourdonem známý i z dřívějšího období) a J. Odella. Metoda je publikována v sérii knih nakladatelství Martin Books, a představuje podobně jako unifikovaná metoda pokus o sjednocení dosavadních objektových zkušeností předchozích metod. Používá velké množství nejrůznějších diagramů a pojmů.

- **Object-Oriented Structured Design notation** autorů A.I. Wassermanna, P.A. Pirchera a R.J. Mullera, publikovaná poprvé v časopise IEEE Computer č. 23(3) 1990. Metoda je zajímavá především notací používaných diagramů, která ovlivnila následné metodologie.
- **OOER notation** autorů K. Gormana a J. Choobineha, poprvé publikovaná na 23. mezinárodní konferenci o informatice (computer science) na Havaji v létě 1991. Metoda používá notaci ER diagramů v klasické Chenově syntaxi, rozšířeně o objektové vlastnosti. Je výhodná pro použití v objektově orientovaných databázích.

Pro výše uvedené metodologie jsou dnes k dispozici i CASE prostředky podporující především jazyky Java, C++ a Smalltalk. Jejich společnou nevýhodou je však skutečnost, že ani jedna z nich není uznávaným standardem, i když se dnes UML již prosadil jako standard pro zápis modelů. Podle statistických výzkumů v USA a Velké Británii z 90. let ani jedna není používána tvůrci objektových programů z více jak 20%. Způsobuje to jednak nedostatek zkušeností s nimi, protože se stále jedná vlastně o první generaci objektových metodologií, a také, že ani jedna nepodporuje v dostatečné míře všechny možnosti, které nabízí OOP a především čisté objektové jazyky a objektové databáze. Všechny důvody zřejmě zmíněným metodám zatím zabránily stát se všeobecně uznávaným standardem. Žádná z uvedených metod zatím není na takové úrovni, aby podporovala všechny žádoucí objektové vlastnosti s minimálně stejnou mírou elegance a jednoduchosti a plnila takovou roli, jako dříve klasická strukturovaná Yourdonova metoda.

Kromě výše uvedených metodologií existuje ještě několik vesměs velmi kvalitních technik nebo i jen nástrojů, jejichž rozsah je však omezen na několik publikací nebo na výuku softwarového inženýrství na vysokých školách, anebo se jedná o firemní know-how.

Výčet zmínek objektových přístupů je dlužno uzavřít odkazem na metodu BORM, která se snaží o komplexní vyvážený přístup ke všem fázím návrhu a realizace programového systému tak, aby vyloučila nebo eliminovala v nejvyšší možné míře zmiňované nedostatky a slabiny ostatních metod (J. Polák - KP FEL ČVUT a Deloitte&Touche, V. Merunka - KI PEF ČZU a KP FEL ČVUT, R.P.Knott - DCS Loughborough University). Metoda BORM (*Business Object Relation Modelling*), byla vyvíjena na základě projektu VAPPIENS a s podporou firmy Deloitte&Touche.

### 3.4 Řízení vývojového týmu

Práce na větším softwarovém celku musí být určitým způsobem řízena, má-li být efektivně dosaženo zamýšlených cílů. Metodika řízení projektů je popsána v množství odborné literatury. Řízením projektu se zabývá **manažer**, jehož hlavní směry činnosti jsou následující:

1. **Komunikace se zadavateli.** Manažer klade návrhy na softwarové projekty různým klientům. Každý takový návrh musí obsahovat odůvodnění a časové a finanční nároky.
2. **Plánování projektu.** Úkolem manažera je vypracovat harmonogram projektu a stanovit pravidla, kterými se bude projekt řídit včetně jednotlivých etap a jejich návaznosti. Je třeba zajistit způsob validace produktu, různé konfigurace, nároky na zaškolení a údržbu.
3. **Řízení rozpočtu** je jednou z klíčových rolí manažera. Patří sem mimo náklady na vlastní tvorbu i náklady např. na techniku, školení a cestování.
4. Úkolem manažera je rovněž **výběr osobností a týmů** pro řešení projektu a jeho složek. Při výběru je třeba respektovat složitost požadavků i rozpočet projektu. V literatuře se doporučuje preferovat do klíčových pozic řešitelských týmů skutečné špičky bez ohledu na vyšší náklady.
5. **Kontrola stavu projektu a schopnost reagovat na problémy**, které vznikají při samotné tvorbě díla.
6. **Prezentace výsledků** během projektu a na konci projektu jak zadavateli, tak i uživatelům.

### 3.4.1 Softwarové profese

Kromě manažera vyžadují jednotlivé etapy vytváření programového celku další různé typy profesí. Některé profese je možné pokrýt jedinou osobou, ale existují i kombinace profesí, které představují ohrožení úspěšného dokončení daného projektu. Při organizaci větších týmů může být dokonce účelné zavedení celých oddělení jednotlivých profesí a využití externích specialistů. Jedná se o následující profese (*Z historických důvodů se uvedené profese vesměs nazývají „programátor“*):

1. **Vedoucí programátor**, který musí být schopen vytvářet jak specifikaci úlohy, tak i návrh implementace a řízení programátorských prací. Na jeho kvalitách nejvíce závisí úspěch celé tvorby.
2. **Ideový programátor**, který také musí být špičkovým profesionálem a také zabývá se vedením prací. Na rozdíl od vedoucího programátora jsou pro tuto profesi důležité především nápady a nikoliv detaily realizace.
3. **Systémový programátor** pro kterého je podstatná podrobná znalost systémového prostředí pro vytvářený produkt.
4. **Výkonný programátor** se zabývá vlastní přípravou programů na základě dodané specifikace.
5. **Specialista na jazyk/systém** jako odborník na konkrétní implementační prostředek který poskytuje ostatním řešitelům např. konzultace.
6. **Testér**.
7. **Oponent**.
8. **Dokumentátor** který je zodpovědný za dostatečnou dokumentaci všech fází projektu.
9. **Knihovník** je osoba, která zajišťuje správu softwarových knihoven. Přejímá od řešitelů hotové komponenty a zajišťuje zpřístupnění komponent ostatním. Udržuje rovněž různé verze.
10. Pracovníci zajišťující **technické služby a administrativu**.

Na straně zadavatele je třeba ještě mít **gestora/kontaktní** osobu, která ručí za uzavření etap projektu – je to pojistka, že projekt jde správným směrem a že se nestane, že zadavatelé na konci vývoje řeknou, že to chtěli jinak. Při tvorbě informačních systémů je skutečně veliký rozdíl mezi tím, kdy zadavatel řekne „to nechci“ (během projektování) a nebo „to jsem nechtěl“ (nad dokončeným systémem). Jedna ze základních pouček softwarového inženýrství totiž říká, že změna funkčnosti provedená po dodání systému vyjde o několik řádů draž než pokud by byla provedena během prvních fází projektu.

### 3.4.2 Organizace pracovních týmů

Při tvorbě softwaru je nutná spolupráce řešitelů. V některých pramenech se uvádí, že vzájemná komunikace řešitelů vyžaduje až přes 50% celého času. Přiměřené organizaci týmu je proto třeba věnovat dostatečnou pozornost. Pracovní týmy se dělí na základě jejich organizace na

**Nestrukturované** (dělba práce dle objemu)

- **Osamělí vlci** - skupina individualit, kteří dokáží odděleně řešit jednotlivé **úlohy**.
- **Horda** - komunikující tým který potřebné práce libovolně rozděljuje podle objemu.
- **Demokratická skupina** - k rozdělení práce dochází na základě dohody, kde všichni se disciplinovaně snaží přispět k výslednému efektu.

Nestrukturované týmy mohou optimálně využívat existující kapacity. Ohrožení nestrukturovaných týmů spočívá v nestejně míře zodpovědnosti jednotlivých členů za výsledek práce.

#### Strukturované týmy (dělbá podle profese)

- **Chirurgický tým** - v týmu je vše podřízeno rozhodování vedoucího programátora, který je zároveň ideovým programátorem. Ostatní poskytují služby podle svých profesí.
- **Tým hlavního programátora** - funkce vedoucího a ideového programátora (popř. dalších funkcí) jsou oddělené. Vedoucí programátor přiděluje a řídí práce ale členové týmu se v jednotlivých situacích opírají o další profese, které v těchto situacích nejsou totálně podřízeny vedoucímu.
- **Vícetýmová organizace** - jednotlivé složky týmu mohou být složeny ze všech uvedených typů.

### 3.4.3 Algoritmizace rozpočtu

Nejopomíjenější složkou rozpočtu a zároveň složkou, která **rozpočet nejvíce ovlivňuje a je sama značně ovlivňována použitou technologií**, jsou závěrečné etapy projektu, především jeho **údržba**. Odhaduje se, že minimálně 50% času tvůrců software představuje čas strávený nad údržbou existujících produktů. Údržbu lze podle metody COCOMO dále rozdělit na

- opravu chyb - asi 17%
- přizpůsobování softwaru novému prostředí - asi 18%
- vylepšování nových verzí - asi 65%

V modelu COCOMO se používá pro **roční** odhad nákladů na údržbu následující vzorec:

$$M = Z \cdot E ,$$

kde **M** jsou náklady na údržbu v člověkoměsících (*man-month*), **Z** je předpokládaný koeficient změny (typicky 10-30%) a **E** jsou náklady na tvorbu v člověkoměsících. Náklady **E** lze stanovit na základě výrazu

$$E = k \cdot R^n ,$$

kde **R** je rozsah produktu v tisících instrukcí programu a **k** a **n** jsou konstanty jejichž hodnoty se odvozují z hotových projektů a velmi se liší pro různé typy projektů. Model rozeznává základní tři typy projektů:

- zcela nová řešení (**k** = 3.6 , **n** = 1.2),
- varianty známých řešení s novými prvky (**k** = 3 , **n** = 1.12) a
- varianty známých řešení (**k** = 2.4 , **n** = 1.05).

Na základě **pracovní náročnosti E** v člověkoměsících se stanovuje **časová náročnost projektu T** v měsících. Výpočet vychází z předpokladu, že limity personálních zdrojů neovlivňují časový průběh řešení a k dispozici je vždy potřebný počet řešitelů. Vzorec pro časovou náročnost má tvar

$$T = 2.5 \cdot E^m ,$$

kde  $\mathbf{m}$  je konstanta pro různé typy úloh ( $\mathbf{m} = 0.32$  pro nová řešení,  $\mathbf{m} = 0.35$  pro střední typ úlohy a  $\mathbf{m} = 0.38$  pro variantní řešení). Potřebný počet  $\mathbf{N}$  řešitelů je potom určen jako

$$\mathbf{N} = \mathbf{E} / \mathbf{T} .$$

Na základě uvedených vztahů lze určit překvapivě nízkou odhadovanou produktivitu práce programátora na 4 až 16 instrukcí na osobu a den dle obtížnosti projektu. Obtížnost  $\mathbf{E}$  projektu je doporučováno volit v závislosti např. na použité technologii upravovat pomocí upřesňujících koeficientů  $\mathbf{x}_i$  jako

$$\mathbf{E} = \mathbf{k} \cdot \mathbf{R}^n \cdot \prod_i \mathbf{x}_i ,$$

kde příslušná úprava může představovat až násobky či zlomky (až 5× není výjimkou) původní hodnoty. Doporučované hodnoty jednotlivých koeficientů  $\mathbf{x}_i$  jsou následující:

1. **Požadovaná spolehlivost** produktu, která je přímo úměrná obtížnosti řešení v rozsahu 0.75 až 1.4.
2. **Požadovaná adaptabilita na změny prostředí**, která je přímo úměrná obtížnosti řešení v rozsahu 0.87 až 1.3.
3. **Rozsah použitých databází**, která je přímo úměrná obtížnosti řešení v rozsahu 0.94 až 1.16.
4. **Složitost produktu** ve smyslu požadavků na speciální vstupy a výstupy či algoritmy zpracování, která je přímo úměrná obtížnosti řešení v rozsahu 0.7 až 1.65.
5. **Časová náročnost řešení** (např. míra požadavků na časové odezvy na hranici možností použité techniky která vyžaduje detailnější algoritmy), která je přímo úměrná obtížnosti řešení v rozsahu 1.0 až 1.66.
6. **Paměťová náročnost řešení** (např. velikost požadované paměti na hranici možností použité techniky která vyžaduje detailnější algoritmy), která je přímo úměrná obtížnosti řešení v rozsahu 1.0 až 1.56.
7. **Dostupnost techniky pro řešení** (zda je technika k dispozici nebo zda se ještě vyvíjí), která je přímo úměrná obtížnosti řešení v rozsahu 0.87 až 1.15.
8. **Schopnost řešitelského týmu**, která je nepřímo úměrná obtížnosti řešení v rozsahu 1.46 až 0.71.
9. **Míra znalosti řešené aplikace**, která je nepřímo úměrná obtížnosti řešení v rozsahu 1.29 až 0.82.
10. **Míra znalosti použitých programovacích prostředků**, která je nepřímo úměrná obtížnosti řešení v rozsahu 1.21 až 0.9.
11. **Výkonnost použitých programovacích prostředků**, která je nepřímo úměrná obtížnosti řešení v rozsahu 1.24 až 0.83.
12. **Míra využívání moderních programovacích technik**, která je nepřímo úměrná obtížnosti řešení v rozsahu 1.24 až 0.82.
13. **Požadavky na zkrácení časového rozvrhu oproti odhadu**, která je nepřímo úměrná obtížnosti řešení v rozsahu 1.23 až 1.0.

### 3.5 Alternativní metody řízení projektů

Pokrok hardwaru a softwaru posledních let umožnil značný rozvoj možností integrovaných vývojových prostředí. Pro všechny rozšířené jazyky (ponejvíce Java, C++, Smalltalk, Object Pascal, ...) je dnes k dispozici řada komplexních vývojových prostředí se širokou škálou možností. Programátor tak má k dispozici velké množství pokročilých funkcí pro správu kódu: vizuální programování, nástroje pro

podporu společného vlastnictví kódu (správa verzí), refaktoring (viz přílušná kapitola), automatické testování a mnoho dalších. Toto vše poskytuje potenciál pro růst produktivity programátorů. Tento růst je však do určité míry **omezen tradičním způsobem řízení procesu tvorby softwaru**.

### 3.5.1 Vývoj pohledu na softwarové projekty

Za poslední desítky let se značně měnil charakter softwarových projektů. Nebývalý **nárůst výkonu osobních počítačů** umožnil jejich využití v nových oblastech. Obrovské **zlevnění výroby** pak umožnilo rozšíření výpočetní techniky do všech oborů lidské činnosti a zpřístupnění široké veřejnosti. Změna chápání softwarových projektů spočívá především v těchto bodech:

1. **Možnosti výpočetní techniky.** Do nedávna byl jediný možný postup práce zdrojový kód modulu – překlad modulu – sestavení s ostatními moduly – spuštění aplikace, což je typické např. pro jazyk C, assembler, atd.
2. **Typ projektů.** Komplexní informační jako dnešní typický „velký“ softwarový produkt se stal běžným v průběhu posledních let. Zpočátku byly typickými projekty vědecké aplikace pro výpočty, aplikace průmyslového řízení, atd.
3. **Cílová skupina.** Dříve byl typickou cílovou skupinou vědecký institut, vládní organizace, armáda, atd. S masovým rozšířením výpočetní techniky jsou však dnes nejčastějšími zákazníky nejrůznější podniky a instituce od nejmenších až po mezinárodní.
4. **Průběh života hotového softwaru.** Zatímco ve vědeckém výpočetním systému se během týdne nespíš nezmění vzorečky, na kterých je založen, detaily způsobu fungování firmy se typicky mění velmi často. Vznikají tak nové požadavky na stávající software. Druhým aspektem tohoto bodu je obecně vzrůstající tempo rychlosti života a změn současné společnosti, které dříve nebylo tak výrazné.

Softwarové inženýrství jakožto souhrn postupů a pravidel pro organizované efektivní vytváření softwarových produktů je tedy nutně ovlivněno těmito faktory.

### 3.5.2 Tradiční přístup k řízení

Tradiční nebo klasické metodiky metodiky řízení softwarových projektů se vyznačují těmito kroky:

1. specifikace (neformální zadání),
2. analýza s dekompozicí na části (formální zadání)
3. návrh (moduly, uživatelské rozhraní, ...),
4. implementace částí,
5. ladění částí,
6. integrace,
7. testování celku,
8. nasazení,
9. údržba.

Bod 4 a částečně i bod 6 je prováděn tímto postupem:

- a) Psaní zdrojového kódu.
- b) Překlad zdrojového kódu + event. sestavení. V případě chyby při překladu návrat k a).
- c) Testování funkčnosti. V případě chyby návrat k a).

Během celého procesu je maximální snaha na co největší **shodu mezi formálním zadáním**, které je výstupem bodu 2., **a výsledným softwarem**, který je dodán v bodě 8. Platí, že cena chyby v zadání, či cena změny v zadání, řádově roste od bodu 1. k bodu 9. Je proto kladen důraz na přesnost zadání a kvalitu analýzy, která je předpokladem úspěšnosti projektu.

### 3.5.2.1 Omezení tradičního přístupu

Klasický model, z důvodů zmíněných v předcházejícím odstavci, **neumožňuje snadno provádět změny zadání v průběhu realizace projektu**. Vývojové firmy se proto snaží předejít v maximální možné míře této situaci, typicky uzavíráním co nezávaznějších a nejpřesnějších smluv se zákazníkem. Je vyžadováno, aby zákazník měl na počátku velmi **přesnou představu o funkcích a vzhledu výsledného systému**. Pro dnešního zákazníka – firmu – je však toto často velmi problematický požadavek. Nejedná se jen o problém komunikace a přesné specifikace aktuální představy, ale může též dojít během doby vývoje softwaru ke **změně situace u zákazníka**, se kterou na počátku nepočítal. Při klasickém modelu je doba dodání systému, i přes používání moderních nástrojů, relativně dlouhá, toto riziko je tedy nezanedbatelné. Zvláště významné je potom u dynamicky se rozvíjejících firem.

### 3.5.2.2 Rozsah použití

Klasické metodiky obecně **nejsou omezeny rozsahem projektů a velikostí týmu**. Co se týká charakteru projektů, jsou metodiky opět velmi univerzální, je tedy možné je s úspěchem použít jak u business zakázek, tak u vědeckých, státních, armádních a jiných zakázek. Různé druhy zakázek se budou lišit pouze v důrazu na cenu, rychlost dodání, respektování norem, testů jakosti, atd.

### 3.5.2.3 Požadavky na vývojové nástroje

Klasický model je **zcela nezávislý na implementačním programovacím jazyku a prostředí**. Neklade žádné požadavky na vývojové nástroje. Moderní objektově orientované programovací jazyky a vývojové nástroje však samozřejmě výrazně urychlují vývoj i testování a mají blahodárný vliv na produktivitu a snížení chybovosti programátorů.

Klasické metodiky **nevyužívají dostatečně potenciál**, který nabízí moderní vývojová prostředí a přidružené technologie: vývoj systému za běhu, automatizované testování, správa společného kódu, atd. Přesněji řečeno, využívají ho jen v „malém“, tj. v rámci vývoje kusů softwaru, nikoliv však pro zlepšení celého procesu. Co se tímto tvrzením přesně myslí si vysvětlíme v následující sekci.

## 3.5.3 Alternativní metody řízení

Posun v zaměření softwarových projektů vedl k celé řadě **snah o zefektivnění a vyšší flexibilitu při řízení**. V souvislosti s moderními objektově orientovanými jazyky a nástroji, kterými se zde zabýváme, má největší význam rodina tzv. **agilních metodik**. Ty totiž nejlépe využívají jejich možností. Základem agilních metodik je tzv. **Agile Modelling Manifesto**, který definuje určitý obecný soubor principů pro řízení softwarových projektů. Soubor metodik vycházejících z těchto principů se potom souhrnně nazývá „agilní metodiky“.

Všechny předpokládají reálnou situaci, kdy **zákazník zpočátku nemá zcela přesnou představu o výsledném softwaru a při vývoji dochází často ke změnám v zadání**. Jejich cílem je, aby softwarový produkt maximálně splňoval očekávání zákazníka. Je proto **vyvíjen a dodáván po částech** (viz. spirální model) **v pořadí, které si určí zákazník**.

### 3.5.3.1 Řízení projektů metodou extrémního programování

V praxi se nejvíce rozšířila agilní metodika s názvem „*Extrémní programování*“ (zkratka XP). Její poněkud humorně nadsazený název odráží skutečnost, že vychází z několika osvědčených pravidel, které jsou jakoby dotaženy do extrémů. Jsou to<sup>11</sup>:

1. **Plánovací hra (*The Planning Game*)**. Jedná se o fázi, kdy jsou na základě požadavků zákazníka společně naplánovány jednotlivé rámcové funkční celky, ze kterých se systém bude skládat a které budou postupně dodávány. V XP se nazývají *User Stories*, protože mají podobu krátkého slovního popisu.
2. **Malé verze (*Small Releases*)**. Jednotlivé funkční celky dodávané v nových verzích musí být relativně malé, aby je bylo možné dodávat v intervalu řádově týdnů. Čím kratší doba dodání, tím větší jistota, že se mezitím nezměnila situace u zákazníka.
3. **Metafora (*System Metaphor*)**. Zákazník a dodavatel softwaru i vývojový tým uvnitř si musí vytvořit určitou metaforu, pomocí které se budou bavit o systému. Jednoduše řečeno to znamená dohodnout se na „neutrálních“, ne příliš odborných pojmech, kterým budou rozumět všichni, aby bylo možné se bavit o systému.
4. **Jednoduchý návrh (*Simple Design*)**. Všechny funkce by měly být navrženy co nejjednodušeji, aby bylo možné v hotovém systému provádět změny a rozšíření.
5. **Testování (*Continuous Testing*)**. Test je v XP většinou kód, který na určitém modelovém případě ověřuje správnou funkčnost určité části systému. Moderní vývojová prostředí umožňují automatické spouštění testů a jejich vyhodnocování. Testování umožňuje především velmi rychle zjistit, jestli při změně v systému nebo jeho rozšíření nedošlo k narušení funkčnosti.
6. **Refaktorizace<sup>12</sup> (*Refactoring*)**. Pokud má být možné kód měnit podle požadavků a na stávajícím kódu budovat další, je nutné, aby byl „čistý“, tj. jednoduchý a dobře čitelný. Častá refaktorizace (samozřejmě do značné míry automatizovaná moderními nástroji) je proto nutností.
7. **Párové programování (*Pair Programming*)**. Výzkumy ukazují, že dva sešraní programátoři pracující společně u jednoho počítače jsou efektivnější, než kdyby každý pracoval sám. Páry nejsou pevné, ale často se mění.
8. **Společné vlastnictví (*Collective Code Ownership*)**. Všichni programátoři mají přístup k celému kódu všech ostatních a mohou ho měnit a vylepšovat<sup>13</sup>. Neprogramuje se tedy stylem „každý na svém písečku a pak to nakonec dáme dohromady“.
9. **Nepřetržitá integrace (*Continuous Integration*)**. Nové funkce a změny jsou ihned začleněny do systému<sup>14</sup> a během vývoje je tedy systém stále celý funkční.
10. **40-ti hodinový týden (*40-Hour Work Week, nověji Sustainable Pace*)**. Přepřacovaní programátoři nebudou odvádět lepší práci. Časté přesčasby bývají signálem problémů projektu.
11. **Zákazník na pracovišti (*On-site Customer*)**. Zákazník (typicky jeho zástupce) by měl být stále k dispozici programátorům. Umožňuje to okamžité zodpovězení případných dotazů a také lepší kontrolu zákazníka nad vznikajícím produktem.
12. **Standardy pro psaní zdrojového textu (*Coding Standards*)**. Aby mohlo řádně fungovat společné vlastnictví kódu, programátoři se musí dohodnout nad společnými pravidly.

<sup>11</sup> Při studiu následujících principů je konfrontujte s jednotlivými změnami chápání softwarových projektů vyjmenovaných na začátku kapitoly. Rovněž se zamyslete nad vzájemnými vztahy těchto principů a jak se navzájem podporují.

<sup>12</sup> O základech refaktorizace pojednává samostatná kapitola.

<sup>13</sup> Zkuste si rozmyslet, které ostatní body podporují fungování tohoto.

<sup>14</sup> Ale samozřejmě ne do systému, se kterým pracuje zákazník, jen do systému na pracovišti programátorů.

Pozornému čtenáři jistě neuniklo, že především body 5, 6, 8, 9 a 12 využívají možností plně objektového dynamického systému Smalltalku.

### 3.5.3.2 Omezení metody XP

Zákazník musí více spolupracovat s týmem. Uvnitř týmu je nutná velmi dobrá komunikace, úspěšnost je mnohem více závislá na dobrém fungování týmu a mezilidských vztazích. Programátoři musí mít větší analytické a rozhodovací schopnosti, protože v jejich kompetencích je mnohem větší odpovědnost. XP se tedy hodí pro dynamické, dobře fungující týmy spolupracujících lidí.

### 3.5.3.3 Rozsah použití

Extrémní programování není zcela univerzální metodika a je vhodné pro nasazení v následujících podmínkách:

1. Malý až středně velký tým (typicky okolo deseti programátorů).
2. Malý a střední rozsah projektů (odpovídající velikosti týmu).
3. Využívání moderních objektových programovacích jazyků a vývojových prostředí.
4. Charakter projektů musí umožňovat dekompozici na funkční celky dodávané postupně. Nehodí se tedy pro kritické aplikace (např. řízení letového provozu), ale spíše pro aplikace typu informační systém.

Je tedy zřejmé, že XP (a obecně agilní metodiky) se hodí pro tvorbu moderních informačních systémů. Existují však obory, kde klasický přístup je stále jedinou možností. Úkolem manažera je potom podle charakteru projektu a týmu rozhodnout, který model řízení je vhodnější.

## 3.6 Návrhové vzory

Zkušené analytici neradi začínají tvorbu modelu „na zelené louce“ a snaží se používat co největší množství znalostí a postupů, které nabyli během práce na předchozích projektech. Tím vlastně si sami pro sebe zobecňují často používané a v praxi osvědčené postupy. Právě znalost takových postupů, což vlastně vyjadřuje míru zkušenosti, rozlišuje dobrého analytika od analytika začátečníka nebo analytika špatného. Celý proces tvorby modelů by šel významně urychlit i zkvalitnit, kdybychom měli možnost pro začátečníky srozumitelným způsobem zaznamenávat potřebné zkušenosti.

Naštěstí nejsme odkázáni na telepatii a nebo na nalévání znalostí nějakým trychtýřem dírou do hlavy, ale můžeme techniky a nástroje objektového modelování použít k reprezentaci diskutovaných znalostí mnohem jednodušším způsobem pomocí návrhových vzorů.

Návrhové vzory představují mechanismus k uchování znalostí o tvorbě modelů, které se týkají znovupoužitelných řešení. Je to významný prostředek k předávání znalostí mezi tvůrci systému. Návrhové vzory mohou představovat společný slovník výhodných postupů pro vývojový tým.

Velkým problémem implementace současných systémů je také jejich složitost. Pokud však budeme systém stavět z větších hotových dílů, než jsou samotné příkazy a další konstrukty programovacích jazyků, tak si návrh systému výrazně ulehčíme. V tomto smyslu představuje používání návrhových vzorů logické pokračování objektového přístupu.

Od používání návrhových vzorů lze očekávat:

1. Výměnu znalostí a s tím spojené řešení problémů, které by méně zkušený analytik nedokázal zvládnout.

2. Usnadnění dokumentace systému. Existuje-li totiž slovník návrhových vzorů, není třeba v dokumentaci konkrétního řešení podrobně opakovat strukturu objektů a stačí se odvolat na použitý návrhový vzor.
3. Zjednodušení tvorby programů. Návrhové vzory mohou být spolu s příkazy programovacího jazyka zahrnuty mezi základní stavební prvky softwarového systému.

### 3.6.1 Co to je návrhový vzor

Návrhové vzory jsou formalizované části znalostí z předchozích úspěšných projektů, které jsou zobrazitelné v podobě diagramů a jsou schopny se vzájemně kombinovat a v nových podmínkách hrát nové role. Používání této techniky spočívá ve vytváření modelu systému metodou skládání a jednotlivých vzorů. Rozlišujeme klasické programátorské návrhové vzory (např. MVC, double dispatching, composite, ...) a procesní návrhové vzory (např. půjčovna čehokoliv, ...).

Znovupoužitelnost, která je důležitou vlastností objektového přístupu, je významný prostředek ke zvýšení kvality softwarových systémů. Je důležité vědět, že znovupoužitelnost se týká nejen softwarových objektů, tak jak je známe z knihoven objektových programovacích jazyků, ale že se může týkat i modelů pro návrh nebo dokonce pro analýzu. Zkušenosti vývojarů neradi začínají návrh nového systému „od nuly“ a snaží se co nejvíce opírat o zkušenosti z předchozích projektů. Jestliže tedy naleznou a ověří nějaké dobré dílčí řešení, tak ho ve své práci používají. Návrhové vzory jsou prostředek k zaznamenání takových řešení. Dnes jsou již známy a v odborné literatuře publikovány desítky návrhových vzorů pro oblast objektového programování především v jazycích Smalltalk a C++. V oblasti business modelů také nejsou návrhové vzory neznámým pojmem, protože například velké konzultační firmy používají v konzultačních službách referenční modely business procesů pro jednotlivé oblasti.

Ke konkrétnímu vysvětlení, co to je návrhový vzor, si ukažme následující příklad: Představme si, že jsme úspěšně sestavili aplikaci pro vedení ordinace praktického lékaře. Náš program obsahuje objekty třídy **Pacient**, **Doktor** a **Vyšetření**, mezi nimi jsou vazby skládání i dědění, máme vyzkoušené, jak a mezi kterými metodami je třeba vést zprávy, aby program plnil optimálně zadání a zároveň byl co nejjednodušší a tím pádem snáze udržitelný. Nyní se stalo že někdo z našeho okolí má za úkol sestavit aplikaci pro zkuškovou evidenci nějaké školy. Protože se nám koukal přes rameno a zná náš systém pro vedení ordinace, tak dostane nápad, že použije naše osvědčené řešení pro sebe. Doktory vymění za učitele, pacienty za studenty a vyšetření za zkoušku. Stalo se tedy, že použil náš návrhový vzor v jiné roli, než jsme jej použili my.

Návrhové vzory jsou tedy různě velké a strukturované objektové modely, které shromažďujeme do katalogů návrhových vzorů. Základní vzory jsou dostupné v odborné literatuře a v poslední době je můžeme nalézt i v příručkách programovacích nástrojů. Rozsáhlé katalogy podrobných návrhových vzorů jsou však z pochopitelných důvodů většinou předmětem firemního tajemství.

Na návrhové vzory lze nahlížet jako na součástky do stavebnice systému. Za návrhový vzor nelze považovat popis nějakého celého, byť úspěšného systému. Pro jedno zadání můžeme použít více návrhových vzorů. V takovém případě se potom snažíme návrhové vzory spojovat tak, že objekt jednoho návrhového vzoru ztotožňujeme s jiným objektem jiného návrhového vzoru. Vzájemné spojování návrhových vzorů v konkrétní úloze tedy znamená, že návrhové vzory mají společné objekty.

### 3.6.2 Jak se návrhový vzor popisuje

Minimální popis návrhového vzoru, jak jej můžeme najít ve slovnících návrhových vzorů je následující:

1. **Název vzoru.** Je to vhodně zvolené výstižné pojmenování. Má-li vzor synonyma, jsou uvedena také.

2. **Popis struktury vzoru.** Typicky malý diagram znázorňující prvky a vazby návrhového vzoru. Nejčastěji se používají diagramy konceptuálních a softwarových objektů, ale používají se i návrhové vzory pro procesy a business objekty.
3. **Příklad použití.** V některých případech je vhodná přímo i ukázka vzorového kódu. Návrhový vzor má uvedeno několik (typicky 2 až 5) příkladů konkrétních navzájem různých rolí (konkrétních situací), ve kterých může být úspěšně použit k řešení. Především touto vlastností se návrhový vzor liší od podrobného dokumentu nějakého konkrétního projektu
4. **Seznam vzorů,** se kterými lze tento vzor kombinovat.
5. **Důsledky použití.** Rozbor důsledků, výhod, nevýhod a možných omezení, které jsou s aplikací návrhového vzoru spojeny.

### 3.6.3 Příklad návrhového vzoru Composite

Tento příklad je z knihy *Návrh programů pomocí vzorů* autorů Gammy, Helmse, Johnsona a Vlissise (jsou známí také jako Gang of Four (GoF) – gang čtyř), která je základní biblí návrhových vzorů a umět používat vzory v ní obsažené patří k základním dovednostem návrháře informačních systémů. V knize je obsaženo 25 základních návrhových vzorů rozdělených do 3 kategorií:

- Tvořivé vzory – zabírají proces tvorby instancí, a tím pomáhají budovat systém na něm nezávislý, například Abstract Factory, Builder, Singleton atd.
- Strukturální vzory – zabývají se skládáním tříd a objektů do rozsáhlejších struktur, příklady jsou Adaptor, Composite, Decorator, Facade atd.
- Vzory chování – se zabývají algoritmy a rozdělením povinností mezi objekty. Vzory chování popisují nejen statickou strukturu, ale i vzájemnou komunikaci mezi objekty. Příklady jsou Chain of Responsibility, Iterator, Observer, Visitor atd.

Dále si na příkladu strukturálního návrhového vzoru *Composite* (česky skladba) ukážeme, jak je takový vzor popsán,

#### 3.6.3.1 Účel

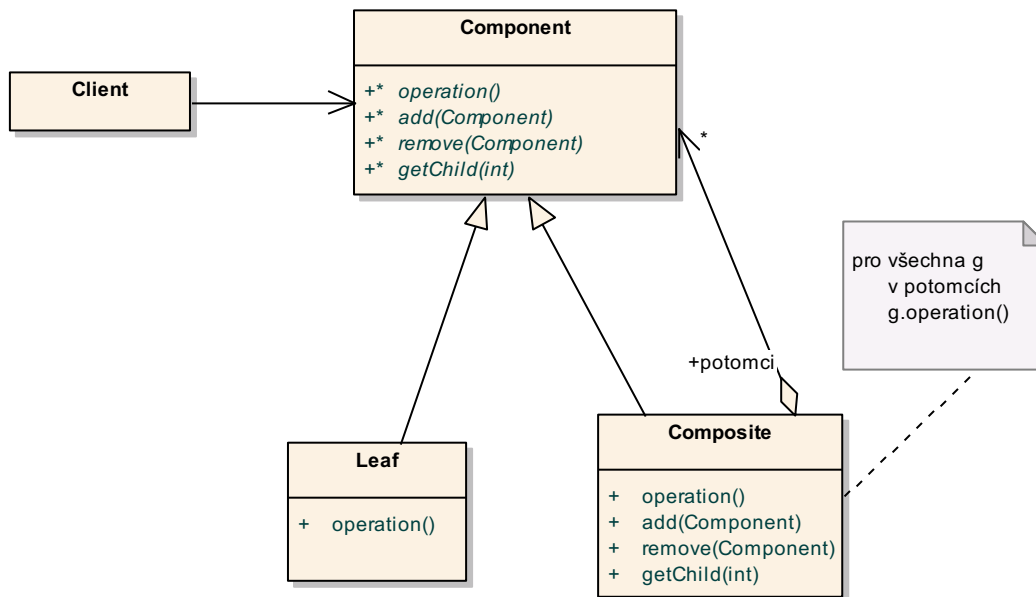
Skládá objekty do stromových struktur k vytváření hierarchií celek-část. Návrhový vzor *Composite* umožňuje jednotně zacházet s jednotlivými objekty i kompozicemi objektů.

#### 3.6.3.2 Použití

Návrhový vzor *Composite* použijeme v těchto příkladech:

- Potřebujeme vyjádřit hierarchii typu celek-část.
- Potřebujeme, aby klienti mohli ignorovat rozdíl mezi kompozicemi objektů a objekty. Klienti přistupují ke všem objektům ve složené struktuře stejně.

### 3.6.3.3 Struktura



### 3.6.3.4 Součásti

Návrhový vzor Composite se skládá z těchto součástí:

- **Component**
  - deklaruje rozhraní pro objekty v tomto návrhovém vzoru,
  - pokud je to vhodné, implementuje výchozí chování všech tříd,
  - deklaruje rozhraní pro přístup a správu svých potomků,
  - (volitelně) definuje rozhraní pro přístup v rodičům komponenty, případně ho i implementuje,
- **Leaf**
  - vyjadřuje listové objekty (list nemá žádné potomky),
  - definuje chování listových (primitivních) objektů.
- **Composite**
  - definuje chování komponent, které mají potomky,
  - uchovává potomkové komponenty,
  - implementuje operace týkající se potomků.
- **Client**
  - používá objekty návrhového vzoru Composite.

### 3.6.3.5 Spolupráce

Klienti používají třídní rozhraní *Component* k iteraci s objekty ve složené struktuře. Je-li příjemcem *Leaf*, je zpráva zpracována přímo. Je-li příjemcem *Composite*, je zpráva obvykle předána dál na zpracování potomkovým komponentům.

### 3.6.3.6 Důsledky

Návrhový vzor Composite má tyto důsledky:

- Definuje hierarchie, které se skládají z primitivních a složených objektů. Primitivní objekty lze složit do komplexnějších objektů, jež můžeme dále rekurzivně skládat. Kdykoli klient očekává primitivní objekt, může rovněž přijímat složený objekt.

- Zjednodušuje klienta. Klienti mohou zacházet se složenými strukturami stejně jako s jednotlivými objekty. Klienti zpravidla nevědí (a nemělo by je to ani zajímat), zda pracují s listem nebo se složenou komponentou.
- Zjednodušuje přidávání nových druhů komponent. Nově definované třídy Composite nebo Leaf fungují automaticky s existujícími strukturami.
- Může návrh příliš zobecnit. Druhou stranou mince, která ulehčuje přiřívání nových komponent, je, že znesnadňuje omezení komponent - někdy chceme, aby návrhový vzor Composite měl pouze některé komponenty.

### 3.6.4 Známá použití

Příklady vzoru Composite lze nalézt téměř v každém objektově orientovaném programovacím jazyce. V prostředí Smalltalk-80 je implementován systém Model-View-Controller pomocí návrhového vzoru Composite.

### 3.6.5 Příbuzné vzory

Propojení komponenta-rodíče se často používá pro návrhový vzor *Chain of responsibility*.

Návrhový vzor *Decorator* se často používá s vzorem *Composite*.

Vzor *Fly-weight* umožňuje sdílení komponent, ale ty již nemohou odkazovat na svoje rodiče.

*Iterator* se může použít pro procházení komponent.

Vzor *Visitor* lokalizuje operace a chování, které by jinak bylo rozvrženo mezi třídami Composite a Leaf.

## 3.7 Softwarové metriky a metoda funkčních bodů

Nástrojem pro podporu formálního měření softwarových produktů jsou softwarové metriky. Jejich sběr a vyhodnocování je důležitou součástí řízení projektů. Aparátem pro softwarové metriky je především numerická matematika a statistický aparát.

Softwarové metriky se dělí podle časového kritéria na

- **statické metriky** zabývající se příslušným systémem jako celkem a
- **dynamické metriky**, které doplňují (pokud je to možné) hodnoty jednotlivých ukazatelů o časový pohled (míra změn v čase, časové výkony a časové intezity jednotlivých veličin).

Podle aplikačního kritéria lze jednotlivé metriky (statické i dynamické) dělit na

- **projektové metriky** (*project metrics*) sloužící především k měření veličin potřebných k určování a porovnávání charakteristik jednotlivých projektů jako celku z pohledu projektového řízení během všech fází životního cyklu. Pro objektový přístup to jsou například následující ukazatele:
  - **míra kvality řídicího procesu**, (v současné době rozpracováváno v sadě norem ISO)
  - **míra kvality vývojového procesu**, (v současné době rozpracováváno v sadě norem ISO)
  - **velikost vyvíjené aplikace** (počet scénářů, tříd, subsystémů, modelů, ...),
  - **velikost pracovního týmu** (průměrný počet člověkodní na scénář, třídu či objekt, průměrný počet vyvinutých entit na pracovníka, ...) a

- **plánovací ukazatele** (počet iterací ve vývoji, počet uzavřených subdodávek, počet etap projektu, ...).
- **návrhové metriky** (*design metrics*) zabývající se podrobnostmi vyvíjené softwarové aplikace a slouží jako efektivní podpora pro testování produktů, vzájemné porovnávání produktů a nástroj pro určování silných a slabých míst návrhu pro potřeby měření, opravování a vylepšování jednotlivých komponent vyvíjené aplikace. Pro objektový přístup to jsou například následující ukazatele:
  - **velikost metod** (průměrný počet posílaných zpráv na metodu, prům. počet příkazů na metodu, rozsah komentářů, stupeň polymorfismu, ...)
  - **velikost objektů** (prům. počet instančních a třídních metod na třídu, rozsah komentářů, prům. počet instančních a třídních proměnných na třídu, ...) a
  - **míra vazeb mezi objekty** (průměrná hloubka dědičnosti, průměrný počet složek objektů, závislost objektů, znovupoužitelnost objektů a metod, prům. počet odstíněných metod na třídu, rozsah komentářů,...).

Na počátku projektu potřebujeme odhadovat velikost budoucího projektu v počtech příkazů či řádcích kódu, ale bohužel jediné co známe, je popis zadání budoucího systému. V takovém případě bychom tedy nemohli použít algoritmy COCOMO. Naštěstí byla již v roce 1979 popsána Albrechtem metrika nazývaná anglicky „function points“, která je česky překládána jako „funkční body“ nebo také „funkční jednice“. Tato metrika slouží k odhadování rozsahu vyvíjeného softwaru metodou výpočtu z rozsahu požadavků na systém, kde každý parametr je násoben příslušnou váhou a násobky jsou potom sečteny. Z výsledného počtu funkčních bodů lze odhadnout velikost budoucího programu, protože pro různé programovací jazyky je na základě statistických měření známo, kolik příkazů je třeba na pokrytí jednoho funkčního bodu. Konkrétní vzorec, který upravil Jones pro potřeby objektového programování v roce 1995 na základě rozsáhlých statistických analýz různých projektů je následující:

$$N = 4 \times inputs + 5 \times outputs + 4 \times queries + 7 \times datfiles + 7 \times interfaces + 3 \times algorithms,$$

kde N je celkový počet funkčních bodů požadované aplikace,  
*inputs* označuje počet požadovaných uživatelských vstupů do aplikace,  
*outputs* označuje počet požadovaných uživatelských výstupů z aplikace,  
*queries* označuje počet požadovaných uživatelských dotazů do aplikace,  
*interfaces* označuje počet požadovaných datových rozhraní aplikace do jiných aplikací a  
*algorithms* označuje počet algoritmů, které bude aplikace používat ve svých výpočtech.

Pro zajímavost si ještě uveďme, že čistý objektový programovací jazyk potřebuje jen 20 až 50 příkazů na jeden funkční bod, což je asi 2× až 5× méně než pro jazyk klasický nebo hybridní (Fenton & Pflieger, Software Metrics, Springer 2000). Ve stejné publikaci se také praví, že výkonnost programátora měřená v naprogramovaných funkčních bodech za jeden měsíc práce logaritmicky klesá s celkovým objemem projektu měřeném ve funkčních bodech. Pro malé projekty mající nejvýše stovky funkčních bodů je typická výkonnost 20 až 50 funkčních bodů na osobu za měsíc, ale u projektů v řádech desítek tisíc funkčních bodů to je jen 1 až 5 funkčních bodů na osobu za měsíc. Podobné výsledky jako původní práce z 80. let přináší pro nejužívanější programovací jazyky i nejnovější studie firmy Software Productivity Research Inc. ([www.spr.com](http://www.spr.com)):

<i>programovací jazyk</i>	<i>počet příkazů na jeden funkční bod (SPR Inc., 2002)</i>	<i>počet příkazů na jeden funkční bod, (Albrecht &amp; Behrens, 1983)</i>
Smalltalk	21	21
Ada 95	49	-
Java	53	-
C++	53	-
Basic	-	64
Lisp	-	64

Prolog	-	64
Modula-2	-	71
PL/1	-	80
RPG	-	80
Pascal	-	91
Fortran	-	106
COBOL	107	106
Algol	-	106
C	128	150
Assembler	-	320

Bouřlivý rozvoj technologií však nějaký posun přinesl. Dříve tu byl problém „jak“ a „čím“ software naprogramovat, protože jsme byli velmi omezeni kapacitními a výkonnostními možnostmi výpočetního stroje. Dnes tohle největší problém není. Máme tu ale o nic méně palčivý jiný problém, který spočívá v otázce správně zjistit, „co“ přesně uživatel chce, abychom optimálně splnili jeho požadavky, systém sestavili efektivně s optimálním využitím našich zdrojů a uživateli byl opravdu k užítku. To tedy znamená, že kromě programátorů dnes potřebujeme především kvalitní analytiku.

## 3.8 Techniky ladění programů

### 3.8.1 Úvod

Stará programátorská pravda říká, že nejdůležitější umění na psaní programů je hledání chyb. I když to je určitá nadsázka, je v ní mnoho pravdy. **I ten nejbystřejší a nejzkušenější programátor se dopouští při psaní programu chyb.** Je proto velmi důležité naučit se chyby efektivně vyhledávat a odstraňovat – této činnosti se v programátorské hantýrce říká „*ladění programu*“.

### 3.8.2 Hledání místa vzniku chyby

Pokud se program nechová tak, jak očekáváme, může to znamenat obecně dvě situace:

1. Provádění programu se přeruší chybou (výjimkou – viz příslušná kapitola).
2. Program běží, ale chová se chybně nebo nějakou činnost nevykonává.

Ačkoliv se to možná na první pohled nezdá, situace 1 je lepší variantou, protože ihned známe místo, kde chyba vzniká. Pokud tedy nevíme, kde chyba vzniká, musíme jí hledat. Máme po ruce čtyři techniky:

1. vizuální kontrola zdrojového kódu / GUI Painteru,
2. kontrolní výpisy,
3. prohlížení objektů za běhu,
4. umělé zastavení programu.

#### 3.8.2.1 Vizuální kontrola zdrojového kódu

Při vizuální kontrole zdrojového kódu procházíme kód a – pokud chyba určitým způsobem souvisí s uživatelským rozhraním (např. tlačítko nepracuje) – také údaje v GUI Painteru, obvykle názvy aspektů a volaných metod (actions)<sup>15</sup>. Častou chybou bývá **nesoulad malých/velkých písmen, čeština či**

<sup>15</sup> Chyby v uživatelském rozhraní se ale obvykle projeví výjimkou, tj. zastavením programu.

**mezery v názvech**, atd. Kontrolujeme především soulad mezi názvy nastavenými v GUI Painteru a těmi, které používáme ve svém kódu.

Ve zdrojovém kódu pak mentálně procházíme činnost jednotlivých konstrukcí, kontrolujeme syntaxi (tečky, středníky, návratový výraz (^), ...).

Manuální způsob kontroly programu dokáže odhalit mnoho jednoduchých chyb z nepozornosti, často je však i triviální chyba „rafinovaně“ ukrytá a je třeba přistoupit k některé z dalších technik.

### 3.8.2.2 Kontrolní výpisy

Kontrolní výpisy umožňují v klíčových místech programu vypsat zprávu. Ta může být jenom ověřovací typu „Jsem tady“ nebo to může být hodnota klíčové proměnné pro zjištění, jestli nabývá správných hodnot. Kontrolní výpis posíláme typicky do:

1. **Transcriptu** zprávou `Transcript show`:

```
Transcript show: promenna; cr.
```

2. **Dialogu** zprávou `Dialog warn`:

```
Dialog warn: 'Promenna a ma hodnotu', a displayString.
```

Rozdíl je v tom, že v prvním případě program po výpisu hodnoty pokračuje dál v provádění, ve druhém se zastaví a čeká na stisk tlačítka.

Pokud tedy víme, že výpis by měl proběhnout desetkrát, necháme si vše přehledně vytisknout do `Transcriptu`. Pokud však je hodnot příliš mnoho a chceme si jen prohlédnout prvních pár, `Dialog` bude lepší volbou.

Je třeba si dát pozor na to, že oba tyto kontrolní výpisy musí být **řetězce**, pokud tedy vypisujeme např. číslo či datum, musíme objektu ještě poslat zprávu `displayString` jako je ukázáno ve druhém příkladu.

### 3.8.2.3 Prohlížení objektů za běhu – Inspector

`Inspector` je velmi šikovný nástroj, který umožňuje prohlížet obsah jakéhokoliv objektu. Objektu, který chceme prohlédnout pošleme zprávu `inspect`. Ta otevře okno `Inspectoru`, program však poběží dál. Jedná se tedy svým způsobem také o variantu „kontrolního výpisu“. Je však třeba si dát pozor, aby v nějaké programové smyčce se oken neotevřelo příliš mnoho.

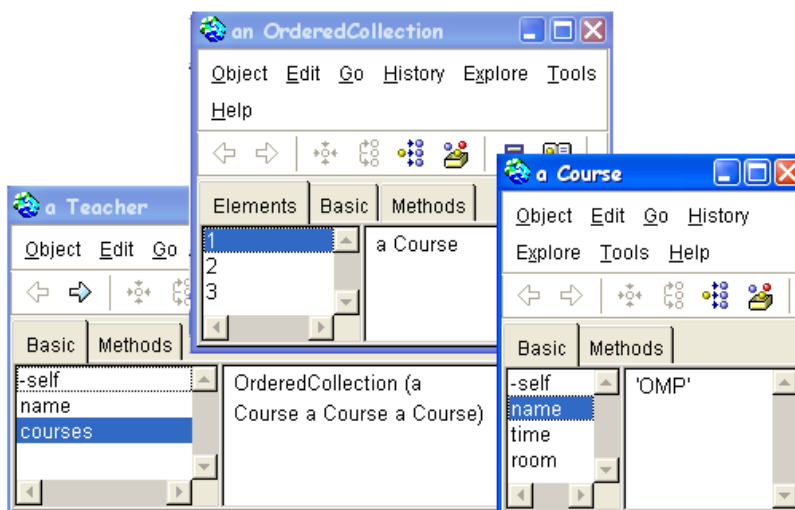
Kromě programového vyvolání `Inspectoru` je možné ho otevřít prakticky odkudkoli na cokoliv: označením výrazu a vyvoláním „`Inspect it`“ z menu pravého tlačítka myši. Toto je možné například v `Transcriptu`, systémovém pořadači, `Debuggeru`, atd.

Pomocí `inspectoru` můžeme prohlížet i výsledky „pokusného“ poslání zprávy. Např.

```
(myCollection collect: [:x | ... ]) inspect
```

vytvoří novou `Collection` a otevře na ní `Inspector`. To umožňuje prohlédnout si výsledek nějaké činnosti „nanečisto“ ještě před tím, než ho skutečně začleníme do programu.

Příklad oken `Inspectoru` vidíme na následujícím obrázku:



V levém sloupečku jsou jednotlivé instanční proměnné, obsah označené proměnné je zobrazen napravo. Co činí Inspector velmi mocným nástrojem, je možnost **zanořování do objektů dvojklikem levého tlačítka myši**.

### 3.8.2.4 Umělé zastavení programu

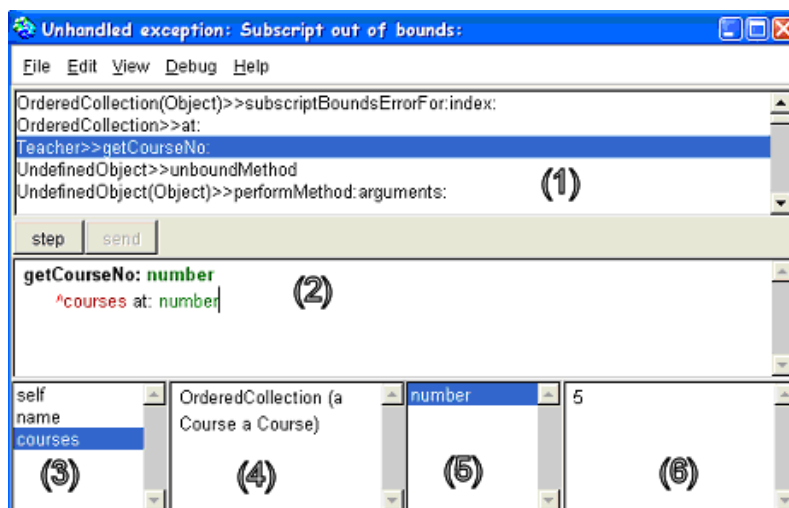
Program lze v kterémkoliv místě přerušit přidáním konstrukce `self halt`. Provádění programu se přeruší a objeví se okno jako při vzniku výjimky (viz kapitola o výjimkách) s textem „Halt encountered“ („Došlo k zastavení“). Obvykle pak pokračujeme do Debuggeru tlačítkem `Debug` (viz následující sekce).

Umělé zastavení používáme v případě, kdy si chceme ověřit větší množství proměnných a zkontrolovat všechny zainteresované objekty v určitém okamžiku provádění programu. Mnoho programátorů dává také přednost umělému zastavení před kontrolními výpisy, protože Debugger umožňuje taktéž zjistit hodnotu proměnné a navíc je možné případně ihned provádět další ladicí činnosti.

### 3.8.3 Debugger

Připomeňme si, že do debuggeru se typicky můžeme dostat stisknutím `Debug` po vzniku neošetřené výjimky nebo konstrukci `self halt`.

Okno debuggeru vidíme na následujícím obrázku:



V horní části (1) je posloupnost zpráv, které vedly k současnému stavu. První zpráva od shora je vlastní zastavení programu, zajímat nás tedy bude až **druhá a následující zprávy**. Druhá zpráva je tedy ta, ve které došlo k zastavení programu. V okně (2) se ukazuje kód příslušné metody a zpráva, u které došlo k zastavení je zvýrazněna. Pokud tedy přejdeme v seznamu zpráv (1) nahoru, de-facto se zanoříme do zavolané metody.

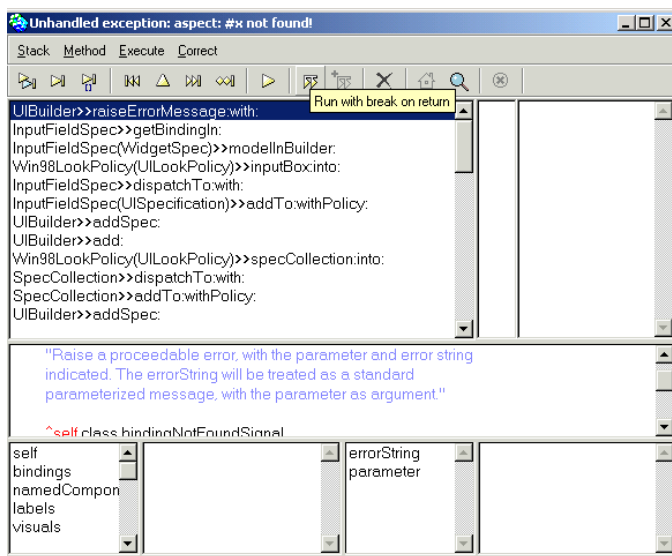
**Ne vždy je hned druhá zpráva místem vzniku chyby.** Např. na obrázku výše došlo k pokusu z `OrderedCollection` číst zprávou `at:` prvek, který neexistuje. Výjimku vyvolala zpráva `at:`, místo chyby je však o úroveň níže: v metodě, která zprávu `at:` zavolala s neexistujícím argumentem. Podle povahy chyby tak může být třeba se zanořit až o několik úrovní do hierarchie volání, dokud nevidíme „náš“ kód se zvýrazněnou zprávou, která vedla nakonec k výjimce.

Okno (3) obsahuje seznam instančních proměnných. Hodnota vybrané položky se zobrazuje v okně (4), podobně jako v Inspectoru. Okna (5) a (6) jsou analogická pro parametry metody a pomocné proměnné. Dvojklikem levého tlačítka myši je možné na jakoukoliv proměnnou otevřít Inspector.

Díky plně dynamickému prostředí VisualWorks je dokonce možné si v okně Debuggeru napsat jakýkoliv výraz s použitím všech aktuálně dostupných proměnných a ten po označení nechat vyhodnotit (`Do it`), vytisknout (`Print it`) nebo prozkoumat v Inspectoru (`Inspect it`).

### 3.8.3.1 Analýza výjimek z uživatelského rozhraní

Vznik výjimky způsobené problémem v uživatelském rozhraní se pozná podle názvů tříd v seznamu zpráv:



Příčinu takové výjimky je pro začátečníka velmi těžké odhalit v Debuggeru, neboť vyžaduje hlubší znalosti interních mechanismů. Nejlepším způsobem nápravy je detailní kontrola důležitých nastavení v GUI Painteru (viz „Vizuální kontrola zdrojového kódu“).

### 3.8.4 Předcházení chybám

Stejně jako v jiných oborech i v programování platí, že prevence je vždy lepší než náprava. Při programování bychom se tak měli držet určitých pravidel vedoucích k minimalizaci počtu chyb. Jsou to zejména:

- **Přehledný zápis kódu.** Díky formátovači zdrojového kódu (CTRL+O) tato činnost není prací navíc, měli bychom jí proto hojně využívat.
- **Výstižné názvy.** Názvy tříd, atributů, metod, proměnných, atd. by měly být vysvětlující. Proměnná `pocetZkousek` je jistě lepší než nic neříkající `x`. Je rovněž dobrou praxí pojmenovávat názvy parametrů metod tak, aby byl zřejmý jejich očekávaný typ, např. `prikazString`.
- **Programování po malých částech.** Smalltalk umožňuje programovat stylem doplňování běžící aplikace. Tento styl programování také pomáhá minimalizovat ztráty času strávené odstraňováním chyb. Je dobré např. vytvořit uživatelské rozhraní, spustit aplikaci a kódy metod tlačítek dodělat za běhu. Případné chyby totiž tak odhalíme mnohem dříve a také objem „podezřelého“ kódu je menší.
- **NEZNÁMÉ VĚCI ZKOUŠET ODDĚLENĚ.** Jsou chyby z nepozornosti a jsou chyby z neznalosti. Předcházející body se týkaly většinou první skupiny, tento bod souvisí s druhou. Při začleňování mechanismu, který používáme poprvé, bychom si tento nejdříve měli „osahat“ mimo, nanečisto, např. ve Workspacu. Např. pokud chceme do svého programu zařadit mechanismus odčítání časů, je lepší si nejdříve zkusit pohrát s třídou `Time` a na pár příkladech si ověřit, že opravdu umíme udělat to, co chceme. Tento bod je velmi důležitý pro tento kurs, neboť se týká především začátečníků. Je proto uveden velkými písmeny.

## 3.9 Úvod do refaktoringu

Termín *refaktoring* (též *refaktORIZACE*, angl. *Refactoring*) je definován jako změna kódu beze změny jeho funkčnosti. Při refaktoringu typicky přejmenováváme třídy, metody, provádíme změny třídních hierarchií, atd. Proč tedy refaktoring provádět, když se funkčně na programu nic nevylepší? Refaktoring zkvalitní kód v následujících ohledech:

- samodokumentace
- pochopitelnost po dlouhé době
- pochopitelnost jinými
- rychlost
- paměťová náročnost
- objem kódu (kompaktnost)
- konzistence

Kód je tedy lidově řečené „čistší“, což umožňuje ho snáze spravovat a rozšiřovat. Při refaktoringu též můžeme odhalit návrhové vzory (viz. příslušná kapitola).

### 3.9.1 Základní refaktorigační úkony

Určité refaktorigační úkony jsou praktikovány při vývoji prakticky ve všech programovacích prostředích, my zde však budeme hovořit výhradně o refaktorigaci objektově orientovaných systémů.

Refaktorigační úkony lze rozdělit do tří kategorií:

- refaktorigace orientovaná na **třídy** – např. přejmenování třídy spolu se změnou všech referencí na ni, přesun proměnné z/do podtřídy, ...
- refaktorigace orientovaná na **metody** – např. přejmenování, nahrazení volání metody jejím kódem, ...

- refaktORIZACE orientovaná na vlastní **kód** – např. extrakce kódu do metody, změna dočasné proměnné na instanční, ...

RefaktORIZACÍ je však celá řada a je mimo rozsah tohoto studijního textu se jimi podrobněji zabývat.

### 3.9.2 RefaktORIZACE v prostředí VisualWorks

VisualWorks od verze 7.0 standardně obsahují tzv. *Refactoring Browser*, který je integrován do systémového pořadače. Ten umožňuje automatizovaně provádět celou řadu refaktORIZACÍ. Příslušné operace lze najít v menu Class (např. Class → Rename, Class → Instance Variables → Push Up, aj.), Method (např. Method → Refactoring → Push Up) a některé jsou v menu pravého tlačítka myši (např. pro pomocnou proměnnou existuje v menu položka Temporary Variable → Convert to Instance Variable). Pro podrobnější seznámení s možnostmi Refactoring Browseru doporučujeme dokument „*Application Development Guide*“ z adresáře /doc.

## 3.10 Problémy objektového návrhu

Dědičnost je jednou z mála výlučných vlastností objektově orientovaného přístupu. Oproti skládání jen nenacházíme v technologiích, které objektovému přístup předcházely. Proto je dědění tak oblíbené v řadě příruček při laických výkladech, co to je OOP.

O dědění se v kuchařkách objektového programování praví, že slouží k tvorbě nových typů pomocí již existujících v systému. To je pravda, ale nesmíme přitom zapomínat na skládání, pomocí kterého nové objekty můžeme vytvářet také. (viz. kapitola 1.8.6 a 2.5.6) I když je implementace nových tříd pomocí skládání, v mnohých případech čistší z pohledu OOP, tak je v programátorské praxi málo oblíbená nebo dokonce neznámá, protože vyžaduje trochu více programátorské práce, než je tomu při použití dědění.

Zhruba řečeno, dědění je vhodné ve většině případů, kdy potřebujeme rychleji realizovat nějaké nové objekty, a kdy se nemusíme zabývat možnými důsledky dědění, protože máme jistotu, že s objekty se bude nakládat jen předepsaným způsobem a stačí nám, že software bude správně fungovat. To je především oblast prototypového návrhu. Pokud však potřebujeme sestrojít nové objekty, které budou dále využívány a mají být součástí stabilní a robustní aplikace, je třeba zvážit, zda by využití skládání nebylo vhodnější.

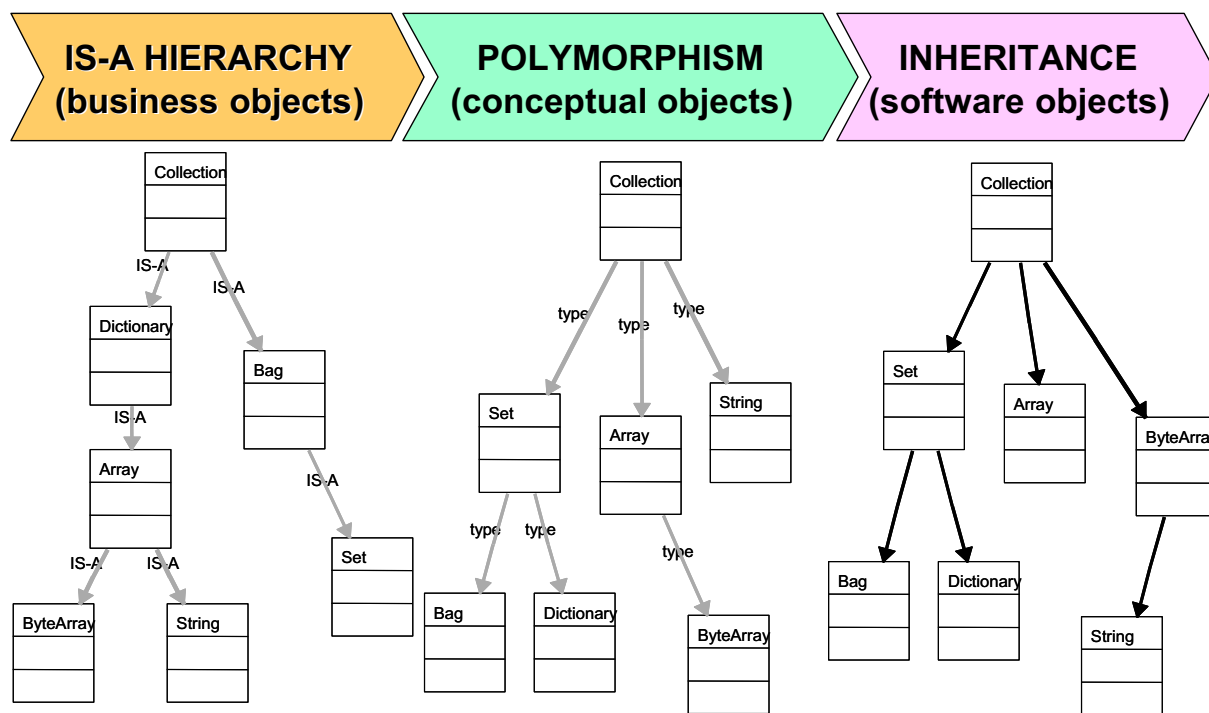
### 3.10.1 Dědění, hierarchie typů a taxonomie nejsou vždy totéž

V předchozí kapitole jsme si ukázali, že nové typy se v objektových systémech realizují pomocí tříd, přičemž ale novou třídu do systému lze vyrobit nejen pomocí dědění, ale i skládáním. Z toho ale vyplývá, že hierarchie dědění a hierarchie typů v jednom systému nemusí vždy znamenat totéž. Navíc při používání dědičnosti máme v programovacích jazycích nástroje pro skrývání metod a dat děděných objektů. Na hierarchii tříd lze proto nahlížet trojím způsobem podle následujících kritérií:

1. Z pohledu návrháře – tvůrce nových objektů. Tato hierarchie je **hierarchií dědičnosti**, protože dědičnost je výborným nástrojem pro tvorbu nových tříd.
2. Z pohledu uživatele – analytika nebo aplikačního programátora, který potřebuje již hotové objekty použít ve svém systému. Tento pohled lze ještě podrobně dělit na
  - 2.1. Z pohledu polymorfismu – objekty na nižších úrovních hierarchie potom musejí být schopny dostávat stejné zprávy, jako objekty vyšších úrovní. Právě tato hierarchie je **hierarchie typů**.

2.2. Z pohledu aplikační domény – instance tříd na nižších úrovních potom musejí být prvky stejné domény, kam patří instance tříd nadřazené třídy. To znamená, že doména nižší úrovně je podmnožinou domény vyšší úrovně. Tato hierarchie je anglicky označována jako IS-A, česky ji můžeme přeložit „je jako“ (nebo „patří k“). Od hierarchie typů se může v konkrétních případech lišit proto, že se nezabývá jen chováním objektů na rozhraní, ale objektem celkově.

U jednoduchých úloh je samozřejmě pravda, že uvedené tři hierarchie jsou totožné. Proto se tímto faktem programátorské kuchařky příliš nezabývají. U komplexnějších úloh však toto tvrzení neplatí a to především při návrhu systémových knihoven, které se opakovaně znovupoužívají při návrhu konkrétních systémů.



Pěkným příkladem je ukázka na obrázku. Na obrázku je (zleva doprava) hierarchie dědění, hierarchie typů a hierarchie „je jako“ pro část systémové knihovny jazyka Smalltalk týkající se sad (collections) objektů. Na obrázku jsou následující třídy:

**Collection** (česky „sada“). To je abstraktní třída, ze které jsou odvozovány jednotlivé konkrétní třídy. Společnou vlastností všech těchto je objektů je schopnost obsahovat jako svoje data další objekty. Sady jsou tedy schránky na objekty.

**Dictionary** (česky „slovník“). Slovník je taková sada, kde každá hodnota v ní uložená má přiřazenou jinou hodnotu (takže dohromady tvoří dvojici), která slouží jako přístupový klíč k dané hodnotě. Slovníky můžeme použít opravdu jako slovníky pro jednoduché překlady z jednoho jazyka do druhého. Dalším často uváděným příkladem použití objektových slovníků je například jejich použití pro telefonní seznam – klíčem jsou jména osob a hodnotami s klíči spojenými jsou telefonní čísla.

**Array** (česky „pole“). Pole lze jednoduše popsat jako slovník, kde klíče hodnot mohou nabývat pouze přirozených čísel 1 až velikost pole. Na hodnoty pole se přistupuje tedy také jakoby pomocí klíčů.

**ByteArray** (česky „bajtové pole“). Jedná se o takové pole, kde povolený okruh hodnot je omezen na celá čísla v intervalu 0 až 256. Ostatní vlastnosti se nemění.

**String** (česky „řetězec znaků“). Na řetězec znaků lze nahlížet jako na pole (Array), kde povolený okruh hodnot je omezen na znaky.

**Bag** (česky „pytel“). Bag je taková sada, do které lze objekty ukládat nebo z ní vybírat. Hodnoty uvnitř pytle ale nejsou přístupné pomocí klíčů, ale pouze skrze svoji hodnotu – pokud hodnotu objektu neznám, nemohu jej odebrat.

**Set** (česky „množina“). Množina je taková sada typu pytel, do které lze stejnou hodnotu vložit jen jednou. Pokud množina hodnotu již obsahuje, tak další vložení stejné hodnoty je ignorováno na rozdíl od výše uvedeného pytle, který násobné výskyty stejné hodnoty dovoluje. Objekty množina svojí funkčností odpovídají matematickému chápání množin. Proto se tak jmenují.

Uvedený popis tříd na obrázku sleduje hierarchii „je jako“ (IS-A). Pokud bychom ale popis soustředili na popis funkčnosti – rozhraní objektů, které je vymezeno okruhem přípustných zpráv, dojdeme k poněkud odlišné **hierarchii typů**. například slovníky mohou dostávat stejné zprávy jako množiny a lze tedy na ně nahlížet jako na podtypy množin. A naopak s řetězci znaků se pracuje značně odlišným způsobem než s poli, takže je můžeme považovat za typ, který s poli souvisí jen málo.

A do třetice hierarchie dědění, která je podstatná pro programovou realizaci uvedených tříd, je také trochu jiná. Řetězce znaků je výhodné implementovat děděním z bajtových polí a přidáním či pozměněním potřebných metod. Naopak pole a bajtová pole se implementačně dost liší, protože obyčejná pole se v paměti realizují jako pole odkazů na objekty kdežto bajtová pole jsou jednoduché úseky paměti. Dědění mezi poli a bajtovými poli proto užitečné není.

Jak bylo ukázáno, problematika typů a tříd a jejich vztah k dědění je složitá. Při návrhu systému je proto nejlepší dědění odsunout na pozdější fáze a zacházet s ním jen jako s implementačním nástrojem. Nejprve je třeba na úrovni objektů reálného světa rozpoznat hierarchii „je jako“ (is-a), potom ji upřesnit vymezením příslušných typů pro konceptuální objekty a až nakonec přemýšlet o optimální implementaci typů pomocí dědění softwarových objektů.

V programovacích jazycích Java, C# a Object Pascal také lze nalézt prostředky (které se ale bohužel velmi málo nebo nesprávně používají) pro oddělení typů a tříd.

### 3.10.2 Úspora za každou cenu

Dědění se často mylně považuje za povinný nástroj, který je nutně nedílnou součástí každého objektového systému. Objektový program, který neobsahuje dědění, je automaticky považován za podezřelý či dokonce z špatný. Výsledkem jsou křečovitě snahy programátorů najít podobnosti mezi objekty za každou cenu a dědit nejrůznější bizarnosti.

Nesprávné použití dědění si ukážeme na příkladě, který v nejrůznějších konkrétních podobách autoři dobře znají:

Mějme za úkol sestavit aplikaci pro evidenci knih v knihovně. Lze si představit, že analýza nás dovede ke třídám **Knih** (popis knihy s názvem, autorem, ISBN, ...), **ExemplářKnihy** (konkrétní výtisk s evidenčním číslem, který se půjčuje a vrací), **Čtenář** (ten, kdo si exempláře půjčuje a vrací) a **Knihovna** (systém jako celek, který spravuje knížky a čtenáře). V takovém systému je několik vazeb skládání (např. mezi čtenářem a exemplářem kvůli uchování informace o výpůjčce, ...), ale uvedené čtyři třídy jsou mezi sebou typově zcela odlišné. Například kniha není nikdy čtenářem, čtenář není nikdy knihovnou, atd. Celá řada programátorů je však tímto faktem vyvedena z míry a ve snaze učinit program více objektovým, najde podobné atributy například u čtenáře, knihy a knihovny a vyrobí jim společného předka. Takto lze například extrahovat atribut **název** a dědit ho jako jméno čtenáře, název knihy, označení celé knihovny. Ještě příšernější je dědění mezi těmito třídami navzájem; například **Knih** může dědit ze **Čtenáře**, koneckonců kniha má vždy svého autora, a když tedy pomocí dědění rozšíříme osobu o ISBN, rok vydání a název, máme **Knihu** se vším všudy a ještě jsme v programu ušetřili pár řádek zdrojového kódu, o čemž přeci OOP je.

Největší chybou takových přístupů není to, že programy nefungují. To není pravda – pokud jsou používány v souladu se zadáním, tak fungovat mohou docela dobře (tím obtížněji se pak vysvětluje, že

program je špatný). Problém je v tom, že nadměrné užívání dědění kromě požadovaných funkcí do objektů přidává i funkce, které přesahují zadání a které jednak mohou být zneužity a nebo komplikují pozdější údržbu nebo modifikace.

### 3.10.3 Vícenásobná dědičnost?

S vícenásobnou dědičností je třeba zacházet velmi opatrně. Na úrovni hierarchie typů ve fázi konceptuálního návrhu je sestavování nových typů z více existujících současně správné. To ale neznamená, že v implementaci budeme otrocky podle hierarchie typů dědit třídy mezi sebou. Jen málo programovacích jazyků dovoluje vícenásobné dědění – je to například CLOS a Eiffel nebo některé dialekty Smalltalku. Z běžně používaných jazyků dovoluje vícenásobné dědění jen C++, ale při bližším prozkoumání vychází najevo, že objekty tříd, která vícenásobně dědí (dědí z více tříd současně), jsou ve skutečnosti objekty poskládané z příslušných částí. V C++ lze tedy na vícenásobné dědění nahlížet jako na syntaktickou zkratku pro skládání objektů, které přináší úsporu zdrojového kódu a těm, kteří ji ovládají dodává pocit magie.

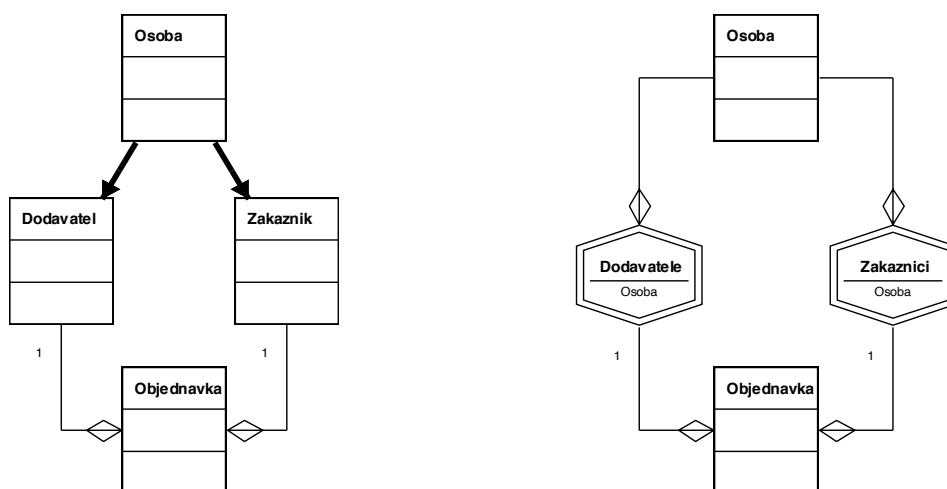
Naštěstí neexistuje v OOP případ struktury, která by byla realizovatelná pouze pomocí vícenásobného dědění, protože vše lze výhodně řešit kombinací jednoduchého dědění a skládání objektů.

### 3.10.4 Třídy versus množiny objektů

Na pojem třídy se v OOP může nahlížet současně dvojím způsobem:

1. Třída je **realizace objektového typu**; ve třídě uchováváme popis struktury objektů tohoto typu a množinu jeho operací/metod. V čistě objektových jazycích (např. CLOS, Smalltalk) se s třídou na rozdíl od hybridních/méně objektově orientovaných jazyků (např. C++, VB, Object Pascal v Delphi a Java) může pracovat jako s objektem se vším všudy diskutovaný popis struktury spolu s operacemi jsou jeho data, se kterými lze za chodu programu pracovat, měnit je, doplňovat přidávat nebo mazat.
2. Úplně jiný pohled na třídu je chápání třídy jako množiny, která jako prvky obsahuje objekty, které dané třídě náleží. Tento pohled na třídu je více abstraktní a ztotožňuje pojem třídy s pojmem množiny - domény, která vymezuje výskyt objektů daného typu.

Naneštěstí se v běžných metodách, jako je UML, oba způsoby nazírání na pojem třídy mísí dohromady a nerozlišuje se mezi třídou jako objektem sám o sobě a mezi třídou jako pomyslnou množinou tvořenou všemi objekty, které do dané třídy patří.



Nesprávné splnutí diskutovaných dvou pojmů vede potom k tomu, že se všechny množiny objektů v modelovaném systému modelují jako třídy. Problém si ukážeme na příkladu na obrázku. V systému,

kde jsou objekty třídy **Osoba**, potřebujeme vymezit z nějakého důvodu dvě množiny osob: **Zákazníky** a **Dodavatele**. Pro analytika, který ovládá jen UML a nezná do hloubky problematiku OOP, se nabízí jednoduché řešení: sestrojít dvě nové podtřídy **Zákazník** a **Dodavatel**, které dědí ze třídy **Osoba**. Toto řešení však není vždy optimální minimálně ze dvou důvodů:

1. Pokud nepotřebujeme v nových podtřídách nové atributy a operace, tak se nové třídy zavádí jen kvůli potřebě zanesení do modelu skutečnosti, že některé osoby jsou a jiné nejsou zákazníky a dodavateli.
2. Přináší potíže v případě, kdy se jedna a ta samá osoba stane zákazníkem a dříve byla dodavatelem a dokonce nebo kdy bude současně zákazníkem i dodavatelem.

Nerozlišování pojmy třída a množina je možné (autoři to doporučují) z důvodu jednoduchosti a srozumitelnosti modelovat jednotně na úrovni hierarchie „je jako“ (is-a) pro objekty reálného světa. Pro konceptuální objekty, kde se tato hierarchie převádí na hierarchii typů, jsou však množiny potřeba a pro objekty softwarové, kde hierarchie typů vede k hierarchii dědění je to už nezbytnou nutností.

### 3.10.5 Podtřídy nebo instance?

Posledním problémem, který souvisí s děděním a týká se objektového modelování, je otázka, do jaké míry je účelné vytvářet nové a nové podtřídy a kdy se rozhodnout ukončit vlnobití nových tříd a různost objektů realizovat pomocí různých hodnot atributů ve třídách. Problém si popíšeme na příkladu:

Mějme část knihovního informačního systému, která se týká časopisů. Časopisy lze dělit na odborné, vědecké, populárně naučné, ... - vytvoříme tedy jednotlivé třídy. Odborné časopisy lze dělit na počítačové, lékařské, ekonomické, ... - takže další podtřídy. Počítačové časopisy jsou Softwarové noviny, Bajt, Computer World, ... . Softwarové noviny lze dělit na ročník 1991, 92, ... a tak dále až na jednotlivé výtisky. Kde ale ukončíme tvorbu nových tříd a budeme vzájemně odlišnosti zapisovat jen do atributů objektů stejné třídy? Program lze sestavit jak jedním extrémem: jedna třída časopis a všechny výtisky různých časopisů jsou její instance s různou hodnotou proměnných, i druhým extrémem: Každé vydání každého časopisu je samostatná třída, která má tolik instancí, kolik čísel daného vydání v knihovně máme. Obojí lze naprogramovat tak, že bude dobře fungovat.

Je potřeba přiznat, že neexistuje žádné jednoznačné pravidlo, jak se v takovém případě rozhodnout. Hlavním kritériem je ale analýza **chování a prezentace objektů navenek**. Pokud nedokážeme najít odlišnosti v chování, operacích a vnějších atributech, tak nemá smysl model štěpit na více tříd. A nezapomínat na to, že stále ještě můžeme používat i množiny objektů – viz. předchozí odstavec.

Dalším podkladem pro rozhodování jsou ještě

- zohlednění potenciální databázové realizace a
- potřeba usnadnit či alespoň nezkomplikovat možné úpravy a rozšiřování systému

Náš příklad lze tedy například vyřešit tak, že třídy budou končit na úrovni počítačového, lékařského, ... časopisu. Jednotlivé výtisky jednotlivých časopisů budou potom objekty lišící se svými atributy a náležející do různých množin, protože různé typy objektů není třeba vždy implementovat různými třídami.

## 4 Objektově orientované databáze – Gemstone

### 4.1 Úvod

První objektově orientované databáze se objevily již ve druhé polovině 80. let a vznikly na základě potřeby uchovávat a databázově zpracovávat v pokud možno nezměněné podobě data z programů napsaných v tehdy se rozvíjejících objektově orientovaných programovacích jazycích. Ve srovnání s relačními databázemi, které v té době byly na vrcholu vývoje, to byly systémy velmi neefektivní a málo výkonné, protože se jednalo o experimentální programy psané jako aplikace v nějakém objektovém programovacím jazyce.

Po více než deseti letech vývoje je však situace jiná. Z praxe již známe případy, kdy nasazení objektové databáze vyřešilo problémy, které relační systém nedokázal zvládnout. Objektové databázové aplikace se objevují již i v ČR. Dnešní objektové databáze mají srovnatelný výkon s velkými relačními systémy – zvládají stovky transakcí za sekundu a tisíce současně připojených uživatelů. S objektovými databázovými aplikacemi se můžeme setkat například v informačních systémech letového provozu (např. Finair, Air France), rezervačních systémech osobní letecké dopravy (např. Fractal s.r.o. u nás v Praze, TourisNet Gran Canaria), informačních systémech pro řízení dopravy zboží (např. Orient Overseas Container Line – jeden z 3 největších dopravců mezi USA a Evropou), informačních systémech dodavatelů elektřiny (např. Floria Power & Light), rezervačních hotelových služeb (např. Navigant International Northwest Travel), systémů pro pojištění (např. povinné ručení automobilů v Argentíně) a další. Objektové databáze také používá pro speciální aplikace mnoho firem v kombinaci s relačními. Jsou to například firmy Texas Instruments, BMW, Opel, Ford, JP Morgan, IBM, Hewlett Packard, AT&T a další.

### 4.2 Objektově orientované a objektově relační databáze

Databázové systémy jsou založené na různých datových modelech. Jde o datový model síťový (a jeho variantu hierarchický datový model), relační, objektově relační a objektově orientovaný. (Někteří autoři také považují za databázové datové modely ještě modely fulltextové, hypertextové a modely založené na sémantických sítích).

Dnes dominuje relační datový model, který ve většině aplikačních oblastí postupně nahradil databáze založené na síťovém datovém modelu. Dnešní praxe však ukazuje, že relační databáze začínají být postupně nahrazovány databázemi objektovými. Pod obecným označením „objektové databáze“ se však skrývají dva vzájemně odlišné datové modely:

1. Objektově relační datový model představuje evoluční trend vývoje. Jde o doplnění relačního datového modelu o možnost práce s některými datovými strukturami, které známe z oblasti objektově orientovaných programovacích jazyků. Většina výrobců velkých relačních databázových systémů (např. Oracle) zvolila tuto variantu. Objektově relační datový model ale ve svých principech zůstává původním relačním datovým modelem.
2. Objektově orientovaný datový model, který představuje revoluční trend vývoje. Jde o nový datový model, který není postaven jako rozšíření relačního datového modelu. Do jisté míry zde jde o renesanci původního síťového datového modelu, který je doplněn o možnost práce s objekty tak, jak je známe z objektového programování.

Relačně objektová technologie je dnes rozšířenější. „Nerelační“ objektově orientovaný datový model má však následující přednosti:

1. Lépe podporuje datové struktury, které známe z objektově orientovaných programovacích jazyků. Není třeba datové struktury tolik transformovat, aby byly uložitelné do databáze. Existují již prakticky použitelné systémy (např. Gemstone, ObjectStore, O2, Versant, ...), které

dovolují v databázi zpracovávat objekty ve stejném tvaru, jak se s nimi nakládá v objektových programovacích jazycích.

2. Protože navazuje na síťový datový model, tak má předpoklady pro efektivnější způsoby zpracování dotazů ve srovnání s relačním datovým modelem. Tato vlastnost se projevuje hlavně u složitých datových struktur, které by se podle relačního datového modelu musely rozkládat do mnoha vzájemně provázaných relačních tabulek.

Příčiny, proč jsou zatím objektové databáze méně v praxi rozšířené, než relační, jsou pravděpodobně následující:

1. Malá praktická znalost objektových databází v komunitě tvůrců softwaru.
2. Dostupnost systémů na trhu a jejich cena. Velké relačně objektové systémy jsou levnější než objektové. (například komerční cena systému Gemstone je asi 2x větší než Oracle).
3. Konservativní myšlení potenciálních uživatelů a samozřejmě také potřeba zpracovávat již vytvořené báze dat v relačních systémech.
4. Chybějící standardy a nedostatečná podpora metod analýzy a návrhu. Návrh standardu ODMG 3.0 není všemi výrobci respektován. Modelovací jazyk UML nepodporuje všechny konstrukce potřebné k modelování objektové databáze. Metody používané pro návrh relačních databází nejsou vhodné k plnému využití možností objektových databází.

Přes uvedené problémy však existují důvody se domnívat, že význam objektových databází v blízké budoucnosti poroste, protože již dnes existuje celá řada aplikací, kde objektové databáze prakticky prokazují svoje přednosti. Společnou vlastností těchto aplikací je velké množství komplexních datových struktur a jejich proměnlivost za chodu systému, které způsobují problémy relačním databázím. Takové systémy mohou pracovat až se stovkami a tisíci různých vzájemně poskládaných datových typů reprezentovaných třídami objektů. Dotazy nad takovými objekty ještě navíc vyžadují vysokou míru vzájemného polymorfismu. (V takových systémech kupříkladu potřebujeme klást dotazy nad množinami obsahující prvky různého typu. A zároveň očekáváme, že při přidání nového datového typu se nebudou muset přepisovat již hotové dotazy.) Typickým příkladem takových systémů jsou datové sklady, které jsou charakteristické dlouhodobým shromažďováním velkého množství nově vznikajících různorodých dat. Takové systémy jsou charakteristické nejen pro řízení velkých podniků, ale také v různých evidenčních systémech státní správy, zdravotnických systémech, informačních systémech obsahujících ekologické informace, zemědělských informačních systémech, historiografických informačních systémech atp.

Na druhou stranu je třeba poznamenat, že relační databáze fungují velmi dobře v oblastech, kde během života systému nedochází k požadavku na změnu struktury databáze a na přidávání dalších datových typů. Relační systém může být výkonný i pokud se databáze skládá z velkého množství záznamů, ale uložených v malém počtu jednoduše strukturovaných relačních tabulek.

Bohužel se v ČR dnes objektovými databázemi žádné pracoviště soustavně nezabývá. Dílčí práce byly vykonány v první polovině 90. let na Fakultě elektrotechniky a informatiky VUT Brno a na Matematicko fyzikální fakultě Komenského univerzity v Bratislavě. Práce obou týmů vedly ke konstrukci experimentálních databázových systémů. Slovenský systém byl později dopracován ve finském projektu tvorby metamodelovacího nástroje na universitě v Jyväskylä – nyní jedním z celosvětově používaným CASE nástrojem Metaedit™. Dnes je situace taková, že na univerzitách v ČR se až na výjimky objektové databáze neobjevují ani ve výuce.

Ve světě je několik univerzitních pracovišť, které se objektovými databázemi zabývají (např. CERN, Université de Genève, Vrije Universiteit Brusel a celá řada v USA jako např. MIT a Stanford University). Výsledky jejich práce jsou využívány v praxi při konstrukci objektových databází. Na internetu existuje mezinárodní sdružení ODMG – Object Database Management Group ([www.odmg.org](http://www.odmg.org)).

Problematika objektových databází je od poloviny 90. let diskutována na odborných konferencích. Od konce 90. let vycházejí v zahraničí odborné publikace, které se zabývají především vlastnostmi vybraných objektových databází. Přestože již existuje mnoho důležitých teoretických prací, které jednotlivě dokazují účelnost objektově orientovaného datového modelu v databázových systémech, tak se zatím v oblasti metod analýzy a návrhu objektových databázových aplikací používají jen postupy původně určené pro práci s relačními systémy a nebo jen intuitivní přístupy založené na zkušenosti s imperativními objektově orientovanými programovacími jazyky.

### 4.3 Objektově orientovaný datový model

Hlavním motivem pro vznik objektového datového modelu (ODM) byly problémy s ukládáním a zpracováním objektů v relačních databázích. Relační datový model (RDM) objektovému programování nevyhovuje, protože je příliš jednoduchý. Z tohoto důvodu vznikl tlak na konstrukci nových databázových systémů, které by lépe dokázaly pracovat s objekty.

Objektový a relační datový model se od sebe výrazně liší. Tabulky jsou v ODM pouze jedna z možných forem výstupní prezentace uložených dat. ODM se nicméně může podobat strukturám síťových databází, jak jsme je znali v systémech IDMS. Na ODM můžeme nahlížet jako na renesanci síťového datového modelu. Při určité míře zjednodušení lze připustit vztah:

*síťový datový model + objektové typy dat + polymorfismus = objektový datový model.*

Vyjmenujme si nyní základní charakteristiky objektového datového modelu:

1. Objektová databáze podporuje více typů množin objektů. (na rozdíl od relačního datového modelu, kde je relační tabulka jediným „druhem množiny“). Společně se označují termínem *collection* (české označení zatím chybí, většina autorů používá termín „sada“ či „kolekce“). V konkrétních databázových systémech to může být až několik desítek různých typů různých vlastností tak, jak je známe z knihoven objektových programovacích jazyků. (Je to například Array, List, OrderedCollection, SortedCollection, Set, Bag, Dictionary, ...)
2. Objektová databáze rozlišuje mezi pojmem třída objektů a množina (kolekce) objektů. Třída je jen realizace datového typu objektů a množina je jen úložiště pro objekty. Na rozdíl od tabulek v RDM, kde role třídy a množiny splývají dohromady, v ODM nemusíme pracovat pouze s množinami, které obsahují jen všechny objekty jedné třídy. Můžeme mít například více množin objektů stejného typu i množinu obsahující objekty z různých tříd. (Pokud takové objekty mají díky polymorfismu nějaké společné atributy, tak nám nic nebrání je držet pohromadě a nad takovou množinou provádět například selekci.)
3. Objekty se skládají z vnitřních datových složek (což mohou být opět jiné objekty) a z metod, které představují funkční stránku každého objektu. Známe nejen tzv. přístupové metody, které jen přímo manipulují s datovými složkami objektu (zapisují nové hodnoty datových složek a nebo čtou hodnoty datových složek), ale i metody složitější, které vypočítávají z objektů taková data, jenž v objektu nebyla jednoduše uložena jako jedna z jeho datových složek. Toto známe z objektového programování. Pro ODM je ale důležité si uvědomit, že mezi atributy objektů patří nejen jejich datové složky (jako v RDM), ale i metody poskytující další data. (Dále v textu je jeden takový příklad atributu „věk“ osoby.)
4. Polymorfismus objektů nevzniká pouze děděním tříd. Pokud mají objekty společné atributy, tak jsou polymorfní i když jejich třídy mezi sebou nedědí.
5. Každý objekt má svoji vlastní identitu, což v objektové databázi znamená, že v rámci jednoho paměťového prostoru má každý objekt systémem přidělen jednoznačný identifikátor obvykle označovaný jako OID (Object Identifier). OID plní úlohu ukazatele do virtuální paměti. OID každého objektu zůstává stejný, i když se v objektu změní všechny jeho datové složky nebo metody. OID se také samozřejmě nemění při změnách objektu na fyzické úrovni, jako např. změna jeho umístění v operační paměti nebo na disku. Vzhledem k existenci konceptu OID můžeme rozlišovat mezi pojmem rovnost dat objektu a totožnost objektu (Dva objekty se

shodnými daty ještě nemusí být totožné). Praktický důsledek konceptu OID je ten, že v ODM není potřeba objektům vytvářet primární klíče. Toto RDM nezná, neboť identita relačních záznamů je dána jen hodnotami atributů. V některých objektových databázích (např. Gemstone) mohou OID zůstat skryté pod aplikačním rozhraním SRBD. V takové databázi potom její uživatel vidí objekty, které se přímo propojují a skládají mezi sebou.

6. V ODM lze v bázi dat pracovat i s takovou soustavou objektů, která je sama o sobě aplikací. Objektová databáze nemusí sloužit jen jako úložiště dat, se kterým manipuluje externí program. Algoritmy programu lze „rozpustit“ v metodách objektů přímo uložených v objektové databázi. Tvorba databázové aplikace na straně klienta je potom velmi zjednodušená, protože v extrémním případě se může jednat jen o prezentační rozhraní výpočetního systému, který celý pracuje „uvnitř“ objektového databázového serveru.
7. Na rozdíl od běžných objektových programovacích jazyků mohou objekty v ODM migrovat mezi různými třídami, v systému může existovat současně více verzí jedné třídy. Různí uživatelé podle svých přístupových práv mohou mít dostupné různé atributy na stejných objektech.

Na závěr si sobě odpovídající pojmy relačního a objektového datového modelu porovnáme v následující tabulce:

RDM	ODM
záznam (řádek tabulky)	objekt (prvek množiny)
tabulka	1) třída objektů (jako datový typ) 2) množina objektů (i z různých tříd)
atribut (položka řádku tabulky)	1) datová složka objektu 2) metoda objektu, která poskytuje data
primární klíč (není ukazatelem do paměti)	OID (je ukazatelem do paměti)

Tab. 1. Porovnání relačního a objektového datového modelu

## 4.4 Jak vytvořit objektovou databázovou aplikaci

Objektový datový model není nadstavbou relačního datového modelu. Je tedy otázkou, jaké metody návrhu použít. Pro relační datový model je k dispozici datová normalizace, metoda syntézy atributů a metoda datové dekompozice podle funkčních závislostí atributů. Bohužel pro objektový datový model zatím není žádná všeobecně uznávaná a používaná technika nebo metoda návrhu. Je možné převzít relační techniky, ale potom dostaneme jen „relační databázi v objektovém prostředí“ a nevyužijeme všechny vlastnosti, které ODM má. Jinou možností je převzít schéma objektů a tříd tak, jak jsou navrženy v aplikaci, která má s databází pracovat. To už je lepší, protože právě proto byl objektový datový model vyvinut. Jenomže struktura objektů výhodná pro aplikaci může komplikovat jejich efektivní databázové zpracování.

Čtenáři se mohou v různých pramenech setkat s různými tvrzeními o objektových databázích, které tuto problematiku zjednodušují a prohlašují například, že objektovou databázi není třeba normalizovat a že objektová databáze je mnohonásobně rychlejší než relační.

Tvrzení o rychlosti platí, ale jen pro případ, kdy se podaří navrhnout objektové schéma tak, aby obsahovalo přímo propojené objekty mezi sebou na rozdíl od relační databáze, která musí používat spojení od cizího klíče z jedné tabulky na primární klíč druhé tabulky. Tedy jinými slovy pokud se podaří objektové schéma navrhnout v duchu zásad síťového datového modelu.

S normalizací to je ještě složitější. Uvažuje se o využití refaktoringu a návrhových vzorů. Určitý pokrok učinil Kent Beck, jeden z pionýrů agilních metodik, ve své knize o metodách refaktoringu prezentuje první tři „objektové normální formy“, které se týkají správného návrhu struktury tříd objektů a jsou odvozeny z relačních normálních forem. V relačním datovém modelu jsou jednotkou funkční závislosti samotné atributy – tedy datové složky v záznamech a ne celé záznamy. Proto se bez normalizace v RDM

neobejdeme. Objektový datový model pracuje s objekty, které navenek vystupují jako nedělitelné jednotky. Proto není problém normalizace tak naléhavý jako v RDM, ale neznamená to, že neexistuje.

Při tvorbě objektové databáze jsme tedy zatím odkázáni jen na zkušenost. Lze však popsat postup, jak objektovou databázi vytvořit:

1. Sestavit konceptuální model úlohy. Zde je možné použít diagram tříd UML nebo Chenův konceptuální ER diagram s rozšířením na vazby generalizace-specializace. Zatím ale modelujeme jen množiny. Atributy zde rozpoznané budou později implementovány nejen jako datové složky ale také jako metody objektů. V tomto kroku se to ještě nerozlišuje.
2. K prvkům množin najít potřebné třídy. Bohužel diagram tříd UML nemá zvláštní symboly pro množiny a pro třídy. Pokud máme množiny z objektů jen jedné třídy, tak to nevadí, ale jinak musíme použít stereotypy nebo přidat nový symbol.
3. Rozhodnout, které atributy budou implementovány datovými složkami a které metodami. Tyto metody pak naprogramovat.
4. Naplnit databázi daty. Tedy vytvářet objekty (instance) tříd a ukládat je do vybraných množin. Vytvořit objekt jako instanci třídy nestačí. Takový objekt totiž ještě není prvkem žádné množiny v databázi.

## 4.5 Gemstone

Gemstone je název objektově orientovaného databázového systému založenému na třídě instancním datovém modelu. Systém nemá nic společného s relačním datovým modelem. Tabulky jsou zde pouze jedna z možných forem výstupní prezentace uložených dat. Datový model Gemstone se nicméně může podobat strukturám síťových databází.

### 4.5.1 Historie Gemstone

Gemstone se vyvíjí od poloviny 80. let. Systém vznikl od počátku jako databázově rozšířený Smalltalk. Tyto rysy si uchoval dodnes a tak je někdy přezdíván jako "databázový Smalltalk". Už v roce 1988 byla k dispozici verze 2.0, která je považována za první prakticky použitelnou a relativně úspěšnou. Současná verze je 6.1. Systém dodržuje standardy CORBA a ODMG.

### 4.5.2 Vlastnosti Gemstone

Gemstone je databázový systém, který běží na platformách Windows, Linux, HP-UX, Solaris a AIX. Lze jej popsat pomocí následujících vlastností:

1. **víceuživatelský objektový server.** Gemstone podporuje až 1000 současně připojených uživatelů, 100 GB dat v jedné databázi a až 100 transakcí za sekundu. Jedna báze dat může být kvůli zvýšení výkonu distribuována mezi více počítačů.
2. **programovatelný systém.** Základním jazykem je Smalltalk DB. Jedná se nejen o jazyk pro definici a manipulaci databázových dat, ale také o plně funkční univerzální programovací jazyk.
3. **klient-server systém.** Gemstone obsahuje rozhraní na jazyky Smalltalk, Java, C, C++. Kromě toho je na server napojitelný jakýkoliv další systém podle standardu CORBA.
4. **transparentní systém.** Rozhraní Gemstone pomocí tzv. konektorů propojuje objekty na serveru s jejich reprezentanty na straně klienta tak, aby aplikační programátor pracoval ve svém programovacím jazyce stejným způsobem s objekty, které jsou jen lokální v jeho aplikaci, jako s objekty uloženými v databázi. V kódu databázové aplikace není třeba data z databáze nějak načítat a poté konvertovat do lokální podoby a pak zpětně konvertovat a ukládat do databáze.

5. **pesimistické i optimistické řízení transakcí.** Pesimistický režim je shodný s režimem, jak jej známe z relačních databází: Je-li nějaký údaj vlivem právě probíhajících operací nějaké transakce uzamknutý, tak není s ním možné pracovat z jiné transakce. Optimistický režim naopak data neuzamkává a případné kolize více transakcí se řeší až v době pokusu o uzavření každé transakce.
6. **připojitelnost na externí zdroje dat.** Gemstone má SQL rozhraní na klasické relační databáze, které dovoluje oběma směry synchronizovat objektovou bázi dat s externí relační databází. Kromě toho podporuje standard CORBA a do serveru lze přidávat uživatelské moduly napsané v jazyce C.
7. **bezpečnost a správa uživatelských účtů.** Na rozdíl od běžného Smalltalku dovoluje Gemstone definovat k objektům různá přístupová práva pro různé uživatele podobným způsobem, jako jiné velké databáze.

## 4.6 Programovací jazyk Smalltalk DB

Jazyk Smalltalk DB je databázovým rozšířením programovacího jazyka Smalltalk-80. Syntaxe jazyka Smalltalk je jednoduchá, ale velmi odlišná od jazyků z rodiny C. Smalltalk elegantně využívá výhody třídně-instančního objektově orientovaného modelu, tj. skládání objektů, dědění, závislosti mezi objekty, polymorfismu 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.

Kód Smalltalku se skládá pouze z posílání zpráv objektům. Všechny konstrukty včetně větvení výpočtu, iterací nad sadami objektů či posílání dat jsou ve skutečnosti různé zprávy posílané příslušným objektům.

Důležitou součástí Smalltalku pro práci s objektovými databázemi jsou bloky výrazů. Bloky výrazů představují vyčleněné části kódu. Blok výrazů je jako celkem také objektem, může být pojmenován, mohou mu být posílány příslušné zprávy, a může být použit v jiných výrazech (zprávách) jako příjemce nebo parametr. Výrazy v blocích se vyhodnocují vždy až při požadavku na jejich spuštění, a ne hned při vytvoření bloku. Tentýž blok proto může v různých situacích vracet různé výsledky. Následující příklad ukazuje blok kódu, který je použit jako parametr zprávy, která má za účel vybrat z pole čísel čísla větší než 5. Toto pole čísel  `#(7 8 3 2 6)` je objekt, který je příjemcem zprávy `select:` s parametrem `[:x | x > 5]`:

```
#(7 8 3 2 6) select: [:x | x > 5].
```

Stejný příklad je také možné napsat tak, že blok, který je použit v parametru zprávy, bude uložen do samostatné proměnné:

```
B := [:x | x > 5].
#(7 8 3 2 6) select: B.
```

S takto vytvořeným objektem `B` je samozřejmě možné dále samostatně pracovat. Například posílání zprávy `value:` s parametrem `10`, což se píše jako

```
B value: 10.
```

nám dává výsledek hodnotu `true`. Pozorný čtenář si jistě všiml podobnosti bloků ve Smalltalku a lambda výrazů (`λx . x>5`) nebo podobnosti se zápisem užívaným v algebraických výrazech jako `{∀x | x>5}`, kterými je Smalltalk charakteristický.

### 4.6.1 Architektura programů ve Smalltalku

Smalltalk nevyužívá přímo strojový kód počítače, na kterém běží. Namísto toho překládá programy do byte kódu, který je za běhu interpretován do hostitelského strojového kódu. Tato koncepce inspirovala například tvůrce Javy a nebo systému OS400 od IBM.

U nejnovějších verzí Smalltalku, který tvoří současný standard Smalltalkových systémů, je jeho architektura navržena jako dynamický překladač, kdy se za chodu aplikace přeložené části binárního kódu Smalltalku ukládají do vyrovnávací paměti, čímž je dosahováno dobré výkonnosti beze ztráty pružnosti systému. Binární kód Smalltalku je navíc asi 3x až 5x kompaktnější než odpovídající strojový kód a je nezávislý na použitém hardwaru počítače. Programy (hotové i rozpracované) mohou být okamžitě přenositelné z platformy na platformu od MacOS a Windows k Linuxu a mnoha typům unixů. K tomuto přispívá nezávislý model grafických objektů a diskových souborů v systémové knihovně.

Smalltalk podporuje koncepci metatříd (viz. další odstavec), paralelní programování (má podporu pro řízení procesů jako např. semaforey nebo vyrovnávací buffery), z dalších vlastností např. ošetřování výjimek v programu, sledování verzí kódu během programování s možností návratu do libovolného z přechozích stavů aj. Programátor může do proměnných přiřazovat nejen data, ale i kód.

Pro práci s databázemi ve Smalltalku je také důležité vědět, že na třídy je také nahlíženo jako na objekty. Mohou být za chodu programu ukládány do proměnných, mohou být parametry zpráv, mohou vznikat, zanikat nebo měnit svůj datový obsah, kterým jsou například kódy metod nebo vazby dědičnosti. A podobně jak se dá zacházet s bloky, tak lze i s metodami samotnými. Objekty, které realizují třídy, jsou dokonce také instancemi jiných tříd. Tyto třídy "vyšší" úrovně jsou již zmíněné metatřídy.

### 4.6.2 Rozdíly mezi Smalltalkem-80 a Smalltalkem DB

Smalltalk-80 je označení pro jazyk, jehož syntaxe byla publikována v roce 1980. Od té doby se jazyk i jeho knihovna stále vyvíjí, ale označení Smalltalk-80 zůstává, protože je dodržováno pravidlo zpětné kompatibility kódu. Nejuznávanějším vývojovým prostředím Smalltalku-80 je systém VisualWorks firmy Cincom. (*Další kvalitní implementace Smalltalku jsou STX a nebo Squeak.*) Tato firma spolupracuje s firmou Gemstone a proto je VisualWorks nejpřirozenější prostředí pro klienty Gemstone i pro správu samotného serveru.

Smalltalk DB je klon standardního Smalltalku. I když se jedná o databázový jazyk, tak si plně zachovává vlastnosti programovacího jazyka. To znamená, že pod serverem Gemstone lze sestavit a spouštět velmi podobné aplikace, jako ty, které se programují v "obyčejném" programovacím prostředí. Server tedy nemusí sloužit jen k ukládání a vybírání dat, ale lze na něj uložit téměř libovolnou část algoritmu aplikace.

Vlastnost	Smalltalk-80	Smalltalk DB
knihovna pro práci s <i>collections</i>	ano	ano (i s indexováním)
knihovna pro práci s <i>magnitude</i> (čísla, znaky, datумы, ...)	ano	ano
knihovna pro práci se streamy a soubory na disku	ano	ano
paralelní procesy, semaforey	ano	ano
knihovna pro grafiku	ano	ne
migrace instancí mezi třídami	ne	ano
více verzí jedné třídy	ne	ano
persistence objektů	ne	ano (ORB i replikacemi)
transakce, uzamykání	ne	ano
<i>selection blocks</i>	ne	ano

*Selection block* je zvláštní typ klasického Smalltalkového bloku, který obsahuje logický výraz odkazující se na datovou hierarchii objektů (= propojení objektů skládáním) v databázi. Zapisují se složenými závorkami a datové propojení se zapisuje tečkou:

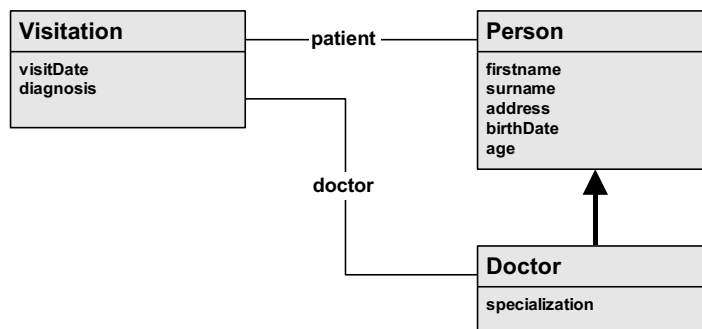
```
Companies select: {:c | c.director.name = 'Smith'}.
```

Na tomto příkladě je dotaz nad množinou objektů `Companies`, který má vybrat ty firmy, jejichž ředitel se jmenuje Smith. Předpokládá se, že každý prvek množiny `Companies` má odkaz `director` na svého ředitele a každý ředitel má odkaz `name` na svoje jméno.

## 4.7 Příklad objektové databáze

### 4.7.1 Popis úlohy

Vlastnosti objektové databáze budou prezentovány pomocí následující úlohy: Mějme jednoduchou databázi pro evidenci návštěv v ordinacích lékařů. U každého pacienta bude evidováno jeho jméno, příjmení, adresa, datum narození a věk. U každého doktora ještě navíc jeho specializace. U návštěvy pacienta u lékaře bude evidováno datum návštěvy a diagnóza. Není vyloučeno, aby se jeden lékař stal druhému lékaři pacientem. Úlohu znázorňuje následující obrázek:

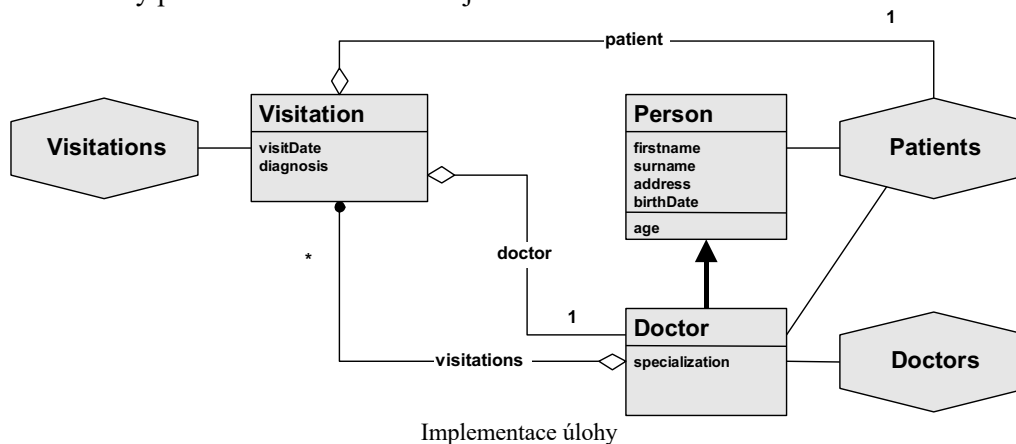


### 4.7.2 Implementace úlohy

1. Požadované atributy objektů je možné implementovat nejen jako jejich datové položky, což by odpovídalo možnostem relačních databází, ale také jako metody objektů. V našem příkladu může být takto řešen atribut `age`, který je vypočitatelný metodou z data narození.
2. V aplikaci lze navrhovat nejen třídy objektů, ale také množiny objektů, které mohou obsahovat jako prvky objekty z různých tříd. V našem příkladu to může být množina `Patients`, která bude obsahovat jako svoje prvky instance třídy `Person` i `Doctor`. Další množiny `Doctors` a `Visitations` budou obsahovat pouze instance jedné třídy a tím tedy budou podobné relačním tabulkám.
3. Protože objektové vývojáře tolik nesvazují požadavky normálních forem, tak si lze zvolit směr, kterým na sebe budou objekty odkazovat. Například je možné odkazovat objekt třídy `Doctor` z objektu třídy `Visitation` a nebo rovněž tak odkazovat z objektu třídy `Doctor` na podmnožinu objektů třídy `Visitation`. Záleží jen na analýze zadání úlohy, která zkušenému vývojáři napoví, jaký směr vazby je pro praktický chod databáze přirozenější. Pokud je nějakým způsobem zabezpečena konzistence báze dat, tak je možné realizovat oba dva směry vazby současně. Tak tomu bude v našem příkladu u vazby mezi doktory a vyšetřeními.

4. Podobně jako směr vazby, tak na vývojáři záleží, jaké množiny objektů zveřejní uživatelům databáze. V naší úloze totiž není nezbytně potřeba zveřejňovat všechny množiny. Pokud například nebude třeba klást příliš často dotazy nad množinou všech doktorů, tak množina *Doctors* není nutná, protože data o doktorech jsou tranzitivně dosažitelná z objektů třídy *Visitation*. V našem příkladě jsou ale z didaktických důvodů všechny množiny ponechány.

Popsaná implementace je znázorněna na následujícím diagramu, kde kromě standardních symbolů UML jsou ještě šestihrany pro znázornění množin objektů:



### 4.7.3 Program v jazyce Smalltalk DB databázového systému Gemstone

Program musí nejprve vytvořit příslušné třídy s metodami, potom vytvořit potřebné množiny a nakonec do nich uložit data, což budou instance vytvořených tříd. V praxi je výhodné samozřejmě v co největší míře využívat vizuální programovací nástroje prostředí. Zde budou ukázány jen zdrojové kódy v jazyce Smalltalk DB, který je databázovým rozšířením univerzálního objektového programovacího jazyka Smalltalk. Databázový systém Gemstone samozřejmě na straně klienta podporuje i jiné programovací jazyky (např. Java a C++) a dovoluje objekty v nich vytvořené konvertovat a mapovat na objekty ve své databázi. Gemstone má také relační interface dodržující standard SQL-92 a objektový interface kompatibilní s CORBA 2.0.

Nejprve vytvoříme třídy. Lze k tomu použít vizuální nástroje. Gemstone má ale také znakový příkazový řádek. Zde je ukázka kódu pro definici třídy *Person* a třídy *Doctor*, která z ní dědí:

```

Object subclass: 'Person'
  instVarNames:
    #(firstName surname
      address birthDate)
  inDictionary: UserClasses.

Person subclass: 'Doctor'
  instVarNames: #(specialization)
  inDictionary: UserClasses.
  
```

Dále je třeba napsat potřebné metody. Na ukázce je metoda *age* třídy *Person*, kterou je realizován atribut věk osob:

```

Method: Person
  age
  ^birthDate isNil
  ifTrue: [nil]
  ifFalse:
    [((Date today subtractDate: birthDate) / 365.2422) truncated]
  
```

V případě oboustranného propojení mezi doktory a vyšetřeními je pro udržení konzistentní databáze výhodné využít možnosti objektového programování a definovat například k doktorům metody na přidávání a odebrání vyšetření tak, aby byla zajištěna konzistence i „z druhé strany“:

```
Method: Doctor
addVisitation: aVisitation
    visitations add: aVisitation.
    aVisitation doctor: self.
```

```
Method: Doctor
removeVisitation: aVisitation
    visitations remove: aVisitation.
    aVisitation doctor: nil.
```

Dále bude třeba vytvořit množiny objektů. Množiny objektů se ukládají do logické paměťové oblasti UserGlobals:

```
UserGlobals at: #Doctors put: Set new.
UserGlobals at: #Patients put: Set new.
UserGlobals at: #Visitations put: Set new.
```

A nyní už zbývá jen naplnit data. Pro uživatele manipulace s daty (jako ostatně všechno) probíhá zcela podle zásad objektového programování, jak ukazuje následující ukázka:

```
d := Doctor new
    firstname: 'Jan';
    surname: 'Novak';
    address: 'Novakova 12';
    birthDate: '4-JAN-1950';
    specialization: 'obvodni';
    visitations: Set new.

p := Person new
    firstname:
    'Karel';
    surname: 'Noha';
    address: 'Zikova 56';
    birthDate: '3-JAN-1954'.

v := Visitation new
    visitDate: '12-DEC-2000';
    diagnosis: 'angina'.
```

Objekty *d*, *p*, *v* je ještě třeba propojit (propojujeme je přímo – spojení odkazem z cizích na primární klíče nepotřebujeme):

```
d addVisitation: v.    v patient: p.
```

a nakonec vložit do množin:

```
Doctors add: d.    Patients add: p.    Visitations add: v.
```

Na následujícím obrázku vidíme ukázkou práce v GUI.

VisualWorks 7.2 Client of Gemstone 6.1

File System Browse Debug Painter Store Tools GemStone Window Help

Logged in Session 1 (linked) for 'student2' on '!@193.84.35.160!gemserver61'

Default\*

GemStone

Browse Admin Tools

OODB Query Tool

GS evaluation evaluate

Doctors select: {d | d.visitations.\*.patient.surname = 'Novak'}.

firstname	surname	address	birthDate	specialization
Ivan	Drago	Sofia, Duga 71	January 1, 1979	surgeon
			st 31, 1977	surgeon
			ary 4, 1950	practitioner

Person >> age

Browser Edit Find View Category Class Protocol Method Tools Help

Category Hierarchy Instance Class Shared Variable

UIPainter-Tools Person accessing age  
 UserClasses Doctor methods birthDateFromS  
 Windows Goodies Visitation name

Source Rewrite Code Critic Bytecode

```

age
^birthDate isNil
  ifTrue: [nil]
  ifFalse: [((Date today subtractDate: birthDate) / 365.2422) truncated]
  
```

Method: #age (metho Par

GemStone Classes Browser on: Session 1 (linked) for 'student2' on '!@193.84....

UserClasses Doctor accessing age  
 UserGlobals Person methods birthDateFromStr  
 Globals Visitation name  
 Published instance class printOn:

```

age
^birthDate isNil
  ifTrue: [nil]
  ifFalse: [((Date today subtractDate: birthDate) / 365.2422) truncated]
  
```

dotazovací konzole

kód na straně klienta

kód na serveru

## 4.8 Příklady dotazů

Následujících několik ukázek dotazů předvede dotazovací možnosti jazyka Smalltalk DB a také jazyka OQL, který je součástí standardu ODMG a podobá se jazyku SQL:

*Najdi vyšetření, které prováděl doktor Dyba:*

```
Visitations select: {:v | v.doctor.surname = 'Dyba'}.
```

```
SELECT *
  FROM v IN Visitations
 WHERE v.doctor.surname = 'Dyba';
```

Gemstone samozřejmě podporuje indexování. Pokud by bylo třeba zvýšit výkon právě předloženého dotazu, tak by správce databáze mohl vytvořit indexy například takto:

```
Visitations createEqualityIndexOn: 'doctor.surname'
  withLastElementClass: String.
```

*Najdi všechny doktory, kteří léčili pacienta Nováka.*

```
Doctors select: {:d | d.visitations.*.patient.surname = 'Novak'}.
```

```
SELECT *
  FROM d IN Doctors
 WHERE EXISTS
   (SELECT * FROM v IN d.visitations WHERE v.patient.surname = 'Novak');
```

Hvězdička v datové cestě parametru dotazu ve Smalltalku DB znamená, že bude třeba vyhledávat do šířky, protože atribut `visitations` doktora `d` není jedním objektem, ale celou množinou dalších objektů, z nichž každý má dále po cestě atributy `patient.surname`. OQL takovou schopnost nemá a tak musíme použít podmínku `EXISTS` na vnořený příkaz `SELECT`.

*Najdi adresy pacientů, kteří měli angínu:*

```
(Visitations select: {:v | v.diagnosis = 'angina'})
  collect: {:v | v.pacient.address}
```

```
SELECT v.pacient.address
  FROM v IN Visitations
 WHERE v.diagnosis = 'angina';
```

*Najdi pacienty starší 60 let.*

```
Patients select: {:p | p.age > 60}.
```

```
SELECT *
  FROM p IN Patients
 WHERE p.age > 60;
```

V podmínce se odvolává na atribut, který je počítán metodou. Využívá se tu také polymorfismu, protože v množině `Patients` jsou instance třídy `Person` i `Doctor`.

Tyto příklady byly záměrně vybrány tak, aby demonstrovaly přednosti propojení objektů díky síťové datové struktuře v objektové databázi. Z dotazů ve Smalltalku i OQL je na první pohled patrné, že srovnatelná relační databáze by byla komplikovanější a odpovídající dotazy složitější.

## 4.9 Shrnutí

Je pravda, že objektově relační databáze dokáží implementovat naši úlohu také. Především poslední verze Oracle dovolují využít hodně hybridních objektově-relačních vlastností. Cílem této kapitoly ale bylo prakticky ukázat čistý objektový datový model. Hybridní objektově relační technologie je totiž ještě méně standardizovaná, než objektová. Tvorba aplikací, která se o tento přístup opírá, se snadno dostane do vleku specifických funkcí posledních verzí konkrétního systému. Oproti tomu zde popsaný přístup je aplikovatelný nejen v Gemstone, ale v jakékoliv objektově orientované databázi (např. ObjectStore, O<sub>2</sub>, Cache, Ontos, Jasmine, ...).

Současná praxe bohužel pod pojmem objektové databáze chápe většinou jen různá rozšíření relačních databází. To je velká chyba, protože o „nerelačních“ databázích u mnoha inženýrů přetrvává představa, že to jsou systémy příliš exotické, málo výkonné a hlavně „nestandardní“. Analytici potom chybují v tom, že považují relační datový model za univerzálně platný přístup pro konceptuální modelování. Znalost objektového datového modelování proto považujeme v dnešní době za důležitou. Jestliže vývojáři z praxe budou znát jen relační datový model, byť jakkoli doplněný o hybridní přístupy, tak budou mít dojem, že „tabulky“ a vazby mezi nimi jsou jediný prakticky použitelný způsob analýzy, návrhu a implementace dat v informačních systémech.

## 5 Objektově relační mapování – ObjectLens

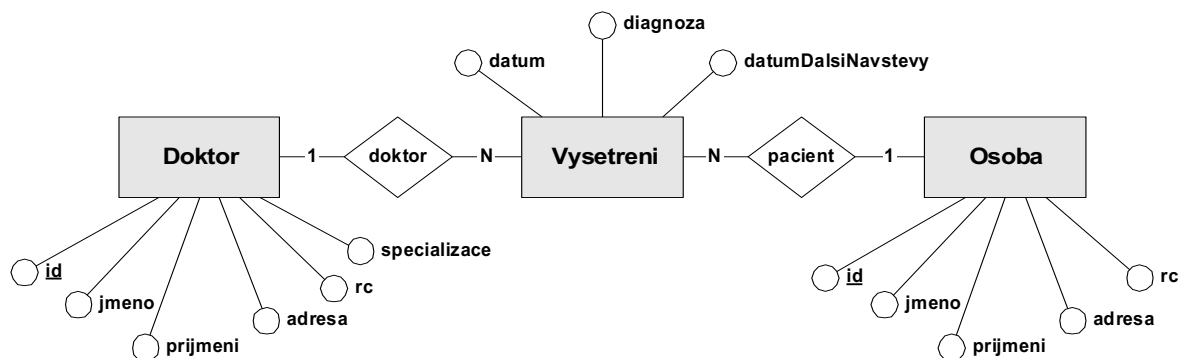
ObjectLens jsou zajímavým příkladem originálního řešení **relačně objektové databázové technologie**. Systém je součástí vývojového prostředí VisualWorks a umožňuje propojení objektů Smalltalku s daty uloženými na relačním databázovém serveru, což tedy znamená, že se jedná o nástroj pro tvorbu objektově relačních klient-server aplikací. Relačním serverem může být například jádro RDBMS systému Oracle a nebo jakákoliv relační databáze s rozhraním ODBC.

Na rozdíl od „běžnějších“ nástrojů pro tvorbu klient-server aplikací jako například produkty Gupta, Forte, Delphi, Optima, ... má systém ObjectLens následující vlastnosti:

- 1) Systém **transformuje relační datový model** na straně serveru na **omezenou variantu objektového datového modelu** na straně VisualWorks (klienta). Tato vlastnost bude podrobněji diskutována dále.
- 2) jazyk SQL se v systému ObjectLens používá především pro komunikaci s rozhraním relačního serveru. Na straně klienta lze s objekty pracovat výhradně pomocí programovacího jazyka Smalltalk, přičemž systém sám **transformuje dotazy v jazyce Smalltalk na odpovídající příkazy SQL** pro server.
- 3) **Transparentnost**. S objekty, které mají data uložena na relačním serveru se pracuje pomocí stejných příkazů a stejným způsobem jako s objekty, které jsou obsaženy pouze v paměti klienta jako běžné proměnné v programovacím jazyce.
- 4) **S výsledky dotazů lze pracovat jako s množinami objektů** shodným způsobem, jak bylo popsáno u objektových databází.
- 5) Systém umožňuje **tvorbu objektů a hierarchie tříd** na základě informace o struktuře databáze na serveru.
- 6) Systém umožňuje **tvorbu relačních tabulek na serveru na základě existujících objektů**, což znamená, že hotové lokální programy lze relativně jednoduše přeměnit na klient-server.
- 7) Systém může **současně pracovat jak s relačními databázemi, tak i s „nerelačními“ objektovými databázemi**.
- 8) Hotovou aplikaci vyvinutou v ObjectLens lze relativně jednoduše **přenést do prostředí „nerelační“ objektové databáze**.

### 5.1 Transformace relačního datového modelu do objektového

Podívejme se nyní podrobněji na transformaci relačního datového modelu na objektový datový model v prostředí ObjectLens. Jako základ si vezměme příklad lékařské ordinace. Stejný příklad byl použit v kapitole o objektových databázích. Nejprve vyjdeme z klasického relačního datového modelu, který je na serveru a který lze znázornit následujícím konceptuálním ER-diagramem:



### 5.1.1 Tvorba tříd objektů na základě entitních množin (tabulek)

V prvním kroku se v nástrojích ObjectLens **pro každou tabulku** na serveru vytvoří **samostatná třída objektů** (je možný i opačný proces, kdy se pro existující třídu vytváří tabulka). Pokud se bude potom pracovat s objekty (instancemi) takto vytvořených tříd, ObjectLens zajistí automatický datový přenos mezi objektem na straně klienta a datovými položkami odpovídajícího záznamu v relační tabulce na straně serveru. Na straně klienta se objekty chovají stejným způsobem jako například běžné proměnné a lze s nimi (i s jejich třídou) pracovat pomocí stejných operací a příkazů jazyka Smalltalk. Vytvořené třídy dovolují:

- **Definovat metody**, které rozšiřují protokol objektů o vlastnosti, které jsou dány metodami dávající hodnoty jako výsledek výpočtu nad datovými složkami objektu. Tyto metody také mohou pracovat s parametry a mohou také provádět shodné operace, jako v případě „nerelační“ objektové databáze. Jako příklad si ukažme metodu `prijdeDnes`, která bude vypadat úplně stejně jako v případě „nerelační“ objektové databáze:

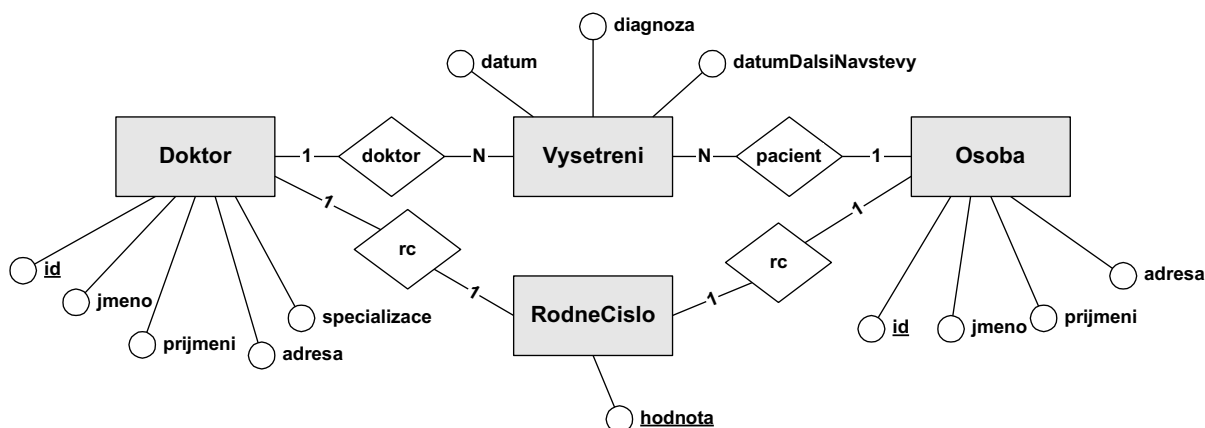
```
prijdeDnes
  ^ datumDalsiNavstevy = Date today.
```

- Mezi třídami využívat vazby **dědění**, **závislosti** a **delegování** stejným způsobem jako v objektové databázi.

Je samozřejmé, že všechny metody, vazby a vlastnosti těchto tříd zůstávají v paměti na straně klienta, protože relační datový model na straně serveru tyto informace nedokáže zpracovat.

### 5.1.2 Úprava relačního schématu

V předchozím kroku bylo ukázáno, že pro každou entitní množinu (tabulku) lze vytvořit samostatnou třídu, ve které lze definovat nejrůznější potřebné metody a vazby s ostatními třídami. V relační databázi však některá data nejsou implementována pomocí tabulek, ale jen pomocí atributů v jiných tabulkách. Pokud budeme potřebovat i k těmto údajům definovat metody a nebo s nimi chcít pracovat jako s objekty, musíme provést převod takového atributu na entitní množinu (tabulku). V našem příkladu je tímto atributem rodné číslo (**rc**) pacientů a doktorů. Po převodu atributu na entitní množinu bude naše databáze na serveru vypadat následovně:



Nyní, když je na serveru tabulka rodných čísel, tak můžeme i k této tabulce přiřadit na straně klienta třídu objektů. Tato třída samozřejmě dovoluje definovat nové metody `pohlavi`, `datum` a `vek`, které jsme již použili v kapitole o objektových databázích:

```
datum
  | x |
  x := hodnota copyFrom: 1 to: 6.
  (c := x at: 3) = $5 ifTrue: [x at: 3 put: $0].
  (c := x at: 3) = $6 ifTrue: [x at: 3 put: $1].
  ^ Date fromString: x.
```

**vek**

```
^ ((Date today - self datum) / 365.2422) truncated.
```

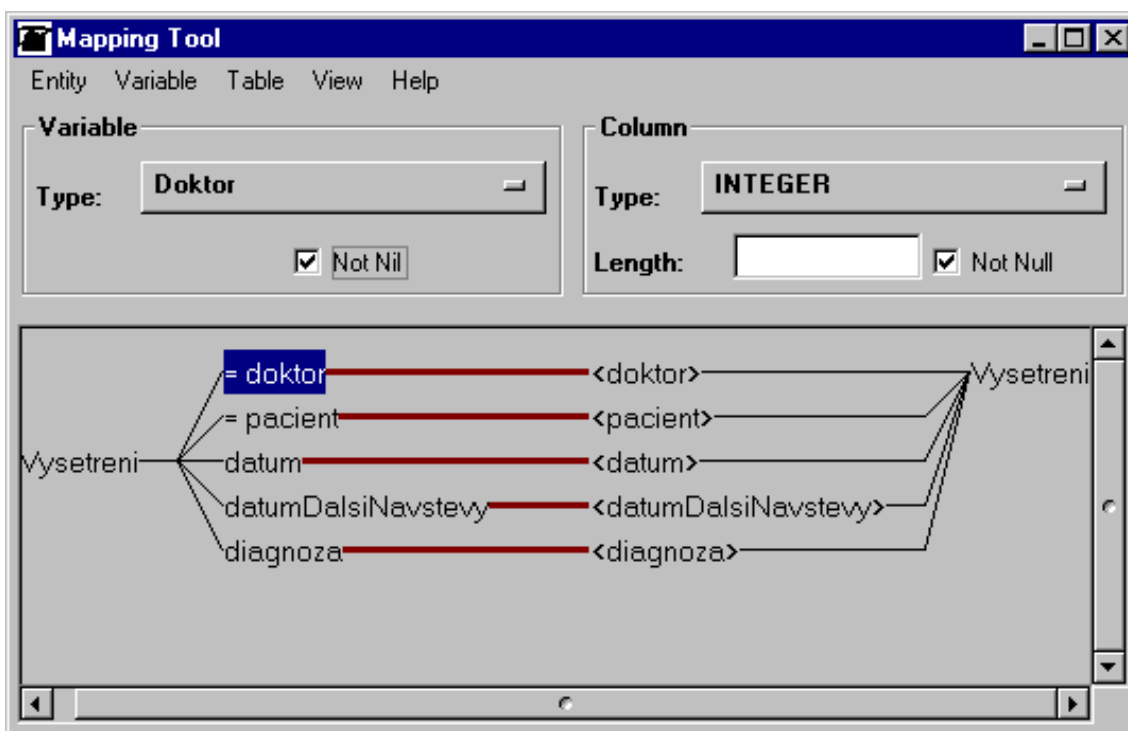
**pohlavi**

```
^ (hodnota at: 3) > $1 ifTrue: ['Z'] ifFalse: ['M'].
```

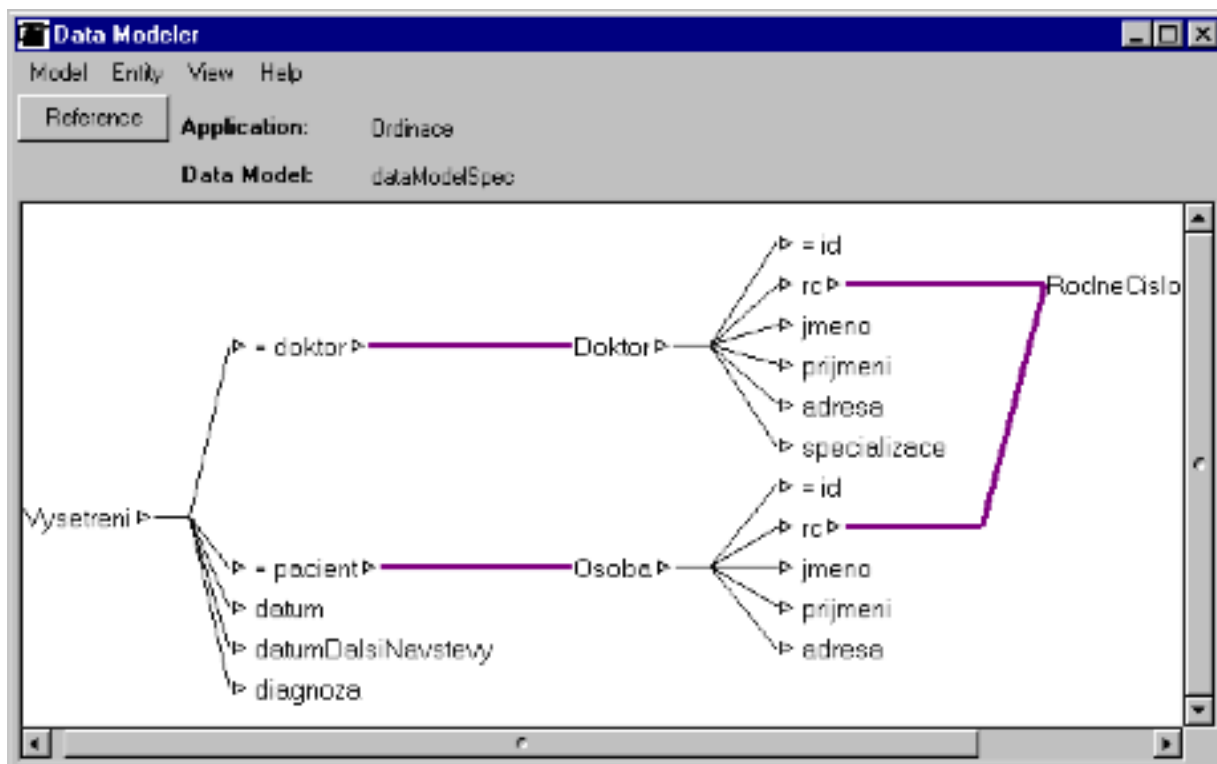
### 5.1.3 Vytvoření vazeb skládání mezi objekty na základě klíčů v tabulkách

Podíváme-li se například na vazbu mezi vyšetřením a doktorem, jak ukazuje ER model, tak zjistíme, že záznamy v tabulce `Vysetreni` mají sekundární klíč (doména „doktor“), který odkazuje na primární klíč (doména „id“) záznamů v tabulce `Doktor`. Tuto vazbu mezi dvěma klíči ObjectLens dovoluje na straně klienta **nahradit přímým skládáním objektů**.

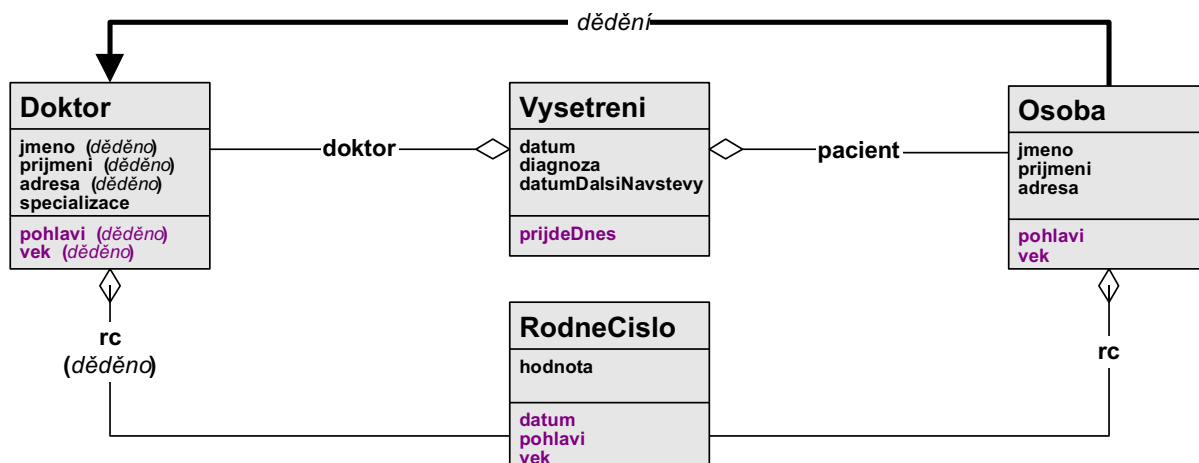
Na straně klienta tedy namísto sekundárního klíče bude každý objekt-instancí třídy `Vysetreni` mít složku, které bude rovnou objektem-instancí třídy `Doktor`. Pro vytvoření vazeb skládání mezi objekty slouží nástroj zvaný Mapping Tool, který je součástí ObjectLens. V tomto nástroji je na levé straně zobrazena strana klienta s příslušnou třídou objektů a na pravé straně je zobrazena strana serveru s odpovídající relační tabulkou záznamů. Všimněte si, že na straně klienta složka `doktor` není typu `Integer`, jak vyplývá z relační tabulky, ale je přímo typu `Doktor`.



Výše naznačeným způsobem se mohou všechny relace **N:1** či **1:1** mezi tabulkami na straně klienta nahradit přímým skládáním objektů do sebe. Jinými slovy to znamená, že na straně serveru bude možné používat při tvorbě dotazů **navigaci sledující vazby skládání** mezi objekty shodným způsobem, jako v objektových databázích. Celou strukturu vzájemně skládaných objektů zobrazuje následující obrázek (Obrázek byl pořízen v programu DataModeller, který je také součástí ObjectLens):



Výsledkem transformace po provedení předchozích kroků je struktura objektů, která připomíná síťový datový model a kterou lze objektovými prostředky grafického jazyka UML zobrazit následovně:

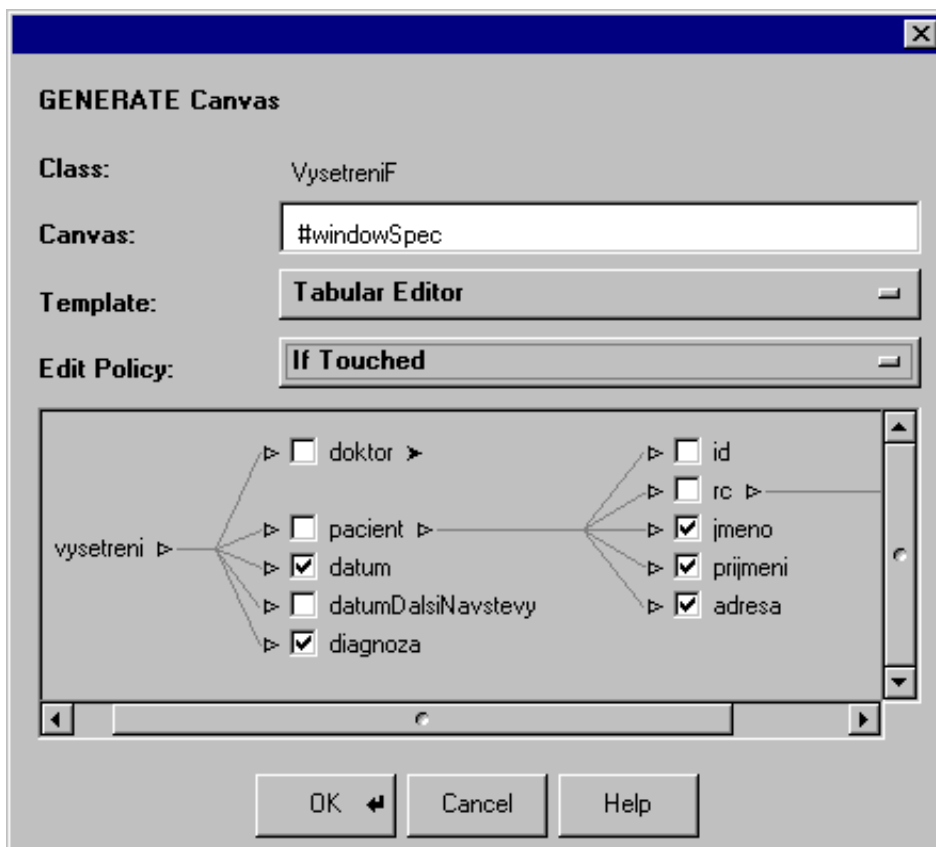


#### 5.1.4 Návrh pohledů nad objektovou strukturou databáze a konstrukce rozhraní

Vytvořená databáze je na straně klienta tvořena především strukturou vzájemně se skládajících objektů různých tříd. Tato struktura, která je zobrazitelná orientovaným grafem (v nástrojích *ObjectLens* jako např. *DataModeler* vypadá jako diagram rozvíjející se zleva doprava), ve kterém uzly grafu představují jednotlivé objekty a hrany grafu představují jejich skládání.

Uživatelské pohledy nad databází, která je popsána uvedenou „sítí“ skládaných objektů, mohou být také s pomocí této sítě snáze konstruovány. Vychází-li totiž pohled či dotaz od nějakého objektu nebo skupiny objektů, může zahrnovat **pouze ty objekty, které jsou skládány** (tj. odkazovány) z objektů, odkud pohled či dotaz vychází. Pro každý výchozí objekt při konstrukci dotazu či pohledu lze potom sestavit podgraf, který představuje část celého schématu a který vymezuje okruh objektů, které mohou

být součástí budovaného dotazu či pohledu. Na obrázku je tvorba pohledu naší databáze vycházející z množiny objektů třídy Vysetreni:



Výsledný pohled je zobrazitelný například v podobě tabulky, což mimo jiné znamená, že **na uživatelské úrovni se objektová databáze může také prezentovat v podobě tabulek:**



## 5.2 Zhodnocení vlastností objektově relačního přístupu ObjectLens

ObjectLens představuje originální přístup k překlenutí rozdílu mezi objektovými a relačními databázemi. Dovoluje totiž za cenu některých omezení **přetvářet relační datový model na straně serveru na objektový datový model na straně klienta**. Největší přínosy a výhody tohoto řešení jsou následující:

1. Relativně jednoduchým a především praktickým způsobem **zpřístupňuje problematiku objektových databází**. Podle našich zkušeností grafické provedení nástrojů ObjectLens také objasňuje problematiku samotného relačního modelování.
2. Praktická znalost technik použitých v ObjectLens usnadňuje a především realizuje **plynulý přechod od relačních k „nerelačním“ objektovým databázím**.

3. Dovoluje budovat **klient-server** software podle **moderních zásad OOP** opírající se o **osvědčené, výkonné a kvalitní relační databázové servery** - v první řadě o systém Oracle. Aplikace pod ObjectLens tedy využívají všech výhod velkých RDBMS, jako například kvalitní transakční zpracování, bezpečnost systému apod.

I když je technologie použitá v ObjectLens zdařilá, tak uvedených vlastností bylo možno dosáhnout pouze za cenu některých zjednodušení a nebo kompromisů na původním objektovém datovém modelu:

1. I když na straně klienta může být použita jakákoliv objektová vazba (např. dědění, závislost, ...), tak **strana serveru umožňuje uchovávat pouze vazby skládání** (pomocí vazeb mezi klíči záznamů provázaných tabulek). Všechny ostatní vazby mezi objekty se nacházejí pouze v paměti klienta.
2. I když je možné na straně klienta pracovat s jakýmkoliv množinami objektů, tak na server jsou ukládány **pouze množiny instance jedné třídy**, které odpovídají příslušným relačním tabulkám vytvořených nad třídami. Všechny ostatní typy objektových množin se nacházejí pouze v paměti klienta.
3. **Metody jsou přítomny pouze v paměti klienta** a nejsou tedy součástí databáze na serveru.
4. databázové objekty musejí mít vyhrazenou minimálně jednu svoji datovou složku se skalárním a atomickým obsahem, která na straně serveru slouží jako **primární klíč**, což v případě použití případě čisté objektové databáze není potřeba.

## 6 Distribuované objekty

Pokud vytvoříme systém, který bude předávat zprávy správným objektům, bez ohledu na to, kde se objekt nachází, získáme *distribuované objekty*. Systém by měl objektům předstírat, že komunikují s lokálním objektem, bez ohledu na umístění adresáta. Tím získáme možnost, že žádný z objektů nemusí být pro komunikaci speciálně upraven. To získáme pomocí tzv. *proxy objektu*.

Proxy (zástupný) objekt, který se tváří jako by byl skutečný objekt, dokáže pouze to, že vezme zprávu kterou dostane, zakóduje ji, doplní cílovou adresou, a po síti ji odešle cílovému objektu. Pomocí proxy objektů je zajištěna transparentnost služeb distribuovaných objektů.

Program, který se stará o zajištění transparentnosti předávání zpráv se nazývá *Object Request Broker (ORB)*.

### 6.1 Object Request Broker (ORB)

Na straně klienta i serveru je potřeba programový modul, který zajišťuje základní komunikační služby. ORB zajišťuje:

- navázání spojení
- odesílání a příjem zpráv (přes síťové služby)
- kódování odesílaných zpráv
- dekódování přijatých zpráv
- předávání zpráv příslušným objektům

Jako příklad si můžeme uvést postupné kroky, jak probíhá předávání zprávy:

- Objekt O1 posílá standardním způsobem zprávu objektu s nímž komunikuje (O2). Objekt O1 nepozná, že komunikuje s proxy objektem.
- Proxy objekt doplní informaci o skutečném objektu a celek předá ORBu.
- ORB zprávu i informaci o umístění cílového objektu zakóduje a pošle jí přes síť ORBu v cílovém prostoru.
- ORB v cílovém prostoru zprávu dekóduje a předá ji objektu O2 v cílovém prostoru.

Pokud by zpráva vracela nějaké výsledky, probíhalo by jejich předání přesně opačně.

ORB zajišťuje některé další služby jako např. obnovení spojení při výpadku, kódování a dekódování dat, řešení chybových stavů atd.

### 6.2 Existující standardy distribuovaných objektů

DO a PDO – Distributed Objects a Portable Distributed Objects. Implementace distribuovaných objektů od firmy NeXT (dnes koupeno firmou Apple). Součástí standardu OpenStep a operačního systému Mac OS X. Založeno na programovacím jazyku Objective C.

- Java-RMI – Remote Method Invocation. Volání vzdálených metod od firmy Sun.
- CORBA – Standard distribuovaných objektů od skupiny Open Management Group (OMG).

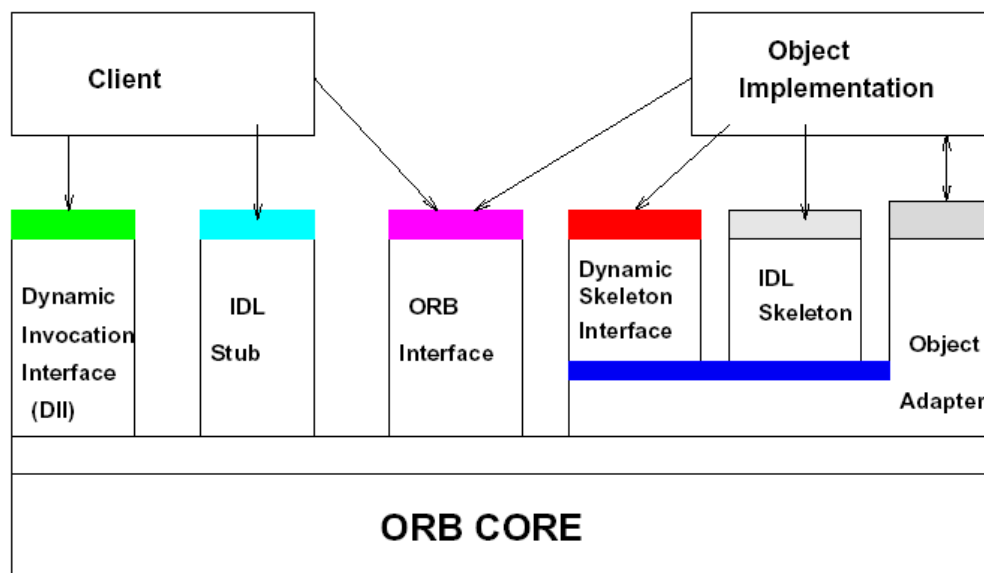
- DCOM – Distributed Common Object Model od Microsoftu. Rozšíření COM/OLE pro distribuované systémy.
- SOAP – Simple Object Access Protocol. Standard pro webové služby založený na XML.
- Opentalk – jednoduché rozšíření Smalltalku o distribuované objekty.

## 6.3 CORBA

*Standard CORBA (Common Object Request Broker Architecture)* je standardem skupiny OMG (Object management Group) pro distribuované objekty. Standard CORBA obsahuje detailní popis návrhu struktury Object Request Brokeru (ORB) umožňujícím komunikaci mezi objekty jako by se nacházely ve stejném adresovém prostoru. Současným standardem je verze 3.0.

### 6.3.1 Základní struktura

Objekty na straně serveru komunikují (posílají zprávy) pomocí stubů nebo DII (Dynamic Invocation Interface viz dále.) a na straně serveru jsou zprávy objektům předávány pomocí skeletonů nebo DSI (Dynamic Skeleton Interface).



### 6.3.2 Interface Definition Language

Distribuované prostředí předpokládá použití různých architektur a programovacích jazyků, mezi kterými nelze jednoduše přenášet definice rozhraní objektů. Proto CORBA definuje jednotný jazyk pro popis rozhraní objektů. Nazývá se *Interface Definition Language (IDL)*.

Protože jazyk IDL je pouze nástrojem k jednotné definici rozhraní objektů, nelze ho použít pro vlastní implementaci nadefinovaných objektů. Deklarace je nutno přeložit do konkrétního programovacího jazyka. Corba standardizuje mapování (tj. překlad) z IDL do C, C++, SmallTalku, Cobolu, Ady, Javy, Pythonu atd.

Syntaxe jazyka IDL je odvozena ze syntaxe C++. Příklad definice rozhraní je na následujícím výpisu:

```
module BANK {
    interface BankAccount {
```

```

// proměnné
enum account_kind {checking, saving};

// výjimky
exception account_not_avaible {string reason;};
exception incorrect_pin {};

// atributy
readonly attribute float balance;
attribute account_kind what_kind_of_account;

// metody
void access (in string account, in string pin)
raises (account_not_avaible, incorrect_pin);
void deposit (in float f, out float new_balance)
raises (account_not_avaible);
void withdraw (in float f, out float new_balance)
raises (account_not_avaible);
}; // konec BankAccount
}; // konec BANK

```

### 6.3.3 Rozhraní CORBA

Specifikace CORBA definuje, že se architektura rozhraní se skládá těchto částí (viz obr):

- Client-side interface – tato část architektury CORBA se skládá z těchto rozhraní mezi ORBem a klientskými objekty:
  - IDL stuby – jsou generovány přímo z IDL a starají se o zakódování zprávy do tvaru vhodném k přenosu (tzv. marshalling) a předává jí jádru ORBu. Je to takzvané Static Invocation Interface (SII).
  - Dynamic Invokation Interface (DII) – používání stubů vyžaduje znalost všech rozhraní volaných objektů již v době překladač programu. DII umožňuje poslat jakoukoli zprávu serverovému objektu bez přítomnosti stubu (a tedy bez znalosti definice rozhraní objektu v IDL).
  - ORB interface - rozhraní ORBu umožňuje, aby funkce ORBu byly přístupné z kódu klienta. Těchto operací je jenom pár. Jsou toho typu, jako např. převedení reference objektu na řetězec atd. ORB interface je jako jediné přístupné klientu i~serveru.
- Implementation side interface – toto rozhraní se skládá z těchto částí:
  - IDL Skeletony – Static Skeleton Interface (SSI) je serverovým ekvivalentem stubů. Provádí tzv. unmarshalling.
  - Dynamic Skeleton Interface (DSI) – Dynamic Skeleton Interface je obdobou DII na straně serveru. Použití DSI umožňuje zpracovat libovolnou příchozí zprávu a tím nahradit skeleton.
  - Object Adapter – provádí správu objektů, spravuje reference na objekty, předává zprávy. Pomocí Object Adapteru přistupují objekty k většině funkcí jádra ORBu.
  - ORB Interface – stejné jako u klienta.
- ORB Core – jádro ORBu provádí základní reprezentaci objektu a komunikační zprávy. Posílá zprávu od klienta na odpovídající adaptér cílového objektu.

### 6.3.4 Object Services

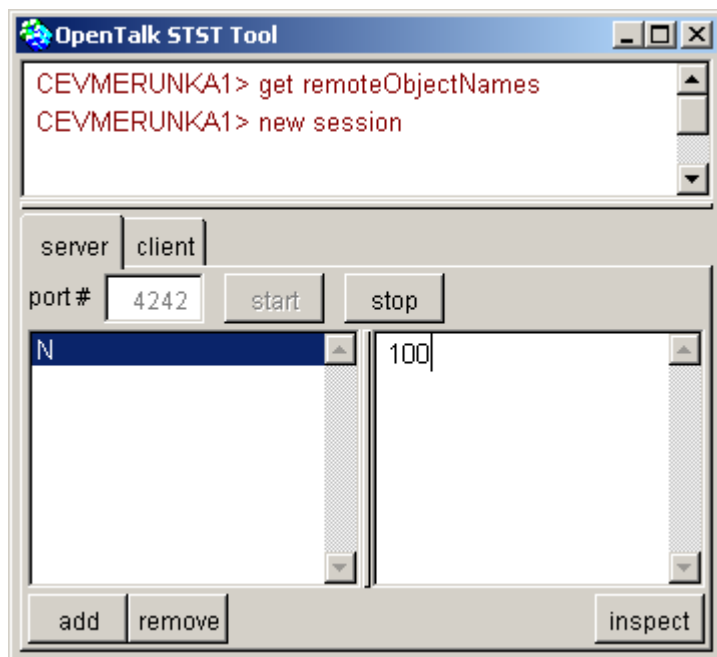
Objektové služby jsou nepovinné součástí standardu CORBA, které zajišťují rozšiřující služby distribuovaným objektům pomocí standardizovaných rozhraní. Jsou to například:

- **Interface Repository** – slouží k získávání informací o rozhraní objektu v době běhu programu. S tímto objektem lze pak pracovat, jako by byl definován pomocí IDL.
- **Object Naming Service** – zajišťuje pojmenovávání objektů (tzv. name binding).
- **Object Event Notification Service** – stará se o předávání událostí objektům.
- **Object Lifecycle Service** – příkazy pro vytváření, rušení, kopírování a přesouvání objektů.
- **Persistent Object Service** – zajišťuje perzistenci objektů.

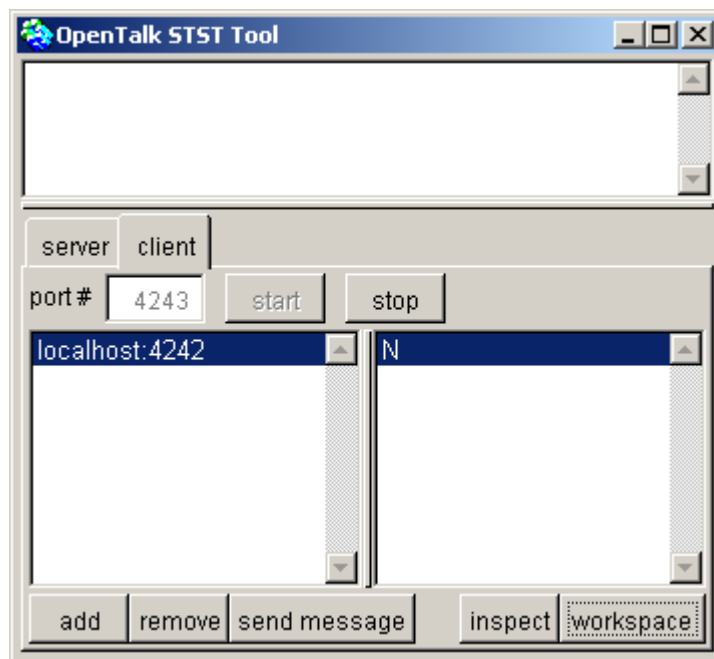
### 6.4 Použití distribuovaných objektů ve Smalltalku

VisualWorks 7.2 podporuje celou řadu služeb pro distribuované objekty. CORBA je implementována v parcelách DST – Distributed Smalltalk. Kromě toho je k dispozici i jednodušší verze object request brokeru, která je vhodná ke vzájemnému propojení dvou počítačů mezi sebou. Tato služba se jmenuje OpenTalk a můžeme si vybrat, zda bude používat pro komunikaci IP-sokety, SOAP, XML, RMI, CGI nebo HTTP protocol. Detaily jsou popsány v samostatném manuálu. (VisualWorks má také komponenty pro generování Webových interfejsů k aplikacím – VisualWave, a dále podporu SMTP, POP3, IMAP, HTTP, HTTPS, FTP protokolů)

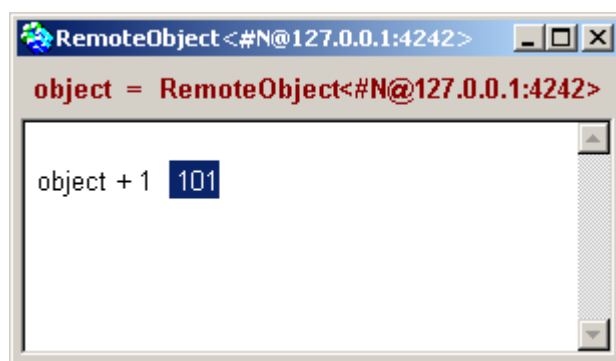
Pro účely výuky a pochopení základních principů práce s distribuovanými objekty je k dispozici nástroj OpenTalk ST-ST Tool, který umožňuje jednoduché propojení dvou počítačů. Server je počítač, který obsahuje databázi distribuovaných objektů. Na ukázce je server, který má v databázi object pojmenovaný N s hodnotou 100:



Na jiném počítači potom lze spustit tzv. klient, což je počítač, který dokáže s objekty serveru pracovat:



Na počítači klienta je potom možné otvoriť Workspace, ktorý pracuje se vzdáleným objektom na serveru:



## 7 Jazyk UML

### 7.1 Úvod

V první polovině 90. let vzniklo mnoho objektových metodologií (OMT, Objectory, OOSD a dále desítky dalších) a každá používala svou vlastní notaci – každá metodologie používala jiný zápis pro vyjádření stejné věci (např. třída se jednou kreslila jako obdélník, v jiné metodologii jako obláček atd.). To byl velký handicap vůči klasickým strukturovaným metodologiím (např. SSADM, Jourdonova Moderní strukturovaná analýza), které používaly stejné nástroje (tj. zejména ER a Data-Flow diagramy) a tím umožnily dobrou srozumitelnost výsledků analýzy. Čím dál víc se ukazovala nutnost společného standardu pro diagramy používané při objektovém vývoji software. Nejpropracovanější byly standardy OML (Object Modelling Language) a UML (Unified Modelling Language). Jazyk UML, který prosazovali nejvýznamnější IT firmy (např. Rational, IBM) sdružené v organizaci OMG (Object Management Group), se stal standardem pro objektový návrh.

### 7.2 Vznik UML

Nejvíce se o vznik jazyka UML zasloužili pánové Grady Booch, James Rumbaugh a Ivar Jacobson (jsou známi pod jménem The Three Amigos). Od začátku 90. let vyvíjeli svoje objektové metodologie OMT (Object Modelling Technique - Rumbaugh), OOD (Object Oriented Design - Booch) Objectory, někdy také nazývanou OOSE (Object Oriented Software Engineering - Jacobson). V roce 1994 se pánové Booch a Rumbaugh (jejich metodologie byli v té době nejrozšířenější) sešli ve společnosti Rational, která si dala za úkol vytvořit unifikovaný objektový modelovací jazyk, a o rok později se k nim přidal Jacobson<sup>16</sup> (jeho metodologie Objectory silná v metodologii). V roce 1996 se ujal návrhu standardu OMG a v roce 1997 byla zveřejněna první specifikace jazyka UML. V současné době je aktuální specifikace verze 2.0.

### 7.3 Diagramy UML

Základem modelování v UML je model, ten lze definovat jako souhrn informací o systému. Modely mohou být několika typů – např. analytické a implementační. Analytický model odpovídá hlavně na otázku **co** by měl systém dělat a implementační **jak**. Diagram je graficky znázorněným pohledem na model. Diagram popisuje část modelu (málokdy celý) a nejčastěji jenom jeden jeho aspekt (například statickou strukturu, dynamickou strukturu). UML definuje těchto 9 diagramů:

1. **Diagram tříd** zachycuje třídy v systému a vztahy mezi nimi.
2. **Diagram komponent** zachycuje komponenty, jejich rozhraní a vztahy mezi komponentami.
3. **Diagram nasazení** znázorňuje infrastrukturu počítačů a přiřazení jednotlivých softwarových komponent na jednotlivé počítače.
4. **Objektový diagram** zachycuje vztahy mezi jednotlivými objekty.
5. **Diagram případů užití** znázorňuje jednotlivé případy užití a jejich vztahy mezi sebou a vztahy k okolí systému.
6. **Sekvenční diagram** zachycuje iterace mezi jednotlivými prvky systému s hlavním hlediskem na čas a následnosti.
7. **Diagram spolupráce** vyjadřuje iterace mezi jednotlivými prvky systému ale s hlediska na spolupráci jednotlivých prvků (sekvenční a diagram spolupráce jsou vzájemně převoditelné)
8. **Stavový diagram** zachycuje stavy a přechody prvků systému.

---

<sup>16</sup> O jazyku UML by se dalo zjednodušeně říci, že přejímá notaci s OMT plus diagramy případů užití z metodologie Objectory.

9. **Diagram aktivit** zachycuje procesy probíhající v systému pomocí aktivit a přechodů mezi nimi.

Diagramy tříd, komponent a nasazení vyjadřují statickou strukturu modelovaného systému, zatímco pomocí ostatních diagramů (objektového, případu užití, sekvenčního, spolupráce, stavového a aktivit) vyjadřujeme chování objektů (tj. dynamickou stránku systému).

### 7.3.1 Použití diagramů UML

Jednou z nejrozšířenějších chybných představ o jazyku UML je, že je metodikou objektového návrhu systémů. Jazyk UML **není metodikou**, je pouze modelovacím jazykem<sup>17</sup>. Jazyk UML se skládá z množiny diagramů, které lze s výhodou použít v různých fázích vývoje systémů. V standardu jazyku UML je definováno jak diagramy kreslit, ne to jak je vytvořit, nebo kdy jaké diagramy použít. To říkají metodologie objektového návrhu systémů založené na UML – jako například UP (Unified Process), poslední verze OMT (Object Modelling Technique) atd.

Klasické schéma použití jednotlivých diagramů v metodologiích je:

1. Pro zachycení požadavků na systém se použijeme **diagram případů užití**.
2. Následně jednotlivé případy užití modelujeme jako procesy pomocí **diagramů aktivit**.
3. Nalezneme základní třídy a navrhujeme **základní (návrhový) diagram tříd** v systému.
4. Dalším krokem je navržení základních interakcí mezi třídami. Použijeme **sekvenční diagram**, případně **diagram spolupráce**.
5. Pokud má třída složité stavové chování, k zachycení jejích stavů a přechodů se použijeme **stavový diagram**.
6. V iteracích zpřesňujeme návrhový diagram tříd. (tj. návrat k bodu 3).
7. Navrhujeme rozdělení tříd do komponent a navrhujeme jejich rozhraní a zaznamenáme toto rozdělení do **komponentového diagramu**.
8. Navrhujeme **softwarový diagram tříd** – upravíme návrhový diagram na příslušné podmínky (např. s přihlédnutím na použitý programovací jazyk, existující aplikace, použitou databázi, rozdělení do komponent atd).
9. A na závěr navrhujeme rozmístění jednotlivých komponent na počítačovou infrastrukturu (tj. na jakém počítači bude konkrétní komponenta umístěna). Použijeme **diagram nasazení**.

## 7.4 Vybrané diagramy UML

Dále bude uveden výběr několika diagramů UML. Popis diagramů bude uveden na příkladech ukazujících základní možnosti. Další možnosti UML lze nalézt v rozsáhlé literatuře.

### 7.4.1 Diagram tříd

Diagram tříd složí k zobrazení statických vztahů mezi třídami. Základním prvkem je třída. Třída je znázorněna obdélníkem rozděleným na tři pole – ty obsahují jméno třídy, atributy třídy (zjednodušeně proměnné obsažené ve třídě) a metody třídy. Pokud je políčko prázdné, lze ho vynechat.

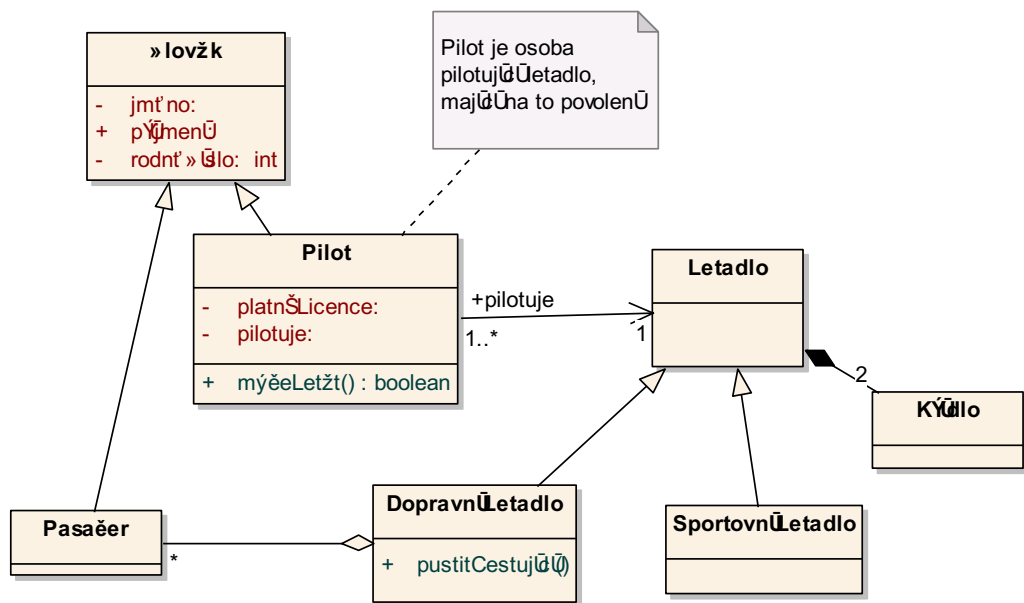
Třídy mají mezi sebou následující vztahy:

---

<sup>17</sup> Kdyby se použila analogie mezi UML a lidským jazykem, tak UML by definovalo pouze jak psát a vyslovovat jednotlivá písmena (maximálně jak je spojovat do slov), zatímco metodologie by určovala jak tvořit nová slova a jak je spojovat do vět aby dávaly smysl.

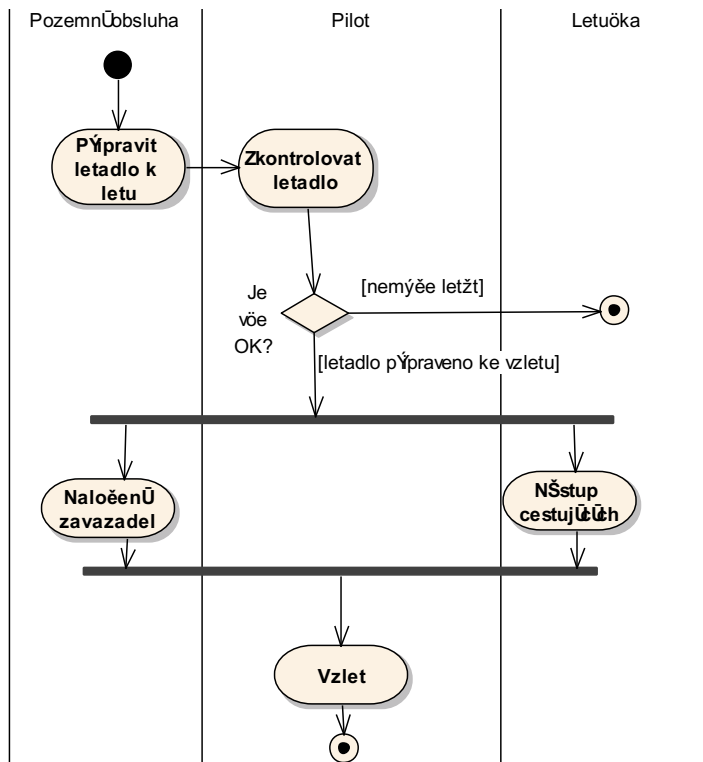
- **Generalizaci** – vztah nadtyp-podtyp, značí se tlustou čarou s šipkou, směřující k obecnější třídě (na našem příkladě vztah Člověk-Pilot).
- **Asociaci** – vztah mezi třídami. Asociace mohou být typů:
  - **Prostá** – obecný vztah mezi objekty (na našem příkladě Pilot-Letadlo), asociace může být pojmenovaná, může mít pojmenovanou roli konce asociace, může mít vyznačenou kardinalitu vztahu (na našem příkladě Letadlo je pilotováno 1 a více piloty a z druhé strany – pilot pilotuje 1 letadlo) a směr. Značí se tenkou čarou mezi třídami.
  - **Kompozice** – znamená vztah celek-část (v našem případě je to vztah mezi letadlem a křídlem, křídlo je částí letadla). Vyjadřuje existenční závislost (Letadlo by bez křídla nemohlo existovat). Značí se jako asociace, ale u celku je vybarvený malý kosočtverec.
  - **Agregace** – vyjadřuje slabší vztah než kompozice (na našem příkladě je mezi DopravnímLetadlem a Pasažerem). Typický rozdíl mezi kompozicí a agregací je, že v případě agregace může být třída členem více agregací. Značí se podobně jako kompozice, ale kosočtverec není vybarven.
- **Závislost** – vztah klient, server – značí se tenkou přerušovanou čarou se směrem (vyjádřeným šipkou) od závislé třídy (serveru) ke klientské třídě.

V diagramech UML (ve všech) lze použít symbol poznámky (v našem případě u Pilot), když chceme prvek lépe vysvětlit.



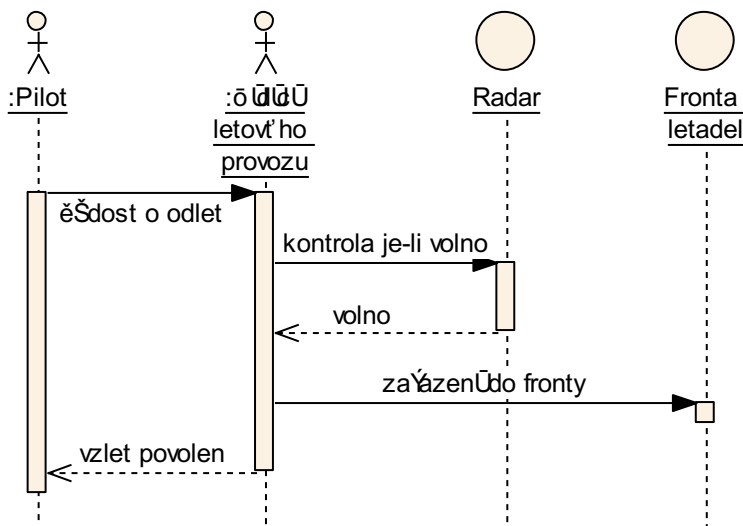
## 7.4.2 Diagram aktivit

Diagram aktivit je jediným neobjektovým diagramem v UML. Vyjadřuje jednotlivé aktivity v procesu a jejich návaznost (tj. vyjadřuje proces). Má podobné vyjadřovací možnosti jako vývojový diagram. V diagramu aktivit lze zachytit rozhodování i paralelní průběh aktivit. Diagram aktivit lze rozdělit pomocí tzv. swimlines (plovací čáry) na části, které jsou v kompetenci jednotlivých objektů. V našem případě se aktivity „Připravit letadlo k letu“ a „Naložit zavazadla“ týkají Pozemní obsluhy. Toto rozdělení není typické, protože některé aktivity se týkají více objektů a někdy by rozdělení diagramu vedlo k jeho znepráhlednění.



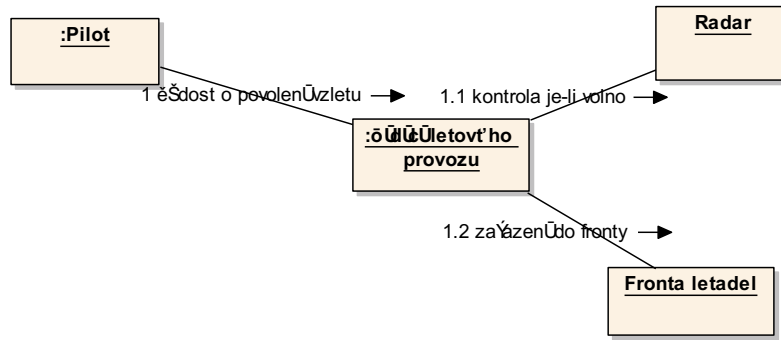
### 7.4.3 Sekvenční diagram

Tento diagram zachycuje iteraci mezi jednotlivými objekty systému. Znárodnuje jednotlivé zprávy (zachycené pomocí čáry s šipkou) a návraty ze zpráv (zachycené přerušovanou čarou).



### 7.4.4 Diagram spolupráce

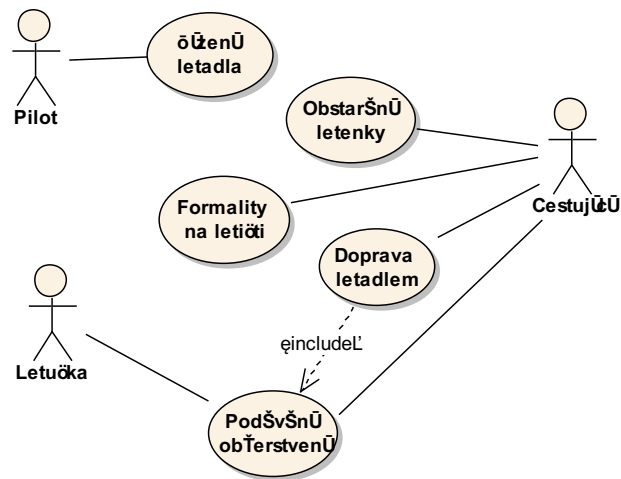
Diagram spolupráce a sekvenční diagram zachycuje stejnou věc (tj. iteraci mezi objekty), ale kladou důraz na něco jiného, sekvenční na časovou stránku věci a spolupráce na vztahy (spolupráci) mezi objekty. Pokud chceme v tomto diagramu nastínit posloupnost zpráv, musíme je očíslovat. Následující příklad ukazuje žádost pilota o povolení vzletu (tj. je to překreslený předcházející sekvenční diagram).



### 7.4.5 Diagram případů užití

Diagram případů užití (Use-Case Diagram) zachycuje jednotlivé požadavky na systém, tak jak se bude jevit zvenčí. Je to typicky úvodní diagram analýzy. Jsou na něm zachyceny

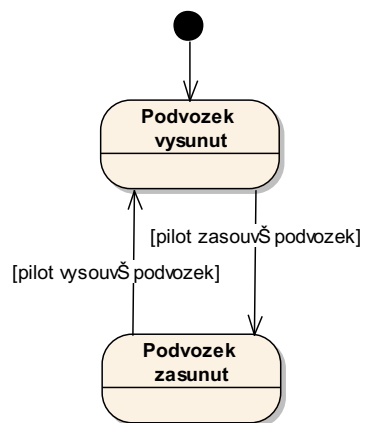
- jednotlivé případy užití – tj. procesy, které jsou přístupné zvenčí systému (nezachycuje vnitřní procesy),
- vztahy mezi případy užití – mezi případy užití může být např. vztah generalizace nebo třeba vztah include (zahrnuje), kdy jeden případ užití je součástí druhého. Zde je vidět použití tzv. stereotypu<sup>18</sup>,
- aktori – jsou to entity zasahující zvenjšku do systému, znak aktora je panáček – typicky se systémem pracuje člověk.
- vztahy mezi aktory a případy užití – aktori operují s jednotlivými případy užití.



### 7.4.6 Stavový diagram

Stavový diagram zachycuje stavy objektu a přechody mezi nimi. Má smysl jen pro objekty, které prochází v době života různými stavy. Pokud bychom např. měli ještě objekt Podvozek, jeho stavový diagram by vypadal takto:

<sup>18</sup> Stereotypy nám slouží k modifikaci jednotlivých prvků UML. Na našem příkladě modifikujeme vztah závislosti pomocí stereotypu <<include>> na vztah zahrnuje. Stereotypy se píší do dvojitéch ostrých závorek a jsou buď standardní (jako třeba include zde) nebo uživatelsky definované. Pomocí uživatelsky definovaných stereotypů si lze rozšířit a upravit UML podle vlastních potřeb.



Text v hranatých závorkách na přechodu obsahuje podmínku, která musí být pro uskutečnění přechodu splněna. Pokud je při přechodu vykonávána nějaká akce, píše se za lomítko. Akce prováděné ve stavech se píše do spodní poloviny oválku stavu.

## 8 Slovníček pojmů

<i>pojem</i>	<i>anglický překlad</i>	<i>vysvětlení</i>
<b>agregace objektů</b>	<b>object aggregation</b>	Zvláštní případ →skládání objektů, kdy je existence a →identita objektu závislá na jiném objektu, který je v něm skládán. (Změna skládaného objektu by v systému způsobila i změnu skládajícího objektu.)
<b>aktivita</b>	<b>activity</b>	Aktivity představují jednotlivé části chování →business objektů tak, jak byly rozpoznány technikou →OBA. V →procesních diagramech se pomocí vybraných aktivit provádějí →přechody mezi →stavy objektů. Aktivity různých objektů mezi sebou komunikují (→komunikace). Z aktivit se odvozují →metody.
<b>aktor</b>	<b>actor</b>	Aktor je specifický název objekt, který je schopen →delegovat zprávy. (→čistý objektový programovací jazyk)
<b>aktorový model výpočtu</b>	<b>actor computational model</b>	Varianta objektově orientovaného chování systému, kde dochází k →delegování →zpráv mezi objekty. (→aktor, čistý objektový programovací jazyk)
<b>analýza</b>	<b>analysis</b>	Souhrnný název pro fáze tvorby systému, při kterých se zjišťuje zadání systému a modeluje se jeho požadovaná funkčnost. Po analýze následuje →návrh. V případě →spirálního způsobu tvorby softwaru se analýza kryje s →expansí systému. V →BORMu se analýza skládá z fáze →strategické analýzy, →úvodní analýzy a →podrobné analýzy.
<b>architektura systému</b>	<b>system architecture</b>	Komplexní model systému, který se skládá z několika vrstev, které jsou samy o sobě modelem zaměřeným na jednu stránku systému. Je to například vrstva →procesů, logický model (popis dat, funkcí a pravidel) a model komponent (např. softwarové aplikace nebo organizační struktury). Prvky z různých vrstev architektury jsou mezi sebou vzájemně provázány, a proto je vhodné při každé změně provést rozbor dopadů na prvky jiných úrovní. (→konvergenční inženýrství, BPR)
<b>AS-IS asociace</b>	<b>AS-IS association</b>	První fáze →BPR. AS-IS znamená „tak, jak to je“ Vztah mezi objekty, který vyjadřuje, že jeden objekt v modelu potřebuje nebo používá nebo k němu jinak přísluší druhý objekt. Tento vztah se nesmí vykládat jako vazba mezi záznamy v tabulkách →relačních databázi a na rozdíl od jiných metod se v BORMu nepoužívá pro →softwarové modelování. Z asociací lze odvodit vazbu →skládání objektů.
<b>asynchronní zpráva, paralelní zpráva</b>	<b>asynchronous message, parallel message, fork message</b>	→Zpráva, na jejíž výsledek →objekt, který zprávu poslal, nečeká. Objekt tedy poslanou zprávou spustil vykonávání nějakých aktivit a současně dál pokračuje ve svých aktivitách. Tento typ zpráv je v →čistých objektových jazycích základem paralelního programování a také umožňuje →delegování a →závislost mezi objekty. (→synchronní zpráva)
<b>atribut, vlastnost</b>	<b>attribute</b>	Vlastnosti →objektu, kterými se objekt prezentuje svému okolí v systému. (Například „barva“, „váha“, „cena“ atd.) V →BORMu by atributy neměly být chápány jen jako data uvnitř objektů, protože atributy lze realizovat i jako →metody, které dané hodnoty poskytují. (→zapouzdření, protokol)
<b>automat</b>	<b>automaton (mn.č. „automata“), finite state machine</b>	Zařízení, které přijímá vstupní informace a jako odpovědi na ně vydává informace výstupní, které závisí jednoznačně na vnitřním stavu tohoto zařízení a na vstupní informaci. Během své činnosti automat vykonává →přechody čímž mění svoje →stavy. Teorie automatů je v →BORMu základem pro modelování chování →objektů v →procesech. (→ORD).
<b>BM</b>	<b>BM</b>	Zkratka pro →business modelování.
<b>BO</b>	<b>BO</b>	Zkratka pro →business objekt.
<b>BORM</b>	<b>BORM</b>	Zkratka pro „Business Object Relation Modeling“. BORM je metodika vyvinutá v rámci výzkumného projektu VAPPIENS Britské rady (British Council) ve spolupráci s Deloitte&Touche. Základní filosofií BORMu je plná podpora →objektového přístupu, využívání →procesního modelování pro →získávání požadavků a myšlenky →konvergenčního inženýrství.

<b>BPR – reengineering business procesů</b>	<b>BPR – Business Process Reengineering</b>	Přehodnocení a rekonstrukce →procesů za účelem jejich zdokonalení. Tato činnost je někdy důležitá pro nalezení správného zadání informačního systému. (→ <i>získávání zadání</i> ) Provádí se ve dvou fázích: V první etapě →AS-IS se podrobně modeluje stávající stav procesů. Po vyhodnocení výsledků první etapy se přistupuje k druhé etapě →TO-BE. Zde se navrhují nové procesy a provádí se návrh nové organizační struktury. Pro podklady na BPR se používá technika →OBA.
<b>business inženýrství</b>	<b>business engineering</b>	Inženýrský obor, který se zabývá analýzou, modelováním a návrhem organizačních a řídicích struktur. (→ <i>BPR</i> ) Některými autory je považován za součást →informačního inženýrství. Podle BORMu jsou tyto aktivity potřebné pro získání správných požadavků na informační systém. (→ <i>získávání požadavků</i> )
<b>business modelování</b>	<b>business modeling</b>	První etapa BORMu, kdy se vytváří model →procesů systému tak, aby došlo k rozpoznání zadání problému a případnému přepracování stávajících procesů a nebo organizační struktury za účelem lepší implementace softwaru. (→ <i>BPR</i> ) Zahrnuje fáze →strategické analýzy a →úvodní analýzy.
<b>business objekt</b>	<b>business object</b>	Objekt reálného světa, který slouží k analýze a popisu zadání systému. (→ <i>participant</i> )
<b>business proces</b>	<b>business process</b>	Business procesy jsou východiskem pro funkční popis podniku nebo organizace. (→ <i>proces, BPR</i> )
<b>C# („cis“)</b>	<b>C# („c-sharp“)</b>	Jazyk z dílny Microsoftu. Velmi podobný jazyku →Java. Jeho první verze byla prakticky dostupná až v letošním roce.
<b>C++ („cé plus plus“)</b>	<b>C++</b>	→Smíšený programovací jazyk. Patří mezi složité programovací jazyky, ale je v něm možné dosáhnout efektivního kódu. Jeho první verze pocházejí z 80. let.
<b>CASE – nástroje pro podporu softwarového inženýrství</b>	<b>CASE – Computer Aided Software Engineering</b>	Programy, které pomáhají analytikům a vývojářům postupovat podle nějaké metodiky. Správný CASE by měl mít podporu týmového vývoje, nástroje pro práci s →diagramy, možnost tvorby reportů a analýz nad projektovými daty a generovat kód. CASE nemusejí být pouze pro metody →softwarového inženýrství, ale i pro →business inženýrství. Pokud podporuje →metamodelování (např. Proforma Workbench, Metaedit), tak lze CASE uživatelsky „nastavovat“ a „vylepšovat“ jím podporovanou metodiku. (→ <i>metamodelování</i> )
<b>cíl</b>	<b>goal</b>	V kontextu →BPR to je součást detailního popisu →aktivity. Cíle vymezují, kam má podnik směřovat. (Například získat dominantní postavení na trhu nebo ve vybraném tržním segmentu.) Jejich znalost je důležitá pro rozhodování nad souvislostmi zadání informačního systému, →podnikových procesů s dalšími atributy organizace. (→ <i>získávání zadání</i> )
<b>CM</b>	<b>CM</b>	Zkratka pro →konceptuální modelování.
<b>CO</b>	<b>CO</b>	Zkratka pro →konceptuální objekt.
<b>CORBA</b>	<b>CORBA</b>	Standard pro práci s objekty v distribuovaných softwarových systémech. Je doporučován sdružením →OMG a je podporován v →objektových databázích. (→ <i>objektově relační databáze, Gemstone</i> )
<b>CSF</b>	<b>CSF</b>	Zkratka pro „Critical Success Factor“ (→ <i>faktor</i> ).
<b>činnost, pracovní činnost</b>	<b>real activity</b>	V kontextu →BPR to je součást detailního popisu →aktivity. Je to konkrétní popis týkající se plnění jednoho pracovního úkolu. (Například „zajištění pracoviště“ nebo „oprava vozidla“). Jedna aktivita typicky obsahuje více činností. (→ <i>BPR</i> )
<b>čistý objektový programovací jazyk</b>	<b>pure object-oriented programming language</b>	Programovací jazyk, který důsledně podporuje vlastnosti objektového přístupu. Takový jazyk není úpravou nějakého neobjektového jazyka. Ve srovnání se →smíšenými jazyky je v nich objektové programování snadnější – například dovolují programovat aplikaci i za jejího chodu. Odhaduje se, že čistý jazyk potřebuje asi 2× až 5× méně příkazů na jeden →funkční bod. Mezi čisté jazyky patří například →Smalltalk.
<b>databáze, SRBD – systém řízení báze dat</b>	<b>database, DBMS – database management system</b>	Software, který dovoluje uchovávat a vydávat data pro potřeby uživatelů a nebo jiných systémů k němu připojených. Databáze je důležitou součástí většiny →informačních systémů. Podle principu činnosti se člení na síťové, →relační, →objektově relační a →objektové.

<b>datový tok</b>	<b>data flow</b>	Data, která si objekty vyměňují při →komunikacích nebo posílání →zpráv. Rozlišují se →parametry zprávy a →návrátové hodnoty.
<b>dědičnost</b>	<b>inheritance</b>	Vazba mezi →softwarovými objekty, pomocí které lze programovat nové objekty (potomky) s využitím již existujících objektů (předků). Nové objekty jsou potom v programu schopny využívat →metody svých předků. Ve →smíšených programovacích jazycích je dědičnost jediná možnost, jak zajistit →polymorfismus mezi objekty. (→ <i>hierarchie typů</i> )
<b>delegování</b>	<b>delegation</b>	Zpracování →zprávy objektem tím způsobem, že objekt, který přijal zprávu, ji nebude zpracovávat a vyvolá přenesení této zprávy na jiný objekt, který ji již dokáže zpracovat. Výpočet se pak dál odehrává jen u delegovaného objektu a objekt, který zprávu od sebe delegoval se ho již neúčastní (ani mu nemusí být předán výsledek). V případě, že implementační prostředí tuto vazbu nepodporuje, musí se ve fázi →softwarového modelování nahrazovat. (→ <i>smíšený programovací jazyk</i> )
<b>deliverable</b>	<b>deliverable</b>	Synonymum pro →výrobek.
<b>design</b>	<b>design</b>	Synonymum pro →návrh.
<b>diagram</b>	<b>diagram</b>	Schéma, které popisuje pomocí značek a dalších dle standardu dohodnutých způsobů kreslení nějaký model. V →BORMu se diagramy označují jako →ORD.
<b>esenciální objekt</b>	<b>essential object</b>	Synonymum pro →business objekt.
<b>expanse</b>	<b>expansion</b>	První část cyklu →spirálního modelu. Zde dochází ke hromadění informací, potřebných pro pochopení zadání a vytvoření aplikace. Expanze končí dokončením →konceptuálního modelu, který na logické úrovni reprezentuje požadované zadání a formálně popisuje řešený problém. (→ <i>konsolidace, analýza</i> )
<b>faktor, kritický faktor pro úspěch</b>	<b>CSF – critical success factor</b>	V kontextu →BPR to je součást detailního popisu →aktivity. Faktor je činitel, který má zásadní vliv na podnikatelský úspěch. (Je to například „existence podnikové informační strategie“, „hodnocení na základě měření“ nebo „dodržování standardů“.) Jejich znalost je důležitá pro rozhodování nad souvislostmi zadání informačního systému, podnikových →procesů s dalšími atributy organizace. (→ <i>získávání zadání</i> )
<b>funkce systému</b>	<b>system function</b>	Požadovaná funkce je nejjednodušším slovním popisem požadovaných →procesů v systému podle →OBA. Pro pozdější bezproblémovou komunikaci analytiků a zadavatelů je vhodné do tohoto seznamu zahrnout a zvláště vyznačit i funkce, které popisují, co se modelovat nebude. Ze seznamu funkcí se odvozuje seznam →scénářů.
<b>funkční body</b>	<b>function points, feature points</b>	→Metrika, která slouží k odhadování pracnosti vyvíjeného softwaru metodou výpočtu funkčních bodů, které se vypočítají z rozsahu požadavků na systém (např. počet datových struktur, uživatelských vstupů a výstupů, ...). Z výsledného počtu funkčních bodů lze odhadnout velikost budoucího programu, protože pro různé →programovací jazyky je na základě statistických měření známo, kolik příkazů je třeba na pokrytí jednoho funkčního bodu. Metoda „feature points“ je variantou této metody pro případ objektové tvorby softwaru.
<b>Gemstone</b>	<b>Gemstone</b>	Představitel čistého →objektového databázového systému. Databázový systém Gemstone používá programovací jazyk →Smalltalk nebo →Java. Podporuje standard →CORBA.
<b>generalizace-specializace</b>	<b>generalization-specialization</b>	Synonymum pro →hierarchie typů.
<b>generický model</b>	<b>generic model</b>	Generický model je takový model, který v sobě kombinuje vlastnosti modelu s vlastnostmi →metamodulu, což může usnadnit návrh systému. (→ <i>metamodelování, znovupoužitelnost, návrhové vzory</i> )
<b>hierarchie typů</b>	<b>type hierarchy</b>	Tato vazba vyjadřuje vztahy mezi →protokoly objektů. To znamená, že objekty nadtypu jsou →polymorfni s objekty podtypu. K upřesnění vazby lze uvést seznam →zpráv, které se polymorfismu týkají. Tato hierarchie objektů vzniká z hierarchie →JE-JAKO a později je z ní odvozována hierarchie →dědičnosti mezi objekty.

<b>identita objektů</b>	<b>object identity</b>	Spolu s →polymorfismem a →zapouzdřením základní vlastnost →objektů, díky které jsou různé objekty v jednom systému nezaměnitelné ani tehdy, mají-li shodná data a nebo →metody. V některých konkrétních softwarových prostředích (→smíšené programovací jazyky) je ale třeba identitu zvlášť zajišťovat. Například v →relačních databázích musí mít objekty pro vzájemné rozlišení navíc zvláštní údaj nazývaný primární klíč (neboli index). V →objektových databázích to není potřeba.
<b>impedanční problém</b>	<b>impedancy problem</b>	Potíže při tvorbě databázových aplikací, kdy je třeba současně používat dva počítačové jazyky. Například →SQL pro manipulaci s daty a →Java pro sestavení aplikace. Čistě →objektové databáze tento problém nemají, protože jejich jazyky dokáží splnit obě funkce. (→Smalltalk)
<b>implementace</b>	<b>implementation</b>	V této fázi se vytváří (programuje, sestavuje či generuje z CASE nástroje), testuje a předává uživateli požadovaný software. Při →spirálním způsobu tvorby softwaru je součástí →konsolidace systému.
<b>informační inženýrství</b>	<b>information engineering</b>	Inženýrský obor, který podle některých autorů zahrnuje →softwarové inženýrství a →business inženýrství. Jedná se o disciplínu, která nahlíží komplexně na organizační a řídicí struktury a →informační systémy. (→sociotechnický systém, konvergenční inženýrství)
<b>informační systém</b>	<b>information system</b>	Systém sloužící k poskytování informací a sestavený podle nějakého zadání složený z hardwarových, softwarových a organizačních prvků.
<b>instance</b>	<b>instance</b>	Objekt, který má svoje vlastnosti (= strukturu dat a metody) popsané v nějaké →třídě. Všechny instance téže třídy tedy mají stejné metody a stejnou strukturu dat a liší se mezi sebou jen konkrétním obsahem těchto dat. Instance se odvozují z →business objektů.
<b>iterativní způsob tvorby softwaru, iterativní model</b>	<b>iterative development lifecycle</b>	Způsob vývoje softwaru, kdy se v případě potřeby navrácí do počátečních fází vývoje. Pro upřesnění zadání se mohou vytvářet →prototypy. Jeho variantou je →spirální model. (→vodopádový model)
<b>Java („džáva“)</b>	<b>Java</b>	→Smíšený programovací jazyk. Jeho první verze pocházejí z 80. let, ale rozšířil se až ve druhé polovině 90. let s nástupem WWW. Je podobný →C++, ale je jednodušší a podporuje více objektových vlastností. Podle některých autorů je považován za →čistý objektový programovací jazyk.
<b>JE-JAKO</b>	<b>IS-A</b>	Vztah hierarchie objektů. Objekty na nižší úrovni této hierarchie jsou prvky domény, která je podmnožinou domény objektů vyšší úrovně. Tento vztah se v →BORMu nesmí vykládat jako →dědičnost mezi →softwarovými objekty, protože je na dědičnost postupně transformován přes →hierarchii typů.
<b>klient</b>	<b>client</b>	Objekt, na kterém jsou →závislé jiné objekty (→server).
<b>klient</b>	<b>client</b>	Softwarová aplikace, která využívá data nebo služby jiné aplikace na jiném počítači v síti. (→server)
<b>kolekce, sada komunikace</b>	<b>collection communication</b>	Synonymum pro →množinu objektů. Řízení →aktivit →business objektů. Komunikace je abstrakce →zpráv mezi objekty. V pozdějších fázích modelování se zprávy z komunikací odvozují.
<b>konceptuální modelování</b>	<b>conceptual modeling</b>	Druhá etapa →BORMu, která zahrnuje tvorbu modelu z →konceptuálních objektů. Takový model stojí napůl cesty mezi zadáním a řešením, kdy je problém považován za rozpoznáný a nalézá se v něm část k softwarovému řešení. V konceptuálních modelech se na systém nahlíží jako na ideální svět počítačových objektů. Mezi modelované vazby patří například →asociace, →skládání, →závislost, →delegování a →hierarchie typů. Zahrnuje fáze →podrobné analýzy a →úvodního návrhu.
<b>konceptuální objekt</b>	<b>conceptual object</b>	Objekt, který se používá v →konceptuálních modelech k popisu první podoby modelu softwaru. Jsou to →instance, →třídy nebo →množiny. Na rozdíl od →softwarových objektů konceptuální objekty nejsou zatíženy detaily konkrétního implementačního prostředí. Konceptuální objekty se odvozují z →business objektů a samy slouží k popisu →softwarových objektů.
<b>konsolidace</b>	<b>consolidation</b>	Druhá část cyklu →spirálního způsobu tvorby softwaru. Model se zde postupně po předchozí myšlenkové "expanzi" stává fungujícím programem. V tomto stadiu může dojít k tomu, že některé z návrhů bude nutno vypustit vzhledem k časovým, kapacitním, implementačním nebo i intelektuálním omezením. (→expanse, návrh, implementace)

<b>kovergenční inženýrství</b>	<b>convergent engineering</b>	Inženýrský obor, který neklade hranici mezi podnikovým systémem a informačním systémem. (→ <i>sociotechnický systém</i> ). Je to směr →informačního inženýrství, který techniky a postupy používané v →softwarovém inženýrství aplikuje do →business inženýrství. Opírá se o →objektový přístup. Je jednou ze základních filosofii →BORMu.
<b>logický objekt metamodel</b>	<b>logical object metamodel</b>	Synonymum pro →konceptuální objekt. Metamodel je model systému, který sám obsahuje prvky k modelování něčeho jiného. Znalost metamodelu metody může pomoci při pochopení metody a také usnadnit návrh konkrétního řešení. (→ <i>metamodelování, generický model</i> )
<b>meta-modelování</b>	<b>metamodeling</b>	Metamodelování znamená tvorbu modelu na vyšší úrovni abstrakce tak, že předmětem modelování je něco, co je samo o sobě modelem něčeho jiného. (→ <i>metamodel</i> ). Dovoluje popsat jednotným způsobem odlišné datové struktury více systémů, což poskytuje možnost skládat je dohromady ve vyšší systém. Zajišťuje standardizaci metodologií a interoperabilitu →CASE nástrojů.
<b>metoda</b>	<b>method</b>	Část programového kódu, kterou objekt spouští, pokud přijme příslušnou →zprávu. Metody mohou například měnit data uvnitř objektů (→ <i>skládání, zapouzdření</i> ), posílat další zprávy dalším objektům nebo provádět →přechody mezi →stavy objektu. Metody se odvozují z →aktivit.
<b>metriky</b>	<b>metrics</b>	Nástroj pro podporu formálního měření kvality softwarových produktů nebo modelů. Aparátem pro softwarové metriky je numerická matematika a statistický aparát. Jejich sběr a vyhodnocování je důležitou součástí řízení projektů →BPR nebo tvorby softwaru. (→ <i>funkční body</i> )
<b>množina objektů, sada, kolekce</b>	<b>collection</b>	Skupina →objektů, které mají dohromady nějaký zvláštní význam pro model, kde se s nimi pracuje nejen po prvcích, ale i jako s celkem. Množiny do systému zavádíme když není vhodné skupinu objektů realizovat jako →třídou. Prvky množiny mohou být objekty různých tříd, pokud jsou mezi sebou →polymorfni. Množiny objektů se odvozují z →business objektů.
<b>modelovací karta, modelová karta</b>	<b>modeling card</b>	Modelovací karta je tabulka, která podrobněji popisuje →objekt. Je součástí techniky →OBA. Může obsahovat →aktivity objektu, s objektem spolupracující objekty a nebo také vlastnosti objektu, které v modelu potřebujeme. (→ <i>proces</i> )
<b>nadtyp-podtyp</b>	<b>supertype-subtype</b>	Synonymum pro →hierarchii typů.
<b>návratová hodnota</b>	<b>return value</b>	Data přenášená jako odpověď na poslání →zprávy mezi objekty. Lze je považovat za →datový tok ve směru opačném k směru poslání zprávy.
<b>návrh, design</b>	<b>design</b>	Souhrnný název pro fáze tvorby systému, při kterých se na základě již provedené →analýzy model přetváří do podoby, kterou je již možné implementovat. (→ <i>implementace</i> ). V případě →spirálního způsobu tvorby softwaru je návrh součástí →konsolidace systému. V →BORMu se návrh skládá z fáze →úvodního návrhu a →podrobného návrhu.
<b>návrhový vzor</b>	<b>design pattern</b>	Návrhové vzory jsou znalosti z předchozích úspěšných projektů. Přestavují mechanismus k uchovávání znalostí o tvorbě modelů, které se týkají →znovupoužitelných řešení. Je to významný prostředek k předávání znalostí mezi tvůrci systému při →softwarovém modelování. Někteří autoři také poukazují na možnost využívání návrhových vzorů při →BPR.
<b>OBA – analýza objektů podle chování</b>	<b>OBA – Object Behavioral Analysis</b>	Technika, která slouží k získávání strukturovaných podkladů ze zadání pro potřeby konstrukce prvotního objektového modelu. Má pět kroků. Výstupy z OBA je například seznam →funkcí systému, →scénářů systému, →modelovacích karet a →procesní diagramy.
<b>objekt</b>	<b>object</b>	Základní prvek modelování objektově orientovaným způsobem. Objekt tvoří jeho data a operace (aktivity nebo metody), které vymezují jeho chování. (→ <i>zapouzdření, protokol, polymorfismus</i> ) Objekty mají v systému →identitu a jejich vlastnosti se mohou v čase měnit. (→ <i>automat</i> ) V →BORMu se nazírání na objekt mění podle fáze projektování. Proto se rozlišují →business objekty, →konceptuální objekty a →softwarové objekty.

<b>objektová databáze</b>	<b>object-oriented database, objectbase</b>	→Databázový systém, který dovoluje uchovávat a také zpracovávat →softwarové objekty přímo v databázi. Takové systémy nepoužívají relační tabulky a pracují na jiném principu. Mezi objektové databáze patří například →Gemstone. (→ <i>objektově-relační databáze, relační databáze</i> )
<b>objektově relační databáze</b>	<b>object-relational database, hybrid object database</b>	→Databázový systém, který uchovává →softwarové objekty v databázi a podporuje jen některé vlastnosti →objektového přístupu. Principem jeho činnosti zůstává relační datový model. Mezi tyto systémy například patří →Oracle od verze 8.0 (→ <i>objektové databáze</i> )
<b>objektový programovací jazyk</b>	<b>object-oriented programming language</b>	Programovací jazyk, který dovoluje využívat ve svých příkazech →objektový přístup. Rozlišují se →smíšené programovací jazyky a →čisté objektové programovací jazyky.
<b>objektový přístup, objektově orientovaný přístup, paradigma</b>	<b>object-oriented paradigm, object-oriented approach, OOP</b>	Souhrn myšlenek o způsobu nazírání na svět, chápání zadání a hledání způsobu jejich řešení. Podle objektového přístupu se systém modeluje jako soustava vzájemně komunikujících →objektů. (→ <i>konvergenční inženýrství, proces</i> ) V užším slova smyslu to je jeden ze způsobů tvorby softwaru. (→ <i>objektový programovací jazyk, objektová databáze</i> )
<b>omezení podle chování</b>	<b>behavioral constraints</b>	Sada pravidel, typicky zobrazovaná ve formě rozhodovacích tabulek, která popisují za jakých podmínek lze konkrétní vazby mezi →business objekty transformovat na vazby mezi →konceptuálními objekty.
<b>OMG</b>	<b>OMG</b>	Object Management Group. ( <a href="http://www.omg.org">http://www.omg.org</a> ) Je to odborné sdružení firem, univerzit a dalších institucí, které podporuje objektovou technologii. Vydává také publikace a standardy (→ <i>OQL, CORBA</i> ).
<b>OOP operace</b>	<b>OOP operation</b>	Zkratka pro Object-Oriented Paradigm. (→ <i>objektový přístup</i> )
<b>OQL</b>	<b>OQL</b>	→Metoda nebo →aktivita objektu.
<b>Oracle</b>	<b>Oracle</b>	Návrh rozšíření jazyka →SQL pro práci s →objekty. Je doporučován sdružením →OMG a je součástí standardu →CORBA.
<b>ORD – diagram vztahů mezi objekty</b>	<b>ORD – Object Relation Diagram</b>	Představitel →relačního databázového systému. Nové verze Oraclu jsou →objektově relační. Používá jazyk →SQL.
<b>parametry zprávy participant</b>	<b>message parameters participant</b>	Diagram, který zobrazuje současně datové, funkční i časové vztahy a souvislosti mezi objekty. OR diagramů je v →BORMu víc druhů podle toho, jaké objekty zobrazují. Například pro fázi →business modelování to je →procesní diagram (nazývaný také procesní mapa) a pro pozdější fáze se používají upravené diagramy UML.
<b>podnikový proces</b>	<b>business process</b>	Data přenášená jako součást →zpráv mezi objekty. Parametry lze považovat za →datový tok ve směru zprávy.
<b>podrobná analýza,</b>	<b>advanced analysis</b>	→Objekt, který se účastní nějakého →procesu a je popsán v nějakém →scénáři a nebo →procesním diagramu.
<b>podrobný návrh, podrobný design</b>	<b>advanced design</b>	Synonymum pro →business proces.
<b>polymorfismus</b>	<b>polymorphism</b>	Vymezení softwarové domény uvnitř modelovaného problému a rozpracování analýzy do detailů jednotlivých druhů a vazeb →konceptuálních objektů. Při →spirálním způsobu tvorby softwaru je součástí →expand systému. (→ <i>analýza</i> )
<b>pozdní vazba</b>	<b>late binding</b>	V této fázi dochází k přeměně prvků již existujícího →konceptuálního modelu do podoby, která je podřízena cílovému implementačnímu prostředí. (→ <i>softwarový model, softwarový objekt</i> ) Při →spirálním způsobu tvorby softwaru je součástí →konsolidace systému. (→ <i>návrh</i> )
		Schopnost objektů reagovat různou →metodou na stejnou →zprávu. Rozlišujeme polymorfismus 1) mezi různými objekty, 2) mezi různými →stavy téhož objektu a 3) v závislosti na objektu, který zprávu vyslal. Polymorfismus významně ulehčuje modelování a způsobuje →znovupoužitelnost objektů. Díky polymorfismu mohou v jednom systému pracovat a zastupovat se objekty s různou strukturou. (→ <i>protokol</i> ) V některých →objektových programovacích jazycích je však implementován jen částečně. (→ <i>dědičnost</i> )
		Druh vazby mezi →zprávou a →metodou, kdy se výběr metody provádí až po poslání zprávy během chodu objektového programu, což je v souladu s filosofií →polymorfismu. Tento typ vazby je na rozdíl od →včasné vazby typický pro →čisté objektové programovací jazyky.

<b>pracovní pozice</b>	<b>job position</b>	V kontextu →BPR to je součást detailního popisu →participantu. Je to konkrétní popis pracovního místa daného pracovníka s případným počtem konkrétních pracovníků na této pozici. Jeden participant typicky obsahuje více pracovních pozic. (→BPR)
<b>problém, issue</b>	<b>issue</b>	Problém je něco, co stojí v cestě při plnění nějakého cíle nebo úkolu a je třeba to vyřešit. Je to například „velké emise do životního prostředí“ nebo „státní regulace“ apod. Jejich znalost je důležitá pro rozhodování nad souvislostmi zadání informačního systému, podnikových →procesů (→business proces) s dalšími atributy organizace. (→získávání zadání, BPR)
<b>proces</b>	<b>process</b>	Sled →aktivit objektů, které dohromady realizují nějakou funkci systému. Mezi procesy může být hierarchie. Procesy popisujeme →scenáři a →procesními diagramy. (→BPR, role objektu v procesu, business proces)
<b>procesní diagram, procesní mapa</b>	<b>process diagram, process map</b>	Tento →diagram představuje mapu všech možných průběhů →procesu pomocí současného zobrazení dvou dimenzí tohoto problému. První dimenzí jsou →role (= průběh aktivit jednoho objektu v procesu) participujících objektů jako →automaty se →stavy a →přechody. Druhou dimenzí je sled →komunikací mezi objekty, který představuje řídicí a datové toky mezi objekty v procesu.
<b>protokol objektu</b>	<b>object protocol</b>	Množina →zpráv nebo →komunikací, které lze objektu poslat. Pomocí protokolu se u objektů popisuje, k čemu v modelu slouží a jaké operace mohou provádět. Má-li více objektů neprázdný průnik protokolů, jsou mezi sebou →polymorfni (→nadtyp-podtyp). Znalost protokolu stačí k tomu, aby se s objektem mohlo v systému pracovat, protože pokud dokáže příslušné zprávy přijímat, tak již nezáleží na jeho vnitřní struktuře. (→zapouzdření, znovupoužitelnost)
<b>prototyp</b>	<b>prototype</b>	Program, který napodobuje funkčnost vytvářeného softwaru, ale byl sestaven s výrazně menšími náklady a nižšími nároky na robustnost. Slouží k upřesnění a ověření zadání. V →BORMu se používají i jako prostředky k demonstraci nově navržených →procesů. (→BPR)
<b>přechod</b>	<b>transition</b>	Jeden úkon →automatu, kterým automat změni svůj stávající →stav na nový stav na základě nějaké přijaté informace. V →BORMu se na →objekty v →procesech nahlíží jako na automaty, kde jejich →přechody jsou realizovány prováděním →aktivit. (→role objektu v procesu)
<b>relační databáze</b>	<b>relational database, table-based database</b>	→Databázový systém, který organizuje data do vzájemně propojených tabulek. Pokud dovoluje přímou práci s objekty a ne pouze s jednoduchými hodnotami např. typu text, číslo nebo datum, tak se hovoří o →objektově relačním systému.
<b>relační vztah</b>	<b>relation</b>	Znázorňuje spojení mezi záznamy v relačních tabulkách →relačních databázích. Pro →objektově databáze se nepoužívá, protože tyto databáze pracují s objekty přímo.
<b>role objektu v procesu</b>	<b>object role in a process</b>	Sled →stavů a →přechodů objektu, který se účastní nějakého procesu. Je to jedna ze dvou dimenzí →procesního diagramu. Na tento sled lze nazírat jako na díleč →diagram, který popisuje →proces ze zorného úhlu diskutovaného objektu, a který lze výhodně použít ke kontrole správnosti a realizovatelnosti celkového modelovaného procesu. (→automat)
<b>sada objektů scénář</b>	<b>collection scenario</b>	Synonymum pro →množinu objektů. Scénář systému je podrobnější popis →procesu v technice →OBA. Odvozují se z →funkcí systému. U scénáře zvlášť popisujeme 1) začátek procesu, 2) vlastní akce procesu, 3) seznam participantů a 4) konec procesu. Mezi scénáři může být hierarchie skládání procesů, hierarchie nadtypů a podtypů a časová návaznost procesů na sebe, pro které se používají stejné značky jako pro objektové diagramy (→UML, ORD). Typické zobrazení scénářů jsou tabulky s uvedenými čtyřmi políčky.
<b>server server</b>	<b>server server</b>	Objekt, který je →závislý na jiném objektu (→klient). Softwarová aplikace, která poskytuje data nebo služby jiným aplikacím na jiných počítačích v síti. (→klient) Může se jednat o →databázový systém.
<b>skládání objektů, kompozice objektů</b>	<b>object composition</b>	Vazba mezi objekty, kdy jeden nebo více objektů představují datové složky jiného objektu. Skládání objektů se v →BORMu odvozuje z →asociací.

<b>SM</b>	<b>SM</b>	Zkratka pro →softwarové modelování.
<b>Smalltalk</b>	<b>Smalltalk</b>	→Čistý objektový programovací jazyk. Jeho první verze byly vyvíjeny již v 70. letech. Smalltalk může být také použit pro práci s daty v →objektových databázích. (→ <i>Gemstone, OQL</i> )
<b>smíšený programovací jazyk, hybridní jazyk</b>	<b>hybrid object-oriented programming language</b>	→Programovací jazyk, který má základ v neobjektovém programování a dovoluje využívat některé vlastnosti objektového přístupu, ale některé, jako např. →závislost nebo →delegování nepodporuje. (→ <i>čistý objektový jazyk</i> ). Mezi smíšené jazyky patří například →C++, →Java a →C#.
<b>SO</b>	<b>SO</b>	Zkratka pro →softwarový objekt.
<b>sociotechnický systém</b>	<b>socio-technical system</b>	→Informační systém včetně jeho podnikového prostředí tvořeného procesy uživatelů, organizační a řídicí strukturou popsány a modelovaný jako jeden související celek. (→ <i>informační inženýrství, konvergenční přístup</i> )
<b>software</b>	<b>software</b>	V kontextu →BPR to je součást detailního popisu →aktivity. Jsou to komponenty nebo moduly zamýšlených nebo již existujících →informačních systémů, které jsou potřeba pro vykonání dané aktivity v →procesu. Je to například „osobní evidence“ nebo „mzdy“ nebo „skladové hospodářství“ apod. (→ <i>architektura systému</i> )
<b>softwarové inženýrství</b>	<b>software engineering</b>	Inženýrský obor, který se zabývá analýzou, modelováním, návrhem a provozem softwarových systémů. Některými autory je považován za součást →informačního inženýrství. Podle →BORMu při tvorbě systému by měly aktivitám softwarového inženýrství předcházet aktivity →business inženýrství.
<b>softwarové modelování</b>	<b>software modeling</b>	Třetí a poslední etapa →BORMu, která zahrnuje tvorbu modelu →softwarových objektů. Jeho prvky a vazby vycházejí z modelu →konceptuálních objektů, ale na rozdíl od něj musejí zohlednit problémy implementačního prostředí. (→ <i>smíšený programovací jazyk, zděděný systém, objektově relační databáze</i> ) Skládá se z fáze →podrobného návrhu a →implementace.
<b>softwarový objekt</b>	<b>software object</b>	→Objekty odvozené z →konceptuálních objektů během →softwarového modelování. Na rozdíl od nich zohledňují omezení implementačního prostředí.
<b>spirální způsob tvorby softwaru, spirální model</b>	<b>spiral development lifecycle</b>	Varianta →iterativního způsobu tvorby softwaru. Fáze projektu se provádějí několikrát za sebou tak, aby při každém opakování došlo ke zpřesnění zadání na základě předchozí implementace. První část jednoho vývojového cyklu se nazývá →expansie a druhá →konsolidace. Tento způsob je vhodný k využití →objektového přístupu. (→ <i>vodopádový model</i> )
<b>SQL</b>	<b>SQL (“sequel”)</b>	Structured Query Language. Programovací jazyk který je standardem pro práci s daty v →relačních databázích. Jeho první verze pochází z počátku 70. let. V současné době probíhá jeho rozšiřování tak, aby mohl pracovat i s objekty. (→ <i>OQL, Smalltalk</i> )
<b>stav</b>	<b>state</b>	Konkrétní konstelace →automatu v čase. Pokud automat přijme nějakou informaci, může to vyvolat přechod z jednoho jeho →stavu do druhého. V →BORMu se nahlíží na objekty v procesech jako na →automaty, které v různých stavech mohou provádět různé →aktivity. (→ <i>role objektu v procesu</i> )
<b>strategická analýza</b>	<b>strategic analysis</b>	První fáze →BORMu, kde dochází k vymezení samotného problému, je stanoveno jeho rozhraní, jsou rozpoznány základní procesy, které se v systému mají odehrávat. Při →spirálním způsobu tvorby softwaru je součástí →expansie systému. (→ <i>získávání zadání, analýza</i> )
<b>synchronní zpráva</b>	<b>synchronous message</b>	→Zpráva, na jejíž výsledek musí →objekt, který zprávu poslal čekat. Objekt tedy pokračuje ve svých aktivitách až když jsou dokončeny všechny aktivity, které byly zprávou vyvolány. (→ <i>asynchronní zpráva</i> )
<b>TO-BE třída</b>	<b>TO-BE class</b>	Druhá fáze →BPR. TO-BE znamená „tak, jak by to mělo být“. V čistých →objektových programovacích jazycích to je zvláštní objekt, který jako svoje data obsahuje vlastnosti (= strukturu dat a metody) pro svoje instance. Ve →smíšených jazycích to je pouze pojem ze syntaxe programovacího jazyka, pomocí kterého se programují vlastnosti objektů. Třídy se v →BORMu odvozují z →business objektů.

<b>událost</b>	<b>event</b>	Podnět z okolí k vykonání → operace nějakého → objektu. V → BORMu jsou zdroje událostí modelovány také jako objekty a jejich události jako → zprávy nebo → komunikace. (→ <i>závislost</i> )
<b>úkol</b>	<b>target</b>	V kontextu → BPR to je součást detailního popisu → aktivity. Úkoly popisují, čeho je třeba dosáhnout. Je to například „snížit surovinovou náročnost“ nebo „zkrátit průběh vyřízení objednávky“. Jejich znalost je důležitá pro rozhodování nad souvislostmi zadání informačního systému, → podnikových procesů s dalšími atributy organizace. (→ <i>získávání zadání</i> )
<b>UML – unifikovaný modelovací jazyk</b>	<b>UML – Unified Modeling Language</b>	UML poprvé publikovali G. Booch, J. Rumbaugh a I. Jacobson v roce 1996. Je doporučovaným standardem pro notaci (způsob kreslení) objektových → diagramů a představuje sjednocení myšlenek původních metod svých autorů. Stále se ještě vyvíjí. Pro potřeby → BORMu je třeba UML doplnit. (→ <i>ORD</i> )
<b>úvodní analýza</b>	<b>initial analysis</b>	Fáze rozpracování samotného problému, jsou mapovány požadované → procesy v systému a vlastnosti základních → objektů, které se na diskutovaných procesech podílejí. Může při ní dojít k → BPR. Při → spirálním způsobu tvorby softwaru je součástí fáze → <i>expanse</i> . (→ <i>analýza</i> )
<b>úvodní návrh, úvodní design</b>	<b>initial design</b>	Je to fáze → BORMu, ve které se → konceptuální model upravuje tak, aby byl schopen softwarové implementace. Z pohledu zadání by mělo již vše být hotovo a rozpoznáno. Úvodní návrh používá shodné nebo velmi podobné nástroje jako předchozí fáze → <i>podrobné analýzy</i> , ale liší se způsobem práce s nimi. Při → spirálním způsobu tvorby softwaru je součástí fáze → <i>konsolidace</i> . (→ <i>návrh</i> )
<b>včasná vazba</b>	<b>early binding</b>	Druh vazby mezi → zprávou a → metodou, kdy již v době překladu kódu programu je ke zprávě překladačem jednoznačně přiřazena metoda, která se má provést, což může vést k omezení míry → <i>polymorfismu</i> mezi objekty. Tento typ vazby je na rozdíl od → <i>pozdní</i> vazby typický pro → <i>smíšené</i> objektové programovací jazyky.
<b>vodopádový způsob tvorby softwaru, vodopádový model</b>	<b>waterfall development lifecycle, sequential development lifecycle</b>	Způsob vývoje softwaru, kdy se ke každé další fázi projektu přistupuje až po úplném dokončení předchozí fáze. Vracení se zpět není přípustné. Tento styl se dobře plánuje a řídí, ale není vhodný v případech, kde na začátku projektu není ještě přesně rozpoznané zadání. Jiné způsoby vývoje jsou → <i>spirální</i> a → <i>iterativní</i> .
<b>vrstva výrobek, deliverable</b>	<b>layer deliverable</b>	Jedna součást → <i>architektury</i> modelovaného systému. Programový výstup z jednoho cyklu → <i>spirálního</i> vývoje softwaru. Označuje se takto nejen → <i>prototyp</i> , ale i produkt „posledního“ cyklu, protože i ten může posloužit jako nové zadání pro další vývojový cyklus. Může také sloužit pro tvorbu nové verze produktu - a to nejen skrze zkušenosti s ním, ale přímo i svým kódem jako výchozí model nové → <i>expanse</i> .
<b>zapouzdření</b>	<b>encapsulation</b>	Spolu s → <i>polymorfismem</i> a → <i>identitou</i> základní vlastnost → objektů. Díky zapouzdření může objekt obsahovat data nebo → <i>metody</i> , se kterými pracuje jen objekt sám, nejsou součástí jeho → <i>protokolu</i> , a neposkytuje je ostatním objektům v systému. Z vnějšího pohledu se potom objekt tváří, jako by tyto vlastnosti neměl.
<b>zařízení</b>	<b>device</b>	Součást detailního popisu → aktivity v → BPR. Zařízení je konkrétní pracovní pomůcka nebo stroj (například „služební auto“ nebo „osobní počítač“ nebo „ochranný oděv“), který je potřeba k vykonání dané aktivity.
<b>závislost</b>	<b>dependency</b>	Závislost vyjadřuje, že provedení nějaké → <i>metody</i> řídicího objektu (→ <i>klienta</i> ) nepřímou bez poslání → <i>zprávy</i> vyvolává provedení nějaké <i>metody</i> řízeného objektu (→ <i>serveru</i> ). V případě, že implementační prostředí tuto vazbu nepodporuje, musí se ve fázi → <i>softwarového</i> modelování nahrazovat. (→ <i>smíšený programovací jazyk</i> )
<b>zděděný systém</b>	<b>legacy system</b>	Existující programy, data nebo organizační struktura či procesy, které nelze měnit nebo ignorovat, což komplikuje tvorbu nového systému ve fázi → <i>softwarového</i> modelování, protože se jim musí struktura nového systému přizpůsobit.

<b>získávání zadání, requirement engineering</b>	<b>requirement engineering</b>	Inženýrský obor, který se zabývá rozpoznáváním, analýzou a verifikací zadání pro →informační systémy. (→ <i>informační inženýrství, business inženýrství, softwarové inženýrství</i> )
<b>znovu- použitelnost</b>	<b>reusability</b>	Schopnost →objektu sloužit v jiném systému jiným způsobem, než pro jaký byl objekt vytvořen. Objekty jsou znovupoužitelné díky →polymorfismu, protože jsou snáze mezi sebou zaměnitelné. (→ <i>protokol</i> ) Znovupoužitelnost je základem techniky →návrhových vzorů.
<b>zpráva</b>	<b>message</b>	Žádost o provedení →metody →objektem, kterému je zpráva poslána. Na rozdíl od volání funkce v klasickém programování je v objektovém programování od sebe oddělená žádost o operaci a její vlastní provedení. (→ <i>polymorfismus, včasná vazba, pozdní vazba</i> ) Zprávy mohou přenášet data. Data ve směru poslání zprávy se nazývají →parametry zprávy, data posílaná opačným směrem jsou →návratové hodnoty. Zprávy jsou →synchronní nebo →asynchronní.

## 9 Použitá a doporučená literatura

1. Abadi M., Cardelli L.: *A Theory of Objects*, Springer 1996, ISBN 0-387-94775-2
2. Ambler S.: *More Process Patterns – Delivering Large-Scale Systems Using Object Technology*, SIGS Books 2000, ISBN 0-521-65262-6
3. Ambler S.: *Process Patterns – Building Large-Scale Systems Using Object Technology*, SIGS Books 2000, ISBN 0-521-64568-9
4. Ambler, Scott W: *Be Realistic about the UML*, <http://www.agilemodeling/essays/references.htm>
5. Arlow, J.; Neustadt, I.: *UML a unifikovaný proces vývoje aplikací*. Computer Press. Brno 2003. ISBN 80-7226-947-X
6. Barry D.: *The Object Database Handbook: How to Select, Implement, and Use Object-Oriented Databases*, ISBN 0471147184
7. Beck K.: *Agile Database Techniques - Effective Strategies for the Agile Software Developer*, John Wiley & Sons; ISBN 0471202835
8. Beck K.: *Smalltalk Best Practice Patterns*, Prentice Hall 1997, ISBN 0-13-476904-X
9. Bertino, E., Martino, L., *Object-Oriented Database Systems, Concepts and Architectures*, Addison-Wesley, 1993. ISBN 0-201-62439-7.
10. Blaha M., Premerlani W.: *Object Oriented Modeling and Design for Database Applications*, Prentice Hall 1998, ISBN 0-13-123829-9
11. Boehm, B. W.: *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, 1981
12. Booch Grady, Rumbaugh James and Jacobson Ivar, *The Unified Modeling Language User Guide*, 1998, Addison-Wesley, ISBN: 0-201-57168-4
13. Cantor Murray, *Object-Oriented Project Management with UML*, 1998, J. Wiley and Sons, ISBN: 0-471-25303-0
14. Carda A., Merunka V., Polák J.: *Umění systémového návrhu - objektivě orientovaná tvorba informačních systémů pomocí původní metody BORM*. Grada, Praha 2003. ISBN 80-247-0424-2.
15. Catell R.G.G. et al.: *The Object Data Standard: ODMG 3.0*, ISBN 1558606475
16. Catell R.G.G. et al.: *The Object Database Standard – ODMG2.0*, Morgan Kaufman 1997, ISBN 1-55860-463-4
17. Cook, Steve; Daniels, John: *Object-Oriented Methods and the Great Object Myth*, Object in Europe, SIGS Publications, Vol 1. Nr. 4, 1994
18. Coterrell Mike, Hughes Bob: *Software Project Management*, 1995, Thomson Computer Press ISBN 1-850-32190-6
19. Čížek, F.: *Filosofie, metodologie, věda*, Svoboda, Praha, 1969
20. Darnton, G., Darnton, M. *Business Process Analysis*, International Thomson Publishing 1997
21. Date C.J., *An Introduction to Database Systems (6th Edition)*, 1995, Addison-Wesley, ISBN: 0-201-82458-2
22. Davis Alan: *Software Requirements - Objects, Functions and States*, 1993, Prentice Hall ISBN 0-13-562174-7
23. Davis M.D., Weyuker E.J.: *Computability, Complexity and Languages – Fundamentals of Theoretical Computer Science*. Academic Press NY, ISBN 0-12-206380-5
24. Delobel C., Lecluse C., Richard P.: *Databases: from Relational to Object-Oriented Systems*, International Thomson Computer Press, 1995. ISBN 1850321248
25. *Domain Specific Methodology – 10 Times Faster Than UML*. Metacase Ltd., Finland, <http://www.metacase.com>

26. Eeles P., Sims O., *Building Business Objects*, John Wiley & Sons, Inc., New York, 1998.
27. Embley D.: *Object Database Development: Concepts and Principles*, ISBN 0201258293
28. *Extreme programming approach*, <http://www.xprogramming.org> and also another page at <http://www.extremeprogramming.org>
29. Fenton N., Pfleger S.L.: *Software Metrics – A Rigorous & Practical Approach*, PWS Publishing 1997, ISBN 0-534-95425-1
30. Fowler M., Kendall S., *UML Distilled (2nd Edition)*, 1999, Addison-Wesley ISBN: 0-201-65783-X
31. Gamma, E.; Helm, R.; Johnson, R., Vlissides, J: *Návrh programů pomocí vzorů – Stavební kameny objektově orientovaných programů*, Grada Publishing sro, Praha 2003
32. Gemstone Object Server - documentation & non-commercial version download, <http://www.gemstone.com>, <http://smalltalk.gemstone.com>.
33. Goldberg A., Rubin K. S.: *Succeeding with Objects - Decision Frameworks for Project Management*, Addison Wesley, Reading Mass, 1995.
34. Graham, Ian; Simons, Anthony J H: *30 Things that Go Wrong in Object Modeling with UML 1.3*, University of Sheffield & IGA Ltd. <http://www.iga.co.uk>
35. Hammer M., Stanton S.A.: *The Reengineering Revolution*, Harper Collins 1994, ISBN 0-88730-736-1
36. Hruška T.: *Objektově orientované databázové technologie*, sborník konference Tvorba softwaru 1995, ISBN 80-901751-3-9
37. Humby, E.: *Programy na základe rozhodovacích tabuliek*, Alfa, Bratislava, 1976
38. Jacobson Ivar: *Object-Oriented Software Engineering - A Use Case Driven Approach*, 1992, Addison Wesley ISBN 0-201-54435-0
39. Knott R P, Polák J, Merunka V.: *BORM - Business Object Relation Modelling Methodology*, final report of the project VAPPIENS, British Council - Know-How Fund, Dept. of Computer Studies, Loughborough University.
40. Knott R P, Polák J, Merunka V.: *Principles of the Object-Oriented Paradigm - Basic Concepts, OO design methodologies, OOA & OOD*, Tutorial Book printed at EastEurOOP93 International Conference, Bratislava November 1993.
41. Knott, Roger P.; Merunka, Vojtěch; Polák, Jiří: *Process Modeling for Object Oriented Analysis using BORM Object Behavioral Analysis*, in Proceedings of Fourth International Conference on Requirements Engineering, IEEE&ACM, Chicago 2000.
42. Kotonya G., and Sommerville I., *Requirements Engineering: Processes and Techniques*, 1999, J. Wiley and Sons,
43. Kroha P.: *Objects and Databases*, McGraw Hill, London 1995, ISBN 0-07-707790-3.
44. Kunz, Werner, Rittel, Horst: *Issues as Elements of Information Systems*, Working Paper 131, The Institute of Urban and Regional Development, University of California, Berkeley, California, 1970
45. Lacko B.: *Komparativní analýza strukturovaného a objektově orientovaného přístupu*, sborník konference OBJEKTY 1996. ČZU Praha 1996 str.69-76
46. Lacko B.: *Vademekum objektově orientované technologie*, sborník konference Programování, Ostrava 1993.
47. Loomis M., Chaundri A.: *Object Databases in Practice*, ISBN 013899725X
48. Mellor Stephen, Shlaer Sally: *Object Lifecycles: Modeling the World in States*, Yourdon Press, 1992
49. Merunka V, Polák J.: *Object-Oriented Analysis and Design - Teaching and Application Issues*, in proceedings of ASU- Association of Simula Users & ACM 1993, pp 22-36, Praha - Průhonice, september 1994

50. Merunka V., *Objekty v databázových systémech*, skriptum PEF ČZU v Praze, 2002.
51. Merunka V., Polák J, Knott R.P. Buldra M.: *Object-Oriented Analysis and Design of IS of Agrarian Chamber Czech Republic in BORM*, in proceedings of 4Front Coordinators conference, Deloitte&Touche, Singapore january 1995.
52. Merunka V.: *Objektový databázový systém Gemstone*, sborník konference OBJEKTY 2003. Ostrava 2003. ISBN 80-248-0274-0
53. Merunka V.: *Objekty v databázových systémech*, skriptum ČZU Praha, Praha 2002. ISBN 80-213-0318-2
54. Merunka, Vojtěch; Polák, Jiří; Rivas, Luis: *BORM – Business Object Relation Modeling*, in Proceedings of WOON - Fifth International Conference on Object-Oriented Programming, St. Petersburg 2001.
55. MetaEdit *CASE Tool (program and documentation)*, Metacase Ltd., Finland, <http://www.metacase.com>
56. Michaelson G.: *An Introduction to Functional Programming through Lambda Calculus*, John Wiley & Sons, 1998.
57. Montlick T.: *The Distributed Smalltalk Survival Guide*, Cambridge University Press 1999, ISBN 0-521-64552-2
58. Page-Jones, M.: *Základy objektově orientovaného návrhu v UML*. Grada Publishing. Praha 2001. ISBN 80-247-0210-X
59. Rittel H., Webber M.: *Dilemmas in a General Theory of Planning*, Policy Sciences 4, Elsevier Scientific Publishing, Amsterdam, pp. 155-159, 1973.
60. Rittel H.: *On the Planning Crisis: Systems Analysis of the First and Second Generations*, Bedriftsoekonomen, Nr. 8, též Reprint No. 107, The Institute of Urban and Regional Development, University of California, Berkeley, California, 1972.
61. Rittel, Horst: *Reflections on the Scientific and Political Significance of Decision Theory*, Working Paper 115, The Institute of Urban and Regional Development, University of California, Berkeley, California, 1969
62. Royce, Walker, *Software Project Management: A Unified Framework*, 1998, Addison Wesley, ISBN:- 0-201-30958-0
63. Rumbaugh James, Blaha Michael, Premerlani William, Eddy Frederic, Lorenzen William: *Object-Oriented Modelling and Design*, 1991, Prentice Hall, ISBN 0-13-630054-5
64. Rumbaugh James, Jacobson Ivar, and Booch Grady, *The Unified Modeling Language Reference Manual*, 1999, Addison-Wesley, ISBN: 0-201-30998-X
65. Ryan T.W.: *Distributed Object Technology – Concepts and Applications*, Hewlett Packard Professional Books 1996, ISBN 0-13-348996-5
66. Satzinger J. W. and Orvik T. U.: *The Object-Oriented Approach - Concepts, Modeling and System Development*, Boyd&Fraser 1996.
67. Sborníky celostátní odborné konference OBJEKTY, <http://objekty.pef.czu.cz>
68. Shlaer S., Mellor S.J.: *Object Lifecycles – Modeling the World in States*, Yourdon Press 1992, ISBN 0-13-629940-7
69. Schmuller, J.: *Myslíme v jazyku UML*. Grada Publishing. Praha 2001. ISBN 80-247-0029-8
70. *Smalltalk programming language*, <http://www.smalltalk.org>, <http://www.cincom.com>
71. Taylor D.: *Business Engineering with Object Technology*, John Wiley 1995.
72. Thomas D.: *UML – The Universal Modeling and Programming Language?*, September 2001, from the library at <http://www.ltt.de>
73. *UML documents*, <http://www.omg.org> or <http://www.rational.com/uml>